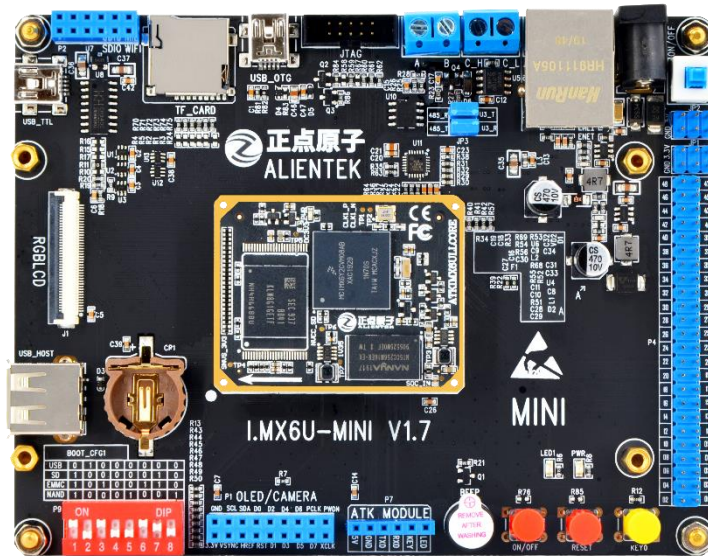
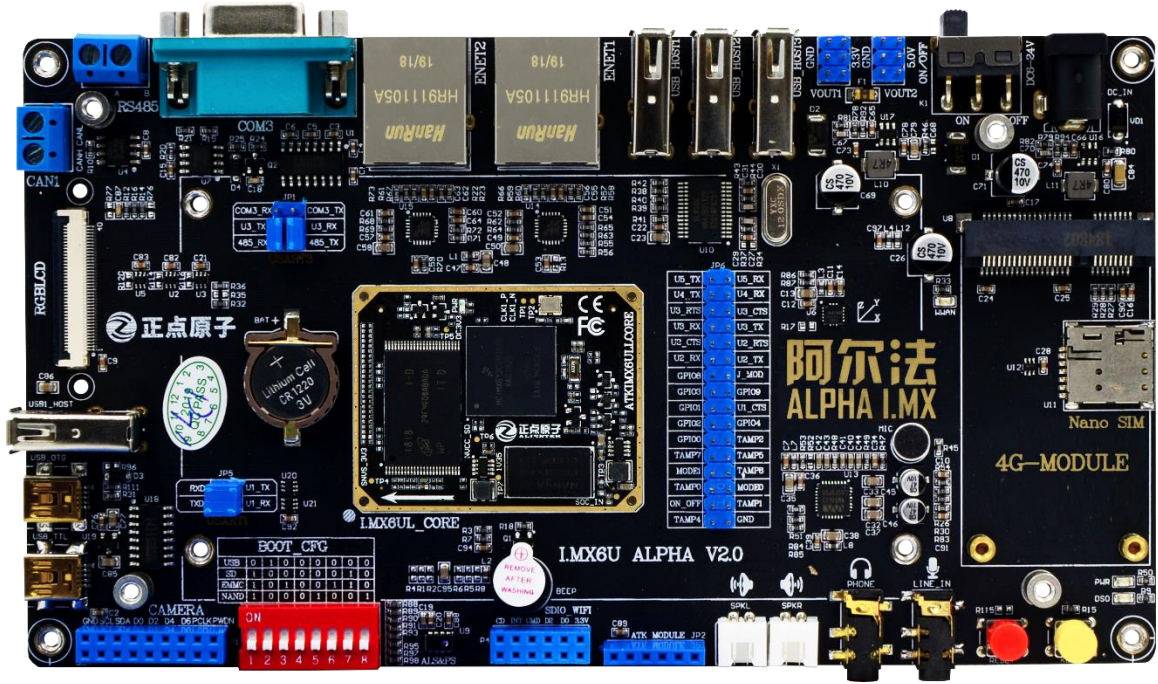


I.MX6U 嵌入式 Linux C 应用编程指南 V1.4

-正点原子 I.MX6U ALPHA/Mini 开发板教程





正点原子公司名称 : 广州市星翼电子科技有限公司

原子哥在线教学平台 : www.yuanzige.com

开源电子网 / 论坛 : <http://www.openedv.com/forum.php>

正点原子淘宝店铺 : <https://openedv.taobao.com>

正点原子官方网站 : www.alientek.com

正点原子 B 站视频 : <https://space.bilibili.com/394620890>

电话: 020-38271790 传真: 020-36773971

请关注正点原子公众号, 资料发布更新我们会通知。

请下载原子哥 APP, 数千讲视频免费学习, 更快更流畅。



扫码关注正点原子公众号



扫码下载“原子哥”APP

文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初始版本	邓涛	邓涛	2020.06.10
V1.1	<p>修正: 文档部分描述有误的地方进行了修正! 页眉处论坛地址修改!</p> <p>重构: 第十六章 GPIO 应用编程内容重构 第十七章输入设备应用编程内容重构 第十九章 FrameBuffer 应用编程内容重构</p> <p>添加: 第十五章点亮 LED 添加 15.1.4 小节 第十八章使用 tslib 库 第二十章在 LCD 上显示 jpeg 图像 第二十一章在 LCD 上显示 png 图片 第二十二章 LCD 横屏切换为竖屏 第二十三章在 LCD 上显示字符 第二十四章 PWM 应用编程 第二十五章 V4L2 摄像头应用编程 第二十六章串口应用编程 第二十七章看门狗应用编程</p>	邓涛	邓涛	2020.07.20
V1.2	<p>修正: 文档部分描述有误的地方进行了修正!</p> <p>添加: 第二十八章音频应用编程 第二十九章网络基础知识 第三十章 socket 编程基础</p>	邓涛	邓涛	2021.09.16
V1.3	<p>修正: 文档部分描述有误的地方进行了修正!</p> <p>添加: 第三十一章 CAN 应用编程基础 第三十二章 CMake 入门与进阶 第三十三章实战小项目之 MQTT 物联网</p>	邓涛	邓涛	2021.11.22
V1.4	<p>添加: 第三十四章实战小项目之视频监控</p>	邓涛	邓涛	2021.12.08

目录

前言	19
第一篇 入门篇	21
第一章 应用编程概念	22
1.1 系统调用	23
1.2 库函数	27
1.3 标准 C 语言函数库	28
1.4 main 函数	30
1.5 本书使用的开发环境	30
1.5.1 Ubuntu 系统下安装 Eclipse	31
第二章 文件 I/O 基础	43
2.1 一个简单的文件 IO 示例	44
2.2 文件描述符	45
2.3 open 打开文件	46
2.4 write 写文件	50
2.5 read 读文件	50
2.6 close 关闭文件	51
2.7 lseek	51
2.8 练习	52
第三章 深入探究文件 I/O	59
3.1 Linux 系统如何管理文件	60
3.1.1 静态文件与 inode	60
3.1.2 文件打开时的状态	62
3.2 返回错误处理与 errno	63
3.2.1 strerror 函数	64
3.2.2 perror 函数	65
3.3 exit、_exit、_Exit	66
3.3.1 _exit()和_Exit()函数	67
3.3.2 exit()函数	67
3.4 空洞文件	68
3.4.1 概念	68
3.4.2 实验测试	68
3.5 O_APPEND 和 O_TRUNC 标志	70
3.5.1 O_TRUNC 标志	70
3.5.2 O_APPEND 标志	71
3.6 多次打开同一个文件	74
3.6.1 验证一些现象	74
3.6.2 多次打开同一文件进行读操作与 O_APPEND 标志	78
3.7 复制文件描述符	83
3.7.1 dup 函数	84

3.7.2 dup2 函数	87
3.8 文件共享	88
3.9 原子操作与竞争冒险	91
3.9.1 竞争冒险简介	91
3.9.2 原子操作	92
3.10 fcntl 和 ioctl	95
3.10.1 fcntl 函数	95
3.10.2 ioctl 函数	99
3.11 截断文件	99
第四章 标准 I/O 库	102
4.1 标准 I/O 库简介	103
4.2 FILE 指针	103
4.3 标准输入、标准输出和标准错误	103
4.4 打开文件 fopen()	104
4.5 读文件和写文件	105
4.6 fseek 定位	109
4.7 检查或复位状态	112
4.7.1 feof() 函数	112
4.7.2 ferror() 函数	113
4.7.3 clearerr() 函数	113
4.8 格式化 I/O	114
4.8.1 格式化输出	114
4.8.2 格式化输入	120
4.8.3 小结	125
4.9 I/O 缓冲	125
4.9.1 文件 I/O 的内核缓冲	125
4.9.2 刷新文件 I/O 的内核缓冲区	126
4.9.3 直接 I/O: 绕过内核缓冲	128
4.9.4 stdio 缓冲	133
4.9.5 I/O 缓冲小节	138
4.10 文件描述符与 FILE 指针互转	139
第五章 文件属性与目录	141
5.1 Linux 系统中的文件类型	142
5.1.1 普通文件	142
5.1.2 目录文件	143
5.1.3 字符设备文件和块设备文件	143
5.1.4 符号链接文件	144
5.1.5 管道文件	144
5.1.6 套接字文件	144
5.1.7 总结	144
5.2 stat 函数	145
5.2.1 struct stat 结构体	145

5.2.2 st_mode 变量.....	146
5.2.3 struct timespec 结构体.....	148
5.2.4 练习.....	148
5.3 fstat 和 lstat 函数.....	152
5.3.1 fstat 函数.....	152
5.3.2 lstat 函数.....	153
5.4 文件属主.....	154
5.4.1 有效用户 ID 和有效组 ID.....	155
5.4.2 chown 函数.....	155
5.4.3 fchown 和 lchown 函数.....	158
5.5 文件访问权限.....	158
5.5.1 普通权限和特殊权限.....	158
5.5.2 目录权限.....	160
5.5.3 检查文件权限 access.....	161
5.5.4 修改文件权限 chmod.....	163
5.5.5 umask 函数.....	164
5.6 文件的时间属性.....	166
5.6.1 utime()、utimes()修改时间属性.....	167
5.6.2 futimens()、utimensat()修改时间属性.....	171
5.7 符号链接(软链接)与硬链接.....	175
5.7.1 创建链接文件.....	176
5.7.2 读取软链接文件.....	179
5.8 目录.....	180
5.8.1 目录存储形式.....	180
5.8.2 创建和删除目录.....	180
5.8.3 打开、读取以及关闭目录.....	182
5.8.4 进程的当前工作目录.....	186
5.9 删除文件.....	189
5.10 文件重命名.....	191
5.11 总结.....	192
第六章 字符串处理.....	193
6.1 字符串输入/输出.....	194
6.1.1 字符串输出.....	194
6.1.2 字符串输入.....	199
6.1.3 总结.....	208
6.2 字符串长度.....	208
6.3 字符串拼接.....	210
6.4 字符串拷贝.....	211
6.5 内存填充.....	213
6.6 字符串比较.....	215
6.7 字符串查找.....	217
6.8 字符串与数字互转.....	220
6.8.1 字符串转整形数据.....	220

6.8.2 字符串转浮点型数据	223
6.8.3 数字转字符串	225
6.9 给应用程序传参	226
6.10 正则表达式	227
6.10.1 初识正则表达式	227
6.11 C 语言中使用正则表达式	228
第七章 系统信息与系统资源	231
7.1 系统信息	232
7.1.1 系统标识 <code>uname</code>	232
7.1.2 <code>sysinfo</code> 函数	233
7.1.3 <code>gethostname</code> 函数	235
7.1.4 <code>sysconf()</code> 函数	236
7.2 时间、日期	237
7.2.1 时间的概念	237
7.2.2 Linux 系统中的时间	239
7.2.3 获取时间 <code>time/gettimeofday</code>	240
7.2.4 时间转换函数	242
7.2.5 设置时间 <code>settimeofday</code>	251
7.2.6 总结	251
7.3 进程时间	252
7.3.1 <code>times</code> 函数	252
7.3.2 <code>clock</code> 函数	255
7.4 产生随机数	256
7.5 休眠	258
7.5.1 秒级休眠: <code>sleep</code>	258
7.5.2 微秒级休眠: <code>usleep</code>	259
7.5.3 高精度休眠: <code>nanosleep</code>	260
7.6 申请堆内存	261
7.6.1 在堆上分配内存: <code>malloc</code> 和 <code>free</code>	261
7.6.2 在堆上分配内存的其它方法	263
7.6.3 分配对其内存	264
7.7 <code>proc</code> 文件系统	268
7.7.1 <code>proc</code> 文件系统的使用	269
第八章 信号: 基础	272
8.1 基本概念	273
8.2 信号的分类	275
8.2.1 可靠信号与不可靠信号	275
8.2.2 实时信号与非实时信号	276
8.3 常见信号与默认行为	276
8.4 进程对信号的处理	278
8.4.1 <code>signal()</code> 函数	279
8.4.2 <code>sigaction()</code> 函数	282

8.5 向进程发送信号	285
8.5.1 kill()函数	285
8.5.2 raise()	287
8.6 alarm()和 pause()函数	288
8.6.1 alarm()函数	289
8.6.2 pause()函数	290
8.7 信号集	292
8.7.1 初始化信号集	292
8.7.2 向信号集中添加/删除信号	292
8.7.3 测试信号是否在信号集中	293
8.8 获取信号的描述信息	293
8.8.1 strsignal()函数	294
8.8.2 psignal()函数	295
8.9 信号掩码(阻塞信号传递)	296
8.10 阻塞等待信号 sigsuspend()	299
8.11 实时信号	301
8.11.1 sigpending()函数	301
8.11.2 发送实时信号	302
8.12 异常退出 abort()函数	304
第九章 进程	307
9.1 进程与程序	308
9.1.1 main()函数由谁调用?	308
9.1.2 程序如何结束?	308
9.1.3 何为进程?	309
9.1.4 进程号	309
9.2 进程的环境变量	312
9.2.1 应用程序中获取环境变量	313
9.2.2 添加/删除/修改环境变量	315
9.2.3 清空环境变量	317
9.2.4 环境变量的作用	318
9.3 进程的内存布局	318
9.4 进程的虚拟地址空间	319
9.5 fork()创建子进程	320
9.6 父、子进程间的文件共享	323
9.7 系统调用 vfork()	327
9.8 fork()之后的竞争条件	329
9.9 进程的诞生与终止	332
9.9.1 进程的诞生	332
9.9.2 进程的终止	333
9.10 监视子进程	335
9.10.1 wait()函数	335
9.10.2 waitpid()函数	338
9.10.3 waitid()函数	342

9.10.4 僵尸进程与孤儿进程	342
9.10.5 SIGCHLD 信号	346
9.11 执行新程序	348
9.11.1 execve()函数.....	348
9.11.2 exec 库函数.....	350
9.11.3 exec 族函数使用示例	352
9.11.4 system()函数.....	354
9.12 进程状态与进程关系	356
9.12.1 进程状态	356
9.12.2 进程关系	357
9.13 守护进程	361
9.13.1 何为守护进程	361
9.13.2 编写守护进程程序	362
9.13.3 SIGHUP 信号	365
9.14 单例模式运行	367
9.14.1 通过文件存在与否进行判断	367
9.14.2 使用文件锁	369
第十章 进程间通信简介	372
10.1 进程间通信简介	373
10.2 进程间通信的机制有哪些?	373
10.3 管道和 FIFO.....	373
10.4 信号	374
10.5 消息队列	374
10.6 信号量	374
10.7 共享内存	374
10.8 套接字 (Socket)	374
第十一章 线程	376
11.1 线程概述	377
11.1.1 线程概念	377
11.1.2 并发和并行	378
11.2 线程 ID	380
11.3 创建线程	381
11.4 终止线程	383
11.5 回收线程	385
11.6 取消线程	387
11.6.1 取消一个线程	387
11.6.2 取消状态以及类型	389
11.6.3 取消点	391
11.6.4 线程可取消性的检测	393
11.7 分离线程	396
11.8 注册线程清理处理函数	398
11.9 线程属性	403

11.9.1 线程栈属性	403
11.9.2 分离状态属性	405
11.10 线程安全	406
11.10.1 线程栈	406
11.10.2 可重入函数	407
11.10.3 线程安全函数	413
11.10.4 一次性初始化	415
11.10.5 线程特有数据	417
11.10.6 线程局部存储	424
11.11 更多细节问题	425
11.11.1 线程与信号	426
第十二章 线程同步	431
12.1 为什么需要线程同步?	432
12.2 互斥锁	435
12.2.1 互斥锁初始化	436
12.2.2 互斥锁加锁和解锁	436
12.2.3 pthread_mutex_trylock()函数	439
12.2.4 销毁互斥锁	441
12.2.5 互斥锁死锁	443
12.2.6 互斥锁的属性	444
12.3 条件变量	445
12.3.1 条件变量初始化	447
12.3.2 通知和等待条件变量	448
12.3.3 条件变量的判断条件	450
12.3.4 条件变量的属性	450
12.4 自旋锁	451
12.4.1 自旋锁初始化	451
12.4.2 自旋锁加锁和解锁	452
12.5 读写锁	454
12.5.1 读写锁初始化	455
12.5.2 读写锁上锁和解锁	455
12.5.3 读写锁的属性	458
12.6 总结	459
12.7 练习	459
第十三章 高级 I/O	460
13.1 非阻塞 I/O	461
13.1.1 阻塞 I/O 与非阻塞 I/O 读文件	461
13.1.2 阻塞 I/O 的优点与缺点	465
13.1.3 使用非阻塞 I/O 实现并发读取	466
13.2 I/O 多路复用	469
13.2.1 何为 I/O 多路复用	469
13.2.2 select()函数介绍	469

13.2.3 poll()函数介绍	473
13.2.4 总结	477
13.3 异步 IO	477
13.4 优化异步 I/O	480
13.4.1 使用实时信号替换默认信号 SIGIO	480
13.4.2 使用 sigaction()函数注册信号处理函数	481
13.4.3 使用示例	481
13.5 存储映射 I/O	484
13.5.1 mmap()和 munmap()函数	484
13.5.2 mprotect()函数	489
13.5.3 msync()函数	489
13.5.4 普通 I/O 与存储映射 I/O 比较	490
13.6 文件锁	492
13.6.1 flock()函数加锁	492
13.6.2 fcntl()函数加锁	497
13.6.3 lockf()函数加锁	509
13.7 小结	509
第十四章 本篇总结	510
第二篇 提高篇	511
第十五章 点亮 LED	512
15.1 应用层操控硬件的两种方式	513
15.1.1 sysfs 文件系统	513
15.1.2 sysfs 与/sys	513
15.1.3 总结	514
15.1.4 标准接口与非标准接口	515
15.2 LED 硬件控制方式	515
15.3 编写 LED 应用程序	517
15.4 在开发板上测试	519
第十六章 GPIO 应用编程	521
16.1 应用层如何操控 GPIO	522
16.2 GPIO 应用编程之输出	524
16.3 GPIO 应用编程之输入	527
16.4 GPIO 应用编程之中断	530
16.5 在开发板上测试	533
16.5.1 GPIO 输出测试	533
16.5.2 GPIO 输入测试	534
16.5.3 GPIO 中断测试	534
第十七章 输入设备应用编程	536
17.1 输入类设备编程介绍	537
17.1.1 什么是输入设备	537
17.1.2 input 子系统	537

17.1.3 读取数据的流程	537
17.1.4 应用程序如何解析数据	537
17.2 读取 struct input_event 数据	540
17.3 在开发板上验证	541
17.4 按键应用编程	543
17.5 触摸屏应用编程	546
17.5.1 解析触摸屏设备上报的数据	546
17.5.2 获取触摸屏的信息	552
17.5.3 单点触摸应用程序	555
17.5.4 多点触摸应用程序	559
17.6 鼠标应用编程	563
第十八章 使用 tslib 库	564
18.1 tslib 简介	565
18.2 tslib 移植	565
18.2.1 下载 tslib 源码	565
18.2.2 编译 tslib 源码	568
18.2.3 tslib 安装目录下的文件夹介绍	570
18.2.4 在开发板上测试 tslib	573
18.3 tslib 库函数介绍	574
18.3.1 打开触摸屏设备	575
18.3.2 配置触摸屏设备	575
18.3.3 读取触摸屏数据	575
18.4 基于 tslib 编写触摸屏应用程序	577
18.4.1 单点触摸应用程序	577
18.4.2 多点触摸应用程序	579
第十九章 FrameBuffer 应用编程	582
19.1 什么是 FrameBuffer	583
19.2 LCD 的基础知识	583
19.3 LCD 应用编程介绍	583
19.3.1 使用 ioctl() 获取屏幕参数信息	583
19.3.2 使用 mmap() 将显示缓冲区映射到用户空间	588
19.4 LCD 应用编程练习之 LCD 基本操作	588
19.5 LCD 应用编程练习之显示 BMP 图片	594
19.5.1 BMP 图像介绍	594
19.5.2 在 LCD 上显示 BMP 图像	602
19.5.3 在开发板上测试	607
第二十章 在 LCD 上显示 jpeg 图像	610
20.1 JPEG 简介	611
20.2 libjpeg 简介	611
20.3 libjpeg 移植	611
20.3.1 下载源码包	611
20.3.2 编译源码	612

20.3.3 安装目录下的文件夹介绍	615
20.3.4 移植到开发板	616
20.4 libjpeg 使用说明	617
20.4.1 错误处理	618
20.4.2 创建解码对象	619
20.4.3 设置数据源	620
20.4.4 读取 jpeg 文件的头信息	620
20.4.5 设置解码处理参数	621
20.4.6 开始解码	621
20.4.7 读取数据	621
20.4.8 结束解码	622
20.4.9 释放/销毁解码对象	622
20.5 libjpeg 应用编程	622
20.6 总结	627
第二十一章 在 LCD 上显示 png 图片	628
21.1 PNG 简介	629
21.2 libpng 简介	629
21.3 zlib 移植	629
21.3.1 下载源码包	629
21.3.2 编译源码	631
21.3.3 安装目录下的文件夹介绍	632
21.3.4 移植到开发板	633
21.4 libpng 移植	633
21.4.1 下载源码包	633
21.4.2 编译源码	634
21.4.3 安装目录下的文件夹介绍	637
21.4.4 移植到开发板	637
21.5 libpng 使用说明	638
21.5.1 libpng 的数据结构	638
21.5.2 创建和初始化 png_struct 对象	638
21.5.3 创建和初始化 png_info 对象	639
21.5.4 设置错误返回点	639
21.5.5 指定数据源	641
21.5.6 读取 png 图像数据并解码	642
21.5.7 读取解码后的数据	645
21.5.8 结束销毁对象	645
21.6 libpng 应用编程	645
第二十二章 LCD 横屏切换为竖屏	652
22.1 横屏显示如何切换为竖屏显示	653
22.2 编写示例代码	655
第二十三章 在 LCD 上显示字符	662
23.1 原始方式: 取模显示字符	663

23.2 freetype 简介	677
23.3 freetype 移植	677
23.3.1 下载 FreeType 源码	677
23.3.2 交叉编译 FreeType 源码	678
23.3.3 安装目录下的文件	681
23.3.4 移植到开发板	682
23.4 freetype 库的使用	682
23.4.1 初始化 FreeType 库	685
23.4.2 加载 face 对象	686
23.4.3 设置字体大小	686
23.4.4 加载字形图像	687
23.5 示例代码	687
第二十四章 PWM 应用编程	695
24.1 应用层如何操控 PWM	696
24.2 编写应用程序	698
24.3 在开发板上测试	701
第二十五章 V4L2 摄像头应用编程	705
25.1 V4L2 简介	706
25.2 V4L2 摄像头应用程序	706
25.2.1 打开摄像头	710
25.2.2 查询设备的属性/能力/功能	710
25.2.3 设置帧格式、帧率	712
25.2.4 申请帧缓冲、内存映射	721
25.2.5 入队	724
25.2.6 开启视频采集	725
25.2.7 读取数据、对数据进行处理	725
25.2.8 结束视频采集	726
25.3 V4L2 摄像头应用编程实战	726
第二十六章 串口应用编程	738
26.1 串口应用编程介绍	739
26.1.1 终端 Terminal	739
26.1.2 串口应用编程	740
26.1.3 struct termios 结构体	741
26.1.4 终端的三种工作模式	745
26.1.5 打开串口设备	746
26.1.6 获取终端当前的配置参数: tcgetattr() 函数	746
26.1.7 对串口终端进行配置	747
26.1.8 缓冲区的处理	749
26.1.9 写入配置、使配置生效: tcsetattr() 函数	749
26.1.10 读写数据: read()、write()	750
26.2 串口应用编程实战	750
26.3 在开发板上进行测试	758

第二十七章 看门狗应用编程	763
27.1 看门狗应用编程介绍	764
27.1.1 打开设备	765
27.1.2 获取设备支持哪些功能: <code>WDIOC_GETSUPPORT</code>	765
27.1.3 获取/设置超时时间: <code>WDIOC_GETTIMEOUT</code> 、 <code>WDIOC_SETTIMEOUT</code>	766
27.1.4 开启/关闭看门狗: <code>WDIOC_SETOPTIONS</code>	766
27.1.5 喂狗: <code>WDIOC_KEEPAALIVE</code>	767
27.2 看门狗应用编程实战	767
第二十八章 音频应用编程	771
28.1 ALSA 概述	772
28.2 alsa-lib 简介	772
28.3 sound 设备节点	775
28.4 alsa-lib 移植	777
28.5 编写一个简单地 alsa-lib 应用程序	784
28.5.1 一些基本概念	784
28.5.2 打开 PCM 设备	788
28.5.3 设置硬件参数	789
28.5.4 读/写数据	794
28.5.5 示例代码之 PCM 播放	795
28.5.6 示例代码值 PCM 录音	805
28.6 使用异步方式	813
28.6.1 PCM 播放示例-异步方式	814
28.6.2 PCM 录音示例-异步方式	823
28.7 使用 <code>poll()</code> 函数	829
28.7.1 使用 <code>poll</code> I/O 多路复用实现读写	829
28.7.2 PCM 播放示例代码	830
28.7.3 PCM 录音示例代码	838
28.8 PCM 设备的状态	844
28.8.1 PCM 播放示例代码-加入状态控制	847
28.8.2 <code>snd_pcm_readi/snd_pcm_writei</code> 错误处理	858
28.9 混音器设置	859
28.9.1 打开混音器: <code>snd_mixer_open</code>	860
28.9.2 Attach 关联设备: <code>snd_mixer_attach</code>	860
28.9.3 注册: <code>snd_mixer_selem_register</code>	861
28.9.4 加载: <code>snd_mixer_load</code>	861
28.9.5 查找元素	861
28.9.6 获取/更改元素的配置值	862
28.9.7 示例程序	865
28.10 回环测试例程	878
28.11 总结	880
28.11.1 ALSA 插件 (plugin)	880
第二十九章 网络基础知识	884

29.1 网络通信概述	885
29.2 网络互连模型: OSI 七层模型	885
29.2.1 TCP/IP 四层/五层模型	888
29.2.2 数据的封装与拆封	889
29.3 IP 地址	890
29.3.1 IP 地址的编址方式	890
29.3.2 IP 地址的分类	890
29.3.3 特殊的 IP 地址	892
29.3.4 如何判断 2 个 IP 地址是否在同一个网段内	893
29.4 TCP/IP 协议	893
29.5 TCP 协议	894
29.5.1 TCP 协议的特性	894
29.5.2 TCP 报文格式	895
29.5.3 建立 TCP 连接: 三次握手	897
29.5.4 关闭 TCP 连接: 四次挥手	899
29.5.5 TCP 状态说明	901
29.5.6 UDP 协议	901
29.6 端口号的概念	902
第三十章 socket 编程基础	903
30.1 socket 简介	904
30.2 socket 编程接口介绍	904
30.2.1 socket()函数	904
30.2.2 bind()函数	906
30.2.3 listen()函数	907
30.2.4 accept()函数	907
30.2.5 connect()函数	908
30.2.6 发送和接收函数	908
30.2.7 close()关闭套接字	909
30.3 IP 地址格式转换函数	909
30.3.1 inet_aton、inet_addr、inet_ntoa 函数	910
30.3.2 inet_ntop、inet_pton 函数	910
30.4 socket 编程实战	912
30.4.1 编写服务器程序	912
30.4.2 编写客户端程序	915
30.4.3 编译测试	917
第三十一章 CAN 应用编程基础	919
31.1 CAN 基础知识	920
31.1.1 什么是 CAN?	920
31.1.2 CAN 的特点	921
31.1.3 CAN 的电气属性	921
31.1.4 CAN 网络拓扑	922
31.1.5 CAN 总线通信模型	923

31.1.6 CAN 帧的种类.....	924
31.2 SocketCan 应用编程.....	925
31.2.1 创建 socket 套接字.....	925
31.2.2 将套接字与 CAN 设备进行绑定.....	925
31.2.3 设置过滤规则.....	926
31.2.4 数据发送/接收.....	926
31.2.5 回环功能设置.....	928
31.3 CAN 应用编程实战.....	928
31.3.1 CAN 数据发送实例.....	932
31.3.2 CAN 数据接收实例.....	935
第三篇 进阶篇.....	939
第三十二章 CMake 入门与进阶.....	940
32.1 cmake 简介.....	941
32.2 cmake 和 Makefile.....	941
32.3 cmake 的使用方法.....	941
32.3.1 示例一: 单个源文件.....	942
32.3.2 示例二: 多个源文件.....	945
32.3.3 示例三: 生成库文件.....	946
32.3.4 示例四: 将源文件组织到不同的目录.....	948
32.3.5 示例五: 将生成的可执行文件和库文件放置到单独的目录下.....	949
32.4 CMakeLists.txt 语法规则.....	950
32.4.1 简单地语法介绍.....	950
32.4.2 部分常用命令.....	951
32.4.3 部分常用变量.....	965
32.4.4 双引号的作用.....	975
32.4.5 条件判断.....	976
32.4.6 循环语句.....	986
32.4.7 数学运算 math.....	991
32.5 cmake 进阶.....	992
32.5.1 定义函数.....	992
32.5.2 宏定义.....	996
32.5.3 文件操作.....	999
32.5.4 设置交叉编译.....	1005
32.5.5 变量的作用域.....	1011
32.5.6 属性.....	1017
32.6 总结.....	1024
第三十三章 实战小项目之 MQTT 物联网.....	1025
33.1 MQTT 简介.....	1026
33.1.1 MQTT 的主要特性.....	1026
33.1.2 MQTT 历史.....	1026
33.1.3 MQTT 版本.....	1028
33.2 MQTT 协议(上).....	1028

33.2.1 MQTT 通信基本原理	1028
33.2.2 连接 MQTT 服务端	1031
33.2.3 断开连接	1035
33.2.4 发布消息、订阅主题与取消订阅主题.....	1035
33.2.5 主题的进阶知识	1038
33.3 MQTT 初体验	1041
33.3.1 下载、安装 MQTT.fx 客户端软件.....	1041
33.3.2 MQTT 服务端	1043
33.3.3 动手测试	1045
33.3.4 使用手机作为客户端	1050
33.4 MQTT 协议 (下)	1055
33.4.1 QoS 是什么?	1055
33.4.2 保留消息	1060
33.4.3 MQTT 的心跳机制	1062
33.4.4 MQTT 的遗嘱机制	1063
33.4.5 MQTT 用户密码认证	1065
33.5 移植 MQTT 客户端库	1071
33.6 MQTT 客户端库 API 介绍.....	1080
33.6.1 MQTTClient_message 结构体.....	1080
33.6.2 创建一个客户端对象	1081
33.6.3 连接服务端	1082
33.6.4 设置回调函数	1083
33.6.5 发布消息	1085
33.6.6 订阅主题和取消订阅主题	1086
33.6.7 断开服务端连接	1087
33.7 编写客户端程序	1087
33.8 演示	1094
第三十四章 实战小项目之视频监控	1100
34.1 视频监控简介	1101
34.2 Nginx 移植	1101
34.2.1 下载 Nginx 源码	1101
34.2.2 下载 nginx-rtmp-module 模块	1102
34.2.3 交叉编译 Nginx	1102
34.3 测试 nginx	1110
34.4 配置 nginx	1111
34.5 使用 FFmpeg 推流	1112

前言

首先欢迎大家阅读本篇文档, 本文档为正点原子 Linux 团队所编写的《Linux C 应用编程指南》, 基于 Linux 系统的 C 语言应用编程学习指南, 其主要内容包括 Linux 基础应用编程知识、嵌入式系统硬件设备应用编程以及实战小项目。本编程指南定义为基础入门文档, 其中并不会对所讲解之内容进行深入剖析, 适用于 Linux 应用编程初学者, 如果您已有多年的 Linux 应用编程经验, 熟悉各种应用场合的 Linux 应用开发, 那本文档并不适合您阅读。对于 Linux 应用编程小白, 如果您对 Linux 应用编程感兴趣亦或是想着将来从事一份与之相关的工作, 那么本文档是一个不错的选择!

虽然本文档定义为基础入门文档, 但是对读者还有一定的要求, 当然要求并不过分, 只需要大家有一定的 C 语言编程以及 Linux 操作系统功底, 那么就满足本文档的学习要求了; 学习应用开发并不要求大家先学会驱动开发, 应用开发与驱动开发本就是两个不同的方向, 将来在工作当中也会负责不同的任务、解决不同的问题, 应用程序负责处理应用层用户需求、逻辑, 而驱动程序负责内核层硬件底层操作, Linux 操作系统向应用层提供了封装好的 API 函数 (也称为系统调用), 应用层通过系统调用可以完成对系统硬件设备的操作、控制; 所以学习应用开发并不要求大家得学会驱动开发, 不过话又说回来, 如果大家学过驱动开发, 那么必然对你学习应用开发是有一定帮助的, 当然学习 Linux 应用开发并不针对于嵌入式行业, 也可从事 PC 端应用程序开发工作以及其它相关工作。

Linux 应用开发学习与具体硬件平台无关, 可以在 PC 端 Linux 操作系统下进行编程开发学习, 譬如 Ubuntu、CentOS、Debian 等, 亦可使用嵌入式 Linux 系统开发平台进行编程实战测试, 譬如正点原子推出的 ALPHA/Mini I.MX6U Linux 开发板、MP157 开发板以及领航者/启明星 ZYNQ 开发板。

如何帮助初学者顺利的转入 Linux 应用开发, 为此笔者将本文章内容规划如下, 内容总共分为三大块:

第一篇、入门篇

该部分内容基本涵盖了应用编程中的各种应用场合, 学习应用编程必须要学习的内容, 本部分内容包括: 文件 IO 操作、文件高级 IO、文件属性、系统信息、进程、线程、进程间通信、信号以及线程同步等内容。为学习后面两篇的内容打下坚实的基础!

第二篇、提高篇

在调篇内容中, 本书将会向大家介绍如何编写应用程序控制硬件外设, 本书以正点原子 ALPHA/Mini I.MX6U 开发板为例, 向大家介绍如何编写应用程序控制板子上的各种硬件外设。譬如 LED、GPIO、PWM、串口、摄像头、LCD、看门狗、音频、网络编程等。

尤其网络编程最为重要, 同样也是最难的。Linux 系统是依靠互联网平台迅速发展起来的, 所以它具有强大的网络功能支持, 也是 Linux 系统的一大特点, Linux 系统向应用层提供了丰富的 API 函数, 譬如 socket 接口及其相关函数。本部分内容会向大家介绍网络基础知识, 譬如网络编程架构、网卡、路由器、交换机、TCP/IP 协议等等, 最后向大家介绍如何使用 socket 接口进行网络编程开发; 本文档定义为基础入门教程, 其中并不会过深地给大家讲解网络编程相关内容, 旨在以引导大家入门为主; 其一在于 Linux 网络编程本就是一门非常难、非常深奥的技能, 市面上有很多关于 Linux/UNIX 网络编程类书籍, 这些书籍专门介绍了网络编程相关知识内容, 而且书本非常厚, 可将其内容之多、难点之多; 其二在于笔者对网络编程了解知之甚少, 掌握的知识、技能太少, 无法向大家传授更加深入的知识、内容, 如果大家以后会从事 Linux 网络编程相关工作, 可以购买此类书籍深入学习、研究。

第三篇、进阶篇

在本书的第三篇内容中, 将会规划一些实战小项目来跟大家一起完成, 譬如 MQTT 实战小项目, 具体的小项目目前还在规划当中, 大家可以期待! 学习第三篇的内容可以帮助提升大家的编程能力、开发能力、以及遇到问题时的解决能力, 当然前提条件是需要掌握好第一篇以及第二篇的内容。

关于本文档内容规划就给大家介绍到这里,对于 Linux 应用编程,其实市面上有很多相关书籍,其中不乏有很多笔者认为非常好的书籍,大家也可以购买这些书籍,用于在学习过程中参考、查阅,这里笔者给大家推荐三本书籍,分别为《UNIX 环境高级编程》、《Linux/UNIX 系统编程手册》(分为上册和下册)、《UNIX 网络编程》,这些书籍可以在淘宝网上购买得到,当然也有 PDF 电子版的可供大家选择;这三本书籍都是非常不错的,内容涵盖广、知识点讲解到位、深入剖析,不管是初学者还是拥有多年编程经验的工程师,这三本书籍都是不错的选择!当然除此之外,还有其它很多不错的书籍,笔者阅读此类书籍甚少,就不便给大家推荐其它的了,如果大家还有其它更好的书籍推荐,欢迎大家在评论区留言!

最后感谢大家阅读本书,您对我们的支持和信任,是我们永往直前的信心和勇气,如果大家在学习过程遇到了什么问题,正点原子为大家提供了一个技术交流平台 <http://www.openedv.com/> 开源电子网,有什么问题可以通过该平台与众多 Linux 开发者一起交流、讨论、一起学习、一起进步、一起成长,同时,由于笔者知识水平有限,文中错漏与不足之处在所难免,恳请各位读者不吝赐教(笔者 QQ: 773904075)。

第一篇 入门篇

本篇作为 Linux 应用开发入门篇, 主要为大家讲解 Linux 应用开发过程当中会用到的基础入门知识, 譬如 Linux 文件 IO 操作、高级 IO 操作、文件属性、系统信息获取与设置、进程、进程间通信、线程以及信号等基础应用编程知识, 为后面学习提高篇、进阶篇章内容打下坚实的基础。

为了更好的学习本篇内容, 推荐大家购买《Linux/UNIX 系统编程手册》、《UNIX 环境高级编程》等这类书籍作为自己的学习参考资料, 在本文档前言部分也给大家介绍了三本书籍, 都是笔者认为写的非常好的 Linux 应用开发类书籍, 同样这些书籍也是笔者编写本文档所使用的主要参考资料, 所以强烈建议大家购买!

第一章 应用编程概念

对于大多数首次接触 Linux 应用编程的读者来说,可能对应用编程(也可称为系统编程)这个概念并不太了解,所以在正式学习 Linux 应用编程之前,笔者有必要向大家介绍这些简单基本的概念,从整体上认识到应用编程为何物?与驱动编程、裸机编程有何不同?

了解本章所介绍的内容是掌握应用编程的先决条件,所以本章主要内容便是对 Linux 应用编程进行一个简单地介绍,让读者对此有一个基本的认识。

本章将会讨论如下主题内容。

- 何为系统调用;
- 何为库函数;
- 应用程序的 main()函数;
- 应用程序开发环境的介绍。

1.1 系统调用

系统调用 (system call) 其实是 Linux 内核提供给应用层的应用编程接口 (API), 是 Linux 应用层进入内核的入口。不止 Linux 系统, 所有的操作系统都会向应用层提供系统调用, 应用程序通过系统调用来使用操作系统提供的各种服务。

通过系统调用, Linux 应用程序可以请求内核以自己的名义执行某些事情, 譬如打开磁盘中的文件、读写文件、关闭文件以及控制其它硬件外设。

通过系统调用 API, 应用层可以实现与内核的交互, 其关系可通过下图简单描述:

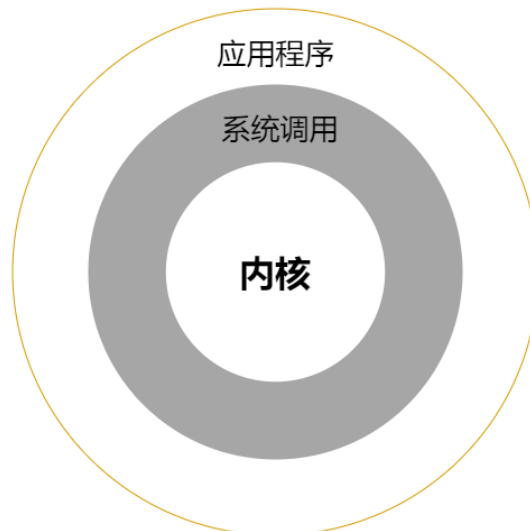


图 1.1.1 内核、系统调用与应用程序

内核提供了一系列的服务、资源、支持一系列功能, 应用程序通过调用系统调用 API 函数来使用内核提供的服务、资源以及各种各样的功能, 如果大家接触过其它操作系统编程, 想必对此并不陌生, 譬如 Windows 应用编程, 操作系统内核一般都会向应用程序提供应用编程接口 API, 否则我们将无法使用操作系统。

应用编程与裸机编程、驱动编程有什么区别?

在学习应用编程之前, 相信大家都有过软件开发经验, 譬如 51、STM32 等单片机软件开发、以及嵌入式 Linux 硬件平台下的驱动开发等, 51、STM32 这类单片机的软件开发通常是裸机程序开发, 并不会涉及到操作系统的概念, 那应用编程与裸机编程以及驱动开发有什么区别呢?

就拿嵌入式 Linux 硬件平台下的软件开发来说, 我们大可将编程分为三种, 分别为裸机编程、Linux 驱动编程以及 Linux 应用编程。首先对于裸机编程这个概念来说很好理解, 一般把没有操作系统支持的编程环境称为裸机编程环境, 譬如单片机上的编程开发, 编写直接在硬件上运行的程序, 没有操作系统支持; 狭义上 Linux 驱动编程指的是基于内核驱动框架开发驱动程序, 驱动开发工程师通过调用 Linux 内核提供的接口完成设备驱动的注册, 驱动程序负责底层硬件操作相关逻辑, 如果学习过 Linux 驱动开发的读者, 想必对此并不陌生; 而 Linux 应用编程 (系统编程) 则指的是基于 Linux 操作系统的应用编程, 在应用程序中通过调用系统调用 API 完成应用程序的功能和逻辑, 应用程序运行于操作系统之上。通常在操作系统下有两种不同的状态: 内核态和用户态, 应用程序运行在用户态、而内核则运行在内核态。

关于应用编程这个概念, 以上给大家解释得很清楚了, 笔者以实现点亮一个 LED 功能为例, 给大家简单地说明三者之间的区别, LED 裸机程序如下所示:

示例代码 1.1.1 LED 裸机程序

```
static void led_on(void)
{
```

```
    /* 点亮 LED 硬件操作代码 */
}

static void led_off(void)
{
    /* 熄灭 LED 硬件操作代码 */
}

int main(void)
{
    /* 用户逻辑 */
    for (;;) {

        led_on();      //点亮 LED
        delay();       //延时
        led_off();     //熄灭 LED
        delay();       //延时
    }
}
```

可以看到在裸机程序当中,LED 硬件操作代码与用户逻辑代码全部都是在同一个源文件(同一个工程)中实现的,硬件操作代码与用户逻辑代码没有隔离,没有操作系统支持,代码编译之后直接在硬件平台运行,俗称“裸跑”。我们再来看一个 Linux 系统下的 LED 驱动程序示例代码,如下所示:

示例代码 1.1.2 Linux 下 LED 驱动程序

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/of_gpio.h>
#include <linux/delay.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>

static void led_on(void)
{
    /* 点亮 LED 硬件操作代码 */
}

static void led_off(void)
{
    /* 熄灭 LED 硬件操作代码 */
}

static int led_open(struct inode *inode, struct file *filp)
{
    /* 打开设备时需要做的事情 */
}
```



```
}

static ssize_t led_write(struct file *filp, const char __user *buf,
                        size_t size, loff_t *offt)
{
    int flag;

    /* 获取应用层 write 的数据,存放在 flag 变量 */
    if (copy_from_user(&flag, buf, size))
        return -EFAULT;

    /* 判断用户写入的数据,如果是 0 则熄灭 LED,如果是非 0 则点亮 LED */
    if (flag)
        led_on();
    else
        led_off();

    return 0;
}

static int led_release(struct inode *inode, struct file *filp)
{
    /* 关闭设备时需要做的事情 */
}

static struct file_operations led_fops = {
    .owner    = THIS_MODULE,
    .open     = led_open,
    .write    = led_write,
    .release  = led_release,
};

static int led_probe(struct platform_device *pdev)
{
    /* 驱动加载时需要做的事情 */
}

static int led_remove(struct platform_device *pdev)
{
    /* 驱动卸载时需要做的事情 */
}

static const struct of_device_id led_of_match[] = {
```

```

    { .compatible = "alientek,led", },
    { /* sentinel */ },
};
MODULE_DEVICE_TABLE(of, led_of_match);

static struct platform_driver led_driver = {
    .driver = {
        .owner          = THIS_MODULE,
        .name           = "led",
        .of_match_table = led_of_match,
    },
    .probe = led_probe,
    .remove = led_remove,
};
module_platform_driver(led_driver);

MODULE_DESCRIPTION("LED Driver");
MODULE_LICENSE("GPL");

```

以上并不是一个完整的 LED 驱动代码, 如果没有接触过 Linux 驱动开发的读者, 看不懂也没有关系, 并无大碍, 此驱动程序使用了最基本的字符设备驱动框架编写而成, 非常简单; led_fops 对象中提供了 open、write、release 方法, 当应用程序调用 open 系统调用打开此 LED 设备时会执行到 led_open 函数, 当调用 close 系统调用关闭 LED 设备时会执行到 led_release 函数, 而调用 write 系统调用时会执行到 led_write 函数, 此驱动程序的设定是当应用层调用 write 写入 0 时熄灭 LED, write 写入非 0 时点亮 LED。

驱动程序属于内核的一部分, 当操作系统启动的时候会加载驱动程序, 可以看到 LED 驱动程序中仅仅实现了点亮/熄灭 LED 硬件操作相关逻辑代码, 应用程序可通过 write 这个系统调用 API 函数控制 LED 亮灭; 接下来我们看看 Linux 系统下的 LED 应用程序示例代码, 如下所示:

示例代码 1.1.3 Linux 下 LED 应用程序

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int fd;
    int data;

    fd = open("/dev/led", O_WRONLY); //打开 LED 设备(假定 LED 的设备文件为/dev/led)
    if (0 > fd)
        return -1;

    for (;;) {

```

```
data = 1;
write(fd, &data, sizeof(data)); //写 1 点亮 LED
sleep(1);           //延时 1 秒

data = 0;
write(fd, &data, sizeof(data)); //写 0 熄灭 LED
sleep(1);           //延时 1 秒
}

close(fd);
return 0;
}
```

此应用程序也非常简单, 仅只需实现用户逻辑代码即可, 循环点亮、熄灭 LED, 并不需要实现硬件操作相关, 示例代码中调用了 `open`、`write`、`close` 这三个系统调用 API 接口, `open` 和 `close` 分别用于打开/关闭 LED 设备, `write` 写入数据传给 LED 驱动, 传入 0 熄灭 LED, 传入非 0 点亮 LED。

LED 应用程序与 LED 驱动程序是分隔、分离的, 它们单独编译, 它们并不是整合在一起的, 应用程序运行在操作系统之上, 有操作系统支持, 应用程序处于用户态, 而驱动程序处于内核态, 与纯粹的裸机程序存在着质的区别。Linux 应用开发与驱动开发是两个不同的方向, 将来在工作当中也会负责不同的任务、解决不同的问题。

关于本小节系统调用概念相关的内容就介绍到这里了, 接下来向大家介绍库函数。

1.2 库函数

前面给大家介绍了系统调用, 系统调用是内核直接向应用层提供的编程接口, 譬如 `open`、`write`、`read`、`close` 等, 关于这些系统调用后面会给大家进行详细介绍。编写应用程序除了使用系统调用之外, 我们还可以使用库函数, 本小节来聊一聊库函数。

库函数也就是 C 语言库函数, C 语言库是应用层使用的一套函数库, 在 Linux 下, 通常以动态 (.so) 库文件的形式提供, 存放在根文件系统/lib 目录下, C 语言库函数构建于系统调用之上, 也就是说库函数其实是由系统调用封装而来的, 当然也不能完全这么说, 原因在于有些库函数并不调用任何系统调用, 譬如一些字符串处理函数 `strlen()`、`strcat()`、`memcpy()`、`memset()`、`strchr()` 等等; 而有些库函数则会使用系统调用来帮它完成实际的操作, 譬如库函数 `fopen` 内部调用了系统调用 `open()` 来帮它打开文件、库函数 `fread()` 就利用了系统调用 `read()` 来完成读文件操作、`fwrite()` 就利用了系统调用 `write()` 来完成写文件操作。

Linux 系统内核提供了一系列的系统调用供应用层使用, 我们直接使用系统调用就可以了呀, 那为何还要设计出库函数呢? 事实上, 有些系统调用使用起来并不是很方便, 于是就出现了 C 语言库, 这些 C 语言库函数的设计是为了提供比底层系统调用更为方便、更为好用、且更具有可移植性的调用接口。

来看一看它们之间的区别:

- 库函数是属于应用层, 而系统调用是内核提供给应用层的编程接口, 属于系统内核的一部分;
- 库函数运行在用户空间, 调用系统调用会由用户空间 (用户态) 陷入到内核空间 (内核态);
- 库函数通常是有缓存的, 而系统调用是无缓存的, 所以在性能、效率上, 库函数通常要优于系统调用;
- 可移植性: 库函数相比于系统调用具有更好的可移植性, 通常对于不同的操作系统, 其内核向应用层提供的系统调用往往都是不同, 譬如系统调用的定义、功能、参数列表、返回值等往往都是不一样的; 而对于 C 语言库函数来说, 由于很多操作系统都实现了 C 语言库, C 语言库在不同的操作

系统之间其接口定义几乎是一样的, 所以库函数在不同操作系统之间相比于系统调用具有更好的可移植性。

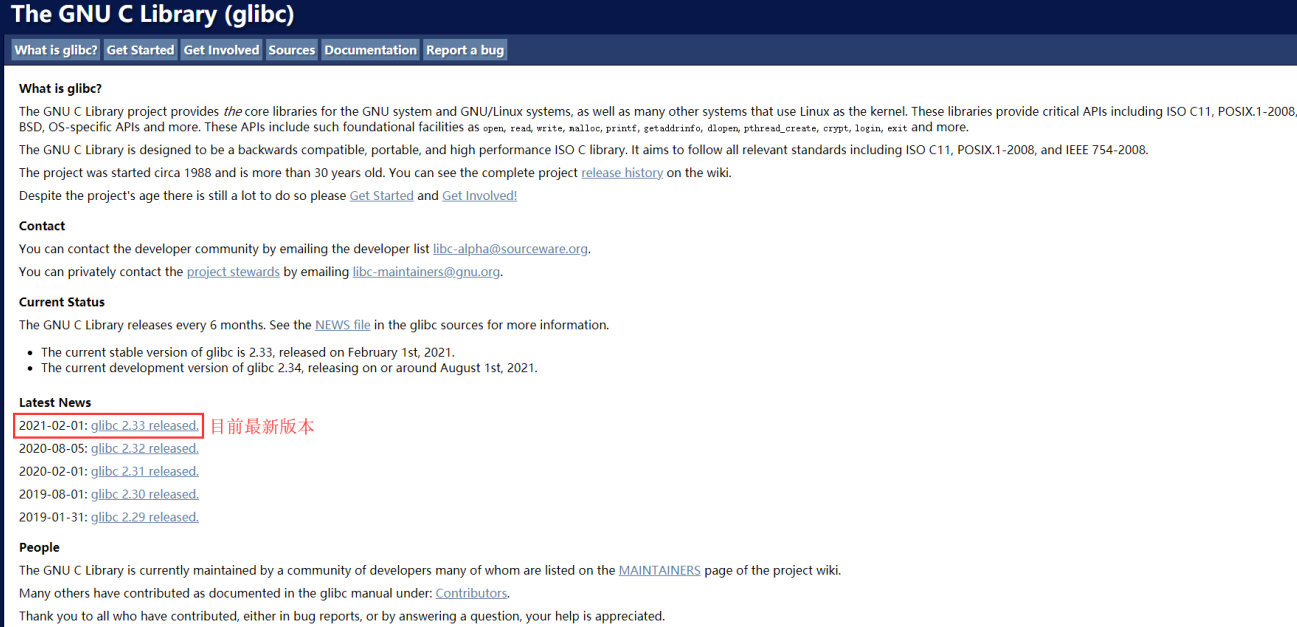
以上便上它们之间一个大致的区别, 从实现者的角度来看, 系统调用与库函数之间有根本的区别, 但从用户使用角度来看, 其区别并不重要, 它们都是 C 语言函数。在实际应用编程中, 库函数和系统调用都会使用到, 所以对于我们来说, 直接把它们当做是 C 函数即可, 知道你自己调用的函数是系统调用还是库函数即可, 不用太过于区分它们之间的差别。

所以应用编程简单点来说就是: 开发 Linux 应用程序, 通过调用内核提供的系统调用或使用 C 库函数来开发具有相应功能的应用程序。

1.3 标准 C 语言函数库

在 Linux 系统下, 使用的 C 语言库为 GNU C 语言函数库 (也叫作 glibc, 其网址为 <http://www.gnu.org/software/libc/>), 作为 Linux 下的标准 C 语言函数库。

进入到 <http://www.gnu.org/software/libc/> 网址, 如下所示:



The GNU C Library (glibc)

[What is glibc?](#) [Get Started](#) [Get Involved](#) [Sources](#) [Documentation](#) [Report a bug](#)

What is glibc?

The GNU C Library project provides *the* core libraries for the GNU system and GNU/Linux systems, as well as many other systems that use Linux as the kernel. These libraries provide critical APIs including ISO C11, POSIX.1-2008, BSD, OS-specific APIs and more. These APIs include such foundational facilities as `open`, `read`, `write`, `malloc`, `printf`, `getaddrinfo`, `dlopen`, `pthread_create`, `crypt`, `login`, `exit` and more.

The GNU C Library is designed to be a backwards compatible, portable, and high performance ISO C library. It aims to follow all relevant standards including ISO C11, POSIX.1-2008, and IEEE 754-2008.

The project was started circa 1988 and is more than 30 years old. You can see the complete project [release history](#) on the wiki.

Despite the project's age there is still a lot to do so please [Get Started](#) and [Get Involved!](#)

Contact

You can contact the developer community by emailing the developer list libc-alpha@sourceware.org.

You can privately contact the [project stewards](#) by emailing libc-maintainers@gnu.org.

Current Status

The GNU C Library releases every 6 months. See the [NEWS file](#) in the glibc sources for more information.

- The current stable version of glibc is 2.33, released on February 1st, 2021.
- The current development version of glibc 2.34, releasing on or around August 1st, 2021.

Latest News

[2021-02-01: glibc 2.33 released.](#) 目前最新版本

[2020-08-05: glibc 2.32 released.](#)

[2020-02-01: glibc 2.31 released.](#)

[2019-08-01: glibc 2.30 released.](#)

[2019-01-31: glibc 2.29 released.](#)

People

The GNU C Library is currently maintained by a community of developers many of whom are listed on the [MAINTAINERS](#) page of the project wiki.

Many others have contributed as documented in the glibc manual under: [Contributors](#).

Thank you to all who have contributed, either in bug reports, or by answering a question, your help is appreciated.

图 1.3.1 glibc 官网

点击上面的 Sources 选项可以查看它的源码实现:

The GNU C Library (glibc)

What is glibc? Get Started Get Involved **Sources** Documentation Report a bug

Download sources

Checkout the latest glibc in development:

```
git clone https://sourceware.org/git/glibc.git
cd glibc
git checkout master
```

从git仓库中获取最新的源码

Releases are available by source branch checkout ([gitweb](#)) and tarball [via ftp](#).

Checkout the latest glibc 2.33 stable release:

```
git clone https://sourceware.org/git/glibc.git
cd glibc
git checkout release/2.33/master
```

从git仓库中获取最新、稳定版本的源码

Release tarballs are available via anonymous ftp at <http://ftp.gnu.org/glibc/> and its mirrors.

还可以通过ftp下载源码

图 1.3.2 获取源码的方式

glibc 源码的获取方式很简单，直接直接从 git 仓库下载，也可以通过 ftp 下载，如果大家有兴趣、或者想要了解某一个库函数它的具体实现，那么就可以获取到它源码来进行分析，好了，这里就不再多说了！

确定 Linux 系统的 glibc 版本

前面提到过了，C 语言库是以动态库文件的形式提供的，通常存放在 /lib 目录，它的命名方式通常是 libc.so.6，不过这个是一个软链接文件，它会链接到真正的库文件。

进入到 Ubuntu 系统的 /lib 目录下，笔者使用的 Ubuntu 版本为 16.04，在我的 /lib 目录下并没有发现 libc.so.6 这个文件，其实是在 /lib/x86_64-linux-gnu 目录下，进入到该目录：

```
dt@dt-virtual-machine:/lib/x86_64-linux-gnu$
dt@dt-virtual-machine:/lib/x86_64-linux-gnu$ ls -l libc.so.6
lrwxrwxrwx 1 root root 12 4月 22 02:13 libc.so.6 -> libc-2.23.so
dt@dt-virtual-machine:/lib/x86_64-linux-gnu$
dt@dt-virtual-machine:/lib/x86_64-linux-gnu$
```

图 1.3.3 libc.so.6 文件

可以看到 libc.so.6 链接到了 libc-2.23.so 库文件，2.23 表示的就是这个 glibc 库的版本号为 2.23。除此之外，我们还可以直接运行该共享库来获取到它的信息，如下所示：

```
dt@dt-virtual-machine:/lib/x86_64-linux-gnu$ ./libc.so.6
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11.3) stable release version 2.23 by Roland McGrath et al.
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Compiled by GNU CC version 5.4.0 20160609.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
dt@dt-virtual-machine:/lib/x86_64-linux-gnu$
```

glibc库的版本号

图 1.3.4 确定 glibc 版本号

从打印信息可以看到，笔者所使用的 Ubuntu 系统对应的 glibc 版本号为 2.23。

1.4 main 函数

对学习过 C 语言编程的读者来说, 譬如单片机编程、Windows 应用编程等, main 函数想必大家再熟悉不过了, 很多编程开发都是以 main 函数作为程序的入口函数, 同样在 Linux 应用程序中, main 函数也是作为应用程序的入口函数存在, main 函数的形参一般会有两种写法, 如果执行应用程序无需传参, 则可以写成如下形式:

示例代码 1.4.1 main 函数写法之无传参

```
int main(void)
{
    /* 代码 */
}
```

如果在执行应用程序的时候需要向应用程序传递参数, 则写法如下:

示例代码 1.4.2 main 函数写法之有传参

```
int main(int argc, char **argv)
{
    /* 代码 */
}
```

argc 形参表示传入参数的个数, 包括应用程序自身路径和程序名, 譬如运行当前目录下的 hello 可执行文件, 并且传入参数, 如下所示:

```
./hello 112233
```

那么此时参数个数为 2, 并且这些参数都是作为字符串的形式传递给 main 函数:

argv[0]等于"./hello"

argv[1]等于"112233"

有传参时 main 函数的写法并不只有这一种, 只是这种写法最常用, 对于其它的写法, 后面学习过程中如果遇到了再给大家进行讲解, 这里暂时先不去管。

1.5 本书使用的开发环境

对于编程开发, 大家可能都比较关心开发环境的问题, 譬如本书将使用什么 IDE 编写应用程序之类的问题, 本小节将对开发环境的问题进行一个简单介绍。

首先, 本书分为三篇内容, 在前言部分已经向各位读者介绍了本书的内容规划, 分为入门篇、提高篇以及进阶篇。因为是 Linux 应用编程, 所以本书将会在 Linux 操作系统环境下编写示例代码, 为了同正点原子的其它开发文档保持一致, 选择 Ubuntu 操作系统, 推荐大家使用 16.04 或 14.04 版本的 Ubuntu 系统, 个人觉得这两个版本比较稳定; 除了使用 Ubuntu 系统外, 大家还可以选择其它 Linux 发行版, 譬如 CentOS、Redhat (红帽) 等。

在 Linux 操作系统下, 也有很多比较好用的 IDE 软件, 可以帮助我们更为轻松的进行软件开发, 譬如 Eclipse、vscode 等, 如果你会使用 Eclipse, 可以在 Ubuntu 系统下安装 Eclipse 进行 Linux 应用开发, 1.5.1 小节介绍了如何在 Ubuntu 系统下安装 Eclipse; 如果你不会使用 Eclipse, 那就建议你使用 vscode, 同样本书也是使用 vscode, 与正点原子的驱动开发文档保持一致; vscode 的安装方法和使用方法本书不再介绍, 相信大家在自己的 Ubuntu 系统下已经安装好了, 如果没有安装的, 直接参考“[开发板光盘资料 A-基础资料/【正点原子】I.MX6U 嵌入式 Linux 驱动开发指南 V1.5.pdf](#)”文档第四章的 4.5 小节内容进行安装即可!

Tips: 注意, 本文档并不介绍 Eclipse IDE 的使用方法, 笔者也用的不多, 只是告诉大家可以在 Ubuntu 使用这个 IDE, 并且后面给出了安装方法, 如果你以前经常使用它开发软件, 你可以尝试使用它在 Ubuntu

下开发 Linux 应用程序,如果你没用过就不要去碰了。用什么 IDE 都不重要,哪怕你直接使用 vi 编写程序都可以,我们的重点是学习应用编程!而不是学习 IDE 怎么用。

vscode 是一个代码编辑器,提供了很多好用的插件,譬如语法检测、高亮显示、智能补全等,大家可以根据自己的选择安装插件,在驱动开发指南文档中也都有介绍。

入门篇章内容中,主要向大家介绍了 Linux 应用编程所涉及到的各个知识点以及各种系统调用和库函数,本部分内容所编写的示例代码,均是在 Ubuntu 系统下进行测试、验证,也就是在 Ubuntu 系统下执行测试程序以验证所学知识内容。所以在该部分内容中,本书使用 vscode+gcc 的方式开发应用程序,vscode 作为代码编辑器、gcc 作为编译器,使用 vscode 编写完代码之后再使用 gcc 编译源码,得到可在 PC 端 Ubuntu 系统下运行的可执行文件。gcc 的使用方法在驱动开发指南中有介绍,参考“[开发板光盘资料 A-基础资料/【正点原子】I.MX6U 嵌入式 Linux 驱动开发指南 V1.5.pdf](#)”文档第三章的内容。

提高篇章内容中,将会以正点原子 ALPHA/Mini I.MX6U 开发板为例,向大家介绍如何编写应用程序控制开发板上的各种硬件外设,所以编写的应用程序是需要开发板上运行的,ALPHA/Mini I.MX6U 开发板是 ARM 架构硬件平台,所以源码需要使用交叉编译工具(ARM 架构下的 gcc 编译器)进行编译,得到可在开发板上运行的可执行文件。所以在第二篇章内容中,本书使用 vscode+ARM gcc (交叉编译工具)的方式开发应用程序。

入门篇和提高篇章内容中所编写的示例代码只有单个.c 源文件,但是对于一个真正的 Linux 应用程序项目来说,其原文件的个数肯定不止一个,可能有几十上百个,这个时候可能就需要用到 Makefile 来组织、管理工程源文件。直接通过 Makefile 来组织工程,对于一个大型软件工程来说通常是比较困难的,需要程序员对 Makefile 能够熟练使用,譬如 Linux 内核源码、U-Boot 源码等都是直接使用 Makefile 进行管理。对于一般程序员来说,很难做到熟练使用 Makefile。在第三篇(进阶篇)章节内容中,会向大家介绍 cmake 工具,使用 cmake 来管理我们的源码工程,相比于直接使用 Makefile,使用 cmake 会大大降低难度,提高我们的开发效率!

进阶篇章内容中,会给大家介绍一些 Linux 应用编程实战小项目,通过学习进一步提升 Linux 应用编程能力,本部内容将会使用 cmake+vscode 进行开发,其实 cmake 在实际的项目当中用的还是比较多的,譬如很多的开源软件都会选择使用 cmake 来构建、配置工程源码,学习了之后,相信大家在今后的工作当中也会用到。

关于本小节的内容到这里就结束了,下一章开始将正式进入到应用编程学习之路,大家加油!

1.5.1 Ubuntu 系统下安装 Eclipse

本小节介绍如何在 Ubuntu 系统下安装 Eclipse,笔者使用的 Ubuntu 系统版本号为 16.04.5,其它版本笔者未进行验证,不过大家试一试,如果出现了问题,再根据提示找到相应的解决办法!

Eclipse 是一个开源的 IDE,基于 java 语言开发的可扩展开发平台,支持很多插件,可用于诸如 java、C/C++、PHP、Android 等编程语言的软件开发,关于更加详细的介绍大家自己去了解,本书不再介绍!

(一)下载 Eclipse

开发板资料包中提供了 Eclipse 安装包,路径为:开发板光盘->3、软件->eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz。

首先我们要下载 Eclipse 安装包,进入到 Eclipse 官网: <https://www.eclipse.org/>,如下所示:

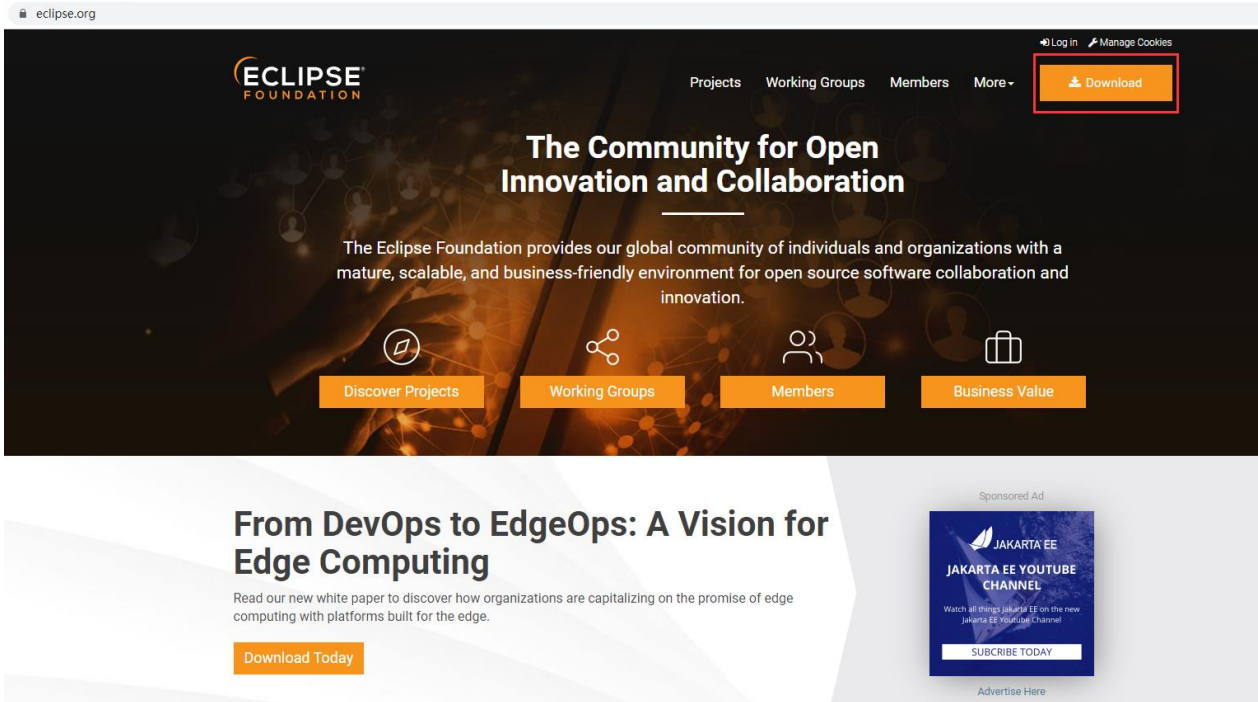


图 1.5.1 Eclipse 官网

点击右上角“Download”按钮进入到下载页面:

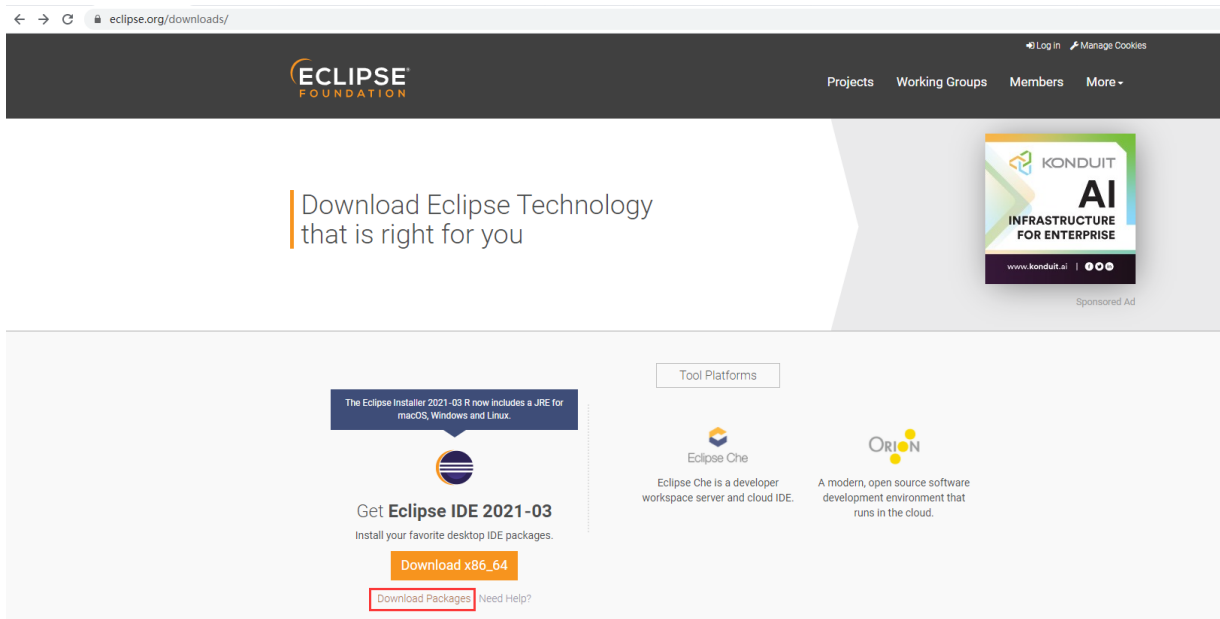


图 1.5.2 Download 页面

点击下边的“Download Package”选项:

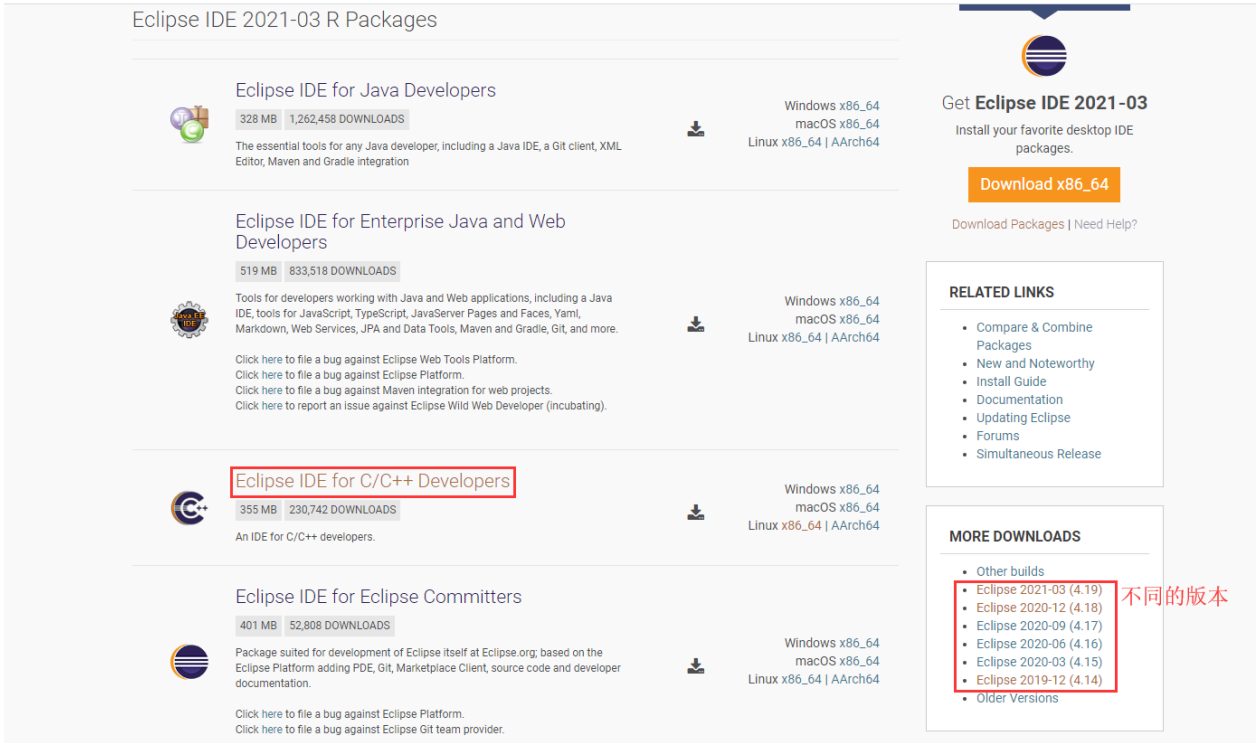


图 1.5.3 package 列表

当前页面显示的是最新版本的软件包，可以看到 Eclipse 的最新版本是 2021-03，上图右下角列出了其它的版本软件包的链接地址，这里笔者选择 2020-12，也就是 4.18 版本作为演示，其它版本笔者没测试过，理论上都是可以的。

点击 4.18 版本链接地址，页面跟上图一样，因为是使用 Eclipse 进行 C 语言应用程序开发，所以选择“Eclipse IDE for C/C++ Developers”软件包，选择 Linux 系统的安装包，点击下图红框中的 x86_64 链接：

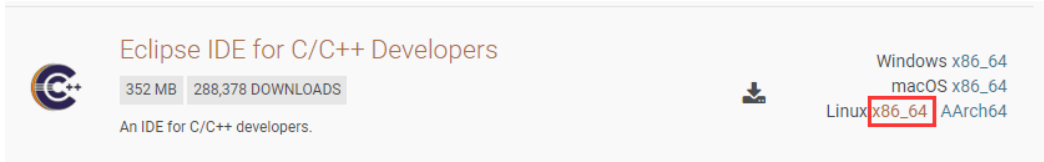


图 1.5.4 C/C++ IDE

点击之后会进入到下载界面，点击下图中的“Download”即可下载：

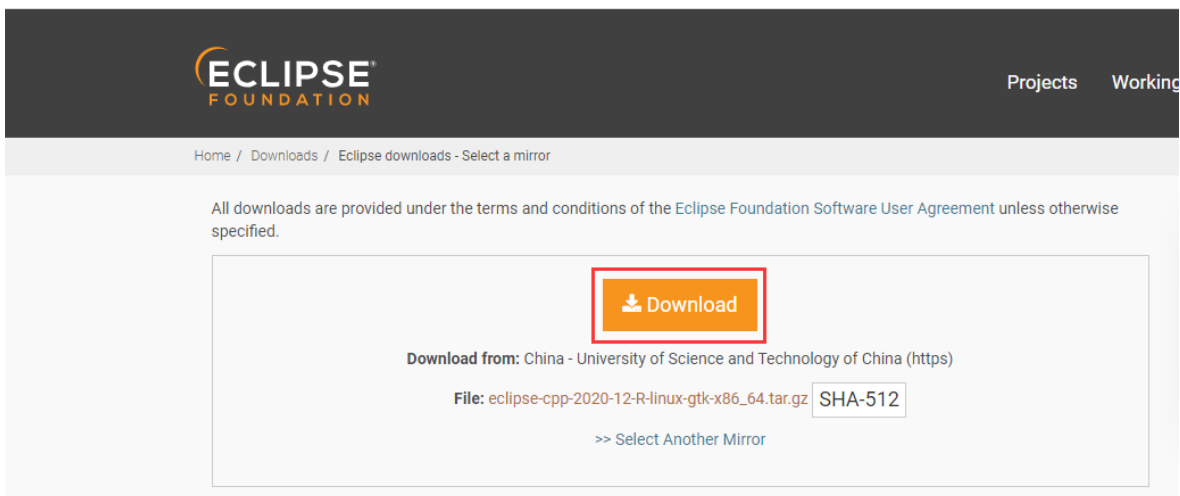


图 1.5.5 下载

下载完之后会得到压缩包文件“eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz”:

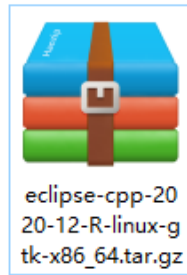


图 1.5.6 Eclipse 压缩包文件

②、下载 jdk

开发板资料包中提供了 jdk，路径为：**开发板光盘->3、软件->jdk-8u291-linux-x64.tar.gz**。

因为 Eclipse 是基于 java 语言开发的，所以它依赖于 java jdk，所以我们要下载，进入到 <https://www.oracle.com/cn/java/technologies/javase-downloads.html> 网页，如下所示：

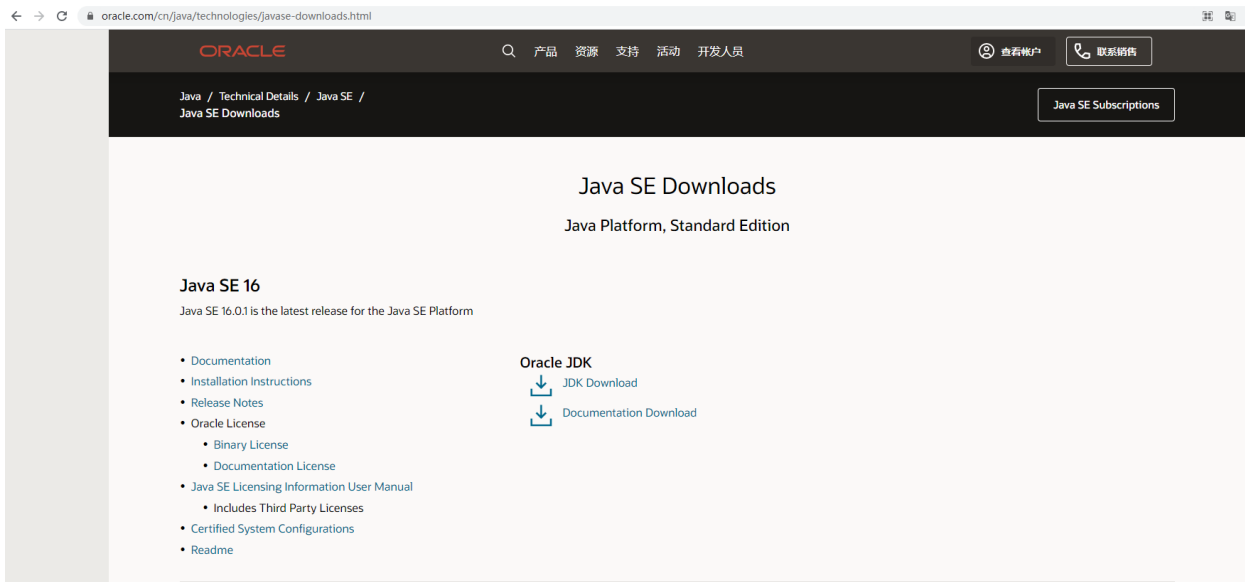


图 1.5.7 jdk 下载主页

我们往下翻找到 Java SE8，如下所示：

Java SE 8

Java SE 8u291 is the latest release for the Java SE 8 Platform.

- Documentation
- Installation Instructions
- Release Notes
- Oracle License
 - Binary License
 - Documentation License
 - BSD License
- Java SE Licensing Information User Manual
 - Includes Third Party Licenses
- Certified System Configurations
- Readme Files
 - JDK ReadMe
 - JRE ReadMe

Oracle JDK

- ↓ **JDK Download**
- ↓ Server JRE Download
- ↓ JRE Download
- ↓ Documentation Download
- ↓ Demos and Samples Download

图 1.5.8 Java SE8

点击 JDK Download 进入到 jdk8 下载页面，往下翻找到软件包下载列表：

Java SE Development Kit 8u291

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE

Product / File Description	File Size	Download
Linux ARM 64 RPM Package	59.1 MB	↓ jdk-8u291-linux-aarch64.rpm
Linux ARM 64 Compressed Archive	70.79 MB	↓ jdk-8u291-linux-aarch64.tar.gz
Linux ARM 32 Hard Float ABI	73.5 MB	↓ jdk-8u291-linux-arm32-vfp-hflt.tar.gz
Linux x86 RPM Package	109.05 MB	↓ jdk-8u291-linux-i586.rpm
Linux x86 Compressed Archive	137.92 MB	↓ jdk-8u291-linux-i586.tar.gz
Linux x64 RPM Package	108.78 MB	↓ jdk-8u291-linux-x64.rpm
Linux x64 Compressed Archive	138.22 MB	↓ jdk-8u291-linux-x64.tar.gz
macOS x64	207.42 MB	↓ jdk-8u291-macosx-x64.dmg

图 1.5.9 jdk8 下载页面

可以看到上图列举出了很多不同操作系统以及硬件平台的 jdk 包，操作系统包括 Linux、macOS 以及 Windows 等，还有不同的硬件平台，笔者用的是 64 位的 Ubuntu 系统，所以选择“Linux x64 Compressed Archive”，点击后边“jdk-8u291-linux-x64.tar.gz”下载。之后会弹出如下界面：

You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software. X

I reviewed and 勾选 accept the Oracle Technology Network License Agreement for Oracle Java SE

Download jdk-8u291-linux-x64.tar.gz  点击下载

图 1.5.10 下载

勾选图中所示的选项，之后点击进行下载。下载需要登录 Oracle 账户，譬如下图所示，填写用户名、密码点击登录之后才能下载；没有账户的读者需要点击下面的“创建账户”进行创建。

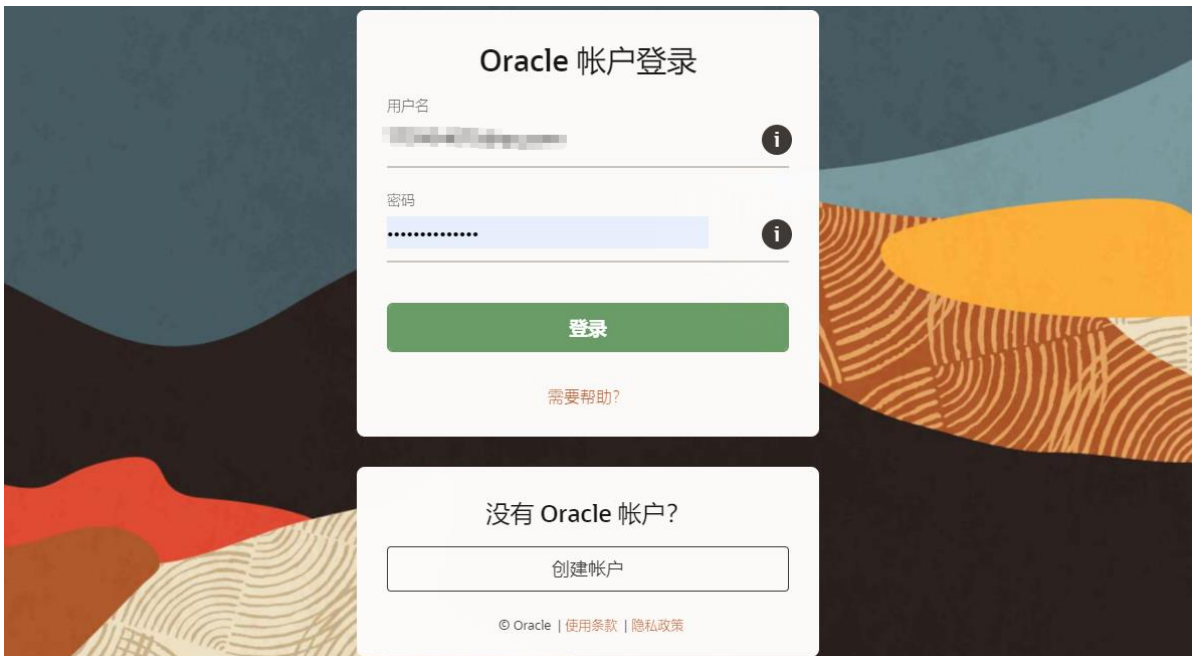


图 1.5.11 登录 Oracle 账户

下载完之后得到一个压缩包文件：



图 1.5.12 jdk 压缩包文件

三、安装 Eclipse

首先把前面下载好的两个压缩包文件拷贝到 Ubuntu 系统用户家目录下，如下所示：

```
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$ 
dt@dt-virtual-machine:~$ ls -l
总用量 508404
-rw-r--r-- 1 dt dt 375618953 6月 3 14:46 eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz
-rw-r--r-- 1 dt dt 8980 6月 3 13:02 examples.desktop
-rw-r--r-- 1 dt dt 144935989 6月 3 14:47 jdk-8u291-linux-x64.tar.gz
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 公共的
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 模板
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 视频
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 图片
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 文档
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 下载
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 音乐
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 桌面
dt@dt-virtual-machine:~$
```

图 1.5.13 将压缩包文件拷贝到 Ubuntu 系统

接着将 eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz 压缩包文件解压到/opt 目录下, 如下所示:

```
sudo tar -xzf eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz -C /opt/
```

```
dt@dt-virtual-machine:~$ 
dt@dt-virtual-machine:~$ sudo tar -xzf eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz -C /opt/
[sudo] dt 的密码:
dt@dt-virtual-machine:~$
```

图 1.5.14 解压 eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz 文件

执行解压命令的时候, 需要加上 sudo, 这样才能有权限将压缩文件解压到/opt 目录下, 解压之后会在 /opt 目录下生成 eclipse 文件夹, 如下所示:

```
dt@dt-virtual-machine:~$ 
dt@dt-virtual-machine:~$ cd /opt/
dt@dt-virtual-machine:/opt$ 
dt@dt-virtual-machine:/opt$ ls
eclipse
dt@dt-virtual-machine:/opt$ 
dt@dt-virtual-machine:/opt$
```

图 1.5.15 eclipse 文件夹

这样 Eclipse 就安装完成了, 我们下载的这个压缩包是一个免安装的, 直接解压就可以了, eclipse 目录就是软件的安装目录了, 如下所示:

```
dt@dt-virtual-machine:/opt$ 
dt@dt-virtual-machine:/opt$ cd eclipse/
dt@dt-virtual-machine:/opt/eclipse$ ls
artifacts.xml configuration dropins eclipse eclipse.ini features icon.xpm notice.html p2 plugins readme
dt@dt-virtual-machine:/opt/eclipse$ 
dt@dt-virtual-machine:/opt/eclipse$ 
dt@dt-virtual-machine:/opt/eclipse$
```

图 1.5.16 Eclipse 安装目录

接下来创建一个桌面快捷方式, 可以通过双击桌面图标来打开 Eclipse 软件。首先进入到桌面目录下, 如下所示:

```
dt@dt-virtual-machine:~$ ls -l
总用量 508404
-rw-r--r-- 1 dt dt 375618953 6月 3 14:46 eclipse-cpp-2020-12-R-linux-gtk-x86_64.tar.gz
-rw-r--r-- 1 dt dt 8980 6月 3 13:02 examples.desktop
-rw-r--r-- 1 dt dt 144935989 6月 3 14:47 jdk-8u291-linux-x64.tar.gz
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 公共的
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 模板
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 视频
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 图片
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 文档
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 下载
drwxr-xr-x 2 dt dt 4096 6月 3 13:14 音乐
drwxr-xr-x 2 dt dt 4096 6月 3 15:51 桌面
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd ./桌面/
dt@dt-virtual-machine:~/桌面$
```

图 1.5.17 进入桌面目录

笔者的 Ubuntu 系统配置的是中文语言环境，如果你的英文环境，那么对应的应该是 Desktop 目录。在该目录下创建一个名为 eclipse.desktop 的文件，将以下内容写入到该文件中：

```
[Desktop Entry]
Encoding=UTF-8
Name=Eclipse
Comment=Eclipse
Exec=/opt/eclipse/eclipse
Icon=/opt/eclipse/icon.xpm
Terminal=false
StartupNotify=true
Type=Application
Categories=Application;Development;
```

```
[Desktop Entry]
Encoding=UTF-8
Name=Eclipse
Comment=Eclipse
Exec=/opt/eclipse/eclipse
Icon=/opt/eclipse/icon.xpm
Terminal=false
StartupNotify=true
Type=Application
Categories=Application;Development;
~
~
```

图 1.5.18 eclipse.desktop 文件中的内容

其中“Exec=”后面为 eclipse 安装目录下的 eclipse 可执行文件的路径，“Icon=”后面为 eclipse 安装目录下的图标图片的路径，编辑完成之后保存退出。最后为 eclipse.desktop 文件添加上可执行权限：

```
chmod u+x eclipse.desktop
```

完成之后，Ubuntu 系统桌面上会出现 Eclipse 软件图标快捷方式：

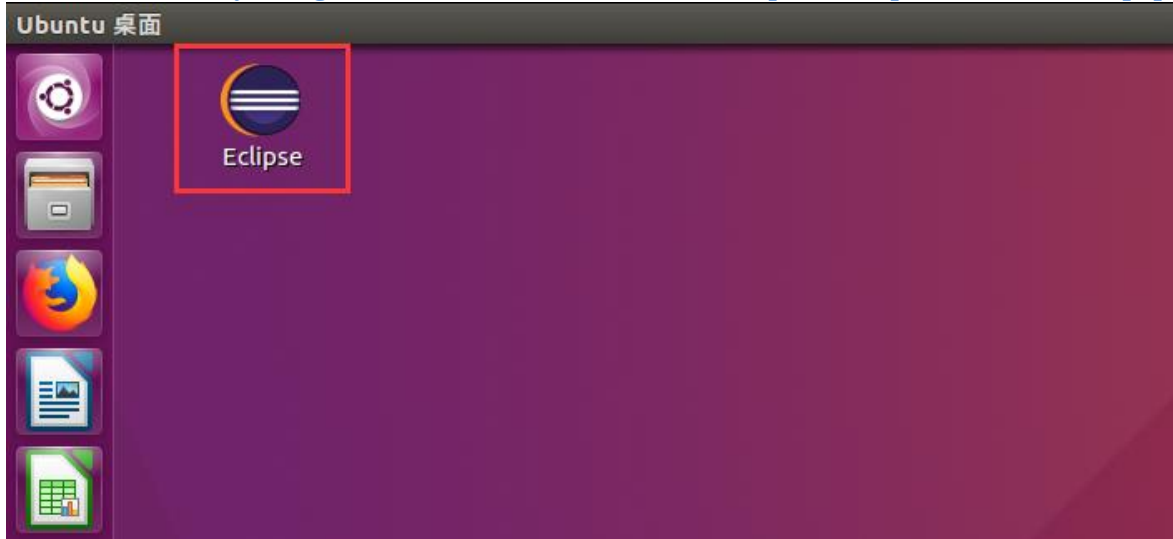


图 1.5.19 桌面图标

四、安装 jdk

将 jdk 压缩包 `jdk-8u291-linux-x64.tar.gz` 解压到 `/opt` 目录下:

```
sudo tar -xzf jdk-8u291-linux-x64.tar.gz -C /opt/
```

```
dt@dt-virtual-machine:~$  
dt@dt-virtual-machine:~$ sudo tar -xzf jdk-8u291-linux-x64.tar.gz -C /opt/  
[sudo] dt 的密码:  
dt@dt-virtual-machine:~$  
dt@dt-virtual-machine:~$
```

图 1.5.20 解压 `jdk-8u291-linux-x64.tar.gz` 文件

解压完成之后会在 `/opt` 目录下生成 `jdk1.8.0_291` 文件夹, 如下所示:

```
dt@dt-virtual-machine:/opt$  
dt@dt-virtual-machine:/opt$ ls  
eclipse  jdk1.8.0_291  
dt@dt-virtual-machine:/opt$  
dt@dt-virtual-machine:/opt$
```

图 1.5.21 `jdk1.8.0_291` 文件夹

接下来需要配置环境变量, 打开用户家目录下的 `.bashrc` (`vi ~/.bashrc`) 文件, 在该文件末尾添加如下内容:

```
# jdk 环境配置  
export JDK_HOME=/opt/jdk1.8.0_291  
export JRE_HOME=${JDK_HOME}/jre  
export CLASSPATH=.:${JDK_HOME}/lib:${JRE_HOME}/lib:$CLASSPATH  
export PATH=${JDK_HOME}/bin:$PATH
```

```

108 # enable programmable completion features (you don't need to enable
109 # this, if it's already enabled in /etc/bash.bashrc and /etc/profile
110 # sources /etc/bash.bashrc).
111 if ! shopt -oq posix; then
112   if [ -f /usr/share/bash-completion/bash_completion ]; then
113     . /usr/share/bash-completion/bash_completion
114   elif [ -f /etc/bash_completion ]; then
115     . /etc/bash_completion
116   fi
117 fi
118
119 # jdk环境配置
120 export JDK_HOME=/opt/jdk1.8.0_291
121 export JRE_HOME=${JDK_HOME}/jre
122 export CLASSPATH=.:${JDK_HOME}/lib:${JRE_HOME}/lib:$CLASSPATH
123 export PATH=${JDK_HOME}/bin:$PATH
124

```

图 1.5.22 配置.bashrc 文件

环境变量 JDK_HOME 指定了 jdk 的路径。文件内容编辑完成之后保存退出，接着使用 source 命令运行.bashrc 脚本使我们的配置生效：

```
source ~/.bashrc
```

jdk 到这里就安装完成了，可以检查看看 jdk 有没有安装成功，在终端下执行 java -version 命令：

```

dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ java -version
java version "1.8.0_291"
Java(TM) SE Runtime Environment (build 1.8.0_291-b10)
Java HotSpot(TM) 64-Bit Server VM (build 25.291-b10, mixed mode)
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$

```

图 1.5.23 验证 jdk 安装是否成功

还有最好一个步骤需要完成，将 eclipse 与 jdk 联系起来，进入到 Eclipse 安装目录下，在该目录下创建一个名为 jre 的文件夹：

```

dt@dt-virtual-machine:~$ cd /opt/eclipse/
dt@dt-virtual-machine:/opt/eclipse$
dt@dt-virtual-machine:/opt/eclipse$ sudo mkdir jre
[sudo] dt 的密码:
dt@dt-virtual-machine:/opt/eclipse$
dt@dt-virtual-machine:/opt/eclipse$ ls
artifacts.xml configuration dropins eclipse eclipse.ini features icon.xpm jre notice.html p2 plugins readme
dt@dt-virtual-machine:/opt/eclipse$
dt@dt-virtual-machine:/opt/eclipse$

```

图 1.5.24 创建 jre 目录

进入到 jre 目录下，创建一个软链接文件指向 jdk 目录下的 bin 目录，如下所示：

```

dt@dt-virtual-machine:/opt/eclipse$ cd jre/
dt@dt-virtual-machine:/opt/eclipse/jre$
dt@dt-virtual-machine:/opt/eclipse/jre$ sudo ln -s /opt/jdk1.8.0_291/bin
dt@dt-virtual-machine:/opt/eclipse/jre$
dt@dt-virtual-machine:/opt/eclipse/jre$ ls
bin
dt@dt-virtual-machine:/opt/eclipse/jre$ ls -l
总用量 0
lrwxrwxrwx 1 root root 21 6月  3 17:39 bin -> /opt/jdk1.8.0_291/bin
dt@dt-virtual-machine:/opt/eclipse/jre$
dt@dt-virtual-machine:/opt/eclipse/jre$

```

图 1.5.25 创建软链接文件

五、测试

接下来打开 Eclipse 软件，创建一个简单的 C 语言工程进行测试，在 Ubuntu 系统桌面上双击 Eclipse 图标打开 Eclipse：

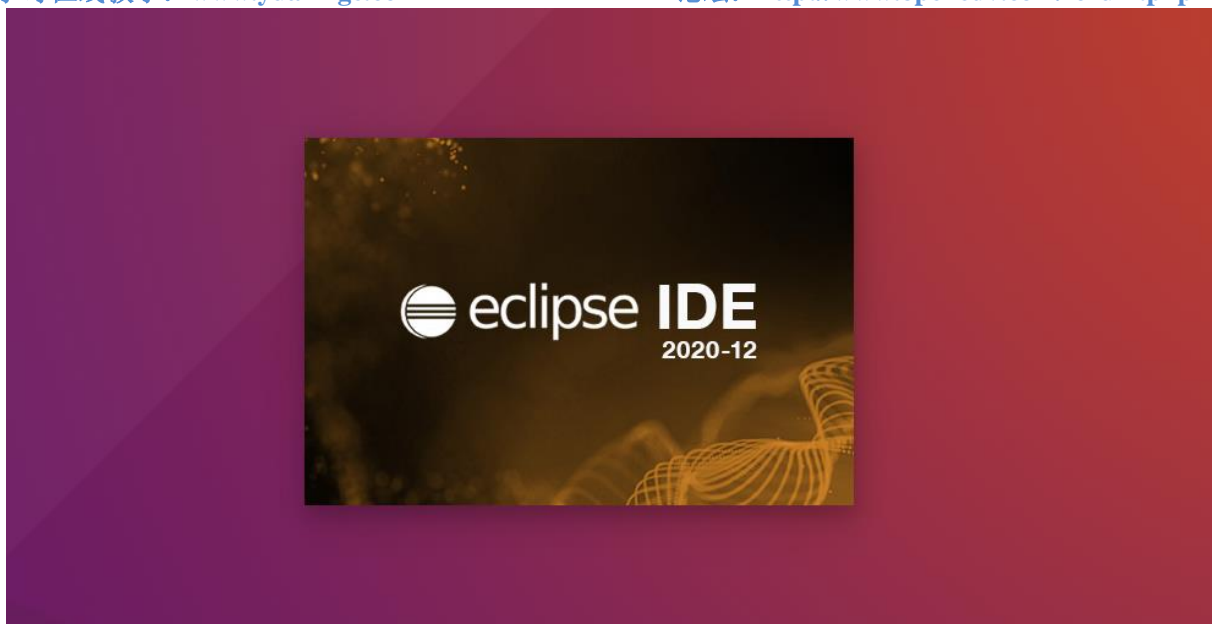


图 1.5.26 启动界面

Eclipse 启动之后会首先让用户设置一个工作目录，如下所示：

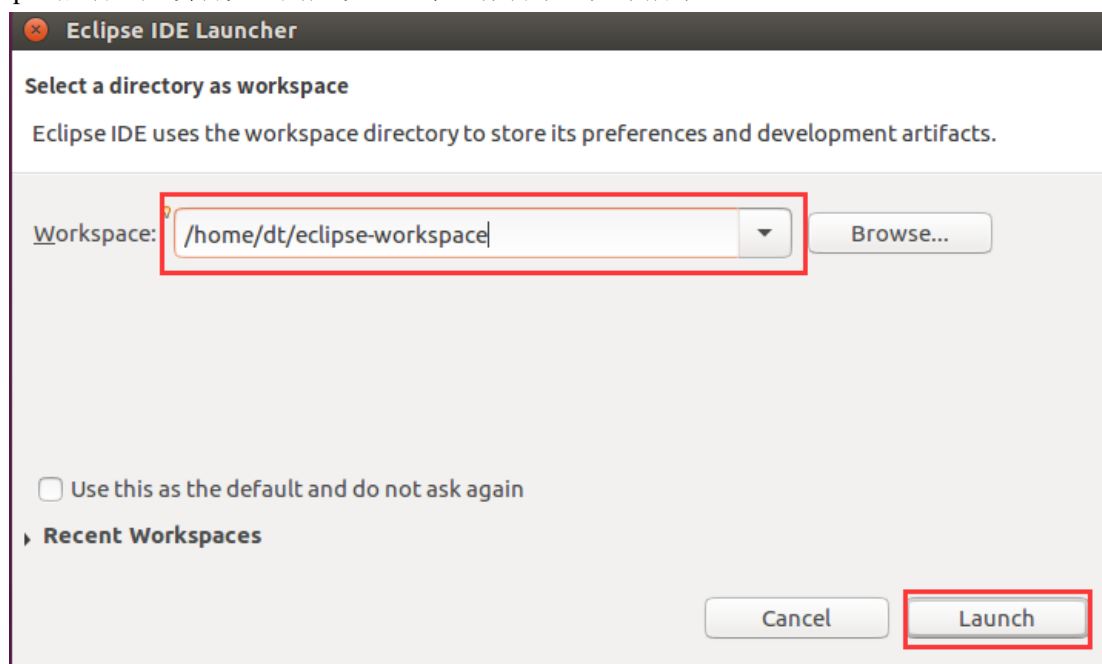


图 1.5.27 设置工作目录

直接使用默认的路径即可，点击 Launch。

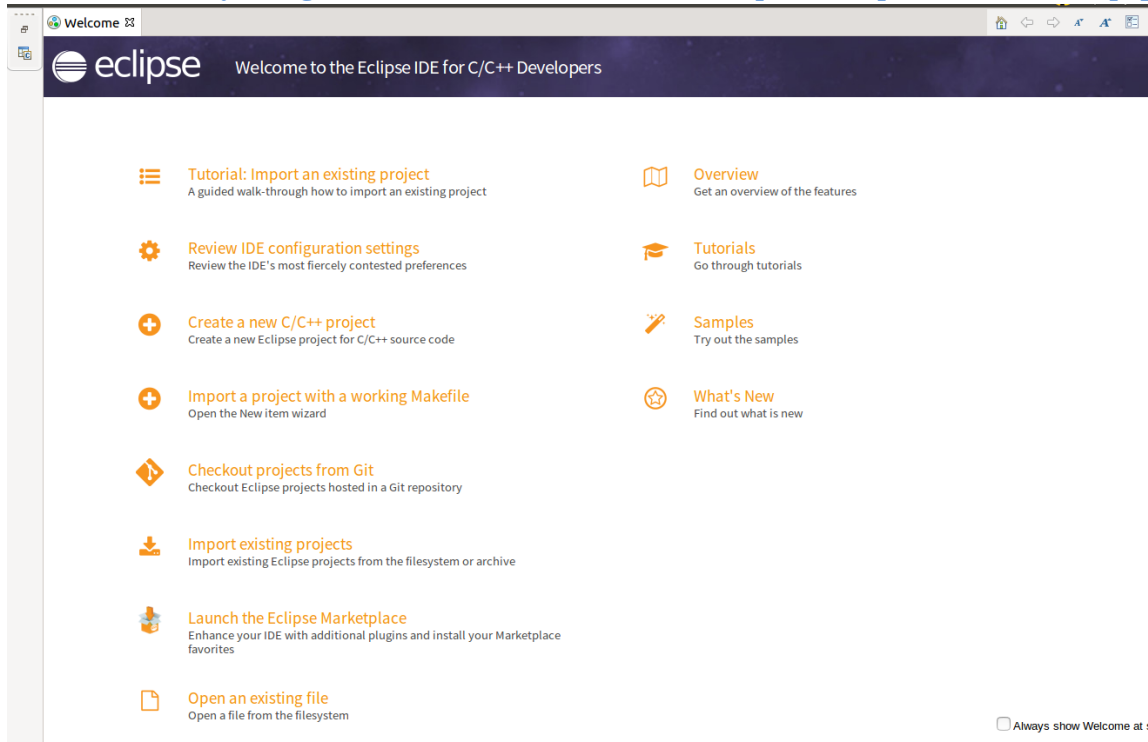


图 1.5.28 软件的欢迎界面

点击“Create a new C/C++ project”可以创建一个 C 语言应用工程，具体的步骤的就不再演示了。本小节内容到此结束。

第二章 文件 I/O 基础

本章给大家介绍 Linux 应用编程中最基础的知识,即文件 I/O (Input、Outout),文件 I/O 指的是对文件的输入/输出操作,说白了就是对文件的读写操作;Linux 下一切皆文件,文件作为 Linux 系统设计思想核心理念,在 Linux 系统下显得尤为重要,所以对文件的 I/O 操作既是基础也是最重要的部分。

本章将向大家介绍 Linux 系统下文件描述符的概念,随后会逐一讲解构成通用 I/O 模型的系统调用,譬如打开文件、关闭文件、从文件中读取数据和向文件中写入数据以及这些系统调用涉及的参数等内容。

本章将会讨论如下主题内容。

- 文件描述符的概念;
- 打开文件 `open()`、关闭文件 `close()`;
- 写文件 `write()`、读文件 `read()`;
- 文件读写位置偏移量。

2.1 一个简单的文件 IO 示例

本章主要介绍文件 IO 操作相关系统调用, 一个通用的 IO 模型通常包括打开文件、读写文件、关闭文件这些基本操作, 主要涉及到 4 个函数: `open()`、`read()`、`write()` 以及 `close()`, 我们先来看一个简单地文件读写示例, 应用程序代码如下所示:

示例代码 2.1.1 一个简单地文件 IO 示例

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    char buff[1024];
    int fd1, fd2;
    int ret;

    /* 打开源文件 src_file(只读方式) */
    fd1 = open("./src_file", O_RDONLY);
    if (-1 == fd1)
        return fd1;

    /* 打开目标文件 dest_file(只写方式) */
    fd2 = open("./dest_file", O_WRONLY);
    if (-1 == fd2) {
        ret = fd2;
        goto out1;
    }

    /* 读取源文件 1KB 数据到 buff 中 */
    ret = read(fd1, buff, sizeof(buff));
    if (-1 == ret)
        goto out2;

    /* 将 buff 中的数据写入目标文件 */
    ret = write(fd2, buff, sizeof(buff));
    if (-1 == ret)
        goto out2;

    ret = 0;

out2:
    /* 关闭目标文件 */
```

```
close(fd2);

out1:
    /* 关闭源文件 */
    close(fd1);
    return ret;
}
```

这段代码非常简单明了, 代码所要实现的功能在注释当中已经描述得很清楚了, 从源文件 `src_file` 中读取 1KB 数据, 然后将其写入到目标文件 `dest_file` 中 (这里假设当前目录下这两个文件都是存在的); 在进行读写操作之前, 首先调用 `open` 函数将源文件和目标文件打开, 成功打开之后再调用 `read` 函数从源文件中读取 1KB 数据, 然后再调用 `write` 函数将这 1KB 数据写入到目标文件中, 至此, 文件读写操作就完成了, 读写操作完成之后, 最后调用 `close` 函数关闭源文件和目标文件。

接下来我们给大家详细介绍这些函数以及相关的内容。

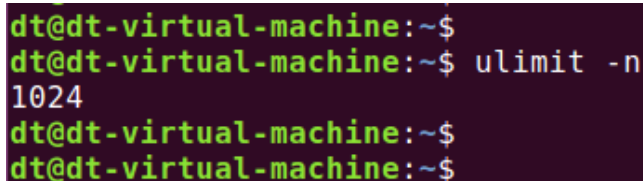
2.2 文件描述符

调用 `open` 函数会有一个返回值, 譬如示例代码 2.1.1 中的 `fd1` 和 `fd2`, 这是一个 `int` 类型的数据, 在 `open` 函数执行成功的情况下, 会返回一个非负整数, 该返回值就是一个文件描述符 (file descriptor), 这说明文件描述符是一个非负整数; 对于 Linux 内核而言, 所有打开的文件都会通过文件描述符进行索引。

当调用 `open` 函数打开一个现有文件或创建一个新文件时, 内核会向进程返回一个文件描述符, 用于指代被打开的文件, 所有执行 IO 操作的系统调用都是通过文件描述符来索引到对应的文件, 譬如示例代码 2.1.1 中, 当调用 `read/write` 函数进行文件读写时, 会将文件描述符传送给 `read/write` 函数, 所以在代码中, `fd1` 就是源文件 `src_file` 被打开时所对应的文件描述符, 而 `fd2` 则是目标文件 `dest_file` 被打开时所对应的文件描述符。

一个进程可以打开多个文件, 但是在 Linux 系统中, 一个进程可以打开的文件数是有限制, 并不是可以无限制打开很多的文件, 大家想一想便可以知道, 打开的文件是需要占用内存资源的, 文件越大、打开的文件越多那占用的内存就越多, 必然会对整个系统造成很大的影响, 如果超过进程可打开的最大文件数限制, 内核将会发送警告信号给对应的进程, 然后结束进程; 在 Linux 系统下, 我们可以通过 `ulimit` 命令来查看进程可打开的最大文件数, 用法如下所示:

```
ulimit -n
```



```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ulimit -n
1024
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 2.2.1 查看进程可打开的最大文件数

该最大值默认情况下是 1024, 也就意味着一个进程最多可以打开 1024 个文件, 当然这个限制数其实是可以设置的, 这个就先不给大家介绍了, 当然除了进程有最大文件数限制外, 其实对于整个 Linux 系统来说, 也有最大限制, 那么关于这些问题, 如果后面的章节内容中涉及到了再给大家进行介绍。

所以对于一个进程来说, 文件描述符是一种有限资源, 文件描述符是从 0 开始分配的, 譬如说进程中第一个被打开的文件对应的文件描述符是 0、第二个文件是 1、第三个文件是 2、第 4 个文件是 3.....以此类推, 所以由此可知, 文件描述符数字最大值为 1023 (0~1023)。每一个被打开的文件在同一个进程中都有一个唯一的文件描述符, 不会重复, 如果文件被关闭后, 它对应的文件描述符将会被释放, 那么这个文件描述符将可以再次分配给其它打开的文件、与对应的文件绑定起来。

每次给打开的文件分配文件描述符都是从最小的没有被使用的文件描述符(0~1023)开始,当之前打开的文件被关闭之后,那么它对应的文件描述符会被释放,释放之后也就成为了一个没有被使用的文件描述符了。

当我们在程序中,调用 `open` 函数打开文件的时候,分配的文件描述符一般都是从 3 开始,这里大家可能要问了,上面不是从 0 开始的吗,确实是如此,但是 0、1、2 这三个文件描述符已经默认被系统占用了,分别分配给了系统标准输入(0)、标准输出(1)以及标准错误(2),关于这个问题,这里不便给大家说太多,毕竟这是后面的内容,这里只是给大家提一下,后面遇到了再具体讲解。

Tips: Linux 系统下,一切皆文件,也包括各种硬件设备,使用 `open` 函数打开任何文件成功情况下便会返回对应的文件描述符 `fd`。每一个硬件设备都会对应于 Linux 系统下的某一个文件,把这类文件称为设备文件。所以设备文件对应的其实是某一硬件设备,应用程序通过对设备文件进行读写等操作、来使用、操控硬件设备,譬如 LCD 显示器、串口、音频、键盘等。

标准输入一般对应的是键盘,可以理解为 0 便是打开键盘对应的设备文件时所得到的文件描述符;标准输出一般指的是 LCD 显示器,可以理解为 1 便是打开 LCD 设备对应的设备文件时所得到的文件描述符;而标准错误一般指的也是 LCD 显示器。

2.3 open 打开文件

在 Linux 系统中要操作一个文件,需要先打开该文件,得到文件描述符,然后再对文件进行相应的读写操作(或其他操作),最后在关闭该文件;`open` 函数用于打开文件,当然除了打开已经存在的文件之外,还可以创建一个新的文件,函数原型如下所示:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

在 Linux 系统下,可以通过 `man` 命令(也叫 `man` 手册)来查看某一个 Linux 系统调用的帮助信息,`man` 命令可以将该系统调用的详细信息显示出来,譬如函数功能介绍、函数原型、参数、返回值以及使用该函数所需包含的头文件等信息;`man` 更像是一份帮助手册,所以也把它称为 `man` 手册,当我们需要查看某个系统调用的功能介绍、使用方法时,不用在网上到处查找,直接通过 `man` 命令便可以搞定,`man` 命令用法如下所示:

```
man 2 open      #查看 open 函数的帮助信息
```



```
OPEN(2) Linux Programmer's Manual
NAME
  open, openat, creat - open and possibly create a file
SYNOPSIS
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>

  int open(const char *pathname, int flags);
  int open(const char *pathname, int flags, mode_t mode);

  int creat(const char *pathname, mode_t mode);

  int openat(int dirfd, const char *pathname, int flags);
  int openat(int dirfd, const char *pathname, int flags, mode_t mode);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
```

图 2.3.1 查看 `open` 函数帮助信息

Tips: man 命令后面跟着两个参数, 数字 2 表示系统调用, man 命令除了可以查看系统调用的帮助信息外, 还可以查看 Linux 命令 (对应数字 1) 以及标准 C 库函数 (对应数字 3) 所对应的帮助信息; 最后一个参数 open 表示需要查看的系统调用函数名。

由于篇幅有限, 此截图只是其中一部分内容, 从图中可知, open 函数有两种原型? 这是为什么呢? 关于这个问题笔者一开始也不理解, 大家都知道 C 语言是不支持重载的, 那既然这样, 只有一种解释了, 那就是可变参函数; 对于 C 语言中的可变参函数, 对此不了解的朋友可以自行百度, 本文档不作说明!

所以由此可知, 在应用程序中调用 open 函数即可传入 2 个参数 (pathname、flags)、也可传入 3 个参数 (pathname、flags、mode), 但是第三个参数 mode 需要在第二个参数 flags 满足条件时才会有效, 稍后将对此进行说明; 从图 2.3.1 可知, 在应用程序中使用 open 函数时, 需要包含 3 个头文件 “#include <sys/types.h>”、“#include <sys/stat.h>”、“#include <fcntl.h>”。

函数参数和返回值含义如下:

pathname: 字符串类型, 用于标识需要打开或创建的文件, 可以包含路径 (绝对路径或相对路径) 信息, 譬如: “./src_file” (当前目录下的 src_file 文件)、“/home/dengtao/hello.c”等; 如果 pathname 是一个符号链接, 会对其进行解引用。

flags: 调用 open 函数时需要提供的标志, 包括文件访问模式标志以及其它文件相关标志, 这些标志使用宏定义进行描述, 都是常量, open 函数提供了非常多的标志, 我们传入 flags 参数时既可以单独使用某一个标志, 也可以通过位或运算 (|) 将多个标志进行组合。这些标志介绍如下:

表 2.3.1 open 函数 flags 参数值介绍

标志	用途	说明
O_RDONLY	以只读方式打开文件	这三个是文件访问权限标志, 传入的 flags 参数中必须要包含其中一种标志, 而且只能包含一种, 打开的文件只能按照这种权限来操作, 譬如使用了 O_RDONLY 标志, 就只能对文件进行读取操作, 不能写操作。
O_WRONLY	以只写方式打开文件	
O_RDWR	以可读可写方式打开文件	
O_CREAT	如果 pathname 参数指向的文件不存在则创建此文件	使用此标志时, 调用 open 函数需要传入第 3 个参数 mode, 参数 mode 用于指定新建文件的访问权限, 稍后将对此进行说明。 open 函数的第 3 个参数只有在使用了 O_CREAT 或 O_TMPFILE 标志时才有效。
O_DIRECTORY	如果 pathname 参数指向的不是一个目录, 则调用 open 失败	
O_EXCL	此标志一般结合 O_CREAT 标志一起使用, 用于专门创建文件。 在 flags 参数同时使用到了 O_CREAT 和 O_EXCL 标志的情况下, 如果 pathname 参数指向的文件已经存在, 则 open 函数返回错误。	可以用于测试一个文件是否存在, 如果不存在则创建此文件, 如果存在则返回错误, 这使得测试和创建两者成为一个原子操作; 关于原子操作, 在后面的内容当中将会对此进行说明。

O_NOFOLLOW	如果 <code>pathname</code> 参数指向的是一个符号链接, 将不对其进行解引用, 直接返回错误。	不加此标志情况下, 如果 <code>pathname</code> 参数是一个符号链接, 会对其进行解引用。
------------	---	---

以上给大家介绍了一些比较常用的标志, `open` 函数的 `flags` 标志并不止这些, 还有很多标志这里并没有给大家进行介绍, 譬如 `O_APPEND`、`O_ASYNC`、`O_DSYNC`、`O_NOATIME`、`O_NONBLOCK`、`O_SYNC` 以及 `O_TRUNC` 等, 对于这些没有提及到的标志, 在后面学习过程中, 也会给大家慢慢介绍。对于初学者来说, 我们需要把表 2.3.1 中所列出的这些标志给弄明白、理解它们的作用和含义。

Tips: 不同内核版本所支持的 `flags` 标志是存在差别的, 譬如说新版本内核所支持的标志可能在老版本是不支持的, 亦或者老版本支持的标志在新版本已经被取消、替代, `man` 手册中对一些标志是从哪个版本开始支持的有简单地说明, 读者可以自行阅读!

前面我们说过, `flags` 参数时既可以单独使用某一个标志, 也可以通过位或运算 (`|`) 将多个标志进行组合, 譬如:

```
open("./src_file", O_RDONLY)           //单独使用某一个标志
open("./src_file", O_RDONLY | O_NOFOLLOW) //多个标志组合
```

mode: 此参数用于指定新建文件的访问权限, 只有当 `flags` 参数中包含 `O_CREAT` 或 `O_TMPFILE` 标志时才有效(`O_TMPFILE` 标志用于创建一个临时文件)。权限对于文件来说是一个很重要的属性, 那么在 Linux 系统中, 我们可以通过 `touch` 命令新建一个文件, 此时文件会有一个默认的权限, 如果需要修改文件权限, 可通过 `chmod` 命令对文件权限进行修改, 譬如在 Linux 系统下我们可以使用 "`ls -l`" 命令来查看到文件所对应的权限。

当我们调用 `open` 函数去新建一个文件时, 也需要指定该文件的权限, 而 `mode` 参数便用于指定此文件的权限, 接下来看看我们该如何通过 `mode` 参数来表示文件的权限, 首先 `mode` 参数的类型是 `mode_t`, 这是一个 `u32` 无符号整形数据, 权限表示方法如下所示:

```
0000 000 000 000 000
                         
      S   U   G   O
```

图 2.3.2 mode 权限表示方法

我们从低位从上看, 每 3 个 bit 位分为一组, 分别表示:

O---这 3 个 bit 位用于表示其他用户的权限;

G---这 3 个 bit 位用于表示同组用户 (group) 的权限, 即与文件所有者有相同组 ID 的所有用户;

U---这 3 个 bit 位用于表示文件所属用户的权限, 即文件或目录的所属者;

S---这 3 个 bit 位用于表示文件的特殊权限, 文件特殊权限一般用的比较少, 这里就不给大家细讲了。

关于什么是文件所属用户、同组用户以及其他用户, 这些都是 Linux 操作系统相关的基础知识, 相信大家理解这些概念; 3 个 bit 位中, 按照 `rwX` 顺序来分配权限位 (特殊权限除外), 最高位 (权值为 4) 表示读权限, 为 1 时表示具有读权限, 为 0 时没有读权限; 中间位 (权值为 2) 表示写权限, 为 1 时表示具有写权限, 为 0 时没有写权限; 最低位 (权值为 1) 表示执行权限, 为 1 时表示具有可执行权限, 为 0 时没有执行权限。接下来我们举几个例子 (特殊权限这里暂时不管, 其 `S` 字段全部为 0):

最高权限表示方法: 11111111 (二进制表示)、777 (八进制表示)、511 (十进制表示);

最高权限这里意味着所有用户对此文件都具有读权限、写权限以及执行权限。

11100000 (二进制表示): 表示文件所属者具有读、写、执行权限, 而同组用户和其他用户不具有任何权限;

100100100 (二进制表示): 表示文件所属者、同组用户以及其他用户都具有读权限, 但都没有写、执行权限。

Tips: open 函数 O_RDONLY、O_WRONLY 以及 O_RDWR 这三个标志表示以什么方式去打开文件, 譬如以只写方式打开 (open 函数得到的文件描述符只能对文件进行写操作, 不能读)、以只读方式打开 (open 函数得到的文件描述符只能对文件进行读操作, 不能写)、以可读可写方式打开 (open 函数得到的文件描述符可对文件进行读和写操作); 与文件权限之间的联系, 只有用户对该文件具有相应权限时, 才可以使用对应的标志去打开文件, 否则会打开失败! 譬如, 我们的程序对该文件只有只读权限, 那么执行 open 函数使用 O_RDWR 或 O_WRONLY 标志将会失败。关于文件权限等相关问题, 将会在 4.1 中给大家介绍。

关于文件权限表示方法的问题, 以上就给大家介绍这么多, 在实际编程中, 我们可以直接使用 Linux 中已经定义好的宏, 不同的宏定义表示不同的权限, 如下所示:

表 2.3.2 open 函数文件权限宏

宏定义	说明
S_IRUSR	允许文件所有者读文件
S_IWUSR	允许文件所有者写文件
S_IXUSR	允许文件所有者执行文件
S_IRWXU	允许文件所有者读、写、执行文件
S_IRGRP	允许同组用户读文件
S_IWGRP	允许同组用户写文件
S_IXGRP	允许同组用户执行文件
S_IRWXG	允许同组用户读、写、执行文件
S_IROTH	允许其他用户读文件
S_IWOTH	允许其他用户写文件
S_IXOTH	允许其他用户执行文件
S_IRWXO	允许其他用户读、写、执行文件
S_ISUID	set-user-ID (特殊权限)
S_ISGID	set-group-ID (特殊权限)
S_ISVTX	sticky (特殊权限)

这些宏既可以单独使用, 也可以通过位或运算将多个宏组合在一起, 譬如:

S_IRUSR | S_IWUSR | S_IROTH

返回值: 成功将返回文件描述符, 文件描述符是一个非负整数; 失败将返回-1。

以上就把 open 函数相关的基础知识给大家介绍完了, 包括函数返回值、参数等信息, 当然在后面的章节内容中, 我们还会更加深入地给大家讲解 open 函数相关的知识点; 接下来我们看一些 open 函数的简答使用示例。

open 函数使用示例

(1)使用 open 函数打开一个已经存在的文件 (例如当前目录下的 app.c 文件), 使用只读方式打开:

```
int fd = open("./app.c", O_RDONLY)
if (-1 == fd)
    return fd;
```

(2)使用 open 函数打开一个已经存在的文件 (例如当前目录下的 app.c 文件), 使用可读可写方式打开:

```
int fd = open("./app.c", O_RDWR)
if (-1 == fd)
    return fd;
```

(3)使用 open 函数打开一个指定的文件 (譬如/home/dengtao/hello), 使用可读可写方式, 如果该文件是一个符号链接文件, 则不对其进行解引用, 直接返回错误:

```
int fd = open("/home/dengtao/hello", O_RDWR | O_NOFOLLOW);
if (-1 == fd)
    return fd;
```

(4)使用 `open` 函数打开一个指定的文件（譬如/home/dengtao/hello），如果该文件不存在则创建该文件，创建该文件时，将文件权限设置如下：

文件所有者拥有读、写、执行权限；
同组用户与其他用户只有读权限。
使用可读可写方式打开：

```
int fd = open("/home/dengtao/hello", O_RDWR | O_CREAT, S_IRWXU | S_IRGRP | S_IROTH);
if (-1 == fd)
    return fd;
```

2.4 write 写文件

调用 `write` 函数可向打开的文件写入数据，其函数原型如下所示（可通过“man 2 write”查看）：

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

首先使用 `write` 函数需要先包含 `unistd.h` 头文件。

函数参数和返回值含义如下：

fd: 文件描述符。关于文件描述符，前面已经给大家进行了简单地讲解，这里不再重述！我们需要将进行写操作的文件所对应的文件描述符传递给 `write` 函数。

buf: 指定写入数据对应的缓冲区。

count: 指定写入的字节数。

返回值: 如果成功将返回写入的字节数（0 表示未写入任何字节），如果此数字小于 `count` 参数，这不是错误，譬如磁盘空间已满，可能会发生这种情况；如果写入出错，则返回-1。

对于普通文件（我们一般操作的大部分文件都是普通文件，譬如常见的文本文件、二进制文件等），不管是读操作还是写操作，一个很重要的问题是：从文件的哪个位置开始进行读写操作？也就是 IO 操作所对应的位置偏移量，读写操作都是从文件的当前位置偏移量处开始，当然当前位置偏移量可以通过 `lseek` 系统调用进行设置，关于此函数后面再讲；默认情况下当前位置偏移量一般是 0，也就是指向了文件起始位置，当调用 `read`、`write` 函数读写操作完成之后，当前位置偏移量也会向后移动对应字节数，譬如当前位置偏移量为 1000 个字节处，调用 `write()` 写入或 `read()` 读取 500 个字节之后，当前位置偏移量将会移动到 1500 个字节处。

2.5 read 读文件

调用 `read` 函数可从打开的文件中读取数据，其函数原型如下所示（可通过“man 2 read”查看）：

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

首先使用 `read` 函数需要先包含 `unistd.h` 头文件。

函数参数和返回值含义如下：

fd: 文件描述符。与 `write` 函数的 `fd` 参数意义相同。

buf: 指定用于存储读取数据的缓冲区。

count: 指定需要读取的字节数。

返回值: 如果读取成功将返回读取到的字节数, 实际读取到的字节数可能会小于 count 参数指定的字节数, 也有可能为 0, 譬如进行读操作时, 当前文件位置偏移量已经到了文件末尾。实际读取到的字节数少于要求读取的字节数, 譬如在到达文件末尾之前有 30 个字节数据, 而要求读取 100 个字节, 则 read 读取成功只能返回 30; 而下一次再调用 read 读, 它将返回 0 (文件末尾)。

2.6 close 关闭文件

可调用 close 函数关闭一个已经打开的文件, 其函数原型如下所示 (可通过"man 2 close"查看):

```
#include <unistd.h>
```

```
int close(int fd);
```

首先使用 close 函数需要先包含 unistd.h 头文件, 当我们对文件进行 IO 操作完成之后, 后续不再对文件进行操作时, 需要将文件关闭。

函数参数和返回值含义如下:

fd: 文件描述符, 需要关闭的文件所对应的文件描述符。

返回值: 如果成功返回 0, 如果失败则返回-1。

除了使用 close 函数显式关闭文件之外, 在 Linux 系统中, 当一个进程终止时, 内核会自动关闭它打开的所有文件, 也就是说在我们的程序中打开了文件, 如果程序终止退出时没有关闭打开的文件, 那么内核会自动将程序中打开的文件关闭。很多程序都利用了这一功能而不显式地用 close 关闭打开的文件。

显式关闭不再需要的文件描述符往往是良好的编程习惯, 会使代码在后续修改时更具有可读性, 也更可靠, 进而言之, 文件描述符是有限资源, 当不再需要时必须将其释放、归还于系统。

2.7 lseek

对于每个打开的文件, 系统都会记录它的读写位置偏移量, 我们也把这个读写位置偏移量称为读写偏移量, 记录了文件当前的读写位置, 当调用 read()或 write()函数对文件进行读写操作时, 就会从当前读写位置偏移量开始进行数据读写。

读写偏移量用于指示 read()或 write()函数操作时文件的起始位置, 会以相对于文件头部的位置偏移量来表示, 文件第一个字节数据的位置偏移量为 0。

当打开文件时, 会将读写偏移量设置为指向文件开始位置处, 以后每次调用 read()、write()将自动对其进行调整, 以指向已读或已写数据后的下一字节, 因此, 连续的调用 read()和 write()函数将使得读写按顺序递增, 对文件进行操作。我们先来看看 lseek 函数的原型, 如下所示 (可通过"man 2 lseek"查看):

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int whence);
```

首先调用 lseek 函数需要包含<sys/types.h>和<unistd.h>两个头文件。

函数参数和返回值含义如下:

fd: 文件描述符。

offset: 偏移量, 以字节为单位。

whence: 用于定义参数 offset 偏移量对应的参考值, 该参数为下列其中一种 (宏定义):

- SEEK_SET: 读写偏移量将指向 offset 字节位置处 (从文件头部开始算);

- **SEEK_CUR**: 读写偏移量将指向当前位置偏移量 + offset 字节位置处, offset 可以为正、也可以为负, 如果是正数表示往后偏移, 如果是负数则表示往前偏移;
- **SEEK_END**: 读写偏移量将指向文件末尾 + offset 字节位置处, 同样 offset 可以为正、也可以为负, 如果是正数表示往后偏移、如果是负数则表示往前偏移。

返回值: 成功将返回从文件头部开始算起的位置偏移量 (字节为单位), 也就是当前的读写位置; 发生错误将返回-1。

使用示例:

(1)将读写位置移动到文件开头处:

```
off_t off = lseek(fd, 0, SEEK_SET);
if (-1 == off)
    return -1;
```

(2)将读写位置移动到文件末尾:

```
off_t off = lseek(fd, 0, SEEK_END);
if (-1 == off)
    return -1;
```

(3)将读写位置移动到偏移文件开头 100 个字节处:

```
off_t off = lseek(fd, 100, SEEK_SET);
if (-1 == off)
    return -1;
```

(4)获取当前读写位置偏移量:

```
off_t off = lseek(fd, 0, SEEK_CUR);
if (-1 == off)
    return -1;
```

函数执行成功将返回文件当前读写位置。

2.8 练习

本章给大家介绍了文件 IO 中的基础知识, 包括文件 IO 常用到的系统调用 `open()`、`read()`、`write()`、`close()` 以及 `lseek()`, 当然关于这些系统调用所涉及到的知识点并没有讲完, 本章所介绍的都是一些基础知识内容, 在后面的学习过程中, 我们会一一给大家深入讲解文件 IO 中的一些问题。

学完本章内容后, 我们来做一些简单地文件 IO 编程练习, 以巩固、提高大家所学知识内容, 以下笔者给大家准备了 4 个简单地实战编程例子, 大家可以尝试自己独立完成。

简单地编程实战例子

(1)打开一个已经存在的文件 (例如 `src_file`), 使用只读方式; 然后打开一个新建文件 (例如 `dest_file`), 使用只写方式, 新建文件的权限设置如下:

文件所有者拥有读、写、执行权限;
同组用户与其他用户只有读权限。

从 `src_file` 文件偏移头部 500 个字节位置开始读取 1Kbyte 字节数据, 然后将读取出来的数据写入到 `dest_file` 文件中, 从文件开头处开始写入, 1Kbyte 字节大小, 操作完成之后使用 `close` 显式关闭所有文件, 然后退出程序。

(2)通过 `open` 函数判断文件是否存在 (例如 `test_file`), 并将判断结果显示出来。

(3)新建一个文件 (例如 `new_file`), 新建文件的权限设置为:

文件所有者拥有读、写、执行权限;

同组用户与其他用户只有读权限。

使用只写方式打开文件, 将文件前 1Kbyte 字节数据填充为 0x00, 将下 1Kbyte 字节数据填充为 0xFF, 操作完成之后显式关闭文件, 退出程序。

(4) 打开一个已经存在的文件 (例如 test_file), 通过 lseek 函数计算该文件的大小, 并打印出来。

以上就是本章内容给大家整理的几个简单地编程实战例子, 大家可以根据本章所需知识内容将这几个例子独立完成, 在本文档配套的资料包中, 笔者会给出相应的示例代码, 建议大家先自己独立思考, 如果实在不知再参考笔者给出的示例代码。

vscode 编写代码

前面给大家介绍了 vscode, 它是一款非常好用的代码编辑器, 我们可以在 Ubuntu 系统下使用 vscode 软件进行的代码编写, 首先在 Ubuntu 系统下创建一个文件夹作为 vscode 的工作目录, 譬如笔者在家目录下创建了一个名为 vscode_ws 的目录, 将其作为 vscode 的工作目录, 如下所示:

```
dt@dt-virtual-machine:~$ cd ~/
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ mkdir vscode_ws
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd vscode_ws/
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ pwd
/home/dt/vscode_ws
dt@dt-virtual-machine:~/vscode_ws$
```

图 2.8.1 创建 vscode 工作目录

在 Ubuntu 系统下打开 vscode 软件, 如下所示:

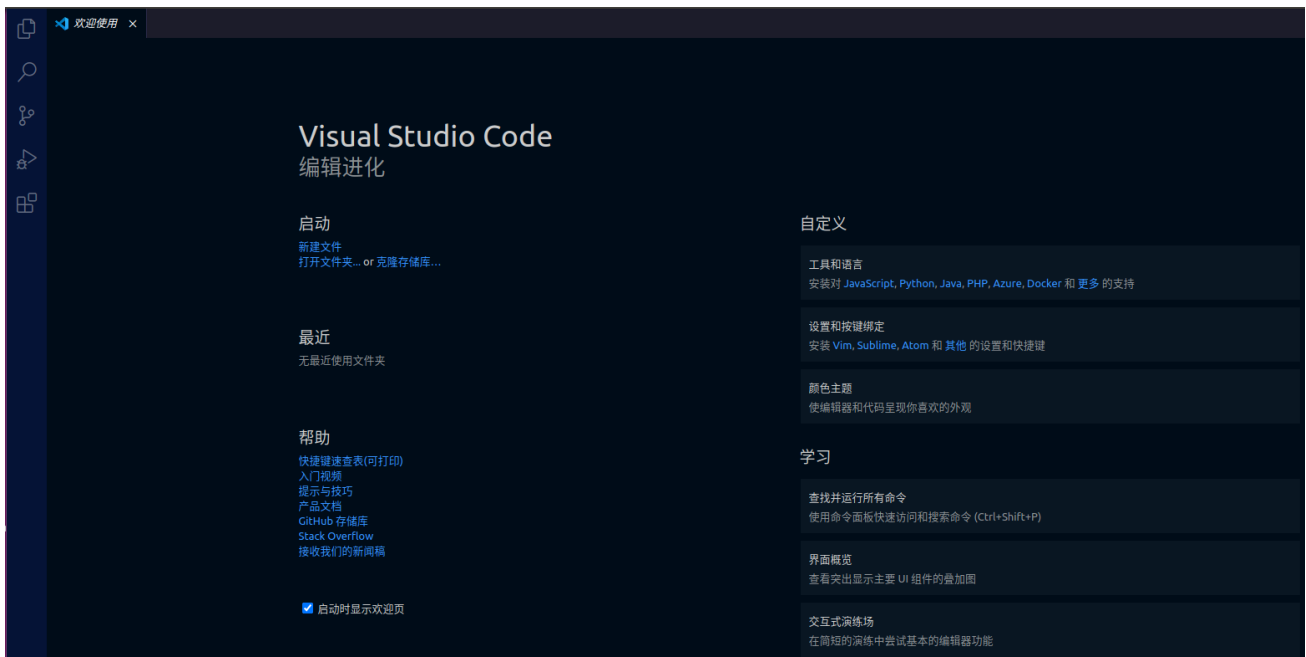


图 2.8.2 打开 vscode 软件

选择上边菜单栏, 点击“文件--->打开文件夹”, 在弹出来的页面中选择需要打开的文件夹, 这里我们选择前面创建好的 vscode_ws 目录, 如下所示:

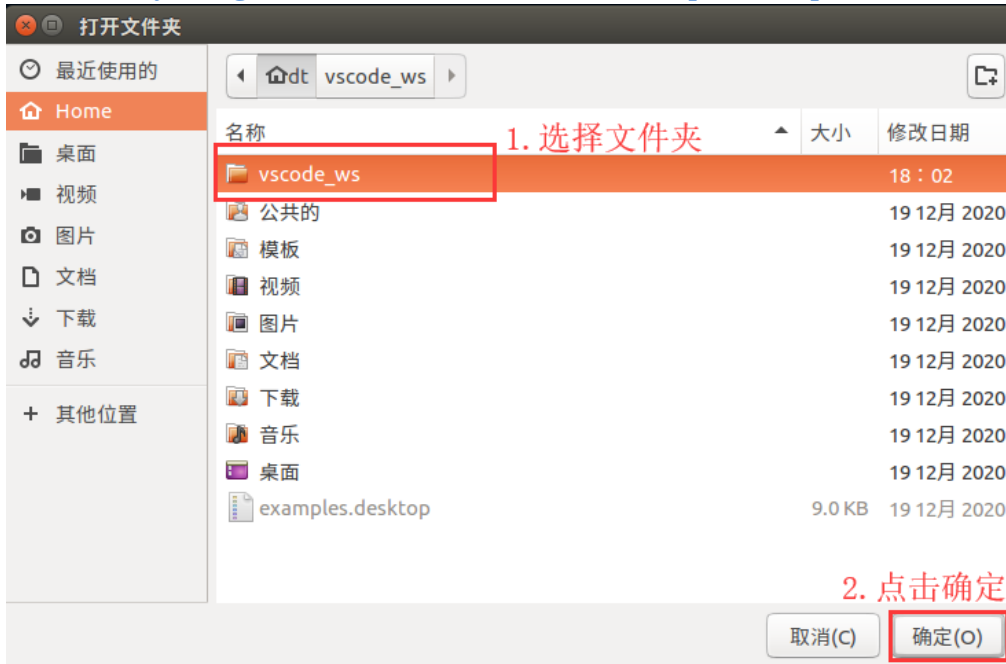


图 2.8.3 打开工作目录

接下来在工作目录下创建一个文件夹作为本章编程实战例程源码存放目录，直接在 vscode 软件进行创建即可，这里笔者将其命名为 1_chapter，如下所示：

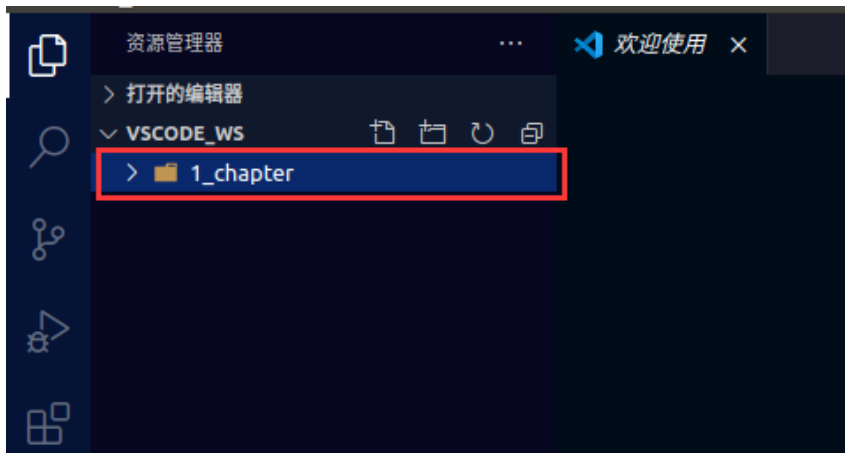


图 2.8.4 在工作目录下创建一个文件夹

接下来在 1_chapter 目录下创建一个.c 源文件，将其命名为 testApp_1.c，此源文件对应上面第一个(1)编程实战例子，如下所示：

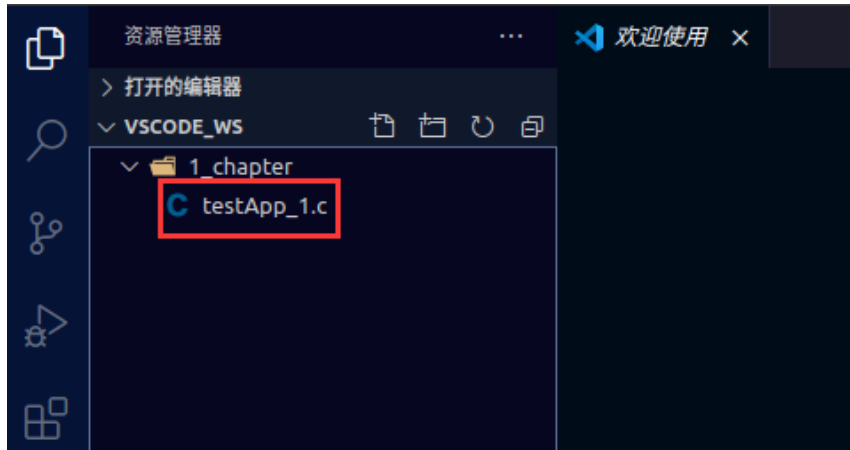


图 2.8.5 创建 C 程序源文件

点击打开 testApp_1.c 文本编辑页面, 接下来就可以在 testApp_1.c 源文件中编写第一个实战例子对应的源代码了, 以下是笔者给大家提供的一份示例代码, 大家可以进行参考, 但还是希望大家能够根据本章所学知识独立完成代码编写:

示例代码 2.8.1 编程实战例子 1

```
/******  
Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.  
文件名 : testApp_1.c  
作者 : 邓涛  
版本 : V1.0  
描述 :  
论坛 : www.openedv.com  
日志 : 初版 V1.0 2021/01/05 创建  
*****/  
  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <stdio.h>  
  
int main(void)  
{  
    char buffer[1024];  
    int fd1, fd2;  
    int ret;  
  
    /* 打开 src_file 文件 */  
    fd1 = open("./src_file", O_RDONLY);  
    if (-1 == fd1) {  
        printf("Error: open src_file failed!\n");  
        return -1;  
    }  
}
```

```
    }

    /* 新建 dest_file 文件并打开 */
    fd2 = open("./dest_file", O_WRONLY | O_CREAT | O_EXCL,
              S_IRWXU | S_IRGRP | S_IROTH);
    if (-1 == fd2) {
        printf("Error: open dest_file failed!\n");
        ret = -1;
        goto err1;
    }

    /* 将 src_file 文件读写位置移动到偏移文件头 500 个字节处 */
    ret = lseek(fd1, 500, SEEK_SET);
    if (-1 == ret)
        goto err2;

    /* 读取 src_file 文件数据, 大小 1KByte */
    ret = read(fd1, buffer, sizeof(buffer));
    if (-1 == ret) {
        printf("Error: read src_file failed!\n");
        goto err2;
    }

    /* 将 dest_file 文件读写位置移动到文件头 */
    ret = lseek(fd2, 0, SEEK_SET);
    if (-1 == ret)
        goto err2;

    /* 将 buffer 中的数据写入 dest_file 文件, 大小 1KByte */
    ret = write(fd2, buffer, sizeof(buffer));
    if (-1 == ret) {
        printf("Error: write dest_file failed!\n");
        goto err2;
    }

    printf("OK: test successful\n");
    ret = 0;

err2:
    close(fd2);

err1:
    close(fd1);
```



```
return ret;
}
```

此代码中用到了库函数 `printf`，是一个格式化输出函数，用于将格式化打印信息输出显示到屏幕上（终端），使用此函数需要包含标准 I/O 库头文件 `<stdio.h>`，对于有 C 语言编程经验的读者来说，如果对该函数的使用方法并不了解，可以查看 4.8.1 小节内容。

编译源码文件

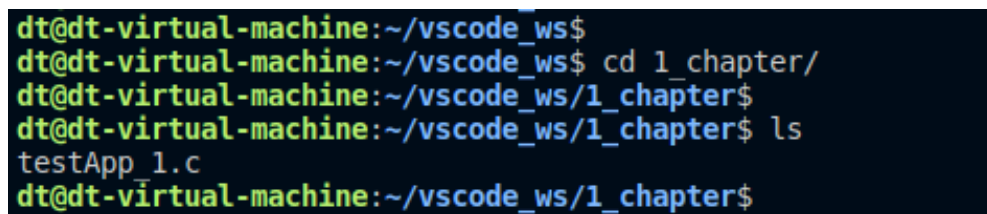
代码编写完成之后，接下来我们需要对源代码进行编译，在 `vscode` 中点击上边菜单“终端--->新终端”打开一个终端，打开的终端将会在下面显示出来，如下所示：



```
问题  输出  调试控制台  终端
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ ls
1 Chapter
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$
```

图 2.8.6 在 `vscode` 中打开终端

`vscode` 提供了终端功能，这样就可以不使用 `Ubuntu` 自带的终端了，非常的方便、不需要在 `Ubuntu` 终端和 `vscode` 软件之间进行切换，进入到 `1_chapter` 目录下，如下所示：



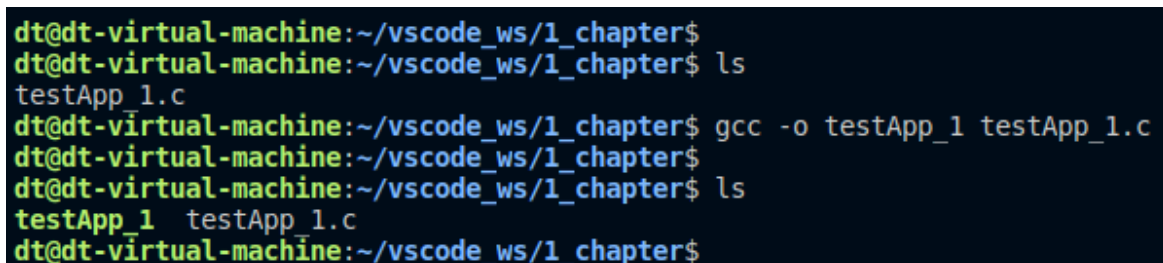
```
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ cd 1_chapter/
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls
testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 2.8.7 进入 `1_chapter` 目录

在该目录下我们直接使用 `Ubuntu` 系统提供的 `gcc` 编译器对源文件进行编译，编译生成一个可在 `Ubuntu` 系统下运行的可执行文件，执行如下命令：

```
gcc -o testApp_1 testApp_1.c
```

`gcc` 是 `Ubuntu` 系统下所使用的 C 语言编译器，`-o` 选项指定编译生成的可执行文件的名字，`testApp_1.c` 表示需要进行编译的 C 语言源文件，`gcc` 命令后面可以携带很多的编译相关的选项，这里不给大家介绍这个内容，这不是本文档我们需要去了解的重点，如果后面有机会可以给大家介绍下。编译完成之后会生成对应的可执行文件，如下所示：



```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls
testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ gcc -o testApp_1 testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls
testApp_1 testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 2.8.8 编译源文件

运行可执行文件测试

编译完成之后, 接下来可以运行测试了, 首先我们先准备一个文件 `src_file`, 这个是例子要求需要打开一个已经存在的文件, 文件大小大于或等于 1Kbyte, 这里笔者已经将 `src_file` 文件放置到 `1_chapter` 目录下了, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls -l
总用量 28
-rw-rw-r-- 1 dt dt 8920 1月  5 18:46 src_file
-rwxrwxr-x 1 dt dt 8920 1月  5 18:46 testApp_1
-rw-rw-r-- 1 dt dt 1583 1月  5 17:37 testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 2.8.9 准备好一个测试需要用到的文件

直接在当前目录下运行 `testApp_1` 可执行文件:

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ./testApp_1
OK: test successful
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls
dest_file src_file testApp_1 testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 2.8.10 运行 `testApp_1` 可执行文件

运行成功之后会生成 `dest_file` 文件, 这就是 `testApp_1.c` 源码中使用 `open` 创建的新文件, 通过查看它的权限可知与源码中设置的权限是相同的, 文件大小为 1Kbyte, 其中这 1Kbyte 字节数据是从 `src_file` 文件中读取过来的。

剩下的几个例子希望大家自己独立完成。

Tips: 最后再给大家说一点, 就是关于函数返回值的问题, 我们可以发现, 本章给大家所介绍的这些函数都是有返回值的, 其实不管是 Linux 应用编程 API 函数, 还是驱动开发中所使用到的函数, 基本上都是有返回值的, 返回值的作用就是告诉开发人员此函数执行的一个状态是怎样的, 执行成功了还是失败了, 在 Linux 系统下, 绝大部分的函数都是返回 0 作为函数调用成功的标识、而返回负数 (譬如 -1) 表示函数调用失败, 如果大家学习过驱动开发, 想必对此并不陌生, 所以很多时候可以使用如下的方式来判断函数执行成功还是失败:

```
if (func()) {
    // 执行失败
} else {
    // 执行成功
}
```

当然以上说的是大部分情况, 并不是所有函数都是这样设计, 所以呢, 这里笔者也给大家一个建议, 自己在进行编程开发的时候, 自定义函数也可以使用这样的一种方法来设计你的函数返回值, 不管是裸机程序亦或是 Linux 应用程序、驱动程序。

第三章 深入探究文件 I/O

经过上一章内容的学习,相信各位读者对 Linux 系统应用编程中的基础文件 I/O 操作有了一定的认识和理解了,能够独立完成一些简单地文件 I/O 编程问题,如果你的工作中仅仅只是涉及到一些简单文件读写操作相关的问题,其实上一章的知识内容已经够你使用了。

当然作为大部分读者来说,我相信你不会止步于此、还想学习更多的知识内容,那本章笔者将会同各位读者一起,来深入探究文件 I/O 中涉及到的一些问题、原理以及所对应的解决方法,譬如 Linux 系统下文件是如何进行管理的、调用函数返回错误该如何处理、open 函数的 O_APPEND、O_TRUNC 标志以及等相关问题。

好了,废话不多说,开始本章的学习吧,加油!

本章将会讨论如下主题内容。

- 对 Linux 下文件的管理方式进行简单介绍;
- 函数返回错误的处理;
- 退出程序 exit()、_Exit()、_exit();
- 空洞文件的概念;
- open 函数的 O_APPEND 和 O_TRUNC 标志;
- 多次打开同一文件;
- 复制文件描述符;
- 文件共享介绍;
- 原子操作与竞争冒险;
- 系统调用 fcntl()和 ioctl()介绍;
- 截断文件;

3.1 Linux 系统如何管理文件

3.1.1 静态文件与 inode

文件在没有被打开的情况下一般都是存放在磁盘中的，譬如电脑硬盘、移动硬盘、U 盘等外部存储设备，文件存放在磁盘文件系统中，并且以一种固定的形式进行存放，我们把他们称为静态文件。

文件储存在硬盘上，硬盘的最小存储单位叫做“扇区”(Sector)，每个扇区储存 512 字节(相当于 0.5KB)，操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个“块”(block)。这种由多个扇区组成的“块”，是文件存取的最小单位。“块”的大小，最常见的是 4KB，即连续八个 sector 组成一个 block。

所以由此可以知道，静态文件对应的数据都是存储在磁盘设备不同的“块”中，那么问题来了，我们在程序中调用 open 函数是如何找到对应文件的数据存储“块”的呢，难道仅仅通过指定的文件路径就可以实现？这里我们就来简单地聊一聊这内部实现的过程。

我们的磁盘在进行分区、格式化的时候会将其分为两个区域，一个是数据区，用于存储文件中的数据；另一个是 inode 区，用于存放 inode table (inode 表)，inode table 中存放的是一个一个的 inode (也成为 inode 节点)，不同的 inode 就可以表示不同的文件，每一个文件都必须对应一个 inode，inode 实质上是一个结构体，这个结构体中有很多的元素，不同的元素记录了文件了不同信息，譬如文件字节大小、文件所有者、文件对应的读/写/执行权限、文件时间戳(创建时间、更新时间等)、文件类型、文件数据存储的 block (块) 位置等等信息，如图 3.1.1 中所示(这里需要注意的是，文件名并不是记录在 inode 中，这个问题后面章节内容再给大家讲)。

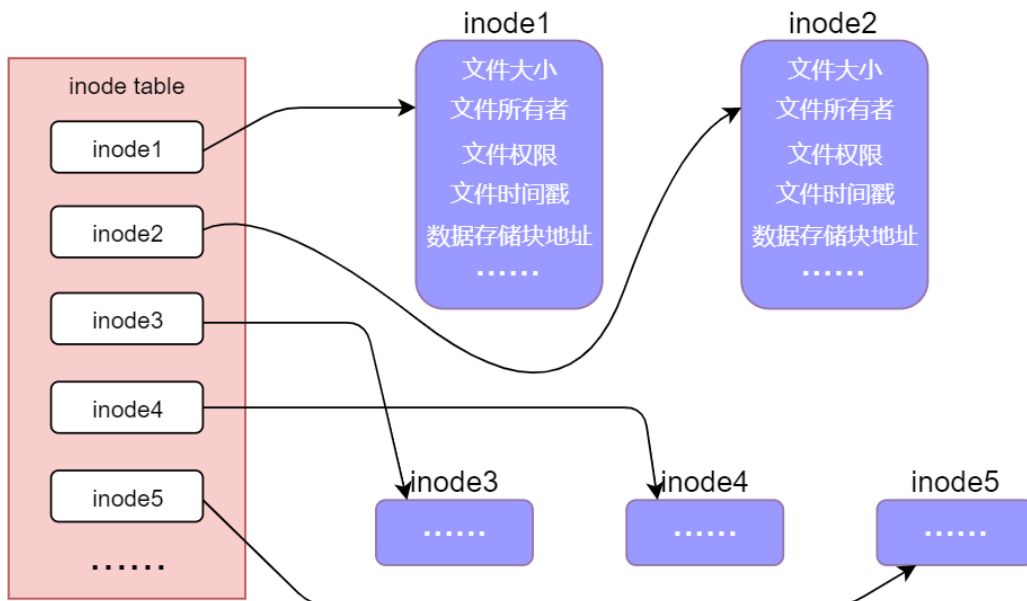


图 3.1.1 inode table 与 inode

所以由此可知，inode table 表本身也需要占用磁盘的存储空间。每一个文件都有唯一的一个 inode，每一个 inode 都有一个与之相对应的数字编号，通过这个数字编号就可以找到 inode table 中所对应的 inode。在 Linux 系统下，我们可以通过“ls -li”命令查看文件的 inode 编号，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls -il
总用量 16
3701769 -rw-rw-r-- 1 dt dt 1583 1月  5 19:54 testApp_1.c
3701836 -rw-rw-r-- 1 dt dt  709 1月  5 20:07 testApp_2.c
3701854 -rw-rw-r-- 1 dt dt 1381 1月  5 20:23 testApp_3.c
3702154 -rw-rw-r-- 1 dt dt  800 1月  5 20:33 testApp_4.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 3.1.2 ls 查看文件的 inode 编号

上图中 ls 打印出来的信息中，每一行前面的一个数字就表示了对应文件的 inode 编号。除此之外，还可以使用 stat 命令查看，用法如下：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ stat testApp_1.c
文件: 'testApp_1.c'
大小: 1583          块: 8          IO 块: 4096    普通文件
设备: 801h/2049d   Inode: 3701769 硬链接: 1
权限: (0664/-rw-rw-r--)  Uid: ( 1000/   dt)  Gid: ( 1000/   dt)
最近访问: 2021-01-05 20:07:30.925816725 +0800
最近更改: 2021-01-05 19:54:20.012237341 +0800
最近改动: 2021-01-05 19:54:20.012237341 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 3.1.3 stat 查看 inode 编号

由以上的介绍大家可以联系到实际操作中，譬如我们在 Windows 下进行 U 盘格式化的时候会有一个“快速格式化”选项，如下所示：

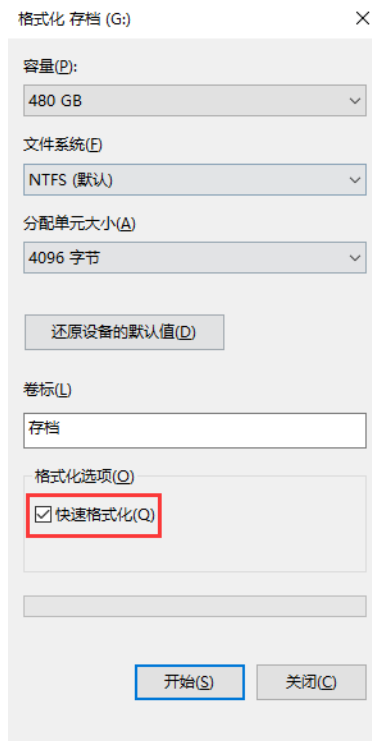


图 3.1.4 Windows 下格式化磁盘

如果勾选了“快速格式化”选项，在进行格式化操作的时候非常的快，而如果不勾选此选项，直接使用普通格式化方式，将会比较慢，那说明这两种格式化方式是存在差异的，其实快速格式化只是删除了 U 盘

中的 inode table 表, 真正存储文件数据的区域并没有动, 所以使用快速格式化的 U 盘, 其中的数据是可以被找回来的。

通过以上介绍可知, 打开一个文件, 系统内部会将这个过程分为三步:

- 1) 系统找到这个文件名所对应的 inode 编号;
- 2) 通过 inode 编号从 inode table 中找到对应的 inode 结构体;
- 3) 根据 inode 结构体中记录的信息, 确定文件数据所在的 block, 并读出数据。

3.1.2 文件打开时的状态

当我们调用 open 函数去打开文件的时候, 内核会申请一段内存 (一段缓冲区), 并且将静态文件的数据内容从磁盘这些存储设备中读取到内存中进行管理、缓存 (也把内存中的这份文件数据叫做动态文件、内核缓冲区)。打开文件后, 以后对这个文件的读写操作, 都是针对内存中这一份动态文件进行相关的操作, 而并不是针对磁盘中存放的静态文件。

当我们对动态文件进行读写操作后, 此时内存中的动态文件和磁盘设备中的静态文件就不同步了, 数据的同步工作由内核完成, 内核会在之后将内存这份动态文件更新 (同步) 到磁盘设备中。由此我们也可以联系到实际操作中, 譬如说:

- 打开一个大文件的时候会较慢;
- 文档写了一半, 没记得保存, 此时电脑因为突然停电直接掉电关机了, 当重启电脑后, 打开编写的文档, 发现之前写的内容已经丢失。

想必各位读者在工作当中都遇到过这种问题吧, 通过上面的介绍, 就解释了为什么会出现这种问题。好, 我们再来说一下, 为什么要这样设计?

因为磁盘、硬盘、U 盘等存储设备基本都是 Flash 块设备, 因为块设备硬件本身有读写限制等特征, 块设备是以一块一块为单位进行读写的 (一个块包含多个扇区, 而一个扇区包含多个字节), 一个字节的改动也需要将该字节所在的 block 全部读取出来进行修改, 修改完成之后再写入块设备中, 所以导致对块设备的读写操作非常不灵活; 而内存可以按字节为单位来操作, 而且可以随机操作任意地址数据, 非常地很灵活, 所以对于操作系统来说, 会先将磁盘中的静态文件读取到内存中进行缓存, 读写操作都是针对这份动态文件, 而不是直接去操作磁盘中的静态文件, 不但操作不灵活, 效率也会下降很多, 因为内存的读写速率远比磁盘读写快得多。

在 Linux 系统中, 内核会为每个进程 (关于进程的概念, 这是后面的内容, 我们可以简单地理解为一个运行的程序就是一个进程, 运行了多个程序那就是存在多个进程) 设置一个专门的数据结构用于管理该进程, 譬如用于记录进程的状态信息、运行特征等, 我们把这个称为进程控制块 (Process control block, 缩写 PCB)。

PCB 数据结构体中有一个指针指向了文件描述符表 (File descriptors), 文件描述符表中的每一个元素索引到对应的文件表 (File table), 文件表也是一个数据结构体, 其中记录了很多文件相关的信息, 譬如文件状态标志、引用计数、当前文件的读写偏移量以及 i-node 指针 (指向该文件对应的 inode) 等, 进程打开的所有文件对应的文件描述符都记录在文件描述符表中, 每一个文件描述符都会指向一个对应的文件表, 其示意图如下所示:

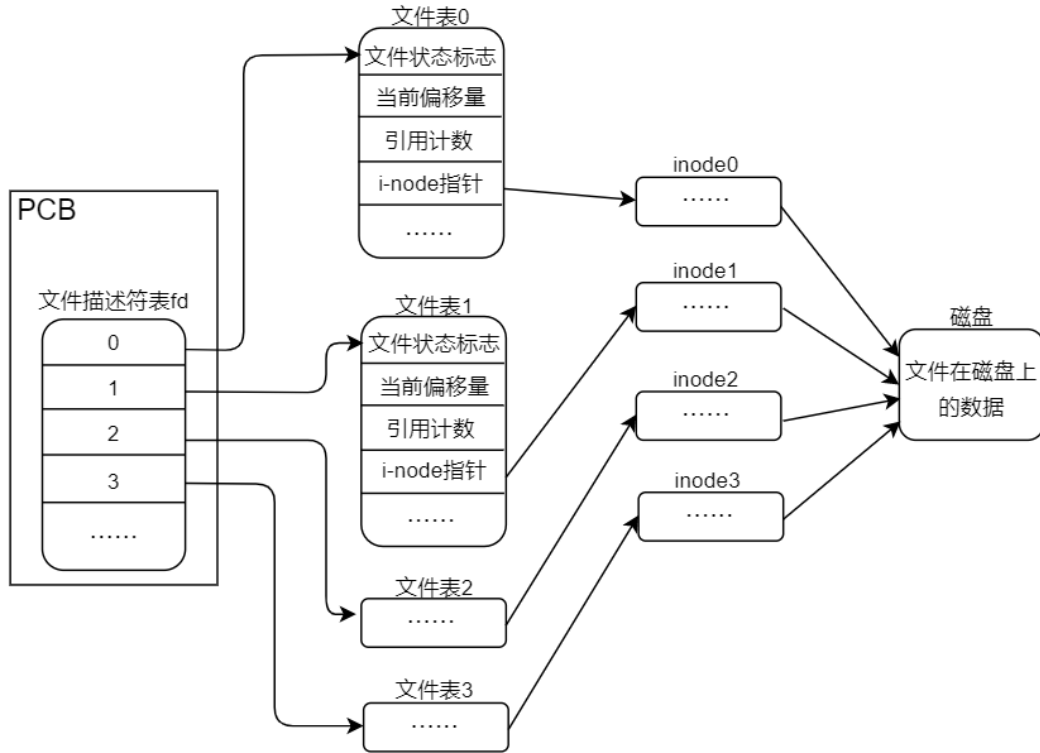


图 3.1.5 文件描述符表、文件表以及 inode 之间的关系

前面给大家介绍了 inode, inode 数据结构体中的元素会记录该文件的数据存储的 block (块), 也就是说可以通过 inode 找到文件数据存在在磁盘设备中的那个位置, 从而把文件数据读取出来。

以上就是本小节给大家介绍到所有内容了, 上面给大家所介绍的内容后面的学习过程中还会用到, 虽然这些理论知识对大家的编程并没有什么影响, 但是会帮助大家理解文件 IO 背后隐藏的一些理论知识, 其实这些理论知识还是非常浅薄的、只是一个大概的认识, 其内部具体的实现是比较复杂的, 当然这个不是我们学习 Linux 应用编程的重点, 操作系统已经帮我们完成了这些具体的实现, 我们要做的仅仅是调用操作系统提供 API 函数来完成自己的工作。

好了, 废话不多说, 我们接着看下一小节内容。

3.2 返回错误处理与 errno

在上一章节中, 笔者给大家编写了很多的示例代码, 大家会发现这些示例代码会有一个共同的特点, 那就是当判断函数执行失败后, 会调用 return 退出程序, 但是对于我们来说, 我们并不知道为什么会出错, 什么原因导致此函数执行失败, 因为执行出错之后它们的返回值都是-1。

难道我们真的就不知道错误原因了吗? 其实不然, 在 Linux 系统下对常见的错误做了一个编号, 每一个编号都代表着每一种不同的错误类型, 当函数执行发生错误的时候, 操作系统会将这个错误所对应的编号赋值给 errno 变量, 每一个进程 (程序) 都维护了自己的 errno 变量, 它是程序中的全局变量, 该变量用于存储就近发生的函数执行错误编号, 也就意味着下一次的错误码会覆盖上一次的错误码。所以由此可知道, 当程序中调用函数发生错误的时候, 操作系统内部会通过设置程序的 errno 变量来告知调用者究竟发生了什么错误!

errno 本质上是一个 int 类型的变量, 用于存储错误编号, 但是需要注意的是, 并不是执行所有的系统调用或 C 库函数出错时, 操作系统都会设置 errno, 那我们如何确定一个函数出错时系统是否会设置 errno 呢? 其实这个通过 man 手册便可以查到, 譬如以 open 函数为例, 执行"man 2 open"打开 open 函数的帮助信息, 找到函数返回值描述段, 如下所示:

```

RETURN VALUE
  open(), openat(), and creat() return the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

ERRORS
  open(), openat(), and creat() can fail with the following errors:

```

图 3.2.1 查看返回值描述信息

从图中红框部分描述文字可知，当函数返回错误时会设置 `errno`，当然这里是以 `open` 函数为例，其它的系统调用也可以这样查找，大家可以自己试试！

在我们的程序当中如何去获取系统所维护的这个 `errno` 变量呢？只需要在我们程序当中包含 `<errno.h>` 头文件即可，你可以直接认为此变量就是在 `<errno.h>` 头文件中的申明的，好，我们来测试下：

```

#include <stdio.h>
#include <errno.h>

int main(void)
{
    printf("%d\n", errno);
    return 0;
}

```

以上的这段代码是不会报错的，大家可以自己试试！

3.2.1 strerror 函数

前面给大家说到了 `errno` 变量，但是 `errno` 仅仅只是一个错误编号，对于开发者来说，即使拿到了 `errno` 也不知道错误为何？还需要对比源码中对此编号的错误定义，可以说非常不友好，这里介绍一个 C 库函数 `strerror()`，该函数可以将对应的 `errno` 转换成适合我们查看的字符串信息，其函数原型如下所示（可通过“`man 3 strerror`”命令查看，注意此函数是 C 库函数，并不是系统调用）：

```

#include <string.h>

char *strerror(int errnum);

```

首先调用此函数需要包含头文件 `<string.h>`。

函数参数和返回值如下：

errnum： 错误编号 `errno`。

返回值： 对应错误编号的字符串描述信息。

测试

接下来我们测试下，测试代码如下：

示例代码 3.2.1 `strerror` 测试代码

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(void)

```



```

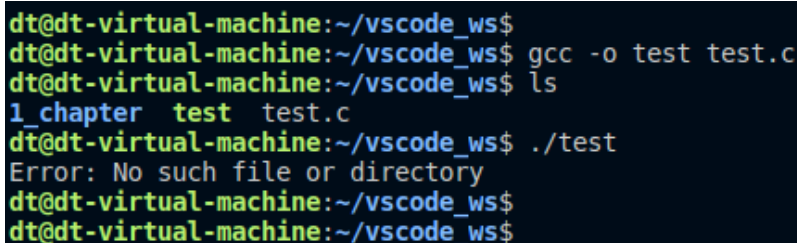
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        printf("Error: %s\n", strerror(errno));
        return -1;
    }

    close(fd);
    return 0;
}

```

编译源代码, 在 Ubuntu 系统下运行测试下, 在当前目录下并不存在 test_file 文件, 测试打印结果如下:



```

dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ gcc -o test test.c
dt@dt-virtual-machine:~/vscode_ws$ ls
1_chapter  test  test.c
dt@dt-virtual-machine:~/vscode_ws$ ./test
Error: No such file or directory
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$

```

图 3.2.2 strerror 测试结果

从打印信息可以知道, strerror 返回的字符串是 "No such file or directory", 所以从打印信息可知, 我们就可以很直观的知道 open 函数执行的错误原因是文件不存在!

3.2.2 perror 函数

除了 strerror 函数之外, 我们还可以使用 perror 函数来查看错误信息, 一般用的最多的还是这个函数, 调用此函数不需要传入 errno, 函数内部会自己去获取 errno 变量的值, 调用此函数会直接将错误提示字符串打印出来, 而不是返回字符串, 除此之外还可以在输出的错误提示字符串之前加入自己的打印信息, 函数原型如下所示 (可通过 "man 3 perror" 命令查看):

```
#include <stdio.h>
```

```
void perror(const char *s);
```

需要包含 <stdio.h> 头文件。

函数参数和返回值含义如下:

s: 在错误提示字符串信息之前, 可加入自己的打印信息, 也可不加, 不加则传入空字符串即可。

返回值: void 无返回值。

测试

接下来我们进行测试, 测试代码如下所示:

示例代码 3.2.2 perror 测试代码

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

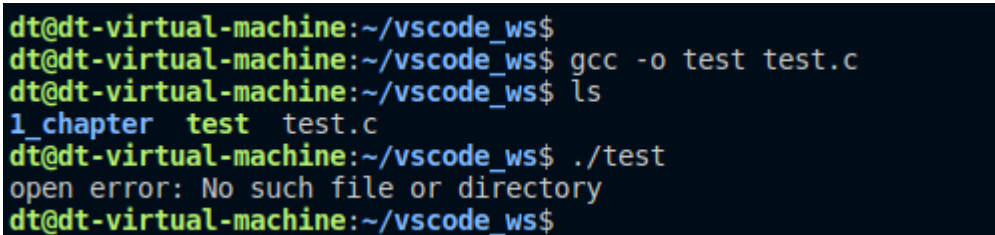
```
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        return -1;
    }

    close(fd);
    return 0;
}
```

编译源代码, 在 Ubuntu 系统下运行测试下, 在当前目录下并不存在 test_file 文件, 测试打印结果如下:



```
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ gcc -o test test.c
dt@dt-virtual-machine:~/vscode_ws$ ls
1_chapter test test.c
dt@dt-virtual-machine:~/vscode_ws$ ./test
open error: No such file or directory
dt@dt-virtual-machine:~/vscode_ws$
```

图 3.2.3 perror 测试结果

从打印信息可以知道, perror 函数打印出来的错误提示字符串是 "No such file or directory", 跟 strerror 函数返回的字符串信息一样, "open error" 便是我们附加的打印信息, 而且从打印信息可知, perror 函数会在附加信息后面自动加入冒号和空格以区分。

以上给大家介绍了 strerror、perror 两个 C 库函数, 都是用于查看函数执行错误时对应的提示信息, 大家用哪个函数都可以, 这里笔者推荐大家使用 perror, 在实际的编程中这个函数用的还是比较多的, 当然除了这两个之外, 其它其它一些类似功能的函数, 这里就不再给大家介绍了, 意义不大!

3.3 exit、_exit、_Exit

当程序在执行某个函数出错的时候, 如果此函数执行失败会导致后面的步骤不能在进行下去时, 应该在出错时终止程序运行, 不应该让程序继续运行下去, 那么如何退出程序、终止程序运行呢? 有过编程经验的读者都知道使用 return, 一般原则程序执行正常退出 return 0, 而执行函数出错退出 return -1, 前面我们所编写的示例代码也是如此。

在 Linux 系统下, 进程(程序)退出可以分为正常退出和异常退出, 注意这里说的异常并不是执行函数出现了错误这种情况, 异常往往更多的是一种不可预料的系统异常, 可能是执行了某个函数时发生的、也有可能是收到了某种信号等, 这里我们只讨论正常退出的情况。

在 Linux 系统下, 进程正常退出除了可以使用 return 之外, 还可以使用 exit()、_exit()以及 _Exit(), 下面我们分别介绍。

3.3.1 _exit()和_Exit()函数

main 函数中使用 return 后返回, return 执行后把控制权交给调用函数, 结束该进程。调用_exit()函数会清除其使用的内存空间, 并销毁其在内核中的各种数据结构, 关闭进程的所有文件描述符, 并结束进程、将控制权交给操作系统。_exit()函数原型如下所示:

```
#include <unistd.h>
```

```
void _exit(int status);
```

调用函数需要传入 status 状态标志, 0 表示正常结束、若为其它值则表示程序执行过程中检测到有错误发生。使用示例如下:

示例代码 3.3.1 _exit()使用示例

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    int fd;
```

```
    /* 打开文件 */
```

```
    fd = open("./test_file", O_RDONLY);
```

```
    if (-1 == fd) {
```

```
        perror("open error");
```

```
        _exit(-1);
```

```
    }
```

```
    close(fd);
```

```
    _exit(0);
```

```
}
```

用法很简单, 大家可以自行测试!

_Exit()函数原型如下所示:

```
#include <stdlib.h>
```

```
void _Exit(int status);
```

_exit()和_Exit()两者等价, 用法作用是一样的, 这里就不再讲了, 需要注意的是这 2 个函数都是系统调用。

3.3.2 exit()函数

exit()函数_exit()函数都是用来终止进程的, exit()是一个标准 C 库函数, 而_exit()和_Exit()是系统调用。执行 exit()会执行一些清理工作, 最后调用_exit()函数。exit()函数原型如下:

```
#include <stdlib.h>
```

```
void exit(int status);
```

该函数是一个标准 C 库函数, 使用该函数需要包含头文件<stdlib.h>, 该函数的用法和_exit()/_Exit()是一样的, 这里就不再多说了。

本小节就给大家介绍了 3 中终止进程的方法:

- main 函数中运行 return;
- 调用 Linux 系统调用_exit()或_Exit();
- 调用 C 标准库函数 exit()。

不管你用哪一种都可以结束进程, 但还是推荐大家使用 exit(), 其实关于 return、exit、_exit/_Exit()之间的区别笔者在上面只是给大家简单地描述了一下, 甚至不太确定我的描述是否正确, 因为笔者并不太多去关心其间的差异, 对这些概念的描述会比较模糊、笼统, 如果大家看不明白可以自己百度搜索相关的内容, 当然对于初学者来说, 不太建议大家去查找这些东西, 至少对你现阶段来说, 意义不是很大。好, 本小节就介绍这么多, 我们接着学习下一小节的内容。

3.4 空洞文件

3.4.1 概念

什么是空洞文件 (hole file)? 在上一章内容中, 笔者给大家介绍了 lseek()系统调用, 使用 lseek 可以修改文件的当前读写位置偏移量, 此函数不但可以改变位置偏移量, 并且还允许文件偏移量超出文件长度, 这是什么意思呢? 譬如有一个 test_file, 该文件的大小是 4K (也就是 4096 个字节), 如果通过 lseek 系统调用将该文件的读写偏移量移动到偏移文件头部 6000 个字节处, 大家想一想会怎样? 如果笔者没有提前告诉大家, 大家觉得不能这样操作, 但事实上 lseek 函数确实可以这样操作。

接下来使用 write()函数对文件进行写入操作, 也就是说此时将是 从偏移文件头部 6000 个字节处开始写入数据, 也就意味着 4096~6000 字节之间出现了一个空洞, 因为这部分空间并没有写入任何数据, 所以形成了空洞, 这部分区域就被称为文件空洞, 那么相应的该文件也被称为空洞文件。

文件空洞部分实际上并不会占用任何物理空间, 直到在某个时刻对空洞部分进行写入数据时才会为它分配对应的空间, 但是空洞文件形成时, 逻辑上该文件的大小是包含了空洞部分的大小的, 这点需要注意。

那说了这么多, 空洞文件有什么用呢? 空洞文件对多线程共同操作文件是及其有用的, 有时候我们创建一个很大的文件, 如果单个线程从头开始依次构建该文件需要很长的时间, 有一种思路就是将文件分为多段, 然后使用多线程来操作, 每个线程负责其中一段数据的写入; 这个有点像我们现实生活当中施工队修路的感觉, 比如说修建一条高速公路, 单个施工队修筑会很慢, 这个时候可以安排多个施工队, 每一个施工队负责修建其中一段, 最后将他们连接起来。

来看一下实际中空洞文件的两个应用场景:

- 在使用迅雷下载文件时, 还未下载完成, 就发现该文件已经占据了全部文件大小的空间, 这也是空洞文件; 下载时如果没有空洞文件, 多线程下载时文件就只能从一个地方写入, 这就不能发挥多线程的作用了; 如果有了空洞文件, 可以从不同的地址同时写入, 就达到了多线程的优势;
- 在创建虚拟机时, 你给虚拟机分配了 100G 的磁盘空间, 但其实系统安装完成之后, 开始也不过只用了 3、4G 的磁盘空间, 如果一开始就把 100G 分配出去, 资源是很大的浪费。

关于空洞文件, 这里就介绍这么多, 上述描述当中多次提到了线程这个概念, 关于线程这是后面的内容, 这里先不给大家讲。

3.4.2 实验测试

这里我们进行相关的测试, 新建一个文件把它做成空洞文件, 示例代码如下所示:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    int fd;
    int ret;
    char buffer[1024];
    int i;

    /* 打开文件 */
    fd = open("./hole_file", O_WRONLY | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件读写位置移动到偏移文件头 4096 个字节(4K)处 */
    ret = lseek(fd, 4096, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    /* 初始化 buffer 为 0xFF */
    memset(buffer, 0xFF, sizeof(buffer));

    /* 循环写入 4 次, 每次写入 1K */
    for (i = 0; i < 4; i++) {

        ret = write(fd, buffer, sizeof(buffer));
        if (-1 == ret) {
            perror("write error");
            goto err;
        }
    }
}
```

```
ret = 0;
err:
/* 关闭文件 */
close(fd);
exit(ret);
}
```

示例代码中, 我们使用 `open` 函数新建了一个文件 `hole_file`, 在 Linux 系统中, 新建文件大小是 0, 也就是没有任何数据写入, 此时使用 `lseek` 函数将读写偏移量移动到 4K 字节处, 再使用 `write` 函数写入数据 `0xFF`, 每次写入 1K, 一共写入 4 次, 也就是写入了 4K 数据, 也就意味着该文件前 4K 是文件空洞部分, 而后 4K 数据才是真正写入的数据。

接下来进行编译测试, 首先确保当前文件目录下不存在 `hole_file` 文件, 测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
hole_file testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -lh hole_file
-rw-r--r-- 1 dt dt 8.0K 1月  8 10:10 hole_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ du -h hole_file
4.0K hole file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

示例代码 3.4.2 空洞文件测试结果

使用 `ls` 命令查看到空洞文件的大小是 8K, 使用 `ls` 命令查看到的大小是文件的逻辑大小, 自然是包括了空洞部分大小和真实数据部分大小; 当使用 `du` 命令查看空洞文件时, 其大小显示为 4K, `du` 命令查看到的大小是文件实际占用存储块的大小。

本小节内容就讲解完了, 最后再向各位抛出一个问题: 若使用 `read` 函数读取文件空洞部分, 读取出来的将会是什么? 关于这个问题大家可以先思考下, 至于结果是什么, 笔者这里便不给出答案了, 大家可以自己动手编写代码进行测试以得出结论。

3.5 O_APPEND 和 O_TRUNC 标志

在上一章给大家讲解 `open` 函数的时候介绍了一些 `open` 函数的 `flags` 标志, 譬如 `O_RDONLY`、`O_WRONLY`、`O_CREAT`、`O_EXCL` 等, 本小节再给大家介绍两个标志, 分别是 `O_APPEND` 和 `O_TRUNC`, 接下来对这两个标志分别进行介绍。

3.5.1 O_TRUNC 标志

`O_TRUNC` 这个标志的作用非常简单, 如果使用了这个标志, 调用 `open` 函数打开文件的时候会将文件原本的内容全部丢弃, 文件大小变为 0; 这里我们直接测试即可! 测试代码如下所示:

示例代码 3.5.1 O_TRUNC 标志测试

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

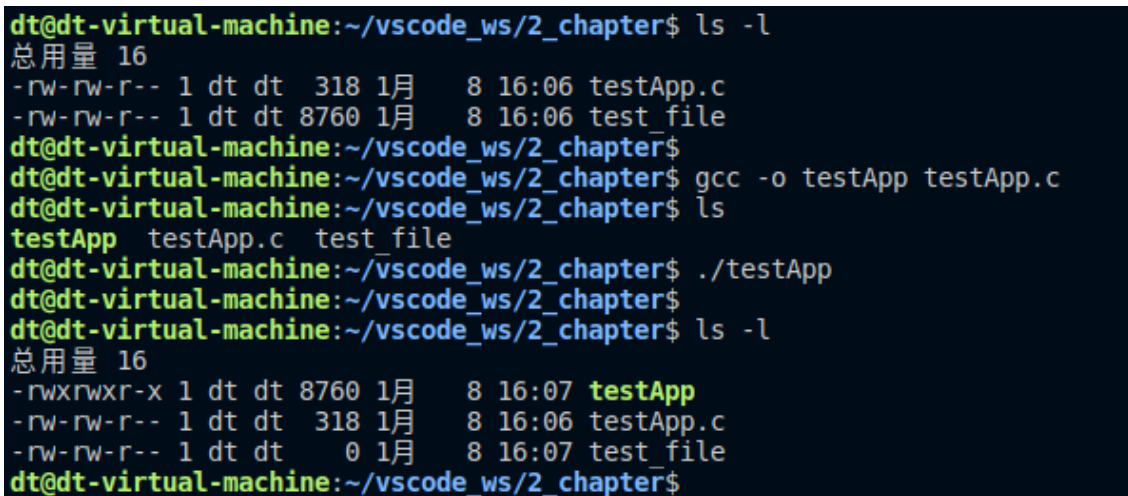
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_WRONLY | O_TRUNC);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

在当前目录下有一个文件 `test_file`，测试代码中使用了 `O_TRUNC` 标志打开该文件，代码中仅仅只是打开该文件，之后调用 `close` 关闭了文件，并没有对其进行读写操作，接下来编译运行来看看测试结果：



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 16
-rw-rw-r-- 1 dt dt 318 1月  8 16:06 testApp.c
-rw-rw-r-- 1 dt dt 8760 1月  8 16:06 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 16
-rwxrwxr-x 1 dt dt 8760 1月  8 16:07 testApp
-rw-rw-r-- 1 dt dt 318 1月  8 16:06 testApp.c
-rw-rw-r-- 1 dt dt  0 1月  8 16:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.5.1 `O_TRUNC` 测试结果

在测试之前 `test_file` 文件中是有数据的，文件大小为 8760 个字节，执行完测试程序后，再使用 `ls` 命令查看文件大小时发现 `test_file` 大小已经变成了 0，也就是说明文件之前的内容已经全部被丢弃了。这就是 `O_TRUNC` 标志的作用了，大家可以自己动手试试。

3.5.2 `O_APPEND` 标志

接下来聊一聊 `O_APPEND` 标志，如果 `open` 函数携带了 `O_APPEND` 标志，调用 `open` 函数打开文件，当每次使用 `write()` 函数对文件进行写操作时，都会自动把文件当前位置偏移量移动到文件末尾，从文件末尾开始写入数据，也就是意味着每次写入数据都是从文件末尾开始。这里我们直接进行测试，测试代码如下所示：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char buffer[16];
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("./test_file", O_RDWR | O_APPEND);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 初始化 buffer 中的数据 */
    memset(buffer, 0x55, sizeof(buffer));

    /* 写入数据: 写入 4 个字节数据 */
    ret = write(fd, buffer, 4);
    if (-1 == ret) {
        perror("write error");
        goto err;
    }

    /* 将 buffer 缓冲区中的数据全部清 0 */
    memset(buffer, 0x00, sizeof(buffer));

    /* 将位置偏移量移动到距离文件末尾 4 个字节处 */
    ret = lseek(fd, -4, SEEK_END);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    /* 读取数据 */
}
```

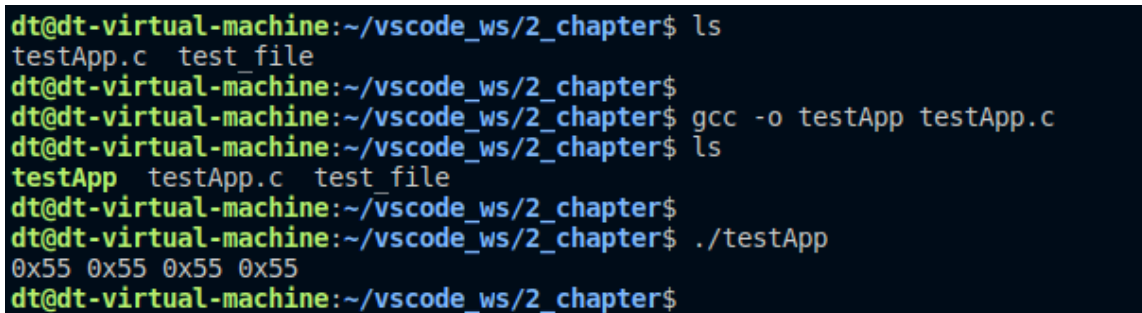


```
ret = read(fd, buffer, 4);
if (-1 == ret) {
    perror("read error");
    goto err;
}

printf("0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[1],
      buffer[2], buffer[3]);

ret = 0;
err:
/* 关闭文件 */
close(fd);
exit(ret);
}
```

测试代码中会去打开当前目录下的 `test_file` 文件, 使用可读可写方式, 并且使用了 `O_APPEND` 标志, 前面笔者给大家提到过, `open` 打开一个文件, 默认的读写位置偏移量会处于文件头, 但测试代码中使用了 `O_APPEND` 标志, 如果 `O_APPEND` 确实能生效的话, 也就意味着调用 `write` 函数会从文件末尾开始写; 代码中写入了 4 个字节数据, 都是 `0x55`, 之后, 使用 `lseek` 函数将位置偏移量移动到距离文件末尾 4 个字节处, 读取 4 个字节 (也就是读取文件最后 4 个字节数据), 之后将其打印出来, 如果上面笔者的描述正确的话, 打印出来的数据就是我们写入的数据, 如果 `O_APPEND` 不能生效, 则打印出来数据就不会是 `0x55`, 接下来编译测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0x55 0x55 0x55 0x55
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.5.2 `O_APPEND` 标志测试结果

从上面打印信息可知, 读取出来的数据确实等于 `0x55`, 说明 `O_APPEND` 标志确实有作用, 当调用 `write()` 函数写文件时, 会自动把文件当前位置偏移量移动到文件末尾。

当然, 本小节内容还并没有结束, 这其中还涉及到一些细节问题需要大家注意, 首先第一点, `O_APPEND` 标志并不会影响读文件, 当读取文件时, `O_APPEND` 标志并不会影响读位置偏移量, 即使使用了 `O_APPEND` 标志, 读文件位置偏移量默认情况下依然是文件头, 关于这个问题大家可以自己进行测试, 编程是一个实践性很强的工作, 有什么不能理解的问题, 可以自己编写程序进行测试。

大家可能会想到使用 `lseek` 函数来改变 `write()` 时的写位置偏移量, 其实这种做法并不会成功, 这就是笔者给大家提的第二个细节, 使用了 `O_APPEND` 标志, 即使是通过 `lseek` 函数也是无法改写文件时对应的位置偏移量 (注意笔者这里说的是写文件, 并不包括读), 写入数据依然是从文件末尾开始, `lseek` 并不会改变写位置偏移量, 这个问题测试方法很简单, 也就是在 `write` 之前使用 `lseek` 修改位置偏移量, 这里笔者就不再给大家测试了, 我还是那句话, 编程是一个实践性很强的工作, 大家只需要把示例代码 3.5.2 进行简单地修改即可!

其实关于第二点细节原因很简单, 当执行 `write()` 函数时, 检测到 `open` 函数携带了 `O_APPEND` 标志, 所以在 `write` 函数内部会自动将写位置偏移量移动到文件末尾, 当然这里也只是笔者的一个简单地猜测, 至于是不是这样, 笔者也无从考证。

到这里本小节的内容就暂时介绍完了, 为什么说是“暂时”? 因为后面的内容中还会聊到 `O_APPEND` 标志, 最后笔者再给大家出一个小问题, 大家可以自己动手测试。

- ◆ 当 `open` 函数同时携带了 `O_APPEND` 和 `O_TRUNC` 两个标志时会有什么作用?

3.6 多次打开同一个文件

大家看到这个小节标题可能会有疑问, 同一个文件还能被多次打开? 事实确实如此, 同一个文件可以被多次打开, 譬如在一个进程中多次打开同一个文件、在多个不同的进程中打开同一个文件, 那么这些操作都是被允许的。本小节就来探讨下多次打开同一个文件会有一些什么现象以及相应的细节问题?

3.6.1 验证一些现象

- 一个进程内多次 `open` 打开同一个文件, 那么会得到多个不同的文件描述符 `fd`, 同理在关闭文件的时候也需要调用 `close` 依次关闭各个文件描述符。

针对这个问题, 我们编写测试代码进行测试, 如下所示:

示例代码 3.6.1 多次打开同一个文件测试代码 1

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd1, fd2, fd3;
    int ret;

    /* 第一次打开文件 */
    fd1 = open("./test_file", O_RDWR);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 第二次打开文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }
}
```

```
    }

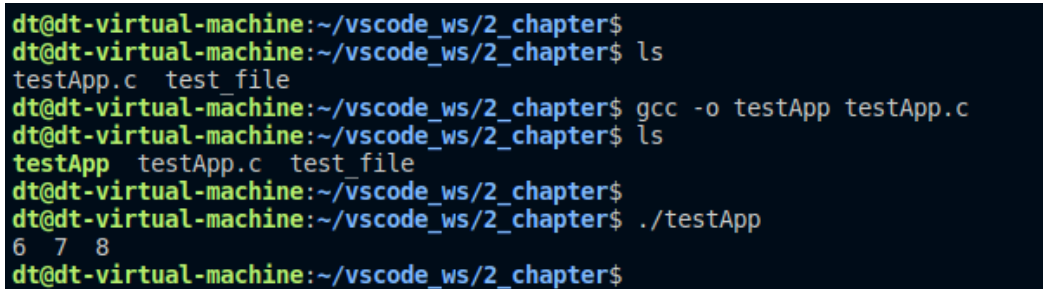
    /* 第三次打开文件 */
    fd3 = open("./test_file", O_RDWR);
    if (-1 == fd3) {
        perror("open error");
        ret = -1;
        goto err2;
    }

    /* 打印出 3 个文件描述符 */
    printf("%d %d %d\n", fd1, fd2, fd3);

    close(fd3);
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

上述示例代码中, 通过 3 次调用 `open` 函数对 `test_file` 文件打开了 3 次, 每一个调用传参一样, 最后将 3 次得到的文件描述符打印出来, 在当前目录下存在 `test_file` 文件, 接下来编译测试, 看看结果如何:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
6 7 8
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.1 打印文件描述符

从打印结果可知, 三次调用 `open` 函数得到的文件描述符分别为 6、7、8, 通过任何一个文件描述符对文件进行 IO 操作都是可以的, 但是需要注意的是, 调用 `open` 函数打开文件使用的是什么权限, 则返回的文件描述符就拥有什么权限, 文件 IO 操作完成之后, 在结束进程之前需要使用 `close` 关闭各个文件描述符。

在图 3.6.1 中, 细心的读者可能会发现, 调用 `open` 函数得到的最小文件描述符是 6, 在上一章节内容中给大家提到过, 程序中分配得到的最小文件描述符一般是 3, 但这里竟然是 6! 这是为何? 其实这个问题跟 `vscode` 有关, 说明 3、4、5 这 3 个文件描述符已经被 `vscode` 软件对应的进程所占用了, 而当前这里执行 `testApp` 文件是在 `vscode` 软件提供的终端下进行的, 所以 `vscode` 可以认为是 `testApp` 进程的父进程, 相反, `testApp` 进程便是 `vscode` 进程的子进程, 子进程会继承父进程的文件描述符。关于子进程和父进程这些都是后面的内容, 这里暂时不给大家进行介绍, 这是只是给大家简单地解释一下, 免得大家误会!

其实可以直接在 Ubuntu 系统的 Terminal 终端执行 testApp, 这时你会发现打印出来的文件描述符分别是 3、4、5, 这里就不给大家演示了。

- 一个进程内多次 open 打开同一个文件, 在内存中并不会存在多份动态文件。

当调用 open 函数的时候, 会将文件数据 (文件内容) 从磁盘等块设备读取到内存中, 将文件数据在内存中进行维护, 内存中的这份文件数据我们就把它称为动态文件! 这是前面给大家介绍的内容, 这里再简单地提一下。这里出现了一个问题: 如果同一个文件被多次打开, 那么该文件所对应的动态文件是否在内存中也存在多份? 也就是说, 多次打开同一个文件是否会将其文件数据多次拷贝到内存中进行维护?

关于这个问题, 各位读者可以简单地思考一下, 这里我们直接编写代码进行测试, 测试代码如下所示:

示例代码 3.6.2 多次打开同一个文件测试代码 2

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char buffer[4];
    int fd1, fd2;
    int ret;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* 通过 fd1 文件描述符写入 4 个字节数据 */
    buffer[0] = 0x11;
    buffer[1] = 0x22;
```

```
buffer[2] = 0x33;
buffer[3] = 0x44;

ret = write(fd1, buffer, 4);
if (-1 == ret) {
    perror("write error");
    goto err2;
}

/* 将读写位置偏移量移动到文件头 */
ret = lseek(fd2, 0, SEEK_SET);
if (-1 == ret) {
    perror("lseek error");
    goto err2;
}

/* 读取数据 */
memset(buffer, 0x00, sizeof(buffer));
ret = read(fd2, buffer, 4);
if (-1 == ret) {
    perror("read error");
    goto err2;
}

printf("0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[1],
        buffer[2], buffer[3]);

ret = 0;
err2:
close(fd2);

err1:
/* 关闭文件 */
close(fd1);
exit(ret);
}
```

当前目录下不存在 `test_file` 文件, 上述代码中, 第一次调用 `open` 函数新建并打开 `test_file` 文件, 第二次调用 `open` 函数再次打开它, 新建文件时, 文件大小为 0; 首先通过文件描述符 `fd1` 写入 4 个字节数据 (0x11/0x22/0x33/0x44), 从文件头开始写; 然后再通过文件描述符 `fd2` 读取 4 个字节数据, 也是从文件头开始读取。假如, 内存中只有一份动态文件, 那么读取得到的数据应该就是 0x11、0x22、0x33、0x44, 如果存在多份动态文件, 那么通过 `fd2` 读取的是与它对应的动态文件中的数据, 那就不是 0x11、0x22、0x33、0x44, 而是读取 0 个字节数据, 因为它的文件大小是 0。

接下来进行编译测试:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0x11 0x22 0x33 0x44
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 3.6.2 测试结果 2

上图中打印显示读取出来的数据是 0x11/0x22/0x33/0x44，所以由此可知，即使多次打开同一个文件，内存中也只有一份动态文件。

- 一个进程内多次 open 打开同一个文件，不同文件描述符所对应的读写位置偏移量是相互独立的。

同一个文件被多次打开，会得到多个不同的文件描述符，也就意味着会有多个不同的文件表，而文件读写偏移量信息就记录在文件表数据结构中，所以从这里可以推测不同的文件描述符所对应的读写偏移量是相互独立的，并没有关联在一起，并且文件表中 i-node 指针指向的都是同一个 inode，如下图所示：

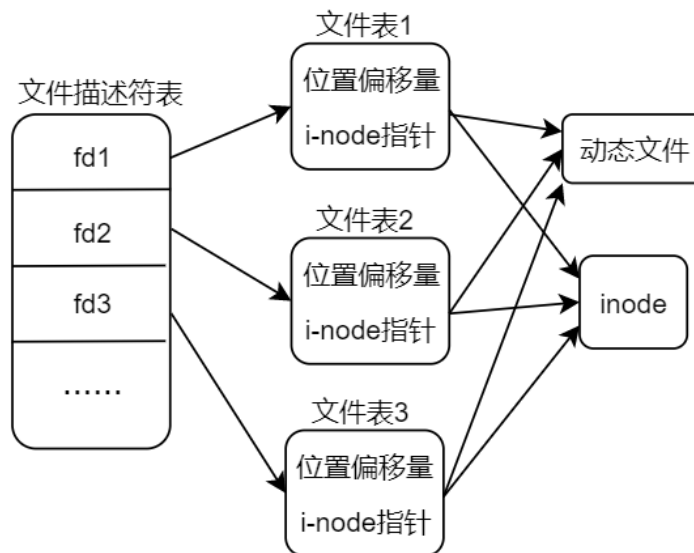


图 3.6.3 多次打开同一个文件--文件描述符表、文件表以及 inode 之间的关系

测试的方法很简单，只需在示例代码 3.6.2 中简单地修改即可，将 lseek 函数调用去掉，然后在编译测试，如果读出的数据依然是 0x11/0x22/0x33/0x44，则表示第三点结论成立，这里不再给大家演示。

Tips: 多个不同的进程中调用 open() 打开磁盘中的同一个文件，同样在内存中也只是维护了一份动态文件，多个进程间共享，它们有各自独立的文件读写位置偏移量。

动态文件何时被关闭呢？当文件的引用计数为 0 时，系统会自动将其关闭，同一个文件被打开多次，文件表中会记录该文件的引用计数，如图 3.1.5 所示，引用计数记录了当前文件被多少个文件描述符 fd 关联。

3.6.2 多次打开同一文件进行读操作与 O_APPEND 标志

重复打开同一个文件，进行写操作，譬如一个进程中两次调用 open 函数打开同一个文件，分别得到两个文件描述符 fd1 和 fd2，使用这两个文件描述符对文件进行写入操作，那么它们是分别写（各从各的位置偏移量开始写）还是接续写（一个写完，另一个接着后面写）？其实这个问题，3.6.1 小节中已经给出了答

案, 因为这两个文件描述符所对应的读写位置偏移量是相互独立的, 所以是分别写, 接下来我们还是编写代码进行测试, 测试代码如下所示:

示例代码 3.6.3 多次打开同一个文件测试代码 3

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* buffer 数据初始化 */
    buffer1[0] = 0x11;
    buffer1[1] = 0x22;
    buffer1[2] = 0x33;
    buffer1[3] = 0x44;

    buffer2[0] = 0xAA;
    buffer2[1] = 0xBB;
    buffer2[2] = 0xCC;
```

```
buffer2[3] = 0xDD;

/* 循环写入数据 */
for (i = 0; i < 4; i++) {

    ret = write(fd1, buffer1, sizeof(buffer1));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }

    ret = write(fd2, buffer2, sizeof(buffer2));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }
}

/* 将读写位置偏移量移动到文件头 */
ret = lseek(fd1, 0, SEEK_SET);
if (-1 == ret) {
    perror("lseek error");
    goto err2;
}

/* 读取数据 */
for (i = 0; i < 8; i++) {

    ret = read(fd1, buffer1, sizeof(buffer1));
    if (-1 == ret) {
        perror("read error");
        goto err2;
    }

    printf("%x%x%x%x", buffer1[0], buffer1[1],
           buffer1[2], buffer1[3]);
}

printf("\n");
ret = 0;
err2:
close(fd2);
```



```
err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

示例代码 3.6.3 中, 重复两次打开 test_file 文件, 分别得到两个文件描述符 fd1、fd2; 首先通过 fd1 写入 4 个字节数据 (0x11、0x22、0x33、0x44) 到文件中, 接着再通过 fd2 写入 4 个字节数据 (0xaa、0xbb、0xcc、0xdd) 到文件中, 循环写入 4 此; 最后再将写入的数据读取出来, 将其打印到终端。如果它们是分别写, 那么读取出来的数据就应该是 aabbccdd....., 因为通过 fd1 写入的数据被 fd2 写入的数据给覆盖了; 如果它们是接续写, 那么读取出来的数据应该是 11223344aabbccdd....., 接下来我们编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
aabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.4 测试结果 3

从打印结果可知, 它们确实是分别写。如果想要实现接续写, 也就是当通过 fd1 写入完成之后, 通过 fd2 写入的数据是接在 fd1 写入的数据之后, 那么该怎么做呢? 当然可以写入数据之前通过 lseek 函数将文件偏移量移动到文件末尾, 如果是这样做, 会存在一些问题, 关于这个问题后面再给大家介绍; 这里我们给大家介绍使用 O_APPEND 标志来解决这个问题, 也就是将分别写更改为接续写。

前面给大家介绍了 open 函数的 O_APPEND 标志, 当 open 函数使用 O_APPEND 标志, 在使用 write 函数进行写入操作时, 会自动将偏移量移动到文件末尾, 也就是每次写入都是从文件末尾开始; 这里结合本小节的内容, 我们再来讨论 O_APPEND 标志, 在多次打开同一个文件进行写操作时, 使用 O_APPEND 标志会有什么样的效果, 接下来进行测试:

示例代码 3.6.4 多次打开同一个文件测试代码 4

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL | O_APPEND,
```

```
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

if (-1 == fd1) {
    perror("open error");
    exit(-1);
}

/* 再次打开 test_file 文件 */
fd2 = open("./test_file", O_RDWR | O_APPEND);
if (-1 == fd2) {
    perror("open error");
    ret = -1;
    goto err1;
}

/* buffer 数据初始化 */
buffer1[0] = 0x11;
buffer1[1] = 0x22;
buffer1[2] = 0x33;
buffer1[3] = 0x44;

buffer2[0] = 0xAA;
buffer2[1] = 0xBB;
buffer2[2] = 0xCC;
buffer2[3] = 0xDD;

/* 循环写入数据 */
for (i = 0; i < 4; i++) {

    ret = write(fd1, buffer1, sizeof(buffer1));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }

    ret = write(fd2, buffer2, sizeof(buffer2));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }
}

/* 将读写位置偏移量移动到文件头 */
ret = lseek(fd1, 0, SEEK_SET);
```

```

    if (-1 == ret) {
        perror("lseek error");
        goto err2;
    }

    /* 读取数据 */
    for (i = 0; i < 8; i++) {

        ret = read(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("read error");
            goto err2;
        }

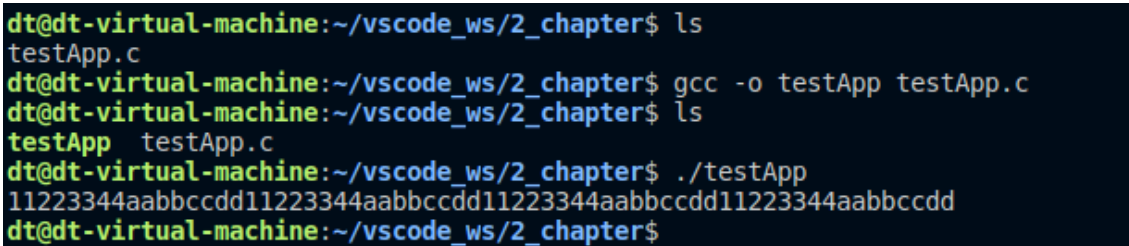
        printf("%x%x%x%x", buffer1[0], buffer1[1],
                buffer1[2], buffer1[3]);
    }

    printf("\n");
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}

```

示例代码 3.6.4 仅仅只是在示例代码 3.6.3 的基础上, open 函数添加了 O_APPEND 标志, 其它内容并没有动过, 接下来编译测试。



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
11223344aabbccdd11223344aabbccdd11223344aabbccdd11223344aabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 3.6.5 测试结果 4

从打印出来的数据可知, 加入了 O_APPEND 标志后, 分别写已经变成了接续写。关于 O_APPEND 标志还涉及到一个原子操作的问题, 后面再给大家介绍, 本小节内容到此!

3.7 复制文件描述符

在 Linux 系统中, open 返回得到的文件描述符 fd 可以进行复制, 复制成功之后可以得到一个新的文件描述符, 使用新的文件描述符和旧的文件描述符都可以对文件进行 IO 操作, 复制得到的文件描述符和旧的

文件描述符拥有相同的权限，譬如使用旧的文件描述符对文件有读写权限，那么新的文件描述符同样也具有读写权限；在 Linux 系统下，可以使用 `dup` 或 `dup2` 这两个系统调用对文件描述符进行复制，本小节就给大家介绍这两个函数的用法以及它们之间的区别。

复制得到的文件描述符与旧的文件描述符都指向了同一个文件表，假设 `fd1` 为原文件描述符，`fd2` 为复制得到的文件描述符，如下图所示：

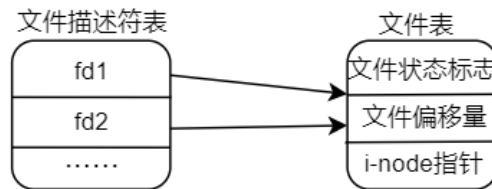


图 3.7.1 指向同一个文件表

因为复制得到的文件描述符与旧的文件描述符指向的是同一个文件表，所以可知，这两个文件描述符的属性是一样，譬如对文件的读写权限、文件状态标志、文件偏移量等，所以从这里也可知道“复制”的含义实则是复制文件表。同样，在使用完毕之后也需要使用 `close` 来关闭文件描述符。

3.7.1 dup 函数

`dup` 函数用于复制文件描述符，此函数原型如下所示（可通过“`man 2 dup`”命令查看）：

```
#include <unistd.h>
```

```
int dup(int oldfd);
```

首先使用此函数需要包含头文件 `<unistd.h>`。

函数参数和返回值含义如下：

oldfd: 需要被复制的文件描述符。

返回值: 成功时将返回一个新的文件描述符，由操作系统分配，分配原则遵循文件描述符分配原则；如果复制失败将返回-1，并且会设置 `errno` 值。

测试

由前面的介绍可知，复制得到的文件描述符与原文件描述符都指向同一个文件表，所以它们的文件读写偏移量是一样的，那么是不是可以在不使用 `O_APPEND` 标志的情况下，通过文件描述符复制来实现接续写，接下来我们编写一个程序进行测试，测试代码如下所示：

示例代码 3.7.1 `dup` 函数测试代码

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int main(void)
```

```
{
```

```
    unsigned char buffer1[4], buffer2[4];
```

```
    int fd1, fd2;
```

```
int ret;
int i;

/* 创建新文件 test_file 并打开 */
fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
           S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if (-1 == fd1) {
    perror("open error");
    exit(-1);
}

/* 复制文件描述符 */
fd2 = dup(fd1);
if (-1 == fd2) {
    perror("dup error");
    ret = -1;
    goto err1;
}

printf("fd1: %d\nfd2: %d\n", fd1, fd2);

/* buffer 数据初始化 */
buffer1[0] = 0x11;
buffer1[1] = 0x22;
buffer1[2] = 0x33;
buffer1[3] = 0x44;

buffer2[0] = 0xAA;
buffer2[1] = 0xBB;
buffer2[2] = 0xCC;
buffer2[3] = 0xDD;

/* 循环写入数据 */
for (i = 0; i < 4; i++) {

    ret = write(fd1, buffer1, sizeof(buffer1));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }

    ret = write(fd2, buffer2, sizeof(buffer2));
    if (-1 == ret) {
```

```
        perror("write error");
        goto err2;
    }
}

/* 将读写位置偏移量移动到文件头 */
ret = lseek(fd1, 0, SEEK_SET);
if (-1 == ret) {
    perror("lseek error");
    goto err2;
}

/* 读取数据 */
for (i = 0; i < 8; i++) {

    ret = read(fd1, buffer1, sizeof(buffer1));
    if (-1 == ret) {
        perror("read error");
        goto err2;
    }

    printf("%x%x%x%x", buffer1[0], buffer1[1],
           buffer1[2], buffer1[3]);
}

printf("\n");
ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

测试代码中, 我们使用了 `dup` 系统调用复制了文件描述符 `fd1`, 得到另一个新的文件描述符 `fd2`, 分别通过 `fd1` 和 `fd2` 对文件进行写操作, 最后读取写入的数据来判断是分别写还是接续写, 接下来编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 7
11223344aabbccdd11223344aabbccdd11223344aabbccdd11223344aabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.7.2 dup 测试结果

由打印信息可知, fd1 等于 6, 复制得到的新的文件描述符为 7 (遵循 fd 分配原则), 打印出来的数据显示为接续写, 所以可知, 通过复制文件描述符可以实现接续写。

3.7.2 dup2 函数

dup 系统调用分配的文件描述符是由系统分配的, 遵循文件描述符分配原则, 并不能自己指定一个文件描述符, 这是 dup 系统调用的一个缺陷; 而 dup2 系统调用修复了这个缺陷, 可以手动指定文件描述符, 而不需要遵循文件描述符分配原则, 当然在实际的编程工作中, 需要根据自己的情况来进行选择。

dup2 函数原型如下所示 (可以通过 "man 2 dup2" 命令查看):

```
#include <unistd.h>
```

```
int dup2(int oldfd, int newfd);
```

同样使用该命令也需要包含 <unistd.h> 头文件。

函数参数和返回值含义如下:

oldfd: 需要被复制的文件描述符。

newfd: 指定一个文件描述符 (需要指定一个当前进程没有使用到的文件描述符)。

返回值: 成功时将返回一个新的文件描述符, 也就是手动指定的文件描述符 newfd; 如果复制失败将返回 -1, 并且会设置 errno 值。

测试

接下来编写一个简单地测试程序, 如下所示:

示例代码 3.7.2 dup2 函数测试代码

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    int fd1, fd2;
```

```
    int ret;
```

```
    /* 创建新文件 test_file 并打开 */
```

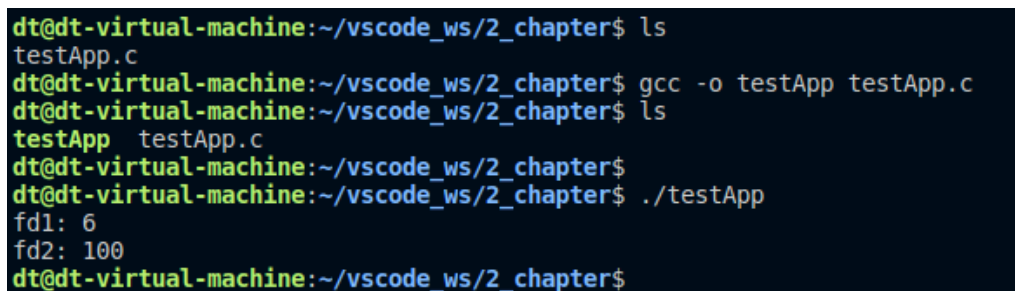
```
fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
           S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if (-1 == fd1) {
    perror("open error");
    exit(-1);
}

/* 复制文件描述符 */
fd2 = dup2(fd1, 100);
if (-1 == fd2) {
    perror("dup error");
    ret = -1;
    goto err1;
}

printf("fd1: %d\nfd2: %d\n", fd1, fd2);
ret = 0;

close(fd2);
err1:
/* 关闭文件 */
close(fd1);
exit(ret);
}
```

测试代码使用 `dup2` 函数复制文件描述符 `fd1`, 指定新的文件描述符为 `100`, 复制成功之后将其打印出来, 结果如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 100
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.7.3 dup2 函数测试结果

由打印信息可知, 复制得到的文件描述符 `fd2` 等于 `100`, 正是我们在 `dup2` 函数中指定的文件描述符。本小节的内容到这里结束了, 最后再强调一点, 文件描述符并不是只能复制一次, 实际上可以对同一个文件描述符 `fd` 调用 `dup` 或 `dup2` 函数复制多次, 得到多个不同的文件描述符。

3.8 文件共享

什么是文件共享? 所谓文件共享指的是同一个文件(譬如磁盘上的同一个文件, 对应同一个 `inode`)被多个独立的读写体同时进行 IO 操作。多个独立的读写体大家可以将其简单地理解为对应于同一个文件的多个不同的文件描述符, 譬如多次打开同一个文件所得到的多个不同的 `fd`, 或使用 `dup()` (或 `dup2`) 函数复制得到的多个不同的 `fd` 等。

同时进行 IO 操作指的是一个读写体操作文件尚未调用 close 关闭的情况下,另一个读写体去操作文件,前面给大家编写的示例代码中就已经涉及到了文件共享的内容了,譬如 3.6 小节中编写的示例代码中,同一个文件对应两个不同的文件描述符 fd1 和 fd2,当使用 fd1 对文件进行写操作之后,并没有关闭 fd1,而此时使用 fd2 对文件再进行写操作,这其实就是一种文件共享。

文件共享的意义有很多,多用于多进程或多线程编程环境中,譬如我们可以通过文件共享的方式来实现多个线程同时操作同一个大文件,以减少文件读写时间、提升效率。

文件共享的核心是:如何制造出多个不同的文件描述符来指向同一个文件。其实方法在上面的内容中都已经给大家介绍过了,譬如多次调用 open 函数重复打开同一个文件得到多个不同的文件描述符、使用 dup() 或 dup2() 函数对文件描述符进行复制以得到多个不同的文件描述符。

常见的三种文件共享的实现方式

(1) 同一个进程中多次调用 open 函数打开同一个文件,各数据结构之间的关系如下图所示:

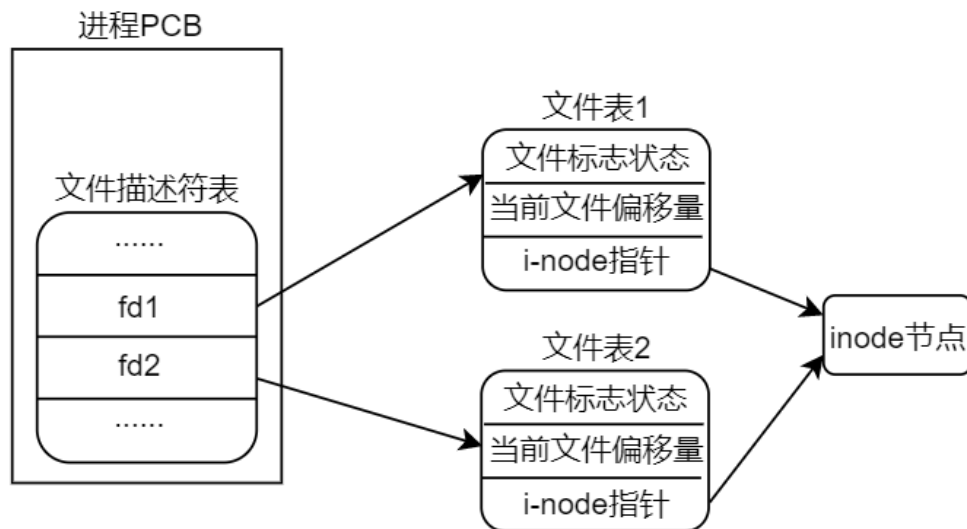


图 3.8.1 同一进程多次 open 打开同一文件各数据结构关系图

这种情况非常简单,多次调用 open 函数打开同一个文件会得到多个不同的文件描述符,并且多个文件描述符对应多个不同的文件表,所有的文件表都索引到了同一个 inode 节点,也就是磁盘上的同一个文件。

(2) 不同进程中分别使用 open 函数打开同一个文件,其数据结构关系图如下所示:

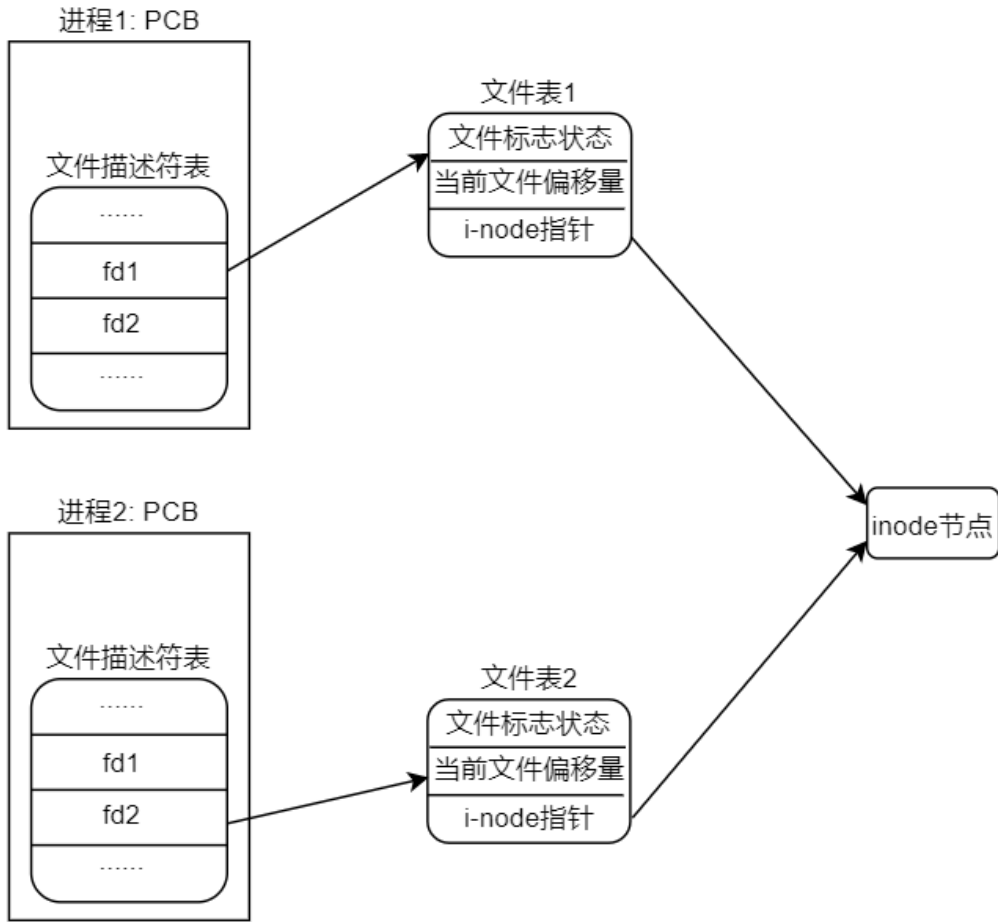


图 3.8.2 不同进程 open 打开同一文件数据结构关系图

进程 1 和进程 2 分别是运行在 Linux 系统上两个独立的进程（理解为两个独立的程序），在他们各自的程序中分别调用 open 函数打开同一个文件，进程 1 对应的文件描述符为 fd1，进程 2 对应的文件描述符为 fd2，fd1 指向了进程 1 的文件表 1，fd2 指向了进程 2 的文件表 2；各自的文件表都索引到了同一个 inode 节点，从而实现共享文件。

(3)同一个进程中通过 dup (dup2) 函数对文件描述符进行复制，其数据结构关系如下图所示：

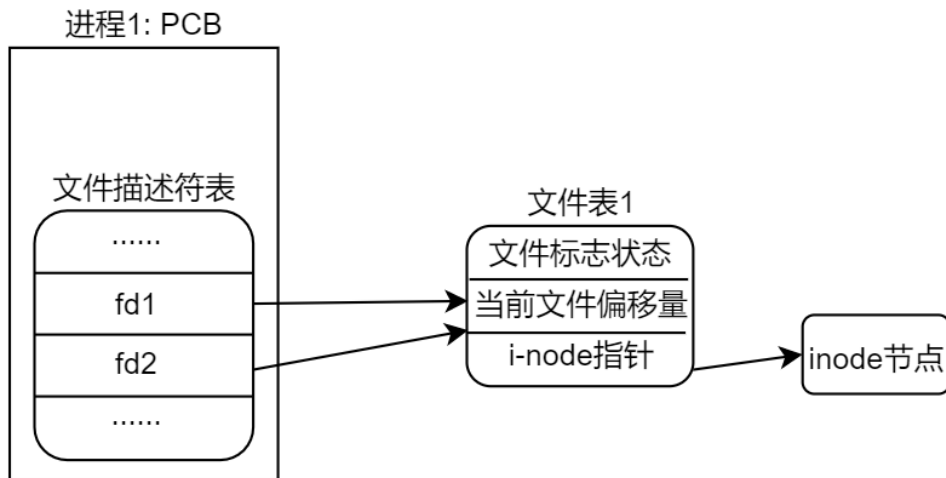


图 3.8.3 dup 复制文件描述符实现文件共享

这种方式上一小节已经给大家进行了详细讲解，这里不再重述！

对于文件共享,存在着竞争冒险,这个是需要大家关注的,下一小节将会向大家介绍。除此之外,我们还需要关心的是文件共享时,不同的读写体之间是分别写还是接续写,这些细节问题大家都要搞清楚。

3.9 原子操作与竞争冒险

Linux 是一个多任务、多进程操作系统,系统中往往运行着多个不同的进程、任务,多个不同的进程就有可能对同一个文件进行 IO 操作,此时该文件便是它们的共享资源,它们共同操作着同一份文件;操作系统级编程不同于大家以前接触的裸机编程,裸机程序中不存在进程、多任务这种概念,而在 Linux 系统中,我们必须留意到多进程环境下可能会导致的竞争冒险。

3.9.1 竞争冒险简介

本小节给大家竞争冒险这个概念,如果学习过 Linux 驱动开发的读者对这些概念应该并不陌生,也就意味着竞争冒险不但存在于 Linux 应用层、也存在于 Linux 内核驱动层。

假设有两个独立的进程 A 和进程 B 都对同一个文件进行追加写操作(也就是在文件末尾写入数据),每一个进程都调用了 open 函数打开了该文件,但未使用 O_APPEND 标志,此时,各数据结构之间的关系如图 3.8.2 所示。每个进程都有它自己的进程控制块 PCB,有自己的文件表(意味着有自己独立的读写位置偏移量),但是共享同一个 inode 节点(也就是对应同一个文件)。假定此时进程 A 处于运行状态,B 未处于等待运行状态,进程 A 调用了 lseek 函数,它将进程 A 的该文件当前位置偏移量设置为 1500 字节处(假设这里是文件末尾),刚好此时进程 A 的时间片耗尽,然后内核切换到了进程 B,进程 B 执行 lseek 函数,也将其对该文件的当前位置偏移量设置为 1500 个字节处(文件末尾)。然后进程 B 调用 write 函数,写入了 100 个字节数据,那么此时在进程 B 中,该文件的当前位置偏移量已经移动到了 1600 字节处。B 进程时间片耗尽,内核又切换到了进程 A,使进程 A 恢复运行,当进程 A 调用 write 函数时,是从进程 A 的该文件当前位置偏移量(1500 字节处)开始写入,此时文件 1500 字节处已经不再是文件末尾了,如果还从 1500 字节处写入就会覆盖进程 B 刚才写入到该文件中的数据。

其上述假设工作流程图如下图所示:

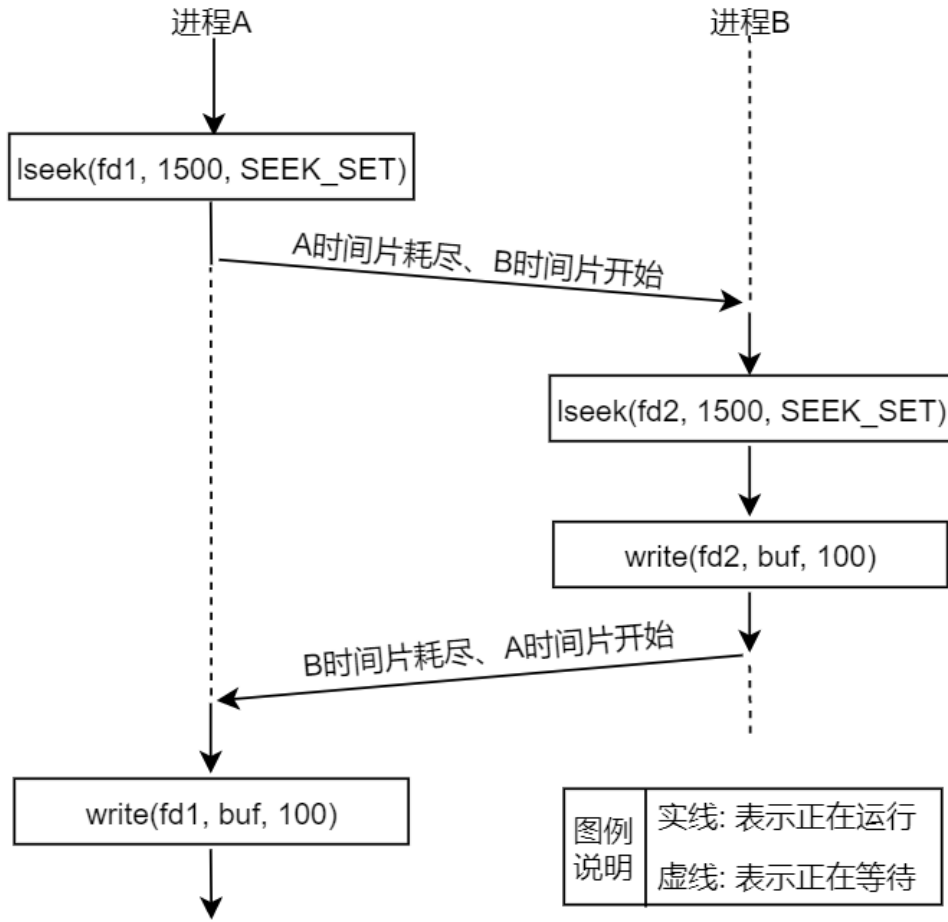


图 3.9.1 假设中的 AB 进程工作流程

以上给大家所描述的这样一种情形就属于竞争状态（也成为竞争冒险），操作共享资源的两个进程（或线程），其操作之后的所得到的结果往往是不可预期的，因为每个进程（或线程）去操作文件的顺序是不可预期的，即这些进程获得 CPU 使用权的先后顺序是不可预期的，完全由操作系统调配，这就是所谓的竞争状态。

既然存在竞争状态，那么该如何规避或消除这种状态呢？接下来给大家介绍原子操作。

3.9.2 原子操作

在上一章给大家介绍 open 函数的时候就提到过“原子操作”这个概念了，同样在 Linux 驱动编程中，也有这个概念，相信学习过 Linux 驱动编程开发的读者应该有印象。

从上一小节给大家提到的示例中可知，上述的问题出在逻辑操作“先定位到文件末尾，然后再写”，它使用了两个分开的函数调用，首先使用 lseek 函数将文件当前位置偏移量移动到文件末尾、然后在使用 write 函数将数据写入到文件。既然知道了问题所在，那么解决办法就是将这两个操作步骤合并成一个原子操作，所谓原子操作，是有多步操作组成的一个操作，原子操作要么一步也不执行，一旦执行，必须要执行完所有步骤，不可能只执行所有步骤中的一个子集。

(1)O_APPEND 实现原子操作

在上一小节给大家提到的示例中，进程 A 和进程 B 都对同一个文件进行追加写操作，导致进程 A 写入的数据覆盖了进程 B 写入的数据，解决办法就是将“先定位到文件末尾，然后写”这两个步骤组成一个原子操作即可，那如何使其变成一个原子操作呢？答案就是 O_APPEND 标志。

前面已经给大家多次提到过了 O_APPEND 标志, 但是并没有给大家介绍 O_APPEND 的一个非常重要的作用, 那就是实现原子操作。当 open 函数的 flags 参数中包含了 O_APPEND 标志, 每次执行 write 写入操作时都会将文件当前写位置偏移量移动到文件末尾, 然后再写入数据, 这里“移动当前写位置偏移量到文件末尾、写入数据”这两个操作步骤就组成了一个原子操作, 加入 O_APPEND 标志后, 不管怎么写入数据都会是从文件末尾写, 这样就不会导致出现“进程 A 写入的数据覆盖了进程 B 写入的数据”这种情况了。

(2)pread()和 pwrite()

pread()和 pwrite()都是系统调用, 与 read()、write()函数的作用一样, 用于读取和写入数据。区别在于, pread()和 pwrite()可用于实现原子操作, 调用 pread 函数或 pwrite 函数可传入一个位置偏移量 offset 参数, 用于指定文件当前读或写的位置偏移量, 所以调用 pread 相当于调用 lseek 后再调用 read; 同理, 调用 pwrite 相当于调用 lseek 后再调用 write。所以可知, 使用 pread 或 pwrite 函数不需要使用 lseek 来调整当前位置偏移量, 并将“移动当前位置偏移量、读或写”这两步操作组成一个原子操作。

pread、pwrite 函数原型如下所示 (可通过“man 2 pread”或“man 2 pwrite”命令来查看):

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

首先调用这两个函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下:

fd、buf、count 参数与 read 或 write 函数意义相同。

offset: 表示当前需要进行读或写的位置偏移量。

返回值: 返回值与 read、write 函数返回值意义一样。

虽然 pread (或 pwrite) 函数相当于 lseek 与 pread (或 pwrite) 函数的集合, 但还是有下列区别:

- 调用 pread 函数时, 无法中断其定位和读操作 (也就是原子操作);
- 不更新文件表中的当前位置偏移量。

关于第二点我们可以编写一个简单地代码进行测试, 测试代码如下所示:

示例代码 3.9.1 pread 函数测试

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    unsigned char buffer[100];
    int fd;
    int ret;

    /* 打开文件 test_file */
    fd = open("./test_file", O_RDWR);
    if (-1 == fd) {
        perror("open error");
    }
}
```

```

        exit(-1);
    }

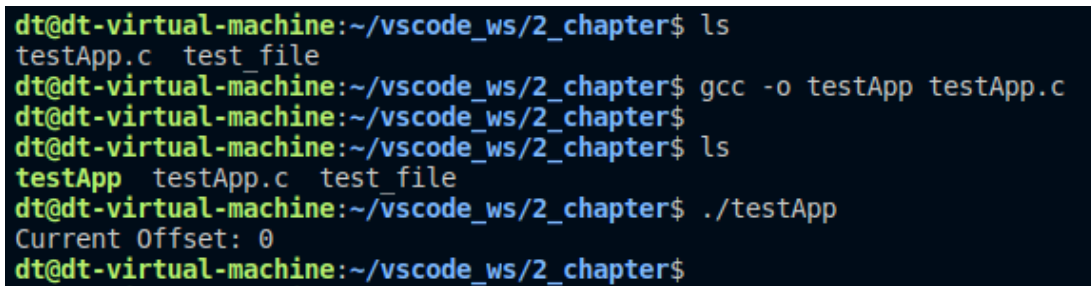
    /* 使用 pread 函数读取数据(从偏移文件头 1024 字节处开始读取) */
    ret = pread(fd, buffer, sizeof(buffer), 1024);
    if (-1 == ret) {
        perror("pread error");
        goto err;
    }

    /* 获取当前位置偏移量 */
    ret = lseek(fd, 0, SEEK_CUR);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    printf("Current Offset: %d\n", ret);
    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}

```

在当前目录下存在一个文件 `test_file`，上述代码中会打开 `test_file` 文件，然后直接使用 `pread` 函数读取 100 个字节数据，从偏移文件头部 1024 字节处，读取完成之后再使用 `lseek` 函数获取到文件当前位置偏移量，并将其打印出来。假如 `pread` 函数会改变文件表中记录的当前位置偏移量，则打印出来的数据应该是 $1024 + 100 = 1124$ ；如果不会改变文件表中记录的当前位置偏移量，则打印出来的数据应该是 0，接下来编译代码测试：



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Current Offset: 0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 3.9.2 pread 函数测试结果

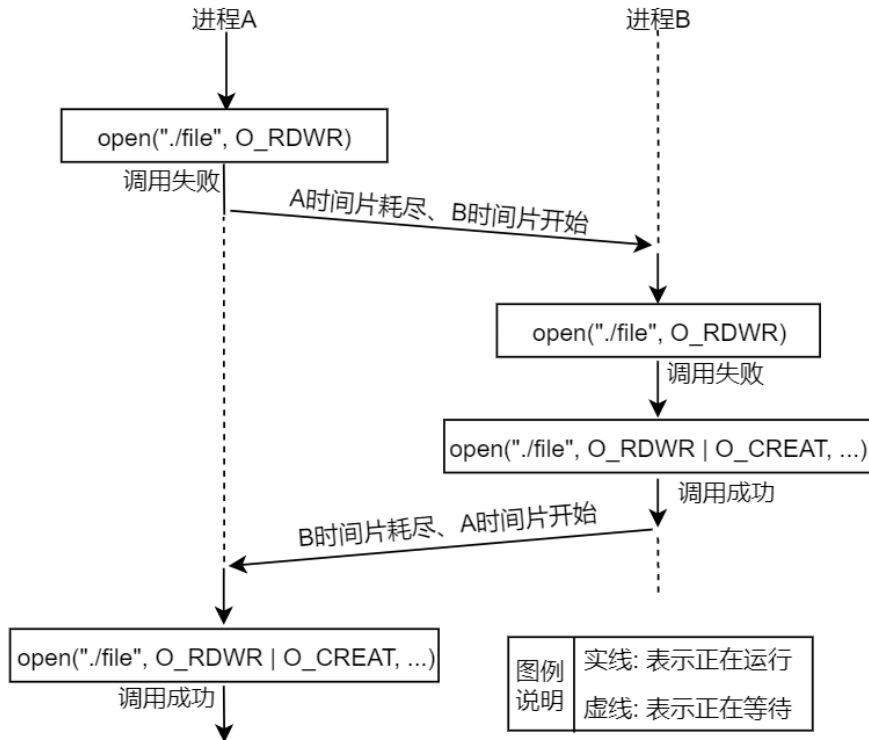
从上图中可知，打印出来的数据为 0，正如前面所介绍那样，`pread` 函数确实不会改变文件表中记录的当前位置偏移量；同理，`pwrite` 函数也是如此，大家可以把 `pread` 换成 `pwrite` 函数再次进行测试，不出意外，打印出来的数据依然是 0。

如果把 `pread` 函数换成 `read`（或 `write`）函数，那么打印出来的数据就是 100 了，因为读取了 100 个字节数据，相应的当前位置偏移量会向后移动 100 个字节。

(3) 创建一个文件

前面给大家介绍 `open` 函数的 `O_EXCL` 标志的时候,也提到了原子操作,其中介绍到:`O_EXCL` 可以用于测试一个文件是否存在,如果不存在则创建此文件,如果存在则返回错误,这使得测试和创建两者成为一个原子操作。接下来给大家,创建文件中存在着的一个竞争状态。

假设有这么一个情况:进程 A 和进程 B 都要去打开同一个文件、并且此文件还不存在。进程 A 当前正在运行状态、进程 B 处于等待状态,进程 A 首先调用 `open("./file", O_RDWR)` 函数尝试去打开文件,结果返回错误,也就是调用 `open` 失败;接着进程 A 时间片耗尽、进程 B 运行,同样进程 B 调用 `open("./file", O_RDWR)` 尝试打开文件,结果也失败,接着进程 B 再次调用 `open("./file", O_RDWR | O_CREAT, ...)` 创建此文件,这一次 `open` 执行成功,文件创建成功;接着进程 B 时间片耗尽、进程 A 继续运行,进程 A 也调用 `open("./file", O_RDWR | O_CREAT, ...)` 创建文件,函数执行成功,如下图所示:



从上面的示例可知,进程 A 和进程 B 都会创建出同一个文件,同一个文件被创建两次这是不允许的,那如何规避这样的问题呢?那就是通过使用 `O_EXCL` 标志,当 `open` 函数中同时指定了 `O_EXCL` 和 `O_CREAT` 标志,如果要打开的文件已经存在,则 `open` 返回错误;如果指定的文件不存在,则创建这个文件,这里就提供了一种机制,保证进程是打开文件的创建者,将“判断文件是否存在、创建文件”这两个步骤合成为一个原子操作,有了原子操作,就保证不会出现图 3.9.3 中所示的情况。

3.10 fcntl 和 ioctl

本小节给大家介绍两个新的系统调用: `fcntl()` 和 `ioctl()`。

3.10.1 fcntl 函数

`fcntl()` 函数可以对一个已经打开的文件描述符执行一系列控制操作,譬如复制一个文件描述符(与 `dup`、`dup2` 作用相同)、获取/设置文件描述符标志、获取/设置文件状态标志等,类似于一个多功能文件描述符管理工具箱。`fcntl()` 函数原型如下所示(可通过“`man 2 fcntl`”命令查看):

```
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, ... /* arg */) 
```

函数参数和返回值含义如下:

fd: 文件描述符。

cmd: 操作命令。此参数表示我们将要对 fd 进行什么操作, cmd 参数支持很多操作命令, 大家可以打开 man 手册查看到这些操作命令的详细介绍, 这些命令都是以 F_XXX 开头的, 譬如 F_DUPFD、F_GETFD、F_SETFD 等, 不同的 cmd 具有不同的作用, cmd 操作命令大致可以分为以下 5 种功能:

- 复制文件描述符 (cmd=F_DUPFD 或 cmd=F_DUPFD_CLOEXEC);
- 获取/设置文件描述符标志 (cmd=F_GETFD 或 cmd=F_SETFD);
- 获取/设置文件状态标志 (cmd=F_GETFL 或 cmd=F_SETFL);
- 获取/设置异步 IO 所有权 (cmd=F_GETOWN 或 cmd=F_SETOWN);
- 获取/设置记录锁 (cmd=F_GETLK 或 cmd=F_SETLK);

这里列举出来, 并不需要全部学会每一个 cmd 的作用, 因为有些内容并没有给大家提及到, 譬如什么异步 IO、锁之类的概念, 在后面的学习过程中, 当学习到相关知识内容的时候再给大家介绍。

...: fcntl 函数是一个可变参函数, 第三个参数需要根据不同的 cmd 来传入对应的实参, 配合 cmd 来使用。

返回值: 执行失败情况下, 返回-1, 并且会设置 errno; 执行成功的情况下, 其返回值与 cmd (操作命令) 有关, 譬如 cmd=F_DUPFD (复制文件描述符) 将返回一个新的文件描述符、cmd=F_GETFD (获取文件描述符标志) 将返回文件描述符标志、cmd=F_GETFL (获取文件状态标志) 将返回文件状态标志等。

fcntl 使用示例

(1)复制文件描述符

前面给大家介绍了 dup 和 dup2, 用于复制文件描述符, 除此之外, 我们还可以通过 fcntl 函数复制文件描述符, 可用的 cmd 包括 F_DUPFD 和 F_DUPFD_CLOEXEC, 这里就只介绍 F_DUPFD, F_DUPFD_CLOEXEC 暂时先不讲。

当 cmd=F_DUPFD 时, 它的作用会根据 fd 复制出一个新的文件描述符, 此时需要传入第三个参数, 第三个参数用于指出新复制出的文件描述符是一个大于或等于该参数的可用文件描述符 (没有使用的文件描述符); 如果第三个参数等于一个已经存在的文件描述符, 则取一个大于该参数的可用文件描述符。

测试代码如下所示:

示例代码 3.10.1 fcntl 复制文件描述符

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd1, fd2;
    int ret;

    /* 打开文件 test_file */
```



```

fd1 = open("./test_file", O_RDONLY);
if (-1 == fd1) {
    perror("open error");
    exit(-1);
}

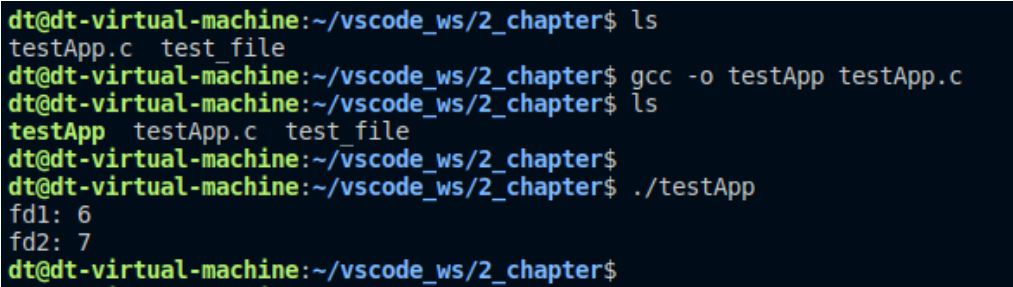
/* 使用 fcntl 函数复制一个文件描述符 */
fd2 = fcntl(fd1, F_DUPFD, 0);
if (-1 == fd2) {
    perror("fcntl error");
    ret = -1;
    goto err;
}

printf("fd1: %d\nfd2: %d\n", fd1, fd2);

ret = 0;
close(fd2);
err:
/* 关闭文件 */
close(fd1);
exit(ret);
}

```

在当前目录下存在 test_file 文件, 上述代码会打开此文件, 得到文件描述符 fd1, 之后再使用 fcntl 函数复制 fd1 得到新的文件描述符 fd2, 并将 fd1 和 fd2 打印出来, 接下来编译运行:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 7
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 3.10.1 fcntl 复制文件描述符测试结果

可知复制得到的文件描述符是 7, 因为在执行 fcntl 函数时, 传入的第三个参数是 0, 也就时指定复制得到的新文件描述符必须要大于或等于 0, 但是因为 0~6 都已经被占用了, 所以分配得到的 fd 就是 7; 如果传入的第三个参数是 100, 那么 fd2 就会等于 100, 大家可以自己动手测试。

(2) 获取/设置文件状态标志

cmd=F_GETFL 可用于获取文件状态标志, cmd=F_SETFL 可用于设置文件状态标志。cmd=F_GETFL 时不需要传入第三个参数, 返回值成功表示获取到的文件状态标志; cmd=F_SETFL 时, 需要传入第三个参数, 此参数表示需要设置的文件状态标志。

这些标志指的就是我们在调用 open 函数时传入的 flags 标志, 可以指定一个或多个 (通过位或 | 运算符组合), 但是文件权限标志 (O_RDONLY、O_WRONLY、O_RDWR) 以及文件创建标志 (O_CREAT、O_EXCL、O_NOCTTY、O_TRUNC) 不能被设置、会被忽略; 在 Linux 系统中, 只有 O_APPEND、O_ASYNC、

O_DIRECT、O_NOATIME 以及 O_NONBLOCK 这些标志可以被修改, 这里面有些标志并没有给大家介绍过, 后面我们在用到的时候再给大家介绍。所以对于一个已经打开的文件描述符, 可以通过这种方式添加或删除标志。

测试代码如下:

示例代码 3.10.2 fcntl 读取/设置文件状态标志

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int fd;
    int ret;
    int flag;

    /* 打开文件 test_file */
    fd = open("./test_file", O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 获取文件状态标志 */
    flag = fcntl(fd, F_GETFL);
    if (-1 == flag) {
        perror("fcntl F_GETFL error");
        ret = -1;
        goto err;
    }

    printf("flags: 0x%x\n", flag);

    /* 设置文件状态标志,添加 O_APPEND 标志 */
    ret = fcntl(fd, F_SETFL, flag | O_APPEND);
    if (-1 == ret) {
        perror("fcntl F_SETFL error");
        goto err;
    }

    ret = 0;
err:
}
```

err:

```

/* 关闭文件 */
close(fd);
exit(ret);
}

```

上述代码会打开 test_file 文件, 得到文件描述符 fd, 之后调用 `fcntl(fd, F_GETFL)` 来获取到当前文件状态标志 flag, 并将其打印来; 接着调用 `fcntl(fd, F_SETFL, flag | O_APPEND)` 设置文件状态标志, 在原标志的基础上添加 O_APPEND 标志。接下来编译测试:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
flags: 0x8002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 3.10.2 fcntl 测试结果 2

以上给大家介绍了 `fcntl` 函数的两种用法, 除了这两种用法之外, 还有其它多种不同的用法, 这里暂时先不介绍了, 后面学习到相应知识点的时候再给大家讲解。

3.10.2 ioctl 函数

`ioctl()` 可以认为是一个文件 IO 操作的杂物箱, 可以处理的事情非常杂、不统一, 一般用于操作特殊文件或硬件外设, 此函数将会在进阶篇中使用到, 譬如可以通过 `ioctl` 获取 LCD 相关信息等, 本小节只是给大家引出这个系统调用, 暂时不会用到。此函数原型如下所示 (可通过 "man 2 ioctl" 命令查看):

```
#include <sys/ioctl.h>
```

```
int ioctl(int fd, unsigned long request, ...);
```

使用此函数需要包含头文件 `<sys/ioctl.h>`。

函数参数和返回值含义如下:

fd: 文件描述符。

request: 此参数与具体要操作的对象有关, 没有统一值, 表示向文件描述符请求相应的操作; 后面用到的时候再给大家介绍。

...: 此函数是一个可变参函数, 第三个参数需要根据 request 参数来决定, 配合 request 来使用。

返回值: 成功返回 0, 失败返回 -1。

关于 `ioctl` 函数就给大家介绍这么多, 目的仅仅是给大家引出这个系统调用, 我们将会在第二篇 <进阶篇> 中给大家细说。

3.11 截断文件

使用系统调用 `truncate()` 或 `ftruncate()` 可将普通文件截断为指定字节长度, 其函数原型如下所示:

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
int truncate(const char *path, off_t length);
```

```
int ftruncate(int fd, off_t length);
```

这两个函数的区别在于: `ftruncate()` 使用文件描述符 `fd` 来指定目标文件, 而 `truncate()` 则直接使用文件路径 `path` 来指定目标文件, 其功能一样。

这两个函数都可以对文件进行截断操作, 将文件截断为参数 `length` 指定的字节长度, 什么是截断? 如果文件目前的大小大于参数 `length` 所指定的大小, 则多余的数据将被丢失, 类似于多余的部分被“砍”掉了; 如果文件目前的大小小于参数 `length` 所指定的大小, 则将其进行扩展, 对扩展部分进行读取将得到空字节“\0”。

使用 `ftruncate()` 函数进行文件截断操作之前, 必须调用 `open()` 函数打开该文件得到文件描述符, 并且必须要具有可写权限, 也就是调用 `open()` 打开文件时需要指定 `O_WRONLY` 或 `O_RDWR`。

调用这两个函数并不会导致文件读写位置偏移量发生改变, 所以截断之后一般需要重新设置文件当前的读写位置偏移量, 以免由于之前所指向的位置已经不存在而发生错误(譬如文件长度变短了, 文件当前所指向的读写位置已不存在)。

调用成功返回 0, 失败将返回 -1, 并设置 `errno` 以指示错误原因。

使用示例

示例代码 3.11.1 演示了文件的截断操作, 分别使用 `ftruncate()` 和 `truncate()` 将当前目录下的文件 `file1` 截断为长度 0、将文件 `file2` 截断为长度 1024 个字节。

示例代码 3.11.1 文件截断操作

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
    int fd;

    /* 打开 file1 文件 */
    if (0 > (fd = open("./file1", O_RDWR))) {
        perror("open error");
        exit(-1);
    }

    /* 使用 ftruncate 将 file1 文件截断为长度 0 字节 */
    if (0 > ftruncate(fd, 0)) {
        perror("ftruncate error");
        exit(-1);
    }

    /* 使用 truncate 将 file2 文件截断为长度 1024 字节 */
    if (0 > truncate("./file2", 1024)) {
        perror("truncate error");
```

```
        exit(-1);
    }

    /* 关闭 file1 退出程序 */
    close(fd);
    exit(0);
}
```

上述代码中, 首先使用 `open()` 函数打开文件 `file1`, 得到文件描述符 `fd`, 接着使用 `ftruncate()` 系统调用将文件截断为 0 长度, 传入 `file1` 文件对应的文件描述符; 接着调用 `truncate()` 系统调用将文件 `file2` 截断为 1024 字节长度, 传入 `file2` 文件的相对路径。

接下来进行测试, 在当前目录下准备两个文件 `file1` 和 `file2`, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1 file2 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 12
-rw-rw-r-- 1 dt dt 592 5月 28 18:25 file1
-rw-rw-r-- 1 dt dt 592 5月 28 18:25 file2
-rw-rw-r-- 1 dt dt 592 5月 28 18:25 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.11.1 准备 `file1` 文件和 `file2` 文件

可以看到 `file1` 和 `file2` 文件此时均为 592 字节大小, 接下来运行测试代码:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1 file2 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1 file2 testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l file1 file2
-rw-rw-r-- 1 dt dt 0 5月 28 18:40 file1
-rw-rw-r-- 1 dt dt 1024 5月 28 18:40 file2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.11.2 测试结果

程序运行之后, `file1` 文件大小变成了 0, 而 `file2` 文件大小变成了 1024 字节, 与测试代码想要实现的功能是一致的。

第四章 标准 I/O 库

本章介绍标准 I/O 库, 不仅是 Linux, 很多其它的操作系统都实现了标准 I/O 库。标准 I/O 虽然是对文件 I/O 进行了封装, 但事实上并不仅仅是如此, 标准 I/O 会处理很多细节, 譬如分配 `stdio` 缓冲区、以优化的块长度执行 I/O 等, 这些处理使用户不必担心如何选择使用正确的块长度。

本章将会讨论如下主题内容。

- 标准 I/O 库简介;
- 流和 FILE 对象;
- 标准输入、标准输出以及标准错误;
- 使用标准 I/O 库函数打开、读写、关闭文件;
- 格式化 I/O, 格式化输出 `printf`、格式化输入 `scanf`;
- 文件 I/O 缓冲, 内核缓冲区和 `stdio` 缓冲区;
- 文件 I/O 与标准 I/O 混合编程。

4.1 标准 I/O 库简介

在第一章介绍应用编程概念时向大家介绍了系统调用与标准 C 语言函数库（以下简称标准 C 库），所谓标准 I/O 库则是标准 C 库中用于文件 I/O 操作（譬如读文件、写文件等）相关的一系列库函数的集合，通常标准 I/O 库函数相关的函数定义都在头文件 `<stdio.h>` 中，所以我们需要在程序源码中包含 `<stdio.h>` 头文件。

标准 I/O 库函数是构建于文件 I/O（`open()`、`read()`、`write()`、`lseek()`、`close()`等）这些系统调用之上的，譬如标准 I/O 库函数 `fopen()` 就利用系统调用 `open()` 来执行打开文件的操作、`fread()` 利用系统调用 `read()` 来执行读文件操作、`fwrite()` 则利用系统调用 `write()` 来执行写文件操作等等。

那既然如此，为何还需要设计标准 I/O 库？直接使用文件 I/O 系统调用不是更好吗？事实上，并非如此，在第一章中我们也提到过，设计库函数是为了提供比底层系统调用更为方便、好用的调用接口，虽然标准 I/O 构建于文件 I/O 之上，但标准 I/O 却有它自己的优势，标准 I/O 和文件 I/O 的区别如下：

- 虽然标准 I/O 和文件 I/O 都是 C 语言函数，但是标准 I/O 是标准 C 库函数，而文件 I/O 则是 Linux 系统调用；
- 标准 I/O 是由文件 I/O 封装而来，标准 I/O 内部实际上是调用文件 I/O 来完成实际操作的；
- 可移植性：标准 I/O 相比于文件 I/O 具有更好的可移植性，通常对于不同的操作系统，其内核向应用层提供的系统调用往往都是不同，譬如系统调用的定义、功能、参数列表、返回值等往往都是不一样的；而对于标准 I/O 来说，由于很多操作系统都实现了标准 I/O 库，标准 I/O 库在不同的操作系统之间其接口定义几乎是一样的，所以标准 I/O 在不同操作系统之间相比于文件 I/O 具有更好的可移植性。
- 性能、效率：标准 I/O 库在用户空间维护了自己的 `stdio` 缓冲区，所以标准 I/O 是带有缓存的，而文件 I/O 在用户空间是不带有缓存的，所以在性能、效率上，标准 I/O 要优于文件 I/O。

关于标准 I/O 库相关介绍就到这了，从下小节开始将正式向大家介绍如何在我们的应用程序中使用标准 I/O 库函数。

4.2 FILE 指针

在 0 中，所介绍的所有文件 I/O 函数（`open()`、`read()`、`write()`、`lseek()`等）都是围绕文件描述符进行的，当调用 `open()` 函数打开一个文件时，即返回一个文件描述符 `fd`，然后该文件描述符就用于后续的 I/O 操作。而对于标准 I/O 库函数来说，它们的操作是围绕 FILE 指针进行的，当使用标准 I/O 库函数打开或创建一个文件时，会返回一个指向 FILE 类型对象的指针（`FILE *`），使用该 FILE 指针与被打开或创建的文件相关联，然后该 FILE 指针就用于后续的标准 I/O 操作（使用标准 I/O 库函数进行 I/O 操作），所以由此可知，FILE 指针的作用相当于文件描述符，只不过 FILE 指针用于标准 I/O 库函数中、而文件描述符则用于文件 I/O 系统调用中。

FILE 是一个结构体数据类型，它包含了标准 I/O 库函数为管理文件所需要的所有信息，包括用于实际 I/O 的文件描述符、指向文件缓冲区的指针、缓冲区的长度、当前缓冲区中的字节数以及出错标志等。FILE 数据结构定义在标准 I/O 库函数头文件 `stdio.h` 中。

4.3 标准输入、标准输出和标准错误

关于标准输入、标准输出以及标准错误这三个概念在 2.2 小节有所提及，所谓标准输入设备指的就是计算机系统的标准的输入设备，通常指的是计算机所连接的键盘；而标准输出设备指的是计算机系统中用于输出标准信息设备，通常指的是计算机所连接的显示器；标准错误设备则指的是计算机系统中用于显示错误信息的设备，通常也指的是显示器设备。

用户通过标准输入设备与系统进行交互, 进程将从标准输入 (stdin) 文件中得到输入数据, 将正常输出数据 (譬如程序中 printf 打印输出的字符串) 输出到标准输出 (stdout) 文件, 而将错误信息 (譬如函数调用报错打印的信息) 输出到标准错误 (stderr) 文件。

标准输出文件和标准错误文件都对应终端的屏幕, 而标准输入文件则对应于键盘。

每个进程启动之后都会默认打开标准输入、标准输出以及标准错误, 得到三个文件描述符, 即 0、1、2, 其中 0 代表标准输入、1 代表标准输出、2 代表标准错误; 在应用编程中可以使用宏 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 分别代表 0、1、2, 这些宏定义在 unistd.h 头文件中:

```
/* Standard file descriptors. */
#define STDIN_FILENO 0 /* Standard input. */
#define STDOUT_FILENO 1 /* Standard output. */
#define STDERR_FILENO 2 /* Standard error output. */
```

0、1、2 这三个是文件描述符, 只能用于文件 I/O (read()、write()等), 那么在标准 I/O 中, 自然是无法使用文件描述符来对文件进行 I/O 操作的, 它们需要围绕 FILE 类型指针来进行, 在 stdio.h 头文件中有相应的定义, 如下:

```
/* Standard streams. */
extern struct _IO_FILE *stdin; /* Standard input stream. */
extern struct _IO_FILE *stdout; /* Standard output stream. */
extern struct _IO_FILE *stderr; /* Standard error output stream. */
/* C89/C99 say they're macros. Make them happy. */
#define stdin stdin
#define stdout stdout
#define stderr stderr
```

Tips: struct _IO_FILE 结构体就是 FILE 结构体, 使用了 typedef 进行了重命名。

所以, 在标准 I/O 中, 可以使用 stdin、stdout、stderr 来表示标准输入、标准输出和标准错误。

4.4 打开文件 fopen()

在 0 所介绍的文件 I/O 中, 使用 open() 系统调用打开或创建文件, 而在标准 I/O 中, 我们将使用库函数 fopen() 打开或创建文件, fopen() 函数原型如下所示:

```
#include <stdio.h>
```

```
FILE *fopen(const char *path, const char *mode);
```

使用该函数需要包含头文件 stdio.h。

函数参数和返回值含义如下:

path: 参数 path 指向文件路径, 可以是绝对路径、也可以是相对路径。

mode: 参数 mode 指定了对该文件的读写权限, 是一个字符串, 稍后介绍。

返回值: 调用成功返回一个指向 FILE 类型对象的指针 (FILE *), 该指针与打开或创建的文件相关联, 后续的标准 I/O 操作将围绕 FILE 指针进行。如果失败则返回 NULL, 并设置 errno 以指示错误原因。

参数 mode 字符串类型, 可取值为如下值之一:

mode	说明	对应于 open() 函数的 flags 参数取值
r	以只读方式打开文件。	O_RDONLY
r+	以可读、可写方式打开文件。	O_RDWR

w	以只写方式打开文件, 如果参数 path 指定的文件存在, 将文件长度截断为 0; 如果指定文件不存在则创建该文件。	O_WRONLY O_CREAT O_TRUNC
w+	以可读、可写方式打开文件, 如果参数 path 指定的文件存在, 将文件长度截断为 0; 如果指定文件不存在则创建该文件。	O_RDWR O_CREAT O_TRUNC
a	以只写方式打开文件, 打开以进行追加内容 (在文件末尾写入), 如果文件不存在则创建该文件。	O_WRONLY O_CREAT O_APPEND
a+	以可读、可写方式打开文件, 以追加方式写入 (在文件末尾写入), 如果文件不存在则创建该文件。	O_RDWR O_CREAT O_APPEND

表 4.4.1 标注 I/O fopen()函数的 mode 参数

新建文件的权限

由 fopen()函数原型可知, fopen()只有两个参数 path 和 mode, 不同于 open()系统调用, 它并没有任何一个参数来指定新建文件的权限。当参数 mode 取值为"w"、"w+"、"a"、"a+"之一时, 如果参数 path 指定的文件不存在, 则会创建该文件, 那么新的文件的权限是如何确定的呢?

虽然调用 fopen()函数新建文件时无法手动指定文件的权限, 但却有一个默认值:

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH (0666)
```

使用示例

使用只读方式打开文件:

```
fopen(path, "r");
```

使用可读、可写方式打开文件:

```
fopen(path, "r+");
```

使用只写方式打开文件, 并将文件长度截断为 0, 如果文件不存在则创建该文件:

```
fopen(path, "w");
```

fclose()关闭文件

调用 fclose()库函数可以关闭一个由 fopen()打开的文件, 其函数原型如下所示:

```
#include <stdio.h>
```

```
int fclose(FILE *stream);
```

参数 stream 为 FILE 类型指针, 调用成功返回 0; 失败将返回 EOF (也就是-1), 并且会设置 errno 来指示错误原因。

4.5 读文件和写文件

当使用 fopen()库函数打开文件之后, 接着我们便可以使用 fread()和 fwrite()库函数对文件进行读、写操作了, 函数原型如下所示:

```
#include <stdio.h>
```

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

库函数 `fread()` 用于读取文件数据, 其参数和返回值含义如下:

ptr: `fread()` 将读取到的数据存放在参数 `ptr` 指向的缓冲区中;

size: `fread()` 从文件读取 `nmemb` 个数据项, 每一个数据项的大小为 `size` 个字节, 所以总共读取的数据大小为 `nmemb * size` 个字节。

nmemb: 参数 `nmemb` 指定了读取数据项的个数。

stream: `FILE` 指针。

返回值: 调用成功时返回读取到的数据项的数目 (数据项数目并不等于实际读取的字节数, 除非参数 `size` 等于 1); 如果发生错误或到达文件末尾, 则 `fread()` 返回的值将小于参数 `nmemb`, 那么到底发生了错误还是到达了文件末尾, `fread()` 不能区分文件结尾和错误, 究竟是哪一种情况, 此时可以使用 `ferror()` 或 `feof()` 函数来判断, 具体参考 4.7 小节内容的介绍。

库函数 `fwrite()` 用于将数据写入到文件中, 其参数和返回值含义如下:

ptr: 将参数 `ptr` 指向的缓冲区中的数据写入到文件中。

size: 参数 `size` 指定了每个数据项的字节大小, 与 `fread()` 函数的 `size` 参数意义相同。

nmemb: 参数 `nmemb` 指定了写入的数据项个数, 与 `fread()` 函数的 `nmemb` 参数意义相同。

stream: `FILE` 指针。

返回值: 调用成功时返回写入的数据项的数目 (数据项数目并不等于实际写入的字节数, 除非参数 `size` 等于 1); 如果发生错误, 则 `fwrite()` 返回的值将小于参数 `nmemb` (或者等于 0)。

由此可知, 库函数 `fread()`、`fwrite()` 中指定读取或写入数据大小的方式与系统调用 `read()`、`write()` 不同, 前者通过 `nmemb` (数据项个数) * `size` (每个数据项的大小) 的方式来指定数据大小, 而后者则直接通过一个 `size` 参数指定数据大小。

譬如要将一个 `struct mystr` 结构体数据写入到文件中, 可按如下方式写入:

```
fwrite(buf, sizeof(struct mystr), 1, file);
```

当然也可以按如下方式写:

```
fwrite(buf, 1, sizeof(struct mystr), file);
```

使用示例

结合使用本小节与上小节所学内容, 我们来编写一个简单地示例代码, 使用标准 I/O 方式对文件进行读写操作。示例代码 4.5.1 演示了使用 `fwrite()` 库函数将数据写入到文件中。

示例代码 4.5.1 标准 I/O 之 `fwrite()` 写文件

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char buf[] = "Hello World!\n";
    FILE *fp = NULL;

    /* 打开文件 */
    if (NULL == (fp = fopen("./test_file", "w"))) {
        perror("fopen error");
        exit(-1);
    }

    printf("文件打开成功!\n");
```

```

/* 写入数据 */
if (sizeof(buf) >
    fwrite(buf, 1, sizeof(buf), fp)) {
    printf("fwrite error\n");
    fclose(fp);
    exit(-1);
}

printf("数据写入成功!\n");

/* 关闭文件 */
fclose(fp);
exit(0);
}

```

首先使用 `fopen()` 函数将当前目录下的 `test_file` 文件打开, 调用 `fopen()` 时 `mode` 参数设置为 "w", 表示以只写的方式打开文件, 并将文件的长度截断为 0, 如果指定文件不存在则创建该文件。打开文件之后调用 `fwrite()` 函数将 "Hello World!" 字符串数据写入到文件中。

写入完成之后, 调用 `fclose()` 函数关闭文件, 退出程序。

编译运行:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
数据写入成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 4.5.1 测试结果

示例代码 4.5.2 演示了使用库函数 `fread()` 从文件中读取数据。

示例代码 4.5.2 标准 I/O 之 `fread()` 读文件

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char buf[50] = {0};
    FILE *fp = NULL;
    int size;

    /* 打开文件 */
    if (NULL == (fp = fopen("./test_file", "r"))) {

```

```
        perror("fopen error");
        exit(-1);
    }

    printf("文件打开成功!\n");

    /* 读取数据 */
    if (12 > (size = fread(buf, 1, 12, fp))) {
        if (ferror(fp)) { //使用 ferror 判断是否是发生错误
            printf("fread error\n");
            fclose(fp);
            exit(-1);
        }

        /* 如果未发生错误则意味着已经到达了文件末尾 */
    }

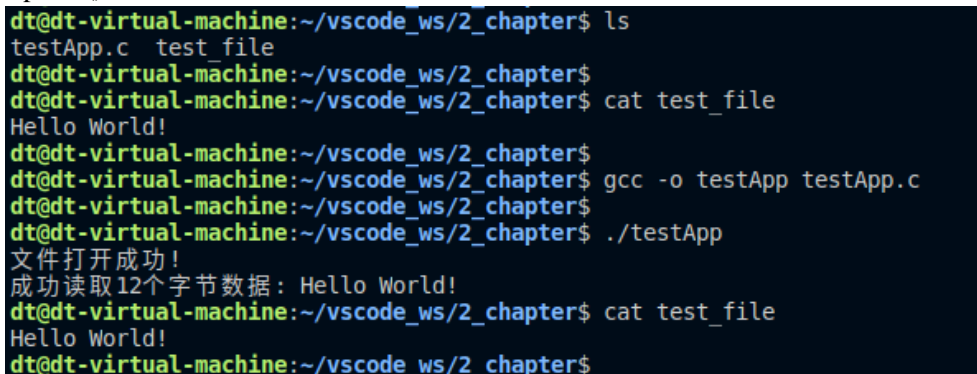
    printf("成功读取%d 个字节数据: %s\n", size, buf);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

首先同样使用 `fopen()` 打开当前目录下的 `test_file` 文件得到 `FILE` 指针, 调用 `fopen()` 时其参数 `mode` 设置为 "r", 表示以只读方式打开文件。

接着使用 `fread()` 函数从文件中读取 `12 * 1=12` 个字节的数据, 将读取到的数据存放在 `buf` 中, 当读取到的字节数小于指定字节数时, 表示发生了错误或者已经到达了文件末尾, 程序中调用了库函数 `ferror()` 来判断是不是发生了错误, 该函数将会在 4.7 小节中介绍。如果未发生错误, 那么就意味着已经到达了文件末尾, 其实也就说明了在调用 `fread()` 读文件时对应的读写位置到文件末尾之间的字节数小于指定的字节数。

最后调用 `printf()` 打印结果, 编译测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
成功读取12个字节数据: Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.5.2 测试结果

4.6 fseek 定位

库函数 `fseek()` 的作用类似于 2.7 小节所学习的系统调用 `lseek()`，用于设置文件读写位置偏移量，`lseek()` 用于文件 I/O，而库函数 `fseek()` 则用于标准 I/O，其函数原型如下所示：

```
#include <stdio.h>
```

```
int fseek(FILE *stream, long offset, int whence);
```

函数参数和返回值含义如下：

stream: FILE 指针。

offset: 与 `lseek()` 函数的 `offset` 参数意义相同。

whence: 与 `lseek()` 函数的 `whence` 参数意义相同。

返回值: 成功返回 0；发生错误将返回 -1，并且会设置 `errno` 以指示错误原因；与 `lseek()` 函数的返回值意义不同，这里要注意！

调用库函数 `fread()`、`fwrite()` 读写文件时，文件的读写位置偏移量会自动递增，使用 `fseek()` 可手动设置文件当前的读写位置偏移量。

譬如将文件的读写位置移动到文件开头处：

```
fseek(file, 0, SEEK_SET);
```

将文件的读写位置移动到文件末尾：

```
fseek(file, 0, SEEK_END);
```

将文件的读写位置移动到 100 个字节偏移量处：

```
fseek(file, 100, SEEK_SET);
```

使用示例

示例代码 4.6.1 使用 `fseek()` 调整文件读写位置

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp = NULL;
    char rd_buf[100] = {0};
    char wr_buf[] = "正点原子 http://www.openedv.com/forum.php\n";
    int ret;

    /* 打开文件 */
    if (NULL == (fp = fopen("./test_file", "w+"))) {
        perror("fopen error");
        exit(-1);
    }
    printf("文件打开成功!\n");

    /* 写文件 */
    if (sizeof(wr_buf) >
        fwrite(wr_buf, 1, sizeof(wr_buf), fp)) {
```

```
        printf("fwrite error\n");
        fclose(fp);
        exit(-1);
    }
    printf("数据写入成功!\n");

    /* 将读写位置移动到文件头部 */
    if (0 > fseek(fp, 0, SEEK_SET)) {
        perror("fseek error");
        fclose(fp);
        exit(-1);
    }

    /* 读文件 */
    if (sizeof(wr_buf) >
        (ret = fread(rd_buf, 1, sizeof(wr_buf), fp))) {
        printf("fread error\n");
        fclose(fp);
        exit(-1);
    }

    printf("成功读取%d 个字节数据: %s\n", ret, rd_buf);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

程序中首先调用 `fopen()` 打开当前目录下的 `test_file` 文件, 参数 `mode` 设置为 `"w+"`; 接着调用 `fwrite()` 将 `wr_buf` 缓冲区中的字符串数据 "正点原子 <http://www.openedv.com/forum.php>" 写入到文件中; 由于调用了 `fwrite()`, 所以此时的读写位置已经发生了改变, 不再是文件头部, 所以程序中调用了 `fseek()` 将读写位置移动到了文件头, 接着调用 `fread()` 从文件头部开始读取刚写入的数据, 读取成功之后打印出信息。

运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
数据写入成功!
成功读取46个字节数据: 正点原子http://www.openedv.com/forum.php

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
正点原子http://www.openedv.com/forum.php
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.6.1 测试结果

ftell()函数

库函数 `ftell()` 可用于获取文件当前的读写位置偏移量, 其函数原型如下所示:

```
#include <stdio.h>
```

```
long ftell(FILE *stream);
```

参数 `stream` 指向对应的文件, 函数调用成功将返回当前读写位置偏移量; 调用失败将返回-1, 并会设置 `errno` 以指示错误原因。

我们可以通过 `fseek()` 和 `ftell()` 来计算出文件的大小, 示例代码如下所示:

示例代码 4.6.2 使用 `fseek()` 和 `ftell()` 函数获取文件大小

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    FILE *fp = NULL;
```

```
    int ret;
```

```
    /* 打开文件 */
```

```
    if (NULL == (fp = fopen("./testApp.c", "r"))) {
```

```
        perror("fopen error");
```

```
        exit(-1);
```

```
    }
```

```
    printf("文件打开成功!\n");
```

```
    /* 将读写位置移动到文件末尾 */
```

```
    if (0 > fseek(fp, 0, SEEK_END)) {
```

```
        perror("fseek error");
```

```
        fclose(fp);
```

```
        exit(-1);
```

```
    }
```

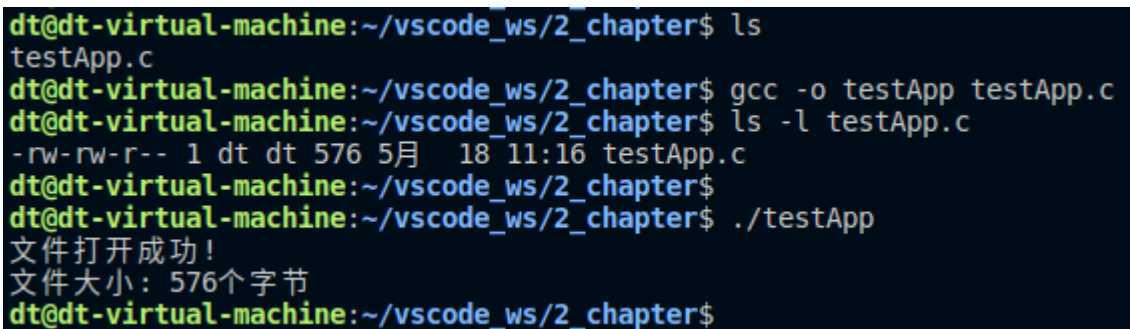
```
/* 获取当前位置偏移量 */
if (0 > (ret = ftell(fp))) {
    perror("ftell error");
    fclose(fp);
    exit(-1);
}

printf("文件大小: %d 个字节\n", ret);

/* 关闭文件 */
fclose(fp);
exit(0);
}
```

首先打开当前目录下的 testApp.c 文件, 将文件的读写位置移动到文件末尾, 然后再获取当前的位置偏移量, 也就得到了整个文件的大小。

运行测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l testApp.c
-rw-rw-r-- 1 dt dt 576 5月 18 11:16 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
文件大小: 576个字节
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.6.2 测试结果

从上图可知, 程序计算出的文件大小与 ls 命令查看到的文件大小是一致的。

4.7 检查或复位状态

调用 fread() 读取数据时, 如果返回值小于参数 nmemb 所指定的值, 表示发生了错误或者已经到了文件末尾 (文件结束 end-of-file), 但 fread() 无法具体确定是哪一种情况; 在这种情况下, 可以通过判断错误标志或 end-of-file 标志来确定具体的情况。

4.7.1 feof() 函数

库函数 feof() 用于测试参数 stream 所指文件的 end-of-file 标志, 如果 end-of-file 标志被设置了, 则调用 feof() 函数将返回一个非零值, 如果 end-of-file 标志没有被设置, 则返回 0。

其函数原型如下所示:

```
#include <stdio.h>
```

```
int feof(FILE *stream);
```

当文件的读写位置移动到了文件末尾时, end-of-file 标志将会被设置。

```
if (feof(file)) {
```

```
/* 到达文件末尾 */
```



```
}  
else {  
    /* 未到达文件末尾 */  
}
```

4.7.2 ferror()函数

库函数 `ferror()` 用于测试参数 `stream` 所指文件的错误标志, 如果错误标志被设置了, 则调用 `ferror()` 函数将返回一个非零值, 如果错误标志没有被设置, 则返回 0。

其函数原型如下所示:

```
#include <stdio.h>
```

```
int ferror(FILE *stream);
```

当对文件的 I/O 操作发生错误时, 错误标志将会被设置。

```
if (ferror(file)) {  
    /* 发生错误 */  
}  
else {  
    /* 未发生错误 */  
}
```

4.7.3 clearerr()函数

库函数 `clearerr()` 用于清除 end-of-file 标志和错误标志, 当调用 `feof()` 或 `ferror()` 校验这些标志后, 通常需要清除这些标志, 避免下次校验时使用到的是上一次设置的值, 此时可以手动调用 `clearerr()` 函数清除标志。

`clearerr()` 函数原型如下所示:

```
#include <stdio.h>
```

```
void clearerr(FILE *stream);
```

此函数没有返回值, 调用将总是会成功!

对于 end-of-file 标志, 除了使用 `clearerr()` 显式清除之外, 当调用 `fseek()` 成功时也会清除文件的 end-of-file 标志。

使用示例

示例代码 4.7.1 `clearerr()` 函数使用示例

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    FILE *fp = NULL;  
    char buf[20] = {0};  
  
    /* 打开文件 */  
    if (NULL == (fp = fopen("./testApp.c", "r"))) {
```

```
    perror("fopen error");
    exit(-1);
}

printf("文件打开成功!\n");

/* 将读写位置移动到文件末尾 */
if (0 > fseek(fp, 0, SEEK_END)) {
    perror("fseek error");
    fclose(fp);
    exit(-1);
}

/* 读文件 */
if (10 > fread(buf, 1, 10, fp)) {
    if (feof(fp))
        printf("end-of-file 标志被设置,已到文件末尾!\n");

    clearerr(fp); //清除标志
}

/* 关闭文件 */
fclose(fp);
exit(0);
}
```

4.8 格式化 I/O

在前面编写的测试代码中,会经常使用到库函数 `printf()`用于输出程序中的打印信息, `printf()`函数可将格式化数据写入到标准输出,所以通常称为格式化输出。除了 `printf()`之外,格式化输出还包括: `fprintf()`、`dprintf()`、`sprintf()`、`snprintf()`这4个库函数。

除了格式化输出之外,自然也有格式化输入,从标准输入中获取格式化数据,格式化输入包括: `scanf()`、`fscanf()`、`sscanf()`这三个库函数,那么本小节将向大家介绍 C 语言库函数的格式化 I/O。

4.8.1 格式化输出

C 库函数提供了 5 个格式化输出函数,包括: `printf()`、`fprintf()`、`dprintf()`、`sprintf()`、`snprintf()`,其函数定义如下所示:

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *buf, const char *format, ...);
int snprintf(char *buf, size_t size, const char *format, ...);
```

可以看到,这 5 个函数都是可变参数函数,它们都有一个共同的参数 `format`,这是一个字符串,称为格式控制字符串,用于指定后续的参数如何进行格式转换,所以才把这些函数称为格式化输出,因为它们可以以调用者指定的格式进行转换输出;学习这些函数的重点就是掌握这个格式控制字符串 `format` 的书写格式以及它们所代表的意义,稍后介绍 `format` 参数的格式。

每个函数除了固定参数之外,还可携带 0 个或多个可变参数。

`printf()`函数用于将格式化数据写入到标准输出;`dprintf()`和 `fprintf()`函数用于将格式化数据写入到指定的文件中,两者不同之处在于,`fprintf()`使用 `FILE` 指针指定对应的文件、而 `dprintf()`则使用文件描述符 `fd` 指定对应的文件;`sprintf()`、`snprintf()`函数可将格式化的数据存储在用户指定的缓冲区 `buf` 中。

printf()函数

前面章节内容编写的示例代码中多次使用了该函数,用于将程序中的字符串信息输出显示到终端(也就是标准输出),相信各位读者学习 C 语言时肯定用过该函数,它是一个可变参数函数,除了一个固定参数 `format` 外,后面还可携带 0 个或多个参数。

函数调用成功返回打印输出的字符数;失败将返回一个负值!

打印“Hello World”:

```
printf("Hello World!\n");
```

打印数字 5:

```
printf("%d\n", 5);
```

fprintf()函数

`fprintf()`可将格式化数据写入到由 `FILE` 指针指定的文件中,譬如将字符串“Hello World”写入到标准错误:

```
fprintf(stderr, "Hello World!\n");
```

向标准错误写入数字 5:

```
fprintf(stderr, "%d\n", 5);
```

函数调用成功返回写入到文件中的字符数;失败将返回一个负值!

dprintf()函数

`dprintf()`可将格式化数据写入到由文件描述符 `fd` 指定的文件中,譬如将字符串“Hello World”写入到标准错误:

```
dprintf(STDERR_FILENO, "Hello World!\n");
```

向标准错误写入数字 5:

```
dprintf(STDERR_FILENO, "%d\n", 5);
```

函数调用成功返回写入到文件中的字符数;失败将返回一个负值!

sprintf()函数

`sprintf()`函数将格式化数据存储在由参数 `buf` 所指定的缓冲区中,譬如将字符串“Hello World”存放在缓冲区中:

```
char buf[100];
```

```
sprintf(buf, "Hello World!\n");
```

当然这种用法并没有意义,事实上,我们一般会使用这个函数进行格式化转换,并将转换后的字符串存放在缓冲区中,譬如将数字 100 转换为字符串“100”,将转换后得到的字符串存放在 `buf` 中:

```
char buf[20] = {0};
```

```
sprintf(buf, "%d", 100);
```

`sprintf()`函数会在字符串尾端自动加上一个字符串终止字符‘\0’。

需要注意的是, `sprintf()` 函数可能会造成由参数 `buf` 指定的缓冲区溢出, 调用者有责任确保该缓冲区足够大, 因为缓冲区溢出会造成程序不稳定甚至安全隐患!

函数调用成功返回写入到 `buf` 中的字节数; 失败将返回一个负值!

snprintf()函数

`sprintf()` 函数可能会发生缓冲区溢出的问题, 存在安全隐患, 为了解决这个问题, 引入了 `snprintf()` 函数; 在该函数中, 使用参数 `size` 显式的指定缓冲区的大小, 如果写入到缓冲区的字节数大于参数 `size` 指定的大小, 超出的部分将会被丢弃! 如果缓冲区空间足够大, `snprintf()` 函数就会返回写入到缓冲区的字符数, 与 `sprintf()` 函数相同, 也会在字符串末尾自动添加终止字符 `\0`。

若发生错误, `snprintf()` 将返回一个负值!

格式控制字符串 format

接下来重点学习以上 5 个函数中的 `format` 参数应该怎么写, 把这个参数称为格式控制字符串, 顾名思义, 首先它是一个字符串的形式, 其次它能够控制后续变参的格式转换。

格式控制字符串由两部分组成: 普通字符 (非 % 字符) 和转换说明。普通字符会进行原样输出, 每个转换说明都会对应后续的一个参数, 通常有几个转换说明就需要提供几个参数 (除固定参数之外的参数), 使之一一对应, 用于控制对应的参数如何进行转换。如下所示:

```
printf("转换说明 1 转换说明 2 转换说明 3", arg1, arg2, arg3);
```

这里只是以 `printf()` 函数举个例子, 实际上并不这样用。三个转换说明与参数进行一一对应, 按照顺序方式一一对应。

每个转换说明都是以 % 字符开头, 其格式如下所示 (使用 [] 括起来的部分是可选的):

```
%[flags][width][.precision][length]type
```

flags: 标志, 可包含 0 个或多个标志;

width: 输出最小宽度, 表示转换后输出字符串的最小宽度;

precision: 精度, 前面有一个点号 ".";

length: 长度修饰符;

type: 转换类型, 指定待转换数据的类型。

可以看到, 只有 % 和 `type` 字段是必须的, 其余都是可选的。下面分别对这些字段进行介绍。

(一)、type 类型

首先说明 `type` (类型), 因为类型是格式控制字符串的重中之重, 是必不可少的组成部分, 其它的字段都是可选的, `type` 用于指定输出数据的类型, `type` 字段使用一个字符 (字母字符) 来表示, 可取值如下:

字符	对应的数据类型	含义	示例说明
d/i	int	输出有符号十进制表示的整数, i 是老式写法	<code>printf("%d\n", 123);</code> 输出:123
o	unsigned int	输出无符号八进制表示的整数 (默认不输出前缀 0, 可在 <code>type</code> 字段指定标志 # 使其输出前缀 0)	<code>printf("%o\n", 123);</code> 输出:173
u	unsigned int	输出无符号十进制表示的整数	<code>printf("%u\n", 123);</code> 输出:123
x/X	unsigned int	输出无符号十六进制表示的整数, x 和 X 的区别在于字母的大小写问题 (x 对应的是 abcdef, X 对应的是	<code>printf("%x\n", 123);</code> 输出:7b <code>printf("%X\n", 123);</code> 输出:7B

		ABCDEF) ; 不输出前缀 0x 或 0X, 可在 type 字段指定标志#使其输出前缀。	
f/F	double	输出浮点数, 单精度浮点数类型和双精度浮点数类型都可以使用, f 和 F 之间的区别就不去管了, 一般表示浮点数使用 f 即可。在没指定精度的情况下, 默认保留小数点后 6 位数字。	printf("%f\n", 520.1314); 输出:520.131400 printf("%F\n", 520.1314); 输出:520.131400
e/E	double	输出以科学计数法表示的浮点数, 使用指数(Exponent)表示浮点数, 此处 e 和 E 的区别在于以科学计数法表示时, 字母“e”的大小写问题。	printf("%e\n", 520.1314); 输出:5.201314e+02 printf("%E\n", 520.1314); 输出:5.201314E+02
g	double	根据数值的长度, 选择以最短的方式输出, %f/%e	printf("%g %g\n", 0.000000123, 0.123); 输出:1.23e-07 0.123
G	double	根据数值的长度, 选择以最短的方式输出, %F/%E	printf("%G %G\n", 0.000000123, 0.123); 输出:1.23E-07 0.123
s	char *	字符串, 输出字符串中的字符直至终止字符'\0'	printf("%s\n", "Hello World"); 输出:Hello World
p	void *	输出十六进制表示的指针	printf("%p\n", "Hello World"); 输出:0x400624
c	char	字符型, 可以把输入的数字按照 ASCII 码相应转换为对应的字符输出	printf("%c\n", 64); 输出:A

表 4.8.1 转换说明中的 type 字段介绍

(二)、flags

flags 规定输出样式, %后面可以跟 0 个或多个以下标志:

字符	名称	作用
#	井号	type 等于 o 时, 输出字符串增加前缀 0。 type 等于 x 或 X 时, 输出字符串增加前缀 0x 或 0X。 type 等于 a、A、e、E、f、F、g 和 G 其中之一时, 在默认情况下, 只有输出小数部分时才会输出小数点, 如果使用 .0 控制不输出小数部分, 那么小数点是不会输出的, 然而在使用了标志#的情况下, 输出结果始终包含小数点; type 等于 g 和 G 时, 保留尾部的 0。
0	数字 0	当 type 不等于 c 或 s 时 (也就是输出数字时, 包括浮点数和整数), 在输出字符串前面补 0, 直到占满指定的最小输出宽度 (位数)。譬如输出正数 100, 指定的最小输出宽度是 5, 那么最终就会输出 00100。如果没有指定标志 0, 则默认使用空格占满指定的最小输出宽度。

-	减号	输出字符串默认情况下是右对齐的，不足最小输出宽度时在左边填充格或 0；使用了-标志，则会变成左对齐，然后在右边填充格，如果同时指定了标志 0 和标志-，则标志-会覆盖标志 0。
" "	空格	输出正数时在前面加一个空格，输出负数时，前面加一个负号-。
+	加号	默认情况下，只有输出负数时，才会输出负号-；正数前面是没有正号+的；而使用了标志+后，输出的数字前面都带有符号（正数为+、负数为-）；如果同时指定了标志+和标志" "（空格），则标志+会覆盖标志空格。

表 4.8.2 转换说明中的 flags 字段介绍

三、width

最小的输出宽度，用十进制数来表示输出的最小位数，若实际的输出位数大于指定的输出的最小位数，则以实际的位数进行输出，若实际的位数小于指定输出的最小位数，则可按照指定的 flags 标志补 0 或补空格。

width 的可能取值如下：

width	描述	示例
数字	十进制正数	<code>printf("%06d", 1000);</code> 输出: 001000
*	星号，不显示指出最小输出宽度，而是以星号代替，会在参数列表中指定	<code>printf("%0*d", 6, 1000);</code> 输出: 001000

表 4.8.3 转换说明中的 width 字段介绍

四、precision 精度

精度字段以点号"."开头，后跟一个十进制正数，可取值如下：

.precision	描述
数字	<p>十进制正数</p> <p>①对于整形(type 等于 d、i、o、u、x 和 X)，precision 表示输出的最小的数字个数，不足补前导零，超过不截断。这里要注意：是数字的个数、与 width 字段是有区别的，width 指的是整个输出字符串的最小位数（最小宽度），并不是数字的最小宽度，譬如： <code>printf("%8.5d\n", 100);</code> 输出: 00100（前面有 3 个空格）； 在这个例子中，width 字段为 8，表示需要整个字符串的输出长度为 8 个字符，.5 则表示数字部分位数最少为 5 个，不足 5 个则在前面补 0，所以 100 需要在前面补两个 0 才能满足这个要求；满足这个要求之后，接着需要使整个字符串长度为 8 个字符，那么只需要在前面补 3 个空格即可（这里是左对齐的情况）！ 会忽略 flags 字段的标志 0，意味着在这种情况下，指定标志 0 和不指定标志 0 都是一样的效果。</p> <p>②对于浮点型(type 等于 a、A、e、E、f、F)，precision 表示小数点后数字的个数，也就是浮点数精度；默认为六位，不足补后置 0，超过则截断。譬如： <code>printf("%.8f\n", 520.1314);</code> 输出:520.13140000</p> <p>③type 等于 g、G 时，表示最大有效位数；</p> <p>④对于字符串(type 等于 s)，precision 表示输出字符串中最大可输出的字符数，不足正常输出，超过则截断。譬如： <code>printf("%.5s\n", "hello world");</code> 输出:hello；超过 5 个字符的部分被丢弃！</p>

*	以星号代替十进制数字, 类似于 width 字段中的*, 表示在参数列表中指定; 譬如: <code>printf("%.5s\n", 5, "hello world");</code> 输出:hello
---	---

表 4.8.4 转换说明中的 precision 字段介绍

⑤、length 长度修饰符

长度修饰符指明待转换数据的长度, 因为 type 字段指定的类型只有 int、unsigned int 以及 double 等几种数据类型, 但是 C 语言内置的数据类型不止这几种, 譬如 16bit 的 short、unsigned short, 8bit 的 char、unsigned char, 也有 64bit 的 long long 等, 为了能够区别不同长度的数据类型, 于是乎, 长度修饰符 (length) 应运而生, 成为转换说明的一部分。

length 长度修饰符也是使用字符 (字母字符) 来表示, 结合 type 字段以确定不同长度的数据类型, 如下所示:

	type					
length	d、i	u、o、x、X	f、F、e、E、g、G	c	s	p
(none)	int	unsigned int	double	char	char *	void *
hh	signed char	unsigned char				
h	short int	unsigned short int				
l	long int	unsigned long int		wint_t	wchar_t	
ll	long long int	unsigned long long int				
L			long double			
j	intmax_t	uintmax_t				
z	size_t	ssize_t				
t	ptrdiff_t	ptrdiff_t				

表 4.8.5 length 长度修饰符说明

譬如:

```
printf("%hd\n", 12345); //将数据以 short int 类型进行转换
printf("%ld\n", 12345); //将数据以 long int 类型进行转换
printf("%lld\n", 12345); //将数据以 long long int 类型进行转换
```

关于格式控制字符串 format 就给大家介绍完了, 这种东西不用去记, 需要时查询即可! 需要说明的是, 转换说明的描述信息需要和与之相对应的参数对应的数据类型要进行匹配, 如果不匹配通常会编译报错或者警告!

示例代码

前面为了说明格式控制字符串 format 的输出效果, 我们使用了 printf() 函数进行演示, 其它格式化输出函数也是一样, 接下来我们编写一个简单的测试程序, 对上面学习的内容进行练习。

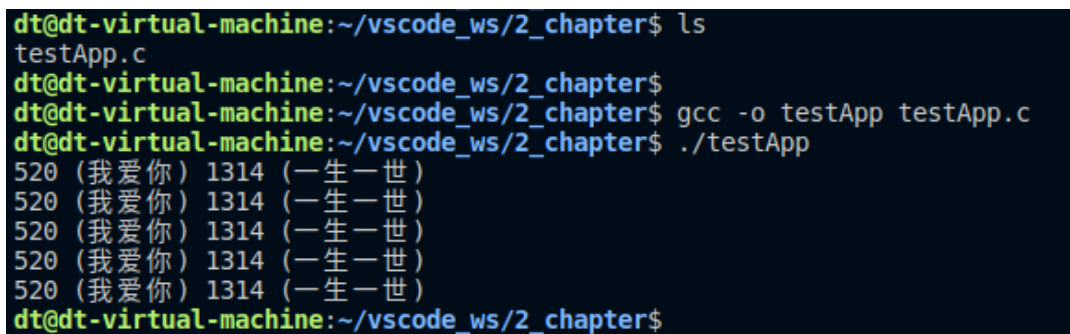
示例代码 4.8.1 格式化输出函数使用练习

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char buf[50] = {0};
```

```
printf("%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");  
fprintf(stdout, "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");  
dprintf(STDOUT_FILENO, "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");  
  
sprintf(buf, "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");  
printf("%s", buf);  
  
memset(buf, 0x00, sizeof(buf));  
snprintf(buf, sizeof(buf), "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");  
printf("%s", buf);  
  
exit(0);  
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
520 (我爱你) 1314 (一生一世)  
520 (我爱你) 1314 (一生一世)  
520 (我爱你) 1314 (一生一世)  
520 (我爱你) 1314 (一生一世)  
520 (我爱你) 1314 (一生一世)  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.8.1 运行结果

关于格式化输出这几个函数其实用法上比较简单,主要是需要掌握格式控制字符串 format 参数的写法,对后续参数列表中不同类型的数据搭配不同的格式控制字符,以实现转换输出、并且控制输出样式。

本小节所学内容,大家可以多多练习!

4.8.2 格式化输入

C 库函数提供了 3 个格式化输入函数,包括: scanf()、fscanf()、sscanf(),其函数定义如下所示:

```
#include <stdio.h>  
  
int scanf(const char *format, ...);  
int fscanf(FILE *stream, const char *format, ...);  
int sscanf(const char *str, const char *format, ...);
```

可以看到,这 3 个格式化输入函数也是可变参函数,它们都有一个共同的参数 format,同样也称为格式控制字符串,用于指定输入数据如何进行格式转换,与格式化输出函数中的 format 参数格式相似,但也有所不同。

每个函数除了固定参数之外,还可携带 0 个或多个可变参数。

scanf()函数可将用户输入(标准输入)的数据进行格式化转换;fscanf()函数从 FILE 指针指定文件中读取数据,并将数据进行格式化转换;sscanf()函数从参数 str 所指向的字符串中读取数据,并将数据进行格式化转换。

scanf()函数

相对于 `printf` 函数, `scanf` 函数就简单得多。`scanf()`函数的功能与 `printf()`函数正好相反, 执行格式化输入功能; 即 `scanf()`函数将用户输入(标准输入)的数据进行格式化转换并进行存储, 它从格式化控制字符串 `format` 参数的最左端开始, 每遇到一个转换说明便将其与下一个输入数据进行“匹配”, 如果二者匹配则继续, 否则结束对后面输入的处理。而每遇到一个转换说明, 便按该转换说明所描述的格式对其后的输入数据进行转换, 然后将转换得到的数据存储于与其对应的输入地址中。以此类推, 直到对整个输入数据的处理结束为止。

从函数原型可以看出, `scanf()`函数也是一个“可变参数函数”, 除第一个参数 `format` 之外, `scanf()`函数还可以有若干个输入地址(指针), 这些指针指向对应的缓冲区, 用于存储格式化转换后的数据; 且对于每一个输入地址, 在格式控制字符串 `format` 参数中都必须有一个转换说明与之一一对应。即从 `format` 字符串的左端第 1 个转换说明对应第 1 个输入地址, 第 2 个格式说明符对应第 2 个输入地址, 第 3 个格式说明符对应第 3 个输入地址, 以此类推。譬如:

```
int a, b, c;  
scanf("%d %d %d", &a, &b, &c);
```

当程序中调用 `scanf()`的时候, 终端会被阻塞, 等待用户输入数据, 此时我们可以通过键盘输入一些字符, 譬如数字、字母或者其它字符, 输入完成按回车即可! 接着来 `scanf()`函数就会对用户输入的数据进行格式转换处理。

函数调用成功后, 将返回成功匹配和分配的输入项的数量; 如果较早匹配失败, 则该数目可能小于所提供的数目, 甚至为零。发生错误则返回负值。

fscanf()函数

`fscanf()`函数从指定文件中读取数据, 作为格式转换的输入数据, 文件通过 `FILE` 指针指定, 所以它有两个固定参数, `FILE` 指针和格式控制字符串 `format`。譬如从标准输入文件中读取数据进行格式化转换:

```
int a, b, c;  
fscanf(stdin, "%d %d %d", &a, &b, &c);
```

此时它的作用与 `scanf()`就是相同的, 因为标准输入文件的数据就是用户输入的数据, 譬如通过键盘输入的数据。

函数调用成功后, 将返回成功匹配和分配的输入项的数量; 如果较早匹配失败, 则该数目可能小于所提供的数目, 甚至为零。发生错误则返回负值。

sscanf()函数

`sscanf()`将从参数 `str` 所指向的字符串缓冲区中读取数据, 作为格式转换的输入数据, 所以它也有两个固定参数, 字符串 `str` 和格式控制字符串 `format`, 譬如:

```
char *str = "5454 hello";  
char buf[10];  
int a;  
  
sscanf(str, "%d %s", &a, buf);
```

函数调用成功后, 将返回成功匹配和分配的输入项的数量; 如果较早匹配失败, 则该数目可能小于所提供的数目, 甚至为零。发生错误则返回负值。

格式控制字符串 format

本小节的重点依然是这个 `format` 参数的格式, 与格式化输出函数中的 `format` 参数格式、写法上比较相似, 但也有一些区别。`format` 字符串包含一个或多个转换说明, 每一个转换说明都是以百分号“%”或者“%n\$”开头 (`n` 是一个十进制数字), 关于“%n\$”这种开头的转换说明就不介绍了, 实际上用的不多。

以百分号开头的转换说明一般格式如下:

```
%[*][width][length]type
```

```
%[m][width][length]type
```

%后面可选择性添加星号*或字母 m, 如果添加了星号*, 格式化输入函数会按照转换说明的指示读取输入, 但是丢弃输入, 意味着不需要对转换后的结果进行存储, 所以也就不需要提供相应的指针参数。

如果添加了 m, 它只能与%s、%c 以及%[一起使用, 调用者无需分配相应的缓冲区来保存格式转换后的数据, 原因在于添加了 m, 这些格式化输入函数内部会自动分配足够大小的缓冲区, 并将缓冲区的地址值通过与该格式转换相对应的指针参数返回出来, 该指针参数应该是指向 char *变量的指针。随后, 当不再需要此缓冲区时, 调用者应调用 free()函数来释放此缓冲区。

譬如:

```
char *buf;

scanf("%ms", &buf);

.....

free(buf);
```

介绍了星号*和字母 m 之后, 再来看看转换说明的格式, 中括号[]表示的部分是可选的, 所以可知, 与格式化输出函数中的 format 参数一样, 只有 type 字段是必须的。

- **width:** 最大字符宽度;
- **length:** 长度修饰符, 与格式化输出函数的 format 参数中的 length 字段意义相同。
- **type:** 指定输入数据的类型。

我们先来看看 type 字段。

(-)type (类型)

此 type 字段与格式化输出函数中的 format 参数的 type 字段是同样的意义, 用于指定输入数据的类型, 如下所示:

字符	对应的数据类型	含义	示例
d	int	匹配一个有符号十进制整数	int a; scanf("%d", &a); 用户输入: 100
i	int	匹配一个有符号整数, 既可以是十进制表示、也可以是八进制或十六进制方式表示, 譬如整数以 0x 或 0X 开头, 则认为是 16 进制, 以 0 开头则认为是八进制, 否则认为是十进制。	int a; scanf("%i", &a); 用户输入: 0x100
o	unsigned int	匹配一个无符号八进制整数	unsigned int a; scanf("%o", &a); 用户输入: 0100
u	unsigned int	匹配一个无符号十进制整数	unsigned int a; scanf("%u", &a); 用户输入: 100
x	unsigned int	匹配一个无符号十六进制整数, 数字以 0x 开头	unsigned int a; scanf("%x", &a); 用户输入: 0x100

X	unsigned int	匹配一个无符号十六进制整数, 数字以 0X 开头	unsigned int a; scanf("%x", &a); 用户输入: 0X100
f	float	匹配一个带符号的浮点数	float a; scanf("%f", &a); 用户输入: 10.123
e	float	等效于 f	
E	float	等效于 f	
g	float	等效于 f	
a	float	等效于 f	
s	char *	匹配字符串, 不匹配空白字符 (包括空格、制表符、换行符), 空白字符作为字符串的分隔符, 所以就是匹配一系列非空白字符。所以存放字符串的缓冲区必须足够大, 会自动添加终止字符"\0"	char buf[100]; scanf("%s", buf); 用户输入: HelloWorld
c	char	匹配一个字符	char c; scanf("%c", &c); 用户输入: A
p	void *	匹配一个指针值	void *a; scanf("%p", &a); 用户输入: 123456
[char *	匹配一组字符序列集合, 以字符串形式存储。 譬如%[a-z]匹配 a 到 z 之间的所有字符 (包括起始字符 a 和结束字符 z), 譬如%[0-9]则表示匹配数字 0 到 9 之间所有数字字符; 减号-是一个连字符, 放在两个字符 (一个起始字符和一个结束字符) 之间, 本身并不参与匹配, 如果需要匹配减号-, 则可将其放在中括号[]旁边, 譬如%[-a-z], 表示匹配减号、以及数字 0 到 9 这些字符; 如果要排除匹配这些字符, 可以使用排除符^, 将其放在最前面, 譬如%[^a-z-], 表示匹配除 a 到 z 这些字符以及-之外的字符。	char buf[100]; scanf("%[a-z0-9]", buf); 匹配字母 a 到 z 以及数字 0 到 9 这些字符。 用户输入: 123abc

表 4.8.6 type 类型描述

(二)、width 最大字符宽度

是一个十进制表示的整数, 用于指定最大字符宽度, 当达到此最大值或发现不匹配的字符时 (以先发生者为准), 字符的读取将停止。大多数 type 类型会丢弃初始的空白字符, 并且这些丢弃的字符不会计入最大字符宽度。对于字符串转换来说, scanf()会在字符串末尾自动添加终止符"\0", 最大字符宽度中不包括此终止符。

譬如调用 scanf()函数如下:

```
scanf("%4s", buf); //匹配字符串, 字符串长度不超过 4 个字符
```

用户输入 abcdefg, 按回车, 那么只能将 adcd 作为一个字符串存储在 buf 数组中。

(三)length 长度修饰符

与格式化输出函数的格式控制字符串 `format` 中的 `length` 字段意义相同, 用于对 `type` 字段进行修饰, 扩展识别更多不同长度的数据类型。如下所示:

	type				
length	d、i	u、o、x、X	e、f、g	c	s
(none)	int	unsigned int	float	char	char *
h	short int	unsigned short int			
hh	signed char	unsigned char			
j	intmax_t	uintmax_t			
l	long int	unsigned long int	double	wchar_t	wchar_t *
L	long long int	unsigned long long int	long double		

表 4.8.7 length 长度修饰符

譬如:

```
scanf("%hd", var); //匹配 short int 类型数据
scanf("%hhd", var); //匹配 signed char 类型数据
scanf("%ld", var); //匹配 long int 类型数据
scanf("%f", var); //匹配 float 类型数据
scanf("%lf", var); //匹配 double 类型数据
scanf("%Lf", var); //匹配 long double 类型数据
```

关于格式化输入函数的 `format` 参数就介绍到这里了, 接下来编写一个简单地示例进行测试。

使用示例

示例代码 4.8.2 scanf()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int a;
    float b;
    char *str;

    printf("请输入一个整数:\n");
    scanf("%d", &a);
    printf("你输入的整数为: %d\n", a);

    printf("请输入一个浮点数:\n");
    scanf("%f", &b);
    printf("你输入的浮点数为: %f\n", b);

    printf("请输入一个字符串:\n");
    scanf("%ms", &str);
    printf("你输入的字符串为: %s\n", str);
```

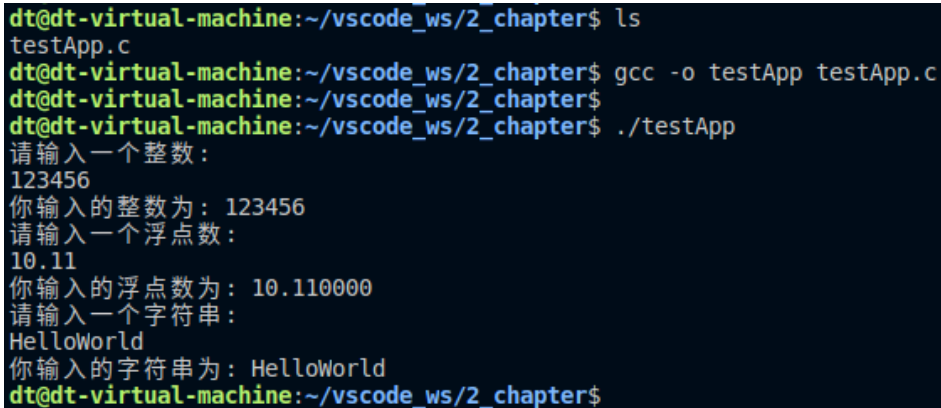
```
free(str);    //释放字符串占用的内存空间

exit(0);

}
```

当程序中调用 `scanf()` 之后, 终端就会被阻塞、等待用户输入数据, 当我们输入完成之后, 按回车即可! 第三个 `scanf()` 函数调用中, 使用 `%m`, 所以我们不需要提供存放字符串的缓冲区, `scanf()` 函数内部会分配缓冲区, 并将缓冲区地址存放在 `str` 这个我们给定的 `char` 指针变量中。使用完之后记得调用 `free()` 释放内存即可。

编译测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入一个整数:
123456
你输入的整数为: 123456
请输入一个浮点数:
10.11
你输入的浮点数为: 10.110000
请输入一个字符串:
HelloWorld
你输入的字符串为: HelloWorld
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.8.2 测试结果

4.8.3 小结

本小节 (4.8) 对标准 I/O 中的格式化 I/O 做了比较详细的介绍, 相信大家对此都比较熟悉了, 当然, 重要的是需要灵活使用它们。关于格式化 I/O 还存在一些比较细节的问题需要我们注意, 主要是围绕缓冲的问题, 关于缓冲的问题将会在下一小节向大家介绍!

4.9 I/O 缓冲

出于速度和效率的考虑, 系统 I/O 调用 (即文件 I/O, `open`、`read`、`write` 等) 和标准 C 语言库 I/O 函数 (即标准 I/O 函数) 在操作磁盘文件时会对数据进行缓冲, 本小节将讨论文件 I/O 和标准 I/O 这两种 I/O 方式的数据缓冲问题, 并讨论其对应用程序性能的影响。

除此之外, 本小节还讨论了屏蔽或影响缓冲的一些技术手段, 以及直接 I/O 技术——绕过内核缓冲直接访问磁盘硬件。

4.9.1 文件 I/O 的内核缓冲

`read()` 和 `write()` 系统调用在进行文件读写操作的时候并不会直接访问磁盘设备, 而是仅仅在用户空间缓冲区和内核缓冲区 (kernel buffer cache) 之间复制数据。譬如调用 `write()` 函数将 5 个字节数据从用户空间内存拷贝到内核空间的缓冲区中:

```
write(fd, "Hello", 5);    //写入 5 个字节数据
```

调用 `write()` 后仅仅只是将这 5 个字节数据拷贝到了内核空间的缓冲区中, 拷贝完成之后函数就返回了, 在后面的某个时刻, 内核会将其缓冲区中的数据写入 (刷新) 到磁盘设备中, 所以由此可知, 系统调用 `write()` 与磁盘操作并不是同步的, `write()` 函数并不会等待数据真正写入到磁盘之后再返回。如果在此期间, 其它进

程调用 `read()` 函数读取该文件的这几个字节数据, 那么内核将自动从缓冲区中读取这几个字节数据返回给应用程序。

与此同理, 对于读文件而言亦是如此, 内核会从磁盘设备中读取文件的数据并存储到内核的缓冲区中, 当调用 `read()` 函数读取数据时, `read()` 调用将从内核缓冲区中读取数据, 直至把缓冲区中的数据读完, 这时, 内核会将文件的下一段内容读入到内核缓冲区中进行缓存。

我们把这个内核缓冲区就称为文件 I/O 的内核缓冲。这样的设计, 目的是为了提高文件 I/O 的速度和效率, 使得系统调用 `read()`、`write()` 的操作更为快速, 不需要等待磁盘操作 (将数据写入到磁盘或从磁盘读出数据), 磁盘操作通常是比较缓慢的。同时这一设计也更为高效, 减少了内核操作磁盘的次数, 譬如线程 1 调用 `write()` 向文件写入数据 "abcd", 线程 2 也调用 `write()` 向文件写入数据 "1234", 这样的话, 数据 "abcd" 和 "1234" 都被缓存在了内核的缓冲区中, 在稍后内核会将它们一起写入到磁盘中, 只发起一次磁盘操作请求; 加入没有内核缓冲区, 那么每一次调用 `write()`, 内核就会执行一次磁盘操作。

前面提到, 当调用 `write()` 之后, 内核稍后会将数据写入到磁盘设备中, 具体是什么时间点写入到磁盘, 这个其实是不确定的, 由内核根据相应的存储算法自动判断。

通过前面的介绍可知, 文件 I/O 的内核缓冲区自然是越大越好, Linux 内核本身对内核缓冲区的大小没有固定上限。内核会分配尽可能多的内核来作为文件 I/O 的内核缓冲区, 但受限于物理内存的总量, 如果系统可用的物理内存越多, 那自然对应的内核缓冲区也就越大, 操作越大的文件也要依赖于更大空间的内核缓冲。

4.9.2 刷新文件 I/O 的内核缓冲区

强制将文件 I/O 内核缓冲区中缓存的数据写入 (刷新) 到磁盘设备中, 对于某些应用程序来说, 可能是很有必要的, 例如, 应用程序在进行某操作之前, 必须要确保前面步骤调用 `write()` 写入到文件的数据已经真正写入到了磁盘中, 诸如一些数据库的日志进程。

联系到一个实际的使用场景, 当我们在 Ubuntu 系统下拷贝文件到 U 盘时, 文件拷贝完成之后, 通常在拔掉 U 盘之前, 需要执行 `sync` 命令进行同步操作, 这个同步操作其实就是将文件 I/O 内核缓冲区中的数据更新到 U 盘硬件设备, 所以如果在没有执行 `sync` 命令时拔掉 U 盘, 很可能就会导致拷贝到 U 盘中的文件遭到破坏!

控制文件 I/O 内核缓冲的系统调用

Linux 中提供了一些系统调用可用于控制文件 I/O 内核缓冲, 包括系统调用 `sync()`、`syncfs()`、`fsync()` 以及 `fdatasync()`。

(-)、`fsync()` 函数

系统调用 `fsync()` 将参数 `fd` 所指文件的内容数据和元数据写入磁盘, 只有在对磁盘设备的写入操作完成之后, `fsync()` 函数才会返回, 其函数原型如下所示:

```
#include <unistd.h>
```

```
int fsync(int fd);
```

参数 `fd` 表示文件描述符, 函数调用成功将返回 0, 失败返回 -1 并设置 `errno` 以指示错误原因。

前面提到了元数据这个概念, 元数据并不是文件内容本身的数据, 而是一些用于记录文件属性相关的数据信息, 譬如文件大小、时间戳、权限等等信息, 这里统称为文件的元数据, 这些信息也是存储在磁盘设备中的, 在 3.1 小节中介绍过。

使用示例

示例代码 4.9.1 实现了一个文件拷贝操作, 将源文件(当前目录下的 rfile 文件)的内容拷贝到目标文件中(当前目录下的 wfile 文件)。

示例代码 4.9.1 fsync()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define BUF_SIZE    4096
#define READ_FILE   "./rfile"
#define WRITE_FILE  "./wfile"

static char buf[BUF_SIZE];

int main(void)
{
    int rfd, wfd;
    size_t size;

    /* 打开源文件 */
    rfd = open(READ_FILE, O_RDONLY);
    if (0 > rfd) {
        perror("open error");
        exit(-1);
    }

    /* 打开目标文件 */
    wfd = open(WRITE_FILE, O_WRONLY | O_CREAT | O_TRUNC, 0664);
    if (0 > wfd) {
        perror("open error");
        exit(-1);
    }

    /* 拷贝数据 */
    while(0 < (size = read(rfd, buf, BUF_SIZE)))
        write(wfd, buf, size);

    /* 对目标文件执行 fsync 同步 */
    fsync(wfd);

    /* 关闭文件退出程序 */
}
```

```
close(rfd);
close(wfd);
exit(0);
}
```

代码没什么好说的,主要就是拷贝完成之后调用 `fsync()` 函数,对目标文件的数据进行了同步操作,整个操作完成之后 `close` 关闭源文件和目标文件、退出程序。

(二)、`fdatasync()` 函数

系统调用 `fdatasync()` 与 `fsync()` 类似,不同之处在于 `fdatasync()` 仅将参数 `fd` 所指文件的内容数据写入磁盘,并不包括文件的元数据;同样,只有在对磁盘设备的写入操作完成之后,`fdatasync()` 函数才会返回,其函数原型如下所示:

```
#include <unistd.h>
```

```
int fdatasync(int fd);
```

(三)、`sync()` 函数

系统调用 `sync()` 会将所有文件 I/O 内核缓冲区中的文件内容数据和元数据全部更新到磁盘设备中,该函数没有参数、也无返回值,意味着它不是对某一个指定的文件进行数据更新,而是刷新所有文件 I/O 内核缓冲区。其函数原型如下所示:

```
#include <unistd.h>
```

```
void sync(void);
```

在 Linux 实现中,调用 `sync()` 函数仅在所有数据已经写入到磁盘设备之后才会返回;然后在其它系统中,`sync()` 实现只是简单调度一下 I/O 传递,在动作未完成之后即可返回。

控制文件 I/O 内核缓冲的标志

调用 `open()` 函数时指定一些标志也可以影响到文件 I/O 内核缓冲,譬如 `O_DSYNC` 标志和 `O_SYNC` 标志,这些标志在 2.3 小节并未向大家介绍过,联系本小节所学内容,接下来向大家简单地介绍下。

(一)、`O_DSYNC` 标志

在调用 `open()` 函数时,指定 `O_DSYNC` 标志,其效果类似于在每个 `write()` 调用之后调用 `fdatasync()` 函数进行数据同步。譬如:

```
fd = open(filepath, O_WRONLY | O_DSYNC);
```

(二)、`O_SYNC` 标志

在调用 `open()` 函数时,指定 `O_SYNC` 标志,使得每个 `write()` 调用都会自动将文件内容数据和元数据刷新到磁盘设备中,其效果类似于在每个 `write()` 调用之后调用 `fsync()` 函数进行数据同步,譬如:

```
fd = open(filepath, O_WRONLY | O_SYNC);
```

对性能的影响

在程序中频繁调用 `fsync()`、`fdatasync()`、`sync()` (或者调用 `open` 时指定 `O_DSYNC` 或 `O_SYNC` 标志) 对性能的影响极大,大部分的应用程序是没有这种需求的,所以在大部分应用程序当中基本不会使用到。

4.9.3 直接 I/O: 绕过内核缓冲

从 Linux 内核 2.4 版本开始, Linux 允许应用程序在执行文件 I/O 操作时绕过内核缓冲区,从用户空间直接将数据传递到文件或磁盘设备,把这种操作也称为直接 I/O (direct I/O) 或裸 I/O (raw I/O)。

在有些情况下,这种操作通常是很有必要的,例如,某应用程序的作用是测试磁盘设备的读写速率,那么在这种应用需要下,我们就需要保证 read/write 操作是直接访问磁盘设备,而不经内核缓冲,如果不能得到这样的保证,必然会导致测试结果出现比较大的误差。

然后,对于大多数应用程序而言,使用直接 I/O 可能会大大降低性能,这是因为为了提高 I/O 性能,内核针对文件 I/O 内核缓冲区做了不少的优化,譬如包括按顺序预读取、在成簇磁盘块上执行 I/O、允许访问同一文件的多个进程共享高速缓存的缓冲区。如果应用程序使用直接 I/O 方式,将无法享受到这些优化措施所带来的性能上的提升,直接 I/O 只在一些特定的需求场合,譬如磁盘速率测试工具、数据库系统等。

我们可针对某一文件或块设备执行直接 I/O,要做到这一点,需要在调用 open() 函数打开文件时,指定 O_DIRECT 标志,该标志至 Linux 内核 2.4.10 版本开始生效,譬如:

```
fd = open(filepath, O_WRONLY | O_DIRECT);
```

直接 I/O 的对齐限制

因为直接 I/O 涉及到对磁盘设备的直接访问,所以在执行直接 I/O 时,必须要遵守以下三个对齐限制要求:

- 应用程序中用于存放数据的缓冲区,其内存起始地址必须以块大小的整数倍进行对齐;
- 写文件时,文件的位置偏移量必须是块大小的整数倍;
- 写入到文件的数据大小必须是块大小的整数倍。

如果不满足以上任何一个要求,调用 write() 均为以错误返回 Invalid argument。以上所说的块大小指的是磁盘设备的物理块大小 (block size),常见的块大小包括 512 字节、1024 字节、2048 以及 4096 字节,那我们如何确定磁盘分区的块大小呢?可以使用 tune2fs 命令进行查看,如下所示:

```
tune2fs -l /dev/sda1 | grep "Block size"
```

-l 后面指定了需要查看的磁盘分区,可以使用 df -h 命令查看 Ubuntu 系统的根文件系统所挂载的磁盘分区:

```
dt@dt-virtual-machine:~$ df -h
文件系统      容量  已用  可用  已用% 挂载点
udev          1.9G   0    1.9G   0% /dev
tmpfs         393M  12M  382M   3% /run
/dev/sda1     75G   7.8G  64G   11% /
tmpfs         2.0G   33M  1.9G   2% /dev/shm
tmpfs         5.0M   4.0K  5.0M   1% /run/lock
tmpfs         2.0G   0    2.0G   0% /sys/fs/cgroup
tmpfs         393M   60K  393M   1% /run/user/1000
dt@dt-virtual-machine:~$
```

图 4.9.1 查看根文件系统挂载的磁盘分区

通过上图可知,Ubuntu 系统的根文件系统挂载在/dev/sda1 磁盘分区下,接着下使用 tune2fs 命令查看该分区的块大小:

```
dt@dt-virtual-machine:~$ sudo tune2fs -l /dev/sda1 | grep "Block size"
[sudo] dt 的密码:
Block size:          4096
dt@dt-virtual-machine:~$
```

图 4.9.2 磁盘块大小

从上图可知/dev/sda1 磁盘分区的块大小为 4096 个字节。

直接 I/O 测试与普通 I/O 对比测试

接下来编写一个使用直接 I/O 方式写文件的测试程序和一个使用普通 I/O 方式写文件的测试程序,进行对比。

示例代码 4.9.2 演示了以直接 I/O 方式写文件的操作, 首先我们需要在程序开头处定义一个宏定义 `_GNU_SOURCE`, 原因在于后面 `open()` 函数需要指定 `O_DIRECT` 标志, 这个宏需要我们在程序中定义了 `O_DIRECT` 宏之后才能使用, 否则编译程序就会报错提示: `O_DIRECT` 未定义。

Tips: `_GNU_SOURCE` 宏可用于开启/禁用 Linux 系统调用和 glibc 库函数的一些功能、特性, 要打开这些特性, 需要在应用程序中定义该宏, 定义该宏之后意味着用户应用程序打开了所有的特性; 默认情况下, `_GNU_SOURCE` 宏并没有被定义, 所以当使用到它控制的一些特性时, 应用程序编译将会报错! 定义该宏的方式有两种:

- 直接在源文件中定义: `#define _GNU_SOURCE`
- gcc 编译时使用 `-D` 选项定义 `_GNU_SOURCE` 宏:

```
gcc -D_GNU_SOURCE -o testApp testApp.c
```

gcc 的 `-D` 选项可用于定义一个宏, 并且该宏定义在整个源码工程中都是生效的, 是一个全局宏定义。使用以上哪种方式都可以。

示例代码 4.9.2 直接 I/O 示例程序

```
/** 使用宏定义 O_DIRECT 需要在程序中定义宏 _GNU_SOURCE
** 不然提示 O_DIRECT 找不到 **/
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

/** 定义一个用于存放数据的 buf, 起始地址以 4096 字节进行对其 **/
static char buf[8192] __attribute__((aligned (4096)));

int main(void)
{
    int fd;
    int count;

    /* 打开文件 */
    fd = open("./test_file",
              O_WRONLY | O_CREAT | O_TRUNC | O_DIRECT,
              0664);
    if (0 > fd) {
        perror("open error");
        exit(-1);
    }

    /* 写文件 */
    count = 10000;
```

```

while(count--) {
    if (4096 != write(fd, buf, 4096)) {
        perror("write error");
        exit(-1);
    }
}

/* 关闭文件退出程序 */
close(fd);
exit(0);
}

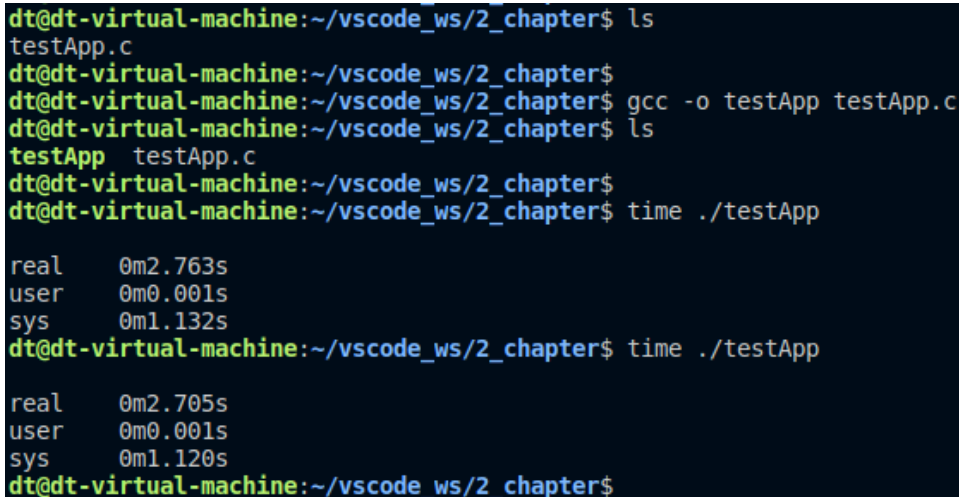
```

前面提到过, 使用直接 I/O 方式需要满足 3 个对齐要求, 程序中定义了一个 static 静态数组 buf, 将其作为数据存放的缓冲区, 在变量定义后加了 __attribute__((aligned (4096))) 修饰, 使其起始地址以 4096 字节进行对其。

Tips: __attribute 是 gcc 支持的一种机制 (也可以写成 __attribute__), 可用于设置函数属性、变量属性以及类型属性等, 对此不了解的读者请自行查找资料学习, 本书不会对此进行介绍!

程序中调用 open() 函数是指定了 O_DIRECT 标志, 使用直接 I/O, 最后通过 while 循环, 将数据写入文件中, 循环 10000 次, 每次写入 4096 个字节数据, 也就是总共写入 4096*10000 个字节 (约等于 40MB)。首次调用 write() 时其文件读写位置偏移量为 0, 之后均以 4096 字节进行递增, 所以满足直接 I/O 方式的位置偏移量必须是块大小的整数倍这个要求; 每次写入大小均是 4096 字节, 所以满足了数据大小必须是块大小的整数倍这个要求。

接下来编译测试:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp
real    0m2.763s
user    0m0.001s
sys     0m1.132s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp
real    0m2.705s
user    0m0.001s
sys     0m1.120s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 4.9.3 直接 I/O 测试结果

通过 time 命令测试可知, 每次执行程序需要花费 2.7 秒左右的时间, 使用直接 I/O 方式向文件写入约 40MB 数据大小。

Tips: 对于直接 I/O 方式的 3 个对齐限制, 大家可以自行进行验证, 譬如修改上述示例代码使之不满足 3 个对齐条件种的任何一个, 然后编译程序进行测试, 会发生 write() 函数会报错, 均是 “Invalid argument” 错误。

对示例代码 4.9.2 进行修改, 使其变成普通 I/O 方式, 其它功能相同, 最终修改后的示例代码如下所示:

示例代码 4.9.3 普通 I/O 方式

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

static char buf[8192];

int main(void)
{
    int fd;
    int count;

    /* 打开文件 */
    fd = open("./test_file", O_WRONLY | O_CREAT | O_TRUNC, 0664);
    if (0 > fd) {
        perror("open error");
        exit(-1);
    }

    /* 写文件 */
    count = 10000;
    while(count--) { //循环 10000 次, 每次写入 4096 个字节数据
        if (4096 != write(fd, buf, 4096)) {
            perror("write error");
            exit(-1);
        }
    }

    /* 关闭文件退出程序 */
    close(fd);
    exit(0);
}
```

再次进行测试:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp

real    0m0.136s
user    0m0.000s
sys     0m0.135s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp

real    0m0.146s
user    0m0.005s
sys     0m0.141s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 4.9.4 普通 I/O 测试结果

使用 `time` 命令得到的程序运行时间大约是 0.13~0.14 秒左右, 相比直接 I/O 方式的 2.7 秒, 时间上提升了 20 倍左右 (测试大小不同、每次写入的大小不同, 均会导致时间上的差别), 原因在于直接 I/O 方式每次 `write()` 调用均是直接对磁盘发起了写操作, 而普通方式只是将用户空间下的数据拷贝到了文件 I/O 内核缓冲区中, 并没直接操作硬件, 所以消耗的时间短, 硬件操作占用的时间远比内存复制占用的时间大得多。

直接 I/O 方式效率、性能比较低, 绝大部分应用程序不会使用直接 I/O 方式对文件进行 I/O 操作, 通常只在一些特殊的应用场合下才可能会使用, 那我们可以使用直接 I/O 方式来测试磁盘设备的读写速率, 这种测试方式相比普通 I/O 方式就会更加准确。

4.9.4 stdio 缓冲

介绍完文件 I/O 的内核缓冲后, 接下来我们聊一聊标准 I/O 的 `stdio` 缓冲。

标准 I/O (`fopen`、`fread`、`fwrite`、`fclose`、`fseek` 等) 是 C 语言标准库函数, 而文件 I/O (`open`、`read`、`write`、`close`、`lseek` 等) 是系统调用, 虽然标准 I/O 是在文件 I/O 基础上进行封装而实现 (譬如 `fopen` 内部实际调用了 `open`、`fread` 内部调用了 `read` 等), 但在效率、性能上标准 I/O 要优于文件 I/O, 其原因在于标准 I/O 实现维护了自己的缓冲区, 我们把这个缓冲区称为 `stdio` 缓冲区, 接下来我们聊一聊标准 I/O 的 `stdio` 缓冲。

前面提到了文件 I/O 内核缓冲, 这是由内核维护的缓冲区, 而标准 I/O 所维护的 `stdio` 缓冲是用户空间的缓冲区, 当应用程序中通过标准 I/O 操作磁盘文件时, 为了减少调用系统调用的次数, 标准 I/O 函数会将用户写入或读取文件的数据缓存在 `stdio` 缓冲区, 然后再一次性将 `stdio` 缓冲区中缓存的数据通过调用系统调用 I/O (文件 I/O) 写入到文件 I/O 内核缓冲区或者拷贝到应用程序的 `buf` 中。

通过这样的优化操作, 当操作磁盘文件时, 在用户空间缓存大块数据以减少调用系统调用的次数, 使得效率、性能得到优化。使用标准 I/O 可以使编程者免于自行处理对数据的缓冲, 无论是调用 `write()` 写入数据、还是调用 `read()` 读取数据。

对 `stdio` 缓冲进行设置

C 语言提供了一些库函数可用于对标准 I/O 的 `stdio` 缓冲区进行相关的一些设置, 包括 `setbuf()`、`setbuffer()` 以及 `setvbuf()`。

(一)、`setvbuf()` 函数

调用 `setvbuf()` 库函数可以对文件的 `stdio` 缓冲区进行设置, 譬如缓冲区的缓冲模式、缓冲区的大小、起始地址等。其函数原型如下所示:

```
#include <stdio.h>
```

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下:

stream: FILE 指针, 用于指定对应的文件, 每一个文件都可以设置它对应的 stdio 缓冲区。

buf: 如果参数 buf 不为 NULL, 那么 buf 指向 size 大小的内存区域将作为该文件的 stdio 缓冲区, 因为 stdio 库会使用 buf 指向的缓冲区, 所以应该以动态 (分配在堆内存, 譬如 malloc, 在 7.6 小节介绍) 或静态的方式在堆中为该缓冲区分配一块空间, 而不是分配在栈上的函数内的自动变量 (局部变量)。如果 buf 等于 NULL, 那么 stdio 库会自动分配一块空间作为该文件的 stdio 缓冲区 (除非参数 mode 配置为非缓冲模式)。

mode: 参数 mode 用于指定缓冲区的缓冲类型, 可取值如下:

- **_IONBF:** 不对 I/O 进行缓冲 (无缓冲)。意味着每个标准 I/O 函数将立即调用 write() 或者 read(), 并且忽略 buf 和 size 参数, 可以分别指定两个参数为 NULL 和 0。标准错误 stderr 默认属于这一种类型, 从而保证错误信息能够立即输出。
- **_IOLBF:** 采用行缓冲 I/O。在这种情况下, 当在输入或输出中遇到换行符 "\n" 时, 标准 I/O 才会执行文件 I/O 操作。对于输出流, 在输出一个换行符前将数据缓存 (除非缓冲区已经被填满), 当输出换行符时, 再将这一行数据通过文件 I/O write() 函数刷入到内核缓冲区中; 对于输入流, 每次读取一行数据。对于终端设备默认采用的就是行缓冲模式, 譬如标准输入和标准输出。
- **_IOFBF:** 采用全缓冲 I/O。在这种情况下, 在填满 stdio 缓冲区后才进行文件 I/O 操作 (read、write)。对于输出流, 当 fwrite 写入文件的数据填满缓冲区时, 才调用 write() 将 stdio 缓冲区中的数据刷入到内核缓冲区; 对于输入流, 每次读取 stdio 缓冲区大小个字节数据。默认普通磁盘上的常规文件默认常用这种缓冲模式。

size: 指定缓冲区的大小。

返回值: 成功返回 0, 失败将返回一个非 0 值, 并且会设置 errno 来指示错误原因。

需要注意的是, 当 stdio 缓冲区中的数据被刷入到内核缓冲区或被读取之后, 这些数据就不会存在于缓冲区中了, 数据被刷入了内核缓冲区或被读走了。

(二)、setbuf() 函数

setbuf() 函数构建与 setvbuf() 之上, 执行类似的任务, 其函数原型如下所示:

```
#include <stdio.h>
```

```
void setbuf(FILE *stream, char *buf);
```

setbuf() 调用除了不返回函数结果 (void) 外, 就相当于:

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

要么将 buf 设置为 NULL 以表示无缓冲, 要么指向由调用者分配的 BUFSIZ 个字节大小的缓冲区 (BUFSIZ 定义于头文件 <stdio.h> 中, 该值通常为 8192)。

(三)、setbuffer() 函数

setbuffer() 函数类似于 setbuf(), 但允许调用者指定 buf 缓冲区的大小, 其函数原型如下所示:

```
#include <stdio.h>
```

```
void setbuffer(FILE *stream, char *buf, size_t size);
```

setbuffer() 调用除了不返回函数结果 (void) 外, 就相当于:

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, size);
```

关于标准 I/O 库 `stdio` 缓冲相关的内容就给大家介绍这么多, 接下来我们进行一些测试, 来说明无缓冲、行缓冲以及全缓冲区之间的区别。

标准输出 `printf()` 的行缓冲模式测试

我们先看看下面这个简单地示例代码, 调用了 `printf()` 函数, 区别在于第二个 `printf()` 没有输出换行符。

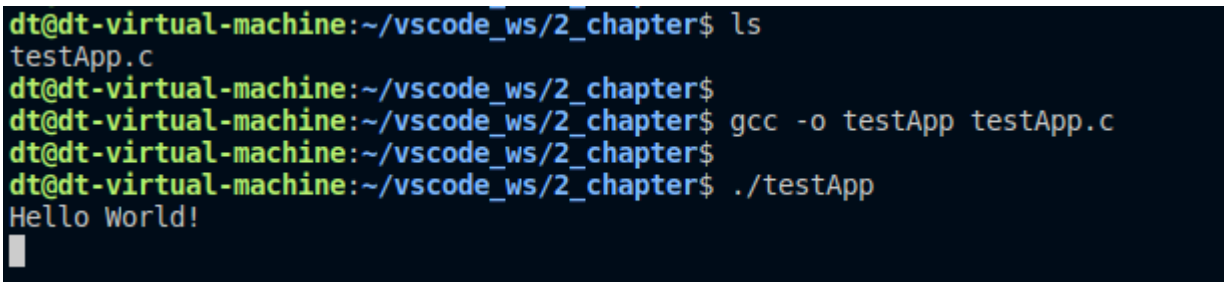
示例代码 4.9.4 `printf()` 输出测试

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Hello World!\n");
    printf("Hello World!");

    for (;;)
        sleep(1);
}
```

`printf()` 函数是标准 I/O 库函数, 向终端设备 (标准输出) 输出打印信息, 编译测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
```

图 4.9.5 运行结果

运行之后可以发现只有第一个 `printf()` 打印的信息显示出来了, 第二个并没有显示出来, 这是为什么呢? 这就是 `stdio` 缓冲的问题, 前面提到了标准输出默认采用的是行缓冲模式, `printf()` 输出的字符串写入到了标准输出的 `stdio` 缓冲区中, 只有输出换行符时 (不考虑缓冲区填满的情况) 才会将这一行数据刷入到内核缓冲区, 也就是写入标准输出文件 (终端设备), 因为第一个 `printf()` 包含了换行符, 所以已经刷入了内核缓冲区, 而第二个 `printf` 并没有包含换行符, 所以第二个 `printf` 输出的 "Hello World!" 还缓存在 `stdio` 缓冲区中, 需要等待一个换行符才可输出到终端。

联系 4.8.2 小节介绍的格式化输入 `scanf()` 函数, 程序中调用 `scanf()` 函数进行阻塞, 用户通过键盘输入数据, 只有在按下回车键 (换行符键) 时程序才会接着往下执行, 因为标准输入默认也是采用了行缓冲模式。

譬如对示例代码 4.9.4 进行修改, 使标准输出变成无缓冲模式, 修改后代码如下所示:

示例代码 4.9.5 将标准输出配置为无缓冲模式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    /* 将标准输出设置为无缓冲模式 */
```

```

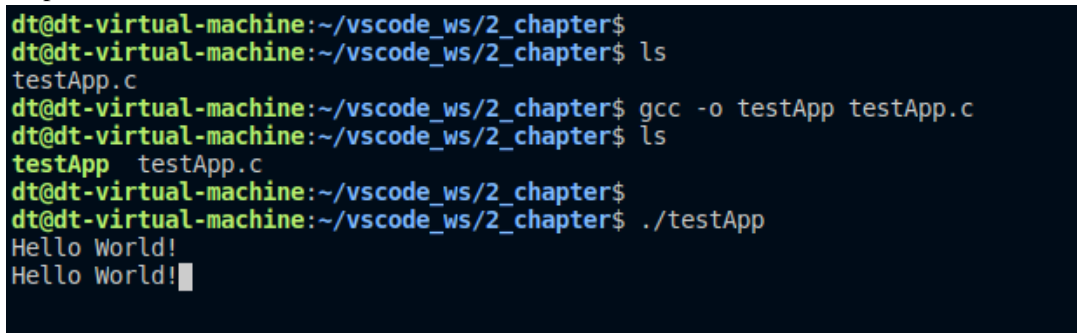
if (setvbuf(stdout, NULL, _IONBF, 0)) {
    perror("setvbuf error");
    exit(0);
}

printf("Hello World!\n");
printf("Hello World!");

for (;;)
    sleep(1);
}

```

在使用 `printf()` 之前, 调用 `setvbuf()` 函数将标准输出的 `stdio` 缓冲设置为无缓冲模式, 接着编译运行:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Hello World!

```

图 4.9.6 无缓冲标准输出测试结果

可以发现该程序却能够成功输出两个“Hello World!”, 并且白色的光标在第二个“Hello World!”后面, 意味着输出没有换行, 与程序中第二个 `printf` 没有加换行符的效果是一直。

所以通过以上两个示例代码对比可知, 标准输出默认是行缓冲模式, 只有输出了换行符时, 才会将换行符这一行字符进行输出显示 (也就是刷入到内核缓冲区), 在没有输出换行符之前, 会将数据缓存在 `stdio` 缓冲区中。

刷新 `stdio` 缓冲区

无论我们采取何种缓冲模式, 在任何时候都可以使用库函数 `fflush()` 来强制刷新 (将输出到 `stdio` 缓冲区中的数据写入到内核缓冲区, 通过 `write()` 函数) `stdio` 缓冲区, 该函数会刷新指定文件的 `stdio` 输出缓冲区, 此函数原型如下所示:

```
#include <stdio.h>
```

```
int fflush(FILE *stream);
```

参数 `stream` 指定需要进行强制刷新的文件, 如果该参数设置为 `NULL`, 则表示刷新所有的 `stdio` 缓冲区。

函数调用成功返回 0, 否则将返回 -1, 并设置 `errno` 以指示错误原因。

接下来我们对示例代码 4.9.4 进行修改, 在第二个 `printf` 后面调用 `fflush()` 函数, 修改后示例代码如下所示:

示例代码 4.9.6 使用 `fflush()` 刷新 `stdio` 缓冲区

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)

```



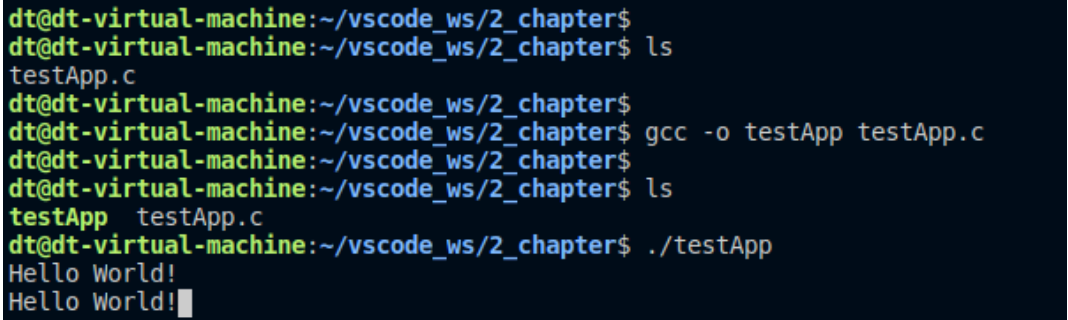
```

{
    printf("Hello World!\n");
    printf("Hello World!");
    fflush(stdout); //刷新标准输出 stdio 缓冲区

    for (;;)
        sleep(1);
}

```

运行测试:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Hello World!

```

图 4.9.7 刷新缓冲区

可以看到, 打印了两次“Hello World!”, 这就是 fflush()的作用了强制刷新 stdio 缓冲区。

除了使用库函数 fflush()之外, 还有其它方法会自动刷新 stdio 缓冲区吗? 是的, 使用库函数 fflush()是一种强制刷新的手段, 在一些其它的情况下, 也会自动刷新 stdio 缓冲区, 譬如当文件关闭时、程序退出时, 接下来我们进行演示。

(一)、关闭文件时刷新 stdio 缓冲区

同样还是直接对示例代码 4.9.4 进行修改, 在调用第二个 printf 函数后关闭标准输出, 如下所示:

示例代码 4.9.7 关闭标准输出

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Hello World!\n");
    printf("Hello World!");
    fclose(stdout); //关闭标准输出

    for (;;)
        sleep(1);
}

```

至于运行结果文档中就不贴出来了, 运行结果与图 4.9.7 是一样的。所以由此可知, 文件关闭时系统会自动刷新该文件的 stdio 缓冲区。

(二)、程序退出时刷新 stdio 缓冲区

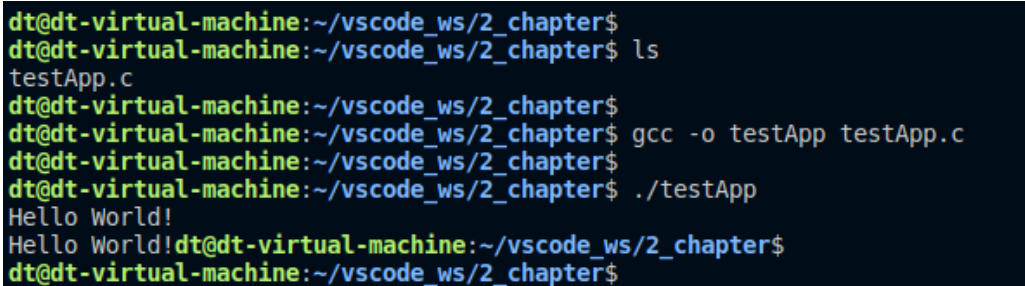
可以看到上面使用的测试程序中, 在最后都使用了一个 for 死循环, 让程序处于休眠状态无法退出, 为什么要这样做呢? 原因在于程序退出时也会自动刷新 stdio 缓冲区, 这样的话就会影响到测试结果。同样对示例代码 4.9.4 进行修改, 去掉 for 死循环, 让程序结束, 修改完之后如下所示:

示例代码 4.9.8 程序结束

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Hello World!\n");
    printf("Hello World!");
}
```

运行结果如下:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Hello World!dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.9.8 测试结果

从结果可知, 当程序退出时, 确实会自动刷新 stdio 缓冲区。但是, 与程序退出方式有关, 如果使用 `exit()`、`return` 或像上述示例代码一样不显式调用相关函数或执行 `return` 语句来结束程序, 这些情况下程序终止时会自动刷新 stdio 缓冲区; 如果使用 `_exit` 或 `_Exit()` 终止程序则不会刷新, 这里各位读者可以自行测试、验证。

关于刷新 stdio 缓冲区相关内容, 最后进行一个总结:

- 调用 `fflush()` 库函数可强制刷新指定文件的 stdio 缓冲区;
- 调用 `fclose()` 关闭文件时会自动刷新文件的 stdio 缓冲区;
- 程序退出时会自动刷新 stdio 缓冲区 (注意区分不同的情况)。

关于本小节内容就给大家介绍这么多, 笔者觉得已经非常详细了, 如果还有不太理解的地方, 希望大家能够自己动手进行测试、验证, 然后总结出相应的结论, 前面笔者一直强调, 编程是一门实践性很强的工作, 一定要学会自己分析、验证。

4.9.5 I/O 缓冲小节

本小节对前面学习的内容进行一个简单地总结, 概括说明文件 I/O 内核缓冲区和 stdio 缓冲区之间的联系与区别, 以及各种 stdio 库函数, 如下图所示:

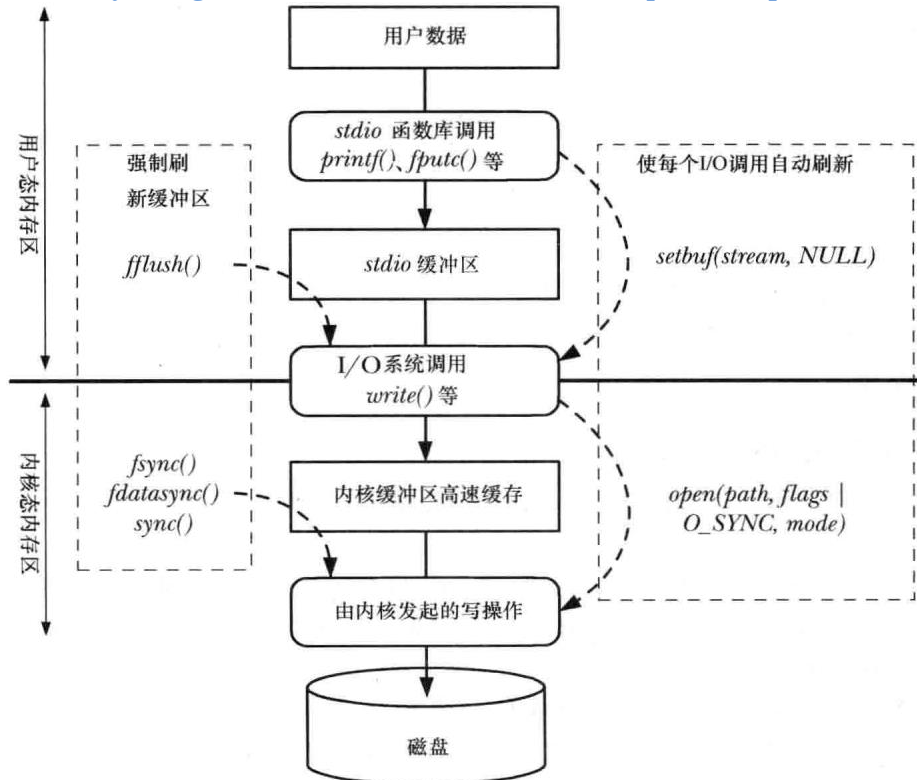


图 4.9.9 I/O 缓冲小结

从图中自上而下，首先应用程序调用标准 I/O 库函数将用户数据写入到 `stdio` 缓冲区中，`stdio` 缓冲区是由 `stdio` 库所维护的用户空间缓冲区。针对不同的缓冲模式，当满足条件时，`stdio` 库会调用文件 I/O（系统调用 I/O）将 `stdio` 缓冲区中缓存的数据写入到内核缓冲区中，内核缓冲区位于内核空间。最终由内核向磁盘设备发起读写操作，将内核缓冲区中的数据写入到磁盘（或者从磁盘设备读取数据到内核缓冲区）。

应用程序调用库函数可以对 `stdio` 缓冲区进行相应的设置，设置缓冲区缓冲模式、缓冲区大小以及由调用者指定一块空间作为 `stdio` 缓冲区，并且可以强制调用 `fflush()` 函数刷新缓冲区；而对于内核缓冲区来说，应用程序可以调用相关系统调用对内核缓冲区进行控制，譬如调用 `fsync()`、`fdatasync()` 或 `sync()` 来刷新内核缓冲区（或通过 `open` 指定 `O_SYNC` 或 `O_DSYNC` 标志），或者使用直接 I/O 绕过内核缓冲区（`open` 函数指定 `O_DIRECT` 标志）。

4.10 文件描述符与 FILE 指针互转

在应用程序中，在同一个文件上执行 I/O 操作时，还可以将文件 I/O（系统调用 I/O）与标准 I/O 混合使用，这个时候我们就需要将文件描述符和 `FILE` 指针对象之间进行转换，此时可以借助于库函数 `fdopen()`、`fileno()` 来完成。

库函数 `fileno()` 可以将标准 I/O 中使用的 `FILE` 指针转换为文件 I/O 中所使用的文件描述符，而 `fdopen()` 则进行着相反的操作，其函数原型如下所示：

```
#include <stdio.h>
```

```
int fileno(FILE *stream);
```

```
FILE *fdopen(int fd, const char *mode);
```

首先使用这两个函数需要包含头文件 `<stdio.h>`。

对于 `fileno()` 函数来说, 根据传入的 `FILE` 指针得到整数文件描述符, 通过返回值得到文件描述符, 如果转换错误将返回 -1, 并且会设置 `errno` 来指示错误原因。得到文件描述符之后, 便可以使用诸如 `read()`、`write()`、`lseek()`、`fcntl()` 等文件 I/O 方式操作文件。

`fdopen()` 函数与 `fileno()` 功能相反, 给定一个文件描述符, 得到该文件对应的 `FILE` 指针, 之后便可以使用诸如 `fread()`、`fwrite()` 等标准 I/O 方式操作文件了。参数 `mode` 与 `fopen()` 函数中的 `mode` 参数含义相同, 具体参考表 4.4.1 中所述, 若该参数与文件描述符 `fd` 的访问模式不一致, 则会导致调用 `fdopen()` 失败。

当混合使用文件 I/O 和标准 I/O 时, 需要特别注意缓冲的问题, 文件 I/O 会直接将数据写入到内核缓冲区进行高速缓存, 而标准 I/O 则会将数据写入到 `stdio` 缓冲区, 之后再调用 `write()` 将 `stdio` 缓冲区中的数据写入到内核缓冲区。譬如下面这段代码:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("print");
    write(STDOUT_FILENO, "write\n", 6);
    exit(0);
}
```

执行结果你会发现, 先输出了 "write" 字符串信息, 接着再输出了 "print" 字符串信息, 产生这个问题的原因很简单, 大家自己去思考下!

第五章 文件属性与目录

在前面的章节内容中，都是围绕普通文件 I/O 操作进行的一系列讨论，譬如打开文件、读写文件、关闭文件等，本章将抛开文件 I/O 相关话题，来讨论 Linux 文件系统的其它特性以及文件相关属性；我们将从系统调用 `stat` 开始，可利用其返回一个包含多种文件属性（包括文件时间戳、文件所有权以及文件权限等）的结构体，逐个说明 `stat` 结构中的每一个成员以了解文件的所有属性，然后将向大家介绍用以改变文件属性的各种系统调用；除此之外，还会向大家介绍 Linux 系统中的符号链接以及目录相关的操作。

本章将会讨论如下主题内容。

- Linux 系统的文件类型；
- `stat` 系统调用；
- 文件各种属性介绍：文件属主、访问权限、时间戳；
- 符号链接与硬链接；
- 目录；
- 删除文件与文件重命名。

5.1 Linux 系统中的文件类型

Linux 下一切皆文件，文件作为 Linux 系统设计思想核心理念，在 Linux 系统下显得尤为重要。在前面章节内容中，我们都是以普通文件（文本文件、二进制文件等）为例来给大家讲解文件 I/O 相关的知识内容；虽然在 Linux 系统中大部分文件都是普通文件，但并不仅仅只有普通文件，那么本小节将向大家介绍 Linux 系统中的文件类型。

在 Windows 系统下，操作系统识别文件类型一般是通过文件名后缀来判断，譬如 C 语言头文件.h、C 语言源文件.c、.txt 文本文件、压缩包文件.zip 等，在 Windows 操作系统下打开文件，首先会识别文件名后缀得到该文件的类型，然后再使用相应的调用相应的程序去打开它；譬如.c 文件，则会使用 C 代码编辑器去打开它；.zip 文件，则会使用解压软件去打开它。

但是在 Linux 系统下，并不会通过文件后缀名来识别一个文件的类型，话虽如此，但并不是意味着大家可以随便给文件加后缀；文件名也好、后缀也好都是给“人”看的，虽然 Linux 系统并不会通过后缀来识别文件，但是文件后缀也要规范、需要根据文件本身的功能属性来添加，譬如 C 源文件就以.c 为后缀、C 头文件就以.h 为后缀、shell 脚本文件就以.sh 为后缀、这是为了我们自己方便查看、浏览。

Linux 系统下一共分为 7 种文件类型，下面依次给大家介绍。

5.1.1 普通文件

普通文件（regular file）在 Linux 系统下是最常见的，譬如文本文件、二进制文件，我们编写的源代码文件这些都是普通文件，也就是一般意义上的文件。普通文件中的数据存在系统磁盘中，可以访问文件中的内容，文件中的内容以字节为单位进行存储于访问。

普通文件可以分为两大类：文本文件和二进制文件。

- **文本文件**：文件中的内容是由文本构成的，所谓文本指的是 ASCII 码字符。文件中的内容其本质上都是数字（因为计算机本身只有 0 和 1，存储在磁盘上的文件内容也都是由 0 和 1 所构成），而文本文件中的数字应该被理解为这个数字所对应的 ASCII 字符码；譬如常见的.c、.h、.sh、.txt 等这些都是文本文件，文本文件的好处就是方便人阅读、浏览以及编写。
- **二进制文件**：二进制文件中存储的本质也是数字，只不过对于二进制文件来说，这些数字并不是文本字符编码，而是真正的数字。譬如 Linux 系统下的可执行文件、C 代码编译之后得到的.o 文件、.bin 文件等都是二进制文件。

在 Linux 系统下，可以通过 stat 命令或者 ls 命令来查看文件类型，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat testApp.c
 文件: 'testApp.c'
 大小: 732          块: 8          IO 块: 4096   普通文件
设备: 801h/2049d   Inode: 3701845 硬链接: 1
权限: (0664/-rw-rw-r--) Uid: ( 1000/   dt)  Gid: ( 1000/   dt)
最近访问: 2021-01-12 18:32:03.192340301 +0800
最近更改: 2021-01-12 18:31:43.581326690 +0800
最近改动: 2021-01-12 18:31:43.581326690 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.1.1 stat 查看文件类型

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l testApp.c
-rw-rw-r-- 1 dt dt 732 1月 12 18:31 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.1.2 ls 查看文件类型

stat 命令非常友好, 会直观把文件类型显示出来; 对于 ls 命令来说, 并没有直观的显示出文件的类型, 而是通过符号表示出来, 在图 5.1.2 中画红色框位置显示出的一串字符中, 其中第一个字符 ('-') 就用于表示文件的类型, 减号 '-' 就表示该文件是一个普通文件; 除此之外, 来看看其它文件类型使用什么字符表示:

- '-' : 普通文件
- 'd' : 目录文件
- 'c' : 字符设备文件
- 'b' : 块设备文件
- 'l' : 符号链接文件
- 's' : 套接字文件
- 'p' : 管道文件

关于普通文件就给大家介绍这么多。

5.1.2 目录文件

目录 (directory) 就是文件夹, 文件夹在 Linux 系统中也是一种文件, 是一种特殊文件, 同样我们也可以使用 vi 编辑器来打开文件夹, 如下所示:

```

=====
" Netrw Directory Listing                               (netrw v155)
" /home/dt/vscode_ws/2_chapter
" Sorted by      name
" Sort sequence: [\/]$,\<core%\(\. \d\+\)\|=|>,\.h$,\.c$,\.cpp$,\~|=|*$,*\,.o$,\.obj$,\.info$,\.swp$,\.bak$,\~$
" Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:special
" =====
./
./
testApp.c
testApp*
test_file
~

```

图 5.1.3 使用 vi 打开文件夹

可以看到, 文件夹中记录了该文件夹本省的路径以及该文件夹下所存放的文件。文件夹作为一种特殊文件, 本身并不适合使用前面给大家介绍的文件 I/O 的方式来读写, 在 Linux 系统下, 会有一些专门的系统调用用于读写文件夹, 这部分内容后面再给大家介绍。

5.1.3 字符设备文件和块设备文件

学过 Linux 驱动编程开发的读者, 对字符设备文件 (character)、块设备文件 (block) 这些文件类型应该并不陌生, Linux 系统下, 一切皆文件, 也包括各种硬件设备。设备文件 (字符设备文件、块设备文件) 对应的是硬件设备, 在 Linux 系统中, 硬件设备会对应到一个设备文件, 应用程序通过对设备文件的读写来操控、使用硬件设备, 譬如 LCD 显示屏、串口、音频、按键等, 在本教程的进阶篇内容中, 将会向大家介绍如何通过设备文件操控、使用硬件设备。

Linux 系统中, 可将硬件设备分为字符设备和块设备, 所以就有了字符设备文件和块设备文件两种文件类型。虽然有设备文件, 但是设备文件并不对应磁盘上的一个文件, 也就是说设备文件并不存在于磁盘中, 而是由文件系统虚拟出来的, 一般是由内存来维护, 当系统关机时, 设备文件都会消失; 字符设备文件一般存放在 Linux 系统/dev/目录下, 所以/dev 也称为虚拟文件系统 devfs。以 Ubuntu 系统为例, 如下所示:

```
dt@dt-virtual-machine:~$ ls -l /dev/
总用量 0
crw----- 1 root root    10, 175 12月 19 12:34 agpgart
crw----- 1 root root    10, 235 12月 19 12:34 autofs
drwxr-xr-x 2 root root    300 12月 19 12:34 block
drwxr-xr-x 2 root root    80 12月 19 12:34 bsg
crw-rw---- 1 root disk   10, 234 12月 22 07:16 btrfs-control
drwxr-xr-x 3 root root    60 12月 19 12:34 bus
lrwxrwxrwx 1 root root    3 12月 19 12:34 cdrom -> sr0
lrwxrwxrwx 1 root root    3 12月 19 12:34 cdrw -> sr0
drwxr-xr-x 2 root root  3760 1月 6 06:22 char
crw----- 1 root root    5, 1 12月 19 12:34 console
lrwxrwxrwx 1 root root    11 12月 19 12:34 core -> /proc/kcore
drwxr-xr-x 8 root root   160 1月 6 06:22 cpu
crw----- 1 root root   10, 59 12月 19 12:34 cpu_dma_latency
crw----- 1 root root   10, 203 12月 19 12:34 cuse
drwxr-xr-x 5 root root   100 12月 19 12:34 disk
crw-rw----+ 1 root audio  14, 9 12月 19 12:34 dmmidi
drwxr-xr-x 2 root root    80 12月 19 12:34 dri
lrwxrwxrwx 1 root root    3 12月 19 12:34 dvd -> sr0
crw----- 1 root root   10, 61 12月 19 12:34 ecryptfs
crw-rw---- 1 root video  29, 0 12月 19 12:34 fb0
lrwxrwxrwx 1 root root    13 12月 19 12:34 fd -> /proc/self/fd
crw-rw-rw- 1 root root    1, 7 12月 19 12:34 full
crw-rw-rw- 1 root root   10, 229 12月 22 07:16 fuse
crw----- 1 root root  245, 0 12月 19 12:34 hidraw0
crw----- 1 root root   10, 228 12月 19 12:34 hpet
drwxr-xr-x 2 root root    0 12月 19 12:34 hugepages
crw----- 1 root root   10, 183 12月 19 12:34 hwrng
lrwxrwxrwx 1 root root   25 12月 19 12:34 initctl -> /run/systemd/initctl/fifo
drwxr-xr-x 4 root root   260 12月 19 12:34 input
crw-r--r-- 1 root root    1, 11 12月 19 12:34 kmsg
drwxr-xr-x 2 root root    60 12月 19 12:34 lightnvm
lrwxrwxrwx 1 root root   28 12月 19 12:34 log -> /run/systemd/journal/dev-log
brw-rw---- 1 root disk    7, 0 12月 22 07:16 loop0
brw-rw---- 1 root disk    7, 1 12月 19 12:34 loop1
```

图 5.1.4 /dev 目录下的设备文件

上图中 agpgart、autofs、btrfs-control、console 等这些都是字符设备文件，而 loop0、loop1 这些便是块设备文件。

5.1.4 符号链接文件

符号链接文件 (link) 类似于 Windows 系统中的快捷方式文件，是一种特殊文件，它的内容指向的是另一个文件路径，当对符号链接文件进行操作时，系统根据情况会对这个操作转移到它指向的文件上去，而不是对它本身进行操作，譬如，读取一个符号链接文件内容时，实际上读到的是它指向的文件的内容。

如果大家理解了 Windows 下的快捷方式，那么就会很容易理解 Linux 下的符号链接文件。图 5.1.4 中的 cdrom、cdrw、fd、initctl 等这些文件都是符号链接文件，箭头所指向的文件路径便是符号链接文件所指向的文件。

关于链接文件，在后面的内容中还会给大家进行介绍，这里暂时给大家介绍这么多！

5.1.5 管道文件

管道文件 (pipe) 主要用于进程间通信，当学习到相关知识内容的时候再给大家详解。

5.1.6 套接字文件

套接字文件 (socket) 也是一种进程间通信的方式，与管道文件不同的是，它们可以在不同主机上的进程间通信，实际上就是网络通信，当学习到网络编程相关知识内容再给大家介绍。

5.1.7 总结

本小节给大家简单地介绍了 Linux 系统中的 7 种文件类型，包括：普通文件、目录、字符设备文件、块设备文件、符号链接文件、管道文件以及套接字文件，下面对它们进行一个简单地概括：

普通文件是最常见的文件类型;
 目录也是一种文件类型;
 设备文件对应于硬件设备;
 符号链接文件类似于 Windows 的快捷方式;
 管道文件用于进程间通信;
 套接字文件用于网络通信。

5.2 stat 函数

Linux 下可以使用 `stat` 命令查看文件的属性, 其实这个命令内部就是通过调用 `stat()` 函数来获取文件属性的, `stat` 函数是 Linux 中的系统调用, 用于获取文件相关的信息, 函数原型如下所示 (可通过 "man 2 stat" 命令查看):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
int stat(const char *pathname, struct stat *buf);
```

首先使用该函数需要包含 `<sys/types.h>`、`<sys/stat.h>` 以及 `<unistd.h>` 这三个头文件。

函数参数及返回值含义如下:

pathname: 用于指定一个需要查看属性的文件路径。

buf: `struct stat` 类型指针, 用于指向一个 `struct stat` 结构体变量。调用 `stat` 函数的时候需要传入一个 `struct stat` 变量的指针, 获取到的文件属性信息就记录在 `struct stat` 结构体中, 稍后给大家介绍 `struct stat` 结构体中有记录了哪些信息。

返回值: 成功返回 0; 失败返回 -1, 并设置 `error`。

5.2.1 struct stat 结构体

`struct stat` 是内核定义的一个结构体, 在 `<sys/stat.h>` 头文件中申明, 所以可以在应用层使用, 这个结构体中的所有元素加起来构成了文件的属性信息, 结构体内容如下所示:

示例代码 5.2.1 struct stat 结构体

```
struct stat
{
    dev_t st_dev;           /* 文件所在设备的 ID */
    ino_t st_ino;          /* 文件对应 inode 节点编号 */
    mode_t st_mode;        /* 文件对应的模式 */
    nlink_t st_nlink;      /* 文件的链接数 */
    uid_t st_uid;          /* 文件所有者的用户 ID */
    gid_t st_gid;          /* 文件所有者的组 ID */
    dev_t st_rdev;         /* 设备号 (指针对设备文件) */
    off_t st_size;         /* 文件大小 (以字节为单位) */
    blksize_t st_blksize;  /* 文件内容存储的块大小 */
    blkcnt_t st_blocks;    /* 文件内容所占块数 */
    struct timespec st_atim; /* 文件最后被访问的时间 */
    struct timespec st_mtim; /* 文件内容最后被修改的时间 */
};
```

```
struct timespec st_ctim;    /* 文件状态最后被改变的时间 */
};
```

st_dev: 该字段用于描述此文件所在的设备。不常用, 可以不用理会。

st_ino: 文件的 inode 编号。

st_mode: 该字段用于描述文件的模式, 譬如文件类型、文件权限都记录在该变量中, 关于该变量的介绍请看 5.2.2 小节。

st_nlink: 该字段用于记录文件的硬链接数, 也就是为该文件创建了多少个硬链接文件。链接文件可以分为软链接(符号链接)文件和硬链接文件, 关于这些内容后面再给大家介绍。

st_uid、st_gid: 这两个字段分别用于描述文件所有者的用户 ID 以及文件所有者的组 ID, 后面再给大家介绍。

st_rdev: 该字段记录了设备号, 设备号只针对于设备文件, 包括字符设备文件和块设备文件, 不用理会。

st_size: 该字段记录了文件的大小(逻辑大小), 以字节为单位。

st_atim、st_mtim、st_ctim: 这三个字段分别用于记录文件最后被访问的时间、文件内容最后被修改的时间以及文件状态最后被改变的时间, 都是 struct timespec 类型变量, 具体介绍请看 5.2.3 小节。

5.2.2 st_mode 变量

st_mode 是 struct stat 结构体中的一个成员变量, 是一个 32 位无符号整形数据, 该变量记录了文件的类型、文件的权限这些信息, 其表示方法如下所示:



图 5.2.1 st_mode 数据信息示意图

看到图 5.2.1 的时候, 大家有没有似曾相识的感觉, 确实, 前面章节内容给大家介绍 open 函数的第三个参数 mode 时也用到类似的图, 如图 2.3.2 所示。唯一不同的在于 open 函数的 mode 参数只涉及到 S、U、G、O 这 12 个 bit 位, 并不包括用于描述文件类型的 4 个 bit 位。

O 对应的 3 个 bit 位用于描述其它用户的权限;

G 对应的 3 个 bit 位用于描述同组用户的权限;

U 对应的 3 个 bit 位用于描述文件所有者的权限;

S 对应的 3 个 bit 位用于描述文件的特殊权限。

这些 bit 位表达内容与 open 函数的 mode 参数相对应, 这里不再重述。同样, 在 mode 参数中表示权限的宏定义, 在这里也是可以使用的, 这些宏定义如下(以下数字使用的是八进制方式表示):

S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	group has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	others (not in group) have read, write, and execute permission

```
S_IROTH    00004    others have read permission
S_IWOTH    00002    others have write permission
S_IXOTH    00001    others have execute permission
```

譬如, 判断文件所有者对该文件是否具有可执行权限, 可以通过以下方法测试 (假设 `st` 是 `struct stat` 类型变量):

```
if (st.st_mode & S_IXUSR) {
    //有权限
} else {
    //无权限
}
```

这里我们重点来看看“文件类型”这 4 个 bit 位, 这 4 个 bit 位用于描述该文件的类型, 譬如该文件是普通文件、还是链接文件、亦或者是一个目录等, 那么就可以通过这 4 个 bit 位数据判断出来, 如下所示:

```
S_IFSOCK   0140000    socket (套接字文件)
S_IFLNK    0120000    symbolic link (链接文件)
S_IFREG    0100000    regular file (普通文件)
S_IFBLK    0060000    block device (块设备文件)
S_IFDIR    0040000    directory (目录)
S_IFCHR    0020000    character device (字符设备文件)
S_IFIFO    0010000    FIFO (管道文件)
```

注意上面这些数字使用的是八进制方式来表示的, 在 C 语言中, 八进制方式表示一个数字需要在数字前面添加一个 0 (零)。所以由上面可知, 当“文件类型”这 4 个 bit 位对应的数字是 14 (八进制) 时, 表示该文件是一个套接字文件、当“文件类型”这 4 个 bit 位对应的数字是 12 (八进制) 时, 表示该文件是一个链接文件、当“文件类型”这 4 个 bit 位对应的数字是 10 (八进制) 时, 表示该文件是一个普通文件等。

所以通过 `st_mode` 变量判断文件类型就很简单了, 如下 (假设 `st` 是 `struct stat` 类型变量):

```
/* 判断是不是普通文件 */
if ((st.st_mode & S_IFMT) == S_IFREG) {
    /* 是 */
}

/* 判断是不是链接文件 */
if ((st.st_mode & S_IFMT) == S_IFLNK) {
    /* 是 */
}
```

`S_IFMT` 宏是文件类型字段位掩码:

```
S_IFMT    0170000
```

除了这样判断之外, 我们还可以使用 Linux 系统封装好的宏来进行判断, 如下所示 (`m` 是 `st_mode` 变量):

```
S_ISREG(m)    #判断是不是普通文件, 如果是返回 true, 否则返回 false
S_ISDIR(m)    #判断是不是目录, 如果是返回 true, 否则返回 false
S_ISCHR(m)    #判断是不是字符设备文件, 如果是返回 true, 否则返回 false
S_ISBLK(m)    #判断是不是块设备文件, 如果是返回 true, 否则返回 false
S_ISFIFO(m)   #判断是不是管道文件, 如果是返回 true, 否则返回 false
S_ISLNK(m)    #判断是不是链接文件, 如果是返回 true, 否则返回 false
```

S_ISSOCK(m) #判断是不是套接字文件, 如果是返回 true, 否则返回 false

有了这些宏之后, 就可以通过如下方式来判断文件类型了:

```
/* 判断是不是普通文件 */
if (S_ISREG(st.st_mode)) {
    /* 是 */
}

/* 判断是不是目录 */
if (S_ISDIR(st.st_mode)) {
    /* 是 */
}
```

关于 st_mode 变量就给大家介绍这么多。

5.2.3 struct timespec 结构体

该结构体定义在<time.h>头文件中, 是 Linux 系统中时间相关的结构体。应用程序中包含了<time.h>头文件, 就可以在应用程序中使用该结构体了, 结构体内容如下所示:

示例代码 5.2.2 struct timespec 结构体

```
struct timespec
{
    time_t tv_sec;          /* 秒 */
    syscall_slong_t tv_nsec; /* 纳秒 */
};
```

struct timespec 结构体中只有两个成员变量, 一个秒 (tv_sec)、一个纳秒 (tv_nsec), time_t 其实指的就是 long int 类型, 所以由此可知, 该结构体所表示的时间可以精确到纳秒, 当然, 对于文件的时间属性来说, 并不需要这么高的精度, 往往只需精确到秒级别即可。

在 Linux 系统中, time_t 时间指的是一个时间段, 从某一个时间点到某一个时间点所经过的秒数, 譬如对于文件的三个时间属性来说, 指的是从过去的某一个时间点 (这个时间点是一个起始基准时间点) 到文件最后被访问、文件内容最后被修改、文件状态最后被改变的这个时间点所经过的秒数。time_t 时间在 Linux 下被称为日历时间, 7.2 小计中对此有详细介绍。

由示例代码 5.2.1 可知, struct stat 结构体中包含了三个文件相关的时间属性, 但这里得到的仅仅只是以秒+微秒为单位的时间值, 对于我们来说, 并不利用查看, 我们一般喜欢的是“2020-10-10 18:30:30”这种形式表示的时间, 直观、明了, 那有没有办法通过秒来得到这种形式表达的时间呢? 答案当然是可以, 譬如可以通过 localtime()/localtime_r()或者 strftime()来得到更利于我们查看的时间表达方式, 关于这些函数的介绍以及使用方法在 7.2.4 小节有详细说明。

5.2.4 练习

到这里本小节内容就给大家介绍完了, 主要给大家介绍了 stat 函数以及由此引出来的一系列知识内容。为了巩固本小节所学内容, 这里出一些简单地编程练习题, 大家可以根据本小节所学知识完成它。

- (1) 获取文件的 inode 节点编号以及文件大小, 并将它们打印出来。
- (2) 获取文件的类型, 判断此文件对于其它用户 (Other) 是否具有可读可写权限。
- (3) 获取文件的时间属性, 包括文件最后被访问的时间、文件内容最后被修改的时间以及文件状态最后被改变的时间, 并使用字符串形式将其打印出来, 包括时间和日期、表示形式自定。

以上就是根据本小节内容整理出来的一些简单的编程练习题, 下面笔者将给出对应的示例代码。

(1)编程实战练习 1

示例代码 5.2.3 编程实战练习 1

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    struct stat file_stat;
    int ret;

    /* 获取文件属性 */
    ret = stat("./test_file", &file_stat);
    if (-1 == ret) {
        perror("stat error");
        exit(-1);
    }

    /* 打印文件大小和 inode 编号 */
    printf("file size: %ld bytes\n"
           "inode number: %ld\n", file_stat.st_size,
           file_stat.st_ino);
    exit(0);
}
```

测试之前先使用 ls 命令查看 test_file 文件的 inode 节点和大小, 如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li test_file
3701841 -rw-rw-r-- 1 dt dt 8864 1月 11 18:09 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.2 ls 命令查看文件的 inode 编号和大小

从图中可以得知, 此文件的大小为 8864 个字节, inode 编号为 3701841; 接下来编译我们的测试程序、并运行:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
file size: 8864 bytes
inode number: 3701841
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.3 编程实战 1 测试结果

(2)编程实战练习 2

示例代码 5.2.4 编程实战练习 2

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    struct stat file_stat;
    int ret;

    /* 获取文件属性 */
    ret = stat("./test_file", &file_stat);
    if (-1 == ret) {
        perror("stat error");
        exit(-1);
    }

    /* 判读文件类型 */
    switch (file_stat.st_mode & S_IFMT) {
        case S_IFSOCK: printf("socket"); break;
        case S_IFLNK: printf("symbolic link"); break;
        case S_IFREG: printf("regular file"); break;
        case S_IFBLK: printf("block device"); break;
        case S_IFDIR: printf("directory"); break;
        case S_IFCHR: printf("character device"); break;
        case S_IFIFO: printf("FIFO"); break;
    }

    printf("\n");

    /* 判断该文件对其它用户是否具有读权限 */
    if (file_stat.st_mode & S_IROTH)
        printf("Read: Yes\n");
    else
        printf("Read: No\n");

    /* 判断该文件对其它用户是否具有写权限 */
    if (file_stat.st_mode & S_IWOTH)
        printf("Write: Yes\n");
    else
        printf("Write: No\n");
}
```

```
exit(0);  
}
```

测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c test_file  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
regular file  
Read: Yes  
Write: No  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.4 编程实战 2 测试结果

(3)编程实战练习 3

示例代码 5.2.5 编程实战练习 3

```
#include <sys/types.h>  
#include <sys/stat.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
int main(void)  
{  
    struct stat file_stat;  
    struct tm file_tm;  
    char time_str[100];  
    int ret;  
  
    /* 获取文件属性 */  
    ret = stat("./test_file", &file_stat);  
    if (-1 == ret) {  
        perror("stat error");  
        exit(-1);  
    }  
  
    /* 打印文件最后被访问的时间 */  
    localtime_r(&file_stat.st_atim.tv_sec, &file_tm);  
    strftime(time_str, sizeof(time_str),  
            "%Y-%m-%d %H:%M:%S", &file_tm);  
    printf("time of last access: %s\n", time_str);  
  
    /* 打印文件内容最后被修改的时间 */  
    localtime_r(&file_stat.st_mtim.tv_sec, &file_tm);  
    strftime(time_str, sizeof(time_str),  
            "%Y-%m-%d %H:%M:%S", &file_tm);
```

```

printf("time of last modification: %s\n", time_str);

/* 打印文件状态最后改变的时间 */
localtime_r(&file_stat.st_ctim.tv_sec, &file_tm);
strftime(time_str, sizeof(time_str),
         "%Y-%m-%d %H:%M:%S", &file_tm);
printf("time of last status change: %s\n", time_str);

exit(0);
}

```

测试:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
time of last access: 2021-01-12 14:52:03
time of last modification: 2021-01-11 18:09:28
time of last status change: 2021-01-12 14:51:39
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 5.2.5 实战编程 3 测试结果

可以使用 stat 命令查看 test_file 文件的这些时间属性, 对比程序打印出来是否正确:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
 文件: 'test file'
 大小: 8864          块: 24          IO 块: 4096   普通文件
设备: 801h/2049d    Inode: 3701841 硬链接: 1
权限: (0664/-rw-rw-r--) Uid: ( 1000/   dt)  Gid: ( 1000/   dt)
最近访问: 2021-01-12 14:52:03.485262883 +0800
最近更改: 2021-01-11 18:09:28.641178667 +0800
最近改动: 2021-01-12 14:51:39.136452379 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 5.2.6 stat 命令查看文件的时间属性

5.3 fstat 和 lstat 函数

前面给大家介绍了 stat 系统调用, 起始除了 stat 函数之外, 还可以使用 fstat 和 lstat 两个系统调用来获取文件属性信息。fstat、lstat 与 stat 的作用一样, 但是参数、细节方面有些许不同。

5.3.1 fstat 函数

fstat 与 stat 区别在于, stat 是从文件名出发得到文件属性信息, 不需要先打开文件; 而 fstat 函数则是从文件描述符出发得到文件属性信息, 所以使用 fstat 函数之前需要先打开文件得到文件描述符。具体该用 stat 还是 fstat, 看具体的情况; 譬如, 并不想通过打开文件来得到文件属性信息, 那么就使用 stat, 如果文件已经打开了, 那么就使用 fstat。

fstat 函数原型如下 (可通过 "man 2 fstat" 命令查看):

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

```



```
int fstat(int fd, struct stat *buf);
```

第一个参数 fd 表示文件描述符, 第二个参数以及返回值与 stat 一样。fstat 函数使用示例如下:

示例代码 5.3.1 fstat 函数使用示例

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    struct stat file_stat;
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 获取文件属性 */
    ret = fstat(fd, &file_stat);
    if (-1 == ret)
        perror("fstat error");

    close(fd);
    exit(ret);
}
```

5.3.2 lstat 函数

lstat()与 stat、fstat 的区别在于, 对于符号链接文件, stat、fstat 查阅的是符号链接文件所指向的文件对应的文件属性信息, 而 lstat 查阅的是符号链接文件本身的属性信息。

lstat 函数原型如下所示:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int lstat(const char *pathname, struct stat *buf);
```

函数参数列表、返回值与 stat 函数一样, 使用方法也一样, 这里不再重述!

5.4 文件属主

Linux 是一个多用户操作系统，系统中一般存在着好几个不同的用户，而 Linux 系统中的每一个文件都有一个与之相关联的用户和用户组，通过这个信息可以判断文件的所有者和所属组。

文件所有者表示该文件属于“谁”，也就是属于哪个用户。一般来说文件在创建时，其所有者就是创建该文件的那个用户。譬如，当前登录用户为 dt，使用 touch 命令创建了一个文件，那么这个文件的所有者就是 dt；同理，在程序中调用 open 函数创建新文件时也是如此，执行该程序的用户是谁，其文件所有者便是谁。

文件所属组则表示该文件属于哪一个用户组。在 Linux 中，系统并不是通过用户名或用户组名来识别不同的用户和用户组，而是通过 ID。ID 就是一个编号，Linux 系统会为每一个用户或用户组分配一个 ID，将用户名或用户组名与对应的 ID 关联起来，所以系统通过用户 ID (UID) 或组 ID (GID) 就可以识别出不同的用户和用户组。

Tips: 用户 ID 简称 UID、用户组 ID 简称 GID。这些都是 Linux 操作系统的基础知识，如果对用户和用户组的概念尚不熟悉，建议先自行学习这些基础知识。

譬如使用 ls 命令或 stat 命令便可以查看到文件的所有者和所属组，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls -l testApp.c
-rw-rw-rw- 1 dt dt 117 1月 14 20:25 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ stat testApp.c
 文件: 'testApp.c'
 大小: 117          块: 8          IO 块: 4096   普通文件
设备: 801h/2049d   Inode: 3701453 硬链接: 1
权限: (0666/-rw-rw-rw-)  Uid: ( 1000/   dt)  Gid: ( 1000/   dt)
最近访问: 2021-01-15 11:07:26.542747368 +0800
最近更改: 2021-01-14 20:25:12.272141838 +0800
最近改动: 2021-01-15 11:39:11.816567723 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
```

图 5.4.1 查看文件的所有者和所属组

由上图可知，testApp.c 文件的用户 ID 是 1000，用户组 ID 也是 1000。

文件的用户 ID 和组 ID 分别由 struct stat 结构体中的 st_uid 和 st_gid 所指定。既然 Linux 下的每一个文件都有与之相关联的用户 ID 和组 ID，那么对于一个进程来说亦是如此，与一个进程相关联的 ID 有 5 个或更多，如下表所示：

表 5.4.1 与进程相关联的用户 ID 和组 ID

ID 类型	作用
实际用户 ID	我们实际上是谁
实际组 ID	
有效用户 ID	用于文件访问权限检查
有效组 ID	
附属组 ID	

- 实际用户 ID 和实际组 ID 标识我们究竟是谁，也就是执行该进程的用户是谁、以及该用户对应的所属组；实际用户 ID 和实际组 ID 确定了进程所属的用户和组。
- 进程的有效用户 ID、有效组 ID 以及附属组 ID 用于文件访问权限检查，详情请查看 5.4.1 小节内容。

5.4.1 有效用户 ID 和有效组 ID

首先对于有效用户 ID 和有效组 ID 来说,这是进程所持有的概念,对于文件来说,并无此属性!有效用户 ID 和有效组 ID 是站在操作系统的角度,用于给操作系统判断当前执行该进程的用户在当前环境下对某个文件是否拥有相应的权限。

在 Linux 系统中,当进程对文件进行读写操作时,系统首先会判断该进程是否具有对该文件的读写权限,那如何判断呢?自然是通过该文件的权限位来判断,struct stat 结构体中的 st_mode 字段中就记录了该文件的权限位以及文件类型。关于文件权限检查相关内容将会在 5.5 小节中说明。

当进行权限检查时,并不是通过进程的实际用户和实际组来参与权限检查的,而是通过有效用户和有效组来参与文件权限检查。通常,绝大部分情况下,进程的有效用户等于实际用户(有效用户 ID 等于实际用户 ID),有效组等于实际组(有效组 ID 等于实际组 ID)。

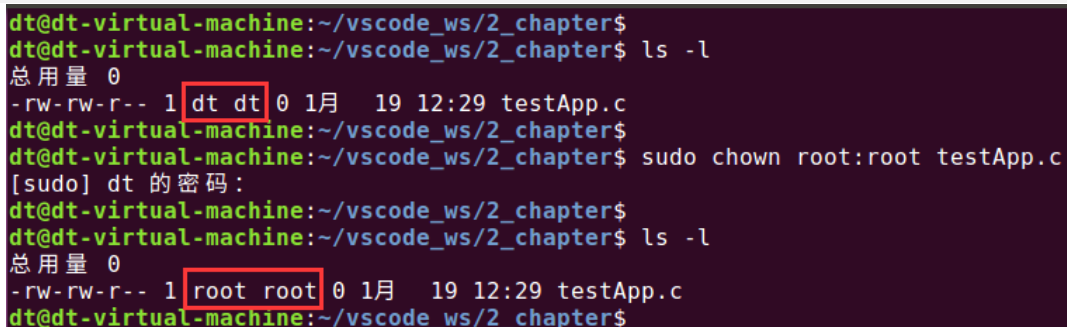
那么大家可能就要问了,什么情况下有效用户 ID 不等于实际用户 ID、有效组 ID 不等于实际组 ID? 那么关于这个问题,后面将给大家揭晓!

Tips: 文中所指的"进程对文件是否拥有 xx 权限"其实质是当前执行该进程的用户是否拥有对文件的 xx 权限。若无特别指出,文中的描述均为此意!

5.4.2 chown 函数

chown 是一个系统调用,该系统调用可用于改变文件的所有者(用户 ID)和所属组(组 ID)。其实在 Linux 系统下也有一个 chown 命令,该命令的作用也是用于改变文件的所有者和所属组,譬如将 testApp.c 文件的所有者和所属组修改为 root:

```
sudo chown root:root testApp.c
```



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 0
-rw-rw-r-- 1 dt dt 0 1月 19 12:29 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo chown root:root testApp.c
[sudo] dt 的密码:
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 0
-rw-rw-r-- 1 root root 0 1月 19 12:29 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.2 使用 chown 命令修改文件所有者和所属组

可以看到,通过该命令确实可以改变文件的所有者和所属组,这个命令内部其实就是调用了 chown 函数来实现功能的, chown 函数原型如下所示(可通过"man 2 chown"命令查看):

```
#include <unistd.h>
```

```
int chown(const char *pathname, uid_t owner, gid_t group);
```

首先,使用该命令需要包含头文件<unistd.h>。

函数参数和返回值如下所示:

pathname: 用于指定一个需要修改所有者和所属组的文件路径。

owner: 将文件的所有者修改为该参数指定的用户(以用户 ID 的形式描述);

group: 将文件的所属组修改为该参数指定的用户组(以用户组 ID 的形式描述);

返回值: 成功返回 0; 失败将返回-1, 并且会设置 errno。

该函数的用法非常简单,只需指定对应的文件路径以及相应的 owner 和 group 参数即可!如果只需要修改文件的用户 ID 和用户组 ID 中的一个,那么又该如何做呢?方法很简单,只需将其中不用修改的 ID(用

用户 ID 或用户组 ID) 与文件当前的 ID (用户 ID 或用户组 ID) 保持一致即可, 即调用 `chown` 函数时传入的用户 ID 或用户组 ID 就是该文件当前的用户 ID 或用户组 ID, 而文件当前的用户 ID 或用户组 ID 可以通过 `stat` 函数查询获取。

虽然该函数用法很简单, 但是有以下两个限制条件:

- 只有超级用户进程能更改文件的用户 ID;
- 普通用户进程可以将文件的组 ID 修改为其所从属的任意附属组 ID, 前提条件是该进程的有效用户 ID 等于文件的用户 ID; 而超级用户进程可以将文件的组 ID 修改为任意值。

所以, 由此可知, 文件的用户 ID 和组 ID 并不是随随便便就可以更改的, 其实这种设计是为系统安全着想, 如果系统中的任何普通用户进程都可以随便更改系统文件的用户 ID 和组 ID, 那么也就意味着任何普通用户对系统文件都有任意权限了, 这对于操作系统来说将是非常不安全的。

测试

接下来看一些 `chown` 函数的使用例程, 如下所示:

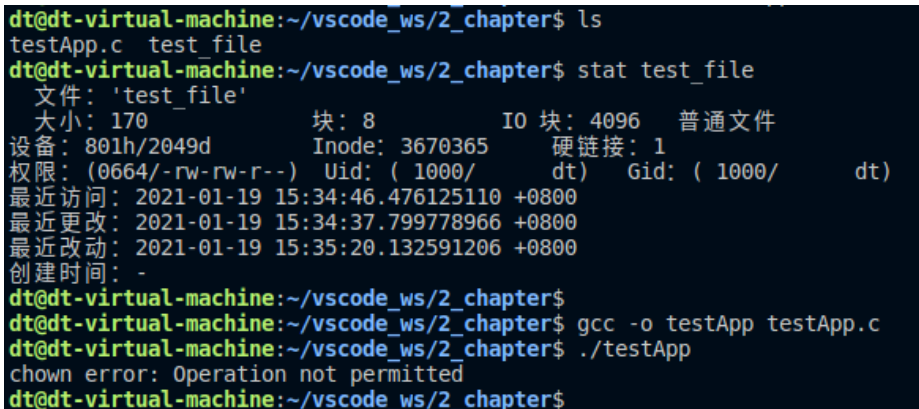
示例代码 5.4.1 `chown` 函数使用示例

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    if (-1 == chown("./test_file", 0, 0)) {
        perror("chown error");
        exit(-1);
    }

    exit(0);
}
```

代码很简单, 直接调用 `chown` 函数将 `test_file` 文件的用户 ID 和用户组 ID 修改为 0、0。0 指的就是 root 用户和 root 用户组, 接下来我们测试下:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
 文件: 'test_file'
 大小: 170          块: 8          IO 块: 4096   普通文件
设备: 801h/2049d   Inode: 3670365 硬链接: 1
权限: (0664/-rw-rw-r--)  Uid: ( 1000/   dt)  Gid: ( 1000/   dt)
最近访问: 2021-01-19 15:34:46.476125110 +0800
最近更改: 2021-01-19 15:34:37.799778966 +0800
最近改动: 2021-01-19 15:35:20.132591206 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
chown error: Operation not permitted
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.3 `chown` 测试结果

在运行测试代码之前, 先使用了 `stat` 命令查看到 `test_file` 文件的用户 ID 和用户组 ID 都等于 1000, 然后执行测试程序, 结果报错 "Operation not permitted", 显示不允许操作; 接下来重新执行程序, 此时加上 `sudo`, 如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码:
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
 文件: 'test_file'
 大小: 170          块: 8          IO 块: 4096   普通文件
设备: 801h/2049d   Inode: 3670365   硬链接: 1
权限: (0664/-rw-rw-r--)  Uid: (  0/   root)  Gid: (  0/   root)
最近访问: 2021-01-19 15:34:46.476125110 +0800
最近更改: 2021-01-19 15:34:37.799778966 +0800
最近改动: 2021-01-19 15:48:41.392432160 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.4 chown 测试结果 2

此时便可以看到, 执行之后没有打印错误提示信息, 说明 `chown` 函数调用成功了, 并且通过 `stat` 命令也可以看到文件的用户 ID 和组 ID 确实都被修改为 0 了 (也就是 root 用户)。原因在于, 加上 `sudo` 执行应用程序, 而此时应用程序便可以临时获得 root 用户的权限, 也就是会以 root 用户的身份运行程序, 也就意味着此时该应用程序的用户 ID (也就是前面给大家提到的实际用户 ID) 变成了 root 超级用户的 ID (也就是 0), 自然 `chown` 函数便可以调用成功。

在 Linux 系统下, 可以使用 `getuid` 和 `getgid` 两个系统调用分别用于获取当前进程的用户 ID 和用户组 ID, 这里说的进程的用户 ID 和用户组 ID 指的就是进程的实际用户 ID 和实际组 ID, 这两个系统调用函数原型如下所示:

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid(void);
gid_t getgid(void);
```

我们可以在示例代码 5.4.1 中加入打印用户 ID 的语句, 如下所示:

示例代码 5.4.2 chown 使用示例 2

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("uid: %d\n", getuid());

    if (-1 == chown("./test_file", 0, 0)) {
        perror("chown error");
        exit(-1);
    }

    exit(0);
}
```

再来重复上面的测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
uid: 1000
chown error: Operation not permitted
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
uid: 0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.5 chown 测试结果 3

很明显可以看到两次执行同一个应用程序它们的用户 ID 是不一样的，因为加上了 `sudo` 使得应用程序的用户 ID 由原本的普通用户 ID 1000 变成了超级用户 ID 0，使得该进程变成了超级用户进程，所以调用 `chown` 函数就不会报错。

关于 `chown` 就给大家介绍这么多，在实际应用编程中，此系统调用被用到的概率并不多，但是理论知识还是得知道。

5.4.3 fchown 和 lchown 函数

这两个同样也是系统调用，作用与 `chown` 函数相同，只是参数、细节方面有些许不同。`fchown()`、`lchown()` 这两个函数与 `chown()` 的区别就像是 `fstat()`、`lstat()` 与 `stat` 的区别，本小节就不再重述这种问题了，如果大家对此还不清楚，可以看 5.3 小节，具体使用 `fchown`、`lchown` 还是 `chown`，看情况而定。

5.5 文件访问权限

`struct stat` 结构体中的 `st_mode` 字段记录了文件的访问权限位。当提及到文件时，指的是前面给大家介绍的任何类型的文件，并不仅仅指的是普通文件；所有文件类型（目录、设备文件）都有访问权限（`access permission`），可能有很多人认为只有普通文件才有访问权限，这是一种误解！

5.5.1 普通权限和特殊权限

文件的权限可以分为两个大类，分别是普通权限和特殊权限（也可称为附加权限）。普通权限包括对文件的读、写以及执行，而特殊权限则包括一些对文件的附加权限，譬如 `Set-User-ID`、`Set-Group-ID` 以及 `Sticky`。接下来，分别对普通权限和特殊权限进行介绍。

普通权限

每个文件都有 9 个普通的访问权限位，可将它们分为 3 类，如下表：

st_mode 权限表示宏	含义
S_IRUSR	文件所有者读权限
S_IWUSR	文件所有者写权限
S_IXUSR	文件所有者执行权限
S_IRGRP	同组用户读权限
S_IWGRP	同组用户写权限
S_IXGRP	同组用户执行权限
S_IROTH	其它用户读权限
S_IWOTH	其它用户写权限
S_IXOTH	其它用户执行权限

表 5.5.1 9 个文件访问权限位

譬如使用 `ls` 命令或 `stat` 命令可以查看到文件的这 9 个访问权限，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 20
-rwxrwxr-x 1 dt dt 8816 1月 19 16:16 testApp
-rw-rw-r-- 1 dt dt 203 1月 19 16:16 testApp.c
-rw-rw-r-- 1 dt dt 170 1月 19 15:34 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
```

图 5.5.1 ls 命令查看文件的 9 个访问权限位

每一行打印信息中，前面的一串字符串就描述了该文件的 9 个访问权限以及文件类型，譬如“-rwxrwxr-x”：



图 5.5.2 文件权限位

最前面的一个字符表示该文件的类型，这个前面给大家介绍过，“-”表示该文件是一个普通文件。

- r 表示具有读权限；
- w 表示具有写权限；
- x 表示具有执行权限；
- 表示无此权限。

当进程每次对文件进行读、写、执行等操作时，内核就会对文件进行访问权限检查，以确定该进程对文件是否拥有相应的权限。而文件的权限检查就涉及到了文件的所有者（st_uid）、文件所属组（st_gid）以及其它用户，当然这里指的是从文件的角度来看；而对于进程来说，参与文件权限检查的是进程的有效用户、有效用户组以及进程的附属组用户。

如何判断权限，首先要搞清楚该进程对于需要进行操作的文件来说是属于哪一类“角色”：

- 如果进程的有效用户 ID 等于文件所有者 ID（st_uid），意味着该进程以文件所有者的角色存在；
- 如果进程的有效用户 ID 并不等于文件所有者 ID，意味着该进程并不是文件所有者身份；但是进程的有效用户组 ID 或进程的附属组 ID 之一等于文件的组 ID（st_gid），那么意味着该进程以文件所属组成员的角色存在，也就是文件所属组的同组用户成员。
- 如果进程的有效用户 ID 不等于文件所有者 ID、并且进程的有效用户组 ID 或进程的所有附属组 ID 均不等于文件的组 ID（st_gid），那么意味着该进程以其它用户的角色存在。
- 如果进程的有效用户 ID 等于 0（root 用户），则无需进行权限检查，直接对该文件拥有最高权限。

确定了进程对于文件来说是属于哪一类“角色”之后，相应的权限就直接“对号入座”即可。接下来聊一聊文件的附加的特殊权限。

特殊权限

st_mode 字段中除了记录文件的 9 个普通权限之外，还记录了文件的 3 个特殊权限，也就是图 5.2.1 中所表示的 S 字段权限位，S 字段三个 bit 位中，从高位到低位依次表示文件的 set-user-ID 位权限、set-group-ID 位权限以及 sticky 位权限，如下所示：

特殊权限	含义
S_ISUID	set-user-ID 位权限
S_ISGID	set-group-ID 位权限
S_ISVTX	Sticky 位权限

表 5.5.2 文件的特殊权限位

这三种权限分别使用 S_ISUID、S_ISGID 和 S_ISVTX 三个宏来表示：

S_ISUID	04000	set-user-ID bit
S_ISGID	02000	set-group-ID bit (see below)
S_ISVTX	01000	sticky bit (see below)

同样, 以上数字使用的是八进制方式表示。对应的 bit 位数字为 1, 则表示设置了该权限、为 0 则表示并未设置该权限; 譬如通过 `st_mode` 变量判断文件是否设置了 set-user-ID 位权限, 代码如下:

```
if (st.st_mode & S_ISUID) {
    //设置了 set-user-ID 位权限
} else {
    //没有设置 set-user-ID 位权限
}
```

这三个权限位具体有什么作用呢? 接下来给大家简单地介绍一下:

- 当进程对文件进行操作的时候、将进行权限检查, 如果文件的 set-user-ID 位权限被设置, 内核会将进程的有效 ID 设置为该文件的用户 ID (文件所有者 ID), 意味着该进程直接获取了文件所有者的权限、以文件所有者的身份操作该文件。
- 当进程对文件进行操作的时候、将进行权限检查, 如果文件的 set-group-ID 位权限被设置, 内核会将进程的有效用户组 ID 设置为该文件的用户组 ID (文件所属组 ID), 意味着该进程直接获取了文件所属组成员的权限、以文件所属组成员的身份操作该文件。

看到这里, 大家可能就要问了, 如果两个权限位同时被设置呢? 关于这个问题, 我们后面可以进行相应的测试, 答案自然会揭晓!

当然, set-user-ID 位和 set-group-ID 位权限的作用并不如此简单, 关于其它的功能本文档便不再叙述了, 因为这些特殊权限位实际中用到的机会确实不多。除此之外, Sticky 位权限也不再给大家介绍了, 笔者对此也不是很了解, 有兴趣的读者可以自行查阅相关的书籍。

Linux 系统下绝大部分的文件都没有设置 set-user-ID 位权限和 set-group-ID 位权限, 所以通常情况下, 进程的有效用户等于实际用户 (有效用户 ID 等于实际用户 ID), 有效组等于实际组 (有效组 ID 等于实际组 ID)。

5.5.2 目录权限

前面我们一直谈论的都是文件的读、写、执行权限, 那对于创建文件、删除文件等这些操作难道就不需要相应的权限了吗? 事实并不如此, 譬如: 有时删除文件或创建文件也会提示"权限不够", 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ touch hello.c
touch: 无法创建 'hello.c': 权限不够
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf testApp.c
rm: 无法删除 'testApp.c': 权限不够
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.3 创建文件、删除文件

那说明删除文件、创建文件这些操作也是需要相应权限的, 那这些权限又是从哪里获取的呢? 答案就是目录。目录 (文件夹) 在 Linux 系统下也是一种文件, 拥有与普通文件相同的权限方案 (S/U/G/O), 只是这些权限的含义另有所指。

- 目录的读权限: 可列出 (譬如: 通过 `ls` 命令) 目录之下的内容 (即目录下有哪些文件)。
- 目录的写权限: 可以在目录下创建文件、删除文件。
- 目录的执行权限: 可访问目录下的文件, 譬如对目录下的文件进行读、写、执行等操作。

拥有对目录的读权限, 用户只能查看目录中的文件列表, 譬如使用 ls 命令进行查看:

```
dt@dt-virtual-machine:~/vscode_ws$ ls -l
总用量 8
drwxr-xr-x 2 dt dt 4096 1月  7 20:53 1_chapter
dr--r-xr-x 2 dt dt 4096 1月 20 15:13 2_chapter
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ ls 2_chapter/
ls: 无法访问 '2_chapter/file1': 权限不够
ls: 无法访问 '2_chapter/file3': 权限不够
ls: 无法访问 '2_chapter/file2': 权限不够
file1 file2 file3
dt@dt-virtual-machine:~/vscode_ws$
```

图 5.5.4 只有读权限查看目录下的文件

通过"ls -l"命令可以查看到 2_chapter 目录对于文件所有者只有读权限, 当前操作的用户正是该目录所有者 dt, 之后通过"ls 2_chapter"命令查看该目录下的文件, 确实获取到了该目录下的 3 个文件: file1、file2、file3, 说明只有读权限时, 可以查看到目录下有哪些文件、显示出文件的名称; 但是会看到上面打印出了一些"权限不够"信息, 这是因为 Ubuntu 发行版对 ls 命令做了别名处理, 执行 ls 命令的时候携带了一些选项, 而这些选项会访问文件的一些信息, 所以导致出现"权限不够"问题, 这也说明, 只拥有读权限、是没法访问目录下的文件的; 为了确保使用的是 ls 命令本身, 执行时需要给出路径的完整路径/bin/ls:

```
dt@dt-virtual-machine:~/vscode_ws$ ls -l
总用量 8
drwxr-xr-x 2 dt dt 4096 1月  7 20:53 1_chapter
dr--r-xr-x 2 dt dt 4096 1月 20 15:13 2_chapter
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ /bin/ls 2_chapter/
file1 file2 file3
dt@dt-virtual-machine:~/vscode_ws$
```

图 5.5.5 使用/bin/ls 再次执行

要想访问目录下的文件, 譬如查看文件的 inode 节点、大小、权限等信息, 还需要对目录拥有执行权限。

反之, 若拥有对目录的执行权限、而无读权限, 只要知道目录内文件的名称, 仍可对其进行访问, 但不能列出目录下的内容 (即目录下包含的其它文件的名称)。

要想在目录下创建文件或删除原有文件, 需要同时拥有对该目录的执行和写权限。

所以由此可知, 如果需要对文件进行读、写或执行等操作, 不光是需要拥有该文件本身的读、写或执行权限, 还需要拥有文件所在目录的执行权限。

5.5.3 检查文件权限 access

通过前面的介绍, 大家应该知道了, 文件的权限检查不单单只讨论文件本身的权限, 还需要涉及到文件所在目录的权限, 只有同时都满足了, 才能通过操作系统的权限检查, 进而才可以对文件进行相关操作; 所以, 程序当中对文件进行相关操作之前, 需要先检查执行进程的用户是否对该文件拥有相应的权限。那如何检查呢? 可以使用 access 系统调用, 函数原型如下:

```
#include <unistd.h>
```

```
int access(const char *pathname, int mode);
```

首先, 使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下:

pathname: 需要进行权限检查的文件路径。

mode: 该参数可以取以下值:

- F_OK: 检查文件是否存在

- R_OK: 检查是否拥有读权限
- W_OK: 检查是否拥有写权限
- X_OK: 检查是否拥有执行权限

除了可以单独使用之外, 还可以通过按位或运算符"|"组合在一起。

返回值: 检查项通过则返回 0, 表示拥有相应的权限并且文件存在; 否则返回-1, 如果多个检查项组合在一起, 只要其中任何一项不通过都会返回-1。

测试

通过 access 函数检查文件是否存在, 若存在、则继续检查执行进程的用户对该文件是否有读、写、执行权限。

示例代码 5.5.1 access 函数使用示例

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MY_FILE    "./test_file"

int main(void)
{
    int ret;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("%s: file does not exist.\n", MY_FILE);
        exit(-1);
    }

    /* 检查权限 */
    ret = access(MY_FILE, R_OK);
    if (!ret)
        printf("Read permission: Yes\n");
    else
        printf("Read permission: NO\n");

    ret = access(MY_FILE, W_OK);
    if (!ret)
        printf("Write permission: Yes\n");
    else
        printf("Write permission: NO\n");

    ret = access(MY_FILE, X_OK);
    if (!ret)
        printf("Execution permission: Yes\n");
```

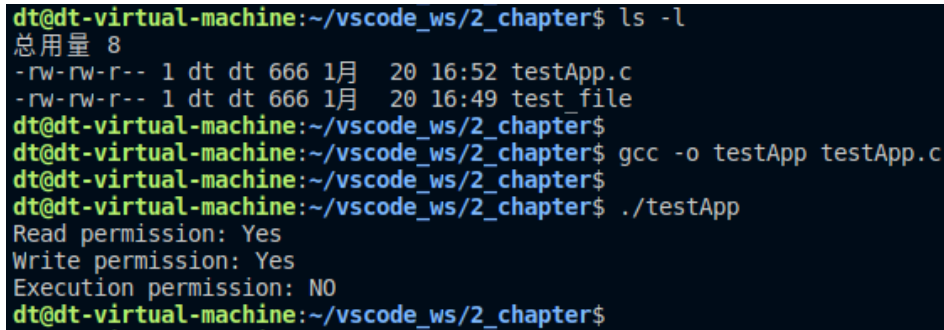
```

else
    printf("Execution permission: NO\n");

exit(0);
}

```

接下来编译测试:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
-rw-rw-r-- 1 dt dt 666 1月 20 16:52 testApp.c
-rw-rw-r-- 1 dt dt 666 1月 20 16:49 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Read permission: Yes
Write permission: Yes
Execution permission: NO
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 5.5.6 access 测试结果

5.5.4 修改文件权限 chmod

在 Linux 系统下,可以使用 chmod 命令修改文件权限,该命令内部实现方法其实是调用了 chmod 函数,chmod 函数是一个系统调用,函数原型如下所示(可通过"man 2 chmod"命令查看):

```
#include <sys/stat.h>
```

```
int chmod(const char *pathname, mode_t mode);
```

首先,使用该函数需要包含头文件<sys/stat.h>。

函数参数及返回值如下所示:

pathname: 需要进行权限修改的文件路径,若该参数所指为符号链接,实际改变权限的文件是符号链接所指向的文件,而不是符号链接文件本身。

mode: 该参数用于描述文件权限,与 open 函数的第三个参数一样,这里不再重述,可以直接使用八进制数据来描述,也可以使用相应的权限宏(单个或通过位或运算符"|"组合)。

返回值: 成功返回 0; 失败返回-1,并设置 errno。

文件权限对于文件来说是非常重要的属性,是不能随随便便被任何用户所修改的,要想更改文件权限,要么是超级用户(root)进程、要么进程有效用户 ID 与文件的用户 ID(文件所有者)相匹配。

fchmod 函数

该函数功能与 chmod 一样,参数略有不同。fchmod()与 chmod()的区别在于使用了文件描述符来代替文件路径,就像是 fstat 与 stat 的区别。函数原型如下所示:

```
#include <sys/stat.h>
```

```
int fchmod(int fd, mode_t mode);
```

使用了文件描述符 fd 代替了文件路径 pathname,其它功能都是一样的。

测试

示例代码 5.5.2 chmod 函数使用示例

```
#include <sys/stat.h>
```

```
#include <stdio.h>
```

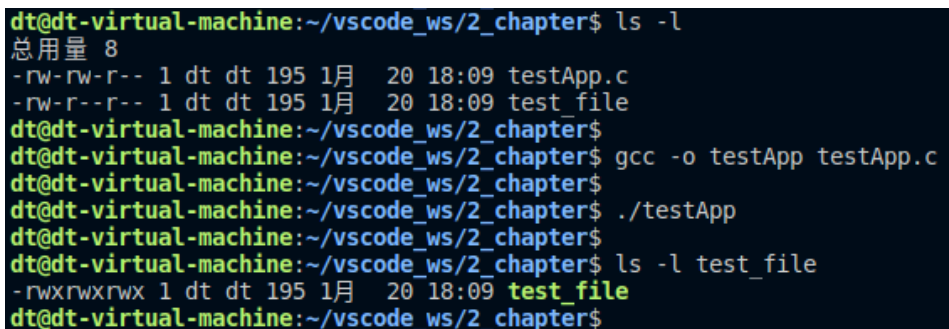
```
#include <stdlib.h>

int main(void)
{
    int ret;

    ret = chmod("./test_file", 0777);
    if (-1 == ret) {
        perror("chmod error");
        exit(-1);
    }

    exit(0);
}
```

上述代码中, 通过调用 `chmod` 函数将当前目录下的 `test_file` 文件, 其权限修改为 `0777` (八进制表示方式, 也可以使用 `S_IRUSR`、`S_IWUSR` 等这些宏来表示), 也就是文件所有者、文件所属组用户以及其它用户都拥有读、写、执行权限, 接下来编译测试:



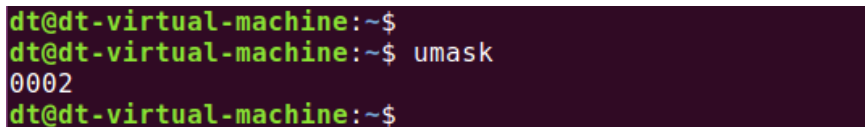
```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
-rw-rw-r-- 1 dt dt 195 1月 20 18:09 testApp.c
-rw-r--r-- 1 dt dt 195 1月 20 18:09 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l test_file
-rwxrwxrwx 1 dt dt 195 1月 20 18:09 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.7 chmod 函数测试结果

执行程序之前, `test_file` 文件的权限为 `rw-r--r--` (`0644`), 程序执行完成之后, 再次查看文件权限为 `rwxrwxrwx` (`0777`), 修改成功!

5.5.5 umask 函数

在 Linux 下有一个 `umask` 命令, 在 Ubuntu 系统下执行看看:



```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ umask
0002
dt@dt-virtual-machine:~$
```

图 5.5.8 运行 umask 命令

可以看到该命令打印出了“0002”, 这个数字表示什么意思呢? 这就要从 `umask` 命令的作用说起了, `umask` 命令用于查看/设置权限掩码, 权限掩码主要用于对新建文件的权限进行屏蔽。权限掩码的表示方式与文件权限的表示方式相同, 但是需要去除特殊权限位, `umask` 不能对特殊权限位进行屏蔽。

当新建文件时, 文件实际的权限并不等于我们所设置的权限, 譬如: 调用 `open` 函数新建文件时, 文件实际的权限并不等于 `mode` 参数所描述的权限, 而是通过如下关系得到实际权限:

```
mode & ~umask
```

譬如调用 `open` 函数新建文件时, `mode` 参数指定为 `0777`, 假设 `umask` 为 `0002`, 那么实际权限为: `0777 & (~0002) = 0775`

前面给大家介绍 open 函数的 mode 参数时,并未向大家提及到 umask,所以这里重新向大家说明。

umask 权限掩码是进程的一种属性,用于指明该进程新建文件或目录时,应屏蔽哪些权限位。进程的 umask 通常继承至其父进程(关于父、子进程相关的内容将会在后面章节给大家介绍),譬如在 Ubuntu shell 终端下执行的应用程序,它的 umask 继承至该 shell 进程。

当然, Linux 系统提供了 umask 函数用于设置进程的权限掩码,该函数是一个系统调用,函数原型如下所示(可通过"man 2 umask"命令查看):

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
mode_t umask(mode_t mask);
```

首先,使用该命令需要包含头文件<sys/types.h>和<sys/stat.h>。

函数参数和返回值含义如下:

mask: 需要设置的权限掩码值,可以发现 make 参数的类型与 open 函数、chmod 函数中的 mode 参数对应的类型一样,所以其表示方式也是一样的,前面也给大家介绍了,既可以使用数字表示(譬如八进制数)也可以直接使用宏(S_IRUSR、S_IWUSR 等)。

返回值: 返回设置之前的 umask 值,也就是旧的 umask。

测试

接下来我们编写一个测试代码,使用 umask()函数修改进程的 umask 权限掩码,测试代码如下所示:

示例代码 5.5.3 umask 函数使用示例

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    mode_t old_mask;
```

```
    old_mask = umask(0003);
```

```
    printf("old mask: %04o\n", old_mask);
```

```
    exit(0);
```

```
}
```

上述代码中,使用 umask 函数将该进程的 umask 设置为 0003(八进制),返回得到的 old_mask 则是设置之前旧的 umask 值,然后将其打印出来:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ umask
0002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
old mask: 0002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.9 umask 函数测试结果

从打印信息可以看出, 旧的 `umask` 等于 `0002`, 这个 `umask` 是从当前 `vscode` 的 `shell` 终端继承下来的, 如果没有修改进程的 `umask` 值, 默认就是从父进程继承下来的 `umask`。

这里再次强调, `umask` 是进程自身的一种属性、A 进程的 `umask` 与 B 进程的 `umask` 无关 (父子进程关系除外)。在 `shell` 终端下可以使用 `umask` 命令设置 `shell` 终端的 `umask` 值, 但是该 `shell` 终端关闭之后、再次打开一个终端, 新打开的终端将与之前关闭的终端并无任何瓜葛!

5.6 文件的时间属性

前面给大家介绍了 3 个文件的时间属性: 文件最后被访问的时间、文件内容最后被修改的时间以及文件状态最后被改变的时间, 分别记录在 `struct stat` 结构体的 `st_atim`、`st_mtim` 以及 `st_ctim` 变量中, 如下所示:

字段	说明
<code>st_atim</code>	文件最后被访问的时间
<code>st_mtim</code>	文件内容最后被修改的时间
<code>st_ctim</code>	文件状态最后被改变的时间

表 5.6.1 与文件相关的 3 个时间属性

- 文件最后被访问的时间: 访问指的是读取文件内容, 文件内容最后一次被读取的时间, 譬如使用 `read()` 函数读取文件内容便会改变该时间属性;
- 文件内容最后被修改的时间: 文件内容发生改变, 譬如使用 `write()` 函数写入数据到文件中便会改变该时间属性;
- 文件状态最后被改变的时间: 状态更改指的是该文件的 `inode` 节点最后一次被修改的时间, 譬如更改文件的访问权限、更改文件的用户 ID、用户组 ID、更改链接数等, 但它们并没有更改文件的实际内容, 也没有访问 (读取) 文件内容。为什么文件状态的更改指的是 `inode` 节点的更改呢? 3.1 小节给大家介绍 `inode` 节点的时候给大家介绍过, `inode` 中包含了很多文件信息, 譬如: 文件字节大小、文件所有者、文件对应的读/写/执行权限、文件时间戳 (时间属性)、文件数据存储的 `block` (块) 等, 所以由此可知, 状态的更改指的就是 `inode` 节点内容的更改。譬如 `chmod()`、`chown()` 等这些函数都能改变该时间属性。

表 5.6.2 列出了一些系统调用或 C 库函数对文件时间属性的影响, 有些操作并不仅仅只会影响文件本身的时间属性, 还会影响到其父目录的相关时间属性。

函数	文件			父目录			注释
	a	m	c	a	m	c	
<code>chmod()</code>			*				与 <code>fchmod()</code> 相同
<code>chown()</code>			*				与 <code>fchown()</code> 和 <code>lchown()</code> 相同
<code>exec()</code>	*						
<code>link()</code>			*		*	*	影响第二个参数的父目录
<code>mkdir()</code>	*	*	*		*	*	
<code>mkfifo()</code>	*	*	*		*	*	
<code>mknod()</code>	*	*	*		*	*	
<code>open()</code>	*	*	*		*	*	新建文件时
<code>read()</code>	*						与 <code>pread()</code> 相同
<code>rename()</code>			*		*	*	
<code>rmdir()</code>					*	*	与 <code>remove()</code> 相同
<code>unlink()</code>			*		*	*	
<code>utime()</code>	*	*	*				与 <code>utimes()</code> 、 <code>futimesat()</code> 相同
<code>write()</code>		*	*				与 <code>pwrite()</code> 相同

表 5.6.2 不同函数对文件时间属性的影响

5.6.1 utime()、utimes()修改时间属性

文件的时间属性虽然会在我们对文件进行相关操作（譬如：读、写）的时候发生改变，但这些改变都是隐式、被动的发生改变，除此之外，还可以使用 Linux 系统提供的系统调用显式的修改文件的时间属性。本小节给大家介绍如何使用 `utime()`和 `utimes()`函数来修改文件的时间属性。

Tips: 只能显式修改文件的最后一次访问时间和文件内容最后被修改的时间，不能显式修改文件状态最后被改变的时间，大家可以想一想为什么？笔者把这个作为思考题留给大家！

utime()函数

`utime()`函数原型如下所示：

```
#include <sys/types.h>
#include <utime.h>
```

```
int utime(const char *filename, const struct utimbuf *times);
```

首先，使用该函数需要包含头文件`<sys/types.h>`和`<utime.h>`。

函数参数和返回值含义如下：

filename: 需要修改时间属性的文件路径。

times: 将时间属性修改为该参数所指定的时间值，`times` 是一个 `struct utimbuf` 结构体类型的指针，稍后给大家介绍，如果将 `times` 参数设置为 `NULL`，则会将文件的访问时间和修改时间设置为系统当前时间。

返回值: 成功返回值 0；失败将返回-1，并会设置 `errno`。

来看看 `struct utimbuf` 结构体：

示例代码 5.6.1 `struct utimbuf` 结构体

```
struct utimbuf {
    time_t actime;        /* 访问时间 */
    time_t modtime;      /* 内容修改时间 */
};
```

该结构体中包含了两个 `time_t` 类型的成员，分别用于表示访问时间和内容修改时间，`time_t` 类型其实就是 `long int` 类型，所以这两个时间是以秒为单位的，所以由此可知，`utime()`函数设置文件的时间属性精度只能到秒。

同样对于文件来说，时间属性也是文件非常重要的属性之一，对文件时间属性的修改也不是任何用户都可以随便修改的，只有以下两种进程可对其进行修改：

- 超级用户进程（以 `root` 身份运行的进程）。
- 有效用户 ID 与该文件用户 ID（文件所有者）相匹配的进程。
- 在参数 `times` 等于 `NULL` 的情况下，对文件拥有写权限的进程。

除以上三种情况之外的用户进程将无法对文件时间戳进行修改。

utime 测试

接下来我们编写一个简单地测试程序，使用 `utime()`函数修改文件的访问时间和内容修改时间，示例代码如下：

示例代码 5.6.2 `utime` 函数使用示例

```
#include <sys/types.h>
#include <utime.h>
#include <unistd.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MY_FILE      "./test_file"

int main(void)
{
    struct utimbuf utm_buf;
    time_t cur_sec;
    int ret;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 获取当前时间 */
    time(&cur_sec);
    utm_buf.actime = cur_sec;
    utm_buf.modtime = cur_sec;

    /* 修改文件时间戳 */
    ret = utime(MY_FILE, &utm_buf);
    if (-1 == ret) {
        perror("utime error");
        exit(-1);
    }

    exit(0);
}
```

上述代码尝试将 test_file 文件的访问时间和内容修改时间修改为当前系统时间。程序中使用到了 time() 函数, time() 是 Linux 系统调用, 用于获取当前时间 (也可以直接将 times 参数设置为 NULL, 这样就不需要使用 time 函数来获取当前时间了), 单位为秒, 关于该函数在后面的章节内容中会给大家介绍, 这里简单地了解一下。接下来编译测试, 在运行程序之前, 先使用 stat 命令查看 test_file 文件的时间戳, 如下:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
 文件: 'test_file'
 大小: 8712      块: 24      IO 块: 4096  普通文件
设备: 801h/2049d      Inode: 3670377      硬链接: 1
权限: (0664/-rw-rw-r--)  Uid: ( 1000/      dt)  Gid: ( 1000/      dt)
最近访问: 2021-01-21 10:27:00.814290077 +0800
最近更改: 2021-01-21 10:26:50.441103476 +0800
最近改动: 2021-01-21 15:12:18.624478845 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.6.1 查看 test_file 文件的时间戳

接下来编译程序、运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
 文件: 'test_file'
 大小: 8712      块: 24      IO 块: 4096  普通文件
设备: 801h/2049d      Inode: 3670377      硬链接: 1
权限: (0664/-rw-rw-r--)  Uid: ( 1000/      dt)  Gid: ( 1000/      dt)
最近访问: 2021-01-21 15:23:05.000000000 +0800
最近更改: 2021-01-21 15:23:05.000000000 +0800
最近改动: 2021-01-21 15:23:05.787446834 +0800
创建时间: -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.6.2 运行测试程序

会发现执行完测试程序之后, test_file 文件的访问时间和内容修改时间均被更改为当前时间了(大家可以使用 date 命令查看当前系统时间), 并且会发现状态更改时间也会修改为当前时间了, 当然这个不是在程序中修改、而是内核帮它自动修改的, 为什么会这样呢? 如果大家理解了之前介绍的知识内容, 完全可以理解这个问题, 这里笔者不再重述!

utimes()函数

utimes()也是系统调用, 功能与 utime()函数一致, 只是参数、细节上有些许不同, utimes()与 utime()最大的区别在于前者可以以微秒级精度来指定时间值, 其函数原型如下所示:

```
#include <sys/time.h>
```

```
int utimes(const char *filename, const struct timeval times[2]);
```

首先, 使用该函数需要包含头文件<sys/time.h>。

函数参数和返回值含义如下:

filename: 需要修改时间属性的文件路径。

times: 将时间属性修改为该参数所指定的时间值, times 是一个 struct timeval 结构体类型的数组, 数组共有两个元素, 第一个元素用于指定访问时间, 第二个元素用于指定内容修改时间, 稍后给大家介绍, 如果 times 参数为 NULL, 则会将文件的访问时间和修改时间设置为当前时间。

返回值: 成功返回 0; 失败返回-1, 并且会设置 errno。

来看看 struct timeval 结构体:

示例代码 5.6.3 struct timeval 结构体

```
struct timeval {
    long tv_sec;      /* 秒 */
```

```
    long tv_usec;      /* 微秒 */  
};
```

该结构体包含了两个成员变量 tv_sec 和 tv_usec, 分别用于表示秒和微秒。
utimes()遵循与 utime()相同的时间戳修改权限规则。

utimes 测试

示例代码 5.6.4 utimes 使用示例

```
#include <unistd.h>  
#include <time.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/time.h>  
  
#define MY_FILE      "./test_file"  
  
int main(void)  
{  
    struct timeval tmval_arr[2];  
    time_t cur_sec;  
    int ret;  
    int i;  
  
    /* 检查文件是否存在 */  
    ret = access(MY_FILE, F_OK);  
    if (-1 == ret) {  
        printf("Error: %s file does not exist!\n", MY_FILE);  
        exit(-1);  
    }  
  
    /* 获取当前时间 */  
    time(&cur_sec);  
    for (i = 0; i < 2; i++) {  
  
        tmval_arr[i].tv_sec = cur_sec;  
        tmval_arr[i].tv_usec = 0;  
    }  
  
    /* 修改文件时间戳 */  
    ret = utimes(MY_FILE, tmval_arr);  
    if (-1 == ret) {  
        perror("utimes error");  
        exit(-1);  
    }  
}
```

```
    exit(0);  
}
```

代码不再给大家进行介绍了, 功能与示例代码 5.6.2 相同, 大家可以自己动手编译、运行测试。

5.6.2 futimens()、utimensat()修改时间属性

除了上面给大家介绍的两个系统调用外, 这里再向大家介绍两个系统调用, 功能与 `utime()` 和 `utimes()` 函数功能一样, 用于显式修改文件时间戳, 它们是 `futimens()` 和 `utimensat()`。

这两个系统调用相对于 `utime` 和 `utimes` 函数有以下三个优点:

- 可按纳秒级精度设置时间戳。相对于提供微秒级精度的 `utimes()`, 这是重大改进!
- 可单独设置某一时间戳。譬如, 只设置访问时间、而修改时间保持不变, 如果要使用 `utime()` 或 `utimes()` 来实现此功能, 则需要首先使用 `stat()` 获取另一个时间戳的值, 然后再将获取值与打算变更的时间戳一同指定。
- 可独立将任一时间戳设置为当前时间。使用 `utime()` 或 `utimes()` 函数虽然也可以通过将 `times` 参数设置为 `NULL` 来达到将时间戳设置为当前时间的效果, 但是不能单独指定某一个时间戳, 必须全部设置为当前时间 (不考虑使用额外函数获取当前时间的方式, 譬如 `time()`)。

futimens()函数

`futimens` 函数原型如下所示 (可通过 "man 2 utimensat" 命令查看):

```
#include <fcntl.h>  
#include <sys/stat.h>  
  
int futimens(int fd, const struct timespec times[2]);
```

函数原型和返回值含义如下:

fd: 文件描述符。

times: 将时间属性修改为该参数所指定的时间值, `times` 指向拥有 2 个 `struct timespec` 结构体类型变量的数组, 数组共有两个元素, 第一个元素用于指定访问时间, 第二个元素用于指定内容修改时间, 该结构体在 5.2.3 小节给大家介绍过了, 这里不再重述!

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

所以由此可知, 使用 `futimens()` 设置文件时间戳, 需要先打开文件获取到文件描述符。

该函数的时间戳可以按下列 4 种方式之一进行指定:

- 如果 `times` 参数是一个空指针, 也就是 `NULL`, 则表示将访问时间和修改时间都设置为当前时间。
- 如果 `times` 参数指向两个 `struct timespec` 结构体类型变量的数组, 任一数组元素的 `tv_nsec` 字段的值设置为 `UTIME_NOW`, 则表示相应的时间戳设置为当前时间, 此时忽略相应的 `tv_sec` 字段。
- 如果 `times` 参数指向两个 `struct timespec` 结构体类型变量的数组, 任一数组元素的 `tv_nsec` 字段的值设置为 `UTIME_OMIT`, 则表示相应的时间戳保持不变, 此时忽略 `tv_sec` 字段。
- 如果 `times` 参数指向两个 `struct timespec` 结构体类型变量的数组, 且 `tv_nsec` 字段的值既不是 `UTIME_NOW` 也不是 `UTIME_OMIT`, 在这种情况下, 相应的时间戳设置为相应的 `tv_sec` 和 `tv_nsec` 字段指定的值。

Tips: `UTIME_NOW` 和 `UTIME_OMIT` 是两个宏定义。

使用 `futimens()` 函数只有以下进程, 可对文件时间戳进行修改:

- 超级用户进程。
- 在参数 `times` 等于 `NULL` 的情况下, 对文件拥有写权限的进程。
- 有效用户 ID 与该文件用户 ID (文件所有者) 相匹配的进程。

futimens()测试

示例代码 5.6.5 futimens 函数使用示例

```
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#define MY_FILE    "./test_file"

int main(void)
{
    struct timespec tmsp_arr[2];
    int ret;
    int fd;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(MY_FILE, O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 修改文件时间戳 */
    #if 1
        ret = futimens(fd, NULL);    //同时设置为当前时间
    #endif

    #if 0
        tmsp_arr[0].tv_nsec = UTIME_OMIT; //访问时间保持不变
        tmsp_arr[1].tv_nsec = UTIME_NOW; //内容修改时间设置为当期时间
        ret = futimens(fd, tmsp_arr);
    #endif
}
```

```
#if 0
    tmsp_arr[0].tv_nsec = UTIME_NOW; //访问时间设置为当前时间
    tmsp_arr[1].tv_nsec = UTIME_OMIT; //内容修改时间保持不变
    ret = futimens(fd, tmsp_arr);
#endif

    if (-1 == ret) {
        perror("futimens error");
        goto err;
    }

err:
    close(fd);
    exit(ret);
}
```

代码不再给大家进行介绍, 大家可以自己动手编译、运行测试。

utimensat()函数

utimensat()与 futimens()函数在功能上是一样的, 同样可以实现纳秒级精度设置时间戳、单独设置某一时间戳、独立将任一时间戳设置为当前时间, 与 futimens()在参数以及细节上存在一些差异, 使用 futimens()函数, 需要先将文件打开, 通过文件描述符进行操作, utimensat()可以直接使用文件路径方式进行操作。utimensat 函数原型如下所示:

```
#include <fcntl.h>
#include <sys/stat.h>

int utimensat(int dirfd, const char *pathname, const struct timespec times[2], int flags);
```

首先, 使用该函数需要包含头文件<fcntl.h>和<sys/stat.h>。

函数参数和返回值含义如下:

dirfd: 该参数可以是一个目录的文件描述符, 也可以是特殊值 AT_FDCWD; 如果 pathname 参数指定的是文件的绝对路径, 则此参数会被忽略。

pathname: 指定文件路径。如果 pathname 参数指定的是一个相对路径、并且 dirfd 参数不等于特殊值 AT_FDCWD, 则实际操作的文件路径是相对于文件描述符 dirfd 指向的目录进行解析。如果 pathname 参数指定的是一个相对路径、并且 dirfd 参数等于特殊值 AT_FDCWD, 则实际操作的文件路径是相对于调用进程的当前工作目录进行解析, 关于进程的工作目录在 5.7 小节中有介绍。

times: 与 futimens()的 times 参数含义相同。

flags: 此参数可以为 0, 也可以设置为 AT_SYMLINK_NOFOLLOW, 如果设置为 AT_SYMLINK_NOFOLLOW, 当 pathname 参数指定的文件是符号链接, 则修改的是该符号链接的时间戳, 而不是它所指向的文件。

返回值: 成功返回 0; 失败返回-1、并会设置时间戳。

utimensat()遵循与 futimens()相同的时间戳修改权限规则。

utimensat()函数测试

示例代码 5.6.6 utimensat 函数使用示例

```
#include <fcntl.h>
```

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define MY_FILE      "/home/dt/vscode_ws/2_chapter/test_file"

int main(void)
{
    struct timespec tmsp_arr[2];
    int ret;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 修改文件时间戳 */
    #if 1
        ret = utimensat(-1, MY_FILE, NULL, AT_SYMLINK_NOFOLLOW); //同时设置为当前时间
    #endif

    #if 0
        tmsp_arr[0].tv_nsec = UTIME_OMIT; //访问时间保持不变
        tmsp_arr[1].tv_nsec = UTIME_NOW; //内容修改时间设置为当期时间
        ret = utimensat(-1, MY_FILE, tmsp_arr, AT_SYMLINK_NOFOLLOW);
    #endif

    #if 0
        tmsp_arr[0].tv_nsec = UTIME_NOW; //访问时间设置为当前时间
        tmsp_arr[1].tv_nsec = UTIME_OMIT; //内容修改时间保持不变
        ret = utimensat(-1, MY_FILE, tmsp_arr, AT_SYMLINK_NOFOLLOW);
    #endif

    if (-1 == ret) {
        perror("futimens error");
        exit(-1);
    }

    exit(0);
}
```

代码不再给大家进行介绍,大家可以自己动手编译、运行测试。

5.7 符号链接(软链接)与硬链接

在 Linux 系统中有两种链接文件,分为软链接(也叫符号链接)文件和硬链接文件,软链接文件也就是前面给大家的 Linux 系统下的七种文件类型之一,其作用类似于 Windows 下的快捷方式。那么硬链接文件又是什么呢?本小节就来聊一聊它们之间的区别。

首先,从使用角度来讲,两者没有任何区别,都与正常的文件访问方式一样,支持读、写以及执行。那它们的区别在哪呢?在底层原理上,为了说明这个问题,先来创建一个硬链接文件,如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls
test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ln test_file hard1
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ln test_file hard2
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls -li
总用量 36
3670377 -r----- 3 dt dt 8712 1月 22 17:07 hard1
3670377 -r----- 3 dt dt 8712 1月 22 17:07 hard2
3670377 -r----- 3 dt dt 8712 1月 22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
```

图 5.7.1 创建硬链接文件

Tips: 使用 ln 命令可以为一个文件创建软链接文件或硬链接文件,用法如下:

硬链接: ln 源文件 链接文件

软链接: ln -s 源文件 链接文件

关于该命令其它用法,可以查看 man 手册。

从图 5.7.1 中可知,使用 ln 命令创建的两个硬链接文件与源文件 test_file 都拥有相同的 inode 号,既然 inode 相同,也就意味着它们指向了物理硬盘的同一个区块,仅仅只是文件名字不同而已,创建出来的硬链接文件与源文件对文件系统来说是完全平等的关系。那么大家可能要问了,如果删除了硬链接文件或源文件其中之一,那文件所对应的 inode 以及文件内容在磁盘中的数据块会被文件系统回收吗?事实上并不会这样,因为 inode 数据结构中会记录文件的链接数,这个链接数指的就是硬链接数,struct stat 结构体中的 st_nlink 成员变量就记录了文件的链接数,这些内容前面已经给大家介绍过了。

当为文件每创建一个硬链接,inode 节点上的链接数就会加一,每删除一个硬链接,inode 节点上的链接数就会减一,直到为 0,inode 节点和对应的数据块才会被文件系统所回收,也就意味着文件已经从文件系统中被删除了。从图 5.7.1 中可知,使用"ls -li"命令查看到,此时链接数为 3(dt 用户名前面的那个数字),我们明明创建了 2 个链接文件,为什么链接数会是 3?其实源文件 test_file 本身就是一个硬链接文件,所以这里才是 3。

当我们删除其中任何一个文件后,链接数就会减少,如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
hard1 hard2 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf hard2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 24
3670377 -r----- 2 dt dt 8712 1月 22 17:07 hard1
3670377 -r----- 2 dt dt 8712 1月 22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf hard1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 12
3670377 -r----- 1 dt dt 8712 1月 22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.2 删除链接文件

接下来再来聊一聊软链接文件，软链接文件与源文件有着不同的 inode 号，如图 5.7.3 所示，所以也就是意味着它们之间有着不同的数据块，但是软链接文件的数据块中存储的是源文件的路径名，链接文件可以通过这个路径找到被链接的源文件，它们之间类似于一种“主从”关系，当源文件被删除之后，软链接文件依然存在，但此时它指向的是一个无效的文件路径，这种链接文件被称为悬空链接，如图 5.7.4 所示。

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ln -s test_file soft1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ln -s test_file soft2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 12
3671798 lrwxrwxrwx 1 dt dt 9 1月 23 10:49 soft1 -> test_file
3671799 lrwxrwxrwx 1 dt dt 9 1月 23 10:49 soft2 -> test_file
3670377 -r----- 1 dt dt 8712 1月 22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.3 创建软链接

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 0
3671798 lrwxrwxrwx 1 dt dt 9 1月 23 10:49 soft1 -> test_file
3671799 lrwxrwxrwx 1 dt dt 9 1月 23 10:49 soft2 -> test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.4 删除源文件

从图中还可看出，inode 节点中记录的链接数并未将软链接计算在内。

介绍完它们之间的区别之后，大家可能觉得硬链接相对于软链接来说有较大的优势，其实并不是这样，对于硬链接来说，存在一些限制情况，如下：

- 不能对目录创建硬链接（超级用户可以创建，但必须在底层文件系统支持的情况下）。

- 硬链接通常要求链接文件和源文件位于同一文件系统中。

而软链接文件的使用并没有上述限制条件，优点如下所示：

- 可以对目录创建软链接；
- 可以跨越不同文件系统；
- 可以对不存在的文件创建软链接。

5.7.1 创建链接文件

在 Linux 系统下，可以使用系统调用创建硬链接文件或软链接文件，本小节向大家介绍如何通过这些系统调用创建链接文件。

创建硬链接 link()

link()系统调用用于创建硬链接文件, 函数原型如下(可通过"man 2 link"命令查看):

```
#include <unistd.h>
```

```
int link(const char *oldpath, const char *newpath);
```

首先, 使用该函数需要包含头文件<unistd.h>。

函数原型和返回值含义如下:

oldpath: 用于指定被链接的源文件路径, 应避免 oldpath 参数指定为软链接文件, 为软链接文件创建硬链接没有意义, 虽然并不会报错。

newpath: 用于指定硬链接文件路径, 如果 newpath 指定的文件路径已存在, 则会产生错误。

返回值: 成功返回 0; 失败将返回-1, 并且会设置 errno。

link 函数测试

接下来我们编写一个简单地程序, 演示 link 函数如何使用:

示例代码 5.7.1 link 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int ret;

    ret = link("./test_file", "./hard");
    if (-1 == ret) {
        perror("link error");
        exit(-1);
    }

    exit(0);
}
```

程序中通过 link 函数为当前目录下的 test_file 文件创建了一个硬链接 hard, 编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
-rw-rw-r-- 1 dt dt 195 1月 23 11:43 testApp.c
-rw-rw-r-- 1 dt dt 195 1月 23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 24
-rw-rw-r-- 2 dt dt 195 1月 23 11:43 hard
-rwxrwxr-x 1 dt dt 8704 1月 23 11:46 testApp
-rw-rw-r-- 1 dt dt 195 1月 23 11:43 testApp.c
-rw-rw-r-- 2 dt dt 195 1月 23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.5 link 函数测试结果

创建软链接 symlink()

symlink()系统调用用于创建软链接文件, 函数原型如下(可通过"man 2 symlink"命令查看):

```
#include <unistd.h>
```

```
int symlink(const char *target, const char *linkpath);
```

首先, 使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下:

target: 用于指定被链接的源文件路径, target 参数指定的也可以是一个软链接文件。

linkpath: 用于指定硬链接文件路径, 如果 newpath 指定的文件路径已存在, 则会产生错误。

返回值: 成功返回 0; 失败将返回-1, 并会设置 errno。

创建软链接时, 并不要求 target 参数指定的文件路径已经存在, 如果文件不存在, 那么创建的软链接将成为“悬空链接”。

symlink 函数测试

接下来我们编写一个简单地程序, 演示 symlink 函数如何使用:

示例代码 5.7.2 symlink 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int ret;

    ret = symlink("./test_file", "./soft");
    if (-1 == ret) {
        perror("symlink error");
        exit(-1);
    }

    exit(0);
}
```

程序中通过 symlink 函数为当前目录下的 test_file 文件创建了一个软链接 soft, 编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 20
lrwxrwxrwx 1 dt dt 11 1月 23 12:06 soft -> ./test_file
-rwxrwxr-x 1 dt dt 8712 1月 23 12:06 testApp
-rw-rw-r-- 1 dt dt 201 1月 23 12:04 testApp.c
-rw-rw-r-- 1 dt dt 195 1月 23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.6 symlink 函数测试结果

5.7.2 读取软链接文件

前面给大家介绍到, 软链接文件数据块中存储的是被链接文件的路径信息, 那如何读取出软链接文件中存储的路径信息呢? 大家认为使用 `read` 函数可以吗? 答案是不可以, 因为使用 `read` 函数之前, 需要先 `open` 打开该文件得到文件描述符, 但是调用 `open` 打开一个链接文件本身是不会成功的, 因为打开的并不是链接文件本身、而是其指向的文件, 所以不能使用 `read` 来读取, 那怎么办呢? 可以使用系统调用 `readlink`。

`readlink` 函数原型如下所示:

```
#include <unistd.h>
```

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

函数参数和返回值含义如下:

pathname: 需要读取的软链接文件路径。只能是软链接文件路径, 不能是其它类型文件, 否则调用函数将报错。

buf: 用于存放路径信息的缓冲区。

bufsiz: 读取大小, 一般读取的大小需要大于链接文件数据块中存储的文件路径信息字节大小。

返回值: 失败将返回-1, 并会设置 `errno`; 成功将返回读取到的字节数。

readlink 函数测试

接下来我们编写一个简单地程序, 演示 `readlink` 函数如何使用:

示例代码 5.7.3 `readlink` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char buf[50];
    int ret;

    memset(buf, 0x0, sizeof(buf));
    ret = readlink("./soft", buf, sizeof(buf));
    if (-1 == ret) {
        perror("readlink error");
        exit(-1);
    }

    printf("%s\n", buf);
    exit(0);
}
```

使用 `readlink` 函数读取当前目录下的软链接文件 `soft`, 并将读取到的信息打印出来, 测试如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
lrwxrwxrwx 1 dt dt 11 1月 23 12:06 soft -> ./test_file
-rw-rw-r-- 1 dt dt 295 1月 23 12:36 testApp.c
-rw-rw-r-- 1 dt dt 195 1月 23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
./test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.7 readlink 函数测试结果

5.8 目录

目录（文件夹）在 Linux 系统也是一种文件，是一种特殊文件，同样可以使用前面给大家介绍 `open`、`read` 等这些系统调用以及 C 库函数对其进行操作，但是目录作为一种特殊文件，并不适合使用前面介绍的文件 I/O 方式进行读写等操作。在 Linux 系统下，会有一些专门的系统调用或 C 库函数用于对文件夹进行操作，譬如：打开、创建文件夹、删除文件夹、读取文件夹以及遍历文件夹中的文件等，那么本小节将向大家介绍目录相关的知识内容。

5.8.1 目录存储形式

3.1 小节中给大家介绍了普通文件的管理形式或存储形式，本小节聊一聊目录这种特殊文件在文件系统中的存储形式，其实目录在文件系统中的存储方式与常规文件类似，常规文件包括了 `inode` 节点以及文件内容数据存储块（block），参考图 3.1.1 所示；但对于目录来说，其存储形式则是由 `inode` 节点和目录块所构成，目录块当中记录了有哪些文件组织在这个目录下，记录它们的文件名以及对应的 `inode` 编号。

其存储形式如下图所示：

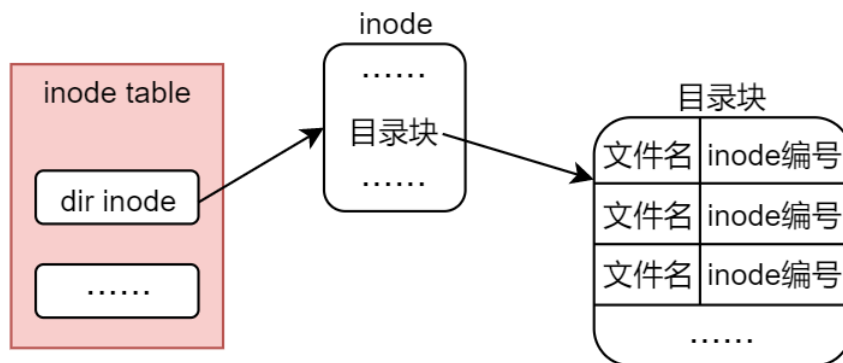


图 5.8.1 目录在文件系统中的存储形式

目录块当中有多个目录项（或叫目录条目），每一个目录项（或目录条目）都会对应到该目录下的某一个文件，目录项当中记录了该文件的文件名以及它的 `inode` 节点编号，所以通过目录的目录块便可以遍历找到该目录下的所有文件以及所对应的 `inode` 节点。

所以对此总结如下：

- 普通文件由 `inode` 节点和数据块构成
- 目录由 `inode` 节点和目录块构成

5.8.2 创建和删除目录

使用 `open` 函数可以创建一个普通文件，但不能用于创建目录文件，在 Linux 系统下，提供了专门用于创建目录 `mkdir()` 以及删除目录 `rmdir` 相关的系统调用。

mkdir 函数

函数原型如下所示:

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

函数参数和返回值含义如下:

pathname: 需要创建的目录路径。

mode: 新建目录的权限设置, 设置方式与 open 函数的 mode 参数一样, 最终权限为 (mode & ~umask)。

返回值: 成功返回 0; 失败将返回-1, 并会设置 errno。

pathname 参数指定的新建目录的路径, 该路径名可以是相对路径, 也可以是绝对路径, 若指定的路径名已经存在, 则调用 mkdir()将会失败。

mode 参数指定了新目录的权限, 目录拥有与普通文件相同的权限位, 但是其表示的含义与普通文件却有不同, 5.5.2 小计对此作了说明。

mkdir 函数测试

示例代码 5.8.1 mkdir 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void)
{
    int ret;

    ret = mkdir("./new_dir", S_IRWXU |
                S_IRGRP | S_IXGRP |
                S_IROTH | S_IXOTH);
    if (-1 == ret) {
        perror("mkdir error");
        exit(-1);
    }

    exit(0);
}
```

上述代码中, 我们通过 mkdir 函数在当前目录下创建了一个目录 new_dir, 并将其权限设置为 0755 (八进制), 编译运行:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 20
drwxr-xr-x 2 dt dt 4096 1月 23 14:50 new_dir
-rwxrwxr-x 1 dt dt 8712 1月 23 14:50 testApp
-rw-rw-r-- 1 dt dt 267 1月 23 14:47 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.2 mkdir 创建目录

rmdir 函数

rmdir()用于删除一个目录

```
#include <unistd.h>
```

```
int rmdir(const char *pathname);
```

首先, 使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下:

pathname: 需要删除的目录对应的路径名, 并且该目录必须是一个空目录, 也就是该目录下只有.和..这两个目录项; pathname 指定的路径名不能是软链接文件, 即使该链接文件指向了一个空目录。

返回值: 成功返回 0; 失败将返回-1, 并会设置 errno。

rmdir 函数测试

示例代码 5.8.2 rmdir 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int ret;

    ret = rmdir("./new_dir");
    if (-1 == ret) {
        perror("rmdir error");
        exit(-1);
    }

    exit(0);
}
```

5.8.3 打开、读取以及关闭目录

打开、读取、关闭一个普通文件可以使用 open()、read()、close(), 而对于目录来说, 可以使用 opendir()、readdir()和 closedir()来打开、读取以及关闭目录, 接下来将向大家介绍这 3 个 C 库函数的用法。

打开文件 opendir

opendir()函数用于打开一个目录,并返回指向该目录的句柄,供后续操作使用。Opendir 是一个 C 库函数, opendir()函数原型如下所示:

```
#include <sys/types.h>
#include <dirent.h>
```

```
DIR *opendir(const char *name);
```

函数参数和返回值含义如下:

name: 指定需要打开的目录路径名,可以是绝对路径,也可以是相对路径。

返回值: 成功将返回指向该目录的句柄,一个 DIR 指针(其实质是一个结构体指针),其作用类似于 open 函数返回的文件描述符 fd,后续对该目录的操作需要使用该 DIR 指针变量;若调用失败,则返回 NULL。

读取目录 readdir

readdir()用于读取目录,获取目录下所有文件的名称以及对应 inode 号。这里给大家介绍的 readdir()是一个 C 库函数(事实上 Linux 系统还提供了一个 readdir 系统调用),其函数原型如下所示:

```
#include <dirent.h>
```

```
struct dirent *readdir(DIR *dirp);
```

首先,使用该函数需要包含头文件<dirent.h>。

函数参数和返回值含义如下:

dirp: 目录句柄 DIR 指针。

返回值: 返回一个指向 struct dirent 结构体的指针,该结构体表示 dirp 指向的目录流中的下一个目录条目。在到达目录流的末尾或发生错误时,它返回 NULL。

Tips: “流”是从自然界中抽象出来的一种概念,有点类似于自然界当中的水流,在文件操作中,文件内容数据类似池塘中存储的水,N 个字节数据被读取出来或将 N 个字节数据写入到文件中,这些数据就构成了字节流。

“流”这个概念是动态的,而不是静态的。编程当中提到这个概念,一般都是与 I/O 相关,所以也经常叫做 I/O 流;但对于目录这种特殊文件来说,这里将目录块中存储的数据称为目录流,存储了一个一个的目录项(目录条目)。

struct dirent 结构体内容如下所示:

示例代码 5.8.3 struct dirent 结构体

```
struct dirent {
    ino_t          d_ino;          /* inode 编号 */
    off_t          d_off;         /* not an offset; see NOTES */
    unsigned short d_reclen;     /* length of this record */
    unsigned char  d_type;       /* type of file; not supported by all filesystem types */
    char           d_name[256]; /* 文件名 */
};
```

对于 struct dirent 结构体,我们只需要关注 d_ino 和 d_name 两个字段即可,分别记录了文件的 inode 编号和文件名,其余字段并不是所有系统都支持,所以也不再给大家介绍,这些字段一般也不会使用到。

每调用一次 readdir(),就会从 dirp 所指向的目录流中读取下一条目录项(目录条目),并返回 struct dirent 结构体指针,指向经静态分配而得的 struct dirent 类型结构,每次调用 readdir()都会覆盖该结构。一旦遇到目录结尾或是出错,readdir()将返回 NULL,针对后一种情况,还会设置 errno 以示具体错误。那如何区别究竟是到了目录末尾还是出错了呢,可通过如下代码进行判断:

```
error = 0;
```

```

direntp = readdir(dirp);
if (NULL == direntp) {
    if (0 != error) {
        /* 出现了错误 */
    } else {
        /* 已经到了目录末尾 */
    }
}
}

```

使用 `readdir()` 返回时并未对文件名进行排序, 而是按照文件在目录中出现的天然次序 (这取决于文件系统向目录添加文件时所遵循的次序, 及其在删除文件后对目录列表中空隙的填补方式)。

当使用 `opendir()` 打开目录时, 目录流将指向了目录列表的头部 (0), 使用 `readdir()` 读取一条目录条目之后, 目录流将会向后移动、指向下一个目录条目。这其实跟 `open()` 类似, 当使用 `open()` 打开文件的时候, 文件位置偏移量默认指向了文件头部, 当使用 `read()` 或 `write()` 进行读写时, 文件偏移量会自动向后移动。

rewinddir 函数

`rewinddir()` 是 C 库函数, 可将目录流重置为目录起点, 以便对 `readdir()` 的下次调用将从目录列表中的第一个文件开始。 `rewinddir` 函数原型如下所示:

```

#include <sys/types.h>
#include <dirent.h>

```

```
void rewinddir(DIR *dirp);
```

首先, 使用该函数需要包含头文件 `<dirent.h>`。

函数参数和返回值含义如下:

dirp: 目录句柄。

返回值: 无返回值。

关闭目录 closedir 函数

`closedir()` 函数用于关闭处于打开状态的目录, 同时释放它所使用的资源, 其函数原型如下所示:

```

#include <sys/types.h>
#include <dirent.h>

```

```
int closedir(DIR *dirp);
```

首先, 使用该函数需要包含头文件 `<sys/types.h>` 和 `<dirent.h>`。

函数参数和返回值含义如下:

dirp: 目录句柄。

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

练习

根据本小节所学知识内容, 可以做一个简单地编程练习, 打开一个目录、并将目录下的所有文件的名称以及其对应 `inode` 编号打印出来。示例代码如下所示:

示例代码 5.8.4 本节编程练习

```

#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>

```



```
#include <errno.h>

int main(void)
{
    struct dirent *dir;
    DIR *dirp;
    int ret = 0;

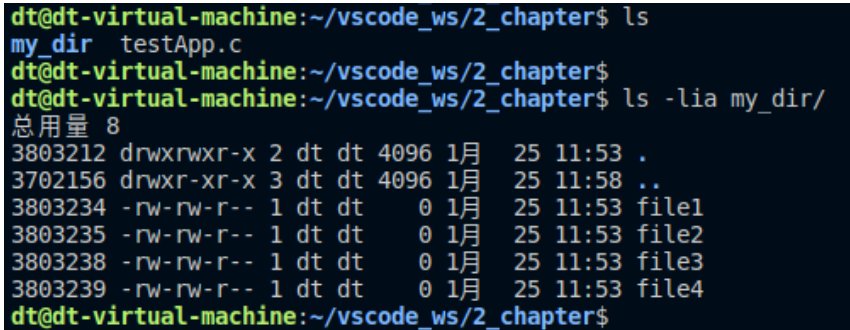
    /* 打开目录 */
    dirp = opendir("./my_dir");
    if (NULL == dirp) {
        perror("opendir error");
        exit(-1);
    }

    /* 循环读取目录流中的所有目录条目 */
    errno = 0;

    while (NULL != (dir = readdir(dirp)))
        printf("%s %ld\n", dir->d_name, dir->d_ino);
    if (0 != errno) {
        perror("readdir error");
        ret = -1;
        goto err;
    } else
        printf("End of directory!\n");

err:
    closedir(dirp);
    exit(ret);
}
```

使用 `opendir()` 打开了当前目录下的 `my_dir` 目录, 该目录下的文件如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
my_dir  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -lia my_dir/
总用量 8
3803212 drwxrwxr-x 2 dt dt 4096 1月 25 11:53 .
3702156 drwxr-xr-x 3 dt dt 4096 1月 25 11:58 ..
3803234 -rw-rw-r-- 1 dt dt  0 1月 25 11:53 file1
3803235 -rw-rw-r-- 1 dt dt  0 1月 25 11:53 file2
3803238 -rw-rw-r-- 1 dt dt  0 1月 25 11:53 file3
3803239 -rw-rw-r-- 1 dt dt  0 1月 25 11:53 file4
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.3 `my_dir` 目录下的文件列表

接下来编译、运行:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
my_dir  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
. 3803212
file4 3803239
file1 3803234
file3 3803238
file2 3803235
.. 3702156
End of directory!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.4 运行测试程序

由此可知, 示例代码 5.8.4 能够将 my_dir 目录下的所有文件全部扫描出来, 打印出它们的名字以及 inode 节点。

5.8.4 进程的当前工作目录

Linux 下的每一个进程都有自己的当前工作目录 (current working directory), 当前工作目录是该进程解析、搜索相对路径名的起点 (不是以 "/" 斜杆开头的绝对路径)。譬如, 代码中调用 open 函数打开文件时, 传入的文件路径使用相对路径方式进行表示, 那么该进程解析这个相对路径名时、会以进程的当前工作目录作为参考目录。

一般情况下, 运行一个进程时、其父进程的当前工作目录将被该进程所继承, 成为该进程的当前工作目录。可通过 getcwd 函数来获取进程的当前工作目录, 如下所示:

```
#include <unistd.h>
```

```
char *getcwd(char *buf, size_t size);
```

这是一个系统调用, 使用该函数之前, 需要包含头文件 <unistd.h>。

函数参数和返回值含义如下:

buf: getcwd() 将内含当前工作目录绝对路径的字符串存放在 buf 缓冲区中。

size: 缓冲区的大小, 分配的缓冲区大小必须要大于字符串长度, 否则调用将会失败。

返回值: 如果调用成功将返回指向 buf 的指针, 失败将返回 NULL, 并设置 errno。

Tips: 若传入的 buf 为 NULL, 且 size 为 0, 则 getcwd() 内部会按需分配一个缓冲区, 并将指向该缓冲区的指针作为函数的返回值, 为了避免内存泄漏, 调用者使用完之后必须调用 free() 来释放这一缓冲区所占内存空间。

测试

接下来, 我们编写一个简单地测试程序用于读取进程的当前工作目录:

示例代码 5.8.5 getcwd 函数测试例程

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char buf[100];
    char *ptr;
```

```

memset(buf, 0x0, sizeof(buf));

ptr = getcwd(buf, sizeof(buf));
if (NULL == ptr) {
    perror("getcwd error");
    exit(-1);
}

printf("Current working directory: %s\n", buf);
exit(0);
}

```

编译运行:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ pwd
/home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Current working directory: /home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 5.8.5 测试结果

改变当前工作目录

系统调用 `chdir()` 和 `fchdir()` 可以用于更改进程的当前工作目录, 函数原型如下所示:

```
#include <unistd.h>
```

```
int chdir(const char *path);
```

```
int fchdir(int fd);
```

首先, 使用这两个函数之一需要包含头文件 `<unistd.h>`。

函数参数和返回值含义如下:

path: 将进程的当前工作目录更改为 `path` 参数指定的目录, 可以是绝对路径、也可以是相对路径, 指定的目录必须要存在, 否则会报错。

fd: 将进程的当前工作目录更改为 `fd` 文件描述符所指定的目录 (譬如使用 `open` 函数打开一个目录)。

返回值: 成功均返回 0; 失败均返回 -1, 并设置 `errno`。

此两函数的区别在于, 指定目录的方式不同, `chdir()` 是以路径的方式进行指定, 而 `fchdir()` 则是通过文件描述符, 文件描述符可调用 `open()` 打开相应的目录时获得。

测试

示例代码 5.8.6 chdir 函数使用示例

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

```

```
int main(void)
{
    char buf[100];
    char *ptr;
    int ret;

    /* 获取更改前的工作目录 */
    memset(buf, 0x0, sizeof(buf));
    ptr = getcwd(buf, sizeof(buf));
    if (NULL == ptr) {
        perror("getcwd error");
        exit(-1);
    }

    printf("Before the change:  %s\n", buf);

    /* 更改进程的当前工作目录 */
    ret = chdir("./new_dir");
    if (-1 == ret) {
        perror("chdir error");
        exit(-1);
    }

    /* 获取更改后的工作目录 */
    memset(buf, 0x0, sizeof(buf));
    ptr = getcwd(buf, sizeof(buf));
    if (NULL == ptr) {
        perror("getcwd error");
        exit(-1);
    }

    printf("After the change:  %s\n", buf);
    exit(0);
}
```

上述程序会在更改工作目录之前获取当前工作目录、并将其打印出来，之后调用 `chdir` 函数将进程的工作目录更改为当前目录下的 `new_dir` 目录，更改成功之后再将进程的当前工作目录获取并打印出来，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
new_dir testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Before the change: /home/dt/vscode_ws/2_chapter
After the change: /home/dt/vscode_ws/2_chapter/new_dir
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.6 编译运行

5.9 删除文件

前面给大家介绍了如何删除一个目录，使用 `rmdir()` 函数即可，显然该函数并不能删除一个普通文件，那如何删除一个普通文件呢？方法就是通过系统调用 `unlink()` 或使用 C 库函数 `remove()`。

使用 `unlink` 函数删除文件

`unlink()` 用于删除一个文件（不包括目录），函数原型如下所示：

```
#include <unistd.h>
```

```
int unlink(const char *pathname);
```

使用该函数需要包含头文件 `<unistd.h>`。

函数参数和返回值含义如下：

pathname: 需要删除的文件路径，可使用相对路径、也可使用绝对路径，如果 `pathname` 参数指定的文件不存在，则调用 `unlink()` 失败。

返回值: 成功返回 0；失败将返回 -1，并设置 `errno`。

前面给大家介绍 `link` 函数，用于创建一个硬链接文件，创建硬链接时，`inode` 节点上的链接数就会增加；`unlink()` 的作用与 `link()` 相反，`unlink()` 系统调用用于移除/删除一个硬链接（从其父级目录下删除该目录条目）。

所以 `unlink()` 系统调用实质上是移除 `pathname` 参数指定的文件路径对应的目录项（从其父级目录中移除该目录项），并将文件的 `inode` 链接计数减 1，如果该文件还有其它硬链接，则任可通过其它链接访问该文件的数据；只有当链接计数变为 0 时，该文件的内容才可被删除。另一个条件也会阻止删除文件的内容--只要有进程打开了该文件，其内容也不能被删除。关闭一个文件时，内核会检查打开该文件的进程个数，如果这个计数达到 0，内核再去检查其链接计数，如果链接计数也是 0，那么就删除该文件对应的内容（也就是文件对应的 `inode` 以及数据块被回收，如果一个文件存在多个硬链接，删除其中任何一个硬链接，其 `inode` 和数据块并没有被回收，还可通过其它硬链接访问文件的数据）。

`unlink()` 系统调用并不会对软链接进行解引用操作，若 `pathname` 指定的文件为软链接文件，则删除软链接文件本身，而非软链接所指定的文件。

测试

示例代码 5.9.1 `unlink` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int ret;
```

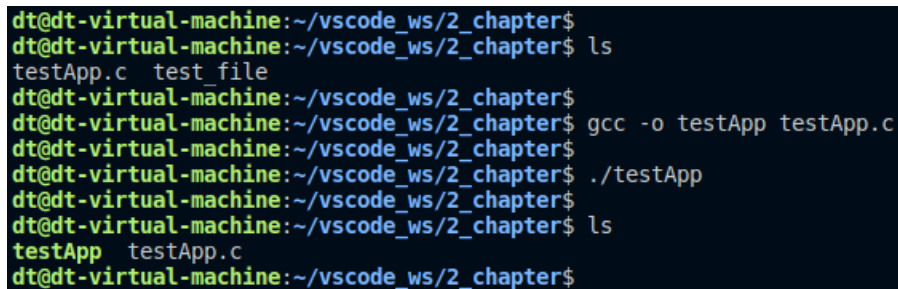
```

ret = unlink("./test_file");
if (-1 == ret) {
    perror("unlink error");
    exit(-1);
}

exit(0);
}

```

上述代码调用 `unlink()` 删除当前目录下的 `test_file` 文件，编译测试：



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 5.9.1 unlink 删除文件

使用 `remove` 函数删除文件

`remove()` 是一个 C 库函数，用于移除一个文件或空目录，其函数原型如下所示：

```
#include <stdio.h>
```

```
int remove(const char *pathname);
```

使用该函数需要包含 C 库函数头文件 `<stdio.h>`。

函数参数和返回值含义如下：

pathname: 需要删除的文件或目录路径，可以是相对路径、也可是决定路径。

返回值: 成功返回 0；失败将返回 -1，并设置 `errno`。

`pathname` 参数指定的是一个非目录文件，那么 `remove()` 去调用 `unlink()`，如果 `pathname` 参数指定的是一个目录，那么 `remove()` 去调用 `rmdir()`。

与 `unlink()`、`rmdir()` 一样，`remove()` 不对软链接进行解引用操作，若 `pathname` 参数指定的是一个软链接文件，则 `remove()` 会删除链接文件本身、而非所指向的文件。

测试

示例代码 5.9.2 `remove` 函数使用示例

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int ret;

    ret = remove("./test_file");
    if (-1 == ret) {

```

```

        perror("remove error");
        exit(-1);
    }

    exit(0);
}

```

5.10 文件重命名

本小节给大家介绍 `rename()` 系统调用, 借助于 `rename()` 既可以对文件进行重命名, 又可以将文件移至同一文件系统中的另一个目录下, 其函数原型如下所示:

```
#include <stdio.h>
```

```
int rename(const char *oldpath, const char *newpath);
```

使用该函数需要包含头文件 `<stdio.h>`。

函数参数和返回值含义如下:

oldpath: 原文件路径。

newpath: 新文件路径。

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

调用 `rename()` 会将现有的一个路径名 `oldpath` 重命名为 `newpath` 参数所指定的路径名。 `rename()` 调用仅操作目录条目, 而不移动文件数据 (不改变文件 `inode` 编号、不移动文件数据块中存储的内容), 重命名既不影响指向该文件的其它硬链接, 也不影响已经打开该文件的进程 (譬如, 在重命名之前该文件已被其它进程打开了, 而且还未被关闭)。

根据 `oldpath`、`newpath` 的不同, 有以下不同的情况需要进行说明:

- 若 `newpath` 参数指定的文件或目录已经存在, 则将其覆盖;
- 若 `newpath` 和 `oldpath` 指向同一个文件, 则不发生变化 (且调用成功)。
- `rename()` 系统调用对其两个参数中的软链接均不进行解引用。如果 `oldpath` 是一个软链接, 那么将重命名该软链接; 如果 `newpath` 是一个软链接, 则会将其移除、被覆盖。
- 如果 `oldpath` 指代文件, 而非目录, 那么就不能将 `newpath` 指定为一个目录的路径名。要想重命名一个文件到某一个目录下, `newpath` 必须包含新的文件名。
- 如果 `oldpath` 指代为一个目录, 在这种情况下, `newpath` 要么不存在, 要么必须指定为一个空目录。
- `oldpath` 和 `newpath` 所指代的文件必须位于同一文件系统。由前面的介绍, 可以得出此结论!
- 不能对 `.` (当前目录) 和 `..` (上一级目录) 进行重命名。

测试

示例代码 5.10.1 `rename` 函数使用示例

```

#include <stdio.h>
#include <stdlib.h>

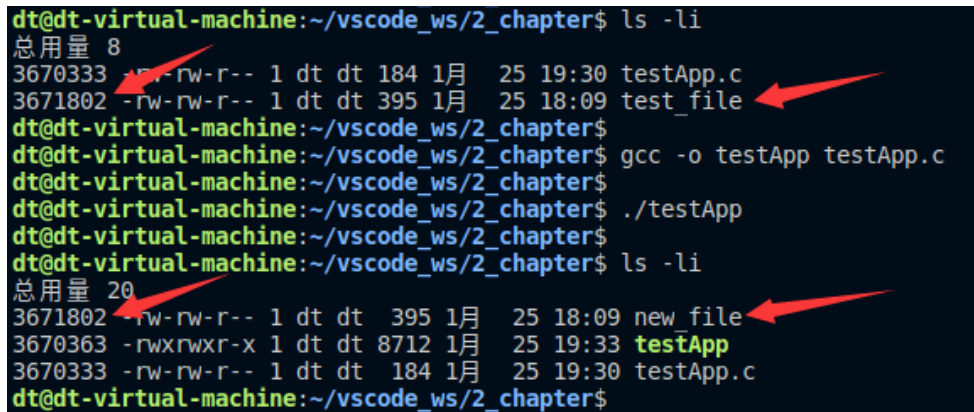
int main(void)
{
    int ret;

    ret = rename("./test_file", "./new_file");
}

```

```
if (-1 == ret) {  
    perror("rename error");  
    exit(-1);  
}  
  
exit(0);  
}
```

将当前目录下的 test_file 文件重命名为 new_file, 接下来编译测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li  
总用量 8  
3670333 -rw-rw-r-- 1 dt dt 184 1月 25 19:30 testApp.c  
3671802 -rw-rw-r-- 1 dt dt 395 1月 25 18:09 test_file  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li  
总用量 20  
3671802 -rw-rw-r-- 1 dt dt 395 1月 25 18:09 new file  
3670363 -rwxrwxr-x 1 dt dt 8712 1月 25 19:33 testApp  
3670333 -rw-rw-r-- 1 dt dt 184 1月 25 19:30 testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.10.1 rename 重命名

从图中可以知道, 使用 rename 进行文件重命名之后, 其 inode 号并未改变。

5.11 总结

本章所介绍的内容比较多, 主要是围绕文件属性以及目录展开的一系列相关话题, 本章开头先给大家介绍 Linux 系统下的 7 种文件类型, 包括普通文件、目录、设备文件 (字符设备文件、块设备文件)、符号链接文件 (软链接文件)、管道文件以及套接字文件。

接着围绕 stat 系统调用, 详细给大家介绍了 struct stat 结构体中的每一个成员, 这使得我们对 Linux 下文件的各个属性都有所了解。接着分别给大家详细介绍了文件属主、文件访问权限、文件时间戳、软链接与硬链接以及目录等相关内容, 让大家知道在应用编程中如何去修改文件的这些属性以及它们所需要满足的条件。

至此, 本章内容到这里就结束了, 相信大家已经学习到了不少知识内容, 大家加油!

第六章 字符串处理

字符串处理在几乎所有的编程语言中都是一个绕不开的话题, 在一些高级语言当中, 对字符串的处理支持度更是完善, 譬如 C++、C#、Python 等。若在 C 语言中想要对字符串进行相关的处理, 譬如将两个字符串进行拼接、字符串查找、两个字符串进行比较等操作, 几乎是需要程序员自己编写字符串处理相关逻辑代码来实现字符串处理功能。

好在 C 语言库函数中已经给我们提供了丰富的字符串处理相关函数, 基本常见的字符串处理需求都可以直接使用这些库函数来实现, 而不需要自己编写代码, 使用这些库函数可以大大减轻编程负担。这些库函数大致可以分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索等几类, 本章将向大家介绍这些库函数的使用方法。

本章将会讨论如下主题内容。

- 字符串输入/输出;
- C 库中提供的字符串处理函数;
- 给应用程序传参;
- 正则表达式。

6.1 字符串输入/输出

在程序当中,经常需要在程序运行过程中打印出一些信息,将其输出显示到标准输出设备 `stdout` (譬如屏幕)或标准错误设备 `stderr` (譬如屏幕),譬如调试信息、报错信息、中间产生的变量的值等等,以实现程序运行状态的掌控和分析。除了向 `stdout` 或 `stderr` 输出打印信息之外,有时程序在运行过程中还需要从标准输入设备 `stdin` (譬如键盘)中读取字符串,将读取到的字符串进行解析,以指导程序的下一步动作、控制程序执行流程。

6.1.1 字符串输出

常用的字符串输出函数有 `putchar()`、`puts()`、`fputc()`、`fputs()`,前面我们经常使用 `printf()`函数来输出字符串信息,而并没有使用到 `putchar()`、`puts()`、`fputc()`、`fputs()`这些函数,原因在于 `printf()`可以按照自己规定的格式输出字符串信息,一般称为格式化输出;而 `putchar()`、`puts()`、`fputc()`、`fputs()`这些函数只能输出字符串,不能进行格式转换。所以由此可知,`printf()`在功能上要比 `putchar()`、`puts()`、`fputc()`、`fputs()`这些函数更加强大,往往在实际编程中,`printf()`用的也会更多,但是 `putchar()`、`puts()`、`fputc()`、`fputs()`这些库函数相比与 `printf()`,在使用上方便、简单。

与 `printf()`一样,`putchar()`、`puts()`、`fputc()`、`fputs()`这些函数也是标准 I/O 函数,属于标准 C 库函数,所以需要包含头文件 `<stdio.h>`,并且它们也使用 `stdio` 缓冲。

puts 函数

`puts()`函数用来向标准输出设备(屏幕、显示器)输出字符串并自行换行。把字符串输出到标准输出设备,将 `'\0'` 转换为换行符 `'\n'`。`puts` 函数原型如下所示(可通过 `"man 3 puts"` 命令查看):

```
#include <stdio.h>
```

```
int puts(const char *s);
```

使用该函数需要包含头文件 `<stdio.h>`。

函数参数和返回值含义如下:

s: 需要进行输出的字符串。

返回值: 成功返回一个非负数;失败将返回 EOF, EOF 其实就是 -1。

使用 `puts()`函数连换行符 `'\n'`都省了,函数内部会自动在其后添加一个换行符。所以,如果只是单纯输出字符串到标准输出设备,而不包含数字格式化转换操作,那么使用 `puts()`会更加方便、简洁;`puts()`虽然方便、简单,但也仅限于输出字符串,功能还是没有 `printf()`强大。

puts 函数测试

示例代码 6.1.1 puts 函数使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    char str[50] = "Linux app puts test";
```

```
    puts("Hello World!");
```

```
    puts(str);
```

```
    exit(0);
```

}

编译运行结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
Hello World!  
Linux app puts test  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.1 puts 输出字符串

putchar 函数

putchar()函数可以把参数 c 指定的字符（一个无符号字符）输出到标准输出设备，其输出可以是一个字符，可以是介于 0~127 之间的一个十进制整型数（包含 0 和 127，输出其对应的 ASCII 码字符），也可以是用 char 类型定义好的一个字符型变量。putchar 函数原型如下所示（可通过"man 3 putchar"命令查看）：

```
#include <stdio.h>
```

```
int putchar(int c);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

c: 需要进行输出的字符。

返回值: 出错将返回 EOF。

putchar 函数测试

示例代码 6.1.2 putchar 函数使用示例

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    putchar('A');  
    putchar('B');  
    putchar('C');  
    putchar('D');  
    putchar('\n');  
    exit(0);  
}
```

编译运行结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
ABCD  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.2 putchar 输出字符

fputc 函数

fputc()与 putchar()类似,也用于输出参数 c 指定的字符(一个无符号字符),与 putchar()区别在于,putchar()只能输出到标准输出设备,而 fputc()可把字符输出到指定的文件中,既可以是标准输出、标准错误设备,也可以是一个普通文件。

fputc 函数原型如下所示:

```
#include <stdio.h>
```

```
int fputc(int c, FILE *stream);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下:

c: 需要进行输出的字符。

stream: 文件指针。

返回值: 成功时返回输出的字符; 出错将返回 EOF。

fputc 测试

(1)使用 fputc 函数将字符输出到标准输出设备。

示例代码 6.1.3 fputc 输出字符到标准输出设备

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    fputc('A', stdout);
```

```
    fputc('B', stdout);
```

```
    fputc('C', stdout);
```

```
    fputc('D', stdout);
```

```
    fputc('\n', stdout);
```

```
    exit(0);
```

```
}
```

编译运行:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
ABCD  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.3 fputc 测试结果 1

(2)使用 fputc 函数将字符输出到一个普通文件。

示例代码 6.1.4 fputc 输出字符到普通文件

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void)  
{  
    FILE *fp = NULL;  
  
    /* 创建一个文件 */  
    fp = fopen("./new_file", "a");  
    if (NULL == fp) {  
        perror("fopen error");  
        exit(-1);  
    }  
  
    /* 输入字符到文件 */  
    fputc('A', fp);  
    fputc('B', fp);  
    fputc('C', fp);  
    fputc('D', fp);  
    fputc('\n', fp);  
  
    /* 关闭文件 */  
    fclose(fp);  
    exit(0);  
}
```

编译运行, 结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
new_file testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat new_file
ABCD
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.4 fputs 测试结果 2

fputs 函数

同理, fputs()与 puts()类似, 也用于输出一条字符串, 与 puts()区别在于, puts()只能输出到标准输出设备, 而 fputs()可把字符串输出到指定的文件中, 既可以是标准输出、标准错误设备, 也可以是一个普通文件。

函数原型如下所示:

```
#include <stdio.h>
```

```
int fputs(const char *s, FILE *stream);
```

函数参数和返回值含义如下:

s: 需要输出的字符串。

stream: 文件指针。

返回值: 成功返回非负数; 失败将返回 EOF。

fputs 测试

(1)使用 fputs 输出字符串到标准输出设备。

示例代码 6.1.5 fputs 输出字符串到标准输出设备

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
    fputs("Hello World! 1\n", stdout);
    fputs("Hello World! 2\n", stdout);
    exit(0);
}
```

编译运行, 结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World! 1
Hello World! 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.5 fputs 测试结果 1

(2)使用 fputs 输出字符串到一个普通文件。

示例代码 6.1.6 fputs 输出字符串到普通文件

```
#include <stdio.h>
#include <stdlib.h>

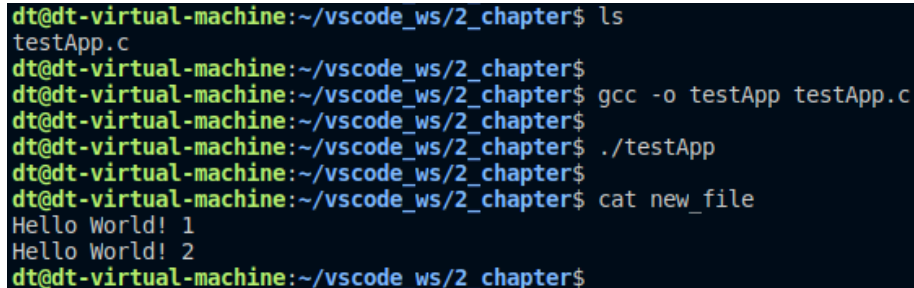
int main(void)
{
    FILE *fp = NULL;

    /* 创建一个文件 */
    fp = fopen("./new_file", "a");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    fputs("Hello World! 1\n", fp);
    fputs("Hello World! 2\n", fp);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

编译运行, 结果如下:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat new_file
Hello World! 1
Hello World! 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.6 fputs 测试结果 2

6.1.2 字符串输入

常用的字符串输入函数有 `gets()`、`getchar()`、`fgetc()`、`fgets()`。与 `printf()` 对应, 在 C 库函数中同样也提供了格式化输入函数 `scanf()`。scanf() 与 `gets()`、`getchar()`、`fgetc()`、`fgets()` 这些函数相比, 在功能上确实有它的优势, 但是在使用上不如它们方便、简单、更易于使用。

与 `scanf()` 一样, `gets()`、`getchar()`、`fgetc()`、`fgets()` 这些函数也是标准 I/O 函数, 属于标准 C 库函数, 所以需要包含头文件 `<stdio.h>`, 并且它们也使用 `stdio` 缓冲。

gets 函数

`gets()` 函数用于从标准输入设备 (譬如键盘) 中获取用户输入的字符串, `gets()` 函数原型如下所示:

```
#include <stdio.h>

char *gets(char *s);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下:

s: 指向字符数组的指针, 用于存储字符串。

返回值: 如果成功, 该函数返回指向 s 的指针; 如果发生错误或者到达末尾时还未读取任何字符, 则返回 NULL。

用户从键盘输入的字符串数据首先会存放在一个输入缓冲区中, gets()函数会从输入缓冲区中读取字符串存储到字符指针变量 s 所指向的内存空间, 当从输入缓冲区中读走字符后, 相应的字符便不存在于缓冲区了。

输入的字符串中就算是有空格也可以直接输入, 字符串输入完成之后按回车即可, gets()函数不检查缓冲区溢出。

使用示例

使用 gets()函数获取用户输入字符串。

示例代码 6.1.7 gets 函数使用示例

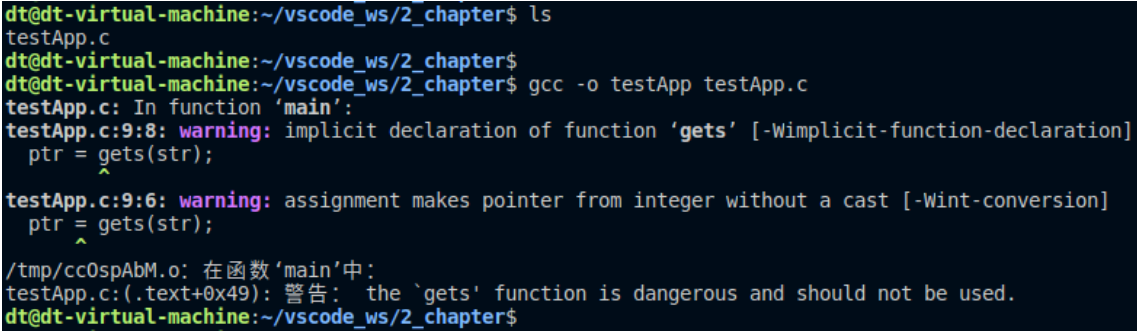
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char str[100] = {0};
    char *ptr = NULL;

    ptr = gets(str);
    if (NULL == ptr)
        exit(-1);
    puts(str);

    exit(0);
}
```

当在 Ubuntu 系统编译代码时, 会出现如下警告信息:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
testApp.c: In function 'main':
testApp.c:9:8: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
    ptr = gets(str);
           ^
testApp.c:9:6: warning: assignment makes pointer from integer without a cast [-Wint-conversion]
    ptr = gets(str);
           ^
/tmp/cc0spAbM.o: 在函数 'main' 中:
testApp.c:(.text+0x49): 警告: the `gets' function is dangerous and should not be used.
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.7 编译代码出现警告信息

出现如上警告信息, 其实是建议我们不要使用 gets()函数, 因为程序中使用 gets()函数是非常不安全的, 可能会出现 bug、出现不可靠性, gets()在某些意外情况下会导致程序陷入不可控状态, 所以一般建议大家不要使用这个函数, 可以使用后面将给大家介绍的 fgets()代替。

这里先不管这个警告信息, 我们直接运行测试:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入字符串: aaa bbb ccc ddd
aaa bbb ccc ddd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入字符串: 'adsada' "dasdasdas"
'adsada' "dasdasdas"
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.8 gets 测试结果

由此可知, 不管我们输入的是空格、单引号、双引号都会作为 gets() 获取到的字符串的一部分, 直到用户输入回车换行符结束。

gets() 与 scanf() 的区别

gets() 除了在功能上不及 scanf 之外, 它们在一些细节上也存在着不同:

- gets() 函数不仅比 scanf 简洁, 而且, 就算输入的字符串中有空格也可以, 因为 gets() 函数允许输入的字符串带有空格、制表符, 输入的空格和制表符也是字符串的一部分, 仅以回车换行符作为字符串的分割符; 而对于 scanf 以 %s 格式输入的时候, 空格、换行符、TAB 制表符等都是作为字符串分割符存在, 即分隔符前后是两个字符串, 读取字符串时并不会将分隔符读取出来作为字符串的组成部分, 一个 %s 只能读取一个字符串, 若要多去多个字符串, 则需要使用多个 %s、并且需要使用多个字符数组存储。
- gets() 会将回车换行符从输入缓冲区中取出来, 然后将其丢弃, 所以使用 gets() 读走缓冲区中的字符串数据之后, 缓冲区中将不会遗留下回车换行符; 而对于 scanf() 来说, 使用 scanf() 读走缓冲区中的字符串数据时, 并不会将分隔符 (空格、TAB 制表符、回车换行符等) 读走将其丢弃, 所以使用 scanf() 读走缓冲区中的字符串数据之后, 缓冲区中依然还存在用户输入的分隔符。

针对上面所提出的两个区别点, 下面我们将进行一些列的代码测试。

(1) 测试 1

示例代码 6.1.8 测试代码 1

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char s1[100] = {0};
    char s2[100] = {0};

    scanf("%s", s1);
    printf("s1: %s\n", s1);

    scanf("%s", s2);
    printf("s2: %s\n", s2);

    exit(0);
}
```

当输入 123_456 回车时, 输出结果如下 (_ 表示空格):

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123 456
s1: 123
s2: 456
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.9 测试结果 1

代码中我们调用了两次 `scanf()`，而事实上我们只输入了一次，输入“123”之后输入空格、再输入“456”，然后按回车，由打印结果可知，字符串 `s1` 等于“123”，字符串 `s2` 等于“456”；当输入完成按回车之后，输入缓冲区中此时存在如下字符：

```
'1'、'2'、'3'、'空格'、'4'、'5'、'6'、'\n'
```

第一个 `scanf()` 读取缓冲区时，将 '1'、'2'、'3' 读走，读走之后，它们将不存在于缓冲区中了，空格被视为字符串分割符，分割符及后面的字符将不会读取（以 `%s` 格式输入情况下）。

第二个 `scanf()` 读取缓冲区时，'4'、'5'、'6' 会被读走，分割符依然不读取。

再次执行测试程序：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123
s1: 123
456
s2: 456
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.10 测试结果 2

当输入“123”回车，之后输出了“123”；之后需要再次输入，接着输入“456”回车，输出“456”。这里输入了两次字符串，原因在于第一次 `scanf()` 读走“123”之后，缓冲区中只剩下回车换行字符，第二次 `scanf()` 不读取换行符，所以需要用户再次输入字符串。

(2)测试 2

示例代码 6.1.9 测试代码 2

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char s[100] = {0};
    char c;

    scanf("%s", s);
    printf("s: %s\n", s);

    scanf("%c", &c);
    printf("c: %d\n", c);

    exit(0);
}
```

执行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123 → 输入123回车
s: 123
c: 10
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.11 测试结果 3

同样这段代码也是调用了两次 `scanf()`，但只是输入了一次字符串，当第一个 `scanf()` 读取之后，缓冲区中只剩下回车换行符；从打印信息可以发现，第二次 `scanf()` 读取时，把换行符也读取出来了（换行符 '\n' 对应的 ASCII 编码值等于 10），因为这里 `scanf` 用的是 `%c` 格式，而不是 `%s`，对于 `%c` 读入时，空格、换行符、TAB 这些都是正常字符。

(3)测试 3

示例代码 6.1.10 测试代码 3

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char s1[100] = {0};
    char s2[100] = {0};

    scanf("%s", s1);
    printf("s1: %s\n", s1);

    gets(s2);
    printf("s2: %s\n", s2);

    exit(0);
}
```

执行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123 → 输入123回车
s1: 123
s2:
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.12 测试结果 4

这段代码先是调用了 `scanf()`，之后调用了 `gets()`，但只输入了一次字符串。`scanf()` 读取之后，缓冲区中只剩下换行符，但是 `gets()` 会将换行符读取出来并将其丢弃，所以说字符串便是一个空字符串。

再次执行测试程序：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123 456
s1: 123
s2: 456
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

输入: 123两个空格456回车

图 6.1.13 测试结果 5

字符串 s1 依然是“123”，scanf 读取完之后，缓冲区此时剩下如下字符串：

'空格'、'空格'、'4'、'5'、'6'、'\n'

gets() 读取时将两个空格以及“456”、换行符全部读取出来，其中换行符会被丢弃、不作为字符串的组成字符，所以字符串 s2 前面就会存在两个空格。

getchar 函数

getchar() 函数用于从标准输入设备中读取一个字符（一个无符号字符），函数原型如下所示：

```
#include <stdio.h>
```

```
int getchar(void);
```

使用该函数需要包含头文件 <stdio.h>。

函数参数和返回值含义如下：

无需传参。

返回值：该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

同样 getchar() 函数也是从输入缓冲区读取字符数据，但只读取一个字符，包括空格、TAB 制表符、换行回车符等。

测试

示例代码 6.1.11 getchar 函数使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    int ch;
```

```
    ch = getchar();
```

```
    printf("ch: %c\n", ch);
```

```
    exit(0);
```

```
}
```

执行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123
ch: 1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.14 测试结果 6

getchar() 只从输入缓冲区中读取一个字符, 与 scanf 以 %c 格式读取一样, 空格、TAB 制表符、回车符都将是正常的字符。即使输入了多个字符, 但 getchar() 仅读取一个字符。

fgets 函数

fgets() 与 gets() 一样用于获取输入的字符串, fgets() 函数原型如下所示

```
#include <stdio.h>
```

```
char *fgets(char *s, int size, FILE *stream);
```

使用该函数需要包含头文件 <stdio.h>。

函数参数和返回值含义如下:

s: 指向字符数组的指针, 用于存储字符串。

size: 这是要读取的最大字符数。

stream: 文件指针。

fgets() 与 gets() 的区别主要是三点:

- gets() 只能从标准输入设备中获取输入字符串, 而 fgets() 既可以从标准输入设备获取字符串、也可以从一个普通文件中获取输入字符串。
- fgets() 可以设置获取字符串的最大字符数。
- gets() 会将缓冲区中的换行符 '\n' 读取出来、将其丢弃、将 '\n' 替换为字符串结束符 '\0'; fgets() 也会将缓冲区中的换行符读取出来, 但并不丢弃, 而是作为字符串组成字符存在, 读取完成之后自动在最后添加字符串结束字符 '\0'。

其它方面与 gets() 函数一样, 包括前面给大家所介绍的与 scanf(%s) 在一些细节方面的区别。

测试

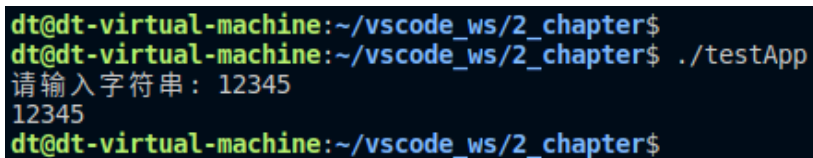
示例代码 6.1.12 fgets 从标准输入设备获取字符串

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char str[100] = {0};

    printf("请输入字符串: ");
    fgets(str, sizeof(str), stdin);
    printf("%s", str);
    exit(0);
}
```

执行测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入字符串: 12345
12345
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.15 测试结果

此段代码中, 使用 printf 打印字符串 str 时并没有在 %s 后面添加 '\n', 但是结果显示, 打印出来的字符串已经换行, 也就意味着 str 字符串本身就包含了换行符 '\n'。

示例代码 6.1.13 fgets 从普通文件中输入字符串

```
#include <stdio.h>
#include <stdlib.h>

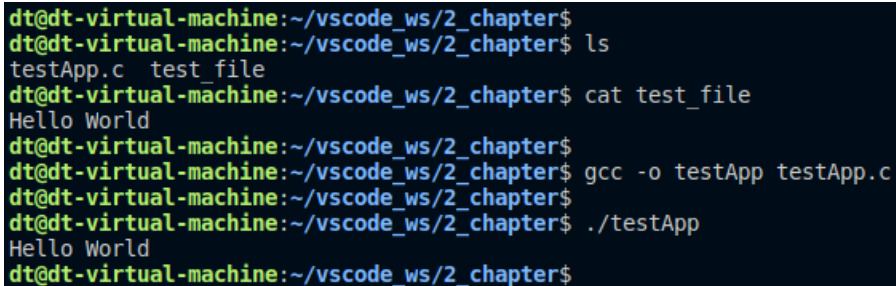
int main(void)
{
    char str[100] = {0};
    FILE *fp = NULL;

    /* 打开文件 */
    fp = fopen("./test_file", "r");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    /* 从文件中输入字符串 */
    fgets(str, sizeof(str), fp);
    printf("%s", str);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

使用 fgets() 读取文件中输入的字符串, 文件指针会随着读取的字节数向前移动。
执行测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.16 测试结果

fgetc 函数

fgetc() 与 getchar() 一样, 用于读取一个输入字符, 函数原型如下所示:

```
#include <stdio.h>
```

```
int fgetc(FILE *stream);
```

使用该函数需要包含头文件 <stdio.h>。

函数参数和返回值含义如下:

stream: 文件指针。

返回值: 该函数以无符号 char 强制转换为 int 的形式返回读取的字符, 如果到达文件末尾或发生读错误, 则返回 EOF。

fgetc()与 getchar()的区别在于, fgetc 可以指定输入字符的文件, 既可以从标准输入设备输入字符, 也可以从一个普通文件中输入字符, 其它方面与 getchar 函数相同。

测试

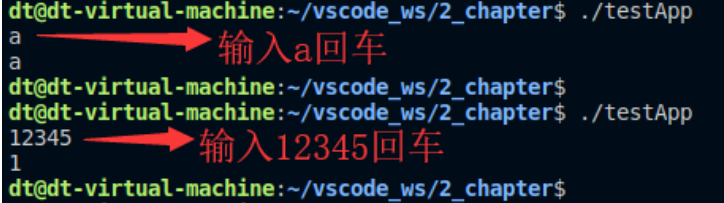
示例代码 6.1.14 fgetc 从标准输入设备中输入字符

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int ch;

    ch = fgetc(stdin);
    printf("%c\n", ch);
    exit(0);
}
```

执行测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
a
a
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
12345
1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.17 测试结果

示例代码 6.1.15 fgetc 从普通文件中输入字符

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int ch;
    FILE *fp = NULL;

    /* 打开文件 */
    fp = fopen("./test_file", "r");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    /* 从文件中输入一个字符 */
    ch = fgetc(fp);
    printf("%c\n", ch);
}
```

```

/* 关闭文件 */
fclose(fp);
exit(0);
}

```

测试结果:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
H
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 6.1.18 测试结果

6.1.3 总结

本小节给大家介绍了一些字符串输入、输出相关的 C 库函数，涉及到的函数比较多，在实际的编程当中，需要根据自己的实际需求以及函数的适用情况进行选择。

6.2 字符串长度

C 语言函数库中提供了一个用于计算字符串长度的函数 `strlen()`，其函数原型如下所示：

```
#include <string.h>
```

```
size_t strlen(const char *s);
```

使用该函数需要包含头文件 `<string.h>`。

函数参数和返回值含义如下：

s: 需要进行长度计算的字符串，字符串必须包含结束字符 `'\0'`。

返回值: 返回字符串长度（以字节为单位），字符串结束字符 `'\0'` 不计算在内。

测试

示例代码 6.2.1 `strlen` 使用示例

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str[] = "Linux app strlen test!";

    printf("String: \"%s\"\n", str);
    printf("Length: %ld\n", strlen(str));
    exit(0);
}

```

运行结果:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
String: "Linux app strlen test!"
Length: 22
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.2.1 strlen 计算字符串长度

sizeof 和 strlen 的区别

在程序当中, 我们通常也会使用 sizeof 来计算长度, 那 strlen 和 sizeof 有什么区别呢?

- sizeof 是 C 语言内置的操作符关键字, 而 strlen 是 C 语言库函数;
- sizeof 仅用于计算数据类型的大小或者变量的大小, 而 strlen 只能以结尾为'\0'的字符串作为参数;
- 编译器在编译时就计算出了 sizeof 的结果, 而 strlen 必须在运行时才能计算出来;
- sizeof 计算数据类型或变量会占用内存的大小, strlen 计算字符串实际长度。

sizeof 和 strlen 测试

示例代码 6.2.2 strlen 和 sizeof 对比测试

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str[50] = "Linux app strlen test!";
    char *ptr = str;

    printf("sizeof: %ld\n", sizeof(str));
    printf("strlen: %ld\n", strlen(str));
    puts("~~~~~");
    printf("sizeof: %ld\n", sizeof(ptr));
    printf("strlen: %ld\n", strlen(ptr));
    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
sizeof: 50
strlen: 22
~~~~~
sizeof: 8
strlen: 22
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.2.2 strlen 和 sizeof 对比测试结果

从打印信息可知, 第一个 `sizeof` 计算的是数组变量 `str` 的大小, 所以等于 50; 而第二个 `sizeof` 计算的是指针变量 `ptr` 的大小, 这里等于 8 个字节, 因为这里笔者是在 Ubuntu 64 位系统下进行的测试, 所以指针占用的内存大小就等于 8 个字节; 而 `strlen` 始终计算的都是字符串的长度。

6.3 字符串拼接

C 语言函数库中提供了 `strcat()` 函数或 `strncat()` 函数用于将两个字符串连接 (拼接) 起来, `strcat` 函数原型如下所示:

```
#include <string.h>
```

```
char *strcat(char *dest, const char *src);
```

使用该函数需要包含头文件 `<string.h>`。

函数参数和返回值含义如下:

dest: 目标字符串。

src: 源字符串。

返回值: 返回指向目标字符串 `dest` 的指针。

`strcat()` 函数会把 `src` 所指向的字符串追加到 `dest` 所指向的字符串末尾, 所以必须要保证 `dest` 有足够的存储空间来容纳两个字符串, 否则会导致溢出错误; `dest` 末尾的 `'\0'` 结束字符会被覆盖, `src` 末尾的结束字符 `'\0'` 会一起被复制过去, 最终的字符串只有一个 `'\0'`。

strcat 测试

使用 `strcat` 函数将字符串 `str2` 连接到字符串 `str1` 末尾, 并将其打印出来。

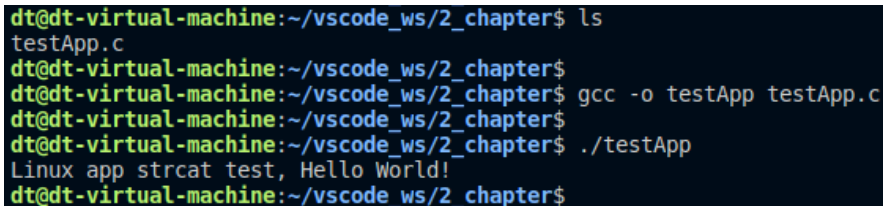
示例代码 6.3.1 `strcat` 使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str1[100] = "Linux app strcat test, ";
    char str2[] = "Hello World!";

    strcat(str1, str2);
    puts(str1);
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Linux app strcat test, Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.3.1 `strcat` 测试结果

`strncat` 函数

strncat()与 strcat()的区别在于, strncat 可以指定源字符串追加到目标字符串的字符数量, strncat 函数原型如下所示:

```
#include <string.h>
```

```
char *strncat(char *dest, const char *src, size_t n);
```

函数参数和返回值含义如下:

dest: 目标字符串。

src: 源字符串。

n: 要追加的最大字符数。

返回值: 返回指向目标字符串 dest 的指针。

如果源字符串 src 包含 n 个或更多个字符, 则 strncat()将 n+1 个字节追加到 dest 目标字符串 (src 中的 n 个字符加上结束字符'\0')。

strncat 测试

示例代码 6.3.2 strncat 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str1[100] = "Linux app strcat test, ";
    char str2[] = "Hello World!";

    strncat(str1, str2, 5);
    puts(str1);
    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Linux app strcat test, Hello
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.3.2 strncat 测试结果

6.4 字符串拷贝

C 语言函数库中提供了 strcpy()函数和 strncpy()函数用于实现字符串拷贝, strcpy 函数原型如下所示:

```
#include <string.h>
```

```
char *strcpy(char *dest, const char *src);
```

函数参数和返回值含义如下:

dest: 目标字符串。

src: 源字符串。

返回值: 返回指向目标字符串 dest 的指针。

strcpy()会把 src (必须包含结束字符'\0') 指向的字符串复制 (包括字符串结束字符'\0') 到 dest, 所以必须保证 dest 指向的内存空间足够大, 能够容纳下 src 字符串, 否则会导致溢出错误。

strcpy 测试

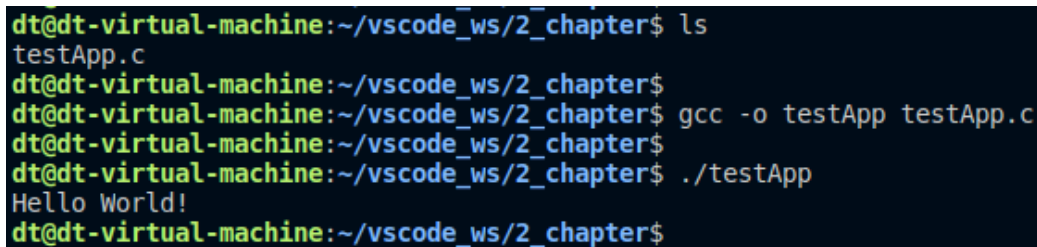
示例代码 6.4.1 strcpy 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str1[100] = {0};
    char str2[] = "Hello World!";

    strcpy(str1, str2);
    puts(str1);
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.4.1 strcpy 测试结果

strncpy 函数

strncpy()与 strcpy()的区别在于, strncpy()可以指定从源字符串 src 复制到目标字符串 dest 的字符数量, strncpy 函数原型如下所示:

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

函数参数和返回值含义如下:

dest: 目标字符串。

src: 源字符串。

n: 从 src 中复制的最大字符数。

返回值: 返回指向目标字符串 dest 的指针。

把 src 所指向的字符串复制到 dest, 最多复制 n 个字符。当 n 小于或等于 src 字符串长度 (不包括结束字符的长度) 时, 则复制过去的字符串中没有包含结束字符'\0'; 当 n 大于 src 字符串长度时, 则会将 src 字

字符串的结束字符 '\0' 也一并拷贝过去, 必须保证 `dest` 指向的内存空间足够大, 能够容纳下拷贝过来的字符串, 否则会导致溢出错误。

strncpy 函数测试

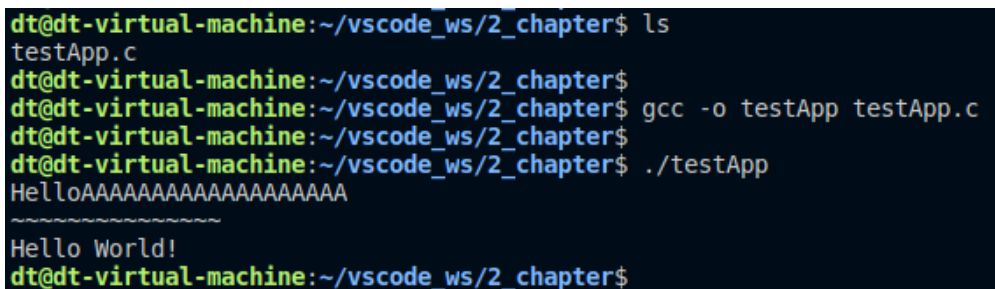
示例代码 6.4.2 strncpy 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str1[100] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAA";
    char str2[] = "Hello World!";

    strncpy(str1, str2, 5);
    puts(str1);
    puts("~~~~~");
    strncpy(str1, str2, 20);
    puts(str1);
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
HelloAAAAAAAAAAAAAAAAAAAA
~~~~~
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.4.2 strncpy 测试结果

memcpy、memmove、bcopy

除了 `strcpy()` 和 `strncpy()` 之外, 其实还可以使用 `memcpy()`、`memmove()` 以及 `bcopy()` 这些库函数实现拷贝操作, 字符串拷贝本质上也只是内存数据的拷贝, 所以这些库函数同样也是适用的, 在实际的编程当中, 这些库函数也是很常用的, 关于这三个库函数, 这里不再给大家介绍, 用法也非常简单, 需要注意的就是目标内存空间与源内存空间是否有重叠的问题。

关于三个库函数的使用方法, 大家可以使用 `man` 手册进行查询。

6.5 内存填充

在编程中, 经常需要将某一块内存中的数据全部设置为指定的值, 譬如在定义数组、结构体这种类型变量时, 通常需要对其进行初始化操作, 而初始化操作一般都是将其占用的内存空间全部填充为 0。

memset 函数

memset()函数用于将某一块内存的数据全部设置为指定的值, 其函数原型如下所示:

```
#include <string.h>
```

```
void *memset(void *s, int c, size_t n);
```

使用该函数需要包含头文件<string.h>。

函数参数和返回值含义如下:

s: 需要进行数据填充的内存空间起始地址。

c: 要被设置的值, 该值以 int 类型传递。

n: 填充的字节数。

返回值: 返回指向内存空间 s 的指针。

参数 c 虽然是以 int 类型传递, 但 memset()函数在填充内存块时是使用该值的无符号字符形式, 也就是函数内部会将该值转换为 unsigned char 类型的数据, 以字节为单位进行数据填充。

memset 测试

对数组 str 进行初始化操作, 将其存储的数据全部设置为 0。

示例代码 6.5.1 memset 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
```

```
{
```

```
    char str[100];
```

```
    memset(str, 0x0, sizeof(str));
```

```
    exit(0);
```

```
}
```

bzero 函数

bzero()函数用于将一段内存空间中的数据全部设置为 0, 函数原型如下所示:

```
#include <strings.h>
```

```
void bzero(void *s, size_t n);
```

函数参数和返回值含义如下:

s: 内存空间的起始地址。

n: 填充的字节数。

返回值: 无返回值。

bzero 测试

对数组 str 进行初始化操作, 将其存储的数据全部设置为 0。

示例代码 6.5.2 bzero 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
{
    char str[100];

    bzero(str, sizeof(str));
    exit(0);
}
```

6.6 字符串比较

C 语言函数库提供了用于字符串比较的函数 `strcmp()` 和 `strncmp()`, `strcmp()` 函数原型如下所示:

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

函数参数和返回值含义如下:

s1: 进行比较的字符串 1。

s2: 进行比较的字符串 2。

返回值:

- 如果返回值小于 0, 则表示 `str1` 小于 `str2`
- 如果返回值大于 0, 则表示 `str1` 大于 `str2`
- 如果返回值等于 0, 则表示字符串 `str1` 等于字符串 `str2`

`strcmp` 进行字符串比较, 主要是通过比较字符串中的字符对应的 ASCII 码值, `strcmp` 会根据 ASCII 编码依次比较 `str1` 和 `str2` 的每一个字符, 直到出现了不同的字符, 或者某一字符串已经到达末尾 (遇见了字符串结束字符 `'\0'`)。

strcmp 测试

示例代码 6.6.1 strcmp 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    printf("%d\n", strcmp("ABC", "ABC"));
    printf("%d\n", strcmp("ABC", "a"));
    printf("%d\n", strcmp("a", "ABC"));
    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0
-1
1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.6.1 strcmp 测试结果

strncmp 函数

strncmp()与 strcmp()函数一样,也用于对字符串进行比较操作,但最多比较前 n 个字符, strncmp()函数原型如下所示:

```
#include <string.h>
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

函数参数和返回值含义如下:

s1: 参与比较的第一个字符串。

s2: 参与比较的第二个字符串。

n: 最多比较前 n 个字符。

返回值: 返回值含义与 strcmp()函数相同。

strncmp 测试

示例代码 6.6.2 strncmp 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    printf("%d\n", strncmp("ABC", "ABC", 3));
    printf("%d\n", strncmp("ABC", "ABCD", 3));
    printf("%d\n", strncmp("ABC", "ABCD", 4));
    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0
0
-1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.6.2 strncmp 测试结果

6.7 字符串查找

字符串查找在平时的编程当中也是一种很常见的操作,譬如从一个给定的字符串当中查找某一个字符或者一个字符串,并获取它的位置。C 语言函数库中也提供了一些用于字符串查找的函数,包括 `strchr()`、`strrchr()`、`strstr()`、`strpbrk()`、`index()`以及 `rindex()`等。

strchr 函数

使用 `strchr()`函数可以查找到给定字符串当中的某一个字符,函数原型如下所示:

```
#include <string.h>
```

```
char *strchr(const char *s, int c);
```

函数参数和返回值含义如下:

s: 给定的目标字符串。

c: 需要查找的字符。

返回值: 返回字符 `c` 第一次在字符串 `s` 中出现的位置,如果未找到字符 `c`,则返回 `NULL`。

字符串结束字符 `'\0'` 也将作为字符串的一部分,因此,如果将参数 `c` 指定为 `'\0'`,则函数将返回指向结束字符的指针。`strchr` 函数在字符串 `s` 中从前到后(或者称为从左到右)查找字符 `c`,找到字符 `c` 第一次出现的位置就返回,返回值指向这个位置,如果找不到字符 `c` 就返回 `NULL`。

strchr 测试

从字符串中查找一个字符,并将其在字符串数组中的下标打印出来。

示例代码 6.7.1 `strchr` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *ptr = NULL;
    char str[] = "Hello World!";

    ptr = strchr(str, 'W');
    if (NULL != ptr) {
        printf("Character: %c\n", *ptr);
        printf("Offset: %ld\n", ptr - str);
    }

    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Character: W
Offset: 6
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.7.1 strchr 测试结果

strrchr 函数

strrchr()与 strchr()函数一样,它同样表示在字符串中查找某一个字符,返回字符第一次在字符串中出现的位置,如果没找到该字符,则返回值 NULL,但两者唯一不同的是,strrchr()函数在字符串中是从后到前(或者称为从右向左)查找字符,找到字符第一次出现的位置就返回,返回值指向这个位置,strrchr()函数原型如下所示:

```
#include <string.h>
```

```
char *strrchr(const char *s, int c);
```

函数参数和返回值含义与 strchr()函数相同。

strrchr 测试

编写程序测试 strrchr()与 strchr()之间的区别。

示例代码 6.7.2 strrchr()与 strchr()之间的区别

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *ptr = NULL;
    char str[] = "I love my home";

    ptr = strchr(str, 'o');
    if (NULL != ptr)
        printf("strchr: %ld\n", ptr - str);

    ptr = strrchr(str, 'o');
    if (NULL != ptr)
        printf("strrchr: %ld\n", ptr - str);

    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strchr: 3
strchr: 11
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.7.2 strchr 与 strchr 对比测试结果

strstr 函数

与 strchr()函数不同的是, strstr()可在给定的字符串 haystack 中查找第一次出现子字符串 needle 的位置, 不包含结束字符'\0', 函数原型如下所示:

```
#include <string.h>
```

```
char *strstr(const char *haystack, const char *needle);
```

函数参数和返回值含义如下:

haystack: 目标字符串。

needle: 需要查找的子字符串。

返回值: 如果目标字符串 haystack 中包含了子字符串 needle, 则返回该字符串首次出现的位置; 如果未能找到子字符串 needle, 则返回 NULL。

strstr 测试

在给定的字符串 "I love my home" 中, 查找 "home" 在该字符串中首次出现的位置。

示例代码 6.7.3 strstr 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *ptr = NULL;
    char str[] = "I love my home";

    ptr = strstr(str, "home");
    if (NULL != ptr) {
        printf("String: %s\n", ptr);
        printf("Offset: %ld\n", ptr - str);
    }

    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
String: home
Offset: 10
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.7.3 strstr 测试结果

其它函数

除了上面介绍的三个函数之外，C 函数库中还提供其它的字符串（或字符）查找函数，譬如 `strpbrk()`、`index()`以及 `rindex()`等，这里便不再给大家一一介绍了，这些函数的用法都比较简单，大家通过 `man` 手册可以快速了解到它们的使用方法。

6.8 字符串与数字互转

在编程中，经常会需要将数字组成的字符串转换为相应的数字、或者将数字转换为字符串，在 C 函数库中同样也提供了相应的函数，本小节就向大家介绍这些函数的用法。

6.8.1 字符串转整形数据

C 函数库中提供了一系列函数用于实现将一个字符串转为整形数据，主要包括 `atoi()`、`atol()`、`atoll()`以及 `strtol()`、`strtoll()`、`strtoul()`、`strtoull()`等，它们之间的区别主要包括以下两个方面：

- 数据类型（`int`、`long int`、`unsigned long` 等）。
- 不同进制方式表示的数字字符串（八进制、十六进制、十进制）。

atoi、atol、atoll 函数

`atoi()`、`atol()`、`atoll()`三个函数可用于将字符串分别转换为 `int`、`long int` 以及 `long long` 类型的数据，它们的函数原型如下：

```
#include <stdlib.h>
```

```
int atoi(const char *nptr);
```

```
long atol(const char *nptr);
```

```
long long atoll(const char *nptr);
```

使用这些函数需要包含头文件 `<stdlib.h>`。

函数参数和返回值含义如下：

nptr: 需要进行转换的字符串。

返回值: 分别返回转换之后得到的 `int` 类型数据、`long int` 类型数据以及 `long long` 类型数据。

目标字符串 `nptr` 中可以包含非数字字符，转换时跳过前面的空格字符（如果目标字符串开头存在空格字符），直到遇上数字字符或正负符号才开始做转换，而再遇到非数字或字符串结束时（`'\0'`）才结束转换，并将结果返回。

使用 `atoi()`、`atol()`、`atoll()`函数只能转换十进制表示的数字字符串，即 0~9。

测试

使用 `atoi()`、`atol()`、`atoll()`这三个函数将一个数字字符串转为十进制数据。

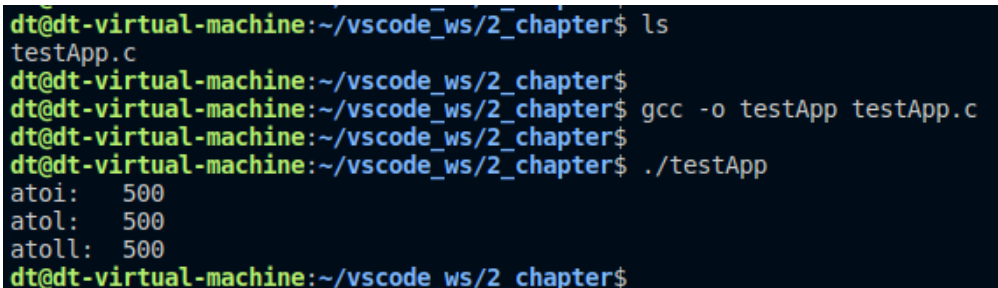
示例代码 6.8.1 `atoi`、`atol`、`atoll` 使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

int main(void)
{
    printf("atoi:  %d\n", atoi("500"));
    printf("atol:   %ld\n", atol("500"));
    printf("atoll:  %lld\n", atoll("500"));
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
atoi:   500
atol:    500
atoll:   500
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.1 atoi、atol、atoll 测试结果

strtol、strtoll 函数

strtol()、strtoll()两个函数可分别将字符串转为 long int 类型数据和 long long int 类型数据,与 atol()、atoll()之间的区别在于,strtol()、strtoll()可以实现将多种不同进制数(譬如二进制表示的数字字符串、八进制表示的数字字符串、十六进制表示的数字字符串)表示的字符串转换为整形数据,其函数原型如下所示:

```
#include <stdlib.h>
```

```
long int strtol(const char *nptr, char **endptr, int base);
```

```
long long int strtoll(const char *nptr, char **endptr, int base);
```

使用这两个函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下:

nptr: 需要进行转换的目标字符串。

endptr: char **类型的指针,如果 endptr 不为 NULL,则 strtol()或 strtoll()会将字符串中第一个无效字符的地址存储在*endptr 中。如果根本没有数字,strtol()或 strtoll()会将 nptr 的原始值存储在*endptr 中(并返回 0)。也可将参数 endptr 设置为 NULL,表示不接收相应信息。

base: 数字基数,参数 base 必须介于 2 和 36 (包含)之间,或者是特殊值 0。参数 base 决定了字符串转换为整数时合法字符的取值范围,譬如,当 base=2 时,合法字符为'0'、'1'(表示是一个二进制表示的数字字符串);当 base=8 时,合法字符为'0'、'1'、'2'、'3'.....'7'(表示是一个八进制表示的数字字符串);当 base=16 时,合法字符为'0'、'1'、'2'、'3'.....'9'、'a'.....'f'(表示是一个十六进制表示的数字字符串);当 base 大于 10 的时候,'a'代表 10、'b'代表 11、'c'代表 12,依次类推,'z'代表 35(不区分大小写)。

返回值: 分别返回转换之后得到的 long int 类型数据以及 long long int 类型数据。

需要进行转换的目标字符串可以以任意数量的空格或者 0 开头,转换时跳过前面的空格字符,直到遇上数字字符或正负符号('+或-')才开始做转换,而再遇到非数字或字符串结束时('/0')才结束转换,并将结果返回。

在 base=0 的情况下, 如果字符串包含一个“0x”前缀, 表示该数字将以 16 为基数; 如果包含的是“0”前缀, 表示该数字将以 8 为基数。

当 base=16 时, 字符串可以使用“0x”前缀。

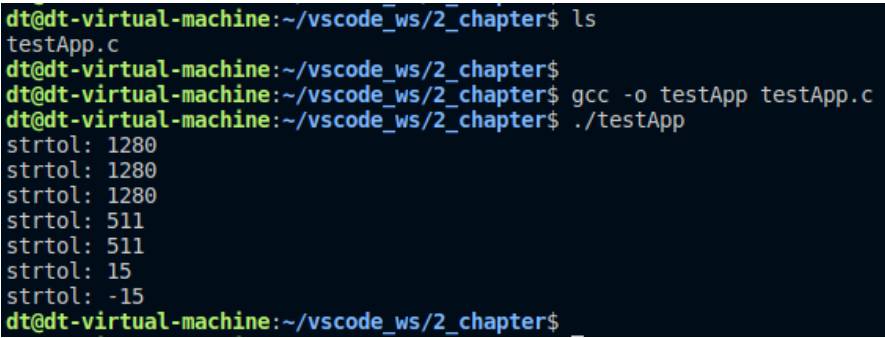
测试

示例代码 6.8.2 strtol 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    printf("strtol: %ld\n", strtol("0x500", NULL, 16));
    printf("strtol: %ld\n", strtol("0x500", NULL, 0));
    printf("strtol: %ld\n", strtol("500", NULL, 16));
    printf("strtol: %ld\n", strtol("0777", NULL, 8));
    printf("strtol: %ld\n", strtol("0777", NULL, 0));
    printf("strtol: %ld\n", strtol("1111", NULL, 2));
    printf("strtol: %ld\n", strtol("-1111", NULL, 2));
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strtol: 1280
strtol: 1280
strtol: 1280
strtol: 511
strtol: 511
strtol: 15
strtol: -15
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.2 strtol 测试结果

strtoul、strtoull 函数

这两个函数使用方法与 strtol()、strtol()一样, 区别在于返回值的类型不同, strtoul()返回值类型是 unsigned long int, strtoull()返回值类型是 unsigned long long int, 函数原型如下所示:

```
#include <stdlib.h>

unsigned long int strtoul(const char *nptr, char **endptr, int base);
unsigned long long int strtoull(const char *nptr, char **endptr, int base);
```

函数参数与 strtol()、strtol()一样, 这里不再重述!

测试

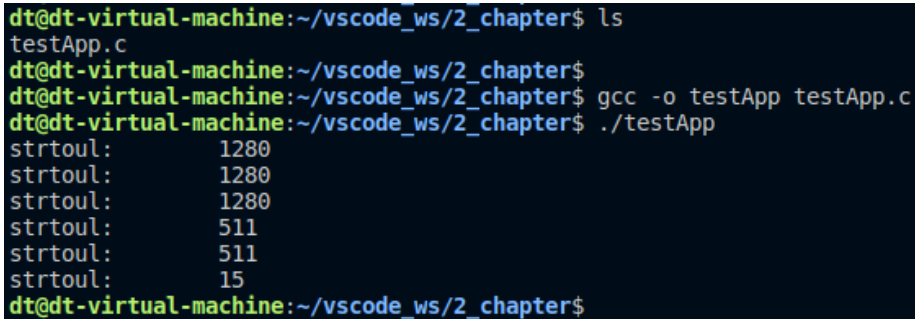
示例代码 6.8.3 strtoul 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

int main(void)
{
    printf("strtoul:    %lu\n", strtoul("0x500", NULL, 16));
    printf("strtoul:    %lu\n", strtoul("0x500", NULL, 0));
    printf("strtoul:    %lu\n", strtoul("500", NULL, 16));
    printf("strtoul:    %lu\n", strtoul("0777", NULL, 8));
    printf("strtoul:    %lu\n", strtoul("0777", NULL, 0));
    printf("strtoul:    %lu\n", strtoul("1111", NULL, 2));
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strtoul:    1280
strtoul:    1280
strtoul:    1280
strtoul:    511
strtoul:    511
strtoul:    15
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.3 strtoul 函数测试结果

6.8.2 字符串转浮点型数据

C 函数库中用于字符串转浮点型数据的函数有 `atof()`、`strtod()`、`strtof()`、`strtold()`。

atof 函数

`atof()` 用于将字符串转换为一个 `double` 类型的浮点数据, 函数原型如下所示:

```
#include <stdlib.h>
```

```
double atof(const char *nptr);
```

使用该函数需要包含头文件 `<stdlib.h>`。

函数参数和返回值含义如下:

nptr: 需要进行转换的字符串。

返回值: 返回转换得到的 `double` 类型数据。

测试

示例代码 6.8.4 `atof` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
{
```

```
printf("atof:   %lf\n", atof("0.123"));
printf("atof:   %lf\n", atof("-1.1185"));
printf("atof:   %lf\n", atof("100.0123"));
exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
atof:   0.123000
atof:  -1.118500
atof:  100.012300
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.4 atof 函数测试结果

strtod、strtof、strtold 函数

strtof()、strtod()以及 strtold()三个库函数可分别将字符串转换为 float 类型数据、double 类型数据、long double 类型数据, 函数原型如下所示:

```
#include <stdlib.h>
```

```
double strtod(const char *nptr, char **endptr);
```

```
float strtof(const char *nptr, char **endptr);
```

```
long double strtold(const char *nptr, char **endptr);
```

使用这些函数需要包含头文件<stdlib.h>。

函数参数与 strtol()含义相同, 但是少了 base 参数。

测试

示例代码 6.8.5 strtof、strtod、strtold 使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main(void)
```

```
{
    printf("strtof:   %f\n", strtof("0.123", NULL));
    printf("strtod:   %lf\n", strtod("-1.1185", NULL));
    printf("strtold:  %Lf\n", strtold("100.0123", NULL));
    exit(0);
}
```

测试结果:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strtod:      0.123000
strtof:      -1.118500
strtold:     100.012300
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.5 字符串转浮点型数据

6.8.3 数字转字符串

数字转换为字符串推荐大家使用前面介绍的格式化 IO 相关库函数, 譬如使用 `printf()` 将数字转字符串、并将其输出到标准输出设备或者使用 `sprintf()` 或 `snprintf()` 将数字转换为字符串并存储在缓冲区中, 具体的使用方法, 3.11 内容中已经给大家进行了详细介绍, 这里不再重述。

示例代码 6.8.6 `sprintf` 数字转字符串

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char str[20] = {0};

    sprintf(str, "%d", 500);
    puts(str);

    memset(str, 0x0, sizeof(str));
    sprintf(str, "%f", 500.111);
    puts(str);

    memset(str, 0x0, sizeof(str));
    sprintf(str, "%u", 500);
    puts(str);

    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
500
500.111000
500
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.6 测试结果

6.9 给应用程序传参

一个能够接受外部传参的应用程序往往使用上会比较灵活, 根据传入不同的参数实现不同的功能, 前面给大家编写的示例代码中, 信息都是硬编码在代码中的, 譬如 `open` 打开的文件路径是固定的, 意味着如果需要打开另一个文件则需要修改代码、修改文件路径, 然后再重新编译、运行, 非常麻烦、不够灵活。其实可以将这些可变的信息通过参数形式传递给应用程序, 譬如, 当执行应用程序的时候, 把需要打开的文件路径作为参数传递给应用程序, 就可以在不重新编译源码的情况下, 通过传递不同的参数打开不同的文件。当然这里只是举个例子, 不同应用程序需根据其需要来设计。

在第一章内容中便给大家介绍了 `main` 函数的两种常用写法, 如果在执行应用程序时, 需要向应用程序传递参数, 则写法如下:

```
int main(int argc, char **argv)
{
    /* 代码 */
}
```

或者写成如下形式:

```
int main(int argc, char *argv[])
{
    /* 代码 */
}
```

传递进来的参数以字符串的形式存在, 字符串的起始地址存储在 `argv` 数组中, 参数 `argc` 表示传递进来的参数个数, 包括应用程序自身路径名, 多个不同的参数之间使用空格分隔开来, 如果参数本身带有空格、则可以使用双引号 " 或者单引号 ' 的形式来表示。

测试

获取执行应用程序时, 向应用程序传递的参数。

示例代码 6.9.1 打印传递给应用程序的参数

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i = 0;

    printf("Number of parameters: %d\n", argc);
    for (i = 0; i < argc; i++)
        printf(" %s\n", argv[i]);

    exit(0);
}
```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 0 1 2
Number of parameters: 4
./testApp
0
1
2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp a b c
Number of parameters: 4
./testApp
a
b
c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.9.1 给应用程序传参

6.10 正则表达式

上面给大家介绍了 C 语言函数库中提供的用于处理字符串相关的一些函数, 这些库函数能够满足基本常见的字符串处理需求, 其库函数内部实现并不复杂, 无非使用到了 for 循环进行处理, 大家可以尝试自己去实现这些函数的功能。

本小节给大家介绍一个新的内容---正则表达式, 在许多的应用程序当中, 通常会有这样的需要: 给定一个字符串, 检查该字符串是否符合某种条件或规则、或者从给定的字符串中找出符合某种条件或规则的子字符串, 将匹配到的字符串提取出来。这种需要在很多的应用程序当中是存在的, 例如, 很多应用程序都有这种校验功能, 譬如检验用户输入的账号或密码是否符合它们定义的规则, 如果不符合规则通常会提示用户按照正确的规则输入用户名或密码。

譬如给定一个字符串, 在程序当中判断该字符串是否是一个 IP 地址, 对于实现这个功能, 大家可能首先想到的是, 使用万能的 for 循环, 当然, 笔者首先肯定的是, 使用 for 循环自然是可以解决这个问题, 但是在程序代码处理上会比较麻烦, 有兴趣的朋友可以自己试一下。

对于这些需求, 其实只需要通过一个正则表达式就可以搞定了, 下一小节开始将向大家介绍正则表达式。

6.10.1 初识正则表达式

正则表达式, 又称为规则表达式 (英语: Regular Expression), 正则表达式通常被用来检索、替换那些符合某个模式 (规则) 的字符串, 正则表达式描述了一种字符串的匹配模式 (pattern), 可以用来检查一个给定的字符串中是否含有某种子字符串、将匹配的字符串替换或者从某个字符串中取出符合某个条件的子字符串。

在 Linux 系统下运行命令的时候, 相信大家都使用过?或*通配符来查找硬盘上的文件或者文本中的某个字符串, ?通配符匹配 0 个或 1 个字符, 而*通配符匹配 0 个或多个字符, 譬如"data?.txt"这样的匹配模式可以将下列文件查找出来:

```
data.dat
data1.dat
data2.dat
datax.dat
dataN.dat
```

尽管使用通配符的方法很有用, 但它还是很有限, 正则表达式则更加强大、更加灵活。

正则表达式其实也是一个字符串, 该字符串由普通字符 (譬如, 数字 0~9、大小写字母以及其它字符) 和特殊字符 (称为“元字符”) 所组成, 由这些字符组成一个“规则字符串”, 这个“规则字符串”用来表达对给定字符串的一种查找、匹配逻辑。

许多程序设计语言都支持正则表达式。譬如, 在 Perl 中就内建了一个功能强大的正则表达式引擎、Python 提供了内置模块 `re` 用于处理正则表达式, 正则表达式这个概念最初是由 Unix 中的工具软件 (例如 `sed` 和 `grep`) 普及开的, 使用过 `sed` 命令的朋友想必对正则表达式并不陌生。同样, 在 C 语言函数库中也提供了用于处理正则表达式的接口供程序员使用。

6.11 C 语言中使用正则表达式

编译正则表达式

匹配正则表达式

释放正则表达式

匹配 URL 的正则表达式:

```
^((ht|f)tps?:)/[-A-Za-z0-9_]+(\.[-A-Za-z0-9_]+)([-A-Za-z0-9_.,@?^=%&:/~+#]*[-A-Za-z0-9_@?^=%&/~+#])?$
```

示例代码 6.11.1 C 应用程序中使用正则表达式

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <regex.h>
#include <string.h>

int main(int argc, char *argv[])
{
    regmatch_t pmatch = {0};
    regex_t reg;
    char errbuf[64];
    int ret;
    char *sptr;
    int length;
    int nmatch; //最多匹配出的结果

    if (4 != argc) {
        /******
        * 执行程序时需要传入两个参数:
        * arg1: 正则表达式
        * arg2: 待测试的字符串
        * arg3: 最多匹配出多少个结果
        *****/
        fprintf(stderr, "usage: %s <regex> <string> <nmatch>\n", argv[0]);
        exit(0);
    }
}
```

```

/* 编译正则表达式 */
if(ret = regcomp(&reg, argv[1], REG_EXTENDED)) {
    regerror(ret, &reg, errbuf, sizeof(errbuf));
    fprintf(stderr, "regcomp error: %s\n", errbuf);
    exit(0);
}

/* 赋值操作 */
sptr = argv[2];          //待测试的字符串
length = strlen(argv[2]); //获取字符串长度
nmatch = atoi(argv[3]); //获取最大匹配数

/* 匹配正则表达式 */
for (int j = 0; j < nmatch; j++) {

    char temp_str[100];

    /* 调用 regexec 匹配正则表达式 */
    if(ret = regexec(&reg, sptr, 1, &pmatch, 0)) {
        regerror(ret, &reg, errbuf, sizeof(errbuf));
        fprintf(stderr, "regexec error: %s\n", errbuf);
        goto out;
    }

    if(-1 != pmatch.rm_so) {
        if (pmatch.rm_so == pmatch.rm_eo) { //空字符串
            sptr += 1;
            length -= 1;
            printf("\n"); //打印出空字符串

            if (0 >= length) //如果已经移动到字符串末尾、则退出
                break;
            continue; //从 for 循环开始执行
        }

        memset(temp_str, 0x00, sizeof(temp_str)); //清零缓冲区
        memcpy(temp_str, sptr + pmatch.rm_so,
                pmatch.rm_eo - pmatch.rm_so); //将匹配出来的子字符串拷贝到缓冲区
        printf("%s\n", temp_str); //打印字符串

        sptr += pmatch.rm_eo;
        length -= pmatch.rm_eo;
        if (0 >= length)

```

```
        break;
    }
}

/* 释放正则表达式 */
out:
regfree(&reg);
exit(0);
}
```

第七章 系统信息与系统资源

在应用程序当中,有时往往需要去获取到一些系统相关的信息,譬如时间、日期、以及其它一些系统相关信息,本章将向大家介绍如何通过 Linux 系统调用或 C 库函数获取系统信息,譬如获取系统时间、日期以及设置系统时间、日期等;除此之外,还会向大家介绍 Linux 系统下的/proc 虚拟文件系统,包括/proc 文件系统是什么以及如何从/proc 文件系统中读取系统、进程有关信息。

除了介绍系统信息内容外,本章还会向大家介绍有关系统资源的使用,譬如系统内存资源的申请与使用等。好了,废话不多少,开始本章内容的学习吧!

- 用于获取系统相关信息的函数;
- 时间、日期;
- 进程时间;
- 使程序进入休眠;
- 在堆中申请内存;
- proc 文件系统介绍;
- 定时器。

7.1 系统信息

7.1.1 系统标识 uname

系统调用 `uname()` 用于获取有关当前操作系统内核的名称和信息, 函数原型如下所示 (可通过 "man 2 uname" 命令查看):

```
#include <sys/utsname.h>
```

```
int uname(struct utsname *buf);
```

使用该函数需要包含头文件 `<sys/utsname.h>`。

函数参数和返回值含义如下:

buf: `struct utsname` 结构体类型指针, 指向一个 `struct utsname` 结构体类型对象。

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

`uname()` 函数用法非常简单, 先定义一个 `struct utsname` 结构体变量, 调用 `uname()` 函数时传入变量的地址即可, `struct utsname` 结构体如下所示:

示例代码 7.1.1 `struct utsname` 结构体

```
struct utsname {
    char sysname[];          /* 当前操作系统的名称 */
    char nodename[];        /* 网络上的名称 (主机名) */
    char release[];         /* 操作系统内核版本 */
    char version[];         /* 操作系统发行版本 */
    char machine[];         /* 硬件架构类型 */
#ifdef _GNU_SOURCE
    char domainname[]; /* 当前域名 */
#endif
};
```

可以看到, `struct utsname` 结构体中的所有成员变量都是字符数组, 所以获取到的信息都是字符串。

测试

编写一个简单地程序, 获取并打印出当前操作系统名称、主机名、内核版本、操作系统发行版本以及处理器硬件架构类型等信息, 测试代码如下:

示例代码 7.1.2 `uname` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/utsname.h>

int main(void)
{
    struct utsname os_info;
    int ret;

    /* 获取信息 */
    ret = uname(&os_info);
    if (-1 == ret) {
```



```

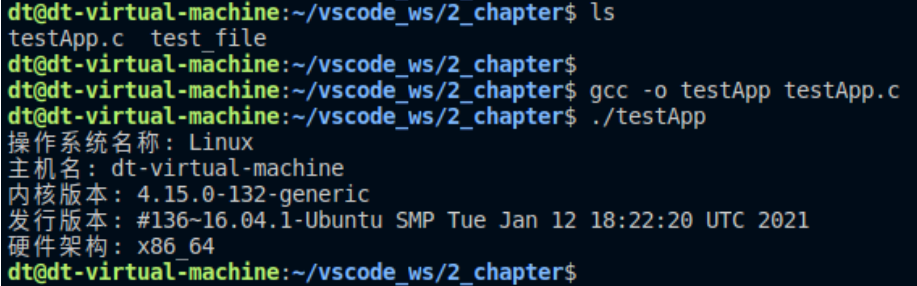
    perror("uname error");
    exit(-1);
}

/* 打印信息 */
printf("操作系统名称: %s\n", os_info.sysname);
printf("主机名: %s\n", os_info.nodename);
printf("内核版本: %s\n", os_info.release);
printf("发行版本: %s\n", os_info.version);
printf("硬件架构: %s\n", os_info.machine);

exit(0);
}

```

运行结果:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
操作系统名称: Linux
主机名: dt-virtual-machine
内核版本: 4.15.0-132-generic
发行版本: #136~16.04.1-Ubuntu SMP Tue Jan 12 18:22:20 UTC 2021
硬件架构: x86_64
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.1.1 运行结果

7.1.2 sysinfo 函数

sysinfo 系统调用可用于获取一些系统统计信息, 其函数原型如下所示:

```
#include <sys/sysinfo.h>
```

```
int sysinfo(struct sysinfo *info);
```

函数参数和返回值含义如下:

info: struct sysinfo 结构体类型指针, 指向一个 struct sysinfo 结构体类型对象。

返回值: 成功返回 0; 失败将返回-1, 并设置 errno。

同样 sysinfo()函数用法也非常简单, 先定义一个 struct sysinfo 结构体变量, 调用 sysinfo()函数时传入变量的地址即可, struct sysinfo 结构体如下所示:

示例代码 7.1.3 struct sysinfo 结构体

```

struct sysinfo {
    long uptime;           /* 自系统启动之后所经过的时间（以秒为单位） */
    unsigned long loads[3]; /* 1, 5, and 15 minute load averages */
    unsigned long totalram; /* 总的可用内存大小 */
    unsigned long freeram; /* 还未被使用的内存大小 */
    unsigned long sharedram; /* Amount of shared memory */
    unsigned long bufferram; /* Memory used by buffers */
    unsigned long totalswap; /* Total swap space size */
    unsigned long freeswap; /* swap space still available */
};

```

```
unsigned short procs;      /* 系统当前进程数量 */
unsigned long totalhigh;   /* Total high memory size */
unsigned long freehigh;   /* Available high memory size */
unsigned int mem_unit;    /* 内存单元大小 (以字节为单位) */
char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding to 64 bytes */
};
```

测试

示例代码 7.1.4 sysinfo 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysinfo.h>

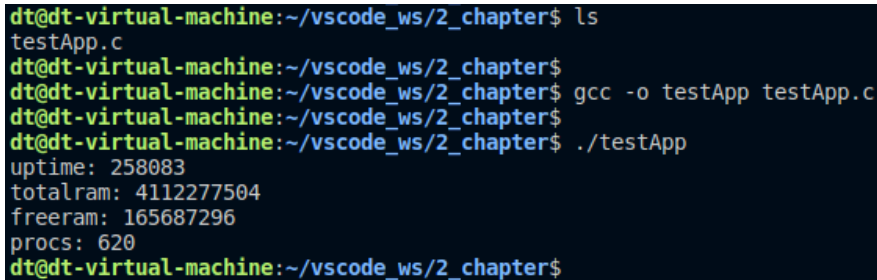
int main(void)
{
    struct sysinfo sys_info;
    int ret;

    /* 获取信息 */
    ret = sysinfo(&sys_info);
    if (-1 == ret) {
        perror("sysinfo error");
        exit(-1);
    }

    /* 打印信息 */
    printf("uptime: %ld\n", sys_info.uptime);
    printf("totalram: %lu\n", sys_info.totalram);
    printf("freeram: %lu\n", sys_info.freeram);
    printf("procs: %u\n", sys_info.procs);

    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
uptime: 258083
totalram: 4112277504
freeram: 165687296
procs: 620
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.2 sysinfo 测试结果

7.1.3 gethostname 函数

此函数可用于单独获取 Linux 系统主机名, 与 struct utsname 数据结构体中的 nodename 变量一样, gethostname 函数原型如下所示 (可通过 "man 2 gethostname" 命令查看):

```
#include <unistd.h>
```

```
int gethostname(char *name, size_t len);
```

使用此函数需要包含头文件 <unistd.h>。

函数参数和返回值含义如下:

name: 指向用于存放主机名字符串的缓冲区。

len: 缓冲区长度。

返回值: 成功返回 0; 失败将返回 -1, 并会设置 errno。

测试

使用 gethostname 函数获取系统主机名:

示例代码 7.1.5 gethostname 函数使用示例

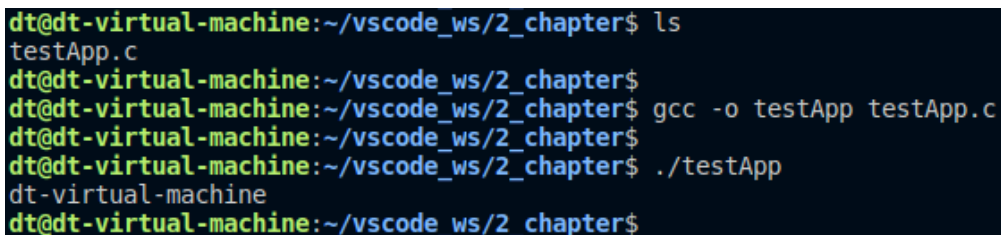
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char hostname[20];
    int ret;

    memset(hostname, 0x0, sizeof(hostname));
    ret = gethostname(hostname, sizeof(hostname));
    if (-1 == ret) {
        perror("gethostname error");
        exit(ret);
    }

    puts(hostname);
    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt-virtual-machine
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.3 获取主机名

7.1.4 sysconf()函数

sysconf()函数是一个库函数,可在运行时获取系统的一些配置信息,譬如页大小(page size)、主机名的最大长度、进程可以打开的最大文件数、每个用户ID的最大并发进程数等。其函数原型如下所示:

```
#include <unistd.h>
```

```
long sysconf(int name);
```

使用该函数需要包含头文件<unistd.h>。

调用 sysconf()函数获取系统的配置信息,参数 name 指定了要获取哪个配置信息,参数 name 可取以下任何一个值(都是宏定义,可通过 man 手册查询):

- **_SC_ARG_MAX:** exec 族函数的参数的最大长度,exec 族函数后面会介绍,这里先不管!
- **_SC_CHILD_MAX:** 每个用户的最大并发进程数,也就是同一个用户可以同时运行的最大进程数。
- **_SC_HOST_NAME_MAX:** 主机名的最大长度。
- **_SC_LOGIN_NAME_MAX:** 登录名的最大长度。
- **_SC_CLK_TCK:** 每秒时钟滴答数,也就是系统节拍率。
- **_SC_OPEN_MAX:** 一个进程可以打开的最大文件数。
- **_SC_PAGESIZE:** 系统页大小(page size)。
- **_SC_TTY_NAME_MAX:** 终端设备名称的最大长度。
-

除以上之外,还有很多,这里就不再一一列举了,可以通过 man 手册进行查看,用的比较多的是 _SC_PAGESIZE 和 _SC_CLK_TCK,在后面章节示例代码中有使用到。

若指定的参数 name 为无效值,则 sysconf()函数返回-1,并将 errno 设置为 EINVAL。否则返回的值便是对应的配置值。注意,返回值是一个 long 类型的数据。

使用示例

获取每个用户的最大并发进程数、系统节拍率和系统页大小。

示例代码 7.1.6 sysconf()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("每个用户的最大并发进程数: %ld\n", sysconf(_SC_CHILD_MAX));
    printf("系统节拍率: %ld\n", sysconf(_SC_CLK_TCK));
    printf("系统页大小: %ld\n", sysconf(_SC_PAGESIZE));
    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
每个用户的最大并发进程数: 15446
系统节拍率: 100
系统页大小: 4096
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.4 测试结果

7.2 时间、日期

本小节向大家介绍下时间、日期相关的系统调用或 C 库函数以及它们的使用方法。

7.2.1 时间的概念

在正式介绍这些时间、日期相关的系统调用或 C 库函数之前, 需要向大家介绍一些时间相关的基本概念, 譬如 GMT 时间、UTC 时间以及时区等。

地球总是自西向东自转, 东边总比西边先看到太阳, 东边的时间也总比西边的早。东边时刻与西边时刻的差值不仅要以時計, 而且还要以分和秒来计算, 这给人们的日常生活和工作都带来许多不便。

GMT 时间

GMT (Greenwich Mean Time) 中文全称是格林威治标准时间, 这个时间系统的概念在 1884 年被确立, 由英国伦敦的格林威治皇家天文台计算并维护, 并在之后的几十年向欧陆其它国家扩散。

由于从 19 实际开始, 因为世界各国往来频繁, 而欧洲大陆、美洲大陆以及亚洲大陆都有各自的时区, 所以为了避免时间混乱, 1884 年, 各国代表在美国华盛顿召开国际大会, 通过协议选出英国伦敦的格林威治作为全球时间的中心点, 决定以通过格林威治的子午线作为划分东西两半球的经线零度线(本初子午线、零度经线), 由此格林威治标准时间因而诞生!

所以 GMT 时间就是英国格林威治当地时间, 也就是零时区(中时区)所在时间, 譬如 GMT 12:00 就是指英国伦敦的格林威治皇家天文台当地的中午 12:00, 与我国的标准时间北京时间(东八区)相差 8 个小时, 即早八个小时, 所以 GMT 12:00 对应的北京时间是 20:00。

UTC 时间

UTC (Coordinated Universal Time) 指的是世界协调时间(又称世界标准时间、世界统一时间), 是经过平均太阳时(以格林威治时间 GMT 为准)、地轴运动修正后的新时标以及以「秒」为单位的国际原子时所综合精算而成的时间, 计算过程相当严谨精密, 因此若以「世界标准时间」的角度来说, UTC 比 GMT 来得更加精准。

GMT 与 UTC 这两者几乎是同一概念, 它们都是指格林威治标准时间, 也就是国际标准时间, 只不过 UTC 时间比 GMT 时间更加精准, 所以在我们的编程当中不用刻意去区分它们之间的区别。

在 Ubuntu 系统下, 可以使用 "date -u" 命令查看到当前的 UTC 时间, 如下所示:

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ date -u
2021年 02月 24日 星期三 02:34:13 UTC
dt@dt-virtual-machine:~$
```

图 7.2.1 查看 UTC 时间

后面显示的 UTC 字样就表示当前查看到的时间是 UTC 时间, 也就是国际标准时间。

时区

全球被划分为 24 个时区, 每一个时区横跨经度 15 度, 以英国格林威治的本初子午线作为零度经线, 将全球划分为东西两半球, 分为东一区、东二区、东三区……东十二区以及西一区、西二区、西三区……西十二区, 而本初子午线所在时区被称为中时区 (或者叫零时区), 划分图如下所示:

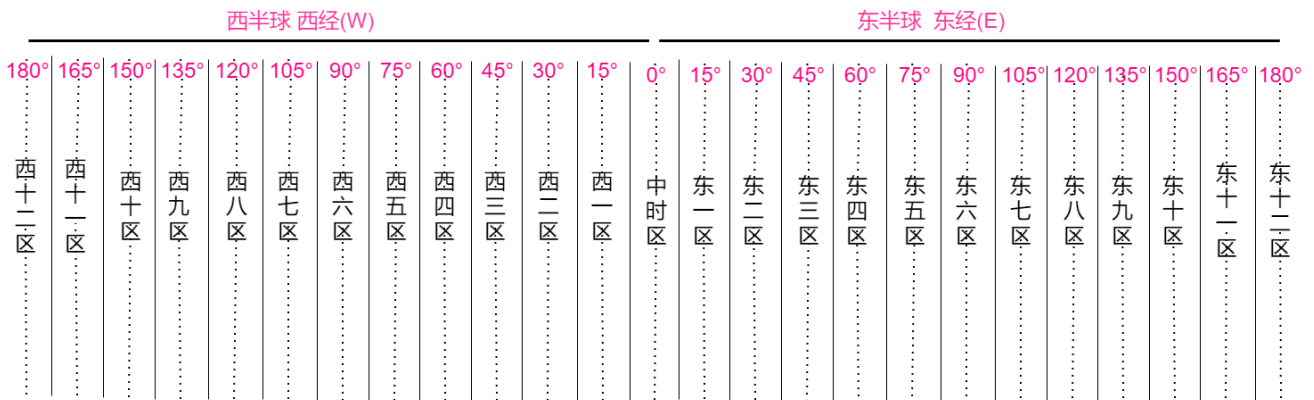


图 7.2.2 全球 24 时区划分

东十二区和西十二区其实是一个时区, 就是十二区, 东十二区与西十二区各横跨经度 7.5 度, 以 180 度经线作为分界线。每个时区的中央经线上的时间就是这个时区内统一采用的时间, 称为区时。相邻两个时区的时间相差 1 小时。例如, 我国东 8 区的时间总比泰国东 7 区的时间早 1 小时, 而比日本东 9 区的时间晚 1 小时。因此, 出国旅行的人, 必须随时调整自己的手表, 才能和当地时间相一致。凡向西走, 每过一个时区, 就要把表向前拨 1 小时 (比如 2 点拨到 1 点); 凡向东走, 每过一个时区, 就要把表向后拨 1 小时 (比如 1 点拨到 2 点)。

实际上, 世界上不少国家和地区都不严格按时区来计算时间。为了在全国范围内采用统一的时间, 一般都把某一个时区的时间作为全国统一采用的时间。例如, 我国把首都北京所在的东 8 区的时间作为全国统一的时间, 称为北京时间, 北京时间就作为我国使用的本地时间, 譬如我们电脑上显示的时间就是北京时间, 我国国土面积广大, 由东到西横跨了 5 个时区, 也就意味着我国最东边的地区与最西边的地区实际上相差了 4、5 个小时。又例如, 英国、法国、荷兰和比利时等国, 虽地处中时区, 但为了和欧洲大多数国家时间相一致, 则采用东 1 区的时间。

譬如在 Ubuntu 系统下, 可以使用 `date` 命令查看系统当前的本地时间, 如下所示:

```
dt@dt-virtual-machine:~$ date
2021年 02月 23日 星期二 16:35:45 CST
dt@dt-virtual-machine:~$
```

图 7.2.3 date 查看本地时间

可以看到显示出来的字符串后面有一个 "CST" 字样, CST 在这里其实指的是 China Standard Time (中国标准时间) 的缩写, 表示当前查看到的时间是中国标准时间, 也就是我国所使用的标准时间--北京时间, 一般在安装 Ubuntu 系统的时候会提示用户设置所在城市, 那么系统便会根据你所设置的城市来确定系统的本地时间对应的时区, 譬如设置的城市为上海, 那么系统的本地时间就是北京时间, 因为我国统一使用北京时间作为本国的标准时间。

在 Ubuntu 系统下, 时区信息通常以标准格式保存在一些文件当中, 这些文件通常位于 `/usr/share/zoneinfo` 目录下, 该目录下的每一个文件 (包括子目录下的文件) 都包含了一个特定国家或地区内时区制度的相关信息, 且往往根据其所描述的城市或地区缩写来加以命名, 譬如 EST (美国东部标准时间)、CET (欧洲中部时间)、UTC (世界标准时间)、Hongkong、Iran、Japan (日本标准时间) 等, 也把这些文件称为时区配置文件, 如下图所示:

```
dt@dt-virtual-machine:/usr/share/zoneinfo$ pwd
/usr/share/zoneinfo
dt@dt-virtual-machine:/usr/share/zoneinfo$ ls
Africa      Cuba      GMT0      Japan      Pacific    Turkey
America     EET      GMT-0     Kwajalein  Poland     UCT
Antarctica  Egypt    GMT+0     leap-seconds.list  Portugal   Universal
Arctic      Eire     Greenwich Libya      posix      US
Asia        EST      Hongkong localtime  posixrules UTC
Atlantic    EST5EDT HST       MET        PRC        WET
Australia   Etc      Iceland   Mexico     PST8PDT    W-SU
Brazil      Europe   Indian    MST        right       zone1970.tab
Canada      Factory  Iran      MST7MDT    ROC        zone.tab
CET         GB       iso3166.tab  Navajo     ROK        Zulu
Chile       GB-Eire  Israel    NZ         Singapore
CST6CDT    GMT      Jamaica   NZ-CHAT    SystemV
```

图 7.2.4 时区信息配置文件

系统的本地时间由时区配置文件/etc/localtime 定义，通常链接到/usr/share/zoneinfo 目录下的某一个文件（或其子目录下的某一个文件）：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls -l /etc/localtime
lrwxrwxrwx 1 root root 33 2月 23 17:14 /etc/localtime -> /usr/share/zoneinfo/Asia/Shanghai
dt@dt-virtual-machine:~$
```

图 7.2.5 /etc/localtime 配置文件

如果我们要修改 Ubuntu 系统本地时间的时区信息，可以直接将/etc/localtime 链接到/usr/share/zoneinfo 目录下的任意一个时区配置文件，譬如 EST（美国东部标准时间），首先进入到/etc 目录下，执行下面的命令：

```
sudo rm -rf localtime          #删除原有链接文件
sudo ln -s /usr/share/zoneinfo/EST localtime  #重新建立链接文件
```

```
dt@dt-virtual-machine:/etc$
dt@dt-virtual-machine:/etc$ sudo rm -rf localtime
dt@dt-virtual-machine:/etc$
dt@dt-virtual-machine:/etc$ sudo ln -s /usr/share/zoneinfo/EST localtime
dt@dt-virtual-machine:/etc$
```

图 7.2.6 修改本地时间对应的时区配置文件

接下来再使用 date 命令查看下系统当前的时间，如下所示：

```
dt@dt-virtual-machine:~$ date
2021年 02月 23日 星期二 04:33:39 EST
dt@dt-virtual-machine:~$
```

图 7.2.7 date 查看时间

可以发现后面的标识变成了 EST，也就意味着当前系统的本地时间变成了 EST 时间（美国东部标准时间）。

关于时区的信息就给大家介绍这么多了。

世界标准时间的意义

世界标准时间指的就是格林威治时间，也就是中时区对应的的时间，用格林威治当地时间作为全球统一时间，用以描述全球性的事件，方便大家记忆、以免混淆。

本小节关于时间的概念就给大家介绍这么多，其中涉及到的很多内容都是初中地理或高中地理中的知识，譬如本初子午线、经线、时区等，相信大家都学过，这里笔者便不再啰嗦！

7.2.2 Linux 系统中的时间

点时间和段时间

通常描述时间有两种方式: 点时间和段时间; 点时间顾名思义指的是某一个时间点, 譬如当前时间是 2021 年 2 月 22 日星期一 11:12 分 35 秒, 所以这里指的就是某一个时间点; 而对于段时间来说, 顾名思义指的是某一个时间段, 譬如早上 8:00 到中午 12:00 这段时间。

实时时钟 RTC

操作系统中一般会有两个时钟, 一个系统时钟 (system clock), 一个实时时钟 (Real time clock), 也叫 RTC; 系统时钟由系统启动之后由内核来维护, 譬如使用 date 命令查看到的就是系统时钟, 所以在系统关机情况下是不存在的; 而实时时钟一般由 RTC 时钟芯片提供, RTC 芯片有相应的电池为其供电, 以保证系统在关机情况下 RTC 能够继续工作、继续计时。

Linux 系统如何记录时间

Linux 系统在开机启动之后首先会读取 RTC 硬件获取实时时钟作为系统时钟的初始值, 之后内核便开始维护自己的系统时钟。所以由此可知, RTC 硬件只有在系统开机启动时会读取一次, 目的是用于对系统时钟进行初始化操作, 之后的运行过程中便不会再对其进行读取操作了。

而在系统关机时, 内核会将系统时钟写入到 RTC 硬件、已进行同步操作。

jiffies 的引入

jiffies 是内核中定义的一个全局变量, 内核使用 jiffies 来记录系统从启动以来的系统节拍数, 所以这个变量用来记录以系统节拍时间为单位的时间长度, Linux 内核在编译配置时定义了一个节拍时间, 使用节拍率 (一秒钟多少个节拍数) 来表示, 譬如常用的节拍率为 100Hz (一秒钟 100 个节拍数, 节拍时间为 1s / 100)、200Hz (一秒钟 200 个节拍, 节拍时间为 1s / 200)、250Hz (一秒钟 250 个节拍, 节拍时间为 1s / 250)、300Hz (一秒钟 300 个节拍, 节拍时间为 1s / 300)、500Hz (一秒钟 500 个节拍, 节拍时间为 1s / 500) 等。由此可以发现配置的节拍率越低, 每一个系统节拍的时间就越短, 也就意味着 jiffies 记录的时间精度越高, 当然, 高节拍率会导致系统中断的产生更加频繁, 频繁的中断会加剧系统的负担, 一般默认情况下都是采用 100Hz 作为系统节拍率。

内核其实通过 jiffies 来维护系统时钟, 全局变量 jiffies 在系统开机启动时会设置一个初始值, 上面也给大家提到过, RTC 实时时钟会在系统开机启动时读取一次, 目的是用于对系统时钟进行初始化, 这里说的初始化其实指的就是对内核的 jiffies 变量进行初始化操作, 具体如何将读取到的实时时钟换算成 jiffies 数值, 这里便不再给大家介绍了。

所以由此可知, 操作系统使用 jiffies 这个全局变量来记录当前时间, 当我们需要获取到系统当前时间点时, 就可以使用 jiffies 变量去计算, 当然并不需要我们手动去计算, Linux 系统提供了相应的系统调用或 C 库函数用于获取当前时间, 譬如系统调用 time()、gettimeofday(), 其实质上就是通过 jiffies 变量换算得到。

7.2.3 获取时间 time/gettimeofday

(1)time 函数

系统调用 time()用于获取当前时间, 以秒为单位, 返回得到的值是自 1970-01-01 00:00:00 +0000 (UTC) 以来的秒数, 其函数原型如下所示 (可通过"man 2 time"命令查看):

```
#include <time.h>
```

```
time_t time(time_t *tloc);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下:

tloc: 如果 tloc 参数不是 NULL, 则返回值也存储在 tloc 指向的内存中。

返回值: 成功则返回自 1970-01-01 00:00:00 +0000 (UTC) 以来的时间值 (以秒为单位); 失败则返回 -1, 并会设置 `errno`。

所以由此可知, `time` 函数获取到的的是一个时间段, 也就是从 1970-01-01 00:00:00 +0000 (UTC) 到现在这段时间所经过的秒数, 所以你要计算现在这个时间点, 只需要使用 `time()` 得到的秒数加 1970-01-01 00:00:00 即可! 当然, 这并不需要我们手动去计算, 可以直接使用相关系统调用或 C 库函数来得到当前时间, 后面再给大家介绍。

自 1970-01-01 00:00:00 +0000 (UTC) 以来经过的总秒数, 我们把这个称之为日历时间或 `time_t` 时间。

测试

使用系统调用 `time()` 获取自 1970-01-01 00:00:00 +0000 (UTC) 以来的时间值:

示例代码 7.2.1 `time` 函数使用示例

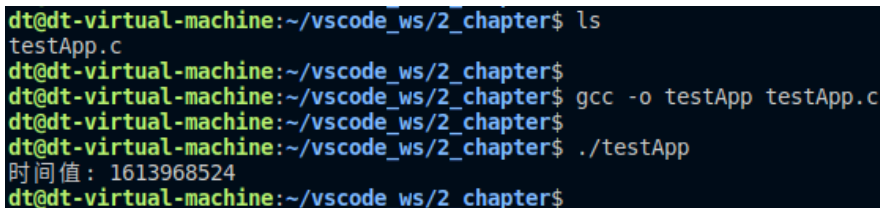
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    time_t t;

    t = time(NULL);
    if (-1 == t) {
        perror("time error");
        exit(-1);
    }

    printf("时间值: %ld\n", t);
    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
时间值: 1613968524
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.8 `time` 函数测试结果

(2) `gettimeofday` 函数

`time()` 获取到的时间只能精确到秒, 如果想要获取更加精确的时间可以使用系统调用 `gettimeofday` 来实现, `gettimeofday()` 函数提供微秒级时间精度, 函数原型如下所示 (可通过 "man 2 `gettimeofday`" 命令查看):

```
#include <sys/time.h>
```

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

使用该函数需要包含头文件 `<sys/time.h>`。

函数参数和返回值含义如下:

tv: 参数 tv 是一个 struct timeval 结构体指针变量, struct timeval 结构体在前面章节内容中已经给大家介绍过, 具体参考示例代码 5.6.3。

tz: 参数 tz 是个历史产物, 早期实现用其来获取系统的时区信息, 目前已遭废弃, 在调用 gettimeofday() 函数时应将参数 tz 设置为 NULL。

返回值: 成功返回 0; 失败将返回-1, 并设置 errno。

获取到的时间值存储在参数 tv 所指向的 struct timeval 结构体变量中, 该结构体包含了两个成员变量 tv_sec 和 tv_usec, 分别用于表示秒和微秒, 所以获取到的时间值就是 tv_sec (秒) + tv_usec (微秒), 同样获取到的秒数与 time() 函数一样, 也是自 1970-01-01 00:00:00 +0000 (UTC) 到现在这段时间所经过的秒数, 也就是日历时间, 所以由此可知 time() 返回得到的值和函数 gettimeofday() 所返回的 tv 参数中 tv_sec 字段的数值相同。

测试

使用 gettimeofday 获取自 1970-01-01 00:00:00 +0000 (UTC) 以来的时间值:

示例代码 7.2.2 gettimeofday 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
    struct timeval tval;
    int ret;

    ret = gettimeofday(&tval, NULL);
    if (-1 == ret) {
        perror("gettimeofday error");
        exit(-1);
    }

    printf("时间值: %ld 秒+%ld 微秒\n", tval.tv_sec, tval.tv_usec);
    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
时间值: 1613976912秒+99626微秒
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.9 gettimeofday 函数测试结果

7.2.4 时间转换函数

通过 time() 或 gettimeofday() 函数可以获取到当前时间点相对于 1970-01-01 00:00:00 +0000 (UTC) 这个时间点所经过时间 (日历时间), 所以获取到的是一个时间段的长度, 但是这并不利于我们查看当前时间,

这个结果对于我们来说非常不友好,那么本小节将向大家介绍一些系统调用或 C 库函数,通过这些 API 可以将 `time()` 或 `gettimeofday()` 函数获取到的秒数转换为利于查看和理解的形式。

(1) ctime 函数

`ctime()` 是一个 C 库函数,可以将日历时间转换为可打印输出的字符串形式, `ctime()` 函数原型如下所示:

```
#include <time.h>
```

```
char *ctime(const time_t *timep);
```

```
char *ctime_r(const time_t *timep, char *buf);
```

使用该函数需要包含头文件 `<time.h>`。

函数参数和返回值含义如下:

timep: `time_t` 时间变量指针。

返回值: 成功将返回一个 `char *` 类型指针,指向转换后得到的字符串;失败将返回 `NULL`。

所以由此可知,使用 `ctime` 函数非常简单,只需将 `time_t` 时间变量的指针传入即可,调用成功便可返回字符串指针,拿到字符串指针之后,可以使用 `printf` 将其打印输出。但是 `ctime()` 是一个不可重入函数,存在一些安全上面的隐患, `ctime_r()` 是 `ctime()` 的可重入版本,一般推荐大家使用可重入函数 `ctime_r()`,可重入函数 `ctime_r()` 多了一个参数 `buf`,也就是缓冲区首地址,所以 `ctime_r()` 函数需要调用者提供用于存放字符串的缓冲区。

Tips: 关于可重入函数与不可重入函数将会在后面章节内容中进行介绍,这里暂时先不去管这个问题,在 Linux 系统中,有一些系统调用或 C 库函数提供了可重入版本与不可重入版本的函数接口,可重入版本函数所对应的函数名一般都会会有一个 `_r` 后缀来表明它是一个可重入函数。

`ctime()` (或 `ctime_r()`) 转换得到的时间是计算机所在地对应的本地时间(譬如在中国对应的便是北京时间),并不是 UTC 时间,接下来编写一段简单地代码进行测试。

测试

示例代码 7.2.3 time/time_r 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    char tm_str[100] = {0};
    time_t tm;

    /* 获取时间 */
    tm = time(NULL);
    if (-1 == tm) {
        perror("time error");
        exit(-1);
    }

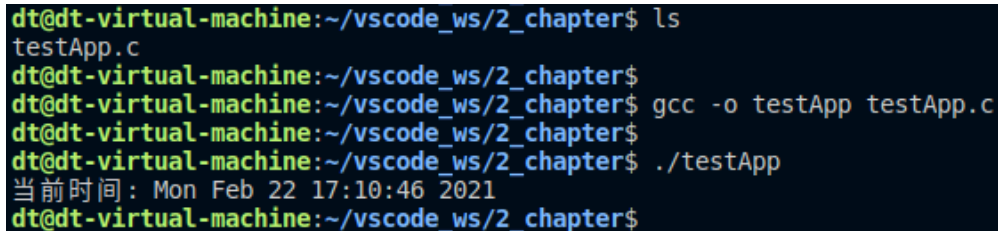
    /* 将时间转换为字符串形式 */
    ctime_r(&tm, tm_str);
```

```

/* 打印输出 */
printf("当前时间: %s", tm_str);
exit(0);
}

```

运行结果:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
当前时间: Mon Feb 22 17:10:46 2021
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.2.10 打印当前时间

从图中可知,打印出来的时间为"Mon Feb 22 17:10:46 2021", Mon 表示星期一,这是一个英文单词的缩写, Feb 表示二月份,这也是一个英文单词的缩写, 22 表示 22 日,所以整个打印信息显示的时间就是 2021 年 2 月 22 日星期一 17 点 10 分 46 秒。

(2) localtime 函数

localtime()函数可以把 time()或 gettimeofday()得到的秒数 (time_t 时间或日历时间) 变成一个 struct tm 结构体所表示的时间, 该时间对应的是本地时间。localtime 函数原型如下:

```
#include <time.h>
```

```
struct tm *localtime(const time_t *timep);
```

```
struct tm *localtime_r(const time_t *timep, struct tm *result);
```

使用该函数需要包含头文件<time.h>, localtime()的可重入版本为 localtime_r()。

函数参数和返回值含义如下:

timep: 需要进行转换的 time_t 时间变量对应的指针, 可通过 time()或 gettimeofday()获取得到。

result: 是一个 struct tm 结构体类型指针, 稍后给大家介绍 struct tm 结构体, 参数 result 是可重入函数 localtime_r()需要额外提供的参数。

返回值: 对于不可重入版本 localtime()来说, 成功则返回一个有效的 struct tm 结构体指针, 而对于可重入版本 localtime_r()来说, 成功执行情况下, 返回值将会等于参数 result; 失败则返回 NULL。

使用不可重入函数 localtime()并不需要调用者提供 struct tm 变量, 而是它会直接返回出来一个 struct tm 结构体指针, 然后直接通过该指针访问里边的成员变量即可! 虽然很方便, 但是存在一些安全隐患, 所以一般不推荐使用不可重入版本。

使用可重入版本 localtime_r()调用者需要自己定义 struct tm 结构体变量、并将该变量指针赋值给参数 result, 在函数内部会对该结构体变量进行赋值操作。

struct tm 结构体如下所示:

示例代码 7.2.4 struct tm 结构体

```

struct tm {
    int tm_sec;      /* 秒(0-60) */
    int tm_min;     /* 分(0-59) */
    int tm_hour;    /* 时(0-23) */
    int tm_mday;    /* 日(1-31) */
    int tm_mon;     /* 月(0-11) */

```

```

int tm_year;      /* 年(这个值表示的是自 1900 年到现在经过的年数) */
int tm_wday;     /* 星期(0-6, 星期日 Sunday = 0、星期一=1...) */
int tm_yday;     /* 一年里的第几天(0-365, 1 Jan = 0) */
int tm_isdst;    /* 夏令时 */
};

```

从 struct tm 结构体内容可知, 该结构体中包含了年月日时分秒星期等信息, 使用 localtime/localtime_r() 便可以将 time_t 时间总秒数分解成了各个独立的时间信息, 易于我们查看和理解。

测试

示例代码 7.2.5 localtime/localtime_r 函数使用示例

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    struct tm t;
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

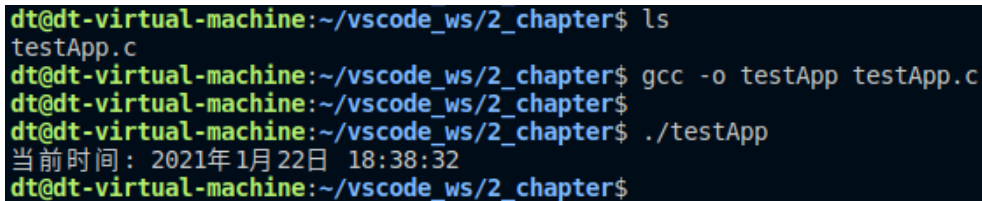
    /* 转换得到本地时间 */
    localtime_r(&sec, &t);

    /* 打印输出 */
    printf("当前时间: %d 年%d 月%d 日 %d:%d:%d\n",
           t.tm_year + 1900, t.tm_mon, t.tm_mday,
           t.tm_hour, t.tm_min, t.tm_sec);

    exit(0);
}

```

运行结果:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
当前时间: 2021年1月22日 18:38:32
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.2.11 localtime/localtime_r 测试结果

(3) gmtime 函数

gmtime()函数也可以把 time_t 时间变成一个 struct tm 结构体所表示的时间,与 localtime()所不同的是, gmtime()函数所得到的是 UTC 国际标准时间,并不是计算机的本地时间,这是它们之间的唯一区别。gmtime()函数原型如下所示:

```
#include <time.h>

struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *timep, struct tm *result);
```

同样使用 gmtime()函数需要包含头文件<time.h>。

gmtime_r()是 gmtime()的可重入版本,同样也是推荐大家使用可重入版本函数 gmtime_r。关于该函数的参数和返回值,这里便不再介绍,与 localtime()是一样的。

测试

使用 localtime 获取本地时间、使用 gmtime 获取 UTC 国际标准时间,并进行对比:

示例代码 7.2.6 gmtime 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    struct tm local_t;
    struct tm utc_t;
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

    /* 转换得到本地时间 */
    localtime_r(&sec, &local_t);

    /* 转换得到国际标准时间 */
    gmtime_r(&sec, &utc_t);

    /* 打印输出 */
    printf("本地时间:  %d年%d月%d日 %d:%d:%d\n",
           local_t.tm_year + 1900, local_t.tm_mon, local_t.tm_mday,
           local_t.tm_hour, local_t.tm_min, local_t.tm_sec);
    printf("UTC 时间:  %d年%d月%d日 %d:%d:%d\n",
           utc_t.tm_year + 1900, utc_t.tm_mon, utc_t.tm_mday,
           utc_t.tm_hour, utc_t.tm_min, utc_t.tm_sec);
}
```

```
exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本地时间:      2021年1月22日 19:30:7
UTC时间:       2021年1月22日 11:30:7
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.12 打印本地时间与 UTC 时间

从打印结果可知,本地时间与 UTC 时间(国际标准时间)相差 8 个小时,因为笔者使用的计算机其对应的本地时间指的便是北京时间,而北京时间要早于国际标准时间 8 个小时(东八区)。

(4) mktime 函数

mktime()函数与 localtime()函数相反, mktime()可以将使用 struct tm 结构体表示的分解时间转换为 time_t 时间(日历时间),同样这也是一个 C 库函数,其函数原型如下所示:

```
#include <time.h>
```

```
time_t mktime(struct tm *tm);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下:

tm: 需要进行转换的 struct tm 结构体变量对应的指针。

返回值: 成功返回转换得到 time_t 时间值;失败返回-1。

测试

示例代码 7.2.7 mktime 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    struct tm local_t;
    time_t sec;

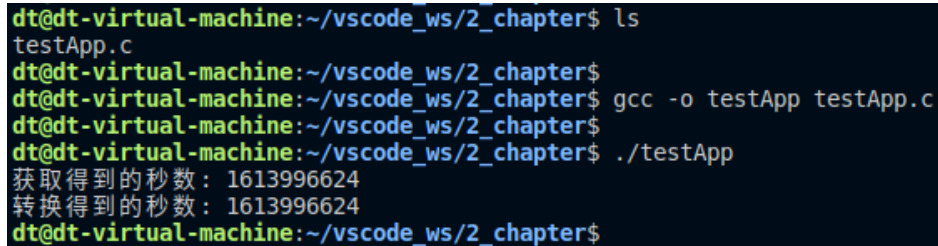
    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

    printf("获取得到的秒数: %ld\n", sec);
    localtime_r(&sec, &local_t);
```

```
printf("转换得到的秒数: %ld\n", mktime(&local_t));

exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
获取到的秒数: 1613996624
转换得到的秒数: 1613996624
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.13 测试结果

(5) asctime 函数

asctime()函数与 ctime()函数的作用一样,也可将时间转换为可打印输出的字符串形式,与 ctime()函数的区别在于,ctime()是将 time_t 时间转换为固定格式字符串、而 asctime()则是将 struct tm 表示的分解时间转换为固定格式的字符串。asctime()函数原型如下所示:

```
#include <time.h>
```

```
char *asctime(const struct tm *tm);
```

```
char *asctime_r(const struct tm *tm, char *buf);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下:

tm: 需要进行转换的 struct tm 表示的时间。

buf: 可重入版本函数 asctime_r 需要额外提供的参数 buf, 指向一个缓冲区, 用于存放转换得到的字符串。

返回值: 转换失败将返回 NULL; 成功将返回一个 char *类型指针, 指向转换后得到的时间字符串, 对于 asctime_r 函数来说, 返回值就等于参数 buf。

测试

示例代码 7.2.8 asctime 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    struct tm local_t;
    char tm_str[100] = {0};
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
```



```

    perror("time error");
    exit(-1);
}

localtime_r(&sec, &local_t);
asctime_r(&local_t, tm_str);
printf("本地时间: %s", tm_str);

exit(0);
}

```

运行结果:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本地时间: Mon Feb 22 20:40:25 2021
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.2.14 测试结果

(6)strftime 函数

除了 asctime()函数之外, 这里再给大家介绍一个 C 库函数 strftime(), 此函数也可以将一个 struct tm 变量表示的分解时间转换为格式化字符串, 并且在功能上比 asctime()和 ctime()更加强大, 它可以根据自己的喜好自定义时间的显示格式, 而 asctime()和 ctime()转换得到的字符串时间格式的固定的。

strftime()函数原型如下所示:

```
#include <time.h>
```

```
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下:

s: 指向一个缓存区的指针, 该缓冲区用于存放生成的字符串。

max: 字符串的最大字节数。

format: 这是一个用字符串表示的字段, 包含了普通字符和特殊格式说明符, 可以是这两种字符的任意组合。特殊格式说明符将会被替换为 struct tm 结构体对象所指时间的相应值, 这些特殊格式说明符如下:

表 7.2.1 strftime 函数特殊格式说明符

说明符	表示含义	实例
%a	星期的缩写	Sun
%A	星期的完整名称	Sunday
%b	月份的缩写	Mar
%B	月份的完整名称	March
%c	系统当前语言环境对应的首选日期和时间表示形式	
%C	世纪 (年/100)	20
%d	十进制数表示一个月中的第几天 (01-31)	15、05
%D	相当于 %m/%d/%y	01/14/21
%e	与 %d 相同, 但是单个数字时, 前导 0 会被去掉	15、5

%F	相当于%Y-%m-%d	2021-01-14
%h	相当于%b	Jan
%H	十进制数表示的 24 小时制的小时 (范围 00-23)	01、22
%I	十进制数表示的 12 小时制的小时 (范围 01-12)	01、11
%j	十进制数表示的一年中的某天 (范围 001-366)	050、285
%k	与%H 相同, 但是单个数字时, 前导 0 会被去掉 (范围 0-23)	1、22
%l	与%I 相同, 但是单个数字时, 前导 0 会被去掉 (范围 1-12)	1、11
%m	十进制数表示的月份 (范围 01-12)	01、10
%M	十进制数表示的分钟 (范围 00-59)	01、55
%n	换行符	
%p	根据给定的时间值, 添加“AM”或“PM”	PM
%P	与%p 相同, 但会使用小写字母表示	pm
%r	相当于%I:%M:%S %p	12:15:31 PM
%R	相当于%H:%M	12:16
%S	十进制数表示的秒数 (范围 00-60)	05、30
%T	相当于%H:%M:%S	12:20:03
%u	十进制数表示的星期 (范围 1-7, 星期一为 1)	1、5
%U	十进制数表示, 当前年份的第几个星期 (范围 00-53), 从第一个星期日作为 01 周的第一天开始	
%W	十进制数表示, 当前年份的第几个星期 (范围 00-53), 从第一个星期一作为第 01 周的第一天开始	
%w	十进制数表示的星期, 范围为 0-6, 星期日为 0	
%x	系统当前语言环境的首选日期表示形式, 没有时间	01/14/21
%X	系统当前语言环境的首选时间表示形式, 没有日期	12:30:16
%y	十进制数表示的年份 (后两字数字)	21
%Y	十进制数表示的年份 (4 个数字)	2021
%%	输出%符号	%

strftime 函数的特殊格式说明符还是比较多的, 不用去记它, 需要用的时候再去查即可!

通过上表可知, 譬如我要想输出"2021-01-14 16:30:25<PM> January Thursday"这样一种形式表示的时间日期, 那么就可以这样来设置 format 参数:

```
"%Y-%m-%d %H:%M:%S<%p> %B %A"
```

tm: 指向 struct tm 结构体对象的指针。

返回值: 如果转换得到的目标字符串不超过最大字节数 (也就是 max), 则返回放置到 s 数组中的字节数; 如果超过了最大字节数, 则返回 0。

测试

示例代码 7.2.9 strftime 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
```

```

struct tm local_t;
char tm_str[100] = {0};
time_t sec;

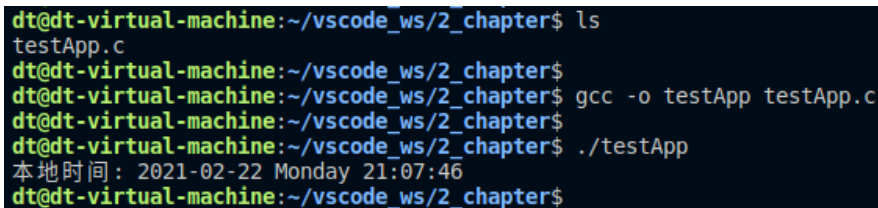
/* 获取时间 */
sec = time(NULL);
if (-1 == sec) {
    perror("time error");
    exit(-1);
}

localtime_r(&sec, &local_t);
strftime(tm_str, sizeof(tm_str), "%Y-%m-%d %A %H:%M:%S", &local_t);
printf("本地时间: %s\n", tm_str);

exit(0);
}

```

运行结果:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本地时间: 2021-02-22 Monday 21:07:46
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.2.15 测试结果

7.2.5 设置时间 settimeofday

使用 `settimeofday()` 函数可以设置时间, 也就是设置系统的本地时间, 函数原型如下所示:

```
#include <sys/time.h>
```

```
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

首先使用该函数需要包含头文件 `<sys/time.h>`。

函数参数和返回值含义如下:

tv: 参数 `tv` 是一个 `struct timeval` 结构体指针变量, `struct timeval` 结构体在前面章节内容中已经给大家介绍了, 需要设置的时间便通过参数 `tv` 指向的 `struct timeval` 结构体变量传递进去。

tz: 参数 `tz` 是个历史产物, 早期实现用其来设置系统的时区信息, 目前已遭废弃, 在调用 `settimeofday()` 函数时应将参数 `tz` 设置为 `NULL`。

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

使用 `settimeofday` 设置系统时间时内核会进行权限检查, 只有超级用户 (`root`) 才可以设置系统时间, 普通用户将无操作权限。

7.2.6 总结

本小节给大家介绍了时间相关的基本概念, 譬如 GMT 时间、UTC 时间以及全球 24 个时区的划分等, 并且给大家介绍了 Linux 系统下常用的时间相关的系统调用和库函数, 主要有 9 个:

time/ctime/localtime/gmtime/mktime/asctime/strftime/gettimeofday/settimeofday, 对这些函数的功能、作用总结如下:

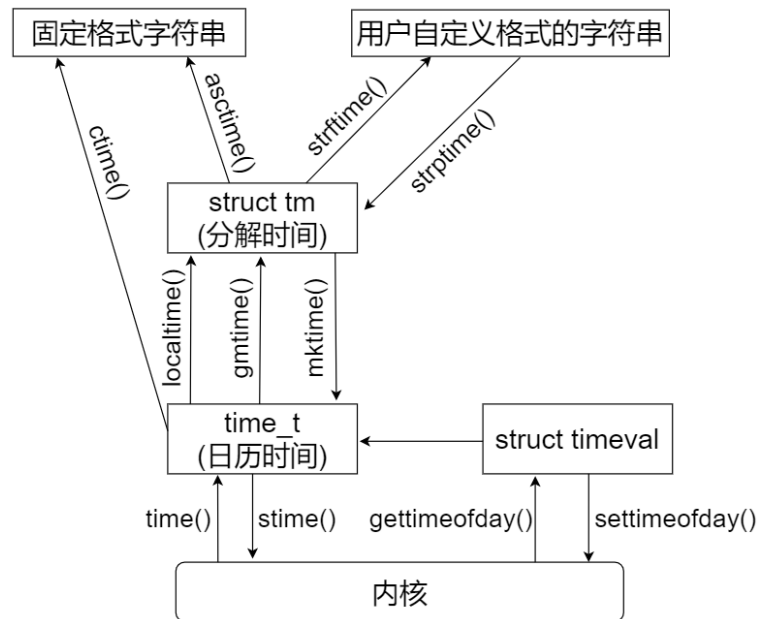


图 7.2.16 时间相关 API 总结

通过上图可以帮助大家快速理解各个函数的功能、作用，大家加油！
本小节到这里就结束了。

7.3 进程时间

进程时间指的是进程从创建后(也就是程序运行后)到目前为止这段时间内使用 CPU 资源的时间总数，出于记录的目的，内核把 CPU 时间（进程时间）分为以下两个部分：

- 用户 CPU 时间：进程在用户空间（用户态）下运行所花费的 CPU 时间。有时也成为虚拟时间（virtual time）。
- 系统 CPU 时间：进程在内核空间（内核态）下运行所花费的 CPU 时间。这是内核执行系统调用或代表进程执行的其它任务（譬如，服务页错误）所花费的时间。

一般来说，进程时间指的是用户 CPU 时间和系统 CPU 时间的总和，也就是总的 CPU 时间。

Tips：进程时间不等于程序的整个生命周期所消耗的时间，如果进程一直处于休眠状态（进程被挂起，不会得到系统调度），那么它并不会使用 CPU 资源，所以休眠的这段时间并不计算在进程时间中。

7.3.1 times 函数

times()函数用于获取当前进程时间，其函数原型如下所示：

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buf);
```

使用该函数需要包含头文件<sys/times.h>。

函数参数和返回值含义如下：

buf: times()会将当前进程时间信息存在一个 struct tms 结构体数据中，所以我们需要提供 struct tms 变量，使用参数 buf 指向该变量，关于 struct tms 结构体稍后给大家介绍。

返回值: 返回值类型为 `clock_t` (实质是 `long` 类型), 调用成功情况下, 将返回从过去任意的一个时间点 (譬如系统启动时间) 所经过的时钟滴答数 (其实就是系统节拍数), 将(节拍数 / 节拍率)便可得到秒数, 返回值可能会超过 `clock_t` 所能表示的范围 (溢出); 调用失败返回 -1, 并设置 `errno`。

如果我们想查看程序运行到某一个位置时的进程时间, 或者计算出程序中的某一段代码执行过程所花费的进程时间, 都可以使用 `times()` 函数来实现。

`struct tms` 结构体内容如下所示:

示例代码 7.3.1 `struct tms` 结构体

```
struct tms {
    clock_t tms_utime; /* user time, 进程的用户 CPU 时间, tms_utime 个系统节拍数 */
    clock_t tms_stime; /* system time, 进程的系统 CPU 时间, tms_stime 个系统节拍数 */
    clock_t tms_cutime; /* user time of children, 已死掉子进程的 tms_utime + tms_cutime 时间总和 */
    clock_t tms_cstime; /* system time of children, 已死掉子进程的 tms_stime + tms_cstime 时间总和 */
};
```

测试

下面我们演示了通过 `times()` 来计算程序中某一段代码执行所耗费的进程时间和总的的时间, 测试程序如下所示:

示例代码 7.3.2 `times` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    struct tms t_buf_start;
    struct tms t_buf_end;
    clock_t t_start;
    clock_t t_end;
    long tck;
    int i, j;

    /* 获取系统的节拍率 */
    tck = sysconf(_SC_CLK_TCK);

    /* 开始时间 */
    t_start = times(&t_buf_start);
    if (-1 == t_start) {
        perror("times error");
        exit(-1);
    }

    /* *****需要进行测试的代码段***** */
    for (i = 0; i < 20000; i++)
```

```

        for (j = 0; j < 20000; j++)
            ;

sleep(1);        //休眠挂起
/* *****end***** */

/* 结束时间 */
t_end = times(&t_buf_end);
if (-1 == t_end) {
    perror("times error");
    exit(-1);
}

/* 打印时间 */
printf("时间总和: %f 秒\n", (t_end - t_start) / (double)tck);
printf("用户 CPU 时间: %f 秒\n", (t_buf_end.tms_utime - t_buf_start.tms_utime) / (double)tck);
printf("系统 CPU 时间: %f 秒\n", (t_buf_end.tms_stime - t_buf_start.tms_stime) / (double)tck);
exit(0);
}

```

首先, 笔者先对测试程序做一个简单地介绍, 程序中使用 `sysconf(_SC_CLK_TCK)` 获取到系统节拍率, 程序还使用了一个库函数 `sleep()`, 该函数也是本章将要向大家介绍的函数, 具体参考 7.5.1 小节中的介绍。示例代码 7.3.2 中对如下代码段进行了测试:

```

for (i = 0; i < 20000; i++)
    for (j = 0; j < 20000; j++)
        ;

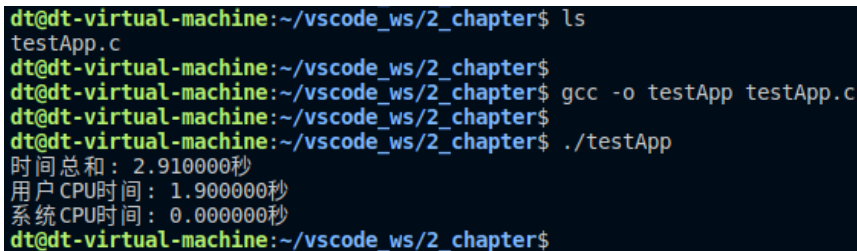
```

```

sleep(1);        //休眠挂起

```

接下来编译运行, 测试结果如下:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
时间总和: 2.910000秒
用户 CPU时间: 1.900000秒
系统 CPU时间: 0.000000秒
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.3.1 测试结果

可以看到用户 CPU 时间为 1.9 秒, 系统 CPU 时间为 0 秒, 也就是说测试的这段代码并没有进入内核态运行, 所以总的进程时间 = 用户 CPU 时间 + 系统 CPU 时间 = 1.9 秒。

图 7.3.1 中显示的时间总和并不是总的进程时间, 前面也给大家解释过, 这个时间总和指的是从起点到终点锁经过的时间, 并不是进程时间, 这里大家要理解。时间总和包括了进程处于休眠状态时消耗的时间 (`sleep` 等会让进程挂起、进入休眠状态), 可以发现时间总和比进程时间多 1 秒, 其实这一秒就是进程处于休眠状态的时间。

7.3.2 clock 函数

库函数 `clock()` 提供了一个更为简单的方式用于进程时间, 它的返回值描述了进程使用的总的 CPU 时间 (也就是进程时间, 包括用户 CPU 时间和系统 CPU 时间), 其函数原型如下所示:

```
#include <time.h>
```

```
clock_t clock(void);
```

使用该函数需要包含头文件 `<time.h>`。

函数参数和返回值含义如下:

无参数。

返回值: 返回值是到目前为止程序的进程时间, 为 `clock_t` 类型, 注意 `clock()` 的返回值并不是系统节拍数, 如果想要获得秒数, 请除以 `CLOCKS_PER_SEC` (这是一个宏)。如果返回的进程时间不可用或其值无法表示, 则该返回值是 -1。

`clock()` 函数虽然可以很方便的获取总的进程时间, 但并不能获取到单独的用户 CPU 时间和系统 CPU 时间, 在实际编程当中, 根据自己的需要选择。

测试

对示例代码 7.3.2 进行简单地修改, 使用 `clock()` 获取到待测试代码段所消耗的进程时间, 如下:

示例代码 7.3.3 clock 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[])
{
    clock_t t_start;
    clock_t t_end;
    int i, j;

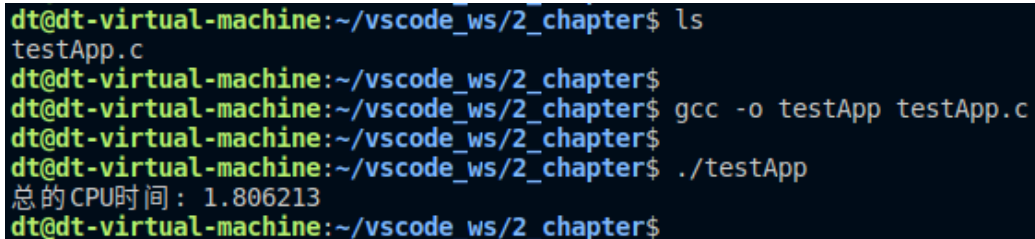
    /* 开始时间 */
    t_start = clock();
    if (-1 == t_start)
        exit(-1);

    /* *****需要进行测试的代码段***** */
    for (i = 0; i < 20000; i++)
        for (j = 0; j < 20000; j++)
            ;
    /* *****end***** */

    /* 结束时间 */
    t_end = clock();
    if (-1 == t_end)
        exit(-1);
}
```

```
/* 打印时间 */  
printf("总的 CPU 时间: %f\n", (t_end - t_start) / (double)CLOCKS_PER_SEC);  
exit(0);  
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
总的 CPU时间: 1.806213  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.3.2 测试结果

7.4 产生随机数

在应用编程当中可能会用到随机数, 譬如老板让你编写一个抽奖的小程序, 编号 0~100, 分为特等奖 1 个、一等奖 2 个、二等级 3 以及三等级 4 个, 也就是说需要从 0~100 个编号中每次随机抽取一个号码, 这就需要用到随机数。那在 Linux 应用编程中如何去产生随机数呢? 本小节就来学习生成随机数。

随机数与伪随机数

随机数是随机出现, 没有任何规律的一组数列。在我们编程当中, 是没有办法获得真正意义上的随机数列的, 这是一种理想的情况, 在我们的程序当中想要使用随机数列, 只能通过算法得到一个伪随机数序列, 那在编程当中说到的随机数, 基本都是指伪随机数。

C 语言函数库中提供了很多函数用于产生伪随机数, 其中最常用的是通过 `rand()`和 `srand()`产生随机数, 本小节就以这两个函数为例向大家介绍如何在我们的程序中获得随机数列。

rand 函数

`rand()`函数用于获取随机数, 多次调用 `rand()`可得到一组随机数序列, 其函数原型如下:

```
#include <stdlib.h>
```

```
int rand(void);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下:

返回值: 返回一个介于 0 到 `RAND_MAX` (包含) 之间的值, 也就是数学上的 $[0, \text{RAND_MAX}]$ 。

程度当中调用 `rand()`可以得到 $[0, \text{RAND_MAX}]$ 之间的伪随机数, 多次调用 `rand()`便可以生成一组伪随机数序列, 但是这里有个问题, 就是每一次运行程序所得到的随机数序列都是相同的, 那如何使得每一次启动应用程序所得到的随机数序列是不一样的呢? 那就通过设置不同的随机数种子, 可通过 `srand()`设置随机数种子。

如果没有调用 `srand()`设置随机数种子的情况下, `rand()`会将 1 作为随机数种子, 如果随机数种子相同, 那么每一次启动应用程序所得到的随机数序列就是一样的, 所以每次启动应用程序需要设置不同的随机数种子, 这样就可以使得程序每次运行所得到随机数序列不同。

srand 函数

使用 `srand()`函数为 `rand()`设置随机数种子, 其函数原型如下所示:


```
#include <stdlib.h>
```

```
void srand(unsigned int seed);
```

函数参数和返回值含义如下:

seed: 指定一个随机数中, int 类型的数据, 一般尝尝将当前时间作为随机数种子赋值给参数 seed, 譬如 time(NULL), 因为每次启动应用程序时间上是一样的, 所以就使得程序中设置的随机数种子在每次启动程序时是不一样的。

返回值: void

常用的用法 `srand(time(NULL));`

测试

使用 rand() 和 srand() 产生一组伪随机数, 数值范围为 [0~100], 将其打印出来:

示例代码 7.4.1 生成随机数示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[])
{
    int random_number_arr[8];
    int count;

    /* 设置随机数种子 */
    srand(time(NULL));

    /* 生成伪随机数 */
    for (count = 0; count < 8; count++)
        random_number_arr[count] = rand() % 100;

    /* 打印随机数数组 */
    printf("[");
    for (count = 0; count < 8; count++) {
        printf("%d", random_number_arr[count]);
        if (count != 8 - 1)
            printf(", ");
    }

    printf("]\n");
    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[48, 58, 25, 54, 67, 51, 96, 79]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[40, 35, 61, 55, 71, 86, 44, 72]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[46, 10, 1, 26, 84, 45, 76, 74]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[61, 15, 68, 29, 87, 53, 3, 37]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.4.1 测试结果

从图中可以发现，每一次得到的[0~100]之间的随机数数组都是不同的（数组不同，不是产生的随机数不同），因为程序中将 rand()的随机数种子设置为 srand(time(NULL))，直接等于 time_t 时间值，意味着每次启动种子都不一样，所以能够产生不同的随机数数组。

本小节关于在 Linux 下使用随机数就给大家介绍这么多，产生随机数的 API 函数并不仅仅只有这些，除此之外，譬如还有 random()、srandom()、initstate()、setstate()等，这里便不再给大家一一介绍了，在我们使用 man 手册查看系统调用或 C 库函数帮助信息时，在帮助信息页面 SEE ALSO 栏会列举出与本函数有关联的一些命令、系统调用或 C 库函数等，如下所示（譬如执行 man 3 srand 查看）：

```
SEE ALSO
    drand48(3), random(3)

COLOPHON
    This page is part of release 4.04 of the Linux man-pages project. A description
    can be found at http://www.kernel.org/doc/man-pages/.
```

图 7.4.2 see also

7.5 休眠

有时需要将进程暂停或休眠一段时间，进入休眠状态之后，程序将暂停运行，直到休眠结束。常用的系统调用和 C 库函数有 sleep()、usleep()以及 nanosleep()，这些函数在应用程序当中通常作为延时使用，譬如延时 1 秒钟，本小节将一一介绍。

7.5.1 秒级休眠: sleep

sleep()是一个 C 库函数，从函数名字面意思便可以知道该函数的作用了，简单地说，sleep()就是让程序“休息”一会，然后再继续工作。其函数原型如下所示：

```
#include <unistd.h>
```

```
unsigned int sleep(unsigned int seconds);
```

使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

seconds: 休眠时长，以秒为单位。

返回值: 如果休眠时长为参数 seconds 所指定的秒数，则返回 0；若被信号中断则返回剩余的秒数。

sleep()是一个秒级别休眠函数，程序在休眠过程中，是可以被其它信号所打断的，关于信号这些内容，将会在后面章节向大家介绍。

测试

编写一个简单地程序，调用 sleep()函数让程序暂停运行（休眠）3 秒钟。

示例代码 7.5.1 sleep 函数使用示例

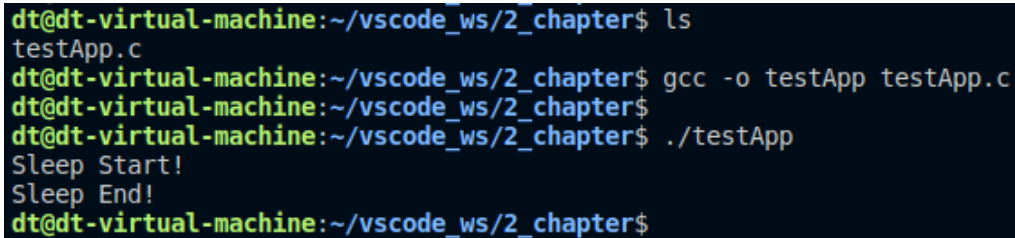
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    puts("Sleep Start!");

    /* 让程序休眠 3 秒钟 */
    sleep(3);

    puts("Sleep End!");
    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Sleep Start!
Sleep End!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.5.1 sleep 测试结果

7.5.2 微秒级休眠: usleep

usleep()同样也是一个C库函数,与sleep()的区别在于休眠时长精度不同,usleep()支持微秒级程序休眠,其函数原型如下所示:

```
#include <unistd.h>
```

```
int usleep(useconds_t usec);
```

函数参数和返回值含义如下:

usec: 休眠时长,以微秒为单位。

返回值: 成功返回 0; 失败返回-1,并设置 errno。

测试

使用 usleep()函数让程序休眠 3 秒钟。

示例代码 7.5.2 usleep 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main(void)
```

```
{
    puts("Sleep Start!");
```

```

/* 让程序休眠 3 秒钟(3*1000*1000 微秒) */
usleep(3 * 1000 * 1000);

puts("Sleep End!");
exit(0);
}

```

运行结果:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Sleep Start!
Sleep End!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 7.5.2 usleep 休眠

7.5.3 高精度休眠: nanosleep

nanosleep()与 sleep()以及 usleep()类似,都用于程序休眠,但 nanosleep()具有更高精度来设置休眠时间长度,支持纳秒级时长设置。与 sleep()、usleep()不同的是,nanosleep()是一个 Linux 系统调用,其函数原型如下所示:

```
#include <time.h>
```

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

使用该函数需要包含头文件<time.h>。

函数参数与返回值含义如下:

req: 一个 struct timespec 结构体指针,指向一个 struct timespec 变量,用于设置休眠时间长度,可精确到纳秒级别。

rem: 也是一个 struct timespec 结构体指针,指向一个 struct timespec 变量,也可设置 NULL。

返回值: 在成功休眠达到请求的时间间隔后,nanosleep()返回 0;如果中途被信号中断或遇到错误,则返回-1,并将剩余时间记录在参数 rem 指向的 struct timespec 结构体变量中(参数 rem 不为 NULL 的情况下,如果为 NULL 表示不接收剩余时间),还会设置 errno 标识错误类型。

在 5.2.3 小节中介绍了 struct timespec 结构体,该结构体包含了两个成员变量,秒(tv_sec)和纳秒(tv_nsec),具体定义可参考示例代码 5.2.2。

测试

示例代码 7.5.3 nanosleep 函数使用示例

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    struct timespec request_t;

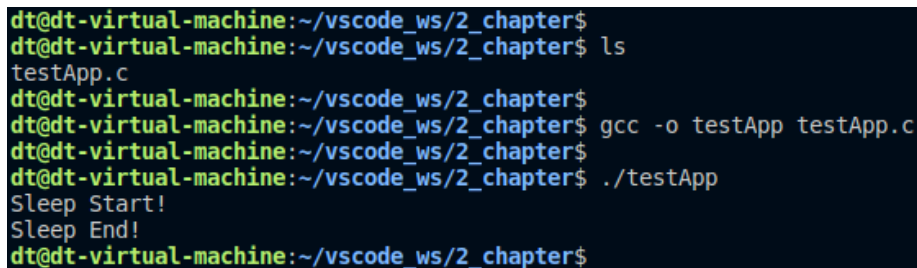
```

```
puts("Sleep Start!");

/* 让程序休眠 3 秒钟 */
request_t.tv_sec = 3;
request_t.tv_nsec = 0;
nanosleep(&request_t, NULL);

puts("Sleep End!");
exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Sleep Start!
Sleep End!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.5.3 nanosleep 休眠

前面说到, 在应用程序当中, 通常使用这些函数作为延时功能, 譬如在程序当中需要延时一秒钟、延时 5 毫秒等应用场景时, 那么就可以使用这些函数来实现; 但是大家需要注意, 休眠状态下, 该进程会失去 CPU 使用权, 退出系统调度队列, 直到休眠结束。在一个裸机程序当中, 通常使用 for 循环 (或双重 for 循环) 语句来实现延时等待, 譬如在 for 循环当中执行 nop 空指令, 也就意味着即使在延时等待情况下, CPU 也是一直都在工作; 由此可知, 应用程序当中使用休眠用作延时功能, 并不是裸机程序中的 nop 空指令延时, 一旦执行 sleep(), 进程便主动交出 CPU 使用权, 暂时退出系统调度队列, 在休眠结束前, 该进程的指令将得不到执行。

7.6 申请堆内存

在操作系统下, 内存资源是由操作系统进行管理、分配的, 当应用程序想要内存时 (这里指的是堆内存), 可以向操作系统申请内存, 然后使用内存; 当不再需要时, 将申请的内存释放、归还给操作系统; 在许多的应用程序当中, 往往都会有这种需求, 譬如为一些数据结构动态分配/释放内存空间, 本小节向大家介绍应用程序如何向操作系统申请堆内存。

7.6.1 在堆上分配内存: malloc 和 free

Linux C 程序当中一般使用 malloc() 函数为程序分配一段堆内存, 而使用 free() 函数来释放这段内存, 先来看下 malloc() 函数原型, 如下所示:

```
#include <stdlib.h>
```

```
void *malloc(size_t size);
```

使用该函数需要包含头文件 <stdlib.h>。

函数参数和返回值含义如下:

size: 需要分配的内存大小, 以字节为单位。

返回值: 返回值为 `void *` 类型, 如果申请分配内存成功, 将返回一个指向该段内存的指针, `void *` 并不是说没有返回值或者返回空指针, 而是返回的指针类型未知, 所以在调用 `malloc()` 时通常需要进行强制类型转换, 将 `void *` 指针类型转换成我们希望的类型; 如果分配内存失败 (譬如系统堆内存不足) 将返回 `NULL`, 如果参数 `size` 为 0, 返回值也是 `NULL`。

`malloc()` 在堆区分配一块指定大小的内存空间, 用来存放数据。这块内存空间在函数执行完成后不会被初始化, 它们的值是未知的, 所以通常需要程序员对 `malloc()` 分配的堆内存进行初始化操作。

在堆上分配的内存, 需要开发者自己手动释放掉, 通常使用 `free()` 函数释放堆内存, `free()` 函数原型如下所示:

```
#include <stdlib.h>
```

```
void free(void *ptr);
```

使用该函数同样需要包含头文件 `<stdlib.h>`。

函数参数和返回值含义如下:

ptr: 指向需要被释放的堆内存对应的指针。

返回值: 无返回值。

测试

示例代码 7.6.1 `malloc()` 和 `free()` 申请/释放堆内存

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC_MEM_SIZE (1 * 1024 * 1024)

int main(int argc, char *argv[])
{
    char *base = NULL;

    /* 申请堆内存 */
    base = (char *)malloc(MALLOC_MEM_SIZE);
    if (NULL == base) {
        printf("malloc error\n");
        exit(-1);
    }

    /* 初始化申请到的堆内存 */
    memset(base, 0x0, MALLOC_MEM_SIZE);

    /* 使用内存 */
    /* ..... */

    /* 释放内存 */
    free(base);
}
```

```
    exit(0);  
}
```

调用 free()还是不调用 free()

在学习文件 IO 基础章节内容时曾向大家介绍过, Linux 系统中, 当一个进程终止时, 内核会自动关闭它没有关闭的所有文件(该进程打开的文件, 但是在进程终止时未调用 close()关闭它)。同样, 对于内存来说, 也是如此! 当进程终止时, 内核会将其占用的所有内存都返还给操作系统, 这包括在堆内存中由 malloc() 函数所分配的内存空间。基于内存的这一自动释放机制, 很多应用程序通常会省略对 free() 函数的调用。

这在程序中分配了多块内存的情况下可能会特别有用, 因为加入多次对 free() 的调用不但会消耗大量的 CPU 时间, 而且可能会使代码趋于复杂。

虽然依靠终止进程来自动释放内存对大多数程序来说是可以接受的, 但最好能够在程序中显式调用 free() 释放内存, 首先其一, 显式调用 free() 能使程序具有更好的可读性和可维护性; 其二, 对于很多程序来说, 申请的内存并不是在程序的生命周期中一直需要, 大多数情况下, 都是根据代码需求动态申请、释放的, 如果申请的内存对程序来说已经不再需要了, 那么就已经把它释放、归还给操作系统, 如果持续占用, 将会导致内存泄漏, 也就是人们常说的“你的程序在吃内存”!

7.6.2 在堆上分配内存的其它方法

除了 malloc() 外, C 函数库中还提供了一系列在堆上分配内存的其它函数, 本小节将逐一介绍。

用 calloc() 分配内存

calloc() 函数用来动态地分配内存空间并初始化为 0, 其函数原型如下所示:

```
#include <stdlib.h>
```

```
void *calloc(size_t nmemb, size_t size);
```

使用该函数同样也需要包含头文件 <stdlib.h>。

calloc() 在堆中动态地分配 nmemb 个长度为 size 的连续空间, 并将每一个字节都初始化为 0。所以它的结果是分配了 nmemb * size 个字节长度的内存空间, 并且每个字节的值都是 0。

返回值: 分配成功返回指向该内存的地址, 失败则返回 NULL。

calloc() 与 malloc() 的一个重要区别是: calloc() 在动态分配完内存后, 自动初始化该内存空间为零, 而 malloc() 不初始化, 里边数据是未知的垃圾数据。下面的两种写法是等价的:

```
// calloc() 分配内存空间并初始化
```

```
char *buf1 = (char *)calloc(10, 2);
```

```
// malloc() 分配内存空间并用 memset() 初始化
```

```
char *buf2 = (char *)malloc(10 * 2);
```

```
memset(buf2, 0, 20);
```

测试

编写测试代码, 将用户输入的一组数字存放到堆内存中, 并打印出来。

示例代码 7.6.2 calloc 函数使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
int *base = NULL;
int i;

/* 校验传参 */
if (2 > argc)
    exit(-1);

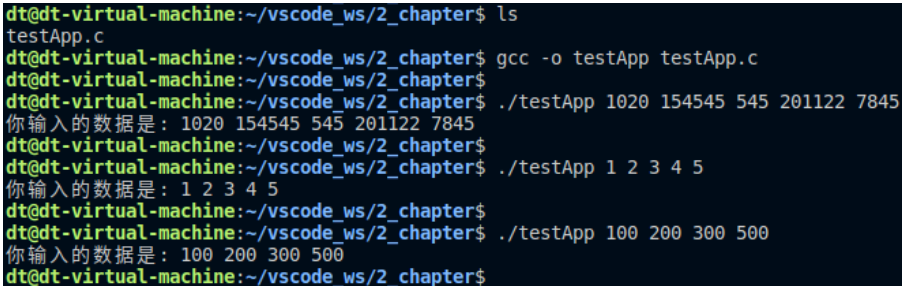
/* 使用 calloc 申请内存 */
base = (int *)calloc(argc - 1, sizeof(int));
if (NULL == base) {
    printf("calloc error\n");
    exit(-1);
}

/* 将字符串转为 int 型数据存放在 base 指向的内存中 */
for (i = 0; i < argc - 1; i++)
    base[i] = atoi(argv[i+1]);

/* 打印 base 数组中的数据 */
printf("你输入的数据是: ");
for (i = 0; i < argc - 1; i++)
    printf("%d ", base[i]);
putchar('\n');

/* 释放内存 */
free(base);
exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 1020 154545 545 201122 7845
你输入的数据是: 1020 154545 545 201122 7845
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 1 2 3 4 5
你输入的数据是: 1 2 3 4 5
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 100 200 300 500
你输入的数据是: 100 200 300 500
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.6.1 测试结果

7.6.3 分配对其内存

C 函数库中还提供了一系列在堆上分配对齐内存的函数, 对齐内存某些应用场合非常有必要, 常用于分配对其内存的库函数有: `posix_memalign()`、`aligned_alloc()`、`memalign()`、`valloc()`、`pvalloc()`, 它们的函数原型如下所示:

```
#include <stdlib.h>
```



```
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
void *valloc(size_t size);

#include <malloc.h>

void *memalign(size_t alignment, size_t size);
void *pvalloc(size_t size);
```

使用 `posix_memalign()`、`aligned_alloc()`、`valloc()`这三个函数时需要包含头文件`<stdlib.h>`，而使用`memalign()`、`pvalloc()`这两个函数时需要包含头文件`<malloc.h>`。前面介绍的`malloc()`、`calloc()`分配内存返回的地址其实也是对齐的，但是它俩的对齐都是固定的，并且对其的字节边界比较小，譬如在 32 位系统中，通常是以 8 字节为边界进行对其，在 64 位系统中是以 16 字节进行对其。如果想实现更大字节的对齐，则需要使用本小节介绍的函数。

posix_memalign()函数

`posix_memalign()`函数用于在堆上分配 `size` 个字节大小的对齐内存空间，将`*memptr` 指向分配的空间，分配的内存地址将是参数 `alignment` 的整数倍。参数 `alignment` 表示对齐字节数，`alignment` 必须是 2 的幂次方（譬如 2^4 、 2^5 、 2^8 等），同时也要是 `sizeof(void *)`的整数倍，对于 32 位系统来说，`sizeof(void *)`等于 4，如果是 64 位系统 `sizeof(void *)`等于 8。

函数参数和返回值含义如下：

memptr: `void **`类型的指针，内存申请成功后会将分配的内存地址存放在`*memptr` 中。

alignment: 设置内存对其的字节数，`alignment` 必须是 2 的幂次方（譬如 2^4 、 2^5 、 2^8 等），同时也要是 `sizeof(void *)`的整数倍。

size: 设置分配的内存大小，以字节为单位，如果参数 `size` 等于 0，那么`*memptr` 中的值是 `NULL`。

返回值: 成功将返回 0；失败返回非 0 值。

示例代码

示例代码 7.6.3 `posix_memalign` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int *base = NULL;
    int ret;

    /* 申请内存: 256 字节对齐 */
    ret = posix_memalign((void **)&base, 256, 1024);
    if (0 != ret) {
        printf("posix_memalign error\n");
        exit(-1);
    }
}
```

```
/* 使用内存 */
// base[0] = 0;
// base[1] = 1;
// base[2] = 2;
// base[3] = 3;

/* 释放内存 */
free(base);
exit(0);
}
```

aligned_alloc()函数

aligned_alloc()函数用于分配 size 个字节大小的内存空间，返回指向该空间的指针。

函数参数和返回值含义如下：

alignment: 用于设置对齐字节大小，alignment 必须是 2 的幂次方（譬如 2⁴、2⁵、2⁸ 等）。

size: 设置分配的内存大小，以字节为单位。参数 size 必须是参数 alignment 的整数倍。

返回值: 成功将返回内存空间的指针，内存空间的起始地址是参数 alignment 的整数倍；失败返回 NULL。

使用示例

示例代码 7.6.4 aligned_alloc 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int *base = NULL;

    /* 申请内存: 256 字节对齐 */
    base = (int *)aligned_alloc(256, 256 * 4);
    if (base == NULL) {
        printf("aligned_alloc error\n");
        exit(-1);
    }

    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

memalign()函数

memalign()与 aligned_alloc()参数是一样的,它们之间的区别在于:对于参数 size 必须是参数 alignment 的整数倍这个限制条件,memalign()并没有这个限制条件。

Tips: memalign()函数已经过时了,并不提倡使用!

使用示例

示例代码 7.6.5 memalign 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    int *base = NULL;

    /* 申请内存: 256 字节对齐 */
    base = (int *)memalign(256, 1024);
    if (base == NULL) {
        printf("memalign error\n");
        exit(-1);
    }

    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

valloc()函数

valloc()分配 size 个字节大小的内存空间,返回指向该内存空间的指针,内存空间的地址是页大小(pagesize)的倍数。

valloc()与 memalign()类似,只不过 valloc()函数内部实现中,使用了页大小作为对齐的长度,在程序当中,可以通过系统调用 getpagesize()来获取内存的页大小。

Tips: valloc()函数已经过时了,并不提倡使用!

使用示例

示例代码 7.6.6 valloc 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
    int *base = NULL;

    /* 申请内存: 1024 个字节 */
    base = (int *)valloc(1024);
    if (base == NULL) {
        printf("valloc error\n");
        exit(-1);
    }

    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

7.7 proc 文件系统

proc 文件系统是一个虚拟文件系统，它以文件系统的方式为应用层访问系统内核数据提供了接口，用户和应用程序可以通过 proc 文件系统得到系统信息和进程相关信息，对 proc 文件系统的读写作为与内核进行通信的一种手段。但是与普通文件不同的是，proc 文件系统是动态创建的，文件本身并不存在于磁盘当中、只存在于内存当中，与 devfs 一样，都被称为虚拟文件系统。

最初构建 proc 文件系统是为了提供有关系统中进程相关的信息，但是由于这个文件系统非常有用，因此内核中的很多信息也开始使用它来报告，或启用动态运行时配置。内核构建 proc 虚拟文件系统，它会将内核运行时的一些关键数据信息以文件的方式呈现在 proc 文件系统下的一些特定文件中，这样相当于将一些不可见的内核中的数据以可视化的方式呈现给应用层。

proc 文件系统挂载在系统的 /proc 目录下，对于内核开发者（譬如驱动开发工程师）来说，proc 文件系统给了开发者一种调试内核的方法：通过查看 /proc/xxx 文件来获取到内核特定数据结构的值，在添加了新功能前后进行对比，就可以判断此功能所产生的影响是否合理。

/proc 目录下中包含了一些目录和虚拟文件，如下所示：

```
dt@dt-virtual-machine:/proc$ pwd
/proc
dt@dt-virtual-machine:/proc$ ls
1      13      1996    2137    2236    234      2461    26      31      37      44      74898    990      kallsyms      self
10     133     2       214     224     2346    247     260     312    37425  45     8        99686    kcore         slabinfo
100038 14      20      2141    225     235     2472    261    3160   37426  46     885     acpi         keys          softirqs
1002   1490   2008    215     2250    236     2476    262    3165   37431  495    887     asound      key-users     stat
100513 15      2029    2158    2255    237     248     263    317    37432  50     888     buddyinfo   kmsg         swaps
101898 152     2030    216     226     2373    249     264    3179   37433  51     801     bus         kpagecgroup  sys          sys
101924 1524    2035    2169    22627   238     25     265    3183   37434  52     89334    cgroups     kpagecount   sysrq-trigger
101994 1560    2056    217     227     239     250     266    3184   37435  53     9        cmdline     kpageflags   sysvipc
102033 1591    206     2170    228     2393    2501    267    32     37436  54     90101    consoles   loadavg      thread-self
102039 16      2068    2173    229     24     251     268    3208   37437  55     90102    cpuinfo     locks        timer_list
1046   1607    2069    2176    2294    240     2513    269    3222   37438  56     904     crypto      mdstat       tty
1057   1612    207     2179    2295    2403    252     27     3235   378    57     905     devices     meminfo      uptime
1071   1621    2073    218     2298    241     253     270    3271   38     58     906     diskstats   misc         version
11     1716    2081    2180    2299    2410    254     271    33     387    6       917     dma         modules      version_signature
113    1735    2086    2182    230     2416    255     272    3304   39     62     92540    driver      mounts       vmallocinfo
114    1739    21     2184    2304    242     2550    273    3315   4       63     93945    execdomains mpt          vmstat
115    18     2100    219    2306    2421    2554    274    3330   40     64     943     fb          mtrr        zoneinfo
116    19     2104    22     231     243     256     28     3372   400    65     98557    filesystems net
117    1903    2115    220    232     2430    2564    2813   34     405    66     98560    fs          pagetypeinfo
118    1904    2117    2209    2320    244     2568    290    3403   406    69     98562    interrupts partitions
12     1909    213     221    2322    2440    257     292    351    42     7       98563    iomem      sched_debug
1229   1911    2130    222    233     245     258     30     352    43     70     98564    ioports    schedstat
124    1993    2131    223    2332    246     259     3090   36     438    71     98639    irq        scsi
```

图 7.7.1 proc 文件系统下的目录和文件

可以看到/proc 目录下有很多以数字命名的文件夹, 譬如 100038、2299、98560, 这些数字对应的其实就是一个一个的进程 PID 号, 每一个进程在内核中都会存在一个编号, 通过此编号来区分不同的进程, 这个编号就是 PID 号, 关于 PID、以及进程相关的信息将会在后面章节内容向大家介绍。

所以这些以数字命名的文件夹中记录了这些进程相关的信息, 不同的信息通过不同的虚拟文件呈现出来, 关于这些信息将会在后面章节内容向大家介绍。

/proc 目录下除了文件夹之外, 还有很多的虚拟文件, 譬如 buddyinfo、cgroups、cmdline、version 等等, 不同的文件记录了不同信息, 关于这些文件记录的信息和意思如下:

- cmdline: 内核启动参数;
- cpuinfo: CPU 相关信息;
- iomem: IO 设备的内存使用情况;
- interrupts: 显示被占用的中断号和占用者相关的信息;
- ioports: IO 端口的使用情况;
- kcore: 系统物理内存映像, 不可读取;
- loadavg: 系统平均负载;
- meminfo: 物理内存和交换分区使用情况;
- modules: 加载的模块列表;
- mounts: 挂载的文件系统列表;
- partitions: 系统识别的分区表;
- swaps: 交换分区的利用情况;
- version: 内核版本信息;
- uptime: 系统运行时间;

7.7.1 proc 文件系统的使用

proc 文件系统的使用就是去读取/proc 目录下的这些文件, 获取文件中记录的信息, 可以直接使用 cat 命令读取, 也可以在应用程序中调用 open()打开、然后再使用 read()函数读取。

使用 cat 命令读取

在 Linux 系统下直接使用 cat 命令查看/proc 目录下的虚拟文件, 譬如"cat /proc/version"查看内核版本相关信息:

```
dt@dt-virtual-machine:~$ cat /proc/version
Linux version 4.15.0-132-generic (buildd@lgw01-amd64-034) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #136~16.04.1-Ubuntu SMP Tue Jan 12 18:22:20 UTC 2021
dt@dt-virtual-machine:~$
```

图 7.7.2 cat 命令查看/proc 目录下的文件

使用 read()函数读取

编写一个简单程序，使用 read()函数读取/proc/version 文件。

示例代码 7.7.1 应用程序中读取 proc 文件系统

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char buf[512] = {0};
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("/proc/version", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 读取文件 */
    ret = read(fd, buf, sizeof(buf));
    if (-1 == ret) {
        perror("read error");
        exit(-1);
    }

    /* 打印信息 */
    puts(buf);

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Linux version 4.15.0-132-generic (build@lgw01-amd64-034) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #136~16.04.1-Ubuntu SMP Tue Jan 12 18:22:20 UTC 2021
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.7.3 测试结果

第八章 信号: 基础

本章将讨论信号, 虽然信号的基本概念比较简单, 但是其所涉及到的细节内容比较多, 所以本章篇幅也会相对比较长。事实上, 在很多应用程序当中, 都会存在处理异步事件这种需求, 而信号提供了一种处理异步事件的方法, 所以信号机制在 Linux 早期版本中就已经提供了支持, 随着 Linux 内核版本的更新迭代, 其对信号机制的支持更加完善。

本章将会讨论如下主题内容。

- 信号的基本概念;
- 信号的分类、Linux 提供的各种不同的信号及其作用;
- 发出信号以及响应信号, 信号由“谁”发送、由“谁”处理以及如何处理;
- 进程在默认情况下对信号的响应方式;
- 使用进程信号掩码来阻塞信号、以及等待信号等相关概念;
- 如何暂停进程的执行, 并等待信号的到达。

8.1 基本概念

信号是事件发生时对进程的通知机制,也可以把它称为软件中断。信号与硬件中断的相似之处在于能够打断程序当前执行的正常流程,其实是在软件层次上对中断机制的一种模拟。大多数情况下,是无法预测信号达到的准确时间,所以,信号提供了一种处理异步事件的方法。

信号的目的是用来通信的

一个具有合适权限的进程能够向另一个进程发送信号,信号的这一用法可作为一种同步技术,甚至是进程间通信(IPC)的原始形式。信号可以由“谁”发出呢?以下列举的很多情况均可以产生信号:

- 硬件发生异常,即硬件检测到错误条件并通知内核,随即再由内核发送相应的信号给相关进程。硬件检测到异常的例子包括执行一条异常的机器语言指令,诸如,除数为0、数组访问越界导致引用了无法访问的内存区域等,这些异常情况都会被硬件检测到,并通知内核,然后内核为该异常情况发生时正在运行的进程发送适当的信号以通知进程。
- 用于在终端下输入了能够产生信号的特殊字符。譬如在终端上按下 CTRL+C 组合按键可以产生中断信号(SIGINT),通过这个方法可以终止在前台运行的进程;按下 CTRL+Z 组合按键可以产生暂停信号(SIGCONT),通过这个方法可以暂停当前前台运行的进程。
- 进程调用 kill()系统调用可将任意信号发送给另一个进程或进程组。当然对此是有所限制的,接收信号的进程和发送信号的进程的所有者必须相同,亦或者发送信号的进程的所有者是 root 超级用户。
- 用户可以通过 kill 命令将信号发送给其它进程。kill 命令想必大家都会使用,通常会通过 kill 命令来“杀死”(终止)一个进程,譬如在终端下执行"kill -9 xxx"来杀死 PID 为 xxx 的进程。kill 命令其内部的实现原理便是通过 kill()系统调用来完成的。
- 发生了软件事件,即当检测到某种软件条件已经发生。这里指的不是硬件产生的条件(如除数为0、引用无法访问的内存区域等),而是软件的触发条件、触发了某种软件条件(进程所设置的定时器已经超时、进程执行的 CPU 时间超限、进程的某个子进程退出等等情况)。

进程同样也可以向自身发送信号,然而发送给进程的诸多信号中,大多数都是来自于内核。

以上便是可以产生信号的多种不同的条件,总的来看,信号的目的都是用于通信的,当发生某种情况下,通过信号将情况“告知”相应的进程,从而达到同步、通信的目的。

信号由谁处理、怎么处理

信号通常是发送给对应的进程,当信号到达后,该进程需要做出相应的处理措施,通常进程会视具体信号执行以下操作之一:

- 忽略信号。也就是说,当信号到达进程后,该进程并不会去理会它、直接忽略,就好像是没有出该信号,信号对该进程不会产生任何影响。事实上,大多数信号都可以使用这种方式进行处理,但有两种信号却决不能被忽略,它们是 SIGKILL 和 SIGSTOP,这两种信号不能被忽略的原因是:它们向内核和超级用户提供了使进程终止或停止的可靠方法。另外,如果忽略某些由硬件异常产生的信号,则进程的运行行为是未定义的。
- 捕获信号。当信号到达进程后,执行预先绑定好的信号处理函数。为了做到这一点,要通知内核在某种信号发生时,执行用户自定义的处理函数,该处理函数中将会对该信号事件作出相应的处理, Linux 系统提供了 signal()系统调用可用于注册信号的处理函数,将会在后面向大家介绍。
- 执行系统默认操作。进程不对该信号事件作出处理,而是交由系统进行处理,每一种信号都会有其对应的系统默认的处理方式,8.3 小节中对此有进行介绍。需要注意的是,对大多数信号来说,系统默认的处理方式就是终止该进程。

信号是异步的

信号是异步事件的经典实例,产生信号的事件对进程而言是随机出现的,进程无法预测该事件产生的准确时间,进程不能够通过简单地测试一个变量或使用系统调用来判断是否产生了一个信号,这就如同硬件中断事件,程序是无法得知中断事件产生的具体时间,只有当产生中断事件时,才会告知程序、然后打断当前程序的正常执行流程、跳转去执行中断服务函数,这就是异步处理方式。

信号本质上是 int 类型数字编号

信号本质上是 int 类型的数字编号,这就好比硬件中断所对应的中断号。内核针对每个信号,都给它定义了一个唯一的整数编号,从数字 1 开始顺序展开。并且每一个信号都有其对应的名字(其实就是一个宏),信号名字与信号编号乃是一一对应关系,但是由于每个信号的实际编号随着系统的不同可能会不一样,所以在程序当中一般都使用信号的符号名(也就是宏定义)。

这些信号在<signal.h>头文件中定义,每个信号都是以 SIGxxx 开头,如下所示:

示例代码 8.1.1 信号定义

```

/* Signals. */
#define SIGHUP      1      /* Hangup (POSIX). */
#define SIGINT     2      /* Interrupt (ANSI). */
#define SIGQUIT    3      /* Quit (POSIX). */
#define SIGILL     4      /* Illegal instruction (ANSI). */
#define SIGTRAP    5      /* Trace trap (POSIX). */
#define SIGABRT    6      /* Abort (ANSI). */
#define SIGIOT     6      /* IOT trap (4.2 BSD). */
#define SIGBUS     7      /* BUS error (4.2 BSD). */
#define SIGFPE     8      /* Floating-point exception (ANSI). */
#define SIGKILL    9      /* Kill, unblockable (POSIX). */
#define SIGUSR1   10      /* User-defined signal 1 (POSIX). */
#define SIGSEGV   11      /* Segmentation violation (ANSI). */
#define SIGUSR2   12      /* User-defined signal 2 (POSIX). */
#define SIGPIPE   13      /* Broken pipe (POSIX). */
#define SIGALRM   14      /* Alarm clock (POSIX). */
#define SIGTERM   15      /* Termination (ANSI). */
#define SIGSTKFLT 16      /* Stack fault. */
#define SIGCLD    SIGCHLD /* Same as SIGCHLD (System V). */
#define SIGCHLD  17      /* Child status has changed (POSIX). */
#define SIGCONT   18      /* Continue (POSIX). */
#define SIGSTOP   19      /* Stop, unblockable (POSIX). */
#define SIGTSTP   20      /* Keyboard stop (POSIX). */
#define SIGTTIN   21      /* Background read from tty (POSIX). */
#define SIGTTOU   22      /* Background write to tty (POSIX). */
#define SIGURG    23      /* Urgent condition on socket (4.2 BSD). */
#define SIGXCPU   24      /* CPU limit exceeded (4.2 BSD). */
#define SIGXFSZ   25      /* File size limit exceeded (4.2 BSD). */
#define SIGVTALRM 26      /* Virtual alarm clock (4.2 BSD). */
#define SIGPROF   27      /* Profiling alarm clock (4.2 BSD). */
#define SIGWINCH  28      /* Window size change (4.3 BSD, Sun). */
#define SIGPOLL   SIGIO  /* Pollable event occurred (System V).

```

```
#define SIGIO      29      /* I/O now possible (4.2 BSD).  */
#define SIGPWR    30      /* Power failure restart (System V).  */
#define SIGSYS    31      /* Bad system call.  */
#define SIGUNUSED 31
```

不存在编号为 0 的信号, 从示例代码 8.1.1 中也可以看到, 信号编号是从 1 开始的, 事实上 kill() 函数对信号编号 0 有着特殊的应用, 关于这个文件将会在后面的内容向大家介绍。

8.2 信号的分类

Linux 系统下可对信号从两个不同的角度进行分类, 从可靠性方面将信号分为可靠信号与不可靠信号; 而从实时性方面将信号分为实时信号与非实时信号, 本小节将对这些信号的分类进行简单地介绍。

8.2.1 可靠信号与不可靠信号

Linux 信号机制基本上是从 UNIX 系统中继承过来的, 早期 UNIX 系统中的信号机制比较简单和原始, 后来在实践中暴露出一些问题, 它的主要问题是:

- 进程每次处理信号后, 就将对信号的响应设置为系统默认操作。在某些情况下, 将导致对信号的错误处理; 因此, 用户如果不希望这样的操作, 那么就要在信号处理函数结尾再一次调用 signal(), 重新为该信号绑定相应的处理函数。
- 因此导致, 早期 UNIX 下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失(处理信号时又来了新的信号, 则导致信号丢失)。

Linux 支持不可靠信号, 但是对不可靠信号机制做了改进: 在调用完信号处理函数后, 不必重新调用 signal()。因此, Linux 下的不可靠信号问题主要指的是信号可能丢失。在 Linux 系统下, 信号值小于 SIGRTMIN (34) 的信号都是不可靠信号, 这就是"不可靠信号"的来源, 所以示例代码 8.1.1 中所列举的信号都是不可靠信号。

随着时间的发展, 实践证明, 有必要对信号的原始机制加以改进和扩充, 所以, 后来出现的各种 UNIX 版本分别在这方面进行了研究, 力图实现"可靠信号"。由于原来定义的信号已有许多应用, 不好再做改动, 最终只好又新增加了一些信号(SIGRTMIN~SIGRTMAX), 并在一开始就把它们定义为可靠信号, 在 Linux 系统下使用"kill -l"命令可查看到所有信号, 如下所示:

```
dt@dt-virtual-machine:~$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT    7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
11) SIGSEGV   12) SIGUSR2   13) SIGPIPE   14) SIGALRM   15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD  18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN   22) SIGTTOU  23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF  28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS    34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
dt@dt-virtual-machine:~$
```

图 8.2.1 kill 命令查看所有信号

Tips: 括号")"前面的数字对应信号的编号, 编号 1~31 所对应的是不可靠信号, 编号 34~64 对应的是可靠信号, 从图中可知, 可靠信号并没有一个具体对应的名字, 而是使用了 SIGRTMIN+N 或 SIGRTMAX-N 的方式来表示。

可靠信号支持排队, 不会丢失, 同时, 信号的发送和绑定也出现了新版本, 信号发送函数 sigqueue() 及信号绑定函数 sigaction()。

早期 UNIX 系统只定义了 31 种信号, 而 Linux 3.x 支持 64 种信号, 编号 1-64(SIGRTMIN=34, SIGRTMAX=64), 将来可能进一步增加, 这需要得到内核的支持。前 31 种信号已经有了预定义值, 每个信号有了确定的用途、含义以及对应的名字, 并且每种信号都有各自的系统默认操作。如按键盘的 CTRL+C 时, 会产生 SIGINT 信号, 对该信号的系统默认操作就是终止进程, 后 32 个信号表示可靠信号。

8.2.2 实时信号与非实时信号

实时信号与非实时信号其实是从时间关系上进行的分类, 与可靠信号与不可靠信号是相互对应的, 非实时信号都不支持排队, 都是不可靠信号; 实时信号都支持排队, 都是可靠信号。实时信号保证了发送的多个信号都能被接收, 实时信号是 POSIX 标准的一部分, 可用于应用进程。

一般我们也把非实时信号(不可靠信号)称为标准信号, 如果文档中用到了这个词, 那么大家要知道, 这里指的就是非实时信号(不可靠信号)。关于更多实时信号相关内容将会在 8.10 小节中介绍。

8.3 常见信号与默认行为

前面说到, Linux 下对标准信号(不可靠信号、非实时信号)的编号为 1~31, 如示例代码 8.1.1 所示, 接下来将介绍这些信号以及这些信号所对应的系统默认操作。

● SIGINT

当用户在终端按下中断字符(通常是 CTRL + C)时, 内核将发送 SIGINT 信号给前台进程组中的每一个进程。该信号的系统默认操作是终止进程的运行。所以通常我们都会使用 CTRL + C 来终止一个占用前台的进程, 原因在于大部分的进程会将该信号交给系统去处理, 从而执行该信号的系统默认操作。

● SIGQUIT

当用户在终端按下退出字符(通常是 CTRL + \)时, 内核将发送 SIGQUIT 信号给前台进程组中的每一个进程。该信号的系统默认操作是终止进程的运行、并生成可用于调试的核心转储文件。进程如果陷入无限循环、或不再响应时, 使用 SIGQUIT 信号就很合适。所以对于一个前台进程, 既可以在终端按下中断字符 CTRL + C、也可以按下退出字符 CTRL + \来终止, 当然前提条件是, 此进程会将 SIGINT 信号或 SIGQUIT 信号交给系统处理(也就是没有将信号忽略或捕获), 进入执行该信号所对应的系统默认操作。

● SIGILL

如果进程试图执行非法(即格式不正确)的机器语言指令, 系统将向进程发送该信号。该信号的系统默认操作是终止进程的运行。

● SIGABRT

当进程调用 abort()系统调用时(进程异常终止), 系统会向该进程发送 SIGABRT 信号。该信号的系统默认操作是终止进程、并生成核心转储文件。

● SIGBUS

产生该信号(总线错误, bus error)表示发生了某种内存访问错误。该信号的系统默认操作是终止进程。

● SIGFPE

该信号因特定类型的算术错误而产生, 譬如除以 0。该信号的系统默认操作是终止进程。

● SIGKILL

此信号为“必杀(sure kill)”信号, 用于杀死进程的终极办法, 此信号无法被进程阻塞、忽略或者捕获, 故而“一击必杀”, 总能终止进程。使用 SIGINT 信号和 SIGQUIT 信号虽然能终止进程, 但是前提条件是进程并没有忽略或捕获这些信号, 如果使用 SIGINT 或 SIGQUIT 无法终止进程, 那就使用“必杀信号”SIGKILL 吧。Linux 下有一个 kill 命令, kill 命令可用于向进程发送信号, 我们会使用“kill -9 xxx”命令来终止一个进程(xxx 表示进程的 pid), 这里的-9 其实指的就是发送编号为 9 的信号, 也就是 SIGKILL 信号。

● SIGUSR1

该信号和 SIGUSR2 信号供程序员自定义使用, 内核绝不会为进程产生这些信号, 在我们的程序中, 可以使用这些信号来互通通知事件的发生, 或是进程彼此同步操作。该信号的系统默认操作是终止进程。

● SIGSEGV

这一信号非常常见, 当应用程序对内存的引用无效时, 操作系统就会向该应用程序发送该信号。引起对内存无效引用的原因很多, C 语言中引发这些事件往往是解引用的指针里包含了错误地址 (譬如, 未初始化的指针), 或者传递了一个无效参数供函数调用等。该信号的系统默认操作是终止进程。

● SIGUSR2

与 SIGUSR1 信号相同。

● SIGPIPE

涉及到管道和 socket, 当进程向已经关闭的管道、FIFO 或套接字写入信息时, 那么系统将发送该信号给进程。该信号的系统默认操作是终止进程。

● SIGALRM

与系统调用 alarm() 或 setitimer() 有关, 应用程序中可以调用 alarm() 或 setitimer() 函数来设置一个定时器, 当定时器定时时间到, 那么内核将会发送 SIGALRM 信号给该应用程序, 关于 alarm() 或 setitimer() 函数的使用, 后面将会进行讲解。该信号的系统默认操作是终止进程。

● SIGTERM

这是用于终止进程的标准信号, 也是 kill 命令所发送的默认信号 (kill xxx, xxx 表示进程 pid), 有时我们会直接使用 "kill -9 xxx" 显式向进程发送 SIGKILL 信号来终止进程, 然而这一做法通常是错误的, 精心设计的应用程序应该会捕获 SIGTERM 信号、并为其绑定一个处理函数, 当该进程收到 SIGTERM 信号时, 会在处理函数中清除临时文件以及释放其它资源, 再而退出程序。如果直接使用 SIGKILL 信号终止进程, 从而跳过了 SIGTERM 信号的处理函数, 通常 SIGKILL 终止进程是不友好的方式、是暴力的方式, 这种方式应该作为最后手段, 应首先尝试使用 SIGTERM, 实在不行再使用最后手段 SIGKILL。

● SIGCHLD

当父进程的某一个子进程终止时, 内核会向父进程发送该信号。当父进程的某一个子进程因收到信号而停止或恢复时, 内核也可能向父进程发送该信号。注意这里说的停止并不是终止, 你可以理解为暂停。该信号的系统默认操作是忽略此信号, 如果父进程希望被告知其子进程的这种状态改变, 则应捕获此信号。

● SIGCLD

与 SIGCHLD 信号同义。

● SIGCONT

将该信号发送给已停止的进程, 进程将会恢复运行。当进程接收到此信号时并不处于停止状态, 系统默认操作是忽略该信号, 但如果进程处于停止状态, 则系统默认操作是使该进程继续运行。

● SIGSTOP

这是一个“必停”信号, 用于停止进程 (注意停止不是终止, 停止只是暂停运行、进程并没有终止), 应用程序无法将该信号忽略或者捕获, 故而总能停止进程。

● SIGTSTP

这也是一个停止信号, 当用户在终端按下停止字符 (通常是 CTRL + Z), 那么系统会将 SIGTSTP 信号发送给前台进程组中的每一个进程, 使其停止运行。

● SIGXCPU

当进程的 CPU 时间超出对应的资源限制时, 内核将发送此信号给该进程。

● SIGVTALRM

应用程序调用 setitimer() 函数设置一个虚拟定时器, 当定时器定时时间到时, 内核将会发送该信号给进程。

● SIGWINCH

在窗口环境中, 当终端窗口尺寸发生变化时 (譬如用户手动调整了大小, 应用程序调用 `ioctl()` 设置了大小等), 系统会向前台进程组中的每一个进程发送该信号。

● SIGPOLL/SIGIO

这两个信号同义。这两个信号将会在高级 IO 章节内容中使用到, 用于提示一个异步 IO 事件的发生, 譬如应用程序打开的文件描述符发生了 I/O 事件时, 内核会向应用程序发送 SIGIO 信号。

● SIGSYS

如果进程发起的系统调用有误, 那么内核将发送该信号给对应的进程。

以上就是关于这些信号的简单介绍内容, 以上所介绍的这些信号并不包括 Linux 下所有的信号, 仅给大家介绍一下常见信号, 表 8.3.1 将对这些信号进行总结。

表 8.3.1 Linux 信号总结

信号名称	编号	描述	系统默认操作
SIGINT	2	终端中断符	term
SIGQUIT	3	终端退出符	term+core
SIGILL	4	非法硬件指令	term+core
SIGABRT	6	异常终止 (abort)	term+core
SIGBUS	7	内存访问错误	term+core
SIGFPE	8	算术异常	term+core
SIGKILL	9	终极终止信号	term
SIGUSR1	10	用户自定义信号 1	term
SIGSEGV	11	无效的内存引用	term+core
SIGUSR2	12	用户自定义信号 2	term
SIGPIPE	13	管道关闭	term
SIGALRM	14	定时器超时 (alarm)	term
SIGTERM	15	终止进程	term
SIGCHLD/SIGCLD	17	子进程终止或停止	ignore
SIGCONT	18	使停止状态的进程继续运行	cont
SIGSTOP	19	停止进程	stop
SIGTSTP	20	终端停止符	stop
SIGXCPU	24	超过 CPU 限制	term+core
SIGVTALRM	26	虚拟定时器超时	term
SIGWINCH	28	终端窗口尺寸发生变化	ignore
SIGPOLL/SIGIO	29	异步 I/O	term/ignore
SIGSYS	31	无效系统调用	term+core

Tips: 上表中, term 表示终止进程; core 表示生成核心转储文件, 核心转储文件可用于调试, 这个便不再介绍了; ignore 表示忽略信号; cont 表示继续运行进程; stop 表示停止进程 (注意停止不等于终止, 而是暂停)。

8.4 进程对信号的处理

当进程接收到内核或用户发送过来的信号之后, 根据具体信号可以采取不同的处理方式: 忽略信号、捕获信号或者执行系统默认操作。Linux 系统提供了系统调用 `signal()` 和 `sigaction()` 两个函数用于设置信号的处理方式, 本小节将向大家介绍这两个系统调用的使用方法。

8.4.1 signal()函数

本节描述系统调用 `signal()`, `signal()`函数是 Linux 系统下设置信号处理方式最简单的接口, 可将信号的处理方式设置为捕获信号、忽略信号以及系统默认操作, 此函数原型如下所示:

```
#include <signal.h>
```

```
typedef void (*sig_t)(int);
```

```
sig_t signal(int signum, sig_t handler);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下:

signum: 此参数指定需要进行设置的信号, 可使用信号名(宏)或信号的数字编号, 建议使用信号名。

handler: `sig_t` 类型的函数指针, 指向信号对应的信号处理函数, 当进程接收到信号后会自动执行该处理函数; 参数 `handler` 既可以设置为用户自定义的函数, 也就是捕获信号时需要执行的处理函数, 也可以设置为 `SIG_IGN` 或 `SIG_DFL`, `SIG_IGN` 表示此进程需要忽略该信号, `SIG_DFL` 则表示设置为系统默认操作。`sig_t` 函数指针的 `int` 类型参数指的是, 当前触发该函数的信号, 可将多个信号绑定到同一个信号处理函数上, 此时就可通过此参数来判断当前触发的是哪个信号。

Tips: `SIG_IGN`、`SIG_DFL` 分别取值如下:

```
/* Fake signal functions. */
```

```
#define SIG_ERR ((sig_t) -1) /* Error return. */
```

```
#define SIG_DFL ((sig_t) 0) /* Default action. */
```

```
#define SIG_IGN ((sig_t) 1) /* Ignore signal. */
```

返回值: 此函数的返回值也是一个 `sig_t` 类型的函数指针, 成功情况下的返回值则是指向在此之前的信号处理函数; 如果出错则返回 `SIG_ERR`, 并会设置 `errno`。

由此可知, `signal()`函数可以根据第二个参数 `handler` 的不同设置情况, 可对信号进行不同的处理。

测试

`signal()`函数的用法其实非常简单, 为信号设置相应的处理方式, 接下来编写一个简单地示例代码对 `signal()`函数进行测试。

示例代码 8.4.1 `signal()`函数使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <signal.h>
```

```
static void sig_handler(int sig)
```

```
{
```

```
    printf("Received signal: %d\n", sig);
```

```
}
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    sig_t ret = NULL;
```

```
    ret = signal(SIGINT, (sig_t)sig_handler);
```

```
    if (SIG_ERR == ret) {
```

```

    perror("signal error");
    exit(-1);
}

/* 死循环 */
for (;;) {}

exit(0);
}

```

在上述示例代码中, 我们通过 `signal()` 函数将 `SIGINT` (2) 信号绑定到了一个用户自定的处理函数上 `sig_handler(int sig)`, 当进程收到 `SIGINT` 信号后会执行该函数然后运行 `printf` 打印语句。当运行程序之后, 程序会卡在 `for` 死循环处, 此时在终端按下中断符 `CTRL + C`, 系统便会给前台进程组中的每一个进程发送 `SIGINT` 信号, 我们测试程序便会收到该信号。

运行测试:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2

```

图 8.4.1 测试结果

当运行程序之后, 程序会占用终端称为一个前台进程, 此时按下中断符便会打印出信息 (^C 表示按下了中断符)。平时大家使用 `CTRL + C` 可以终止一个进程, 而这里却不能通过这种方式来终止这个测试程序, 原因在于测试程序中捕获了该信号, 而对应的处理方式仅仅只是打印一条语句、而并不终止进程。

那此时该怎么关闭这个测试程序呢? 前面给大家介绍了“一击必杀”信号 `SIGKILL` (编号为 9), 可向该进程发送 `SIGKILL` 暴力终止该进程, 当然一般不推荐大家这样使用, 如果实在没办法才采取这种措施。新打开一个终端, 使用 `ps` 命令找到该进程的 `pid` 号, 再使用 `kill` 命令, 如下所示:

```

dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ps -aux | grep testApp | grep -v grep
dt      13946  99.7  0.0  4352  644 pts/19  R+   16:17  12:29  ./testApp
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ kill -9 13946
dt@dt-virtual-machine:~$

```

图 8.4.2 一击必杀

此时测试程序就会强制终止:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
已杀死
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 8.4.3 测试程序被终止

Tips: 普通用户只能杀死该用户自己的进程, 无权限杀死其它用户的进程。

我们再执行一次测试程序, 这里将测试程序放在后台运行, 然后再按下中断符:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[1] 14158
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
  PID TTY          TIME CMD
 3304 pts/19    00:00:13 bash
 14158 pts/19    00:07:23 testApp
 14203 pts/19    00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.4.4 测试结果

按下中断符发现进程并没有收到 SIGINT 信号, 原因很简单, 因为进程并不是前台进程、而是一个后台进程, 按下中断符时系统并不会给后台进程发送 SIGINT 信号。可以使用 kill 命令手动发送信号给我们的进程:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
  PID TTY          TIME CMD
 3304 pts/19    00:00:13 bash
 14158 pts/19    00:11:10 testApp
 14226 pts/19    00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.4.5 测试结果

两种不同状态下信号的处理方式

通过上面的介绍, 以及我们的测试实验, 不知大家是否出现了一个疑问? 如果程序中没有调用 signal() 函数为信号设置相应的处理方式, 亦或者程序刚启动起来并未运行到 signal() 处, 那么这时进程接收到一个信号后是如何处理的呢? 带着这个问题来聊一聊。

● 程序启动

当一个应用程序刚启动的时候(或者程序中没有调用 signal() 函数), 通常情况下, 进程对所有信号的处理方式都设置为系统默认操作。所以如果在我们的程序当中, 没有调用 signal() 为信号设置处理方式, 则默认的处理方式便是系统默认操作。

所以为什么大家平时都可以使用 CTRL + C 中断符来终止一个进程, 因为大部分情况下, 应用程序中并不会为 SIGINT 信号设置处理方式, 所以该信号的处理方式便是系统默认操作, 当接收到信号之后便执行系统默认操作, 而 SIGINT 信号的系统默认操作便是终止进程。

● 进程创建

当一个进程调用 fork() 创建子进程时, 其子进程将会继承父进程的信号处理方式, 因为子进程在开始时复制了父进程的内存映像, 所以信号捕获函数的地址在子进程中是有意义的。

8.4.2 sigaction()函数

除了 `signal()` 之外, `sigaction()` 系统调用是设置信号处理方式的另一选择, 事实上, 推荐大家使用 `sigaction()` 函数。虽然 `signal()` 函数简单好用, 而 `sigaction()` 更为复杂, 但作为回报, `sigaction()` 也更具灵活性以及移植性。

`sigaction()` 允许单独获取信号的处理函数而不是设置, 并且还可以设置各种属性对调用信号处理函数时的行为施以更加精准的控制, 其函数原型如下所示:

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

使用该函数需要包含头文件 `<signal.h>`。

函数参数和返回值含义如下:

signum: 需要设置的信号, 除了 `SIGKILL` 信号和 `SIGSTOP` 信号之外的任何信号。

act: `act` 参数是一个 `struct sigaction` 类型指针, 指向一个 `struct sigaction` 数据结构, 该数据结构描述了信号的处理方式, 稍后介绍该数据结构; 如果参数 `act` 不为 `NULL`, 则表示需要为信号设置新的处理方式; 如果参数 `act` 为 `NULL`, 则表示无需改变信号当前的处理方式。

oldact: `oldact` 参数也是一个 `struct sigaction` 类型指针, 指向一个 `struct sigaction` 数据结构。如果参数 `oldact` 不为 `NULL`, 则会将信号之前的处理方式等信息通过参数 `oldact` 返回出来; 如果无意获取此类信息, 那么可将该参数设置为 `NULL`。

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

struct sigaction 结构体

示例代码 8.4.2 struct sigaction 结构体

```
struct sigaction {  
    void (*sa_handler)(int);  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*sa_restorer)(void);  
};
```

结构体成员介绍:

- **sa_handler:** 指定信号处理函数, 与 `signal()` 函数的 `handler` 参数相同。
- **sa_sigaction:** 也用于指定信号处理函数, 这是一个替代的信号处理函数, 他提供了更多的参数, 可以通过该函数获取到更多信息, 这些信号通过 `siginfo_t` 参数获取, 稍后介绍该数据结构; `sa_handler` 和 `sa_sigaction` 是互斥的, 不能同时设置, 对于标准信号来说, 使用 `sa_handler` 就可以了, 可通过 `SA_SIGINFO` 标志进行选择。
- **sa_mask:** 参数 `sa_mask` 定义了一组信号, 当进程在执行由 `sa_handler` 所定义的信号处理函数之前, 会先将这组信号添加到进程的信号掩码字段中, 当进程执行完处理函数之后再恢复信号掩码, 将这组信号从信号掩码字段中删除。当进程在执行信号处理函数期间, 可能又收到了同样的信号或其它信号, 从而打断当前信号处理函数的执行, 这就好点像中断嵌套; 通常我们在执行信号处理函数期间不希望被另一个信号所打断, 那么怎么做呢? 那么就是通过信号掩码来实现, 如果进程接收到了信号掩码中的这些信号, 那么这个信号将会被阻塞暂时不能得到处理, 直到这些信号从进程的信号掩码中移除。在信号处理函数调用时, 进程会自动将当前处理的信号添加到信号掩码字段中, 这样保证了在处理一个给定的信号时, 如果此信号再次发生, 那么它将会被阻塞。如果用户还需要在阻塞其它的信号, 则可以通过设置参

数 `sa_mask` 来完成 (此参数是 `sigset_t` 类型变量, 关于该类型的介绍信息请看 8.6.1 小节内容, 关于信号掩码还会在 8.7.1 小节中进一步介绍), 信号掩码可以避免一些信号之间的竞争状态 (也称为竞态)。

- `sa_restorer`: 该成员已过时, 不要再使用了。
- `sa_flags`: 参数 `sa_flags` 指定了一组标志, 这些标志用于控制信号的处理过程, 可设置为如下这些标志 (多个标志使用位或 "|" 组合):

SA_NOCLDSTOP

如果 `signum` 为 `SIGCHLD`, 则子进程停止时 (即当它们接收到 `SIGSTOP`、`SIGTSTP`、`SIGTTIN` 或 `SIGTTOU` 中的一种时) 或恢复 (即它们接收到 `SIGCONT`) 时不会收到 `SIGCHLD` 信号。

SA_NOCLDWAIT

如果 `signum` 是 `SIGCHLD`, 则在子进程终止时不要将其转变为僵尸进程。

SA_NODEFER

不要阻塞从某个信号自身的信号处理函数中接收此信号。也就是说当进程此时正在执行某个信号的处理函数, 默认情况下, 进程会自动将该信号添加到进程的信号掩码字段中, 从而在执行信号处理函数期间阻塞该信号, 默认情况下, 我们期望进程在处理一个信号时阻塞同种信号, 否则引起一些竞态条件; 如果设置了 `SA_NODEFER` 标志, 则表示不对它进行阻塞。

SA_RESETHAND

执行完信号处理函数之后, 将信号的处理方式设置为系统默认操作。

SA_RESTART

被信号中断的系统调用, 在信号处理完成之后将自动重新发起。

SA_SIGINFO

如果设置了该标志, 则表示使用 `sa_sigaction` 作为信号处理函数、而不是 `sa_handler`, 关于 `sa_sigaction` 信号处理函数的参数信息。

以上就是关于 `struct sigaction` 结构体相关的内容介绍了, 接下编写程序进行实战测试。

siginfo_t 结构体

示例代码 8.4.3 siginfo_t 结构体

```
siginfo_t {
    int          si_signo;    /* Signal number */
    int          si_errno;    /* An errno value */
    int          si_code;     /* Signal code */
    int          si_trapno;   /* Trap number that caused hardware-generated signal(unused on most architectures) */
    pid_t       si_pid;      /* Sending process ID */
    uid_t       si_uid;      /* Real user ID of sending process */
    int         si_status;    /* Exit value or signal */
    clock_t     si_utime;    /* User time consumed */
    clock_t     si_stime;    /* System time consumed */
    sigval_t    si_value;    /* Signal value */
    int         si_int;      /* POSIX.1b signal */
    void        *si_ptr;     /* POSIX.1b signal */
    int         si_overrun;   /* Timer overrun count; POSIX.1b timers */
    int         si_timerid;   /* Timer ID; POSIX.1b timers */
    void        *si_addr;    /* Memory location which caused fault */
    long        si_band;     /* Band event (was int in glibc 2.3.2 and earlier) */
}
```

```
int      si_fd;          /* File descriptor */
short    si_addr_lsb;   /* Least significant bit of address(since Linux 2.6.32) */
void     *si_call_addr; /* Address of system call instruction(since Linux 3.5) */
int      si_syscall;    /* Number of attempted system call(since Linux 3.5) */
unsigned int si_arch;   /* Architecture of attempted system call(since Linux 3.5) */
}
```

这个结构体就不给大家介绍了，使用 man 手册查看 `sigaction()` 函数帮助信息时，在下面会有介绍。

测试

这里使用 `sigaction()` 函数实现与示例代码 8.4.1 相同的功能。

示例代码 8.4.4 `sigaction()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void sig_handler(int sig)
{
    printf("Received signal: %d\n", sig);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int ret;

    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    ret = sigaction(SIGINT, &sig, NULL);
    if (-1 == ret) {
        perror("sigaction error");
        exit(-1);
    }

    /* 死循环 */
    for (;;) {}

    exit(0);
}
```

运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
```

图 8.4.6 测试结果

关于信号处理函数说明

一般而言, 将信号处理函数设计越简单越好, 这就好比中断处理函数, 越快越好, 不要在处理函数中做大量消耗 CPU 时间的事情, 这一个重要的原因在于, 设计的越简单这将降低引发信号竞争条件的风险。

8.5 向进程发送信号

与 kill 命令相类似, Linux 系统提供了 kill()系统调用, 一个进程可通过 kill()向另一个进程发送信号; 除了 kill()系统调用之外, Linux 系统还提供了系统调用 killpg()以及库函数 raise(), 也可用于实现发送信号的功能, 本小节将向大家进行介绍。

8.5.1 kill()函数

kill()系统调用可将信号发送给指定的进程或进程组中的每一个进程, 其函数原型如下所示:

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

使用该函数需要包含头文件<sys/types.h>和<signal.h>。

函数参数和返回值含义如下:

pid: 参数 pid 为正数的情况下, 用于指定接收此信号的进程 pid; 除此之外, 参数 pid 也可设置为 0 或 -1 以及小于-1 等不同值, 稍后给说明。

sig: 参数 sig 指定需要发送的信号, 也可设置为 0, 如果参数 sig 设置为 0 则表示不发送信号, 但任执行错误检查, 这通常可用于检查参数 pid 指定的进程是否存在。

返回值: 成功返回 0; 失败将返回-1, 并设置 errno。

参数 pid 不同取值含义:

- 如果 pid 为正, 则信号 sig 将发送到 pid 指定的进程。
- 如果 pid 等于 0, 则将 sig 发送到当前进程的进程组中的每个进程。
- 如果 pid 等于-1, 则将 sig 发送到当前进程有权发送信号的每个进程, 但进程 1 (init) 除外。
- 如果 pid 小于-1, 则将 sig 发送到 ID 为-pid 的进程组中的每个进程。

进程中将信号发送给另一个进程是需要权限的, 并不是可以随便给任何一个进程发送信号, 超级用户 root 进程可以将信号发送给任何进程, 但对于非超级用户 (普通用户) 进程来说, 其基本规则是发送者进程的实际用户 ID 或有效用户 ID 必须等于接收者进程的实际用户 ID 或有效用户 ID。

从上面介绍可知, 当 sig 为 0 时, 仍可进行正常执行的错误检查, 但不会发送信号, 这通常可用于确定一个特定的进程是否存在, 如果向一个不存在的进程发送信号, kill() 将会返回 -1, errno 将被设置为 ESRCH, 表示进程不存在。

测试

(1) 使用 kill() 函数向一个指定的进程发送信号。

示例代码 8.5.1 kill() 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int pid;

    /* 判断传参个数 */
    if (2 > argc)
        exit(-1);

    /* 将传入的字符串转为整形数字 */
    pid = atoi(argv[1]);
    printf("pid: %d\n", pid);

    /* 向 pid 指定的进程发送信号 */
    if (-1 == kill(pid, SIGINT)) {
        perror("kill error");
        exit(-1);
    }

    exit(0);
}
```

以上代码通过 kill() 函数向指定进程发送 SIGINT 信号, 可通过外部传参将接收信号的进程 pid 传入到程序中, 再执行该测试代码之前, 需要运行先一个用于接收此信号的进程, 接收信号的进程直接使用示例代码 8.4.4 程序。

运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp1 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp1 &
[1] 21825
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
PID TTY TIME CMD
3304 pts/19 00:00:13 bash
21825 pts/19 00:00:03 testApp1
21826 pts/19 00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 21825
pid: 21825
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 21825
pid: 21825
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 21825
pid: 21825
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.5.1 测试结果

testApp1 是示例代码 8.4.4 对应的程序, testApp 则是示例代码 8.5.1 对应的程序, 首先执行"./testApp1 &"将接收信号的程序置于后台运行(其进程 pid 为 21825), 接着执行"./testApp 21825"向接收信号的进程发送 SIGINT 信号。

8.5.2 raise()

有时进程需要向自身发送信号, raise()函数可用于实现这一要求, raise()函数原型如下所示(此函数为 C 库函数):

```
#include <signal.h>
```

```
int raise(int sig);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下:

sig: 需要发送的信号。

返回值: 成功返回 0; 失败将返回非零值。

raise()其实等价于:

```
kill(getpid(), sig);
```

Tips: getpid()函数用于获取进程自身的 pid。

测试

示例代码 8.5.2 raise()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void sig_handler(int sig)
{
    printf("Received signal: %d\n", sig);
}
```

```
int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int ret;

    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    ret = sigaction(SIGINT, &sig, NULL);
    if (-1 == ret) {
        perror("sigaction error");
        exit(-1);
    }

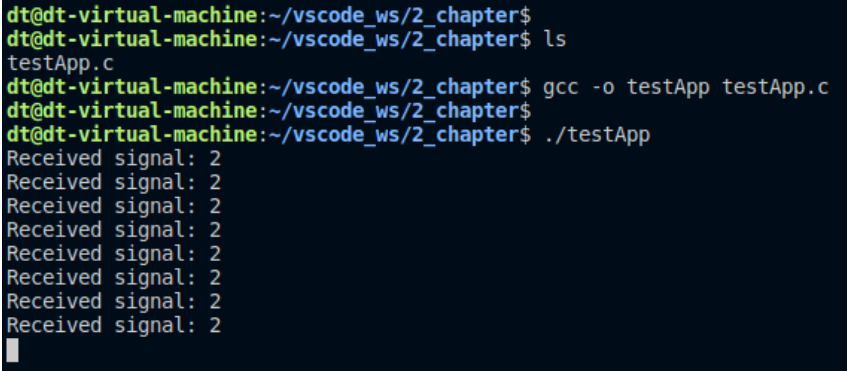
    for (;;) {

        /* 向自身发送 SIGINT 信号 */
        if (0 != raise(SIGINT)) {
            printf("raise error\n");
            exit(-1);
        }

        sleep(3); // 每隔 3 秒发送一次
    }

    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
```

图 8.5.2 测试结果

8.6 alarm()和 pause()函数

本小节向大家介绍两个系统调用 alarm()和 pause()。

8.6.1 alarm()函数

使用 alarm()函数可以设置一个定时器(闹钟),当定时器定时时间到时,内核会向进程发送 SIGALRM 信号,其函数原型如下所示:

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```

函数参数和返回值:

seconds: 设置定时时间,以秒为单位;如果参数 seconds 等于 0,则表示取消之前设置的 alarm 闹钟。

返回值: 如果在调用 alarm()时,之前已经为该进程设置了 alarm 闹钟还没有超时,则该闹钟的剩余值作为本次 alarm()函数调用的返回值,之前设置的闹钟则被新的替代;否则返回 0。

参数 seconds 的值是产生 SIGALRM 信号需要经过的时钟秒数,当这一刻到达时,由内核产生该信号,每个进程只能设置一个 alarm 闹钟;虽然 SIGALRM 信号的系统默认操作是终止进程,但是如果程序当中设置了 alarm 闹钟,但大多数使用闹钟的进程都会捕获此信号。

需要注意的是 alarm 闹钟并不能循环触发,只能触发一次,若想要实现循环触发,可以在 SIGALRM 信号处理函数中再次调用 alarm()函数设置定时器。

测试

使用 alarm()来设计一个闹钟。

示例代码 8.6.1 alarm()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void sig_handler(int sig)
{
    puts("Alarm timeout");
    exit(0);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int second;

    /* 检验传参个数 */
    if (2 > argc)
        exit(-1);

    /* 为 SIGALRM 信号绑定处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGALRM, &sig, NULL)) {
```

```

    perror("sigaction error");
    exit(-1);
}

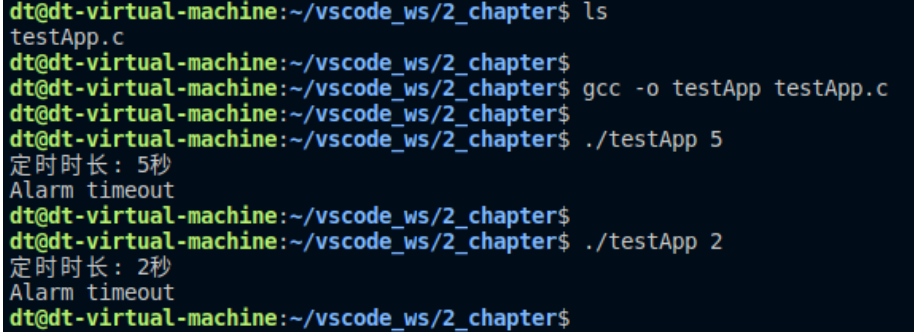
/* 启动 alarm 定时器 */
second = atoi(argv[1]);
printf("定时时长: %d 秒\n", second);
alarm(second);

/* 循环 */
for (;;)
    sleep(1);

exit(0);
}

```

运行测试:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 5
定时时长: 5秒
Alarm timeout
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 2
定时时长: 2秒
Alarm timeout
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 8.6.1 测试结果

8.6.2 pause()函数

pause()系统调用可以使得进程暂停运行、进入休眠状态,直到进程捕获到一个信号为止,只有执行了信号处理函数并从其返回时, pause()才返回,在这种情况下, pause()返回-1,并且将 errno 设置为 EINTR。其函数原型如下所示:

```
#include <unistd.h>
```

```
int pause(void);
```

测试

通过 alarm()和 pause()函数模拟 sleep 功能。

示例代码 8.6.2 alarm()和 pause()模拟 sleep

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <signal.h>
```

```
#include <unistd.h>
```

```
static void sig_handler(int sig)
```

```
{
```

```
    puts("Alarm timeout");
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int second;

    /* 检验传参个数 */
    if (2 > argc)
        exit(-1);

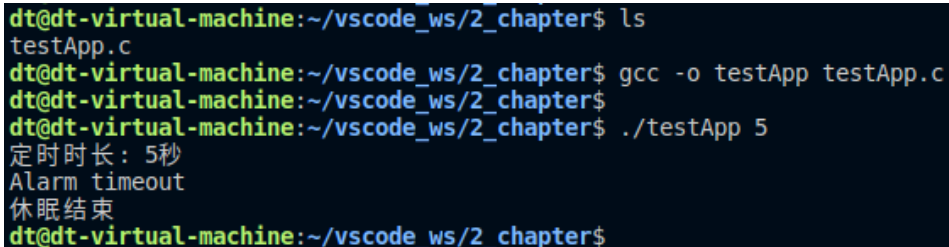
    /* 为 SIGALRM 信号绑定处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGALRM, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    /* 启动 alarm 定时器 */
    second = atoi(argv[1]);
    printf("定时时长: %d 秒\n", second);
    alarm(second);

    /* 进入休眠状态 */
    pause();
    puts("休眠结束");

    exit(0);
}
```

运行测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 5
定时时长: 5秒
Alarm timeout
休眠结束
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.6.2 测试结果

8.7 信号集

通常我们需要有一个能表示多个信号（一组信号）的数据类型---信号集（signal set），很多系统调用都使用到了信号集这种数据类型来作为参数传递，譬如 `sigaction()`函数、`sigprocmask()`函数、`sigpending()`函数等。本小节向大家介绍信号集这个数据类型。

信号集其实就是 `sigset_t` 类型数据结构，来看看：

```
#define _SIGSET_NWORDS    (1024 / (8 * sizeof (unsigned long int)))
typedef struct
{
    unsigned long int __val[_SIGSET_NWORDS];
} sigset_t;
```

使用这个结构体可以表示一组信号，将多个信号添加到该数据结构中，当然 Linux 系统了用于操作 `sigset_t` 信号集的 API，譬如 `sigemptyset()`、`sigfillset()`、`sigaddset()`、`sigdelset()`、`sigismember()`，接下来向大家介绍。

8.7.1 初始化信号集

`sigemptyset()`和 `sigfillset()`用于初始化信号集。`sigemptyset()`初始化信号集，使其不包含任何信号；而 `sigfillset()`函数初始化信号集，使其包含所有信号（包括所有实时信号），函数原型如下：

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

使用这些函数需要包含头文件 `<signal.h>`。

函数参数和返回值含义如下：

set: 指向需要进行初始化的信号集变量。

返回值: 成功返回 0；失败将返回-1，并设置 `errno`。

使用示例

初始化为空信号集：

```
sigset_t sig_set;
```

```
sigemptyset(&sig_set);
```

初始化信号集，使其包含所有信号：

```
sigset_t sig_set;
```

```
sigfillset(&sig_set);
```

8.7.2 向信号集中添加/删除信号

分别使用 `sigaddset()`和 `sigdelset()`函数向信号集中添加或移除一个信号，函数原型如下：

```
#include <signal.h>

int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

函数参数和返回值含义如下:

set: 指向信号集。

signum: 需要添加/删除的信号。

返回值: 成功返回 0; 失败将返回-1, 并设置 `errno`。

使用示例

向信号集中添加信号:

```
sigset_t sig_set;
```

```
sigemptyset(&sig_set);
```

```
sigaddset(&sig_set, SIGINT);
```

从信号集中移除信号:

```
sigset_t sig_set;
```

```
sigfillset(&sig_set);
```

```
sigdelset(&sig_set, SIGINT);
```

8.7.3 测试信号是否在信号集中

使用 `sigismember()` 函数可以测试某一个信号是否在指定的信号集中, 函数原型如下所示:

```
#include <signal.h>
```

```
int sigismember(const sigset_t *set, int signum);
```

函数参数和返回值含义如下:

set: 指定信号集。

signum: 需要进行测试的信号。

返回值: 如果信号 `signum` 在信号集 `set` 中, 则返回 1; 如果不在信号集 `set` 中, 则返回 0; 失败则返回-1, 并设置 `errno`。

使用示例

判断 `SIGINT` 信号是否在 `sig_set` 信号集中:

```
sigset_t sig_set;
```

```
.....
```

```
if (1 == sigismember(&sig_set, SIGINT))
```

```
    puts("信号集中包含 SIGINT 信号");
```

```
else if (!sigismember(&sig_set, SIGINT))
```

```
    puts("信号集中不包含 SIGINT 信号");
```

8.8 获取信号的描述信息

在 Linux 下, 每个信号都有一串与之相对应的字符串描述信息, 用于对该信号进行相应的描述。这些字符串位于 `sys_siglist` 数组中, `sys_siglist` 数组是一个 `char *` 类型的数组, 数组中的每一个元素存放的是一个字符串指针, 指向一个信号描述信息。譬如, 可以使用 `sys_siglist[SIGINT]` 来获取对 `SIGINT` 信号的描述。我们编写一个简单地程序进行测试:

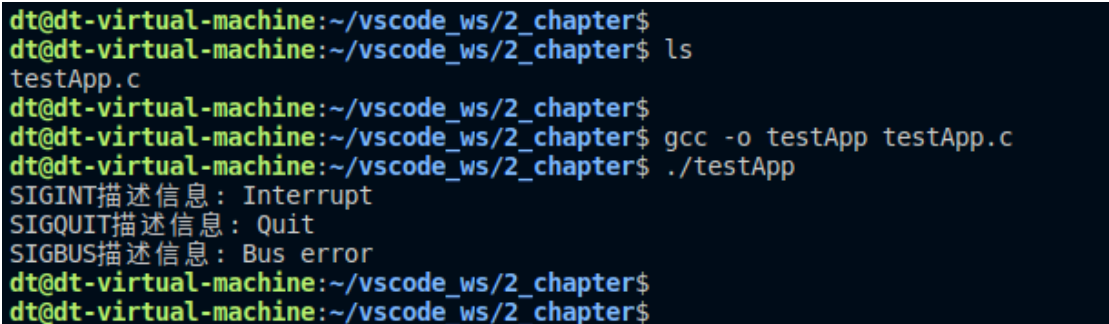
Tips: 使用 `sys_siglist` 数组需要包含 `<signal.h>` 头文件。

[示例代码 8.8.1 从 sys_siglist 数组获取信号描述信息](#)

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("SIGINT 描述信息: %s\n", sys_siglist[SIGINT]);
    printf("SIGQUIT 描述信息: %s\n", sys_siglist[SIGQUIT]);
    printf("SIGBUS 描述信息: %s\n", sys_siglist[SIGBUS]);
    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
SIGINT描述信息: Interrupt
SIGQUIT描述信息: Quit
SIGBUS描述信息: Bus error
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.8.1 测试结果

从图中打印信息可知, 这个描述信息其实非常简洁, 没什么太多的信息。

8.8.1 strsignal()函数

除了直接使用 `sys_siglist` 数组获取描述信息之外, 还可以使用 `strsignal()` 函数。较之于直接引用 `sys_siglist` 数组, 更推荐使用 `strsignal()` 函数, 其函数原型如下所示:

```
#include <string.h>
```

```
char *strsignal(int sig);
```

使用 `strsignal()` 函数需要包含头文件 `<string.h>`, 这是一个库函数。

调用 `strsignal()` 函数将会获取到参数 `sig` 指定的信号对应的描述信息, 返回该描述信息字符串的指针; 函数会对参数 `sig` 进行检查, 若传入的 `sig` 无效, 则会返回 "Unknown signal" 信息。

使用示例

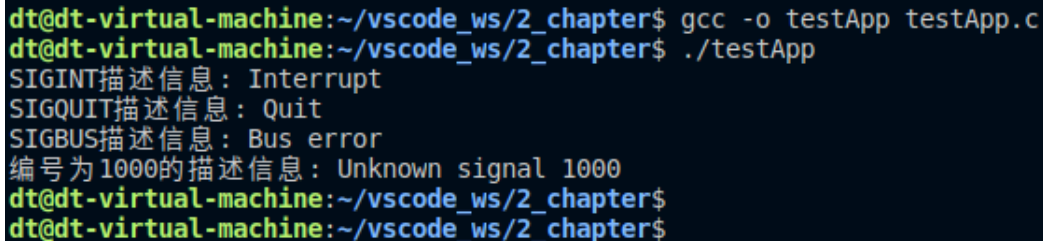
示例代码 8.8.2 strsignal() 函数使用示例

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    printf("SIGINT 描述信息: %s\n", strsignal(SIGINT));
}
```

```
printf("SIGQUIT 描述信息: %s\n", strsignal(SIGQUIT));
printf("SIGBUS 描述信息: %s\n", strsignal(SIGBUS));
printf("编号为 1000 的描述信息: %s\n", strsignal(1000));
exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
SIGINT描述信息: Interrupt
SIGQUIT描述信息: Quit
SIGBUS描述信息: Bus error
编号为1000的描述信息: Unknown signal 1000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.8.2 测试结果

8.8.2 psignal()函数

psignal()可以在标准错误 (stderr) 上输出信号描述信息, 其函数原型如下所示:

```
#include <signal.h>
```

```
void psignal(int sig, const char *s);
```

调用 psignal()函数会将参数 sig 指定的信号对应的描述信息输出到标准错误, 并且还允许调用者添加一些输出信息, 由参数 s 指定; 所以整个输出信息由字符串 s、冒号、空格、描述信号编号 sig 的字符串和尾随的换行符组成。

使用示例

示例代码 8.8.3 psignal()函数使用示例

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    psignal(SIGINT, "SIGINT 信号描述信息");
    psignal(SIGQUIT, "SIGQUIT 信号描述信息");
    psignal(SIGBUS, "SIGBUS 信号描述信息");
    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
SIGINT信号描述信息: Interrupt
SIGQUIT信号描述信息: Quit
SIGBUS信号描述信息: Bus error
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.8.3 运行结果

8.9 信号掩码(阻塞信号传递)

内核为每一个进程维护了一个信号掩码(其实就是一个信号集),即一组信号。当进程接收到一个属于信号掩码中定义的信号时,该信号将会被阻塞、无法传递给进程进行处理,那么内核会将其阻塞,直到该信号从信号掩码中移除,内核才会将该信号传递给进程从而得到处理。

向信号掩码中添加一个信号,通常有如下几种方式:

- 当应用程序调用 `signal()`或 `sigaction()`函数为某一个信号设置处理方式时,进程会自动将该信号添加到信号掩码中,这样保证了在处理一个给定的信号时,如果此信号再次发生,那么它将会被阻塞;当然对于 `sigaction()`而言,是否会如此,需要根据 `sigaction()`函数是否设置了 `SA_NODEFER` 标志而定;当信号处理函数结束返回后,会自动将该信号从信号掩码中移除。
- 使用 `sigaction()`函数为信号设置处理方式时,可以额外指定一组信号,当调用信号处理函数时将该组信号自动添加到信号掩码中,当信号处理函数结束返回后,再将这组信号从信号掩码中移除;通过 `sa_mask` 参数进行设置,参考 8.4.2 小节内容。
- 除了以上两种方式之外,还可以使用 `sigprocmask()`系统调用,随时可以显式地向信号掩码中添加/移除信号。

`sigprocmask()`函数原型如下所示:

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下:

how: 参数 `how` 指定了调用函数时的一些行为。

set: 将参数 `set` 指向的信号集内的所有信号添加到信号掩码中或者从信号掩码中移除;如果参数 `set` 为 `NULL`,则表示无需对当前信号掩码作出改动。

oldset: 如果参数 `oldset` 不为 `NULL`,在向信号掩码中添加新的信号之前,获取到进程当前的信号掩码,存放在 `oldset` 所指定的信号集中;如果为 `NULL` 则表示不获取当前的信号掩码。

返回值: 成功返回 0;失败将返回-1,并设置 `errno`。

参数 `how` 可以设置为以下宏:

- `SIG_BLOCK`: 将参数 `set` 所指向的信号集内的所有信号添加到进程的信号掩码中。换言之,将信号掩码设置为当前值与 `set` 的并集。
- `SIG_UNBLOCK`: 将参数 `set` 指向的信号集内的所有信号从进程信号掩码中移除。
- `SIG_SETMASK`: 进程信号掩码直接设置为参数 `set` 指向的信号集。

使用示例

将信号 `SIGINT` 添加到进程的信号掩码中:


```
int ret;

/* 定义信号集 */
sigset_t sig_set;

/* 将信号集初始化为空 */
sigemptyset(&sig_set);

/* 向信号集中添加 SIGINT 信号 */
sigaddset(&sig_set, SIGINT);

/* 向进程的信号掩码中添加信号 */
ret = sigprocmask(SIG_BLOCK, &sig_set, NULL);
if (-1 == ret) {
    perror("sigprocmask error");
    exit(-1);
}
```

从信号掩码中移除 SIGINT 信号:

```
int ret;

/* 定义信号集 */
sigset_t sig_set;

/* 将信号集初始化为空 */
sigemptyset(&sig_set);

/* 向信号集中添加 SIGINT 信号 */
sigaddset(&sig_set, SIGINT);

/* 从信号掩码中移除信号 */
ret = sigprocmask(SIG_UNBLOCK, &sig_set, NULL);
if (-1 == ret) {
    perror("sigprocmask error");
    exit(-1);
}
```

下面我们编写一个简单地测试代码，验证信号掩码的作用，测试代码如下所示：

示例代码 8.9.1 测试信号掩码的作用

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void sig_handler(int sig)
```

```
{
    printf("执行信号处理函数...\n");
}

int main(void)
{
    struct sigaction sig = {0};
    sigset_t sig_set;

    /* 注册信号处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGINT, &sig, NULL))
        exit(-1);

    /* 信号集初始化 */
    sigemptyset(&sig_set);
    sigaddset(&sig_set, SIGINT);

    /* 向信号掩码中添加信号 */
    if (-1 == sigprocmask(SIG_BLOCK, &sig_set, NULL))
        exit(-1);

    /* 向自己发送信号 */
    raise(SIGINT);

    /* 休眠 2 秒 */
    sleep(2);
    printf("休眠结束\n");

    /* 从信号掩码中移除添加的信号 */
    if (-1 == sigprocmask(SIG_UNBLOCK, &sig_set, NULL))
        exit(-1);

    exit(0);
}
```

上述代码中,我们为 SIGINT 信号注册了一个处理函数 sig_handler,当进程接收到该信号之后就会执行它;然后调用 sigprocmask 函数将 SIGINT 信号添加到信号掩码中,然后再调用 raise(SIGINT)向自己发送一个 SIGINT 信号,如果信号掩码没有生效、也就意味着 SIGINT 信号不会被阻塞,那么调用 raise(SIGINT)之后应该就会立马执行 sig_handler 函数,从而打印出"执行信号处理函数..."字符串信息;如果设置的信号掩码生效了,则并不会立马执行信号处理函数,而是在 2 秒后才执行,因为程序中使用 sleep(2)休眠了 2 秒钟之后,才将 SIGINT 信号从信号掩码中移除,故而进程才会处理该信号,在移除之前接收到该信号会将其阻塞。

编译测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
休眠结束
执行信号处理函数...
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.9.1 测试结果

8.10 阻塞等待信号 sigsuspend()

上一小节已经说明,更改进程的信号掩码可以阻塞所选择的信号,或解除对它们的阻塞。使用这种技术可以保护不希望由信号中断的关键代码段。如果希望对一个信号解除阻塞后,然后调用 `pause()`以等待之前被阻塞的信号的传递,这将如何?譬如有如下代码段:

```
sigset_t new_set, old_set;

/* 信号集初始化 */
sigemptyset(&new_set);
sigaddset(&new_set, SIGINT);

/* 向信号掩码中添加信号 */
if (-1 == sigprocmask(SIG_BLOCK, &new_set, &old_set))
    exit(-1);

/* 受保护的关键代码段 */
.....
/*****/

/* 恢复信号掩码 */
if (-1 == sigprocmask(SIG_SETMASK, &old_set, NULL))
    exit(-1);

pause();/* 等待信号唤醒 */
```

执行受保护的关键代码时不希望被 `SIGINT` 信号打断,所以在执行关键代码之前将 `SIGINT` 信号添加到进程的信号掩码中,执行完毕之后再恢复之前的信号掩码。最后调用了 `pause()`阻塞等待被信号唤醒,如果此时发生了信号则会被唤醒、从 `pause` 返回继续执行;考虑到这样一种情况,如果信号的传递恰好发生在第二次调用 `sigprocmask()`之后、`pause()`之前,如果确实发生了这种情况,就会产生一个问题,信号传递过来就会导致执行信号的处理函数,而从处理函数返回后又回到主程序继续执行,从而进入到 `pause()`被阻塞,知道下一次信号发生时才会被唤醒,这有违代码的本意。

虽然信号传递发生在这个时间段的可能性并不大,但并不是完全没有可能,这必然是一个缺陷,要避免这个问题,需要将恢复信号掩码和 `pause()`挂起进程这两个动作封装成一个原子操作,这正是 `sigsuspend()`系统调用的目的所在, `sigsuspend()`函数原型如下所示:

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

使用该函数需要包含头文件 `#include <signal.h>`。

函数参数和返回值含义如下:

mask: 参数 `mask` 指向一个信号集。

返回值: `sigsuspend()` 始终返回 -1, 并设置 `errno` 来指示错误 (通常为 `EINTR`), 表示被信号所中断, 如果调用失败, 将 `errno` 设置为 `EFAULT`。

`sigsuspend()` 函数会将参数 `mask` 所指向的信号集来替换进程的信号掩码, 也就是将进程的信号掩码设置为参数 `mask` 所指向的信号集, 然后挂起进程, 直到捕获到信号被唤醒 (如果捕获的信号是 `mask` 信号集中的成员, 将不会唤醒、继续挂起)、并从信号处理函数返回, 一旦从信号处理函数返回, `sigsuspend()` 会将进程的信号掩码恢复成调用前的值。

调用 `sigsuspend()` 函数相当于以不可中断 (原子操作) 的方式执行以下操作:

```
sigprocmask(SIG_SETMASK, &mask, &old_mask);
pause();
sigprocmask(SIG_SETMASK, &old_mask, NULL);
```

使用示例

示例代码 8.10.1 `sigsuspend()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void sig_handler(int sig)
{
    printf("执行信号处理函数...\n");
}

int main(void)
{
    struct sigaction sig = {0};
    sigset_t new_mask, old_mask, wait_mask;

    /* 信号集初始化 */
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGINT);
    sigemptyset(&wait_mask);

    /* 注册信号处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGINT, &sig, NULL))
        exit(-1);

    /* 向信号掩码中添加信号 */
    if (-1 == sigprocmask(SIG_BLOCK, &new_mask, &old_mask))
```

```
        exit(-1);

    /* 执行保护代码段 */
    puts("执行保护代码段");
    /*******/

    /* 挂起、等待信号唤醒 */
    if (-1 != sigsuspend(&wait_mask))
        exit(-1);

    /* 恢复信号掩码 */
    if (-1 == sigprocmask(SIG_SETMASK, &old_mask, NULL))
        exit(-1);

    exit(0);
}
```

在上述代码中,我们希望执行受保护代码段时不被 SIGINT 中断信号打断,所以在执行保护代码段之前将 SIGINT 信号添加到进程的信号掩码中,执行完受保护的代码段之后,调用 sigsuspend()挂起进程,等待被信号唤醒,被唤醒之后再解除 SIGINT 信号的阻塞状态。

8.11 实时信号

如果进程当前正在执行信号处理函数,在处理信号期间接收到了新的信号,如果该信号是信号掩码中的成员,那么内核会将其阻塞,将该信号添加到进程的等待信号集(等待被处理,处于等待状态的信号)中,为了确定进程中处于等待状态的是哪些信号,可以使用 sigpending()函数获取。

8.11.1 sigpending()函数

其函数原型如下所示:

```
#include <signal.h>
```

```
int sigpending(sigset_t *set);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下:

set: 处于等待状态的信号会存放在参数 set 所指向的信号集中。

返回值: 成功返回 0; 失败将返回-1, 并设置 errno。

使用示例

判断 SIGINT 信号当前是否处于等待状态

```
/* 定义信号集 */
```

```
sigset_t sig_set;
```

```
/* 将信号集初始化为空 */
```

```
sigemptyset(&sig_set);
```

```
/* 获取当前处于等待状态的信号 */
sigpending(&sig_set);

/* 判断 SIGINT 信号是否处于等待状态 */
if (1 == sigismember(&sig_set, SIGINT))
    puts("SIGINT 信号处于等待状态");
else if (!sigismember(&sig_set, SIGINT))
    puts("SIGINT 信号未处于等待状态");
```

8.11.2 发送实时信号

等待信号集只是一个掩码, 仅表明一个信号是否发生, 而不能表示其发生的次数。换言之, 如果一个同一个信号在阻塞状态下产生了多次, 那么会将该信号记录在等待信号集中, 并在之后仅传递一次 (仅当做发生了一次), 这是标准信号的缺点之一。

实时信号较之于标准信号, 其优势如下:

- 实时信号的信号范围有所扩大, 可应用于应用程序自定义的目的, 而标准信号仅提供了两个信号可用于应用程序自定义使用: **SIGUSR1** 和 **SIGUSR2**。
- 内核对于实时信号所采取的是队列化管理。如果将某一实时信号多次发送给另一个进程, 那么将会多次传递此信号。相反, 对于某一标准信号正在等待某一进程, 而此时即使再次向该进程发送此信号, 信号也只会传递一次。
- 当发送一个实时信号时, 可为信号指定伴随数据 (一整形数据或者指针值), 供接收信号的进程在它的信号处理函数中获取。
- 不同实时信号的传递顺序得到保障。如果有多个不同的实时信号处于等待状态, 那么将率先传递具有最小编号的信号。换言之, 信号的编号越小, 其优先级越高, 如果是同一类型的多个信号在排队, 那么信号 (以及伴随数据) 的传递顺序与信号发送来时的顺序保持一致。

Linux 内核定义了 31 个不同的实时信号, 信号编号范围为 34~64, 使用 **SIGRTMIN** 表示编号最小的实时信号, 使用 **SIGRTMAX** 表示编号最大的实时信号, 其它信号编号可使用这两个宏加上一个整数或减去一个整数。

应用程序当中使用实时信号, 需要有以下两点要求:

- 发送进程使用 **sigqueue()** 系统调用向另一个进程发送实时信号以及伴随数据。
- 接收实时信号的进程要为该信号建立一个信号处理函数, 使用 **sigaction** 函数为信号建立处理函数, 并加入 **SA_SIGINFO**, 这样信号处理函数才能够接收到实时信号以及伴随数据, 也就是要使用 **sa_sigaction** 指针指向的处理函数, 而不是 **sa_handler**, 当然允许应用程序使用 **sa_handler**, 但这样就不能获取到实时信号的伴随数据了。

使用 **sigqueue()** 函数发送实时信号, 其函数原型如下所示:

```
#include <signal.h>
```

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

使用该函数需要包含头文件 **<signal.h>**。

函数参数和返回值含义如下:

pid: 指定接收信号的进程对应的 **pid**, 将信号发送给该进程。

sig: 指定需要发送的信号。与 **kill()** 函数一样, 也可将参数 **sig** 设置为 0, 用于检查参数 **pid** 所指定的进程是否存在。

value: 参数 **value** 指定了信号的伴随数据, **union sigval** 数据类型。

返回值: 成功将返回 0; 失败将返回-1, 并设置 errno。

union sigval 数据类型 (共用体) 如下所示:

```
typedef union sigval
{
    int sival_int;
    void *sival_ptr;
} sigval_t;
```

携带的伴随数据, 既可以指定一个整形的数据, 也可以指定一个指针。

使用示例

(1) 发送进程使用 sigqueue() 系统调用向另一个进程发送实时信号

示例代码 8.11.1 使用 sigqueue() 函数发送信号

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char *argv[])
{
    union sigval sig_val;
    int pid;
    int sig;

    /* 判断传参个数 */
    if (3 > argc)
        exit(-1);

    /* 获取用户传递的参数 */
    pid = atoi(argv[1]);
    sig = atoi(argv[2]);
    printf("pid: %d\nsignal: %d\n", pid, sig);

    /* 发送信号 */
    sig_val.sival_int = 10; // 伴随数据
    if (-1 == sigqueue(pid, sig, sig_val)) {
        perror("sigqueue error");
        exit(-1);
    }

    puts("信号发送成功!");
    exit(0);
}
```

(2) 接收进程使用 sigaction() 函数为信号绑定处理函数

示例代码 8.11.2 使用 sigaction() 函数为实时信号绑定处理函数

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void sig_handler(int sig, siginfo_t *info, void *context)
{
    sigval_t sig_val = info->si_value;

    printf("接收到实时信号: %d\n", sig);
    printf("伴随数据为: %d\n", sig_val.sival_int);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int num;

    /* 判断传参个数 */
    if (2 > argc)
        exit(-1);

    /* 获取用户传递的参数 */
    num = atoi(argv[1]);

    /* 为实时信号绑定处理函数 */
    sig.sa_sigaction = sig_handler;
    sig.sa_flags = SA_SIGINFO;
    if (-1 == sigaction(num, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    /* 死循环 */
    for (;;)
        sleep(1);

    exit(0);
}
```

8.12 异常退出 abort()函数

在 3.3 小节中给大家介绍了应用程序中结束进程的几种方法, 譬如使用 `exit()`、`_exit()`或`_Exit()`这些函数来终止进程, 然后这些方法使用于正常退出应用程序, 而对于异常退出程序, 则一般使用 `abort()`库函数, 使用 `abort()`终止进程运行, 会生成核心转储文件, 可用于判断程序调用 `abort()`时的程序状态。

abort()函数原型如下所示:

```
#include <stdlib.h>
```

```
void abort(void);
```

函数 abort()通常产生 SIGABRT 信号来终止调用该函数的进程, SIGABRT 信号的系统默认操作是终止进程运行、并生成核心转储文件;当调用 abort()函数之后,内核会向进程发送 SIGABRT 信号。

使用示例

示例代码 8.12.1 abort()终止进程

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void sig_handler(int sig)
{
    printf("接收到信号: %d\n", sig);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};

    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGABRT, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    sleep(2);
    abort(); // 调用 abort
    for (;;)
        sleep(1);

    exit(0);
}
```

运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
接收到信号: 6
已放弃 (核心已转储)
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.12.1 测试结果

从打印信息可知,即使在我们的程序当中捕获了 SIGABRT 信号,但是程序依然会无情的终止,无论阻塞或忽略 SIGABRT 信号, abort()调用均不收到影响,总会成功终止进程。

第九章 进程

本章将讨论进程相关的知识内容,虽然在前面章节内容已经多次向大家提到了进程这个概念,但并未真正地向大家解释这个概念;在本章,我们将一起来学习 Linux 下进程相关的知识内容,虽然进程的基本概念比较简单,但是其所涉及到的细节内容比较多,所以本章篇幅也会相对比较长,所以,大家加油!

本章将会讨论如下主题内容。

- 程序与进程基本概念;
- 程序的开始与结束;
- 进程的环境变量与虚拟地址空间;
- 进程 ID;
- `fork()`创建子进程;
- 进程的消亡与诞生;
- 僵尸进程与孤儿进程;
- 父进程监视子进程;
- 进程关系与进程的六种状态;
- 守护进程;
- 进程间通信概述。

9.1 进程与程序

9.1.1 main()函数由谁调用?

C 语言程序总是从 main 函数开始执行, main()函数的原型是:

```
int main(void)
```

或

```
int main(int argc, char *argv[])
```

如果需要向应用程序传参, 则选择第二种写法。不知大家是否想过“谁”调用了 main()函数? 事实上, 操作系统下的应用程序在运行 main()函数之前需要先执行一段引导代码, 最终由这段引导代码去调用应用程序的 main()函数, 我们在编写应用程序的时候, 不用考虑引导代码的问题, 在编译链接时, 由链接器将引导代码链接到我们的应用程序当中, 一起构成最终的可执行文件。

当执行应用程序时, 在 Linux 下输入可执行文件的相对路径或绝对路径就可以运行该程序, 譬如./app 或/home/dt/app, 还可根据应用程序是否接受传参在执行命令时在后面添加传入的参数信息, 譬如./app arg1 arg2 或/home/dt/app arg1 arg2。程序运行需要通过操作系统的加载器来实现, 加载器是操作系统中的程序, 当执行程序时, 加载器负责将此应用程序加载内存中去执行。

所以由此可知, 对于操作系统下的应用程序来说, 链接器和加载器都是很重要的角色!

再来看看 argc 和 argv 传参是如何实现的呢? 譬如./app arg1 arg2, 这两个参数 arg1 和 arg2 是如何传递给应用程序的 main 函数的呢? 当在终端执行程序时, 命令行参数 (command-line argument) 由 shell 进程逐一进行解析, shell 进程会将这些参数传递给加载器, 加载器加载应用程序时会将其传递给应用程序引导代码, 当引导程序调用 main()函数时, 在由它最终传递给 main()函数, 如此一来, 在我们的应用程序当中便可以获取到命令行参数了。

9.1.2 程序如何结束?

程序结束其实就是进程终止, 进程终止的方式通常有多种, 大体上分为正常终止和异常终止, 正常终止包括:

- main()函数中通过 return 语句返回来终止进程;
- 应用程序中调用 exit()函数终止进程;
- 应用程序中调用 _exit()或 _Exit()终止进程;

以上这些是在前面的课程中给大家介绍的, 异常终止包括:

- 应用程序中调用 abort()函数终止进程;
- 进程接收到一个信号, 譬如 SIGKILL 信号。

注册进程终止处理函数 atexit()

atexit()库函数用于注册一个进程在正常终止时要调用的函数, 其函数原型如下所示:

```
#include <stdlib.h>
```

```
int atexit(void (*function)(void));
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下:

function: 函数指针, 指向注册的函数, 此函数无需传入参数、无返回值。

返回值: 成功返回 0; 失败返回非 0。

测试

编写一个测试程序, 使用 `atexit()` 函数注册一个进程在正常终止时需要调用的函数, 测试代码如下。

示例代码 9.1.1 `atexit()` 函数使用示例

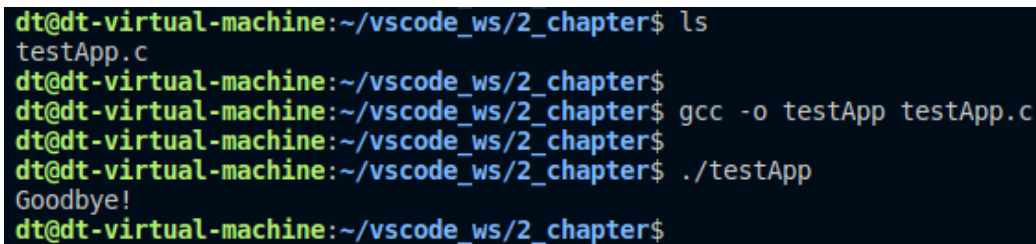
```
#include <stdio.h>
#include <stdlib.h>

static void bye(void)
{
    puts("Goodbye!");
}

int main(int argc, char *argv[])
{
    if (atexit(bye)) {
        fprintf(stderr, "cannot set exit function\n");
        exit(-1);
    }

    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Goodbye!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.1.1 测试结果

需要说明的是, 如果程序当中使用了 `_exit()` 或 `_Exit()` 终止进程而并非是 `exit()` 函数, 那么将不会执行注册的终止处理函数。

9.1.3 何为进程?

本小节正式向大家介绍进程这个概念, 前面的内容中也已经多次提到了, 其实这个概念本身非常简单, 进程其实就是一个可执行程序的实例, 这句话如何理解呢? 可执行程序就是一个可执行文件, 文件是一个静态的概念, 存放磁盘中, 如果可执行文件没有被运行, 那它将不会产生什么作用, 当它被运行之后, 它将会对系统环境产生一定的影响, 所以可执行程序的实例就是可执行文件被运行。

进程是一个动态过程, 而非静态文件, 它是程序的一次运行过程, 当应用程序被加载到内存中运行之后它就称为了一个进程, 当程序运行结束后也就意味着进程终止, 这就是进程的一个生命周期。

9.1.4 进程号

Linux 系统下的每一个进程都有一个进程号 (process ID, 简称 PID), 进程号是一个正数, 用于唯一标识系统中的某一个进程。在 Ubuntu 系统下执行 `ps` 命令可以查到系统中进程相关的一些信息, 包括每个进程的进程号, 如下所示:

```
dt@dt-virtual-machine:~$ ps -aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	185416	4196	?	Ss	3月09	0:27	/sbin/init splash
root	2	0.0	0.0	0	0	?	S	3月09	0:00	[kthreadd]
root	4	0.0	0.0	0	0	?	I<	3月09	0:00	[kworker/0:0H]
root	6	0.0	0.0	0	0	?	I<	3月09	0:00	[mm_percpu_wq]
root	7	0.0	0.0	0	0	?	S	3月09	0:00	[ksoftirqd/0]
root	8	0.0	0.0	0	0	?	I	3月09	10:22	[rcu_sched]
root	9	0.0	0.0	0	0	?	I	3月09	0:00	[rcu_bh]
root	10	0.0	0.0	0	0	?	S	3月09	0:00	[migration/0]
root	11	0.0	0.0	0	0	?	S	3月09	0:04	[watchdog/0]
root	12	0.0	0.0	0	0	?	S	3月09	0:00	[cpuhp/0]
root	13	0.0	0.0	0	0	?	S	3月09	0:00	[cpuhp/1]
root	14	0.0	0.0	0	0	?	S	3月09	0:04	[watchdog/1]
root	15	0.0	0.0	0	0	?	S	3月09	0:00	[migration/1]
root	16	0.0	0.0	0	0	?	S	3月09	0:00	[ksoftirqd/1]
root	18	0.0	0.0	0	0	?	I<	3月09	0:00	[kworker/1:0H]
root	19	0.0	0.0	0	0	?	S	3月09	0:00	[cpuhp/2]
root	20	0.0	0.0	0	0	?	S	3月09	0:03	[watchdog/2]
root	21	0.0	0.0	0	0	?	S	3月09	0:00	[migration/2]
root	22	0.0	0.0	0	0	?	S	3月09	0:01	[ksoftirqd/2]
root	24	0.0	0.0	0	0	?	I<	3月09	0:00	[kworker/2:0H]
root	25	0.0	0.0	0	0	?	S	3月09	0:00	[cpuhp/3]
root	26	0.0	0.0	0	0	?	S	3月09	0:03	[watchdog/3]
root	27	0.0	0.0	0	0	?	S	3月09	0:00	[migration/3]
root	28	0.0	0.0	0	0	?	S	3月09	0:00	[ksoftirqd/3]

图 9.1.2 ps 命令查看进程信息

上图中红框标识显示的便是每个进程所对应的进程号，进程号的作用就是用于唯一标识系统中某一个进程，在某些系统调用中，进程号可以作为传入参数、有时也可作为返回值。譬如系统调用 `kill()` 允许调用者向某一个进程发送一个信号，如何表示这个进程呢？则是通过进程号进行标识。

在应用程序中，可通过系统调用 `getpid()` 来获取本进程的进程号，其函数原型如下所示：

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

使用该函数需要包含头文件 `<sys/types.h>` 和 `<unistd.h>`。

函数返回值为 `pid_t` 类型变量，便是对应的进程号。

使用示例

使用 `getpid()` 函数获取进程的进程号。

示例代码 9.1.2 getpid() 使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
int main(void)
```

```
{
```

```
    pid_t pid = getpid();
```

```
    printf("本进程的 PID 为: %d\n", pid);
```

```
    exit(0);
```

```
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本进程的PID为: 46136
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.1.3 测试结果

除了 `getpid()` 用于获取本进程的进程号之外, 还可以使用 `getppid()` 系统调用获取父进程的进程号, 其函数原型如下所示:

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t getppid(void);
```

返回值对应的便是父进程的进程号。

使用示例

获取进程的进程号和父进程的进程号。

示例代码 9.1.3 `getppid()` 使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t pid = getpid(); //获取本进程 pid
    printf("本进程的 PID 为: %d\n", pid);

    pid = getppid(); //获取父进程 pid
    printf("父进程的 PID 为: %d\n", pid);

    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本进程的PID为: 46306
父进程的PID为: 3304
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.1.4 测试结果

9.2 进程的环境变量

每一个进程都有一组与其相关的环境变量, 这些环境变量以字符串形式存储在一个字符串数组列表中, 把这个数组称为环境列表。其中每个字符串都是以“名称=值 (name=value)”形式定义, 所以环境变量是“名称-值”的成对集合, 譬如在 shell 终端下可以使用 env 命令查看到 shell 进程的所有环境变量, 如下所示:

```
dt@dt-virtual-machine:~$ env
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dt
GPG_AGENT_INFO=/home/dt/.gnupg/S.gpg-agent:0:1
SHELL=/bin/bash
TERM=xterm-256color
VTE_VERSION=4205
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=58720266
OLDPWD=/proc
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1911
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
JRE_HOME=/opt/jdk1.8.0_271/jre
USER=dt
LS_COLORS=rs=0:di=01:34:ln=01:36:mh=00:pi=40:33:so=01:35:do=01:35:bd=40:33:01:cd=40:33:01:or=40:31:01:mi=00:su=37:41:sg=30:43:ca=30:41:tw=30:42:
*.tgz=01:31:*.arc=01:31:*.arj=01:31:*.taz=01:31:*.lha=01:31:*.lz4=01:31:*.lzh=01:31:*.lzma=01:31:*.tlz=01:31:*.txz=01:31:*.tzo=01:31:*.t7z=01:31:
l:31:*.gz=01:31:*.lrz=01:31:*.lzo=01:31:*.xz=01:31:*.bz2=01:31:*.bz=01:31:*.tbz=01:31:*.tbz2=01:31:*.taz=01:31:*.deb=01:31:*.rpm=01:31:
sar=01:31:*.rar=01:31:*.alz=01:31:*.ace=01:31:*.zoo=01:31:*.cpio=01:31:*.7z=01:31:*.rz=01:31:*.cab=01:31:*.jpg=01:35:*.jpeg=01:35:*.gif=01:35:*.
=01:35:*.tga=01:35:*.xbm=01:35:*.xpm=01:35:*.tif=01:35:*.tiff=01:35:*.png=01:35:*.svg=01:35:*.svgz=01:35:*.mng=01:35:*.pcx=01:35:*.mov=01:35:*.m
=01:35:*.webm=01:35:*.ogm=01:35:*.mp4=01:35:*.m4v=01:35:*.mp4v=01:35:*.vob=01:35:*.qt=01:35:*.nuv=01:35:*.wmv=01:35:*.asf=01:35:*.rm=01:35:*.rmv
l:35:*.flv=01:35:*.gl=01:35:*.dl=01:35:*.xcf=01:35:*.xwd=01:35:*.yuv=01:35:*.cgm=01:35:*.emf=01:35:*.ogv=01:35:*.ogx=01:35:*.aac=00:36:*.au=00:3
*.midi=00:36:*.mka=00:36:*.mp3=00:36:*.mpc=00:36:*.ogg=00:36:*.ra=00:36:*.wav=00:36:*.oga=00:36:*.opus=00:36:*.spx=00:36:*.xspf=00:36:
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/opt/jdk1.8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/g
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=fcitx
QT_QPA_PLATFORMTHEME=appmenu-qt5
XDG_SESSION_TYPE=x11
PWD=/home/dt
JOB=gnome-session
XMODIFIERS=@im=fcitx
GNOME_KEYRING_PID=
LANG=zh_CN.UTF-8
GDM_LANG=zh_CN
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
TZ=:Asia/Harbin
```

图 9.2.1 env 命令查看环境变量

使用 export 命令还可以添加一个新的环境变量或删除一个环境变量:

```
export LINUX_APP=123456 # 添加 LINUX_APP 环境变量
```



```

dt@dt-virtual-machine:~$ export LINUX_APP=123456
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ env
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dt
GPG_AGENT_INFO=/home/dt/.gnupg/S.gpg-agent:0:1
SHELL=/bin/bash
TERM=xterm-256color
VTE_VERSION=4205
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=58720266
OLDPWD=/proc
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1911
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
JRE_HOME=/opt/jdk1.8.0_271/jre
USER=dt
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;37;01:su=01;35;00:st=02;35:ex=01;32:*.tar=01;31;*.tgz=01;31;*.arc=01;31;*.arj=01;31;*.taz=01;31;*.lha=01;31;*.lz4=01;31;*.lzh=01;31;*.lzma=01;31;*.lzo=01;31;*.xz=01;31;*.bz2=01;31;*.bz=01;31;*.tbz=01;31;*.rar=01;31;*.alz=01;31;*.ace=01;31;*.zoo=01;31;*.cpio=01;31;*.7z=01;31;*.rz=01;31;*.rpm=01;35;*.tga=01;35;*.xbm=01;35;*.xpm=01;35;*.tif=01;35;*.tiff=01;35;*.png=01;35;*.svg=01;35;*.s=01;35;*.webm=01;35;*.ogm=01;35;*.mp4=01;35;*.m4v=01;35;*.mp4v=01;35;*.vob=01;35;*.qt=01;35;*.n1;35;*.flv=01;35;*.gl=01;35;*.dl=01;35;*.xcf=01;35;*.xwd=01;35;*.yuv=01;35;*.cgm=01;35;*.emf=01;35;*.mid=00;36;*.mka=00;36;*.mp3=00;36;*.mpc=00;36;*.ogg=00;36;*.ra=00;36;*.wav=00;36;*.oga=00;36;*.opus=00;36
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DEFAULTS_PATH=/usr/share/acconf/ubuntu.default.path
LINUX_APP=123456
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg

```

图 9.2.2 export 添加环境变量

使用"export -n LINUX_APP"命令则可以删除 LINUX_APP 环境变量。

```
export -n LINUX_APP          # 删除 LINUX_APP 环境变量
```

9.2.1 应用程序中获取环境变量

在我们的应用程序当中也可以获取当前进程的环境变量，事实上，进程的环境变量是从其父进程中继承过来的，譬如在 shell 终端下执行一个应用程序，那么该进程的环境变量就是从其父进程（shell 进程）中继承过来的。新的进程在创建之前，会继承其父进程的环境变量副本。

环境变量存放在一个字符串数组中，在应用程序中，通过 environ 变量指向它，environ 是一个全局变量，在我们的应用程序中只需申明它即可使用，如下所示：

```
extern char **environ;      // 申明外部全局变量 environ
```

测试

编写应用程序，获取进程的所有环境变量。

示例代码 9.2.1 获取进程环境变量

```

#include <stdio.h>
#include <stdlib.h>

extern char **environ;

int main(int argc, char *argv[])
{
    int i;

    /* 打印进程的环境变量 */
    for (i = 0; NULL != environ[i]; i++)

```

```
puts(envIRON[i]);

exit(0);
}
```

通过字符串数组元素是否等于 NULL 来判断是否已经到了数组的末尾。

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
XDG_VTNR=7
XDG_SESSION_ID=c2
TERM_PROGRAM=vscode
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dt
GIO_LAUNCHED_DESKTOP_FILE_PID=3179
SESSION=ubuntu
GPG_AGENT_INFO=/home/dt/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
TERM_PROGRAM_VERSION=1.52.1
OLDPWD=/home/dt/vscode_ws
ORIGINAL_XDG_CURRENT_DESKTOP=Unity
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1911
GTK_MODULES=gail:atk-bridge:unity-gtk-module
JRE_HOME=/opt/jdk1.8.0_271/jre
USER=dt
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:z=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.ogg=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flac=01;35:*.d1=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m3u=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/dt-virtual-machine:@/tmp/.ICE-unix/2131,unix/dt-virtual-machine:/tmp/.ICE-unix/2131
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
GIO_LAUNCHED_DESKTOP_FILE=/home/dt/桌面/code.desktop
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
```

图 9.2.3 测试结果

获取指定环境变量 `getenv()`

如果只想要获取某个指定的环境变量,可以使用库函数 `getenv()`,其函数原型如下所示:

```
#include <stdlib.h>
```

```
char *getenv(const char *name);
```

使用该函数需要包含头文件 `<stdlib.h>`。

函数参数和返回值含义如下:

name: 指定获取的环境变量名称。

返回值: 如果存放该环境变量,则返回该环境变量的值对应字符串的指针;如果不存在该环境变量,则返回 NULL。

使用 `getenv()` 需要注意,不应该去修改其返回的字符串,修改该字符串意味着修改了环境变量对应的值, Linux 提供了相应的修改函数,如果需要修改环境变量的值应该使用这些函数,不应直接改动该字符串。

使用示例

示例代码 9.2.2 `getenv()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
    const char *str_val = NULL;

    if (2 > argc) {
        fprintf(stderr, "Error: 请传入环境变量名称\n");
        exit(-1);
    }

    /* 获取环境变量 */
    str_val = getenv(argv[1]);
    if (NULL == str_val) {
        fprintf(stderr, "Error: 不存在[%s]环境变量\n", argv[1]);
        exit(-1);
    }

    /* 打印环境变量的值 */
    printf("环境变量的值: %s\n", str_val);
    exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp PATH
环境变量的值: /opt/jdk1.8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/opt/jdk1.8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.2.4 测试结果

9.2.2 添加/删除/修改环境变量

C 语言函数库中提供了用于修改、添加、删除环境变量的函数, 譬如 `putenv()`、`setenv()`、`unsetenv()`、`clearenv()` 函数等。

putenv() 函数

`putenv()` 函数可向进程的环境变量数组中添加一个新的环境变量, 或者修改一个已经存在的环境变量对应的值, 其函数原型如下所示:

```
#include <stdlib.h>
```

```
int putenv(char *string);
```

使用该函数需要包含头文件 `<stdlib.h>`。

函数参数和返回值含义如下:

string: 参数 `string` 是一个字符串指针, 指向 `name=value` 形式的字符串。

返回值: 成功返回 0; 失败将返回非 0 值, 并设置 `errno`。

该函数调用成功之后, 参数 `string` 所指向的字符串就成为了进程环境变量的一部分了, 换言之, `putenv()` 函数将设定 `environ` 变量 (字符串数组) 中的某个元素 (字符串指针) 指向该 `string` 字符串, 而不是指向它的复制副本, 这里需要注意! 因此, 不能随意修改参数 `string` 所指向的内容, 这将影响进程的环境变量, 出于这种原因, 参数 `string` 不应为自动变量 (即在栈中分配的字符串数组)。

测试

使用 `putenv()` 函数为当前进程添加一个环境变量。

示例代码 9.2.3 `putenv()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    if (2 > argc) {
        fprintf(stderr, "Error: 传入 name=value\n");
        exit(-1);
    }

    /* 添加/修改环境变量 */
    if (putenv(argv[1])) {
        perror("putenv error");
        exit(-1);
    }

    exit(0);
}
```

`setenv()` 函数

`setenv()` 函数可以替代 `putenv()` 函数, 用于向进程的环境变量列表中添加一个新的环境变量或修改现有环境变量对应的值, 其函数原型如下所示:

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int overwrite);
```

使用该函数需要包含头文件 `<stdlib.h>`。

函数参数和返回值含义如下:

name: 需要添加或修改的环境变量名称。

value: 环境变量的值。

overwrite: 若参数 `name` 标识的环境变量已经存在, 在参数 `overwrite` 为 0 的情况下, `setenv()` 函数将不改变现有环境变量的值, 也就是说本次调用没有产生任何影响; 如果参数 `overwrite` 的值为非 0, 若参数 `name` 标识的环境变量已经存在, 则覆盖, 不存在则表示添加新的环境变量。

返回值: 成功返回 0; 失败将返回 -1, 并设置 `errno`。

`setenv()` 函数为形如 `name=value` 的字符串分配一块内存缓冲区, 并将参数 `name` 和参数 `value` 所指向的字符串复制到此缓冲区中, 以此来创建一个新的环境变量, 所以, 由此可知, `setenv()` 与 `putenv()` 函数有两个区别:

- putenv()函数并不会为 name=value 字符串分配内存;
- setenv()可通过参数 overwrite 控制是否需要修改现有变量的值而仅以添加变量为目的,显然 putenv()并不能进行控制。

推荐大家使用 setenv()函数, 这样使用自动变量作为 setenv()的参数也不会有问题。

使用示例

示例代码 9.2.4 setenv()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    if (3 > argc) {
        fprintf(stderr, "Error: 传入 name value\n");
        exit(-1);
    }

    /* 添加环境变量 */
    if (setenv(argv[1], argv[2], 0)) {
        perror("setenv error");
        exit(-1);
    }

    exit(0);
}
```

除了上面给大家介绍的函数之外, 我们还可以通过一种更简单地方式向进程环境变量表中添加环境变量, 用法如下:

```
NAME=value ./app
```

在执行程序的时候, 在其路径前面添加环境变量, 以 name=value 的形式添加, 如果是多个环境变量, 则在 ./app 前面放置多对 name=value 即可, 以空格分隔。

unsetenv()函数

unsetenv()函数可以从环境变量表中移除参数 name 标识的环境变量, 其函数原型如下所示:

```
#include <stdlib.h>

int unsetenv(const char *name);
```

9.2.3 清空环境变量

有时, 需要清除环境变量表中的所有变量, 然后再进行重建, 可以通过将全局变量 environ 赋值为 NULL 来清空所有变量。

```
environ = NULL;
```

也可通过 clearenv()函数来操作, 函数原型如下所示:

```
#include <stdlib.h>
```

```
int clearenv(void);
```

clearenv()函数内部的做法其实就是将environ赋值为NULL。在某些情况下,使用setenv()函数和clearenv()函数可能会导致程序内存泄漏,前面提到过, setenv()函数会为环境变量分配一块内存缓冲区,随之称为进程的一部分;而调用clearenv()函数时没有释放该缓冲区(clearenv()调用并不知晓该缓冲区的存在,故而也无法将其释放),反复调用者两个函数的程序,会不断产生内存泄漏。

9.2.4 环境变量的作用

环境变量常见的用途之一是在shell中,每一个环境变量都有它所表示的含义,譬如HOME环境变量表示用户的家目录,USER环境变量表示当前用户名,SHELL环境变量表示shell解析器名称,PWD环境变量表示当前所在目录等,在我们自己的应用程序当中,也可以使用进程的环境变量。

9.3 进程的内存布局

历史沿袭至今,C语言程序一直都是由以下几部分组成的:

- **正文段**。也可称为代码段,这是CPU执行的机器语言指令部分,文本段具有只读属性,以防止程序由于意外而修改其指令;正文段是可以共享的,即使在多个进程间也可同时运行同一段程序。
- **初始化数据段**。通常将此段称为数据段,包含了显式初始化的全局变量和静态变量,当程序加载到内存中时,从可执行文件中读取这些变量的值。
- **未初始化数据段**。包含了未进行显式初始化的全局变量和静态变量,通常将此段称为bss段,这一名词来源于早期汇编程序中的一个操作符,意思是“由符号开始的块”(block started by symbol),在程序开始执行之前,系统会将本段内所有内存初始化为0,可执行文件并没有为bss段变量分配存储空间,在可执行文件中只需记录bss段的位置及其所需大小,直到程序运行时,由加载器来分配这一段内存空间。
- **栈**。函数内的局部变量以及每次函数调用时所需保存的信息都放在此段中,每次调用函数时,函数传递的实参以及函数返回值等也都存放在栈中。栈是一个动态增长和收缩的段,由栈帧组成,系统会为每个当前调用的函数分配一个栈帧,栈帧中存储了函数的局部变量(所谓自动变量)、实参和返回值。
- **堆**。可在运行时动态进行内存分配的一块区域,譬如使用malloc()分配的内存空间,就是从系统堆内存中申请分配的。

Linux下的size命令可以查看二进制可执行文件的文本段、数据段、bss段的段大小:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ size testApp
text  data  bss  dec  hex filename
1453  560   16  2029  7ed testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.3.1 size 命令

图 9.3.2 显示了这些段在内存中的典型布局方式,当然,并不要求具体的实现一定是以这种方式安排其存储空间,但这是一种便于我们说明的典型方式。

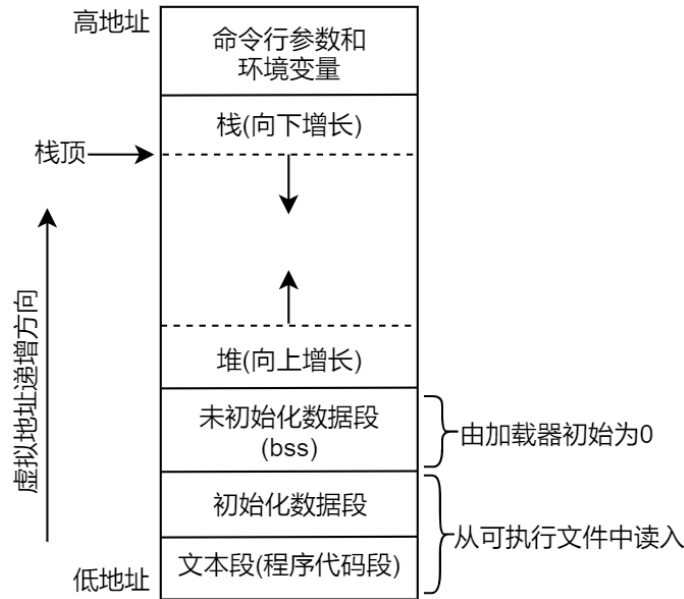


图 9.3.2 在 Linux/x86-32 体系中进程内存布局

9.4 进程的虚拟地址空间

上一小节我们讨论了 C 语言程序的构成以及运行时进程在内存中的布局方式，在 Linux 系统中，采用了虚拟内存管理技术，事实上大多数现在操作系统都是如此！在 Linux 系统中，每一个进程都在自己独立的地址空间中运行，在 32 位系统中，每个进程的逻辑地址空间均为 4GB，这 4GB 的内存空间按照 3:1 的比例进行分配，其中用户进程享有 3G 的空间，而内核独自享有剩下的 1G 空间，如下所示：

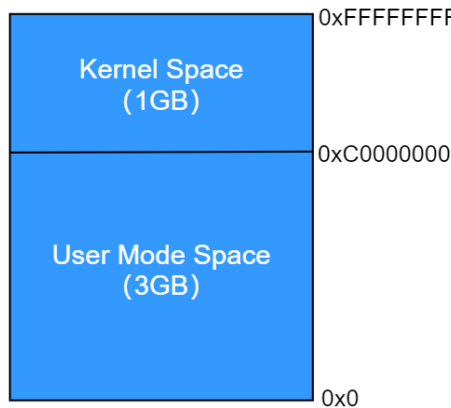


图 9.4.1 Linux 系统下逻辑地址空间划分

学习过驱动开发的读者对“虚拟地址”这个概念应该并不陌生，虚拟地址会通过硬件 MMU（内存管理单元）映射到实际的物理地址空间中，建立虚拟地址到物理地址的映射关系后，对虚拟地址的读写操作实际上就是对物理地址的读写操作，MMU 会将物理地址“翻译”为对应的物理地址，其关系如下所示：



图 9.4.2 虚拟地址到物理地址的映射关系

Linux 系统下, 应用程序运行在一个虚拟地址空间中, 所以程序中读写的内存地址对应也是虚拟地址, 并不是真正的物理地址, 譬如应用程序中读写 0x80800000 这个地址, 实际上并不对应于硬件的 0x80800000 这个物理地址。

为什么需要引入虚拟地址呢?

计算机物理内存的大小是固定的, 就是计算机的实际物理内存, 试想一下, 如果操作系统没有虚拟地址机制, 所有的应用程序访问的内存地址就是实际的物理地址, 所以要将所有应用程序加载到内存中, 但是我们实际的物理内存只有 4G, 所以就会出现一些问题:

- 当多个程序需要运行时, 必须保证这些程序用到的内存总量要小于计算机实际的物理内存的大小。
- **内存使用效率低。**内存空间不足时, 就需要将其它程序暂时拷贝到硬盘中, 然后将新的程序装入内存。然而由于大量的数据装入装出, 内存的使用效率就会非常低。
- **进程地址空间不隔离。**由于程序是直接访问物理内存的, 所以每一个进程都可以修改其它进程的内存数据, 甚至修改内核地址空间中的数据, 所以有些恶意程序可以随意修改别的进程, 就会造成一些破坏, 系统不安全、不稳定。
- **无法确定程序的链接地址。**程序运行时, 链接地址和运行地址必须一致, 否则程序无法运行! 因为程序代码加载到内存的地址是由系统随机分配的, 是无法预知的, 所以程序的运行地址在编译程序时是无法确认的。

针对以上的一些问题, 就引入了虚拟地址机制, 程序访问存储器所使用的逻辑地址就是虚拟地址, 通过逻辑地址映射到真正的物理内存上。所有应用程序运行在自己的虚拟地址空间中, 使得进程的虚拟地址空间和物理地址空间隔离开来, 这样做带来了很多的优点:

- 进程与进程、进程与内核相互隔离。一个进程不能读取或修改另一个进程或内核的内存数据, 这是因为每一个进程的虚拟地址空间映射到了不同的物理地址空间。提高了系统的安全性与稳定性。
- 在某些应用场合下, 两个或者更多进程能够共享内存。因为每个进程都有自己的映射表, 可以让不同进程的虚拟地址空间映射到相同的物理地址空间中。通常, 共享内存可用于实现进程间通信。
- 便于实现内存保护机制。譬如在多个进程共享内存时, 允许每个进程对内存采取不同的保护措施, 例如, 一个进程可能以只读方式访问内存, 而另一进程则能够以可读可写的方式访问。
- 编译应用程序时, 无需关心链接地址。前面提到了, 当程序运行时, 要求链接地址与运行地址一致, 在引入了虚拟地址机制后, 便无需关心这个问题。

关于本小节的内容就介绍这么多, 理解本小节的内容可以帮助我们更好地理解后面小节中将要介绍的内容。

9.5 fork() 创建子进程

一个现有的进程可以调用 `fork()` 函数创建一个新的进程, 调用 `fork()` 函数的进程称为父进程, 由 `fork()` 函数创建出来的进程被称为子进程 (child process), `fork()` 函数原型如下所示 (`fork()` 为系统调用):

```
#include <unistd.h>
```

```
pid_t fork(void);
```

使用该函数需要包含头文件 `<unistd.h>`。

在诸多的应用中, 创建多个进程是任务分解时行之有效的方法, 譬如, 某一网络服务器进程可在监听客户端请求的同时, 为处理每一个请求事件而创建一个新的子进程, 与此同时, 服务器进程会继续监听更多的客户端连接请求。在一个大型的应用程序任务中, 创建子进程通常会简化应用程序的设计, 同时提高了系统的并发性 (即同时能够处理更多的任务或请求, 多个进程在宏观上实现同时运行)。

理解 `fork()` 系统调用的关键在于, 完成对其调用后将存在两个进程, 一个是原进程 (父进程)、另一个则是创建出来的子进程, 并且每个进程都会从 `fork()` 函数的返回处继续执行, 会导致调用 `fork()` 返回两次值, 子进程返回一个值、父进程返回一个值。在程序代码中, 可通过返回值来区分是子进程还是父进程。

`fork()` 调用成功后, 将会在父进程中返回子进程的 PID, 而在子进程中返回值是 0; 如果调用失败, 父进程返回值 -1, 不创建子进程, 并设置 `errno`。

`fork()` 调用成功后, 子进程和父进程会继续执行 `fork()` 调用之后的指令, 子进程、父进程各自在自己的进程空间中运行。事实上, 子进程是父进程的一个副本, 譬如子进程拷贝了父进程的数据段、堆、栈以及继承了父进程打开的文件描述符, 父进程与子进程并不共享这些存储空间, 这是子进程对父进程相应部分存储空间的完全复制, 执行 `fork()` 之后, 每个进程均可修改各自的栈数据以及堆段中的变量, 而并不影响另一个进程。

虽然子进程是父进程的一个副本, 但是对于程序代码段 (文本段) 来说, 两个进程执行相同的代码段, 因为代码段是只读的, 也就是说父子进程共享代码段, 在内存中只存在一份代码段数据。

使用示例 1

使用 `fork()` 创建子进程。

示例代码 9.5.1 `fork()` 使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;

    pid = fork();
    switch (pid) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
            printf("这是子进程打印信息<pid: %d, 父进程 pid: %d>\n",
                getpid(), getppid());
            _exit(0); //子进程使用_exit()退出

        default:
            printf("这是父进程打印信息<pid: %d, 子进程 pid: %d>\n",
                getpid(), pid);
            exit(0);
    }
}
```

上述示例代码中, `case 0` 是子进程的分支, 这里使用了 `_exit()` 结束进程而没有使用 `exit()`。

Tips: C 库函数 `exit()` 建立在系统调用 `_exit()` 之上, 这两个函数在 3.3 小节中向大家介绍过, 这里我们强调, 在调用了 `fork()` 之后, 父、子进程中一般只有一个会通过调用 `exit()` 退出进程, 而另一个则应使用 `_exit()` 退出, 具体原因将会在后面章节内容中向大家做进一步说明!

直接测试运行查看打印结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
这是父进程打印信息<pid: 46802, 子进程pid: 46803>
这是子进程打印信息<pid: 46803, 父进程pid: 46802>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.5.1 测试结果

从打印结果可知, `fork()` 之后的语句被执行了两次, 所以 `switch...case` 语句被执行了两次, 第一次进入到了 "case 0" 分支, 通过上面的介绍可知, `fork()` 返回值为 0 表示当前处于子进程; 在子进程中我们通过 `getpid()` 获取到子进程自己的 PID (46802), 通过 `getppid()` 获取到父进程的 PID (46803), 将其打印出来。

第二次进入到了 `default` 分支, 表示当前处于父进程, 此时 `fork()` 函数的返回值便是创建出来的子进程对应的 PID。

`fork()` 函数调用完成之后, 父进程、子进程会各自继续执行 `fork()` 之后的指令, 最终父进程会执行到 `exit()` 结束进程, 而子进程则会通过 `_exit()` 结束进程。

使用示例 2

示例代码 9.5.2 `fork()` 函数使用示例 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    pid_t pid;

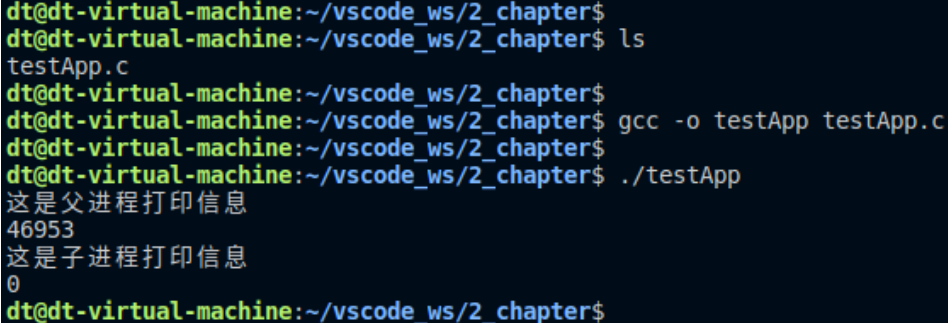
    pid = fork();
    switch (pid) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        printf("这是子进程打印信息\n");
        printf("%d\n", pid);
        _exit(0);

    default:
        printf("这是父进程打印信息\n");
```

```
    printf("%d\n", pid);
    exit(0);
}
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
这是父进程打印信息
46953
这是子进程打印信息
0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.5.2 测试结果

在 `exit()` 函数之前添加了打印信息, 而从上图中可以知道, 打印的 `pid` 值并不相同, 0 表示子进程打印出来的, 46953 表示的是父进程打印出来的, 所以从这里可以证实, `fork()` 函数调用完成之后, 父进程、子进程会各自继续执行 `fork()` 之后的指令, 它们共享代码段, 但并不共享数据段、堆、栈等, 而是子进程拥有父进程数据段、堆、栈等副本, 所以对于同一个局部变量, 它们打印出来的值是不相同的, 因为 `fork()` 调用返回值不同, 在父、子进程中赋予了 `pid` 不同的值。

关于子进程

子进程被创建出来之后, 便是一个独立的进程, 拥有自己独立的进程空间, 系统内唯一的进程号, 拥有自己独立的 PCB (进程控制块), 子进程会被内核同等调度执行, 参与到系统的进程调度中。

子进程与父进程之间的这种关系被称为父子进程关系, 父子进程关系相比于普通的进程间关系多多少少存在一些关联与“羁绊”, 关于这些关联与“羁绊”我们将会在后面的课程中为大家介绍。

Tips: 系统调度。Linux 系统是一个多任务、多进程、多线程的操作系统, 一般来说系统启动之后会运行成百甚至上千个不同的进程, 那么对于单核 CPU 计算机来说, 在某一个时间它只能运行某一个进程的代码指令, 那其它进程怎么办呢 (多核处理器也是如此, 同一时间每个核它只能运行某一个进程的代码)? 这里就出现了调度的问题, 系统是这样做的, 每一个进程 (或线程) 执行一段固定的时间, 时间到了之后切换执行下一个进程或线程, 依次轮流执行, 这就称为调度, 由操作系统负责这件事情, 当然系统调度的实现本身是一件非常复杂的事情, 需要考虑的因素很多, 这里只是让大家有个简单地认识, 系统调度的基本单元是线程, 关于线程, 后面章节内容将会向大家介绍。

9.6 父、子进程间的文件共享

调用 `fork()` 函数之后, 子进程会获得父进程所有文件描述符的副本, 这些副本的创建方式类似于 `dup()`, 这也意味着父、子进程对应的文件描述符均指向相同的文件表, 如下图所示:

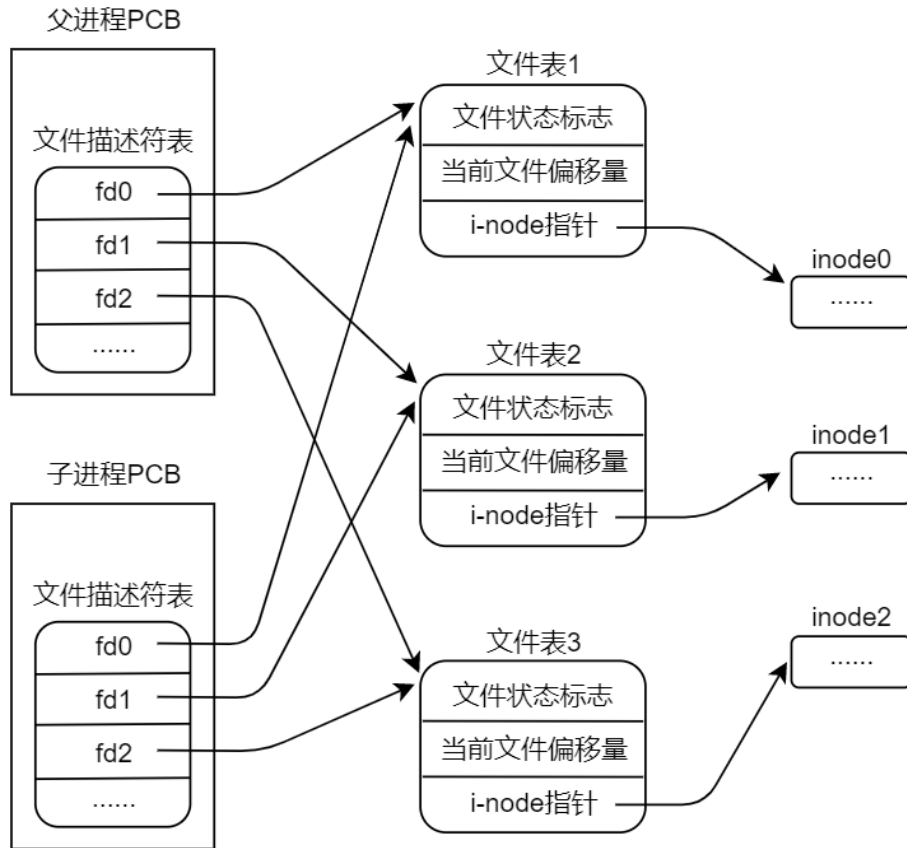


图 9.6.1 父、子进程间的文件共享

由此可知,子进程拷贝了父进程的文件描述符表,使得父、子进程中对应的文件描述符指向了相同的文件表,也意味着父、子进程中对应的文件描述符指向了磁盘中相同的文件,因而这些文件在父、子进程间实现了共享,譬如,如果子进程更新了文件偏移量,那么这个改变也会影响到父进程中相应文件描述符的位置偏移量。

接下来我们进行一个测试,父进程打开文件之后,然后 `fork()` 创建子进程,此时子进程继承了父进程打开的文件描述符(父进程文件描述符的副本),然后父、子进程同时对文件进行写入操作,测试代码如下所示:

示例代码 9.6.1 子进程继承父进程文件描述符实现文件共享

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int main(void)
{
    pid_t pid;
    int fd;
    int i;
```

```

fd = open("./test.txt", O_RDWR | O_TRUNC);
if (0 > fd) {
    perror("open error");
    exit(-1);
}

pid = fork();
switch (pid) {
case -1:
    perror("fork error");
    close(fd);
    exit(-1);

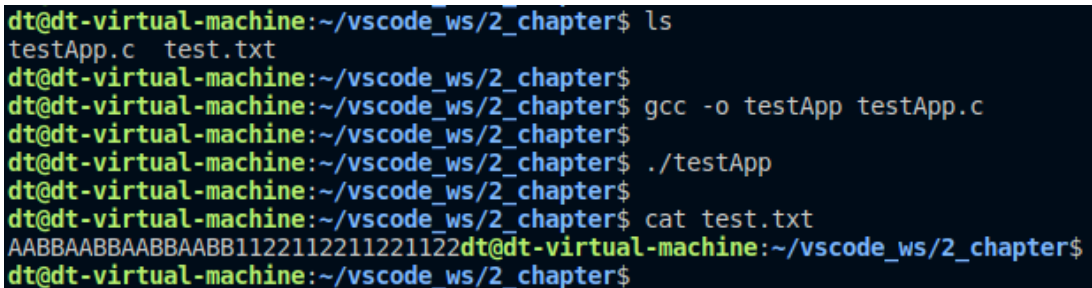
case 0:
    /* 子进程 */
    for (i = 0; i < 4; i++) //循环写入 4 次
        write(fd, "1122", 4);
    close(fd);
    _exit(0);

default:
    /* 父进程 */
    for (i = 0; i < 4; i++) //循环写入 4 次
        write(fd, "AABB", 4);
    close(fd);
    exit(0);
}
}

```

上述代码中，父进程 `open` 打开文件之后，才调用 `fork()` 创建了子进程，所以子进程继承了父进程打开的文件描述符 `fd`，我们需要验证的便是两个进程对文件的写入操作是分别各自写入、还是每次都在文件末尾接续写入。

运行测试：



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test.txt
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test.txt
AABB
AABB
AABB
AABB
1122
1122
1122
1122
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.6.2 测试结果

有上述测试结果可知，此种情况下，父、子进程分别对同一个文件进行写入操作，结果是接续写，不管是父进程，还是子进程，在每次写入时都是从文件的末尾写入，很像使用了 `O_APPEND` 标志的效果。其原因也非常简单，图 9.6.1 中便给出了答案，子进程继承了父进程的文件描述符，两个文件描述符都指向了一

个相同的文件表,意味着它们的文件偏移量是同一个、绑定在了一起,相互影响,子进程改变了文件的位置偏移量就会作用到父进程,同理,父进程改变了文件的位置偏移量就会作用到子进程。

再来测试另外一种情况,父进程在调用 fork()之后,此时父进程和子进程都去打开同一个文件,然后再对文件进行写入操作,测试代码如下:

示例代码 9.6.2 父、子各自打开同一个文件实现文件共享

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
    pid_t pid;
    int fd;
    int i;

    pid = fork();
    switch (pid) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
            /* 子进程 */
            fd = open("./test.txt", O_WRONLY);
            if (0 > fd) {
                perror("open error");
                _exit(-1);
            }

            for (i = 0; i < 4; i++) //循环写入 4 次
                write(fd, "1122", 4);
            close(fd);
            _exit(0);

        default:
            /* 父进程 */
            fd = open("./test.txt", O_WRONLY);
            if (0 > fd) {
                perror("open error");
                exit(-1);
            }
    }
}
```

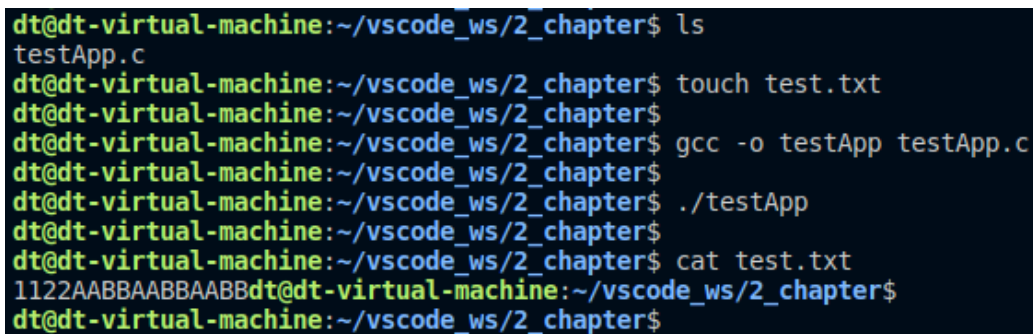
```

    }

    for (i = 0; i < 4; i++) //循环写入 4 次
        write(fd, "AABB", 4);
    close(fd);
    exit(0);
}
}

```

在上述示例中，父进程调用 `fork()` 之后，然后在父、子进程中都去打开 `test.txt` 文件，然后在对其进行写入操作，子进程调用了 4 次 `write`、每次写入“1122”；而父进程调用了 4 次 `write`、每次写入“AABB”，测试结果如下：



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ touch test.txt
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test.txt
1122AABBAABBAABBdt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.6.3 测试结果

从测试结果可知，这种文件共享方式实现的是一种两个进程分别各自对文件进行写入操作，因为父、子进程的这两个文件描述符分别指向的是不同的文件表，意味着它们有各自的文件偏移量，一个进程修改了文件偏移量并不会影响另一个进程的文件偏移量，所以写入的数据会出现覆盖的情况。

fork()函数使用场景

`fork()` 函数有以下两种用法：

- 父进程希望子进程复制自己，使父进程和子进程同时执行不同的代码段。这在网络服务进程中是常见的，父进程等待客户端的服务请求，当接收到客户端发送的请求事件后，调用 `fork()` 创建一个子进程，使子进程去处理此请求、而父进程可以继续等待下一个服务请求。
- 一个进程要执行不同的程序。譬如在程序 `app1` 中调用 `fork()` 函数创建了子进程，此时子进程是要去执行另一个程序 `app2`，也就是子进程需要执行的代码是 `app2` 程序对应的代码，子进程将从 `app2` 程序的 `main` 函数开始运行。这种情况，通常在子进程从 `fork()` 函数返回之后立即调用 `exec` 族函数来实现，关于 `exec` 函数将在后面内容向大家介绍。

9.7 系统调用 vfork()

除了 `fork()` 系统调用之外，Linux 系统还提供了 `vfork()` 系统调用用于创建子进程，`vfork()` 与 `fork()` 函数在功能上是相同的，并且返回值也相同，在一些细节上存在区别，`vfork()` 函数原型如下所示：

```

#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);

```

使用该函数需要包含头文件 `<sys/types.h>` 和 `<unistd.h>`。

从前面的介绍可知,可以将 `fork()` 认作对父进程的数据段、堆段、栈段以及其它一些数据结构创建拷贝,由此可以看出,使用 `fork()` 系统调用的代价是很大的,它复制了父进程中的数据段和堆栈段中的绝大部分内容,这将会消耗比较多的时间,效率会有所降低,而且太浪费,原因有很多,其中之一在于, `fork()` 函数之后子进程通常会调用 `exec` 函数,也就是 `fork()` 第二种使用场景下,这使得子进程不再执行父程序中的代码段,而是执行新程序的代码段,从新程序的 `main` 函数开始执行、并为新程序重新初始化其数据段、堆段、栈段等;那么在这种情况下,子进程并不需要用到父进程的数据段、堆段、栈段(譬如父程序中定义的局部变量、全局变量等)中的数据,此时就会导致浪费时间、效率降低。

事实上,现代 Linux 系统采用了一些技术来避免这种浪费,其中很重要的一点就是内核采用了写时复制 (`copy-on-write`) 技术,关于这种技术的实现细节就不给大家介绍了,有兴趣读者可以自己搜索相应的文档了解。

出于这一原因,引入了 `vfork()` 系统调用,虽然在一些细节上有所不同,但其效率要高于 `fork()` 函数。类似于 `fork()`, `vfork()` 可以为调用该函数的进程创建一个新的子进程,然而, `vfork()` 是为子进程立即执行 `exec()` 新的程序而专门设计的,也就是 `fork()` 函数的第二个使用场景。

`vfork()` 与 `fork()` 函数主要有以下两个区别:

- `vfork()` 与 `fork()` 一样都创建了子进程,但 `vfork()` 函数并不会将父进程的地址空间完全复制到子进程中,因为子进程会立即调用 `exec` (或 `_exit`),于是也就不会引用该地址空间的数据。不过在子进程调用 `exec` 或 `_exit` 之前,它在父进程的空间中运行、子进程共享父进程的内存。这种优化工作方式的实现提高效率;但如果子进程修改了父进程的数据(除了 `vfork` 返回值的变量)、进行了函数调用、或者没有调用 `exec` 或 `_exit` 就返回将可能带来未知的结果。
- 另一个区别在于, `vfork()` 保证子进程先运行,子进程调用 `exec` 之后父进程才可能被调度运行。

虽然 `vfork()` 系统调用在效率上要优于 `fork()`,但是 `vfork()` 可能会导致一些难以察觉的程序 bug,所以尽量避免使用 `vfork()` 来创建子进程,虽然 `fork()` 在效率上并没有 `vfork()` 高,但是现代的 Linux 系统内核已经采用了写时复制技术来实现 `fork()`,其效率较之于早期的 `fork()` 实现要高出许多,除非速度绝对重要的场合,我们的程序当中应舍弃 `vfork()` 而使用 `fork()`。

使用示例

示例代码 9.7.1 `vfork()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
    pid_t pid;
    int num = 100;

    pid = vfork();
    switch (pid) {
    case -1:
        perror("vfork error");
        exit(-1);

    case 0:
```



```

/* 子进程 */
printf("子进程打印信息\n");
printf("子进程打印 num: %d\n", num);
_exit(0);

```

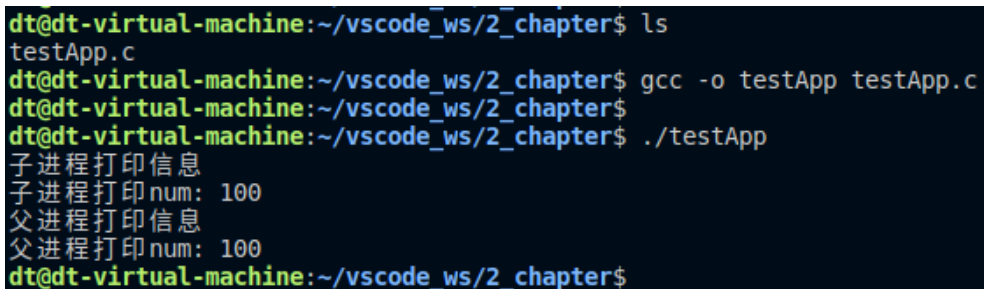
default:

```

/* 父进程 */
printf("父进程打印信息\n");
printf("父进程打印 num: %d\n", num);
exit(0);
}
}

```

测试结果:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程打印信息
子进程打印 num: 100
父进程打印信息
父进程打印 num: 100
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.7.1 测试结果

在正式的使用场合下,一般应在子进程中立即调用 `exec`, 如果 `exec` 调用失败,子进程则应调用 `_exit()` 退出 (`vfork` 产生的子进程不应调用 `exit` 退出,因为这会导致对父进程 `stdio` 缓冲区的刷新和关闭)。上述示例代码只是一个简单地演示,并不是 `vfork()` 的真正用法,后面学习到 `exec` 的时候还会再给大家进行介绍。

9.8 fork() 之后的竞争条件

调用 `fork()` 之后,子进程成为了一个独立的进程,可被系统调度运行,而父进程也继续被系统调度运行,这里出现了一个问题,调用 `fork` 之后,无法确定父、子两个进程谁将率先访问 CPU,也就是说无法确认谁先被系统调用运行(在多核处理器中,它们可能会同时各自访问一个 CPU),这将导致谁先运行、谁后运行这个顺序是不确定的,譬如有如下示例代码:

示例代码 9.8.1 fork() 竞争条件测试代码

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:

```

```

/* 子进程 */
printf("子进程打印信息\n");
_exit(0);

default:
/* 父进程 */
printf("父进程打印信息\n");
exit(0);
}
}

```

示例代码中我们是无法确认"子进程打印信息"和"父进程打印信息"谁先会被打印出来,有时子进程先被执行,打印出"子进程打印信息",而有时父进程会先被执行,打印出"子进程打印信息",测试结果如下所示:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
父进程打印信息
子进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
父进程打印信息
子进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
父进程打印信息
子进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
父进程打印信息
子进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
父进程打印信息
子进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程打印信息
父进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
父进程打印信息
子进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.8.1 测试结果

从测试结果可知,虽然绝大部分情况下,父进程会先于子进程被执行,但是并不排除子进程先于父进程被执行的可能性。而对于有些特定的应用程序,它对于执行的顺序有一定要求的,譬如它必须要求父进程先运行,或者必须要求子进程先运行,程序产生正确的结果它依赖于特定的执行顺序,那么将可能因竞争条件而导致失败、无法得到正确的结果。

那如何明确保证某一特性执行顺序呢?这个时候可以通过采用采用某种同步技术来实现,譬如前面给大家介绍的信号,如果要让子进程先运行,则可使父进程被阻塞,等到子进程来唤醒它,示例代码如下所示:

示例代码 9.8.2 利用信号来调整进程间动作

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

static void sig_handler(int sig)
{

```

```
    printf("接收到信号\n");
}

int main(void)
{
    struct sigaction sig = {0};
    sigset_t wait_mask;

    /* 初始化信号集 */
    sigemptyset(&wait_mask);

    /* 设置信号处理方式 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGUSR1, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        printf("子进程开始执行\n");
        printf("子进程打印信息\n");
        printf("~~~~~\n");
        sleep(2);
        kill(getppid(), SIGUSR1); //发送信号给父进程、唤醒它
        _exit(0);

    default:
        /* 父进程 */
        if (-1 != sigsuspend(&wait_mask))//挂起、阻塞
            exit(-1);

        printf("父进程开始执行\n");
        printf("父进程打印信息\n");
        exit(0);
    }
}
```

示例代码比较简单, 这里我们希望子进程先运行打印相应信息, 之后再执行父进程打印信息, 在父进程分支中, 直接调用了 `sigsuspend()` 使父进程进入挂起状态, 由子进程通过 `kill` 命令发送信号唤醒, 测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程开始执行
子进程打印信息
~~~~~
接收到信号
父进程开始执行
父进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.8.2 测试结果

9.9 进程的诞生与终止

9.9.1 进程的诞生

一个进程可以通过 `fork()` 或 `vfork()` 等系统调用创建一个子进程, 一个新的进程就此诞生! 事实上, Linux 系统下的所有进程都是由其父进程创建而来, 譬如在 shell 终端通过命令的方式执行一个程序 `./app`, 那么 `app` 进程就是由 shell 终端进程创建出来的, shell 终端就是该进程的父进程。

既然所有进程都是由其父进程创建出来的, 那么总有一个最原始的父进程吧, 否则其它进程是怎么创建出来的呢? 确实如此, 在 Ubuntu 系统下使用 "`ps -aux`" 命令可以查看到系统下所有进程信息, 如下:

```
dt@dt-virtual-machine:~$ ps -aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root             1  0.0  0.1 185416  4664 ?        Ss   3月09   0:29 /sbin/init splash
root             2  0.0  0.0      0     0 ?        S    3月09   0:00 [kthreadd]
root             4  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/0:0H]
root             6  0.0  0.0      0     0 ?        I<   3月09   0:00 [mm_percpu_wq]
root             7  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/0]
root             8  0.0  0.0      0     0 ?        I    3月09  11:19 [rcu_sched]
root             9  0.0  0.0      0     0 ?        I    3月09   0:00 [rcu_bh]
root            10  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/0]
root            11  0.0  0.0      0     0 ?        S    3月09   0:05 [watchdog/0]
root            12  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/0]
root            13  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/1]
root            14  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/1]
root            15  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/1]
root            16  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/1]
root            18  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/1:0H]
root            19  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/2]
root            20  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/2]
root            21  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/2]
root            22  0.0  0.0      0     0 ?        S    3月09   0:01 [ksoftirqd/2]
root            24  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/2:0H]
root            25  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/3]
root            26  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/3]
```

图 9.9.1 查看所有进程信息

上图中进程号为 1 的进程便是所有进程的父进程, 通常称为 `init` 进程, 它是 Linux 系统启动之后运行的第一个进程, 它管理着系统上所有其它进程, `init` 进程是由内核启动, 因此理论上说它没有父进程。

`init` 进程的 PID 总是为 1, 它是所有子进程的父进程, 一切从 1 开始、一切从 `init` 进程开始!

一个进程的生命周期便是从创建开始直至其终止。

9.9.2 进程的终止

通常, 进程有两种终止方式: 异常终止和正常终止, 分别在 3.3 小节和 8.12 小节中给大家介绍过, 如前所述, 进程的正常终止有多种不同的方式, 譬如在 `main` 函数中使用 `return` 返回、调用 `exit()` 函数结束进程、调用 `_exit()` 或 `_Exit()` 函数结束进程等。

异常终止通常也有多种不同的方式, 譬如在程序当中调用 `abort()` 函数异常终止进程、当进程接收到某些信号导致异常终止等。

`_exit()` 函数和 `exit()` 函数的 `status` 参数定义了进程的终止状态 (termination status), 父进程可以调用 `wait()` 函数以获取该状态。虽然参数 `status` 定义为 `int` 类型, 但仅有低 8 位表示它的终止状态, 一般来说, 终止状态为 0 表示进程成功终止, 而非 0 值则表示进程在执行过程中出现了一些错误而终止, 譬如文件打开失败、读写失败等等, 对非 0 返回值的解析并无定例。

在我们的程序当中, 一般使用 `exit()` 库函数而非 `_exit()` 系统调用, 原因在于 `exit()` 最终也会通过 `_exit()` 终止进程, 但在此之前, 它将会完成一些其它的工作, `exit()` 函数会执行的动作如下:

- 如果程序中注册了进程终止处理函数, 那么会调用终止处理函数。在 9.1.2 小节给大家介绍如何注册进程的终止处理函数;
- 刷新 `stdio` 流缓冲区。关于 `stdio` 流缓冲区的问题, 稍后编写一个简单地测试程序进行说明;
- 执行 `_exit()` 系统调用。

所以, 由此可知, `exit()` 函数会比 `_exit()` 会多做一些事情, 包括执行终止处理函数、刷新 `stdio` 流缓冲以及调用 `_exit()`, 在前面曾提到过, 在我们的程序当中, 父、子进程不应都使用 `exit()` 终止, 只能有一个进程使用 `exit()`、而另一个则使用 `_exit()` 退出, 当然一般推荐的是子进程使用 `_exit()` 退出、而父进程则使用 `exit()` 退出。其原因就在于调用 `exit()` 函数终止进程时会刷新进程的 `stdio` 缓冲区。接下来我们便通过一个示例代码进行说明:

示例代码 9.9.1 `exit()` 之 `stdio` 缓冲测试代码 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Hello World!\n");

    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        exit(0);

    default:
        /* 父进程 */
        exit(0);
    }
}
```

}

在上述代码中, 在 `fork()` 创建子进程之前, 我们通过 `printf()` 打印了一行包括换行符 `\n` 在内字符串, 在 `fork()` 创建子进程之后, 都使用 `exit()` 退出进程, 正常的情况下程序就只会打印一行 "Hello World!", 这是一个正常的情况, 事实上也确实如此, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.9.2 测试结果

打印结果确实如我们所料, 接下来将代码进行简单地修改, 把 `printf()` 打印的字符串最后面的换行符 `\n` 去掉, 如下所示:

示例代码 9.9.2 `exit()` 之 `stdio` 缓冲测试代码 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("Hello World!");

    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        exit(0);

    default:
        /* 父进程 */
        exit(0);
    }
}
```

`printf` 中将字符串后面的 `\n` 换行符给去掉了, 接下再进行测试, 结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!Hello World!dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.9.3 测试结果

从打印结果可知, "Hello World!" 被打印了两次, 这是怎么回事呢? 在程序当中明明只使用了 `printf` 打印了一次字符串。要解释这个问题, 首先要知道, 进程的用户空间内存中维护了 `stdio` 缓冲区, 0 小节给大家介绍过, 因此通过 `fork()` 创建子进程时会复制这些缓冲区。标准输出设备默认使用的是行缓冲, 当检测到换行符 `\n` 时会立即显示函数 `printf()` 输出的字符串, 在示例代码 9.9.1 中 `printf` 输出的字符串中包含了换行符, 所以会立即读走缓冲区中的数据并显示, 读走之后此时缓冲区就空了, 子进程虽然拷贝了父进程的缓冲区, 但是空的, 虽然父、子进程使用 `exit()` 退出时会刷新各自的缓冲区, 但对于空缓冲区自然无数据可读。

而对于示例代码 9.9.2 来说, `printf()` 并没有添加换行符 `\n`, 当调用 `printf()` 时并不会立即读取缓冲区中的数据进行显示, 由此 `fork()` 之后创建的子进程也自然拷贝了缓冲区的数据, 当它们调用 `exit()` 函数时, 都会刷新各自的缓冲区、显示字符串, 所以就会看到打印出了两次相同的字符串。

可以采用以下任一方法来避免重复的输出结果:

- 对于行缓冲设备, 可以加上对应换行符, 譬如 `printf` 打印输出字符串时在字符串后面添加 `\n` 换行符, 对于 `puts()` 函数来说, 本身会自动添加换行符;
- 在调用 `fork()` 之前, 使用函数 `fflush()` 来刷新 `stdio` 缓冲区, 当然, 作为另一种选择, 也可以使用 `setvbuf()` 和 `setbuf()` 来关闭 `stdio` 流的缓冲功能, 这些内容在 3.11 中已经给大家介绍过;
- 子进程调用 `_exit()` 退出进程、而非使用 `exit()`, 调用 `_exit()` 在退出时便不会刷新 `stdio` 缓冲区, 这也解释前面为什么我们要在子进程中使用 `_exit()` 退出这样做的一个原因。将示例代码 9.9.2 中子进程的退出操作 `exit()` 替换成 `_exit()` 进行测试, 打印的结果便只会显示一次字符串, 大家自己动手试一试!

关于本小节的内容, 到这里就结束了, 虽然笔者觉得自己已经介绍得很详细了, 如果大家觉得还有不懂的地方, 可以自己编写程序进行测试、验证, 编程是一门动手实践性很强的工作, 大家要善于从中发现一些问题, 然后自己能够编写程序进行测试、验证, 大家加油!

9.10 监视子进程

在很多应用程序的设计中, 父进程需要知道子进程于何时被终止, 并且需要知道子进程的终止状态信息, 是正常终止、还是异常终止亦或者被信号终止等, 意味着父进程会对子进程进行监视, 本小节我们就来学习下如何通过系统调用 `wait()` 以及其它变体来监视子进程的状态改变。

9.10.1 `wait()` 函数

对于许多需要创建子进程的进程来说, 有时设计需要监视子进程的终止时间以及终止时的一些状态信息, 在某些设计需求下这是很有必要的。系统调用 `wait()` 可以等待进程的任一子进程终止, 同时获取子进程的终止状态信息, 其函数原型如下所示:

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

使用该函数需要包含头文件 `<sys/types.h>` 和 `<sys/wait.h>`。

函数参数和返回值含义如下:

status: 参数 `status` 用于存放子进程终止时的状态信息, 参数 `status` 可以为 `NULL`, 表示不接收子进程终止时的状态信息。

返回值: 若成功则返回终止的子进程对应的进程号; 失败则返回 -1。

系统调用 `wait()` 将执行如下动作:

- 调用 `wait()` 函数, 如果其所有子进程都还在运行, 则 `wait()` 会一直阻塞等待, 直到某一个子进程终止;

- 如果进程调用 `wait()`, 但是该进程并没有子进程, 也就意味着该进程并没有需要等待的子进程, 那么 `wait()` 将返回错误, 也就是返回 -1、并且会将 `errno` 设置为 `ECHILD`。
- 如果进程调用 `wait()` 之前, 它的子进程当中已经有一个或多个子进程已经终止了, 那么调用 `wait()` 也不会阻塞。 `wait()` 函数的作用除了获取子进程的终止状态信息之外, 更重要的一点, 就是回收子进程的一些资源, 俗称为子进程“收尸”, 关于这个问题后面再给大家进行介绍。所以在调用 `wait()` 函数之前, 已经有子进程终止了, 意味着正等待着父进程为其“收尸”, 所以调用 `wait()` 将不会阻塞, 而是会立即替孩子进程“收尸”、处理它的“后事”, 然后返回到正常的程序流程中, 一次 `wait()` 调用只能处理一次。

参数 `status` 不为 `NULL` 的情况下, 则 `wait()` 会将子进程的终止时的状态信息存储在它指向的 `int` 变量中, 可以通过以下宏来检查 `status` 参数:

- **WIFEXITED(status):** 如果子进程正常终止, 则返回 `true`;
- **WEXITSTATUS(status):** 返回子进程退出状态, 是一个数值, 其实就是子进程调用 `_exit()` 或 `exit()` 时指定的退出状态; `wait()` 获取得到的 `status` 参数并不是调用 `_exit()` 或 `exit()` 时指定的状态, 可通过 `WEXITSTATUS` 宏转换;
- **WIFSIGNALED(status):** 如果子进程被信号终止, 则返回 `true`;
- **WTERMSIG(status):** 返回导致子进程终止的信号编号。如果子进程是被信号所终止, 则可以通过此宏获取终止子进程的信号;
- **WCOREDUMP(status):** 如果子进程终止时产生了核心转储文件, 则返回 `true`;

还有一些其它的宏定义, 这里就不给一一介绍了, 具体的请查看 `man` 手册。

使用示例

示例代码 9.10.1 `wait()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(void)
{
    int status;
    int ret;
    int i;

    /* 循环创建 3 个子进程 */
    for (i = 1; i <= 3; i++) {
        switch (fork()) {
            case -1:
                perror("fork error");
                exit(-1);

            case 0:
                /* 子进程 */
```



```
printf("子进程<%d>被创建\n", getpid());
sleep(i);
_exit(i);

default:
    /* 父进程 */
    break;
}
}

sleep(1);
printf("~~~~~\n");
for (i = 1; i <= 3; i++) {

    ret = wait(&status);
    if (-1 == ret) {
        if (ECHILD == errno) {
            printf("没有需要等待回收的子进程\n");
            exit(0);
        }
        else {
            perror("wait error");
            exit(-1);
        }
    }

    printf("回收子进程<%d>, 终止状态<%d>\n", ret,
           WEXITSTATUS(status));
}

exit(0);
}
```

示例代码中, 通过 for 循环创建了 3 个子进程, 父进程中循环调用 wait() 函数等待回收子进程, 并将本次回收的子进程进程号以及终止状态打印出来, 编译测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<109075>被创建
子进程<109076>被创建
子进程<109077>被创建
~~~~~
回收子进程<109075>, 终止状态<1>
回收子进程<109076>, 终止状态<2>
回收子进程<109077>, 终止状态<3>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.1 测试结果

9.10.2 waitpid()函数

使用 wait()系统调用存在着一些限制, 这些限制包括如下:

- 如果父进程创建了多个子进程, 使用 wait()将无法等待某个特定的子进程的完成, 只能按照顺序等待下一个子进程的终止, 一个一个来、谁先终止就先处理谁;
- 如果子进程没有终止, 正在运行, 那么 wait()总是保持阻塞, 有时我们希望执行非阻塞等待, 是否有子进程终止, 通过判断即可得知;
- 使用 wait()只能发现那些被终止的子进程, 对于子进程因某个信号 (譬如 SIGSTOP 信号) 而停止 (注意, 这里停止指的暂停运行), 或是已停止的子进程收到 SIGCONT 信号后恢复执行的情况就无能为力了。

而设计 waitpid()则可以突破这些限制, waitpid()系统调用函数原型如下所示:

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *status, int options);
```

使用该函数需要包含头文件<sys/types.h>和<sys/wait.h>。

函数参数和返回值含义如下:

pid: 参数 pid 用于表示需要等待的某个具体子进程, 关于参数 pid 的取值范围如下:

- 如果 pid 大于 0, 表示等待进程号为 pid 的子进程;
- 如果 pid 等于 0, 则等待与调用进程 (父进程) 同一个进程组的所有子进程;
- 如果 pid 小于 -1, 则会等待进程组标识符与 pid 绝对值相等的所有子进程;
- 如果 pid 等于 -1, 则等待任意子进程。wait(&status)与 waitpid(-1, &status, 0)等价。

status: 与 wait()函数的 status 参数意义相同。

options: 稍后介绍。

返回值: 返回值与 wait()函数的返回值意义基本相同, 在参数 options 包含了 WNOHANG 标志的情况下, 返回值会出现 0, 稍后介绍。

参数 options 是一个位掩码, 可以包括 0 个或多个如下标志:

- **WNOHANG:** 如果子进程没有发生状态改变 (终止、暂停), 则立即返回, 也就是执行非阻塞等待, 可以实现轮训 poll, 通过返回值可以判断是否有子进程发生状态改变, 若返回值等于 0 表示没有发生改变。
- **WUNTRACED:** 除了返回终止的子进程的状态信息外, 还返回因信号而停止 (暂停运行) 的子进程状态信息;

- **WCONTINUED:** 返回那些因收到 SIGCONT 信号而恢复运行的子进程的状态信息。

从以上的介绍可知, `waitpid()`在功能上要强于 `wait()`函数, 它弥补了 `wait()`函数所带来的一些限制, 具体的编程使用当中, 可根据自己的需求进行选择。

使用示例

使用 `waitpid()`替换 `wait()`, 改写示例代码 9.10.1。

示例代码 9.10.2 `waitpid()`阻塞方式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(void)
{
    int status;
    int ret;
    int i;

    /* 循环创建 3 个子进程 */
    for (i = 1; i <= 3; i++) {
        switch (fork()) {
            case -1:
                perror("fork error");
                exit(-1);

            case 0:
                /* 子进程 */
                printf("子进程<%d>被创建\n", getpid());
                sleep(i);
                _exit(i);

            default:
                /* 父进程 */
                break;
        }
    }

    sleep(1);
    printf("~~~~~\n");
    for (i = 1; i <= 3; i++) {

        ret = waitpid(-1, &status, 0);
```

```
if (-1 == ret) {
    if (ECHILD == errno) {
        printf("不需要等待回收的子进程\n");
        exit(0);
    }
    else {
        perror("wait error");
        exit(-1);
    }
}

printf("回收子进程<%d>, 终止状态<%d>\n", ret,
       WEXITSTATUS(status));
}

exit(0);
}
```

将 `wait(&status)` 替换成了 `waitpid(-1, &status, 0)`, 通过上面的介绍可知, `waitpid()` 函数的这种参数配置情况与 `wait()` 函数是完全等价的, 运行结果与示例代码 9.10.1 运行结果相同, 这里不再演示!

将上述代码进行简单修改, 将其修改成轮训方式, 如下所示:

示例代码 9.10.3 `waitpid()` 轮训方式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(void)
{
    int status;
    int ret;
    int i;

    /* 循环创建 3 个子进程 */
    for (i = 1; i <= 3; i++) {
        switch (fork()) {
            case -1:
                perror("fork error");
                exit(-1);

            case 0:
                /* 子进程 */
```

```
printf("子进程<%d>被创建\n", getpid());
sleep(i);
_exit(i);

default:
    /* 父进程 */
    break;
}
}

sleep(1);
printf("~~~~~\n");
for (;;) {

    ret = waitpid(-1, &status, WNOHANG);
    if (0 > ret) {
        if (ECHILD == errno)
            exit(0);
        else {
            perror("wait error");
            exit(-1);
        }
    }
    else if (0 == ret)
        continue;
    else
        printf("回收子进程<%d>, 终止状态<%d>\n", ret,
                WEXITSTATUS(status));
}

exit(0);
}
```

将 `waitpid()` 函数的 `options` 参数添加 `WNOHANG` 标志, 将 `waitpid()` 配置成非阻塞模式, 使用轮训的方式依次回收各个子进程, 测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<111741>被创建
子进程<111742>被创建
子进程<111743>被创建
~~~~~
回收子进程<111741>, 终止状态<1>
回收子进程<111742>, 终止状态<2>
回收子进程<111743>, 终止状态<3>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.2 测试结果

9.10.3 waitid()函数

除了以上给大家介绍的 `wait()` 和 `waitpid()` 系统调用之外, 还有一个 `waitid()` 系统调用, `waitid()` 与 `waitpid()` 类似, 不过 `waitid()` 提供了更多的扩展功能, 具体的使用方法笔者便不再介绍, 大家有兴趣可以自己通过 `man` 进行学习。

9.10.4 僵尸进程与孤儿进程

当一个进程创建子进程之后, 它们俩就成为父子进程关系, 父进程与子进程的生命周期往往是不相同的, 这里就会出现两个问题:

- 父进程先于子进程结束。
- 子进程先于父进程结束。

本小节我们就来讨论下这两种不同的情况。

孤儿进程

父进程先于子进程结束, 也就是意味着, 此时子进程变成了一个“孤儿”, 我们把这种进程就称为孤儿进程。在 Linux 系统当中, 所有的孤儿进程都自动成为 `init` 进程 (进程号为 1) 的子进程, 换言之, 某一子进程的父进程结束后, 该子进程调用 `getppid()` 将返回 1, `init` 进程变成了孤儿进程的“养父”; 这是判定某一子进程的“生父”是否还“在世”的方法之一, 通过下面的代码进行测试:

示例代码 9.10.4 孤儿进程测试

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    /* 创建子进程 */
    switch (fork()) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
```

```

/* 子进程 */
printf("子进程<%d>被创建, 父进程<%d>\n", getpid(), getppid());
sleep(3); //休眠 3 秒钟等父进程结束
printf("父进程<%d>\n", getppid());//再次获取父进程 pid
_exit(0);

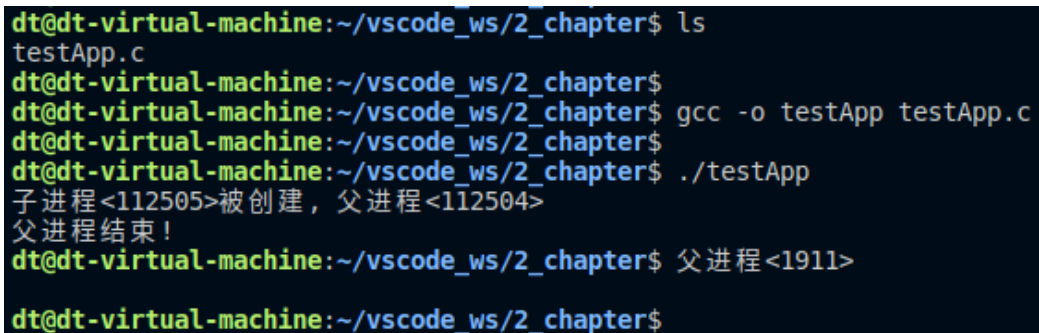
default:
/* 父进程 */
break;
}

sleep(1);//休眠 1 秒
printf("父进程结束!\n");
exit(0);
}

```

在上述代码中, 子进程休眠 3 秒钟, 保证父进程先结束, 而父进程休眠 1 秒钟, 保证子进程能够打印出第一个 `printf()`, 也就是在父进程结束前, 打印子进程的父进程进程号; 子进程 3 秒休眠时间过后, 再次打印父进程的进程号, 此时它的“生父”已经结束了。

我们来看看打印结果:



```

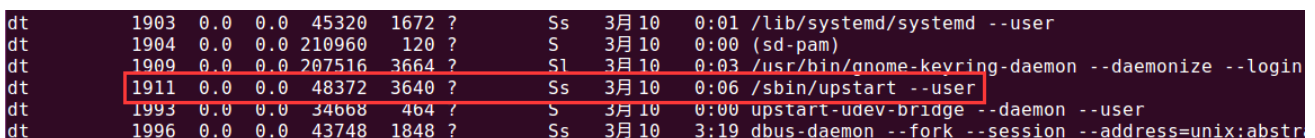
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<112505>被创建, 父进程<112504>
父进程结束!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ 父进程<1911>

dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.10.3 测试结果

可以发现, 打印结果并不是 1, 意味着并不是 `init` 进程, 而是 1911, 这是怎么回事呢? 通过“`ps -axu`”查询可知, 进程号 1911 对应的是 `upstart` 进程, 如下所示:



```

dt      1903  0.0  0.0  45320  1672 ?        Ss   3月10   0:01 /lib/systemd/systemd --user
dt      1904  0.0  0.0  210960  120 ?        S    3月10   0:00 (sd-pam)
dt      1909  0.0  0.0  207516  3664 ?        Sl   3月10   0:03 /usr/bin/gnome-keyring-daemon --daemonize --login
dt      1911  0.0  0.0  48372  3640 ?        Ss   3月10   0:06 /sbin/upstart --user
dt      1993  0.0  0.0  34668  464 ?        S    3月10   0:00 upstart-udev-bridge --daemon --user
dt      1996  0.0  0.0  43748  1848 ?        Ss   3月10   3:19 dbus-daemon --fork --session --address=unix:abstr

```

图 9.10.4 upstart 进程

事实上, `/sbin/upstart` 进程与 Ubuntu 系统图形化界面有关系, 是图形化界面下的一个后台守护进程, 可负责“收养”孤儿进程, 所以图形化界面下, `upstart` 进程就自动成为了孤儿进程的父进程, 这里笔者是在 Ubuntu 16.04 版本下进行的测试, 可能不同的版本这里看到的结果会有不同。

既然在图形化界面下孤儿进程的父进程不是 `init` 进程, 那么我们进入 Ubuntu 字符界面, 按 `Ctrl + Alt + F1` 进入, 如下所示:

```

Ubuntu 16.04.4 LTS dt-virtual-machine tty1

dt-virtual-machine login: dt
Password:
Last login: Mon Mar 29 12:21:14 CST 2021 on tty1
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.15.0-132-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

150  ◆◆◆◆◆◆◆◆◆◆
  7  ◆◆◆◆◆◆◆◆◆◆

*** ◆◆◆◆◆◆◆◆◆◆ ***
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$

```

图 9.10.5 Ubuntu 字符界面

输入 Linux 用户名和密码登录，我们在运行一次：

```

dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd vscode_ws/2_chapter/
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
◆◆◆ <112685>◆◆◆ , ◆◆◆ <112684>
◆◆◆◆◆◆◆◆◆◆ !
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ◆◆◆ <1>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.10.6 测试结果

字符界面模式下无法显示中文，所以出现了很多白色小方块，从打印结果可以发现，此时孤儿进程的父进程就成了 init 进程，大家可以自己测试下，按 Ctrl + Alt + F7 回到 Ubuntu 图形化界面。

僵尸进程

进程结束之后，通常需要其父进程为其“收尸”，回收子进程占用的一些内存资源，父进程通过调用 wait()（或其变体 waitpid()、waitid()等）函数回收子进程资源，归还给系统。

如果子进程先于父进程结束，此时父进程还未来得及给予子进程“收尸”，那么此时子进程就变成了一个僵尸进程。子进程结束后其父进程并没有来得及立马给它“收尸”，子进程处于“曝尸荒野”的状态，在这么一个状态下，我们就将子进程成为僵尸进程；至于名字由来，肯定是对电影情节的一种效仿！

当父进程调用 wait()（或其变体，下文不再强调）为子进程“收尸”后，僵尸进程就会被内核彻底删除。另外一种情况，如果父进程并没有调用 wait()函数然后就退出了，那么此时 init 进程将会接管它的子进程并自动调用 wait()，故而从系统中移除僵尸进程。

如果父进程创建了某一子进程，子进程已经结束，而父进程还在正常运行，但父进程并未调用 wait()回收子进程，此时子进程变成一个僵尸进程。首先来说，这样的程序设计是有问题的，如果系统中存在大量的僵尸进程，它们势必会填满内核进程表，从而阻碍新进程的创建。需要注意的是，僵尸进程是无法通过信号将其杀死的，即使是“一击必杀”信号 SIGKILL 也无法将其杀死，那么这种情况下，只能杀死僵尸进程的父进程（或等待其父进程终止），这样 init 进程将会接管这些僵尸进程，从而将它们从系统中清理掉！所以，在我们的一个程序设计中，一定要监视子进程的状态变化，如果子进程终止了，要调用 wait()将其回收，避免僵尸进程。

示例代码

编写示例代码, 产生一个僵尸进程。

示例代码 9.10.5 产生僵尸进程

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    /* 创建子进程 */
    switch (fork()) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
            /* 子进程 */
            printf("子进程<%d>被创建\n", getpid());
            sleep(1);
            printf("子进程结束\n");
            _exit(0);

        default:
            /* 父进程 */
            break;
    }

    for (;;)
        sleep(1);

    exit(0);
}
```

在上述代码中, 子进程已经退出, 但其父进程并没调用 `wait()` 为其“收尸”, 使得子进程成为一个僵尸进程, 使用命令“`ps -aux`”可以查看到该僵尸进程, 测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[2] 113455
子进程<113456>被创建
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ 子进程结束
root  113241  0.0  0.0      0   0 ?        I   14:54   0:00 [kworker/u256:1]
root  113307  0.0  0.0      0   0 ?        I   15:09   0:00 [kworker/u256:0]
root  113356  0.0  0.0      0   0 ?        I   15:18   0:00 [kworker/u256:2]
root  113363  0.0  0.0      0   0 ?        I   15:19   0:00 [kworker/2:2]
root  113432  0.0  0.0      0   0 ?        I   15:29   0:00 [kworker/2:0]
dt    113455  0.0  0.0    4220  644 pts/19  S   15:31   0:00 ./testApp
dt    113456  0.0  0.0      0   0 pts/19  Z   15:31   0:00 [testApp] <defunct>
dt    113459  0.0  0.0   39104 3336 pts/19  R+  15:32   0:00 ps -aux
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.7 测试结果

通过命令可以查看到子进程 113456 依然存在，可以看到它的状态栏显示的是“Z”（zombie，僵尸），表示它是一个僵尸进程。僵尸进程无法被信号杀死，大家可以试试，要么等待其父进程终止、要么杀死其父进程，让 init 进程来处理，当我们杀死其父进程之后，僵尸进程也会被随之清理。

9.10.5 SIGCHLD 信号

SIGCHLD 信号在第八章中给大家介绍过，当发生以下两种情况时，父进程会收到该信号：

- 当父进程的某个子进程终止时，父进程会收到 SIGCHLD 信号；
- 当父进程的某个子进程因收到信号而停止（暂停运行）或恢复时，内核也可能向父进程发送该信号。

子进程的终止属于异步事件，父进程事先是无法预知的，如果父进程有自己需要做的事情，它不能一直 wait() 阻塞等待子进程终止（或轮训），这样父进程将啥事也做不了，那么有什么办法来解决这样的尴尬情况，当然有办法，那就是通过 SIGCHLD 信号。

那既然子进程状态改变时（终止、暂停或恢复），父进程会收到 SIGCHLD 信号，SIGCHLD 信号的系统默认处理方式是将之忽略，所以我们要捕获它、绑定信号处理函数，在信号处理函数中调用 wait() 收回子进程，回收完毕之后再回到父进程自己的工作流中。

不过，使用这一方式时需要掌握一些窍门！

由 8.4.1 和 8.4.2 小节介绍可知，当调用信号处理函数时，会暂时将引发调用的信号添加到进程的信号掩码中（除非 sigaction() 指定了 SA_NODEFER 标志），这样一来，当 SIGCHLD 信号处理函数正在为一个终止的子进程“收尸”时，如果相继有两个子进程终止，即使产生了两次 SIGCHLD 信号，父进程也只能捕获到一次 SIGCHLD 信号，结果是，父进程的 SIGCHLD 信号处理函数每次只调用一次 wait()，那么就会导致有些僵尸进程成为“漏网之鱼”。

解决方案就是：在 SIGCHLD 信号处理函数中循环以非阻塞方式来调用 waitpid()，直至再无其它终止的子进程需要处理为止，所以，通常 SIGCHLD 信号处理函数内部代码如下所示：

```
while (waitpid(-1, NULL, WNOHANG) > 0)
    continue;
```

上述代码一直循环下去，直至 waitpid() 返回 0，表明再无僵尸进程存在；或者返回 -1，表明有错误发生。应在创建任何子进程之前，为 SIGCHLD 信号绑定处理函数。

使用示例

通过 SIGCHLD 信号实现异步方式监视子进程。

示例代码 9.10.6 异步方式监视 wait 回收子进程

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

static void wait_child(int sig)
{
    /* 替子进程收尸 */
    printf("父进程回收子进程\n");
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
}

int main(void)
{
    struct sigaction sig = {0};

    /* 为 SIGCHLD 信号绑定处理函数 */
    sigemptyset(&sig.sa_mask);
    sig.sa_handler = wait_child;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGCHLD, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    /* 创建子进程 */
    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        printf("子进程<%d>被创建\n", getpid());
        sleep(1);
        printf("子进程结束\n");
        _exit(0);
    }
}
```

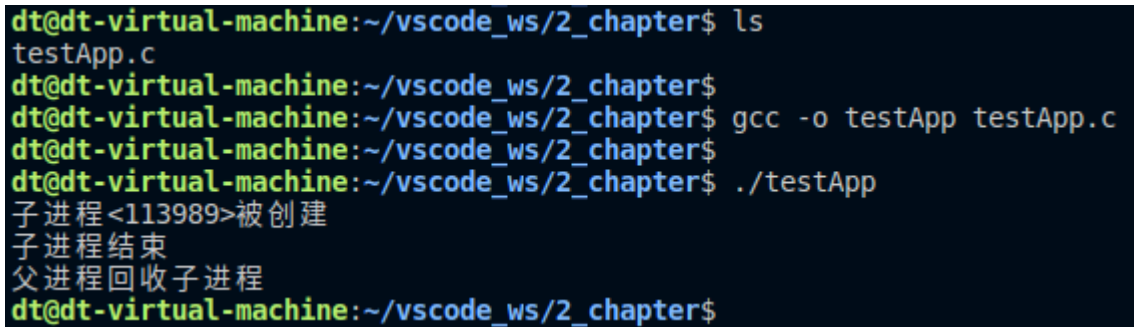
```

default:
    /* 父进程 */
    break;
}

sleep(3);
exit(0);
}

```

运行结果如下:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<113989>被创建
子进程结束
父进程回收子进程
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 9.10.8 测试结果

9.11 执行新程序

在前面已经大家提到了 `exec` 函数, 当子进程的工作不再是运行父进程的代码段, 而是运行另一个新程序的代码, 那么这个时候子进程可以通过 `exec` 函数来实现运行另一个新的程序。本小节我们就来学习下, 如何在程序中运行一个新的程序, 从新程序的 `main()` 函数开始运行。

9.11.1 `execve()` 函数

系统调用 `execve()` 可以将新程序加载到某一进程的内存空间, 通过调用 `execve()` 函数将一个外部的可执行文件加载到进程的内存空间运行, 使用新的程序替换旧的程序, 而进程的栈、数据、以及堆数据会被新程序的相应部件所替换, 然后从新程序的 `main()` 函数开始执行。

`execve()` 函数原型如下所示:

```
#include <unistd.h>
```

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

使用该函数需要包含头文件 `<unistd.h>`。

函数参数和返回值含义如下:

filename: 参数 `filename` 指向需要载入当前进程空间的新程序的路径名, 既可以是绝对路径、也可以是相对路径。

argv: 参数 `argv` 则指定了传递给新程序的命令行参数。是一个字符串数组, 该数组对应于 `main(int argc, char *argv[])` 函数的第二个参数 `argv`, 且格式也与之相同, 是由字符串指针所组成的数组, 以 `NULL` 结束。 `argv[0]` 对应的便是新程序自身路径名。

envp: 参数 `envp` 也是一个字符串指针数组, 指定了新程序的环境变量列表, 参数 `envp` 其实对应于新程序的 `environ` 数组, 同样也是以 `NULL` 结束, 所指向的字符串格式为 `name=value`。

返回值: `execve` 调用成功将不会返回; 失败将返回 -1, 并设置 `errno`。

对 `execve()` 的成功调用将永不返回, 而且也无需检查它的返回值, 实际上, 一旦该函数返回, 就表明它发生了错误。

基于系统调用 `execve()`, 还提供了一系列以 `exec` 为前缀命名的库函数, 虽然函数参数各异, 当其功能相同, 通常将这些函数 (包括系统调用 `execve()`) 称为 `exec` 族函数, 所以 `exec` 函数并不是指某一个函数, 而是 `exec` 族函数, 下一小节将会向大家介绍这些库函数。

通常将调用这些 `exec` 函数加载一个外部新程序的过程称为 `exec` 操作。

使用示例

编写一个简单地程序, 在测试程序 `testApp` 当中通过 `execve()` 函数运行另一个新程序 `newApp`。

示例代码 9.11.1 `execve()` 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *arg_arr[5];
    char *env_arr[5] = {"NAME=app", "AGE=25",
                       "SEX=man", NULL};

    if (2 > argc)
        exit(-1);

    arg_arr[0] = argv[1];
    arg_arr[1] = "Hello";
    arg_arr[2] = "World";
    arg_arr[3] = NULL;
    execve(argv[1], arg_arr, env_arr);

    perror("execve error");
    exit(-1);
}
```

将上述程序编译成一个可执行文件 `testApp`。

接着编写新程序, 在新程序当中打印出环境变量和传参, 如下所示:

示例代码 9.11.2 新程序

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

int main(int argc, char *argv[])
{
    char **ep = NULL;
    int j;
```

```
for (j = 0; j < argc; j++)
    printf("argv[%d]: %s\n", j, argv[j]);

puts("env:");
for (ep = environ; *ep != NULL; ep++)
    printf("    %s\n", *ep);

exit(0);
}
```

将新程序编译成 newApp 可执行文件, 两份程序编译好之后, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
newApp testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.1 编译程序

接下来进行测试, 运行 testApp 程序, 传入一个参数, 该参数便是新程序 newApp 的可执行文件路径:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
newApp testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./newApp
argv[0]: ./newApp
argv[1]: Hello
argv[2]: World
env:
  NAME=app
  AGE=25
  SEX=man
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.2 测试结果

由上图打印结果可知, 在我们的 testApp 程序中, 成功通过 `execve()` 运行了另一个新的程序 newApp, 当 newApp 程序运行完成退出后, testApp 进程就结束了。

示例代码 9.11.1 中 `execve()` 函数的使用并不是它真正的应用场景, 通常由 `fork()` 生成的子进程对 `execve()` 的调用最为频繁, 也就是子进程执行 `exec` 操作; 示例代码 9.11.1 中的 `execve` 用法在实际的应用不常见, 这里只是给大家进行演示说明。

说到这里, 我们来分析一个问题, 为什么需要在子进程中执行新程序? 其实这个问题非常简单, 虽然可以直接在子进程分支编写子进程需要运行的代码, 但是不够灵活, 扩展性不够好, 直接将子进程需要运行的代码单独放在一个可执行文件中不是更好吗, 所以就出现了 `exec` 操作。

9.11.2 exec 库函数

`exec` 族函数包括多个不同的函数, 这些函数命名都以 `exec` 为前缀, 上一小节给大家介绍的 `execve()` 函数也属于 `exec` 族函数中的一员, 但它属于系统调用; 本小节我们介绍 `exec` 族函数中的库函数, 这些库函数都是基于系统调用 `execve()` 而实现的, 虽然参数各异、但功能相同, 包括: `execl()`、`execlp()`、`execle()`、`execv()`、`execvp()`、`execvpe()`, 它们的函数原型如下所示:

```
#include <unistd.h>
```

```
extern char **environ;
```

```
int execl(const char *path, const char *arg, ... /* (char *) NULL */);
```

```
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
```

```
int execl_e(const char *path, const char *arg, ... /* (char *) NULL, char * const envp[] */);
```

```
int execv(const char *path, char *const argv[]);
```

```
int execvp(const char *file, char *const argv[]);
```

```
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

使用这些函数需要包含头文件<unistd.h>。

接下来简单地介绍下它们之间的区别:

- execl()和 execv()都是基本的 exec 函数, 都可用于执行一个新程序, 它们之间的区别在于参数格式不同; 参数 path 意义和格式都相同, 与系统调用 execve()的 filename 参数相同, 指向新程序的路径名, 既可以是绝对路径、也可以是相对路径。execl()和 execv()不同的在于第二个参数, execv()的 argv 参数与 execve()的 argv 参数相同, 也是字符串指针数组; 而 execl()把参数列表依次排列, 使用可变参数形式传递, 本质上也是多个字符串, 以 NULL 结尾, 如下所示:

```
// execv 传参
```

```
char *arg_arr[5];
```

```
arg_arr[0] = "./newApp";
```

```
arg_arr[1] = "Hello";
```

```
arg_arr[2] = "World";
```

```
arg_arr[3] = NULL;
```

```
execv("./newApp", arg_arr);
```

```
// execl 传参
```

```
execl("./newApp", "./newApp", "Hello", "World", NULL);
```

- execlp()和 execvp()在 execl()和 execv()基础上加了一个 p, 这个 p 其实表示的是 PATH; execl()和 execv()要求提供新程序的路径名, 而 execlp()和 execvp()则允许只提供新程序文件名, 系统会在由环境变量 PATH 所指定的目录列表中寻找相应的可执行文件, 如果执行的新程序是一个 Linux 命令, 这将很有用; 当然, execlp()和 execvp()函数也兼容相对路径和绝对路径的方式。
- execl_e()和 execvpe()这两个函数在命名上加了一个 e, 这个 e 其实表示的是 environment 环境变量, 意味着这两个函数可以指定自定义的环境变量列表给新程序, 参数 envp 与系统调用 execve()的 envp 参数相同, 也是字符串指针数组, 使用方式如下所示:

```
// execvpe 传参
```

```
char *env_arr[5] = {"NAME=app", "AGE=25",  
                  "SEX=man", NULL};
```

```
char *arg_arr[5];
```

```
arg_arr[0] = "./newApp";
```

```
arg_arr[1] = "Hello";
```

```
arg_arr[2] = "World";
```

```
arg_arr[3] = NULL;
execvpe("./newApp", arg_arr, env_arr);
```

```
// execl 传参
execl("./newApp", "./newApp", "Hello", "World", NULL, env_arr);
```

给大家介绍完这些 exec 函数之后, 下面将进行实战。

9.11.3 exec 族函数使用示例

使用以上给大家介绍的 6 个 exec 库函数运行 ls 命令, 并加入参数-a 和-l。

1、execl()函数运行 ls 命令。

示例代码 9.11.3 execl 执行 ls 命令

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    execl("/bin/ls", "ls", "-a", "-l", NULL);
    perror("execl error");
    exit(-1);
}
```

2、execv()函数运行 ls 命令。

示例代码 9.11.4 execv()执行 ls 命令

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    char *arg_arr[5];

    arg_arr[0] = "ls";
    arg_arr[1] = "-a";
    arg_arr[2] = "-l";
    arg_arr[3] = NULL;
    execv("/bin/ls", arg_arr);

    perror("execv error");
    exit(-1);
}
```

3、execlp()函数运行 ls 命令。

示例代码 9.11.5 execlp() 执行 ls 命令

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    execlp("ls", "ls", "-a", "-l", NULL);
    perror("execlp error");
    exit(-1);
}
```

4、execvp()函数运行 ls 命令。

示例代码 9.11.6 execvp() 执行 ls 命令

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    char *arg_arr[5];

    arg_arr[0] = "ls";
    arg_arr[1] = "-a";
    arg_arr[2] = "-l";
    arg_arr[3] = NULL;
    execvp("ls", arg_arr);

    perror("execvp error");
    exit(-1);
}
```

5、execle()函数运行 ls 命令。

示例代码 9.11.7 execle() 执行 ls 命令

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char **environ;

int main(void)
{
    execle("/bin/ls", "ls", "-a", "-l", NULL, environ);
    perror("execle error");
}
```

```
    exit(-1);
}
```

6、execvpe()函数运行 ls 命令。

示例代码 9.11.8 execvpe() 执行 ls 命令

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

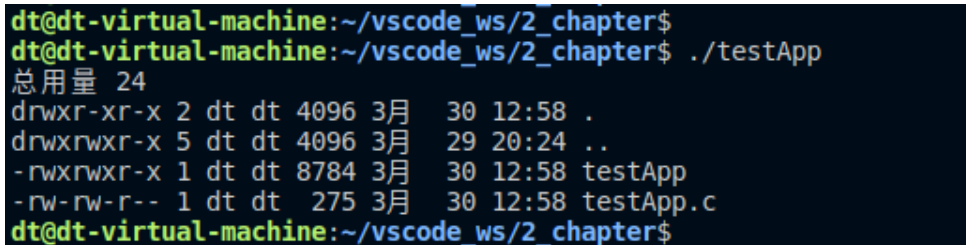
extern char **environ;

int main(void)
{
    char *arg_arr[5];

    arg_arr[0] = "ls";
    arg_arr[1] = "-a";
    arg_arr[2] = "-l";
    arg_arr[3] = NULL;
    execvpe("ls", arg_arr, environ);

    perror("execvpe error");
    exit(-1);
}
```

以上所有的这些示例代码，运行结果都是一样的，与"ls -al"命令效果相同，如下所示：



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
总用量 24
drwxr-xr-x 2 dt dt 4096 3月  30 12:58 .
drwxrwxr-x 5 dt dt 4096 3月  29 20:24 ..
-rwxrwxr-x 1 dt dt 8784 3月  30 12:58 testApp
-rw-rw-r-- 1 dt dt  275 3月  30 12:58 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.3 测试结果

9.11.4 system()函数

使用 system()函数可以很方便地在我们的程序当中执行任意 shell 命令，本小节来学习下 system()函数的用法，以及介绍 system()函数的实现方法。

首先来看看 system()函数原型，如下所示：

```
#include <stdlib.h>
```

```
int system(const char *command);
```

这是一个库函数，使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

command: 参数 `command` 指向需要执行的 shell 命令, 以字符串的形式提供, 譬如 `"ls -al"`、`"echo HelloWorld"`等。

返回值: 关于 `system()`函数的返回值有多种不同的情况, 稍后给大家介绍。

`system()`函数其内部的是通过调用 `fork()`、`execl()`以及 `waitpid()`这三个函数来实现它的功能, 首先 `system()`会调用 `fork()`创建一个子进程来运行 shell (可以把这个子进程成为 shell 进程), 并通过 shell 执行参数 `command` 所指定的命令。譬如:

```
system("ls -la")
```

```
system("echo HelloWorld")
```

`system()`的返回值如下:

- 当参数 `command` 为 `NULL`, 如果 shell 可用则返回一个非 0 值, 若不可用则返回 0; 针对一些非 UNIX 系统, 该系统上可能是没有 shell 的, 这样就会导致 shell 不可能; 如果 `command` 参数不为 `NULL`, 则返回值从以下的各种情况所决定。
- 如果无法创建子进程或无法获取子进程的终止状态, 那么 `system()`返回-1;
- 如果子进程不能执行 shell, 则 `system()`的返回值就好像是子进程通过调用 `_exit(127)`终止了;
- 如果所有的系统调用都成功, `system()`函数会返回执行 `command` 的 shell 进程的终止状态。

`system()`的主要优点在于使用上方便简单, 编程时无需自己处理对 `fork()`、`exec` 函数、`waitpid()`以及 `exit()`等调用细节, `system()`内部会代为处理; 当然这些优点通常是以牺牲效率为代价的, 使用 `system()`运行 shell 命令需要至少创建两个进程, 一个进程用于运行 shell、另外一个或多个进程则用于运行参数 `command` 中解析出来的命令, 每一个命令都会调用一次 `exec` 函数来执行; 所以从这里可以看出, 使用 `system()`函数其效率会大打折扣, 如果我们的程序对效率或速度有所要求, 那么建议大家不是直接使用 `system()`。

使用示例

以下示例代码演示了 `system()`函数的用法, 执行测试程序时, 将需要执行的命令通过参数传递给 `main()`函数, 在 `main` 函数中调用 `system()`来执行该条命令。

示例代码 9.11.9 `system()`函数使用示例

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int ret;

    if (2 > argc)
        exit(-1);

    ret = system(argv[1]);
    if (-1 == ret)
        fputs("system error.\n", stderr);
    else {
        if (WIFEXITED(ret) && (127 == WEXITSTATUS(ret)))
            fputs("could not invoke shell.\n", stderr);
    }

    exit(0);
}
```

}

运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp pwd
/home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 'ls -al'
总用量 24
drwxr-xr-x 2 dt dt 4096 3月 30 17:13 .
drwxrwxr-x 5 dt dt 4096 3月 29 20:24 ..
-rwxrwxr-x 1 dt dt 8752 3月 30 17:13 testApp
-rw-rw-r-- 1 dt dt 312 3月 30 17:11 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.4 测试结果

9.12 进程状态与进程关系

本小节来聊一聊关于进程状态与进程关系相关的话题。

9.12.1 进程状态

Linux 系统下进程通常存在 6 种不同的状态, 分为: 就绪态、运行态、僵尸态、可中断睡眠状态(浅度睡眠)、不可中断睡眠状态(深度睡眠)以及暂停态。

- 就绪态(Ready): 指该进程满足被 CPU 调度的所有条件但此时并没有被调度执行, 只要得到 CPU 就能够直接运行; 意味着该进程已经准备好被 CPU 执行, 当一个进程的时间片到达, 操作系统调度程序会从就绪态链表中调度一个进程;
- 运行态: 指该进程当前正在被 CPU 调度运行, 处于就绪态的进程得到 CPU 调度就会进入运行态;
- 僵尸态: 僵尸态进程其实指的就是僵尸进程, 指该进程已经结束、但其父进程还未给它“收尸”;
- 可中断睡眠状态: 可中断睡眠也称为浅度睡眠, 表示睡的不够“死”, 还可以被唤醒, 一般来说可以通过信号来唤醒;
- 不可中断睡眠状态: 不可中断睡眠称为深度睡眠, 深度睡眠无法被信号唤醒, 只能等待相应的条件成立才能结束睡眠状态。把浅度睡眠和深度睡眠统称为等待态(或者叫阻塞态), 表示进程处于一种等待状态, 等待某种条件成立之后便会进入到就绪态; 所以, 处于等待态的进程是无法参与进程系统调度的。
- 暂停态: 暂停并不是进程的终止, 表示进程暂停运行, 一般可通过信号将进程暂停, 譬如 SIGSTOP 信号; 处于暂停态的进程是可以恢复进入到就绪态的, 譬如收到 SIGCONT 信号。

一个新创建的进程会处于就绪态, 只要得到 CPU 就能被执行。以下列出了进程各个状态之间的转换关系, 如下所示:

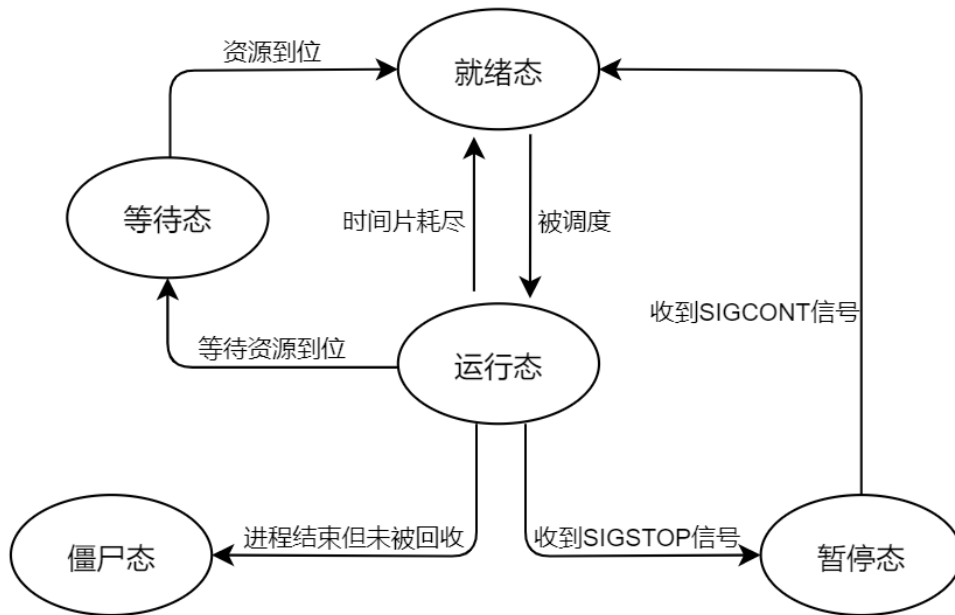


图 9.12.1 进程各状态之间的切换

9.12.2 进程关系

介绍完进程状态之后，接下来聊一聊进程关系，在 Linux 系统下，每个进程都有自己唯一的标识：进程号（进程 ID、PID），也有自己的生命周期，进程都有自己的父进程、而父进程也有父进程，这就形成了一个以 init 进程为根的进程家族树；当子进程终止时，父进程会得到通知并能取得子进程的退出状态。

除此之外，进程间还存在着其它一些层次关系，譬如进程组和会话；所以，由此可知，进程间存在着多种不同的关系，主要包括：无关系（相互独立）、父子进程关系、进程组以及会话。

1、无关系

两个进程间没有任何关系，相互独立。

2、父子进程关系

两个进程间构成父子进程关系，譬如一个进程 fork() 创建出了另一个进程，那么这两个进程间就构成了父子进程关系，调用 fork() 的进程称为父进程、而被 fork() 创建出来的进程称为子进程；当然，如果“生父”先与子进程结束，那么 init 进程（“养父”）就会成为子进程的父进程，它们之间同样也是父子进程关系。

3、进程组

每个进程除了有一个进程 ID、父进程 ID 之外，还有一个进程组 ID，用于标识该进程属于哪一个进程组，进程组是一个或多个进程的集合，这些进程并不是孤立的，它们彼此之间或者存在父子、兄弟关系，或者在功能上有联系。

Linux 系统设计进程组实质上是为了方便对进程进行管理。假设为了完成一个任务，需要并发运行 100 个进程，但当处于某种场景时需要终止这 100 个进程，若没有进程组就需要一个一个去终止，这样非常麻烦且容易出现一些问题；有了进程组的概念之后，就可以将这 100 个进程设置为一个进程组，这些进程共享一个进程组 ID，这样一来，终止这 100 个进程只需要终止该进程组即可。

关于进程组需要注意以下以下内容：

- 每个进程必定属于某一个进程组、且只能属于一个进程组；
- 每一个进程组有一个组长进程，组长进程的 ID 就等于进程组 ID；
- 在组长进程的 ID 前面加上一个负号即是操作进程组；

- 组长进程不能再创建新的进程组;
- 只要进程组中还存在一个进程, 则该进程组就存在, 这与其组长进程是否终止无关;
- 一个进程组可以包含一个或多个进程, 进程组的生命周期从被创建开始, 到其内所有进程终止或离开该进程组;
- 默认情况下, 新创建的进程会继承父进程的进程组 ID。

通过系统调用 `getpgrp()`或 `getpgid()`可以获取进程对应的进程组 ID, 其函数原型如下所示:

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
pid_t getpgrp(void);
```

首先使用该函数需要包含头文件`<unistd.h>`。

这两个函数都用于获取进程组 ID, `getpgrp()`没有参数, 返回值总是调用者进程对应的进程组 ID; 而对于 `getpgid()`函数来说, 可通过参数 `pid` 指定获取对应进程的进程组 ID, 如果参数 `pid` 为 0 表示获取调用者进程的进程组 ID。

`getpgid()`函数成功将返回进程组 ID; 失败将返回-1、并设置 `errno`。

所以由此可知, `getpgrp()`就等价于 `getpgid(0)`。

使用示例

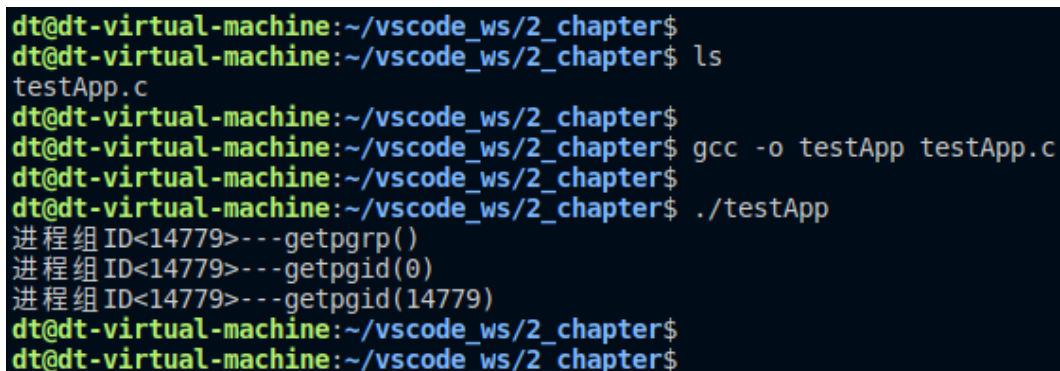
示例代码 9.12.1 获取进程组 ID

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    pid_t pid = getpid();

    printf("进程组 ID<%d>---getpgrp()\n", getpgrp());
    printf("进程组 ID<%d>---getpgid(0)\n", getpgid(0));
    printf("进程组 ID<%d>---getpgid(%d)\n", getpgid(pid), pid);
    exit(0);
}
```

测试结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
进程组 ID<14779>---getpgrp()
进程组 ID<14779>---getpgid(0)
进程组 ID<14779>---getpgid(14779)
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.12.2 测试结果

从上面的结果可以发现, 其新创建的进程对应的进程组 ID 等于该进程的 ID。

调用系统调用 `setpgid()` 或 `setpgrp()` 可以加入一个现有的进程组或创建一个新的进程组, 其函数原型如下所示:

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t ppid);
int setpgrp(void);
```

使用这些函数同样需要包含头文件 `<unistd.h>`。

`setpgid()` 函数将参数 `pid` 指定的进程的进程组 ID 设置为参数 `ppid`。如果这两个参数相等 (`pid==ppid`), 则由 `pid` 指定的进程变成为进程组的组长进程, 创建了一个新的进程组; 如果参数 `pid` 等于 0, 则使用调用者的进程 ID; 另外, 如果参数 `ppid` 等于 0, 则创建一个新的进程组, 由参数 `pid` 指定的进程作为进程组组长进程。

`setpgrp()` 函数等价于 `setpgid(0, 0)`。

一个进程只能为它自己或它的子进程设置进程组 ID, 在它的子进程调用 `exec` 函数后, 它就不能更改该子进程的进程组 ID 了。

使用示例

示例代码 9.12.2 创建进程组或加入一个现有进程组

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("更改前进程组 ID<%d>\n", getpgrp());
    setpgrp();
    printf("更改后进程组 ID<%d>\n", getpgrp());
    exit(0);
}
```

4、会话

介绍完进程组之后, 再来看下会话, 会话是一个或多个进程组的集合, 其与进程组、进程之间的关系如下图所示:

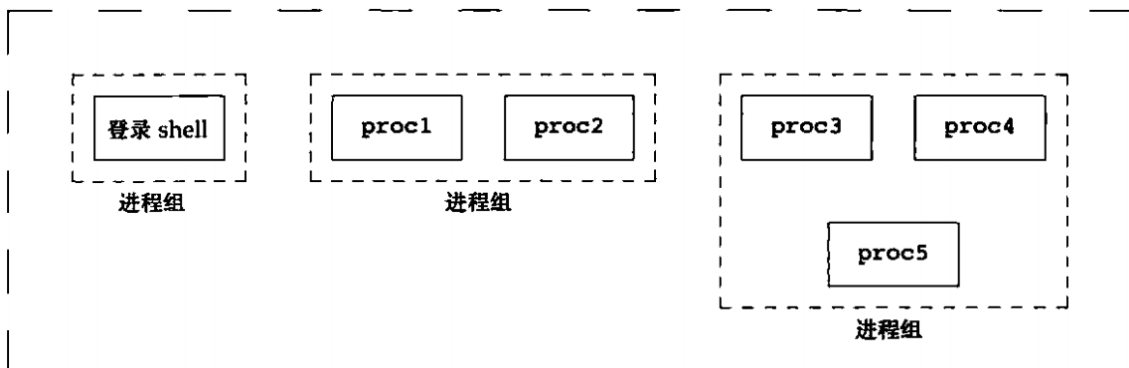


图 9.12.3 会话

一个会话可包含一个或多个进程组, 但只能有一个前台进程组, 其它的是后台进程组; 每个会话都有一个会话首领 (`leader`), 即创建会话的进程。一个会话可以有控制终端、也可没有控制终端, 在有控制终端

的情况下也只能连接一个控制终端,这通常是登录到其上的终端设备(在终端登录情况下)或伪终端设备(譬如通过 SSH 协议网络登录),一个会话中的进程组可被分为一个前台进程组以及一个或多个后台进程组。

会话的首领进程连接一个终端之后,该终端就成为会话的控制终端,与控制终端建立连接的会话首领进程被称为控制进程;产生在终端上的输入和信号将发送给会话的前台进程组中的所有进程,譬如 Ctrl+C(产生 SIGINT 信号)、Ctrl+Z(产生 SIGTSTP 信号)、Ctrl+\(产生 SIGQUIT 信号)等等这些由控制终端产生的信号。

当用户在某个终端登录时,一个新的会话就开始了;当我们在 Linux 系统下打开了多个终端窗口时,实际上就是创建了多个终端会话。

一个进程组由组长进程的 ID 标识,而对于会话来说,会话的首领进程的进程组 ID 将作为该会话的标识,也就是会话 ID(sid),在默认情况下,新创建的进程会继承父进程的会话 ID。通过系统调用 getsid() 可以获取进程的会话 ID,其函数原型如下所示:

```
#include <unistd.h>
```

```
pid_t getsid(pid_t pid);
```

使用该函数需要包含头文件<unistd.h>,如果参数 pid 为 0,则返回调用者进程的会话 ID;如果参数 pid 不为 0,则返回参数 pid 指定的进程对应的会话 ID。成功情况下,该函数返回会话 ID,失败则返回-1、并设置 errno。

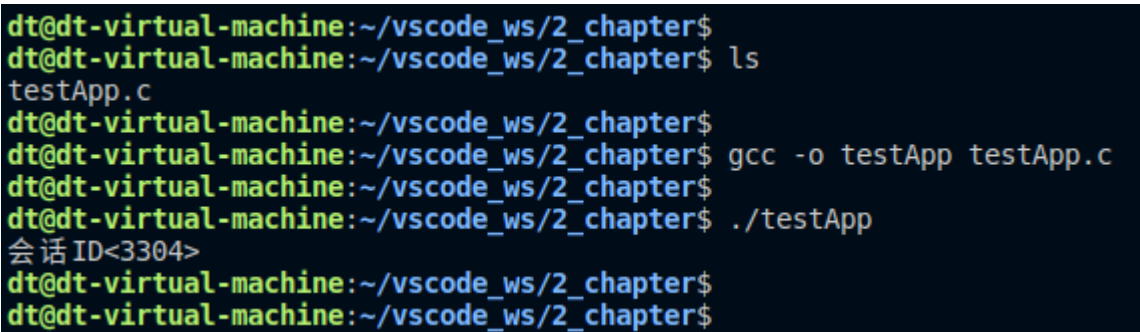
使用示例

示例代码 9.12.3 获取进程的会话 ID

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    printf("会话 ID<%d>\n", getsid(0));
    exit(0);
}
```

打印结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
会话 ID<3304>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.12.4 测试结果

使用系统调用 setsid() 可以创建一个会话,其函数原型如下所示:

```
#include <unistd.h>
```

```
pid_t setsid(void);
```


如果调用者进程不是进程组的组长进程, 调用 `setsid()` 将创建一个新的会话, 调用者进程是新会话的首领进程, 同样也是一个新的进程组的组长进程, 调用 `setsid()` 创建的会话将没有控制终端。

`setsid()` 调用成功将返回新会话的会话 ID; 失败将返回 -1, 并设置 `errno`。

9.13 守护进程

本小节学习守护进程, 将对守护进程的概念以及如何编写一个守护进程程序进行介绍。

9.13.1 何为守护进程

守护进程 (Daemon) 也称为精灵进程, 是运行在后台的一种特殊进程, 它独立于控制终端并且周期性地执行某种任务或等待处理某些事情的发生, 主要表现为以下两个特点:

- **长期运行。**守护进程是一种生存期很长的一种进程, 它们一般在系统启动时开始运行, 除非强行终止, 否则直到系统关机都会保持运行。与守护进程相比, 普通进程都是在用户登录或运行程序时创建, 在运行结束或用户注销时终止, 但守护进程不受用户登录注销的影响, 它们将会一直运行着、直到系统关机。
- **与控制终端脱离。**在 Linux 中, 系统与用户交互的界面称为终端, 每一个从终端开始运行的进程都会依附于这个终端, 这是上一小节给大家介绍的控制终端, 也就是会话的控制终端。当控制终端被关闭的时候, 该会话就会退出, 由控制终端运行的所有进程都会被终止, 这使得普通进程都是和运行该进程的终端相绑定的; 但守护进程能突破这种限制, 它脱离终端并且在后台运行, 脱离终端的目的是为了避免进程在运行的过程中的信息在终端显示并且进程也不会被任何终端所产生的信息所打断。

守护进程是一种很有用的进程。Linux 中大多数服务器就是用守护进程实现的, 譬如, Internet 服务器 `inetd`、Web 服务器 `httpd` 等。同时, 守护进程完成许多系统任务, 譬如作业规划进程 `cron` 等。

守护进程 `Daemon`, 通常简称为 `d`, 一般进程名后面带有 `d` 就表示它是一个守护进程。守护进程与终端无任何关联, 用户的登录与注销与守护进程无关、不受其影响, 守护进程自成进程组、自成会话, 即 `pid=gid=sid`。通过命令 "`ps -ajx`" 查看系统所有的进程, 如下所示:

```
dt@dt-virtual-machine:~$ ps -ajx
PPID    PID    PGID    SID    TTY          TPGID  STAT    UID    TIME  COMMAND
0        1      1        1      ?            -1    Ss      0      0:35  /sbin/init splash
0        2      0        0      ?            -1    S       0      0:01  [kthreadd]
2        4      0        0      ?            -1    I<      0      0:00  [kworker/0:0H]
2        6      0        0      ?            -1    I<      0      0:00  [mm_percpu_wq]
2        7      0        0      ?            -1    S       0      0:00  [ksoftirqd/0]
2        8      0        0      ?            -1    I       0      14:05 [rcu_sched]
2        9      0        0      ?            -1    I       0      0:00  [rcu_bh]
2       10      0        0      ?            -1    S       0      0:00  [migration/0]
2       11      0        0      ?            -1    S       0      0:06  [watchdog/0]
2       12      0        0      ?            -1    S       0      0:00  [cpuhp/0]
2       13      0        0      ?            -1    S       0      0:00  [cpuhp/1]
2       14      0        0      ?            -1    S       0      0:05  [watchdog/1]
2       15      0        0      ?            -1    S       0      0:00  [migration/1]
2       16      0        0      ?            -1    S       0      0:00  [ksoftirqd/1]
2       18      0        0      ?            -1    I<      0      0:00  [kworker/1:0H]
2       19      0        0      ?            -1    S       0      0:00  [cpuhp/2]
2       20      0        0      ?            -1    S       0      0:05  [watchdog/2]
2       21      0        0      ?            -1    S       0      0:00  [migration/2]
2       22      0        0      ?            -1    S       0      0:02  [ksoftirqd/2]
2       24      0        0      ?            -1    I<      0      0:00  [kworker/2:0H]
2       25      0        0      ?            -1    S       0      0:00  [cpuhp/3]
2       26      0        0      ?            -1    S       0      0:05  [watchdog/3]
2       27      0        0      ?            -1    S       0      0:00  [migration/3]
2       28      0        0      ?            -1    S       0      0:00  [ksoftirqd/3]
2       30      0        0      ?            -1    I<      0      0:00  [kworker/3:0H]
2       31      0        0      ?            -1    S       0      0:00  [cpuhp/4]
2       32      0        0      ?            -1    S       0      0:05  [watchdog/4]
2       33      0        0      ?            -1    S       0      0:00  [migration/4]
2       34      0        0      ?            -1    S       0      0:10  [ksoftirqd/4]
2       36      0        0      ?            -1    I<      0      0:00  [kworker/4:0H]
```

图 9.13.1 查看系统中的所有进程

TTY 一栏是问号? 表示该进程没有控制终端, 也就是守护进程, 其中 COMMAND 一栏使用中括号[]括起来的表示内核线程, 这些线程是在内核里创建, 没有用户空间代码, 因此没有程序文件名和命令行, 通常采用 k 开头的名字, 表示 Kernel。

9.13.2 编写守护进程程序

如何将自己编写的程序运行之后变成一个守护进程呢? 本小节就来学习如何编写守护进程程序, 编写守护进程一般包含如下几个步骤:

1) 创建子进程、终止父进程

父进程调用 `fork()` 创建子进程, 然后父进程使用 `exit()` 退出, 这样做实现了下面几点。第一, 如果该守护进程是作为一条简单地 shell 命令启动, 那么父进程终止会让 shell 认为这条命令已经执行完毕。第二, 虽然子进程继承了父进程的进程组 ID, 但它有自己独立的进程 ID, 这保证了子进程不是一个进程组的组长进程, 这是下面将要调用 `setsid` 函数的先决条件!

2) 子进程调用 `setsid` 创建会话

这步是关键, 在子进程中调用上一小节给大家介绍的 `setsid()` 函数创建新的会话, 由于之前子进程并不是进程组的组长进程, 所以调用 `setsid()` 会使得子进程创建一个新的会话, 子进程成为新会话的首领进程, 同样也创建了新的进程组、子进程成为组长进程, 此时创建的会话将没有控制终端。所以这里调用 `setsid` 有三个作用: 让子进程摆脱原会话的控制、让子进程摆脱原进程组的控制和让子进程摆脱原控制终端的控制。

在调用 `fork` 函数时, 子进程继承了父进程的会话、进程组、控制终端等, 虽然父进程退出了, 但原先的会话期、进程组、控制终端等并没有改变, 因此, 那还不是真正意义上使两者独立开来。`setsid` 函数能够使子进程完全独立出来, 从而脱离所有其他进程的控制。

3) 将工作目录更改为根目录

子进程是继承了父进程的当前工作目录, 由于在进程运行中, 当前目录所在的文件系统是不能卸载的, 这对以后使用会造成很多的麻烦。因此通常的做法是让 “/” 作为守护进程的当前目录, 当然也可以指定其它目录来作为守护进程的工作目录。

4) 重设文件权限掩码 umask

文件权限掩码 umask 用于对新建文件的权限位进行屏蔽, 在 5.5.5 小节中有介绍。由于使用 fork 函数新建的子进程继承了父进程的文件权限掩码, 这就给予进程使用文件带来了诸多的麻烦。因此, 把文件权限掩码设置为 0, 确保子进程有最大操作权限、这样可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是 umask, 通常的使用方法为 umask(0)。

5) 关闭不再需要的文件描述符

子进程继承了父进程的所有文件描述符, 这些被打开的文件可能永远不会被守护进程(此时守护进程指的就是子进程, 父进程退出、子进程成为守护进程)读或写, 但它们一样消耗系统资源, 可能导致所在的文件系统无法卸载, 所以必须关闭这些文件, 这使得守护进程不再持有从其父进程继承过来的任何文件描述符。

6) 将文件描述符号为 0、1、2 定位到/dev/null

将守护进程的标准输入、标准输出以及标准错误重定向到/dev/null, 这使得守护进程的输出无处显示、也无从交互式用户那里接收输入。

7) 其它: 忽略 SIGCHLD 信号

处理 SIGCHLD 信号不是必须的, 但对于某些进程, 特别是并发服务器进程往往是特别重要的, 服务器进程在接收到客户端请求时会创建子进程去处理该请求, 如果子进程结束之后, 父进程没有去 wait 回收子进程, 则子进程将成为僵尸进程; 如果父进程 wait 等待子进程退出, 将又会增加父进程的负担、也就是增加服务器的负担, 影响服务器进程的并发性能, 在 Linux 下, 可以将 SIGCHLD 信号的处理方式设置为 SIG_IGN, 也就是忽略该信号, 可让内核将僵尸进程转交给 init 进程去处理, 这样既不会产生僵尸进程、又省去了服务器进程回收子进程所占用的时间。

守护进程一般以单例模式运行, 关于单例模式运行请看 9.13.3 小节内容。

接下来, 我们根据上面的介绍的步骤, 来编写一个守护进程程序, 示例代码如下所示:

示例代码 9.13.1 守护进程示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>

int main(void)
{
    pid_t pid;
    int i;

    /* 创建子进程 */
    pid = fork();
    if (0 > pid) {
        perror("fork error");
        exit(-1);
    }
    else if (0 < pid)//父进程
```

```
    exit(0);    //直接退出

/*
 *子进程
 */

/* 1.创建新的会话、脱离控制终端 */
if (0 > setsid()) {
    perror("setsid error");
    exit(-1);
}

/* 2.设置当前工作目录为根目录 */
if (0 > chdir("/")) {
    perror("chdir error");
    exit(-1);
}

/* 3.重设文件权限掩码 umask */
umask(0);

/* 4.关闭所有文件描述符 */
for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);

/* 5.将文件描述符为 0、1、2 定位到/dev/null */
open("/dev/null", O_RDWR);
dup(0);
dup(0);

/* 6.忽略 SIGCHLD 信号 */
signal(SIGCHLD, SIG_IGN);

/* 正式进入到守护进程 */
for (;;) {
    sleep(1);
    puts("守护进程运行中.....");
}

exit(0);
}
```

整个代码的编写都是根据上面的介绍来完成的,这里就不再啰嗦,示例代码中使用到的函数在前面都已经学习过,其中第 4 步中调用 `sysconf(_SC_OPEN_MAX)`用于获取当前系统允许进程打开的最大文件数量。

我们在守护进程中添加了死循环, 每隔 1 秒钟打印一行字符串信息, 接下来编译运行:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.13.2 测试结果

运行之后, 没有任何打印信息输出, 原因在于守护进程已经脱离了控制终端, 它的打印信息并不会输出显示到终端, 在代码中已经将标准输入、输出以及错误重定位到了/dev/null, /dev/null 是一个黑洞文件, 自然是看不到输出信息。

使用"ps -ajx"命令查看进程, 如下所示:

```
  2  20555      0      0 ?          -1 I      0      0:00 [kworker/2:2]
  2  20568      0      0 ?          -1 T      0      0:00 [kworker/u256:2]
1911 20634    20634    20634 ?          -1 Ss     1000    0:00 ./testApp
3304 20002    20002    3304 pts/19    20002 R+      1000    0:00 ps -ajx
3372 21681    2131     2131 ?          -1 Sl      1000    1:09 /home/dt/.vscode/extensions/ms-vscode
3315 22627    2131     2131 ?          -1 Sl      1000    1:16 /usr/share/code/code /usr/share/code/
  2  37425      0      0 ?          -1 I<      0      0:00 [xfsalloc]
  2  37426      0      0 ?          -1 I<      0      0:00 [xfs_mru_cache]
  2  37431      0      0 ?          -1 S        0      0:00 [jfsIO]
  2  37432      0      0 ?          -1 S        0      0:00 [jfsCommit]
  2  37433      0      0 ?          -1 S        0      0:00 [jfsCommit]
  2  37434      0      0 ?          -1 S        0      0:00 [jfsCommit]
  2  37435      0      0 ?          -1 S        0      0:00 [jfsCommit]
  2  37436      0      0 ?          -1 S        0      0:00 [jfsCommit]
  2  37437      0      0 ?          -1 S        0      0:00 [jfsCommit]
  2  37438      0      0 ?          -1 S        0      0:00 [jfsSync]
1911 79017    1996     1996 ?          -1 Sl      1000    0:00 /usr/lib/gvfs/gvfsd-http --spawner :1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.13.3 查看守护进程

从上图可知, testApp 进程成为了一个守护进程, 与控制台脱离, 当关闭当前控制终端时, testApp 进程并不会受到影响, 依然会正常继续运行; 而对于普通进程来说, 终端关闭, 那么由该终端运行的所有进程都会被强制关闭, 因为它们处于同一个会话。关于这个问题, 大家可以自己去测试下, 对比测试普通进程与守护进程, 当终端关闭之后是否还在继续运行。

守护进程可以通过终端命令行启动, 但通常它们是由系统初始化脚本进行启动, 譬如/etc/rc*或/etc/init.d/*等。

9.13.3 SIGHUP 信号

当用户准备退出会话时, 系统向该会话发出 SIGHUP 信号, 会话将 SIGHUP 信号发送给所有子进程, 子进程接收到 SIGHUP 信号后, 便会自动终止, 当所有会话中的所有进程都退出时, 会话也就终止了; 因为程序当中一般不会对 SIGHUP 信号进行处理, 所以对应的处理方式为系统默认方式, SIGHUP 信号的系统默认处理方式便是终止进程。

上面解释了, 为什么子进程会随着会话的退出而退出, 因为它收到了 SIGHUP 信号。不管是前台进程还是后台进程都会收到该信号。如果忽略该信号, 将会出现什么样的结果? 接下来我们编写一个简单地测试程序, 示例代码如下所示:

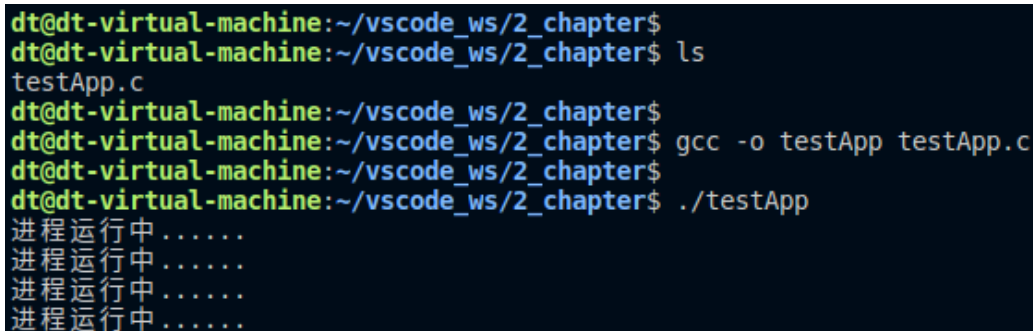
示例代码 9.13.2 忽略 SIGHUP 示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
```

```
int main(void)
{
    signal(SIGHUP, SIG_IGN);

    for (;;) {
        sleep(1);
        puts("进程运行中.....");
    }
}
```

代码很简单, 调用 `signal()` 函数将 `SIGHUP` 信号的处理方式设置为忽略。测试运行



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
进程运行中.....
进程运行中.....
进程运行中.....
进程运行中.....
```

图 9.13.4 测试结果

成功运行之后, 关闭终端

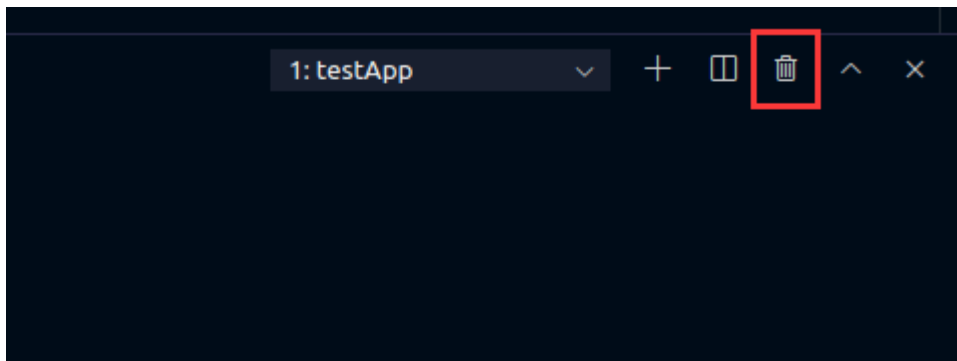
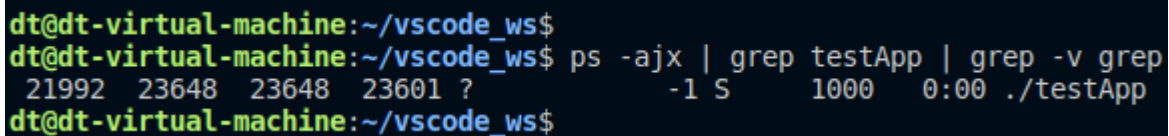


图 9.13.5 关闭终端

再重新打开终端, 使用 `ps` 命令查看 `testApp` 进程是否存在, 如下所示:



```
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ ps -ajx | grep testApp | grep -v grep
21992 23648 23648 23601 ?        -1 S    1000   0:00 ./testApp
dt@dt-virtual-machine:~/vscode_ws$
```

图 9.13.6 查看 testApp 进程

可以发现 `testApp` 进程依然还在运行, 但此时它已经变成了守护进程, 脱离了控制终端, 所以由此可知, 当程序当中忽略 `SIGHUP` 信号之后, 进程不会随着终端退出而退出, 事实上, 控制终端只是会话中的一个进程, 只有会话中的所有进程退出后, 会话才会结束; 很显然当程序中忽略了 `SIGHUP` 信号, 导致该进程不会终止, 所以会话也依然会存在, 从上图可知, 其会话 ID 等于 23601, 但此时会话已经没有控制终端了。

9.14 单例模式运行

通常情况下,一个程序可以被多次执行,即程序在还没有结束的情况下,又再次执行该程序,也就是系统中同时存在多个该程序的实例化对象(进程),譬如大家所熟悉的聊天软件 QQ,我们可以在电脑上同时登陆多个 QQ 账号,譬如还有一些游戏也是如此,在一台电脑上同时登陆多个游戏账号,只要你电脑不卡机、随便你开几个号。

但对于有些程序设计来说,不允许出现这种情况,程序只能被执行一次,只要该程序没有结束,就无法再次运行,我们把这种情况称为单例模式运行。譬如系统中守护进程,这些守护进程一般都是服务器进程,服务器程序只需要运行一次即可,能够在系统整个的运行过程中提供相应的服务支持,多次同时运行并没有意义、甚至还会带来错误!

如果希望我们的程序具有单例模式运行的功能,应该如何去实现呢?

9.14.1 通过文件存在与否进行判断

首先这是一个非常简单且容易想到的方法:用一个文件的存在与否来做标志,在程序运行正式代码之前,先判断一个特定的文件是否存在,如果存在则表明进程已经运行,此时应该立马退出;如果不存在则表明进程没有运行,然后创建该文件,当程序结束时再删除该文件即可!

这种方法是大家比较容易想到的,通过一个特定文件的存在与否来做判断,当然这个特定的文件的命名要弄的特殊一点,避免在文件系统中不会真的存在该文件,接下来我们编写一个程序进行测试。

示例代码 9.14.1 简单方式实现单例模式运行

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define LOCK_FILE    "./testApp.lock"

static void delete_file(void)
{
    remove(LOCK_FILE);
}

int main(void)
{
    /* 打开文件 */
    int fd = open(LOCK_FILE, O_RDONLY | O_CREAT | O_EXCL, 0666);
    if (-1 == fd) {
        fputs("不能重复执行该程序!\n", stderr);
        exit(-1);
    }

    /* 注册进程终止处理函数 */
```

```
if (atexit(delete_file))
    exit(-1);

puts("程序运行中...");
sleep(10);
puts("程序结束");

close(fd);           //关闭文件
exit(0);
}
```

在上述示例代码中,通过当前目录下的 testApp.lock 文件作为特定文件进行判断该文件是否存在,当然这里只是举个例子,如果在实际应用编程中使用了这种方法,这个特定文件需要存放在一个特定的路径下。

代码中以 O_RDONLY|O_CREAT|O_EXCL 的方式打开文件,如果文件不存在则创建文件,如果文件存在则 open 会报错返回-1;使用 atexit 注册进程终止处理函数,当程序退出时,使用 remove()删除该文件。

运行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[1] 460
程序运行中...
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ 程序结束

[1]+ 已完成 ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
程序运行中...
程序结束
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.14.1 测试结果

在上面测试中,首先第一次以后台方式运行了 testApp 程序,之后再运行 testApp 程序,由于文件已经存在,所以 open()调用会失败,所以意味着进程正在运行中,所以会打印相应的字符串然后退出。直到第一次运行的程序结束时,才能执行 testApp 程序,这样就实现了一个简单地具有单例模式运行功能的程序。

虽然上面实现了一个简单地单例模式运行的程序,但是仔细一想其实有很大的问题,主要包括如下三个方面:

- 程序中使用_exit()退出,那么将无法执行 delete_file()函数,意味着无法删除这个特定的文件;
- 程序异常退出。程序异常同样无法执行到进程终止处理函数 delete_file(),同样将导致无法删除这个特定的文件;
- 计算机掉电关机。这种情况就更加直接了,计算机可能在程序运行到任意位置时发生掉电关机的情况,这是无法预料的;如果文件没有删除就发生了这种情况,计算机重启之后文件依然存在,导致程序无法执行。

针对第一种情况, 我们使用 `exit()` 代替 `_exit()` 可以很好的解决这种问题; 但是对于第二种情况来说, 异常退出, 譬如进程接收到信号导致异常终止, 有一种解决办法便是设置信号处理方式为忽略信号, 这样当进程接收到信号时就会被忽略, 或者是针对某些信号注册信号处理函数, 譬如 `SIGTERM`、`SIGINT` 等, 在信号处理函数中删除文件然后再退出进程; 但依然有个问题, 并不是所有信号都可被忽略或捕获的, 譬如 `SIGKILL` 和 `SIGSTOP`, 这两个信号是无法被忽略和捕获的, 故而这种也不靠谱。

针对第三种情况的解决办法便是, 使得该特定文件会随着系统的重启而销毁, 这个怎么做呢? 其实这个非常简单, 将文件放置到系统 `/tmp` 目录下, `/tmp` 是一个临时文件系统, 当系统重启之后 `/tmp` 目录下的文件就会被销毁, 所以该目录下的文件的生命周期便是系统运行周期。

由此可知, 虽然针对第一种情况和第三种情况都有相应的解决办法, 但对于第二种情况来说, 其解决办法并不靠谱, 所以使用这种方法实现单例模式运行并不靠谱。

9.14.2 使用文件锁

介绍完上面第一种比较容易想到的方法外, 接下来介绍一种靠谱的方法, 使用文件锁来实现, 事实上这种方式才是实现单例模式运行靠谱的方法。

同样也需要通过一个特定的文件来实现, 当程序启动之后, 首先打开该文件, 调用 `open` 时一般使用 `O_WRONLY|O_CREAT` 标志, 当文件不存在则创建该文件, 然后尝试去获取文件锁, 若是成功, 则将程序的进程号 (PID) 写入到该文件中, 写入后不要关闭文件或解锁 (释放文件锁), 保证进程一直持有该文件锁; 若是程序获取锁失败, 代表程序已经被运行、则退出本次启动。

Tips: 当程序退出或文件关闭之后, 文件锁会自动解锁!

文件锁属于本书高级 I/O 章节内容, 在 13.6 小节对此做了详细介绍, 这里就不再说明, 通过系统调用 `flock()`、`fcntl()` 或库函数 `lockf()` 均可实现对文件进行上锁, 本小节我们以系统调用 `flock()` 为例, 系统调用 `flock()` 产生的是咨询锁 (建议性锁)、并不能产生强制性锁。

接下来编写一个示例代码, 使用 `flock()` 函数对文件上锁, 实现程序以单例模式运行, 如下所示:

示例代码 9.14.2 使用文件锁实现单例模式运行

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

#define LOCK_FILE    "./testApp.pid"

int main(void)
{
    char str[20] = {0};
    int fd;

    /* 打开 lock 文件, 如果文件不存在则创建 */
    fd = open(LOCK_FILE, O_WRONLY | O_CREAT, 0666);
    if (-1 == fd) {
        perror("open error");
    }
}
```

```
        exit(-1);
    }

    /* 以非阻塞方式获取文件锁 */
    if (-1 == flock(fd, LOCK_EX | LOCK_NB)) {
        fputs("不能重复执行该程序!\n", stderr);
        close(fd);
        exit(-1);
    }

    puts("程序运行中...");

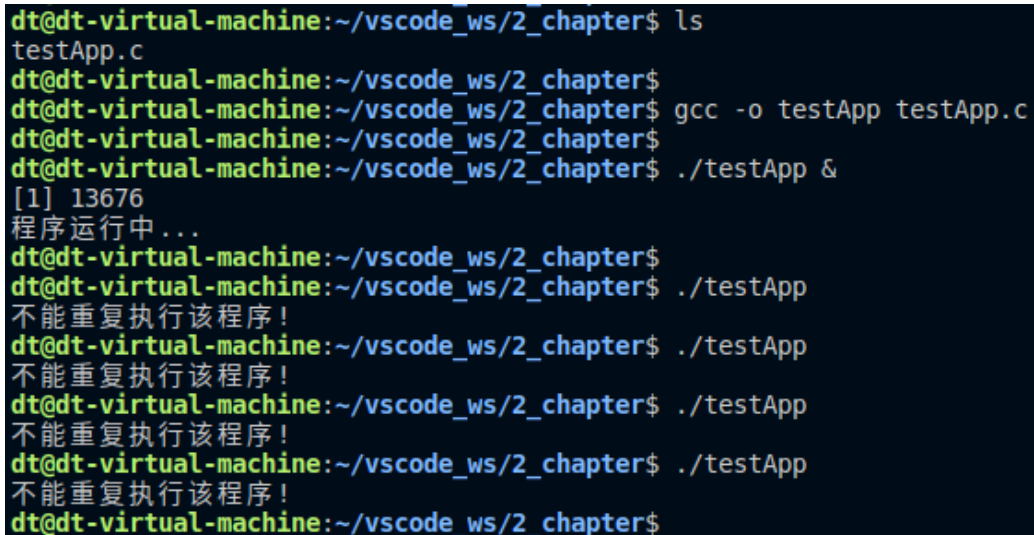
    ftruncate(fd, 0); //将文件长度截断为 0
    sprintf(str, "%d\n", getpid());
    write(fd, str, strlen(str)); //写入 pid

    for (;;)
        sleep(1);

    exit(0);
}
```

程序启动首先打开一个特定的文件, 这里只是举例, 以当前目录下的 `testApp.pid` 文件作为特定文件, 以 `O_WRONLY|O_CREAT` 方式打开, 如果文件不存在则创建该文件; 打开文件之后使用 `flock` 尝试获取文件锁, 调用 `flock()` 时指定了互斥锁标志 `LOCK_NB`, 意味着同时只能有一个进程拥有该锁, 如果获取锁失败, 表示该程序已经启动了, 无需再次执行, 然后退出; 如果获取锁成功, 将进程的 PID 写入到该文件中, 当程序退出时, 会自动解锁、关闭文件。

运行测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[1] 13676
程序运行中...
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.14.2 测试结果

这种机制在一些程序尤其是服务器程序中很常见, 服务器程序使用这种方法来保证程序的单例模式运行; 在 Linux 系统中/var/run/目录下有很多以.pid 为后缀结尾的文件, 这个实际上是为了保证程序以单例模式运行而设计的, 作为程序实现单例模式运行所需的特定文件, 如下所示:

```
dt@dt-virtual-machine:/var/run$ pwd
/var/run
dt@dt-virtual-machine:/var/run$
dt@dt-virtual-machine:/var/run$ ls
acpid.pid          lightdm          snapd.socket
acpid.socket       lightdm.pid     sshd
agetty.reload     lock            sshd.pid
alsa              log             sudo
apport.lock       mlocate.daily.lock systemd
avahi-daemon      motd.dynamic   thermald
blkid             mount          tmpfiles.d
crond.pid         network        udev
crond.reboot     NetworkManager udisks2
cups             plymouth      unattended-upgrades.lock
dbus            pppconfig     unattended-upgrades.progress
dhclient-ens33.pid reboot-required user
dnsmasq         reboot-required.pkgs utmp
do-not-hibernate resolvconf    uuid
firefox-restart-required rsyslogd.pid vmblock-fuse
initctl        sendsigs.omit.d vmtotalsd.pid
initramfs      shm           vmware
irqbalance.pid snapd-snap.socket
```

图 9.14.3 /var/run 目录下的文件

这些以.pid 为后缀的文件, 命名方式通常是程序名+.pid, 譬如 acpid.pid 对应的程序便是 acpid、lightdm.pid 对应的程序便是 lightdm 等等。如果我们要去实现一个以单例模式运行的程序, 譬如一个守护进程, 那么也应该将这个特定文件放置于 Linux 系统/var/run/目录下, 并且文件的命名方式为 name.pid (name 表示进程名)。

关于实现单例模式运行相关内容就给大家介绍这么多, 最常用的还是使用文件锁, 第一种方法通过文件存在否与进行判断事实上并不靠谱; 除此之外, 还有其它一些方法也可用于实现单例模式运行, 譬如在程序启动时通过 ps 判断进程是否存在等, 关于更多的方法, 欢迎大家留言!

第十章 进程间通信简介

前面学习了进程相关的内容,介绍了如何通过 `fork()`或 `vfork()`创建子进程,以及在子进程中通过 `exec()`函数执行一个新的程序。本章向大家介绍一个新的内容---进程间通信。

所谓进程间通信指的是系统中两个进程之间的通信,不同的进程都在各自的地址空间中、相互独立、隔离,所以它们是处于不同的地址空间中,因此相互通信比较难,Linux 内核提供了多种进程间通信的机制。本章就来聊一聊这些进程间通信的手段,让大家对此有一个基本的认识!

10.1 进程间通信简介

进程间通信 (interprocess communication, 简称 IPC) 指两个进程之间的通信。系统中的每一个进程都有各自的地址空间, 并且相互独立、隔离, 每个进程都处于自己的地址空间中。所以同一个进程的不同模块 (譬如不同的函数) 之间进行通信都是很简单的, 譬如使用全局变量等。

但是, 两个不同的进程之间要进行通信通常是比较难的, 因为这两个进程处于不同的地址空间中; 通常情况下, 大部分的程序是不要考虑进程间通信的, 因为大家所接触绝大部分程序都是单进程程序 (可以有多个线程), 对于一些复杂、大型的应用程序, 则会根据实际需要将其设计成多进程程序, 譬如 GUI、服务区应用程序等。

在一些中小型应用程序中通常不会将应用程序设计成多进程程序, 自然而然便不需要考虑进程间通信的问题, 所以, 本章内容以了解为主、了解进程间通信以及内核提供的进程间通信机制, 并不详解介绍进程间通信, 如果大家在今后的工作当中参与开发的应用程序是一个多进程程序、需要考虑进程间通信的问题, 此时再去深入学习这方面的知识!

10.2 进程间通信的机制有哪些?

Linux 内核提供了多种 IPC 机制, 基本是从 UNIX 系统继承而来, 而对 UNIX 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD (加州大学伯克利分校的伯克利软件发布中心) 在进程间通信方面的侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行了系统的改进和扩充, 形成了 “System V IPC”, 通信进程局限在单个计算机内; 后者则跳过了该限制, 形成了基于套接字 (Socket, 也就是网络) 的进程间通信机制。Linux 则把两者继承了下来, 如下如所示:

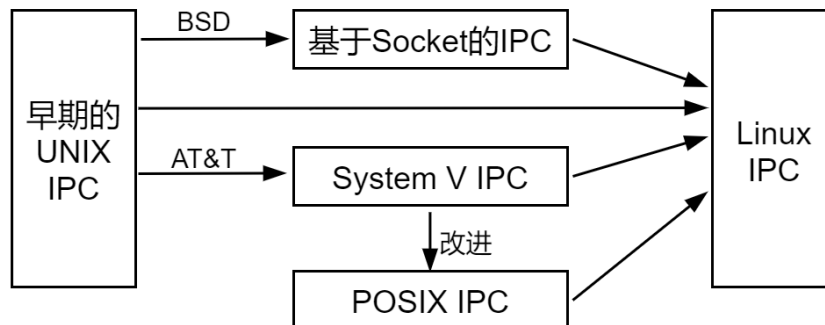


图 10.2.1 Linux 继承的进程间通信手段

其中, 早期的 UNIX IPC 包括: 管道、FIFO、信号; System V IPC 包括: System V 信号量、System V 消息队列、System V 共享内存; 上图中还出现了 POSIX IPC, 事实上, 较早的 System V IPC 存在着一些不足之处, 而 POSIX IPC 则是在 System V IPC 的基础上进行改进所形成的, 弥补了 System V IPC 的一些不足之处。POSIX IPC 包括: POSIX 信号量、POSIX 消息队列、POSIX 共享内存。

总结如下:

- UNIX IPC: 管道、FIFO、信号;
- System V IPC: 信号量、消息队列、共享内存;
- POSIX IPC: 信号量、消息队列、共享内存;
- Socket IPC: 基于 Socket 进程间通信。

10.3 管道和 FIFO

管道是 UNIX 系统上最古老的 IPC 方法, 它在 20 世纪 70 年代早期 UNIX 的第三个版本上就出现了。把一个进程连接到另一个进程的数据流称为管道, 管道被抽象成一个文件, 5.1 小节曾提及过管道文件 (pipe) 这样一种文件类型。

管道包括三种:

- 普通管道 `pipe`: 通常有两种限制, 一是单工, 数据只能单向传输; 二是只能在父子或者兄弟进程间使用;
- 流管道 `s_pipe`: 去除了普通管道的第一种限制, 为半双工, 可以双向传输; 只能在父子或兄弟进程间使用;
- 有名管道 `name_pipe` (FIFO): 去除了普通管道的第二种限制, 并且允许在不相关 (不是父子或兄弟关系) 的进程间进行通讯。

普通管道可用于具有亲缘关系的进程间通信, 并且数据只能单向传输, 如果要实现双向传输, 则必须要使用两个管道; 而流管道去除了普通管道的第一种限制, 可以半双工的方式实现双向传输, 但也只能在具有亲缘关系的进程间通信; 而有名管道 (FIFO) 则同时突破了普通管道的两种限制, 即可实现双向传输、又能在非亲缘关系的进程间通信。

10.4 信号

关于信号相关的内容在本书第八章中给大家介绍过, 用于通知接收信号的进程有某种事件发生, 所以可用于进程间通信; 除了用于进程间通信之外, 进程还可以发送信号给进程本身。

10.5 消息队列

消息队列是消息的链表, 存放在内核中并由消息队列标识符标识, 消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺陷。消息队列包括 POSIX 消息队列和 System V 消息队列。

消息队列是 UNIX 下不同进程之间实现共享资源的一种机制, UNIX 允许不同进程将格式化的数据流以消息队列形式发送给任意进程, 有足够权限的进程可以向队列中添加消息, 被赋予读权限的进程则可以读走队列中的消息。

10.6 信号量

信号量是一个计数器, 与其它进程间通信方式不大相同, 它主要用于控制多个进程间或一个进程内的多个线程间对共享资源的访问, 相当于内存中的标志, 进程可以根据它判定是否能够访问某些共享资源, 同时, 进程也可以修改该标志, 除了用于共享资源的访问控制外, 还可用于进程同步。

它常作为一种锁机制, 防止某进程在访问资源时其它进程也访问该资源, 因此, 主要作为进程间以及同一个进程内不同线程之间的同步手段。Linux 提供了一组精心设计的信号量接口来对信号量进行操作, 它们声明在头文件 `sys/sem.h` 中。

10.7 共享内存

共享内存就是映射一段能被其它进程所访问的内存, 这段共享内存由一个进程创建, 但其它的多个进程都可以访问, 使得多个进程可以访问同一块内存空间。共享内存是最快的 IPC 方式, 它是针对其它进程间通信方式运行效率低而专门设计的, 它往往与其它通信机制, 譬如结合信号量来使用, 以实现进程间的同步和通信。

10.8 套接字 (Socket)

Socket 是一种 IPC 方法, 是基于网络的 IPC 方法, 允许位于同一主机 (计算机) 或使用网络连接起来的的不同主机上的应用程序之间交换数据, 说白了就是网络通信, 在提高篇章内容中将会向大家介绍 Linux 系统下的网络编程。

在一个典型的客户端/服务器场景中，应用程序使用 socket 进行通信的方式如下：

- 各个应用程序创建一个 socket。socket 是一个允许通信的“设备”，两个应用程序都需要用到它。
- 服务器将自己的 socket 绑定到一个众所周知的地址（名称）上使得客户端能够定位到它的位置。

第十一章 线程

上一章,学习了进程相关的知识内容,对进程有了一个比较全面的认识和理解;本章开始,将学习 Linux 应用编程中非常重要的编程技巧---线程(Thread);与进程类似,线程是允许应用程序并发执行多个任务的一种机制,线程参与系统调度,事实上,系统调度的最小单元是线程、而并非进程。虽然线程的概念比较简单,但是其所涉及到的内容比较多,所以本章篇幅会相对比较长,大家加油!

本章将会讨论如下主题内容。

- 线程的基本概念,线程 VS 进程;
- 线程标识;
- 线程创建与回收;
- 线程取消;
- 线程终止;
- 线程分离;
- 线程同步技术;
- 线程安全。

11.1 线程概述

11.1.1 线程概念

什么是线程?

线程是参与系统调度的最小单位。它被包含在进程之中,是进程中的实际运行单位。一个线程指的是进程中一个单一顺序的控制流(或者说是执行路线、执行流),一个进程中可以创建多个线程,多个线程实现并发运行,每个线程执行不同的任务。譬如某应用程序设计了两个需要并发运行的任务 task1 和 task2,可将两个不同的任务分别放置在两个线程中。

线程是如何创建起来的?

当一个程序启动时,就有一个进程被操作系统(OS)创建,与此同时一个线程也立刻运行,该线程通常叫做程序的主线程(Main Thread),因为它是程序一开始时就运行的线程。应用程序都是以 main() 做为入口开始运行的,所以 main() 函数就是主线程的入口函数,main() 函数所执行的任务就是主线程需要执行的任务。

所以由此可知,任何一个进程都包含一个主线程,只有主线程的进程称为单线程进程,譬如前面章节内容中所编写的所有应用程序都是单线程程序,它们只有主线程;既然有单线程进程,那自然就存在多线程进程,所谓多线程指的是除了主线程以外,还包含其它的线程,其它线程通常由主线程来创建(调用 pthread_create 创建一个新的线程),那么创建的新线程就是主线程的子线程。

主线程的重要性体现在两方面:

- 其它新的线程(也就是子线程)是由主线程创建的;
- 主线程通常会在最后结束运行,执行各种清理工作,譬如回收各个子线程。

线程的特点?

线程是程序最基本的运行单位,而进程不能运行,真正运行的是进程中的线程。当启动应用程序后,系统就创建了一个进程,可以认为进程仅仅是一个容器,它包含了线程运行所需的数据结构、环境变量等信息。

同一进程中的多个线程将共享该进程中的全部系统资源,如虚拟地址空间,文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈(call stack, 我们称为线程栈),自己的寄存器环境(register context)、自己的线程本地存储(thread-local storage)。

在多线程应用程序中,通常一个进程中包括了多个线程,每个线程都可以参与系统调度、被 CPU 执行,线程具有以下一些特点:

- 线程不单独存在、而是包含在进程中;
- 线程是参与系统调度的基本单位;
- 可并发执行。同一进程的多个线程之间可并发执行,在宏观上实现同时运行的效果;
- 共享进程资源。同一进程中的各个线程,可以共享该进程所拥有的资源,这首先表现在:所有线程都具有相同的地址空间(进程的地址空间),这意味着,线程可以访问该地址空间的每一个虚地址;此外,还可以访问进程所拥有的已打开文件、定时器、信号量等等。

线程与进程?

进程创建多个子进程可以实现并发处理多任务(本质上便是多个单线程进程),多线程同样也可以实现(一个多线程进程)并发处理多任务的需求,那我们究竟选择哪种处理方式呢?首先我们就需要来分析下多进程和多线程两种编程模型的优势和劣势。

多进程编程的劣势:

- 进程间切换开销大。多个进程同时运行（指宏观上同时运行，无特别说明，均指宏观上），微观上依然是轮流切换运行，进程间切换开销远大于同一进程的多个线程间切换的开销，通常对于一些中小型应用程序来说不划算。
- 进程间通信较为麻烦。每个进程都在各自的地址空间中、相互独立、隔离，处在于不同的地址空间中，因此相互通信较为麻烦，在上一章节给大家有所介绍。

解决方案便是使用多线程编程，多线程能够弥补上面的问题：

- 同一进程的多个线程间切换开销比较小。
- 同一进程的多个线程间通信容易。它们共享了进程的地址空间，所以它们都是在同一个地址空间中，通信容易。
- 线程创建的速度远大于进程创建的速度。
- 多线程在多核处理器上更有优势！

综上所述，多线程编程相比于多进程编程的优势是比较明显的，在实际的应用当中多线程远比多进程应用更为广泛。那既然如此，为何还存在多进程编程模型呢？难道多线程编程就不存在缺点吗？当然不是，多线程也有它的缺点、劣势，譬如多线程编程难度高，对程序员的编程功底要求比较高，因为在多线程环境下需要考虑很多的问题，例如线程安全问题、信号处理的问题等，编写与调试一个多线程程序比单线程程序困难得多。

当然除此之外，还有一些其它的缺点，这里就不再一一列举了。多进程编程通常会用在一些大型应用程序项目中，譬如网络服务器应用程序，在中小型应用程序中用的比较少。

11.1.2 并发和并行

在前面的内容中，曾多次提到了并发这个概念，与此相类似的概念还有并行、串行，这里和大家聊一聊这些概念含义的区别。

对于串行比较容易理解，它指的是一种顺序执行，譬如先完成 task1，接着做 task2、直到完成 task2，然后做 task3、直到完成 task3.....依次按照顺序完成每一件事情，必须要完成上一件事才能去做下一件事，只有一个执行单元，这就是串行运行。

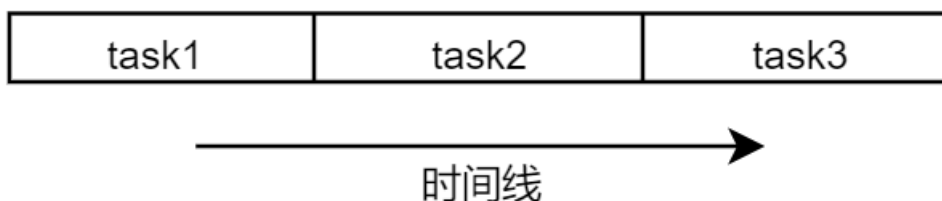


图 11.1.1 串行运行示意图

并行与串行则截然不同，并行指的是可以并排/并列执行多个任务，这样的系统，它通常有多个执行单元，所以可以实现并行运行，譬如并行运行 task1、task2、task3。

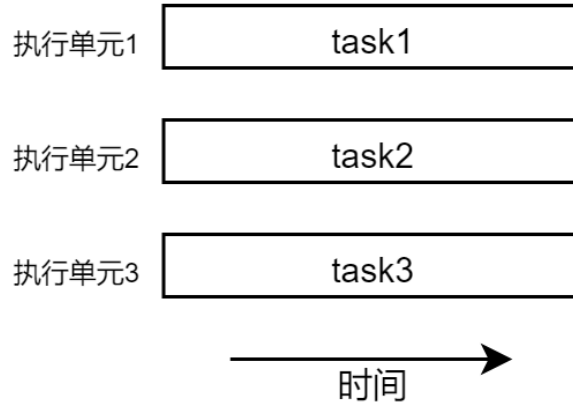


图 11.1.2 并行运行示意图 1

并行运行并不一定要同时开始运行、同时结束运行,只需满足在某一个时间段上存在多个任务被多个执行单元同时在运行着,譬如:

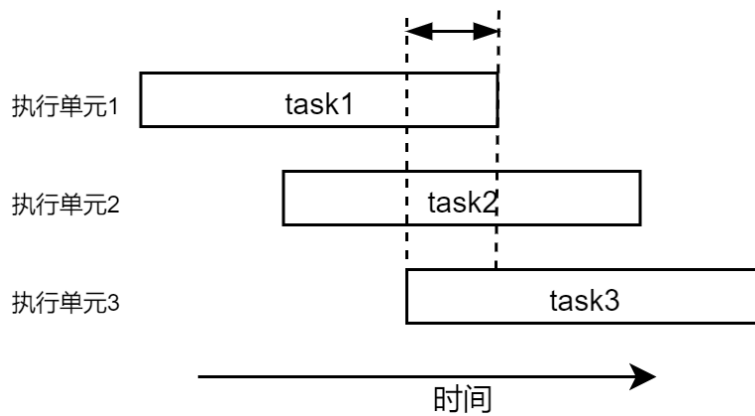


图 11.1.3 并行运行示意图 2

相比于串行和并行,并发强调的是一种时分复用,与串行的区别在于,它不必等待上一个任务完成之后在做下一个任务,可以打断当前执行的任务切换执行下一个任何,这就是时分复用。在同一个执行单元上,将时间分解成不同的片段(时间片),每个任务执行一段时间,时间一到则切换执行下一个任务,依次这样轮训(交叉/交替执行),这就是并发运行。如下图所示:

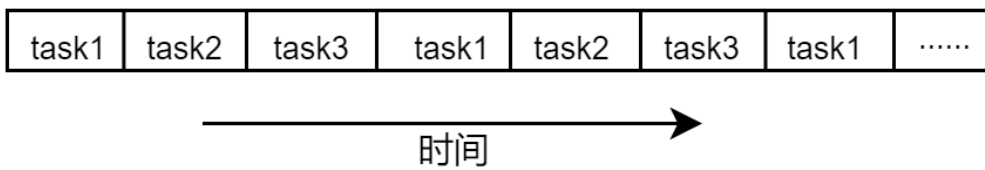


图 11.1.4 并发运行示例图

笔者在网络上看到了很多比较有意思、形象生动的比喻,用来说明串行、并行以及并发这三个概念的区别,这里笔者截取其中的一个:

- 你吃饭吃到一半,电话来了,你一直到吃完了以后才去接电话,这就说明你不支持并发也不支持并行,仅仅只是串行。
- 你吃饭吃到一半,电话来了,你停下吃饭去接了电话,电话接完后继续吃饭,这说明你支持并发。
- 你吃饭吃到一半,电话来了,你一边打电话一边吃饭,这说明你支持并行。

这里再次进行总结:

- 串行: 一件事、一件事接着做
- 并发: 交替做不同的事;
- 并行: 同时做不同的事。

需要注意的是, 并行运行情况下的多个执行单元, 每一个执行单元同样也可以以并发方式运行。

从通用角度上介绍完这三个概念之后, 类比到计算机系统中, 首先我们需要知道两个前提条件:

- 多核处理器和单核处理器: 对于单核处理器来说, 只有一个执行单元, 同时只能执行一条指令; 而对于多核处理来说, 有多个执行单元, 可以并行执行多条指令, 譬如 8 核处理器, 那么可以并行执行 8 条不同的指令。
- 计算机操作系统中, 通常同时运行着几十上百个不同的线程, 在单核或多核处理系统中都是如此!

对于单核处理器系统来说, 它只有一个执行单元 (譬如 I.MX6U 硬件平台, 单核 Cortex-A7 SoC), 只能采用并发运行系统中的线程, 而肯定不可能是串行, 而事实上确实如此。内核实现了调度算法, 用于控制系统中所有线程的调度, 简单点来说, 系统中所有参与调度的线程会加入到系统的调度队列中, 它们由内核控制, 每一个线程执行一段时间后, 由系统调度切换执行调度队列中下一个线程, 依次进行。在前面章节内容中也给大家简单地提到过系统调用的问题, 关于更加详细的内容, 这里便不再介绍了, 我们只需有个大概的认识、了解即可!

对于多核处理器系统来说, 它拥有多个执行单元, 在操作系统中, 多个执行单元以并行方式运行多个线程, 同时每一个执行单元以并发方式运行系统中的多个线程。

同时运行

计算机处理器运行速度是非常快的, 在单个处理核心虽然以并发方式运行着系统中的线程 (微观上交替/交叉方式运行不同的线程), 但在宏观上所表现出来的效果是同时运行着系统中的所有线程, 因为处理器的运算速度太快了, 交替轮训一次所花费的时间在宏观上几乎是可以忽略不计的, 所以表示出来的效果就是同时运行着所有线程。

这就好比现实生活中所看到的一些事情, 它所给带来的视角效果, 譬如一辆车在高速上行驶, 有时你会感觉到车的轮毂没有转动, 一种视角暂留现象, 因为车轮转动速度太快了, 人眼是看不清的, 会感觉车轮好像是静止的, 事实上, 车轮肯定是在转动着。

本小节的内容到这里就结束了, 理解了本小节的内容, 对于后面内容的将会有很大的帮助、也可以帮助大家快速理解后面的内容, 大家加油!

11.2 线程 ID

就像每个进程都有一个进程 ID 一样, 每个线程也有其对应的标识, 称为线程 ID。进程 ID 在整个系统中是唯一的, 但线程 ID 不同, 线程 ID 只有在它所属的进程上下文中才有意义。

进程 ID 使用 `pid_t` 数据类型来表示, 它是一个非负整数。而线程 ID 使用 `pthread_t` 数据类型来表示, 一个线程可通过库函数 `pthread_self()` 来获取自己的线程 ID, 其函数原型如下所示:

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

使用该函数需要包含头文件 `<pthread.h>`。

该函数调用总是成功, 返回当前线程的线程 ID。

可以使用 `pthread_equal()` 函数来检查两个线程 ID 是否相等, 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

如果两个线程 ID `t1` 和 `t2` 相等, 则 `pthread_equal()` 返回一个非零值; 否则返回 0。在 Linux 系统中, 使用无符号长整型 (`unsigned long int`) 来表示 `pthread_t` 数据类型, 但是在其它系统当中, 则不一定是无符号

长整型, 所以我们将 `pthread_t` 作为一种不透明的数据类型加以对待, 所以 `pthread_equal()` 函数用于比较两个线程 ID 是否相等是有用的。

线程 ID 在应用程序中非常有用, 原因如下:

- 很多线程相关函数, 譬如后面将要学习的 `pthread_cancel()`、`pthread_detach()`、`pthread_join()` 等, 它们都是利用线程 ID 来标识要操作的目标线程;
- 在一些应用程序中, 以特定线程的线程 ID 作为动态数据结构的标签, 这某些应用场合颇为有用, 既可以用来标识整个数据结构的创建者或属主线程, 又可以确定随后对该数据结构执行操作的具体线程。

11.3 创建线程

启动程序时, 创建的进程只是一个单线程的进程, 称之为初始线程或主线程, 本小节我们讨论如何创建一个新的线程。

主线程可以使用库函数 `pthread_create()` 负责创建一个新的线程, 创建出来的新线程被称为主线程的子线程, 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

使用该函数需要包含头文件 `<pthread.h>`。

函数参数和返回值含义如下:

thread: `pthread_t` 类型指针, 当 `pthread_create()` 成功返回时, 新创建的线程的线程 ID 会保存在参数 `thread` 所指向的内存中, 后续的线程相关函数会使用该标识来引用此线程。

attr: `pthread_attr_t` 类型指针, 指向 `pthread_attr_t` 类型的缓冲区, `pthread_attr_t` 数据类型定义了线程的各种属性, 关于线程属性将会在 11.8 小节介绍。如果将参数 `attr` 设置为 `NULL`, 那么表示将线程的所有属性设置为默认值, 以此创建新线程。

start_routine: 参数 `start_routine` 是一个函数指针, 指向一个函数, 新创建的线程从 `start_routine()` 函数开始运行, 该函数返回值类型为 `void *`, 并且该函数的参数只有一个 `void *`, 其实这个参数就是 `pthread_create()` 函数的第四个参数 `arg`。如果需要向 `start_routine()` 传递的参数有一个以上, 那么需要把这些参数放到一个结构体中, 然后把这个结构体对象的地址作为 `arg` 参数传入。

arg: 传递给 `start_routine()` 函数的参数。一般情况下, 需要将 `arg` 指向一个全局或堆变量, 意思就是在线程的生命周期中, 该 `arg` 指向的对象必须存在, 否则如果线程中访问了该对象将会出现错误。当然也可将参数 `arg` 设置为 `NULL`, 表示不需要传入参数给 `start_routine()` 函数。

返回值: 成功返回 0; 失败时将返回一个错误号, 并且参数 `thread` 指向的内容是不确定的。

注意 `pthread_create()` 在调用失败时通常会返回错误码, 它并不像其它库函数或系统调用一样设置 `errno`, 每个线程都提供了全局变量 `errno` 的副本, 这只是为了与使用 `errno` 到的函数进行兼容, 在线程中, 从函数中返回错误码更为清晰整洁, 不需要依赖那些随着函数执行不断变化的全局变量, 这样可以把错误的范围限制在引起出错的函数中。

线程创建成功, 新线程就会加入到系统调度队列中, 获取到 CPU 之后就会立马从 `start_routine()` 函数开始运行该线程的任务; 调用 `pthread_create()` 函数后, 通常我们无法确定系统接着会调度哪一个线程来使用 CPU 资源, 先调度主线程还是新创建的线程呢 (而在多核 CPU 或多 CPU 系统中, 多核线程可能会在不同的核心上同时执行)? 如果程序对执行顺序有强制要求, 那么就必须采用一些同步技术来实现。这与前面学习父、子进程时也出现了这个问题, 无法确定父进程、子进程谁先被系统调度。

使用示例

使用 `pthread_create()` 函数创建一个除主线程之外的新线程, 示例代码如下所示:

示例代码 11.3.1 `pthread_create()` 创建线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程: 进程 ID<%d> 线程 ID<%lu>\n", getpid(), pthread_self());
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    int ret;

    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "Error: %s\n", strerror(ret));
        exit(-1);
    }

    printf("主线程: 进程 ID<%d> 线程 ID<%lu>\n", getpid(), pthread_self());
    sleep(1);
    exit(0);
}
```

应该将 `pthread_t` 作为一种不透明的数据类型加以对待, 但是在示例代码中需要打印线程 ID, 所以要明确其数据类型, 示例代码中使用了 `printf()` 函数打印线程 ID 时, 将其作为 `unsigned long int` 数据类型, 在 Linux 系统下, 确实是使用 `unsigned long int` 来表示 `pthread_t`, 所以这样做没有问题!

主线程休眠了 1 秒钟, 原因在于, 如果主线程不进行休眠, 它就可能会立马退出, 这样可能会导致新创建的线程还没有机会运行, 整个进程就结束了。

在主线程和新线程中, 分别通过 `getpid()` 和 `pthread_self()` 来获取进程 ID 和线程 ID, 将结果打印出来, 运行结果如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
/tmp/cc8nwjLD.o: 在函数 'main' 中:
testApp.c:(.text+0x69): 对 'pthread_create' 未定义的引用
collect2: error: ld returned 1 exit status
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.3.1 编译报错

编译时出现了错误, 提示“对‘pthread_create’未定义的引用”, 示例代码确实已经包含了<pthread.h>头文件, 但为什么会出现这样的报错, 仔细看, 这个报错是出现在程序代码链接时、而并非是编译过程, 所以可知这是链接库的文件, 如何解决呢?

```
gcc -o testApp testApp.c -lpthread
```

使用-l选项指定链接库 pthread, 原因在于 pthread 不在 gcc 的默认链接库中, 所以需要手动指定。再次编译便不会有问题了, 如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
主线程: 进程 ID<30793> 线程 ID<140458989827840>
新线程: 进程 ID<30793> 线程 ID<140458981488384>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.3.2 测试结果

从打印信息可知, 正如前面所介绍那样, 两个线程的进程 ID 相同, 说明新创建的线程与主线程本来就属于同一个进程, 但是它们的线程 ID 不同。从打印结果可知, Linux 系统下线程 ID 数值非常大, 看起来像是一个指针。

11.4 终止线程

在示例代码 11.3.1 中, 我们在新线程的启动函数(线程 start 函数) new_thread_start() 通过 return 返回之后, 意味着该线程已经终止了, 除了在线程 start 函数中执行 return 语句终止线程外, 终止线程的方式还有多种, 可以通过如下方式终止线程的运行:

- 线程的 start 函数执行 return 语句并返回指定值, 返回值就是线程的退出码;
- 线程调用 pthread_exit() 函数;
- 调用 pthread_cancel() 取消线程(将在 11.6 小节介绍);

如果进程中的任意线程调用 exit()、_exit() 或者 _Exit(), 那么将会导致整个进程终止, 这里需要注意!

pthread_exit() 函数将终止调用它的线程, 其函数原型如下所示:

```
#include <pthread.h>
```

```
void pthread_exit(void *retval);
```

使用该函数需要包含头文件<pthread.h>。

参数 retval 的数据类型为 void *, 指定了线程的返回值、也就是线程的退出码, 该返回值可由另一个线程通过调用 pthread_join() 来获取; 同理, 如果线程是在 start 函数中执行 return 语句终止, 那么 return 的返回值也是可以通过 pthread_join() 来获取的。

参数 `retval` 所指向的内容不应分配于线程栈中, 因为线程终止后, 将无法确定线程栈的内容是否有效; 出于同样的理由, 也不应在线程栈中分配线程 `start` 函数的返回值。

调用 `pthread_exit()` 相当于在线程的 `start` 函数中执行 `return` 语句, 不同之处在于, 可在线程 `start` 函数所调用的任意函数中调用 `pthread_exit()` 来终止线程。如果主线程调用了 `pthread_exit()`, 那么主线程也会终止, 但其它线程依然正常运行, 直到进程中的所有线程终止才会使得进程终止。

使用示例

示例代码 11.4.1 `pthread_exit()` 终止线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程 start\n");
    sleep(1);
    printf("新线程 end\n");
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid;
    int ret;

    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "Error: %s\n", strerror(ret));
        exit(-1);
    }

    printf("主线程 end\n");
    pthread_exit(NULL);
    exit(0);
}
```

新线程中调用 `sleep()` 休眠, 保证主线程先调用 `pthread_exit()` 终止, 休眠结束之后新线程也调用 `pthread_exit()` 终止, 编译测试看看打印结果:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
主线程 end
新线程 start
新线程 end
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.4.1 测试结果

正如上面介绍到,主线程调用 `pthread_exit()` 终止之后,整个进程并没有结束,而新线程还在继续运行。

11.5 回收线程

在父、子进程当中,父进程可通过 `wait()` 函数(或其变体 `waitpid()`) 阻塞等待子进程退出并获取其终止状态,回收子进程资源;而在线程当中,也需要如此,通过调用 `pthread_join()` 函数来阻塞等待线程的终止,并获取线程的退出码,回收线程资源;`pthread_join()` 函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

使用该函数需要包含头文件 `<pthread.h>`。

函数参数和返回值含义如下:

thread: `pthread_join()` 等待指定线程的终止,通过参数 `thread` (线程 ID) 指定需要等待的线程;

retval: 如果参数 `retval` 不为 `NULL`,则 `pthread_join()` 将目标线程的退出状态(即目标线程通过 `pthread_exit()` 退出时指定的返回值或者在线程 `start` 函数中执行 `return` 语句对应的返回值)复制到 `*retval` 所指向的内存区域;如果目标线程被 `pthread_cancel()` 取消,则将 `PTHREAD_CANCELED` 放在 `*retval` 中。如果对目标线程的终止状态不感兴趣,则可将参数 `retval` 设置为 `NULL`。

返回值: 成功返回 0; 失败将返回错误码。

调用 `pthread_join()` 函数将会以阻塞的形式等待指定的线程终止,如果该线程已经终止,则 `pthread_join()` 立刻返回。如果多个线程同时尝试调用 `pthread_join()` 等待指定线程的终止,那么结果将是不确定的。

若线程并未分离(`detached`, 将在 11.6.1 小节介绍),则必须使用 `pthread_join()` 来等待线程终止,回收线程资源;如果线程终止后,其它线程没有调用 `pthread_join()` 函数来回收该线程,那么该线程将变成僵尸线程,与僵尸进程的概念相类似;同样,僵尸线程除了浪费系统资源外,若僵尸线程积累过多,那么会导致应用程序无法创建新的线程。

当然,如果进程中存在着僵尸线程并未得到回收,当进程终止之后,进程会被其父进程回收,所以僵尸线程同样也会被回收。

所以,通过上面的介绍可知,`pthread_join()` 执行的功能类似于针对进程的 `waitpid()` 调用,不过二者之间存在一些显著差别:

- 线程之间关系是对等的。进程中的任意线程均可调用 `pthread_join()` 函数来等待另一个线程的终止。譬如,如果线程 A 创建了线程 B,线程 B 再创建线程 C,那么线程 A 可以调用 `pthread_join()` 等待线程 C 的终止,线程 C 也可以调用 `pthread_join()` 等待线程 A 的终止;这与进程间层次关系不同,父进程如果使用 `fork()` 创建了子进程,那么它也是唯一能够对子进程调用 `wait()` 的进程,线程之间不存在这样的关系。
- 不能以非阻塞的方式调用 `pthread_join()`。对于进程,调用 `waitpid()` 既可以实现阻塞方式等待、也可以实现非阻塞方式等待。

使用示例

示例代码 11.5.1 pthread_join() 等待线程终止

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程 start\n");
    sleep(2);
    printf("新线程 end\n");
    pthread_exit((void *)10);
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

主线程调用 `pthread_create()` 创建新线程之后, 新线程执行 `new_thread_start()` 函数, 而在主线程中调用 `pthread_join()` 阻塞等待新线程终止, 新线程终止后, `pthread_join()` 返回, 将目标线程的退出码保存在 `*tret` 所指向的内存中。测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 start
新线程 end
新线程终止, code=10
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.5.1 测试结果

11.6 取消线程

在通常情况下,进程中的多个线程会并发执行,每个线程各司其职,直到线程的任务完成之后,该线程中会调用 `pthread_exit()` 退出,或在线程 `start` 函数执行 `return` 语句退出。

有时候,在程序设计需求当中,需要向一个线程发送一个请求,要求它立刻退出,我们把这种操作称为取消线程,也就是向指定的线程发送一个请求,要求其立刻终止、退出。譬如,一组线程正在执行一个运算,一旦某个线程检测到错误发生,需要其它线程退出,取消线程这项功能就派上用场了。

本小节就来讨论 Linux 系统下的线程取消机制。

11.6.1 取消一个线程

通过调用 `pthread_cancel()` 库函数向一个指定的线程发送取消请求,其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

使用该函数需要包含头文件 `<pthread.h>`, 参数 `thread` 指定需要取消的目标线程; 成功返回 0, 失败将返回错误码。

发出取消请求之后,函数 `pthread_cancel()` 立即返回,不会等待目标线程的退出。默认情况下,目标线程也会立刻退出,其行为表现为如同调用了参数为 `PTHREAD_CANCELED` (其实就是 `(void *)-1`) 的 `pthread_exit()` 函数,但是,线程可以设置自己不被取消或者控制如何被取消 (11.6.2 小节介绍),所以 `pthread_cancel()` 并不会等待线程终止,仅仅只是提出请求。

使用示例

示例代码 11.6.1 `pthread_cancel()` 取消线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程--running\n");
```

```
    for (;;)
        sleep(1);
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 向新线程发送取消请求 */
    ret = pthread_cancel(tid);
    if (ret) {
        fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

主线程创建新线程，新线程 `new_thread_start()` 函数直接运行 `for` 死循环；主线程休眠一段时间后，调用 `pthread_cancel()` 向新线程发送取消请求，接着再调用 `pthread_join()` 等待新线程终止、获取其终止状态，将线程退出码打印出来。测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 -- running
新线程终止, code=-1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.6.1 测试结果

由打印结果可知, 当主线程发送取消请求之后, 新线程便退出了, 而且退出码为-1, 也就是 PTHREAD_CANCELLED。

11.6.2 取消状态以及类型

默认情况下, 线程是响应其它线程发送过来的取消请求的, 响应请求然后退出线程。当然, 线程可以选择不被取消或者控制如何被取消, 通过 `pthread_setcancelstate()`和 `pthread_setcanceltype()`来设置线程的取消性状态和类型。

```
#include <pthread.h>
```

```
int pthread_setcancelstate(int state, int *oldstate);
```

```
int pthread_setcanceltype(int type, int *oldtype);
```

使用这些函数需要包含头文件 `<pthread.h>`, `pthread_setcancelstate()`函数会将调用线程的取消性状态设置为参数 `state` 中给定的值, 并将线程之前的取消性状态保存在参数 `oldstate` 指向的缓冲区中, 如果对之前的状态不感兴趣, Linux 允许将参数 `oldstate` 设置为 `NULL`; `pthread_setcancelstate()`调用成功将返回 0, 失败返回非 0 值的错误码。

`pthread_setcancelstate()`函数执行的设置取消性状态和获取旧状态操作, 这两步是一个原子操作。

参数 `state` 必须是以下值之一:

- **PTHREAD_CANCEL_ENABLE:** 线程可以取消, 这是新创建的线程取消性状态的默认值, 所以新建线程以及主线程默认都是可以取消的。
- **PTHREAD_CANCEL_DISABLE:** 线程不可被取消, 如果此类线程接收到取消请求, 则会将请求挂起, 直至线程的取消性状态变为 PTHREAD_CANCEL_ENABLE。

使用示例

修改示例代码 11.6.1, 在新线程的 `new_thread_start()`函数中调用 `pthread_setcancelstate()`函数将线程的取消性状态设置为 PTHREAD_CANCEL_DISABLE, 我们来试试, 此时主线程还能不能取消新线程, 示例代码如下所示:

示例代码 11.6.2 pthread_setcancelstate()使用示例

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <pthread.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
static void *new_thread_start(void *arg)
```

```
{
    /* 设置为不可被取消 */
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    for (;;) {
        printf("新线程--running\n");
        sleep(2);
    }
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 向新线程发送取消请求 */
    ret = pthread_cancel(tid);
    if (ret) {
        fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

新线程 `new_thread_start()` 函数中调用 `pthread_setcancelstate()` 将自己设置为不可被取消, 主线程延时 1 秒钟之后调用 `pthread_cancel()` 向新线程发送取消请求, 那么此时新线程是不会终止的, `pthread_cancel()` 立刻返回之后进入到 `pthread_join()` 函数, 那么此时会被阻塞等待新线程终止, 接下来运行测试看看, 结果会不会是这样:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 --running
新线程 --running
新线程 --running
新线程 --running
新线程 --running
新线程 --running
新线程 --running
```

图 11.6.2 测试结果

测试结果确实如此, 将一直重复打印“新线程--running”, 因为新线程是一个死循环(测试完成按 `Ctrl+C` 退出)。

`pthread_setcanceltype()` 函数

如果线程的取消性状态为 `PTHREAD_CANCEL_ENABLE`, 那么对取消请求的处理则取决于线程的取消性类型, 该类型可以通过调用 `pthread_setcanceltype()` 函数来设置, 它的参数 `type` 指定了需要设置的类型, 而线程之前的取消性类型则会保存在参数 `oldtype` 所指向的缓冲区中, 如果对之前的类型不感兴趣, Linux 下允许将参数 `oldtype` 设置为 `NULL`。同样 `pthread_setcanceltype()` 函数调用成功将返回 0, 失败返回非 0 值的错误码。

`pthread_setcanceltype()` 函数执行的设置取消性类型和获取旧类型操作, 这两步是一个原子操作。

参数 `type` 必须是以下值之一:

- **PTHREAD_CANCEL_DEFERRED:** 取消请求到来时, 线程还是继续运行, 取消请求被挂起, 直到线程到达某个取消点 (cancellation point, 将在 11.6.3 小节介绍) 为止, 这是所有新建线程包括主线程默认的取消性类型。
- **PTHREAD_CANCEL_ASYNCHRONOUS:** 可能会在任何时间点 (也许是立即取消, 但不一定) 取消线程, 这种取消性类型应用场景很少, 不再介绍!

当某个线程调用 `fork()` 创建子进程时, 子进程会继承调用线程的取消性状态和取消性类型, 而当某线程调用 `exec` 函数时, 会将新程序主线程的取消性状态和类型重置为默认值, 也就是 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DEFERRED`。

11.6.3 取消点

若将线程的取消性类型设置为 `PTHREAD_CANCEL_DEFERRED` 时 (线程可以取消状态下), 收到其它线程发送过来的取消请求时, 仅当线程抵达某个取消点时, 取消请求才会起作用。

那什么是取消点呢? 所谓取消点其实就是一系列函数, 当执行到这些函数的时候, 才会真正响应取消请求, 这些函数就是取消点; 在没有出现取消点时, 取消请求是无法得到处理的, 究其原因在于系统认为, 但没有到达取消点时, 线程此时正在执行的工作是不能被停止的, 正在执行关键代码, 此时终止线程将可能会导致出现意想不到的异常发生。

取消点函数包括哪些呢? 下表给大家简单地列出了一些:

表 11.6.1 可作为取消点的函数

accept()	mq_timedsend()	pthread_join()	sendto()
aio_suspend()	msgrcv()	pthread_testcancel()	sigsuspend()
clock_nanosleep()	msgsnd()	pwrite()	sigtimedwait()
close()	msync()	read()	sigwait()
connect()	nanosleep()	readv()	sigwaitinfo()
creat()	open()	recv()	sleep()
fcntl()	openat()	recvfrom()	system()
fdatasync()	pause()	recvmsg()	tcdrain()
fsync()	poll()	select()	wait()
lockf()	pread()	sem_timedwait()	waitid()
mq_receive()	pselect()	sem_wait()	waitpid()
mq_send()	pthread_cond_timedwait()	send()	write()
mq_timedreceive()	pthread_cond_wait()	sendmsg()	writev()

除了表 11.6.1 所列函数之外, 还有大量的函数, 系统实现可以将其作为取消点, 这里便不再一一列举出来了, 大家也可以通过 man 手册进行查询, 命令为"man 7 pthreads", 如下所示:

```

Cancellation points
POSIX.1 specifies that certain functions must, and certain other functions may, be cancellation points. If a thread is cancelable,
a cancellation request is pending for the thread, then the thread is canceled when it calls a function that is a cancellation point.

The following functions are required to be cancellation points by POSIX.1-2001 and/or POSIX.1-2008:

accept()
aio_suspend()
clock_nanosleep()
close()
connect()
creat()
fcntl() F_SETLKW
fdatasync()
fsync()
getmsg()
getpmsg()
lockf() F_LOCK
mq_receive()
mq_send()
mq_timedreceive()
mq_timedsend()
msgrcv()
msgsnd()
msync()
nanosleep()
open()
openat() [Added in POSIX.1-2008]
pause()
poll()
pread()
pselect()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()
pthread_testcancel()

```

图 11.6.3 查看可作为取消点的函数

线程在调用这些函数时, 如果收到了取消请求, 那么线程便会遭到取消; 除了这些作为取消点的函数之外, 不得将任何其它函数视为取消点 (亦即, 调用这些函数不会招致取消)。

示例代码 11.6.1 中, 新线程处于 for 循环之中, 调用 sleep() 休眠, 由表 11.6.1 可知, sleep() 函数可以作为取消点 (printf 可能也是), 当新线程接收到取消请求之后, 便会立马退出, 当如果将其修改为如下:

```

static void *new_thread_start(void *arg)
{
    printf("新线程--running\n");
    for (;;) {
    }
    return (void *)0;
}

```



```
}
```

那么线程将永远无法被取消, 因为这里不存在取消点。大家可以将代码进行修改测试, 看结果是不是如此!

11.6.4 线程可取消性的检测

假设线程执行的是一个不含取消点的循环(譬如 for 循环、while 循环), 那么这时线程永远也不会响应取消请求, 也就意味着除了线程自己主动退出, 其它线程将无法通过向它发送取消请求而终止它, 就如上小节最后给大家列举的例子。

在实际应用程序当中, 确实会遇到这种情况, 线程最终运行在一个循环当中, 该循环体内执行的函数不存在任何一个取消点, 但实际项目需求是: 该线程必须可以被其它线程通过发送取消请求的方式终止, 那这个时候怎么办? 此时可以使用 `pthread_testcancel()`, 该函数目的很简单, 就是产生一个取消点, 线程如果已有处于挂起状态的取消请求, 那么只要调用该函数, 线程就会随之终止。其函数原型如下所示:

```
#include <pthread.h>
```

```
void pthread_testcancel(void);
```

功能测试

接下来进行一个测试, 主线程创建一个新的进程, 新进程的取消性状态和类型置为默认, 新进程最终执行的是一个不含取消点的循环; 主线程向新线程发送取消请求, 示例代码如下所示:

示例代码 11.6.3 不含取消点的循环

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <pthread.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
static void *new_thread_start(void *arg)
```

```
{
```

```
    printf("新线程--start run\n");
```

```
    for (;;) {
```

```
    }
```

```
    return (void *)0;
```

```
}
```

```
int main(void)
```

```
{
```

```
    pthread_t tid;
```

```
    void *tret;
```

```
    int ret;
```

```

/* 创建新线程 */
ret = pthread_create(&tid, NULL, new_thread_start, NULL);
if (ret) {
    fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
    exit(-1);
}

sleep(1);

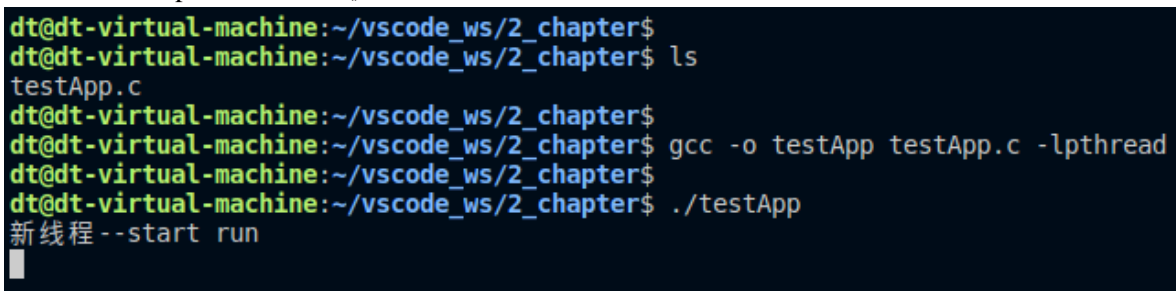
/* 向新线程发送取消请求 */
ret = pthread_cancel(tid);
if (ret) {
    fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
    exit(-1);
}

/* 等待新线程终止 */
ret = pthread_join(tid, &tret);
if (ret) {
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}
printf("新线程终止, code=%ld\n", (long)tret);

exit(0);
}

```

新线程的 `new_thread_start()` 函数中是一个 `for` 死循环, 没有执行任何函数, 所以是一个没有取消点的循环体, 主线程调用 `pthread_cancel()` 是无法将其终止的, 接下来测试下结果是否如此:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 --start run

```

图 11.6.4 测试结果

执行完之后, 程序一直会没有退出, 说明主线程确实无法终止新线程。接下来再做一个测试, 在 `new_thread_start` 函数的 `for` 循环体中执行 `pthread_testcancel()` 函数, 如下所示:

示例代码 11.6.4 使用 `pthread_testcancel()` 产生取消点

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

```

```
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程--start run\n");
    for ( ;; ) {
        pthread_testcancel();
    }
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 向新线程发送取消请求 */
    ret = pthread_cancel(tid);
    if (ret) {
        fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);
}
```

```
    exit(0);
}
```

如果 `pthread_testcancel()` 可以产生取消点, 那么主线程便可以终止新线程, 测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 --start run
新线程终止, code=-1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.6.5 测试结果

从打印结果可知, 确实如上面介绍那样, `pthread_testcancel()` 函数就是取消点。

11.7 分离线程

默认情况下, 当线程终止时, 其它线程可以通过调用 `pthread_join()` 获取其返回状态、回收线程资源, 有时, 程序员并不关系线程的返回状态, 只是希望系统在线程终止时能够自动回收线程资源并将其移除。在这种情况下, 可以调用 `pthread_detach()` 将指定线程进行分离, 也就是分离线程, `pthread_detach()` 函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t thread);
```

使用该函数需要包含头文件 `<pthread.h>`, 参数 `thread` 指定需要分离的线程, 函数 `pthread_detach()` 调用成功将返回 0; 失败将返回一个错误码。

一个线程既可以将另一个线程分离, 同时也可以将自己分离, 譬如:

```
pthread_detach(pthread_self());
```

一旦线程处于分离状态, 就不能再使用 `pthread_join()` 来获取其终止状态, 此过程是不可逆的, 一旦处于分离状态之后便不能再恢复到之前的状态。处于分离状态的线程, 当其终止后, 能够自动回收线程资源。

使用示例

示例代码 11.7.1 `pthread_detach()` 分离线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    int ret;

    /* 自行分离 */
    ret = pthread_detach(pthread_self());
    if (ret) {
```

```
        fprintf(stderr, "pthread_detach error: %s\n", strerror(ret));
        return NULL;
    }

    printf("新线程 start\n");
    sleep(2); //休眠 2 秒钟
    printf("新线程 end\n");
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1); //休眠 1 秒钟

    /* 等待新线程终止 */
    ret = pthread_join(tid, NULL);
    if (ret)
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));

    pthread_exit(NULL);
}
```

示例代码中, 主线程创建新的线程之后, 休眠 1 秒钟, 调用 `pthread_join()` 等待新线程终止; 新线程调用 `pthread_detach(pthread_self())` 将自己分离, 休眠 2 秒钟之后 `pthread_exit()` 退出线程; 主线程休眠 1 秒钟是能够确保调用 `pthread_join()` 函数时新线程已经将自己分离了, 所以按照上面的介绍可知, 此时主线程调用 `pthread_join()` 必然会失败, 测试结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 start
pthread_join error: Invalid argument
新线程 end
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.7.1 测试结果

打印结果正如我们所料，主线程调用 `pthread_join()` 确实会出错，错误提示为“Invalid argument”。

11.8 注册线程清理处理函数

9.1.2 小节学习了 `atexit()` 函数，使用 `atexit()` 函数注册进程终止处理函数，当进程调用 `exit()` 退出时就会执行进程终止处理函数；其实，当线程退出时也可以这样做，当线程终止退出时，去执行这样的处理函数，我们把这个称为线程清理函数（thread cleanup handler）。

与进程不同，一个线程可以注册多个清理函数，这些清理函数记录在栈中，每个线程都可以拥有一个清理函数栈，栈是一种先进后出的数据结构，也就是说它们的执行顺序与注册（添加）顺序相反，当执行完所有清理函数后，线程终止。

线程通过函数 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 分别负责向调用线程的清理函数栈中添加和移除清理函数，函数原型如下所示：

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

使用这些函数需要包含头文件 `<pthread.h>`。

调用 `pthread_cleanup_push()` 向清理函数栈中添加一个清理函数，第一个参数 `routine` 是一个函数指针，指向一个需要添加的清理函数，`routine()` 函数无返回值，只有一个 `void *` 类型参数；第二个参数 `arg`，当调用清理函数 `routine()` 时，将 `arg` 作为 `routine()` 函数的参数。

既然有添加，自然就会伴随着删除，就好比对应入栈和出栈，调用函数 `pthread_cleanup_pop()` 可以将清理函数栈中最顶层（也就是最后添加的函数，最后入栈）的函数移除。

当线程执行以下动作时，清理函数栈中的清理函数才会被执行：

- 线程调用 `pthread_exit()` 退出时；
- 线程响应取消请求时；
- 用非 0 参数调用 `pthread_cleanup_pop()`

除了以上三种情况之外，其它方式终止线程将不会执行线程清理函数，譬如在线程 `start` 函数中执行 `return` 语句退出时不会执行清理函数。

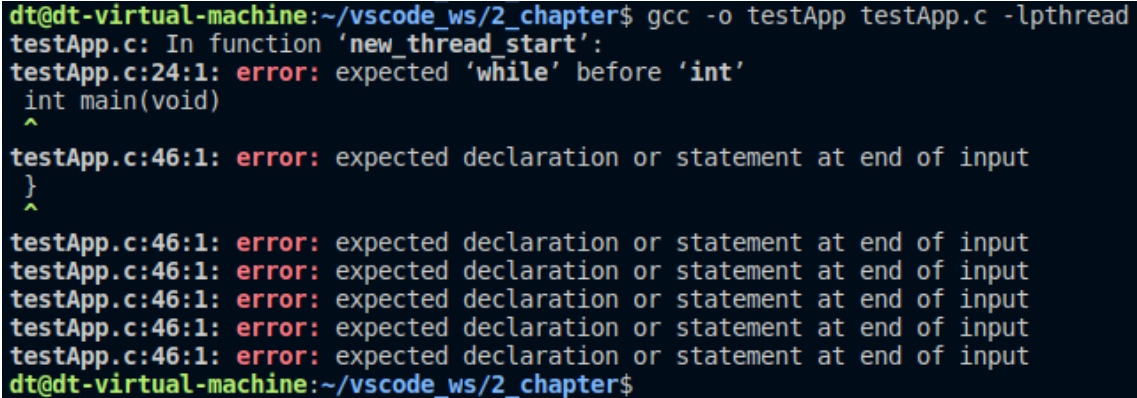
函数 `pthread_cleanup_pop()` 的 `execute` 参数，可以取值为 0，也可以为非 0；如果为 0，清理函数不会被调用，只是将清理函数栈中最顶层的函数移除；如果参数 `execute` 为非 0，则除了将清理函数栈中最顶层的函数移除之外，还会该清理函数。

尽管上面我们将 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 称之为函数，但它们是通过宏来实现，可展开为分别由 `{` 和 `}` 所包裹的语句序列，所以必须在与线程相同的作用域中以匹配对的形式使用，必须一一对应着来使用，譬如：

```
pthread_cleanup_push(cleanup, NULL);
```

```
pthread_cleanup_push(cleanup, NULL);
pthread_cleanup_push(cleanup, NULL);
.....
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
```

否则会编译报错, 如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
testApp.c: In function 'new_thread_start':
testApp.c:24:1: error: expected 'while' before 'int'
  int main(void)
  ^
testApp.c:46:1: error: expected declaration or statement at end of input
  }
  ^
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.8.1 编译报错

使用示例

示例代码 11.8.1 给出了一个使用线程清理函数的例子, 虽然例子并没有什么实际作用, 当它描述了其中所涉及到的清理机制。

示例代码 11.8.1 pthread_cleanup_push()注册线程清理函数

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

static void *new_thread_start(void *arg)
{
    printf("新线程--start run\n");
    pthread_cleanup_push(cleanup, "第 1 次调用");
    pthread_cleanup_push(cleanup, "第 2 次调用");
    pthread_cleanup_push(cleanup, "第 3 次调用");

    sleep(2);
```

```
pthread_exit((void *)0);    //线程终止

/* 为了与 pthread_cleanup_push 配对,不添加程序编译会通不过 */
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

主线程创建新线程之后,调用 `pthread_join()` 等待新线程终止;新线程调用 `pthread_cleanup_push()` 函数添加线程清理函数,调用了三次,但每次添加的都是同一个函数,只是传入的参数不同;清理函数添加完成,休眠一段时间之后,调用 `pthread_exit()` 退出。之后还调用了 3 次 `pthread_cleanup_pop()`, 在这里的目的仅仅只是为了与 `pthread_cleanup_push()` 配对使用,否则编译不通过。接下来编译运行:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--start run
cleanup: 第3次调用
cleanup: 第2次调用
cleanup: 第1次调用
新线程终止, code=0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.8.2 测试结果

从打印结果可知, 先添加到线程清理函数栈中的函数会后被执行, 添加顺序与执行顺序相反。

将新线程中调用的 `pthread_exit()` 替换为 `return`, 在进行测试, 发现并不会执行清理函数。

有时在线程功能设计中, 线程清理函数并不一定需要在线程退出时才执行, 譬如当完成某一个步骤之后, 就需要执行线程清理函数, 此时我们可以调用 `pthread_cleanup_pop()` 并传入非 0 参数, 来手动执行线程清理函数, 示例代码如下所示:

示例代码 11.8.2 手动执行线程清理函数

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

static void *new_thread_start(void *arg)
{
    printf("新线程--start run\n");
    pthread_cleanup_push(cleanup, "第 1 次调用");
    pthread_cleanup_push(cleanup, "第 2 次调用");
    pthread_cleanup_push(cleanup, "第 3 次调用");

    pthread_cleanup_pop(1);    //执行最顶层的清理函数
    printf("~~~~~\n");
    sleep(2);
    pthread_exit((void *)0);    //线程终止

    /* 为了与 pthread_cleanup_push 配对 */
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
}
```

```
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

上述代码中, 在新线程调用 `pthread_exit()` 之前, 先调用 `pthread_cleanup_pop(1)` 手动运行了最顶层的清理函数, 并将其从栈中移除, 测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程 --start run
cleanup: 第3次调用
~~~~~
cleanup: 第2次调用
cleanup: 第1次调用
新线程终止, code=0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.8.3 测试结果

从打印结果可知, 调用 `pthread_cleanup_pop(1)` 执行了最后一次注册的清理函数, 调用 `pthread_exit()` 退出线程时执行了 2 次清理函数, 因为前面调用 `pthread_cleanup_pop()` 已经将顶层的清理函数移除栈中了, 自然在退出时就不会再执行了。

11.9 线程属性

如前所述, 调用 `pthread_create()` 创建线程, 可对新建线程的各种属性进行设置。在 Linux 下, 使用 `pthread_attr_t` 数据类型定义线程的所有属性, 本书并不打算详细讨论这些属性, 以介绍为主, 简单地了解下线程属性。

调用 `pthread_create()` 创建线程时, 参数 `attr` 设置为 `NULL`, 表示使用属性的默认值创建线程。如果不使用默认值, 参数 `attr` 必须要指向一个 `pthread_attr_t` 对象, 而不能使用 `NULL`。当定义 `pthread_attr_t` 对象之后, 需要使用 `pthread_attr_init()` 函数对该对象进行初始化操作, 当对象不再使用时, 需要使用 `pthread_attr_destroy()` 函数将其销毁, 函数原型如下所示:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

使用这些函数需要包含头文件 `<pthread.h>`, 参数 `attr` 指向一个 `pthread_attr_t` 对象, 即需要进行初始化的线程属性对象。在调用成功时返回 0, 失败将返回一个非 0 值的错误码。

调用 `pthread_attr_init()` 函数会将指定的 `pthread_attr_t` 对象中定义的各种线程属性初始化为它们各自对应的默认值。

`pthread_attr_t` 数据结构中包含的属性比较多, 本小节并不会一一指出, 可能比较关注属性包括: 线程栈的位置和大小、线程调度策略和优先级, 以及线程的分离状态属性等。Linux 为 `pthread_attr_t` 对象的每种属性提供了设置属性的接口以及获取属性的接口。

11.9.1 线程栈属性

每个线程都有自己的栈空间, `pthread_attr_t` 数据结构中定义了栈的起始地址以及栈大小, 调用函数 `pthread_attr_getstack()` 可以获取这些信息, 函数 `pthread_attr_setstack()` 对栈起始地址和栈大小进行设置, 其函数原型如下所示:

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *attr, void **stackaddr, size_t *stacksize);
```

使用这些函数需要包含头文件 `<pthread.h>`, 函数 `pthread_attr_getstack()`, 参数和返回值含义如下:

attr: 参数 `attr` 指向线程属性对象。

stackaddr: 调用 `pthread_attr_getstack()` 可获取栈起始地址, 并将起始地址信息保存在 `*stackaddr` 中;

stacksize: 调用 `pthread_attr_getstack()` 可获取栈大小, 并将栈大小信息保存在参数 `stacksize` 所指向的内存中;

返回值: 成功返回 0, 失败将返回一个非 0 值的错误码。

函数 `pthread_attr_setstack()`, 参数和返回值含义如下:

attr: 参数 `attr` 指向线程属性对象。

stackaddr: 设置栈起始地址为指定值。

stacksize: 设置栈大小为指定值;

返回值: 成功返回 0, 失败将返回一个非 0 值的错误码。

如果想单独获取或设置栈大小、栈起始地址, 可以使用下面这些函数:

```
#include <pthread.h>
```

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
```

使用示例

创建新的线程，将线程的栈大小设置为 4Kbyte。

示例代码 11.9.1 设置线程栈大小 pthread_attr_getstack()

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

static void *new_thread_start(void *arg)
{
    puts("Hello World!");
    return (void *)0;
}

int main(int argc, char *argv[])
{
    pthread_attr_t attr;
    size_t stacksize;
    pthread_t tid;
    int ret;

    /* 对 attr 对象进行初始化 */
    pthread_attr_init(&attr);

    /* 设置栈大小为 4K */
    pthread_attr_setstacksize(&attr, 4096);

    /* 创建新线程 */
    ret = pthread_create(&tid, &attr, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
}
```

```
    }

    /* 销毁 attr 对象 */
    pthread_attr_destroy(&attr);
    exit(0);
}
```

11.9.2 分离状态属性

前面介绍了线程分离的概念, 如果对现已创建的某个线程的终止状态不感兴趣, 可以使用 `pthread_detach()` 函数将其分离, 那么该线程在退出时, 操作系统会自动回收它所占用的资源。

如果我们在创建线程时就确定要将该线程分离, 可以修改 `pthread_attr_t` 结构中的 `detachstate` 线程属性, 让线程一开始运行就处于分离状态。调用函数 `pthread_attr_setdetachstate()` 设置 `detachstate` 线程属性, 调用 `pthread_attr_getdetachstate()` 获取 `detachstate` 线程属性, 其函数原型如下所示:

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

需要包含头文件 `<pthread.h>`, 参数 `attr` 指向 `pthread_attr_t` 对象; 调用 `pthread_attr_setdetachstate()` 函数将 `detachstate` 线程属性设置为参数 `detachstate` 所指定的值, 参数 `detachstate` 取值如下:

- **PTHREAD_CREATE_DETACHED:** 新建线程一开始运行便处于分离状态, 以分离状态启动线程, 无法被其它线程调用 `pthread_join()` 回收, 线程结束后由操作系统收回其所占用的资源;
- **PTHREAD_CREATE_JOINABLE:** 这是 `detachstate` 线程属性的默认值, 正常启动线程, 可以被其它线程获取终止状态信息。

函数 `pthread_attr_getdetachstate()` 用于获取 `detachstate` 线程属性, 将 `detachstate` 线程属性保存在参数 `detachstate` 所指定的内存中。

使用示例

示例代码 11.9.2 给出了以分离状态启动线程的示例。

示例代码 11.9.2 以分离状态启动线程

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    puts("Hello World!");
    return (void *)0;
}

int main(int argc, char *argv[])
{
    pthread_attr_t attr;
```

```
pthread_t tid;
int ret;

/* 对 attr 对象进行初始化 */
pthread_attr_init(&attr);

/* 设置以分离状态启动线程 */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

/* 创建新线程 */
ret = pthread_create(&tid, &attr, new_thread_start, NULL);
if (ret) {
    fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
    exit(-1);
}

sleep(1);

/* 销毁 attr 对象 */
pthread_attr_destroy(&attr);
exit(0);
}
```

11.10 线程安全

当我们编写的程序是一个多线程应用程序时,就不得不考虑到线程安全的问题,确保我们编写的程序是一个线程安全(thread-safe)的多线程应用程序,什么是线程安全以及如何保证线程安全?带着这些问题,本小节将讨论线程安全相关的话题。

Tips: 在阅读本小节内容之前,建议先阅读第十二章内容,这章内容原本计划是放在本小节内容之前的,但由于排版问题,不得不将其单独列为一章。

11.10.1 线程栈

进程中创建的每个线程都有自己的栈地址空间,将其称为线程栈。譬如主线程调用 pthread_create() 创建了一个新的线程,那么这个新的线程有它自己独立的栈地址空间、而主线程也有它自己独立的栈地址空间。通过 11.9.1 小节可知,在创建一个新的线程时,可以配置线程栈的大小以及起始地址,当然在大部分情况下,保持默认即可!

既然每个线程都有自己的栈地址空间,那么每个线程运行过程中所定义的自动变量(局部变量)都是分配在自己的线程栈中的,它们不会相互干扰。在示例代码 11.10.1 中,主线程创建了 5 个新的线程,这 5 个线程使用同一个 start 函数 new_thread,该函数中定义了局部变量 number 和 tid 以及 arg 参数,意味着这 5 个线程的线程栈中都各自为这些变量分配了内存空间,任何一个线程修改了 number 或 tid 都不会影响其它线程。

示例代码 11.10.1 线程栈示例

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <pthread.h>

static void *new_thread(void *arg)
{
    int number = *((int *)arg);
    unsigned long int tid = pthread_self();
    printf("当前为<%d>号线程, 线程 ID<%lu>\n", number, tid);
    return (void *)0;
}

static int nums[5] = {0, 1, 2, 3, 4};

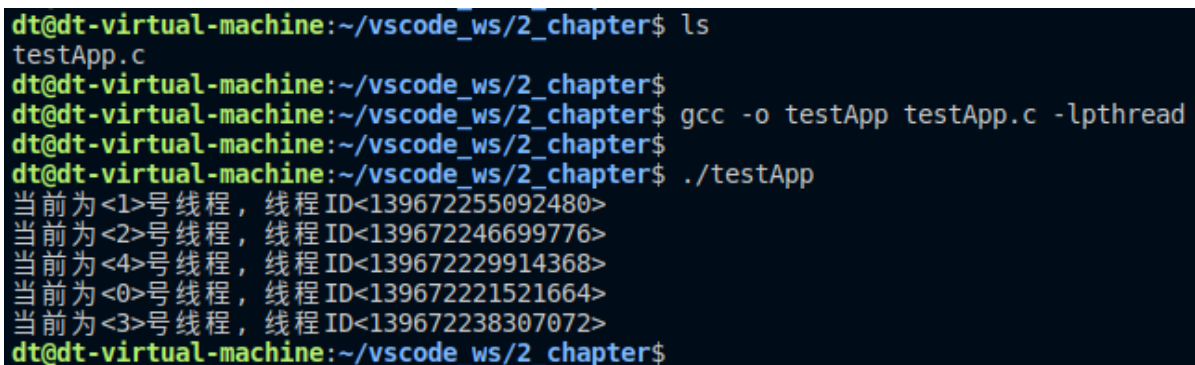
int main(int argc, char *argv[])
{
    pthread_t tid[5];
    int j;

    /* 创建 5 个线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j], NULL, new_thread, &nums[j]);

    /* 等待线程结束 */
    for (j = 0; j < 5; j++)
        pthread_join(tid[j], NULL); //回收线程

    exit(0);
}
```

运行结果:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
当前为<1>号线程, 线程 ID<139672255092480>
当前为<2>号线程, 线程 ID<139672246699776>
当前为<4>号线程, 线程 ID<139672229914368>
当前为<0>号线程, 线程 ID<139672221521664>
当前为<3>号线程, 线程 ID<139672238307072>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.1 测试结果

11.10.2 可重入函数

要解释可重入 (Reentrant) 函数为何物, 首先需要区分单线程程序和多线程程序。本章开头部分已向各位读者进行了详细介绍, 单线程程序只有一条执行流 (一个线程就是一条执行流), 贯穿程序始终; 而对于多线程程序而言, 同一进程却存在多条独立、并发的执行流。

进程中执行流的数量除了与线程有关之外,与信号处理也有关联。因为信号是异步的,进程可能会在其运行过程中的任何时间点收到信号,进而跳转、执行信号处理函数,从而在一个单线程进程(包含信号处理)中形成了两条(即主程序和信号处理函数)独立的执行流。

接下来再来介绍什么是可重入函数,如果一个函数被同一进程的多个不同的执行流同时调用,每次函数调用总是能产生正确的结果(或者叫产生预期的结果),把这样的函数就称为可重入函数。

Tips: 上面所说的同时指的是宏观上同时调用,实质上也就是该函数被多个执行流并发/并行调用,无特别说明,本章内容所提到的同时均指宏观上的概念。

重入指的是同一个函数被不同执行流调用,前一个执行流还没有执行完该函数、另一个执行流又开始调用该函数了,其实就是同一个函数被多个执行流并发/并行调用,在宏观角度上理解指的就是被多个执行流同时调用。

看到这里大家可能会有点不解,我们使用示例进行讲解。示例代码 11.10.2 是一个单线程与信号处理关联的程序。main()函数中调用 signal()函数为 SIGINT 信号注册了一个信号处理函数 sig_handler,信号处理函数 sig_handler 会调用 func 函数;main()函数最终会进入到一个循环中,循环调用 func()。

示例代码 11.10.2 信号与可重入问题

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void func(void)
{
    /*..... */
}

static void sig_handler(int sig)
{
    func();
}

int main(int argc, char *argv[])
{
    sig_t ret = NULL;

    ret = signal(SIGINT, (sig_t)sig_handler);
    if (SIG_ERR == ret) {
        perror("signal error");
        exit(-1);
    }

    /* 死循环 */
    for (;;)
        func();

    exit(0);
}
```


当 main()函数正在执行 func()函数代码, 此时进程收到了 SIGINT 信号, 便会打断当前正常执行流程、跳转到 sig_handler()函数执行, 进而调用 func、执行 func()函数代码; 这里就出现了主程序与信号处理函数并发调用 func()的情况, 示意图如下所示:

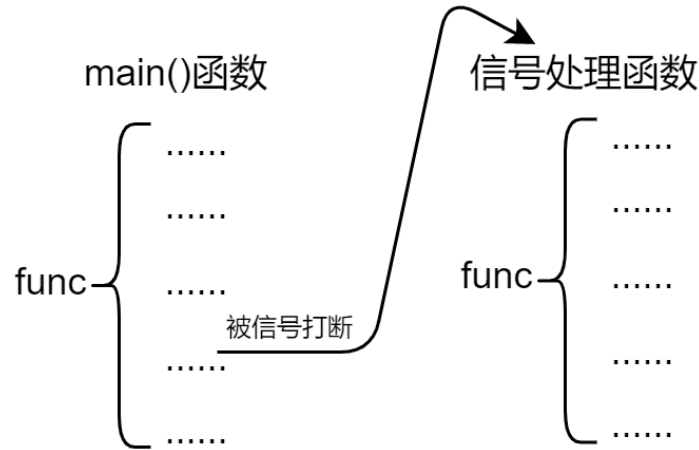


图 11.10.2 被信号打断

在信号处理函数中, 执行完 func()之后, 信号处理函数退出、返回到主程序流程, 也就是被信号打断的位置处继续运行。如果每次出现这种情况执行 func()函数都能产生正确的结果, 那么 func()函数就是一个可重入函数。

接着再来看看在多线程环境下, 示例代码 11.10.1 是一个多线程程序, 主线程调用 pthread_create()函数创建了 5 个新的线程, 这 5 个线程使用同一个入口函数 new_thread; 所以它们执行的代码是一样的, 除了参数 arg 不同之外; 在这种情况下, 这 5 个线程中的多个线程就可能会出现并发调用 pthread_self()函数的情况。

以上举例说明了函数被多个执行流同时调用的两种情况:

- 在一个含有信号处理的程序当中, 主程序正执行函数 func(), 此时进程接收到信号, 主程序被打断, 跳转到信号处理函数中执行, 信号处理函数中也调用了 func()。
- 在多线程环境下, 多个线程并发调用同一个函数。

所以由此可知, 在多线程环境以及信号处理有关应用程序中, 需要注意不可重入函数的问题, 如果多条执行流同时调用一个不可重入函数则可能会得不到预期的结果、甚至有可能导致程序崩溃! 不止是在应用程序中, 在一个包含了中断处理的裸机应用程序中亦是如此! 所以不可重入函数通常存在着一定的安全隐患。

可重入函数的分类

笔者认为可重入函数可以分为两类:

- **绝对的可重入函数:** 所谓绝对, 指的是该函数不管如何调用, 都刚断言它是可重入的, 都能得到预期的结果。
- **带条件的可重入函数:** 指的是在满足某个/某些条件的情况下, 可以断言该函数是可重入的, 不管怎么调用都能得到预期的结果。

绝对可重入函数

笔者查阅过很多的书籍以及网络文章, 并未发现有提出过这种分类, 所以这完全是笔者个人对此的一个理解, 首先来看一下绝对可重入函数的一个例子, 如下所示:

函数 func()就是一个标准的绝对可重入函数:

```
static int func(int a)
```

```
{
    int local;
    int j;

    for (local = 0, j = 0; j < 5; j++) {
        local += a * a;
        a += 2;
    }

    return local;
}
```

该函数内操作的变量均是函数内部定义的自动变量（局部变量），每次调用函数，都会在栈内存空间为局部变量分配内存，当函数调用结束返回时、再由系统回收这些变量占用的栈内存，所以局部变量生命周期只限于函数执行期间。

除此之外，该函数的参数和返回值均是值类型、而并非是引用类型（就是指针）。

如果多条执行流同时调用函数 `func()`，那必然会在栈空间中存在多份局部变量，每条执行流操作各自的局部变量，相互不影响，所以即使函数同时被调用，依然每次都能得到正确的结果。所以上面列举的函数 `func()` 就是一个非常标准的绝对可重入函数，函数内部仅操作了函数内定义的局部变量，除了使用栈上的变量以外不依赖于任何环境变量，这样的函数就是 `purecode`（纯代码）可重入，可以允许该函数的多个副本同时在运行，由于它们使用的是分离的栈，所以不会相互干扰！

总结下绝对可重入函数的特点：

- 函数内所使用到的变量均为局部变量，换句话说，该函数内的操作的内存地址均为本地栈地址；
- 函数参数和返回值均是值类型；
- 函数内调用的其它函数也均是绝对可重入函数。

带条件的可重入函数

带条件的可重入函数通常需要满足一定的条件时才是可重入函数，我们来看一个不可重入函数的例子，如下所示：

```
static int glob = 0;

static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        local = glob;
        local++;
        glob = local;
    }
}
```

当多个执行流同时调用该函数，全局变量 `glob` 的最终值将不得而知，最终可能会得不到正确的结果，因为全局变量 `glob` 将成为多个线程间的共享数据，它们都会对 `glob` 变量进行读写操作、会导致数据不一致

的问题, 关于这个问题在 12.1 小节中给大家做了详细说明。这个函数就是典型的不可重入函数, 函数运行需要读取、修改全局变量 `glob`, 该变量并非在函数自己的栈上, 意味着该函数运行依赖于外部环境变量。

但如果对上面的函数进行修改, 函数 `func()`内仅读取全局变量 `glob` 的值, 而不更改它的值:

```
static int glob = 0;

static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        local = glob;
        local++;
        printf("local=%d\n", local);
    }
}
```

修改完之后, 函数 `func()`内仅读取了变量 `glob`, 而并未更改 `glob` 的值, 那么此时函数 `func()`就是一个可重入函数了; 但是这里需要注意, 它需要满足一个条件, 这个条件就是: 当多个执行流同时调用函数 `func()`时, 全局变量 `glob` 的值绝对不会在其它某个地方被更改; 譬如线程 1 和线程 2 同时调用了函数 `func()`, 但是另一个线程 3 在线程 1 和线程 2 同时调用了函数 `func()`的时候, 可能会发生更改变量 `glob` 值的情况, 如果是这样, 那么函数 `func()`依然是不可重入函数。这就是有条件的可重入函数的概念, 这通常需要程序员本身去规避这类问题, 标准 C 语言函数库中也存在很多这类带条件的可重入函数, 后面给大家看一下。

再来看一个例子:

```
static void func(int *arg)
{
    int local = *arg;
    int j;

    for (j = 0; j < 10; j++)
        local++;

    *arg = local;
}
```

这是一个参数为引用类型的函数, 传入了一个指针, 并在函数内部读写该指针所指向的内存地址, 该函数是一个可重入函数, 但同样需要满足一定的条件; 如果多个执行流同时调用该函数时, 所传入的指针是共享变量的地址, 那么在这种情况下, 最终可能得不到预期的结果; 因为在这种情况下, 函数 `func()`所读写的便是多个执行流的共享数据, 会出现数据不一致的情况, 所以是不安全的。

但如果每个执行流所传入的指针是其本地变量(局部变量)对应的地址, 那就是没有问题的, 所以呢, 这个函数就是一个带条件的可重入函数。

总结

相信笔者列举了这么多例子, 大家应该明白了什么是可重入函数以及绝对可重入函数和带条件的可重入函数的区别, 还有很多的例子这里就不再一一列举了, 相信通过笔者的介绍大家应该知道如何去判断它们了。

很多的 C 库函数有两个版本: 可重入版本和不可重入版本, 可重入版本函数其名称后面加上了“_r”, 用于表明该函数是一个可重入函数; 而不可重入版本函数其名称后面没有“_r”, 前面章节内容中也已经遇到过很多次了, 譬如 `asctime()/asctime_r()`、`ctime()/ctime_r()`、`localtime()/localtime_r()`等。

通过 man 手册可以查询到它们“ATTRIBUTES”信息, 譬如执行“`man 3 ctime`”, 在帮助页面上往下翻便可以找到, 如下所示:

ATTRIBUTES		
For an explanation of the terms used in this section, see <code>attributes(7)</code> .		
Interface	Attribute	Value
<code>asctime()</code>	Thread safety	MT-Unsafe race:asctime locale
<code>asctime_r()</code>	Thread safety	MT-Safe locale
<code>ctime()</code>	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale
<code>ctime_r()</code> , <code>gmtime_r()</code> , <code>localtime_r()</code> , <code>mktime()</code>	Thread safety	MT-Safe env locale
<code>gmtime()</code> , <code>localtime()</code>	Thread safety	MT-Unsafe race:tmbuf env locale

图 11.10.3 `asctime()/asctime_r()`函数的 ATTRIBUTES 信息

可以看到上图中有些函数 Value 这栏会显示 MT-Unsafe、而有些函数显示的却是 MT-Safe。MT 指的是 multithreaded (多线程), 所以 MT-Unsafe 就是多线程不安全、MT-Safe 指的是多线程安全, 通常习惯上将 MT-Safe 和 MT-Unsafe 称为线程安全或线程不安全。

Value 值为 MT-Safe 修饰的函数表示该函数是一个线程安全函数, 使用 MT-Unsafe 修饰的函数表示它是一个线程不安全函数, 下一小节会给大家介绍什么是线程安全函数。从上图可以看出, `asctime_r()/ctime_r()/gmtime_r()/localtime_r()`这些可重入函数都是线程安全函数, 但这些函数都是带条件的可重入函数, 可以发现在 MT-Safe 标签后面会携带诸如 env 或 locale 之类的标签, 这其实就表示该函数需要在满足 env 或 locale 条件的情况下才是可重入函数; 如果是绝对可重入函数, MT-Safe 标签后面不会携带任何标签, 譬如数学库函数 `sqrt`:

ATTRIBUTES		
For an explanation of the terms used in this section, see <code>attributes(7)</code> .		
Interface	Attribute	Value
<code>sqrt()</code> , <code>sqrtf()</code> , <code>sqrtl()</code>	Thread safety	MT-Safe

图 11.10.4 `sqrt` 函数的 ATTRIBUTES 信息

诸如 env 或 locale 等标签, 可以通过 man 手册进行查询, 命令为“`man 7 attributes`”, 这文档里边的内容反正笔者是没太看懂, 不知所云; 但是经过我的对比 env 或 locale 这两个标签还是很容易理解的。这两个标签在 man 测试里边出现的频率相对于其它的标签要大, 这里笔者就简单地提一下:

- **env:** 这个标签指的是该函数内部会读取进程的某个/某些环境变量, 譬如 `getenv()`函数, 前面也给大家介绍过, 进程的环境变量其实就是程序的一个全局变量, 前面也讲了, 对于这类读取(但没更改)了全局变量的可重入函数应该要满足的条件, 这里就不再重述了;
- **local:** local 指的是本地, 很容易理解, 通常该类函数传入了指针, 前面也提到了传入了指针的可重入函数应该要满足什么样的条件才是可重入的, 这里也不再重述!

本小节内容写得有点多了, 笔者觉得讲的是比较清楚了, 下小节给大家介绍线程安全函数。

11.10.3 线程安全函数

了解了可重入函数之后, 再来看看线程安全函数。

一个函数被多个线程(其实也是多个执行流, 但是不包括由信号处理函数所产生的执行流)同时调用时, 它总会一直产生正确的结果, 把这样的函数称为线程安全函数。线程安全函数包括可重入函数, 可重入函数是线程安全函数的一个真子集, 也就是说可重入函数一定是线程安全函数, 但线程安全函数不一定是可重入函数, 它们之间的关系如下:

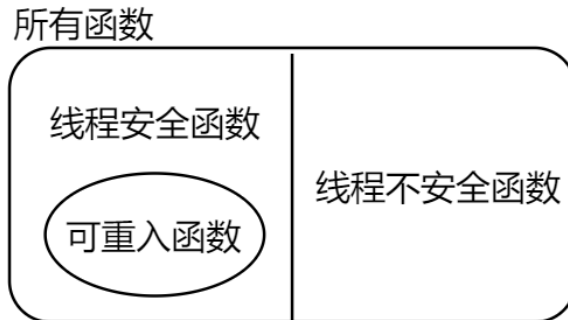


图 11.10.5 线程安全函数与可重入函数

譬如下面这个函数是一个不可重入函数, 同样也是一个线程不安全函数(上小节的最后一个例子):

```
static int glob = 0;

static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        local = glob;
        local++;
        glob = local;
    }
}
```

如果对该函数进行修改, 使用线程同步技术(譬如互斥锁)对共享变量 `glob` 的访问进行保护, 在读写该变量之前先上锁、读写完成之后在解锁。这样, 该函数就变成了一个线程安全函数, 但是它依然不是可重入函数, 因为该函数更改了外部全局变量的值。

可重入函数只是单纯从语言语法角度分析它的可重入性质, 不涉及到一些具体的实现机制, 譬如线程同步技术, 这是判断可重入函数和线程安全函数的区别, 因为你单从概念上去分析的话, 其实可以发现可重入函数和线程安全函数好像说的是同一个东西, “一个函数被多个线程同时调用时, 它总会一直产生正确的结果, 把这样的函数称为线程安全函数”, 多个线程指的就是多个执行流(不包括信号处理函数执行流), 所以从这里看跟可重入函数的概念是很相似的。

判断一个函数是否为线程安全函数的方法是, 该函数被多个线程同时调用是否总能产生正确的结果, 如果每次都能产生预期的结果则表示该函数是一个线程安全函数。判断一个函数是否为可重入函数的方法是, 从语言语法角度分析, 该函数被多个执行流同时调用是否总能产生正确的结果, 如果每次都能产生预期的结果则表示该函数是一个可重入函数。

POSIX.1-2001 和 POSIX.1-2008 标准中规定的所有函数都必须是线程安全函数, 但以下函数除外:

asctime()	basename()	catgets()	crypt()
ctermid()	ctime()	dbm_clearerr()	dbm_close()
dbm_delete()	dbm_error()	dbm_fetch()	dbm_firstkey()
dbm_nextkey()	dbm_open()	dbm_store()	dirname()
dlderror()	drand48()	ecvt()	encrypt()
endgrent()	endpwent()	endutxent()	fcvt()
ftw()	gcvt()	getc_unlocked()	getchar_unlocked()
getdate()	getenv()	getgrent()	getgrgid()
getgrnam()	gethostbyaddr()	gethostbyname()	gethostent()
getlogin()	getnetbyaddr()	getnetbyname()	getnetent()
getopt()	getprotobyname()	getprotobyname()	getprotoent()
getpwent()	getpwnam()	getpwuid()	getservbyname()
getservbyport()	getservent()	getutxent()	getutxid()
getutxline()	gmtime()	hcreate()	hdestroy()
hsearch()	inet_ntoa()	l64a()	lgamma()
lgammaf()	lgammal()	localeconv()	localtime()
lrand48()	mrnd48()	nftw()	nl_langinfo()
ptsname()	putc_unlocked()	putchar_unlocked()	putenv()
pututxline()	rand()	readdir()	setenv()
setgrent()	setkey()	setpwent()	setutxent()
strerror()	strsignal()	strtok()	system()
tmpnam()	ttyname()	unsetenv()	wcrtomb()
wcsrtombs()	wcstombs()	wctomb()	

表 11.10.1 POSIX.1-2001 和 POSIX.1-2008 中列出的线程不安全函数

以上所列举出的这些函数被认为是线程不安全函数，大家也可以通过 man 手册查询到这些函数，"man 7 pthreads"，如下所示：

```

Thread-safe functions
A thread-safe function is one that can be safely (i.e., it will deliver the same results regardless of whether it is) called from multiple threads.

POSIX.1-2001 and POSIX.1-2008 require that all functions specified in the standard shall be thread-safe, except for the following functions:

asctime()
basename()
catgets()
crypt()
ctermid() if passed a non-NULL argument
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlderror()
drand48()
ecvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
encrypt()
endgrent()
endpwent()
endutxent()
fcvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
ftw()

```

图 11.10.6 通过 man 手册查询到线程不安全函数

如果想确认某个函数是不是线程安全函数可以

上小节给大家提到过，man 手册可以查看库函数的 ATTRIBUTES 信息，如果函数被标记为 MT-Safe，则表示该函数是一个线程安全函数，如果被标记为 MT-Unsafe，则意味着该函数是一个非线程安全函数，对

于非线程安全函数, 在多线程编程环境下尤其要注意, 如果某函数可能会被多个线程同时调用时, 该函数不能是非线程安全函数, 一定要是线程安全函数, 否则将会出现意想不到的结果、甚至使得整个程序崩溃!

对于一个中大型的多线程应用程序项目来说, 能够保证整个程序的安全性, 这是非常重要的, 程序员必须要正确对待线程安全以及信号处理这类在多线程环境下敏感的问题, 这通常对程序员提出了更高的要求。

11.10.4 一次性初始化

在多线程编程环境下, 有些代码段只需要执行一次, 譬如一些初始化相关的代码段, 通常比较容易想到的就是将其放在 `main()` 主函数进行初始化, 这样也就是意味着该段代码只在主线程中被调用, 只执行过一次。大家想一下这样的问题: 当你写了一个 C 函数 `func()`, 该函数可能会被多个线程调用, 并且该函数中有一段初始化代码, 该段代码只能被执行一次 (无论哪个线程执行都可以)、如果执行多次会出现问题, 如下所示:

```
static void func(void)
{
    /* 只能执行一次的代码段 */
    init_once();
    /*******/

    .....

    .....
}
```

大家可能会问, 怎么会有这样的需求呢? 当然有, 譬如下小节将要介绍的线程特有数据就需要有这样的需求, 那我们如何去保证这段代码只能被执行一次呢 (被进程中的任一线程执行都可以)? 本小节向大家介绍 `pthread_once()` 函数, 该函数原型如下所示:

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

在多线程编程环境下, 尽管 `pthread_once()` 调用会出现在多个线程中, 但该函数会保证 `init_routine()` 函数仅执行一次, 究竟在哪个线程中执行是不定的, 是由内核调度来决定。函数参数和返回值含义如下:

once_control: 这是一个 `pthread_once_t` 类型指针, 在调用 `pthread_once()` 函数之前, 我们需要定义了一个 `pthread_once_t` 类型的静态变量, 调用 `pthread_once()` 时参数 `once_control` 指向该变量。通常在定义变量时会使用 `PTHREAD_ONCE_INIT` 宏对其进行初始化, 譬如:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

init_routine: 一个函数指针, 参数 `init_routine` 所指向的函数就是要求只能被执行一次的代码段, `pthread_once()` 函数内部会调用 `init_routine()`, 即使 `pthread_once()` 函数会被多次执行, 但它能保证 `init_routine()` 仅被执行一次。

返回值: 调用成功返回 0; 失败则返回错误编码以指示错误原因。

如果参数 `once_control` 指向的 `pthread_once_t` 类型变量, 其初值不是 `PTHREAD_ONCE_INIT`, `pthread_once()` 的行为将是不正常的; `PTHREAD_ONCE_INIT` 宏在 `<pthread.h>` 头文件中定义。

如果在一个线程调用 `pthread_once()` 时, 另外一个线程也调用了 `pthread_once`, 则该线程将会被阻塞等待, 直到第一个完成初始化后返回。换言之, 当调用 `pthread_once` 成功返回时, 调用总是能够肯定所有的状态已经初始化完成了。

使用示例

接下来我们测试下, 当 `pthread_once()` 被多次调用时, `init_routine()` 函数是不是只会被执行一次, 示例代码如下所示:

示例代码 11.10.3 pthread_once() 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static pthread_once_t once = PTHREAD_ONCE_INIT;

static void initialize_once(void)
{
    printf("initialize_once 被执行: 线程 ID<%lu>\n", pthread_self());
}

static void func(void)
{
    pthread_once(&once, initialize_once); // 执行一次性初始化函数
    printf("函数 func 执行完毕.\n");
}

static void *thread_start(void *arg)
{
    printf("线程%d 被创建: 线程 ID<%lu>\n", *((int *)arg), pthread_self());
    func(); // 调用函数 func
    pthread_exit(NULL); // 线程终止
}

static int nums[5] = {0, 1, 2, 3, 4};

int main(void)
{
    pthread_t tid[5];
    int j;

    /* 创建 5 个线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j], NULL, thread_start, &nums[j]);

    /* 等待线程结束 */
    for (j = 0; j < 5; j++)
        pthread_join(tid[j], NULL); // 回收线程
}
```



```
exit(0);  
}
```

程序中调用 `pthread_create()` 创建了 5 个子线程, 新线程的入口函数均为 `thread_start()`, `thread_start()` 函数会调用 `func()`, 并在 `func()` 函数调用 `pthread_once()`, 需要执行的一次性初始化函数为 `initialize_once()`, 换言之, `pthread_once()` 函数会被执行 5 次, 每个子线程各自执行一次。

编译运行:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp  
线程 1 被创建: 线程 ID<140621723465472>  
initialize once 被执行: 线程 ID<140621723465472> ←  
线程 3 被创建: 线程 ID<140621706680064>  
函数 func 执行完毕.  
线程 0 被创建: 线程 ID<140621698287360>  
函数 func 执行完毕.  
线程 2 被创建: 线程 ID<140621715072768>  
函数 func 执行完毕.  
线程 0 被创建: 线程 ID<140621689894656>  
函数 func 执行完毕.  
函数 func 执行完毕.  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.7 测试结果

从打印信息可知, `initialize_once()` 函数确实只被执行了一次, 也就是被编号为 1 的线程所执行, 其它线程均未执行该函数。

11.10.5 线程特有数据

线程特有数据也称为线程私有数据, 简单点说, 就是为每个调用线程分别维护一份变量的副本 (copy), 每个线程通过特有数据键 (key) 访问时, 这个特有数据键都会获取到本线程绑定的变量副本。这样就可以避免变量成为多个线程间的共享数据。

C 库中有很多函数都是非线程安全函数, 非线程安全函数在多线程环境下, 被多个线程同时调用时将会发生意想不到的结果, 得不到预期的结果。譬如很多库函数都会返回一个字符串指针, 譬如 `asctime()`、`ctime()`、`localtime()` 等, 返回出来的字符串可以被调用线程直接使用, 但该字符串缓冲区通常是这些函数内部所维护的静态数组或者是某个全局数组 (这里笔者只是猜测, 具体是哪一种我也不清楚, 没有翻看这些函数内部的实现)。

既然如此, 多次调用这些函数返回的字符串其实指向的是同一个缓冲区, 每次调用都会刷新缓冲区中的数据。这些函数是非线程安全的, 譬如当 `ctime()` 被多个线程同时调用时, 返回的字符串中的数据可能是混乱的, 因为某一线程调用它时, 缓冲区中的数据可能被另一个调用线程修改了。针对这些非线程安全函数, 可以使用线程特有数据将其变为线程安全函数, 线程特有数据通常会在编写一些库函数的时使用到, 后面我们会演示如何使用线程特有数据。

线程特有数据的核心思想其实非常简单, 就是为每一个调用线程 (调用某函数的线程, 该函数就是我们通过线程特有数据将其实现为线程安全的函数) 分配属于该线程的私有数据区, 为每个调用线程分别维护一份变量的副本。

线程特有数据主要涉及到 3 个函数: `pthread_key_create()`、`pthread_setspecific()` 以及 `pthread_getspecific()`, 接下来一一向大家进行介绍。

`pthread_key_create()` 函数

在为线程分配私有数据区之前, 需要调用 `pthread_key_create()` 函数创建一个特有数据键 (key), 并且只需要在首个调用的线程中创建一次即可, 所以通常会使用到上小节所学习的 `pthread_once()` 函数。`pthread_key_create()` 函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

使用该函数需要包含头文件 `<pthread.h>`。

函数参数和返回值含义如下:

key: 调用该函数会创建一个特有数据键, 并通过参数 `key` 所指向的缓冲区返回给调用者, 参数 `key` 是一个 `pthread_key_t` 类型的指针, 可以把 `pthread_key_t` 称为 `key` 类型。调用 `pthread_key_create()` 之前, 需要定义一个 `pthread_key_t` 类型变量, 调用 `pthread_key_create()` 时参数 `key` 指向 `pthread_key_t` 类型变量。

destructor: 参数 `destructor` 是一个函数指针, 指向一个自定义的函数, 其格式如下:

```
void destructor(void *value)
```

```
{  
    /* code */  
}
```

调用 `pthread_key_create()` 函数允许调用者指定一个自定义的解构函数 (类似于 C++ 中的析构函数), 使用参数 `destructor` 指向该函数; 该函数通常用于释放与特有数据键关联的线程私有数据区占用的内存空间, 当使用线程特有数据的线程终止时, `destructor()` 函数会被自动调用。

返回值: 成功返回 0; 失败将返回一个错误编号以指示错误原因, 返回的错误编号其实就是全局变量 `errno`, 可以使用诸如 `strerror()` 函数查看其错误字符串信息。

pthread_setspecific() 函数

调用 `pthread_key_create()` 函数创建特有数据键 (key) 后通常需要为调用线程分配私有数据缓冲区, 譬如通过 `malloc()` (或类似函数) 申请堆内存, 每个调用线程分配一次, 且只会在线程初次调用此函数时分配。为线程分配私有数据缓冲区之后, 通常需要调用 `pthread_setspecific()` 函数, `pthread_setspecific()` 函数其实完成了这样的操作: 首先保存指向线程私有数据缓冲区的指针, 并将其与特有数据键以及当前调用线程关联起来; 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

函数参数和返回值含义如下:

key: `pthread_key_t` 类型变量, 参数 `key` 应赋值为调用 `pthread_key_create()` 函数时创建的特有数据键, 也就是 `pthread_key_create()` 函数的参数 `key` 所指向的 `pthread_key_t` 变量。

value: 参数 `value` 是一个 `void` 类型的指针, 指向由调用者分配的一块内存, 作为线程的私有数据缓冲区, 当线程终止时, 会自动调用参数 `key` 指定的特有数据键对应的解构函数来释放这一块动态申请的内存空间。

返回值: 调用成功返回 0; 失败将返回一个错误编码, 可以使用诸如 `strerror()` 函数查看其错误字符串信息。

pthread_getspecific() 函数

调用 `pthread_setspecific()` 函数将线程私有数据缓冲区与调用线程以及特有数据键关联之后, 便可以使用 `pthread_getspecific()` 函数来获取调用线程的私有数据区了。其函数原型如下所示:

```
#include <pthread.h>
```

```
void *pthread_getspecific(pthread_key_t key);
```

参数 `key` 应赋值为调用 `pthread_key_create()` 函数时创建的特有数据键, 也就是 `pthread_key_create()` 函数的参数 `key` 指向的 `pthread_key_t` 变量。

`pthread_getspecific()` 函数应返回当前调用线程关联到特有数据键的私有数据缓冲区, 返回值是一个指针, 指向该缓冲区。如果当前调用线程并没有设置线程私有数据缓冲区与特有数据键进行关联, 则返回值应为 `NULL`, 函数中可以利用这一点来判断当前调用线程是否为初次调用该函数, 如果是初次调用, 则必须为该线程分配私有数据缓冲区。

pthread_key_delete()函数

除了以上介绍的三个函数外, 如果需要删除一个特有数据键 (`key`) 可以使用函数 `pthread_key_delete()`, `pthread_key_delete()` 函数删除先前由 `pthread_key_create()` 创建的键。其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_key_delete(pthread_key_t key);
```

参数 `key` 为要删除的键。函数调用成功返回 0, 失败将返回一个错误编号。

调用 `pthread_key_delete()` 函数将释放参数 `key` 指定的特有数据键, 可以供下一次调用 `pthread_key_create()` 时使用; 调用 `pthread_key_delete()` 时, 它并不将查当前是否有线程正在使用该键所关联的线程私有数据缓冲区, 所以它并不会触发键的解构函数, 也就不会释放键关联的线程私有数据区占用的内存资源, 并且调用 `pthread_key_delete()` 后, 当线程终止时也不再执行键的解构函数。所以, 通常在调用 `pthread_key_delete()` 之前, 必须确保以下条件:

- 所有线程已经释放了私有数据区 (显式调用解构函数或线程终止)。
- 参数 `key` 指定的特有数据键将不再使用。

任何在调用 `pthread_key_delete()` 之后使用键的操作都会导致未定义的行为, 譬如调用 `pthread_setspecific()` 或 `pthread_getspecific()` 将会以错误形式返回。

使用示例

接下来编写一个使用线程特有数据的例子, 很多书籍上都会使用 `strerror()` 函数作为例子, 这个函数曾在 3.2 小节向大家介绍过, 通过 `man` 手册查询到 `strerror()` 函数是一个非线程安全函数, 其实它有对应的可重入版本 `strerror_r()`, 可重入版本 `strerror_r()` 函数则是一个线程安全函数。

这里暂且不管 `strerror_r()` 函数, 我们来聊一聊 `strerror()` 函数, 函数内部的实现方式, 这里简单地提一下: 调用 `strerror()` 函数, 需要传入一个错误编号, 错误编号赋值给参数 `errno`, 在 Linux 系统中, 每一个错误编号都会对应一个字符串, 用于描述该错误, `strerror()` 函数会根据传入的 `errno` 找到对应的字符串, 返回指向该字符串的指针。

事实上, 在 Linux 的实现中, 标准 C 语言函数库 (`glibc`) 提供的 `strerror()` 函数是线程安全的, 但在 `man` 手册中记录它是一个非线程安全函数, 笔者猜测可能在某些操作系统的 C 语言函数库实现中, 该函数是非线程安全函数的; 但在 `glibc` 库中, 它确实是线程安全函数, 为此笔者还特意去查看了 `glibc` 库中 `strerror` 函数的源码, 证实了这一点, 这里大家一定要注意。

以下是 `strerror()` 函数以非线程安全方式实现的一种写法 (具体的写法不止这一种, 这里只是以此为列):

示例代码 11.10.4 `strerror()` 函数以非线程安全方式实现的一种写法

```
#define _GNU_SOURCE
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```

#define MAX_ERROR_LEN 256
static char buf[MAX_ERROR_LEN];

static char *strerror(int errnum)
{
    if (errnum < 0 || errnum >= _sys_nerr || NULL == _sys_errlist[errnum])
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errnum);
    else {
        strncpy(buf, _sys_errlist[errnum], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';//终止字符
    }

    return buf;
}

```

再次说明, glibc 库中 `strerror()` 是线程安全函数, 本文为了向大家介绍/使用线程特有数据, 以非线程安全方式实现了 `strerror()` 函数。

首先在源码中需要定义 `_GNU_SOURCE` 宏, `_GNU_SOURCE` 宏在前面章节已有介绍, 这里不再重述! 源码中需要定义 `_GNU_SOURCE` 宏, 不然编译源码将会提示 `_sys_nerr` 和 `_sys_errlist` 找不到。该函数利用了 glibc 定义的一对全局变量: `_sys_errlist` 是一个指针数组, 其中的每一个元素指向一个与 `errno` 错误编号相匹配的描述性字符串; `_sys_nerr` 表示 `_sys_errlist` 数组中元素的个数。

可以看到该函数返回的字符串指针, 其实是一个静态数组, 当多个线程同时调用该函数时, 那么 `buf` 缓冲区中的数据将会出现混乱, 因为前一个调用线程拷贝到 `buf` 中的数据可能会被后一个调用线程重写覆盖等情况。

对此, 我们可以对示例代码 11.10.4 进行测试, 让多个线程都调用它, 看看测试结果, 测试代码如下:

示例代码 11.10.5 非线程安全版 `strerror` 测试

```

#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define MAX_ERROR_LEN 256
static char buf[MAX_ERROR_LEN];

/*****
 * 为了避免与库函数 strerror 重名
 * 这里将其改成 my_strerror
 *****/
static char *my_strerror(int errnum)
{
    if (errnum < 0 || errnum >= _sys_nerr || NULL == _sys_errlist[errnum])

```

```
    snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errnum);
else {
    strncpy(buf, _sys_errlist[errnum], MAX_ERROR_LEN - 1);
    buf[MAX_ERROR_LEN - 1] = '\0'; //终止字符
}

return buf;
}

static void *thread_start(void *arg)
{
    char *str = my_strerror(2); //获取错误编号为 2 的错误描述信息
    printf("子线程: str (%p) = %s\n", str, str);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    char *str = NULL;
    int ret;

    str = my_strerror(1); //获取错误编号为 1 的错误描述信息

    /* 创建子线程 */
    if (ret = pthread_create(&tid, NULL, thread_start, NULL)) {
        fprintf(stderr, "pthread_create error: %d\n", ret);
        exit(-1);
    }

    /* 等待回收子线程 */
    if (ret = pthread_join(tid, NULL)) {
        fprintf(stderr, "pthread_join error: %d\n", ret);
        exit(-1);
    }

    printf("主线程: str (%p) = %s\n", str, str);
    exit(0);
}
```

主线程首先调用 `my_strerror()` 获取到了编号为 1 的错误描述信息, 接着创建了一个子线程, 在子线程中调用 `my_strerror()` 获取编号为 2 的错误描述信息, 并将其打印出来, 包括字符串的地址值; 子线程结束后, 主线程也打印了之前获取到的错误描述信息。我们想看到的结果是, 主线程和子线程打印的错误描述信息是不一样的, 因为错误编号不同, 但上面的测试结果证实它们打印的结果是相同的:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -pthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子线程: str (0x601500) = No such file or directory
主线程: str (0x601500) = No such file or directory
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.8 测试结果

从以上测试结果可知,子线程和主线程锁获取到的错误描述信息是相同的,字符串指针指向的是同一个缓冲区;原因就在于,my_strerror()函数是一个非线程安全函数,函数内部修改了全局静态变量、并返回了它的指针,每一次调用访问的都是同一个静态变量,所以后一次调用会覆盖掉前一次调用的结果。

接下来我们使用本小节所介绍的线程特有数据技术对示例代码 11.10.4 中 strerror()函数进行修改,如下所示:

示例代码 11.10.6 使用线程特有数据实现线程安全的 strerror()函数

```
#define _GNU_SOURCE

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_ERROR_LEN 256

static pthread_once_t once = PTHREAD_ONCE_INIT;
static pthread_key_t strerror_key;

static void destructor(void *buf)
{
    free(buf); //释放内存
}

static void create_key(void)
{
    /* 创建一个键(key), 并且绑定键的解构函数 */
    if (pthread_key_create(&strerror_key, destructor))
        pthread_exit(NULL);
}

/*****
 * 对 strerror 函数重写
 * 使其变成为一个线程安全函数
 *****/

static char *strerror(int errnum)
```

```
{
    char *buf;

    /* 创建一个键(只执行一次 create_key) */
    if (pthread_once(&once, create_key))
        pthread_exit(NULL);

    /* 获取 */
    buf = pthread_getspecific(sterror_key);
    if (NULL == buf) { //首次调用 my_sterror 函数, 则需给调用线程分配线程私有数据
        buf = malloc(MAX_ERROR_LEN); //分配内存
        if (NULL == buf)
            pthread_exit(NULL);

        /* 保存缓冲区地址,与键、线程关联起来 */
        if (pthread_setspecific(sterror_key, buf))
            pthread_exit(NULL);
    }

    if (errno < 0 || errno >= _sys_nerr || NULL == _sys_errlist[errno])
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errno);
    else {
        strncpy(buf, _sys_errlist[errno], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0'; //终止字符
    }

    return buf;
}
```

改进版的 `sterror()` 所做的第一步是调用 `pthread_once()`, 以确保只会执行一次 `create_key()` 函数, 而在 `create_key()` 函数中便是调用 `pthread_key_create()` 创建了一个键、并绑定了相应的解构函数 `destructor()`, 解构函数用于释放与键关联的所有线程私有数据所占的内存空间。

接着, 函数 `sterror()` 调用 `pthread_getspecific()` 以获取该调用线程与键相关联的私有数据缓冲区地址, 如果返回为 `NULL`, 则表明该线程是首次调用 `sterror()` 函数, 因为函数会调用 `malloc()` 为其分配一个新的私有数据缓冲区, 并调用 `pthread_setspecific()` 来保存缓冲区地址、并与键以及该调用线程建立关联。如果 `pthread_getspecific()` 函数的返回值并不等于 `NULL`, 那么该值将指向以存在的私有数据缓冲区, 此缓冲区由之前对 `sterror()` 的调用所分配。

剩余部分代码与示例代码 11.10.4 非线性安全版的 `sterror()` 实现类似, 唯一的区别在于, `buf` 是线程特有数据的缓冲区地址, 而非全局的静态变量。

改进版的 `sterror` 就是一个线程安全函数, 编写一个线程安全函数当然要保证该函数中调用的其它函数也必须是线程安全的, 那如何确认自己调用的函数是线程安全函数呢? 其实非常简单, 前面也给大家介绍过, 譬如通过 `man` 手册查看函数的 `ATTRIBUTES` 描述信息, 或者查看 `man` 手册中记录的非线程安全函数列表 (执行 "`man 7 pthreads`" 命令查看)、进行对比。

Tips: 有时会发现 ATTRIBUTES 描述信息与非线程安全函数列表不一致, 譬如 ATTRIBUTES 描述信息中显示该函数是 MT-Unsafe (非线程安全函数) 标识的, 但是却没记录在非线程安全函数列表中, 此时我们应该以列表为准! 默认该函数是线程安全的。

大家可以去测试下改进版的 `strerror`, 这里笔者便不再给大家演示了, 需要注意的是, 在测试代码中定义的 `strerror` 函数其名字需要改成其它的名称, 避免与库函数 `strerror` 重名。

11.10.6 线程局部存储

通常情况下, 程序中定义的全局变量是进程中所有线程共享的, 所有线程都可以访问这些全局变量; 而线程局部存储在定义全局或静态变量时, 使用 `__thread` 修饰符修饰变量, 此时, 每个线程都会拥有一份对该变量的拷贝。线程局部存储中的变量将一直存在, 直至线程终止, 届时会自动释放这一存储。

线程局部存储的主要优点在于, 比线程特有数据的使用要简单。要创建线程局部变量, 只需简单地在全局或静态变量的声明中包含 `__thread` 修饰符即可! 譬如:

```
static __thread char buf[512];
```

但凡带有这种修饰符的变量, 每个线程都拥有一份对变量的拷贝, 意味着每个线程访问的都是该变量在本线程的副本, 从而避免了全局变量成为多个线程的共享数据。

关于线程局部变量的声明和使用, 需要注意以下几点:

- 如果变量声明中使用了关键字 `static` 或 `extern`, 那么关键字 `__thread` 必须紧随其后。
- 与一般的全局或静态变量申明一眼, 线程局部变量在申明时可设置一个初始值。
- 可以使用 C 语言取值操作符 (`&`) 来获取线程局部变量的地址。

Tips: 线程局部存储需要内核、Pthreads 以及 GCC 编译器的支持。

使用示例

我们编写一个简单的程序来测试线程局部存储, 示例代码如下所示:

示例代码 11.10.7 线程局部存储测试

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

static __thread char buf[100];

static void *thread_start(void *arg)
{
    strcpy(buf, "Child Thread\n");
    printf("子线程: buf (%p) = %s", buf, buf);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    strcpy(buf, "Main Thread\n");
```



```
/* 创建子线程 */
if (ret = pthread_create(&tid, NULL, thread_start, NULL)) {
    fprintf(stderr, "pthread_create error: %d\n", ret);
    exit(-1);
}

/* 等待回收子线程 */
if (ret = pthread_join(tid, NULL)) {
    fprintf(stderr, "pthread_join error: %d\n", ret);
    exit(-1);
}

printf("主线程: buf (%p) = %s", buf, buf);
exit(0);
}
```

程序中定义了一个全局变量 buf, 使用 `__thread` 修饰, 使其变为线程局部变量; 主线程中首先调用 `strcpy` 拷贝了字符串到 buf 缓冲区中, 随后创建了一个子线程, 子线程也调用了 `strcpy()` 向 buf 缓冲区拷贝了数据; 并调用 `printf` 打印 buf 缓冲区存储的字符串以及 buf 缓冲区的指针值。

子线程终止后, 主线程也打印 buf 缓冲区中存储的字符串以及 buf 缓冲区的指针值, 运行结果如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子线程: buf (0x7f87b6db369c) = Child Thread
主线程: buf (0x7f87b75a671c) = Main Thread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.9 测试结果

从地址便可以看出, 主线程和子线程中使用的 buf 绝不是同一个变量, 这就是线程局部存储, 使得每个线程都拥有一份对变量的拷贝, 各个线程操作各自的变量不会影响其它线程。

大家可以使用线程局部存储方式对示例代码 11.10.4 `strerror` 函数进行修改, 使其成为一个线程安全函数。

11.11 更多细节问题

本小节将对线程各方面的细节做深入讨论, 其主要包括线程与信号之间牵扯的问题、线程与进程控制 (`fork()`、`exec()`、`exit()`等) 之间的交互。之所以出现了这些问题, 其原因在于线程技术的问世晚于信号、进程控制等, 然而线程的出现必须要能够兼容现有的这些技术, 不能出现冲突, 这就使得线程与它们之间的结合使用将会变得比较复杂! 当中所涉及到的细节问题也会比较多。

11.11.1 线程与信号

Linux 信号模型是基于进程模型而设计的, 信号的问世远早于线程; 自然而然, 线程与信号之间就会存在一些冲突, 其主要原因在于: 信号既要能够在传统的单线程进程中保持它原有的功能、特性, 与此同时, 又需要设计出能够适用于多线程环境的新特性!

信号与多线程模型之间结合使用, 将会变得比较复杂, 需要考虑的问题将会更多, 在实际应用开发当中, 如果能够避免我们应尽量避免此类事情的发生; 但尽管如此, 事实上, 信号与多线程模型确实存在于实际的应用开发项目中。本小节我们就来讨论信号与线程之间牵扯的问题。

(1)、信号如何映射到线程

信号模型在一些方面是属于进程层面(由进程中的所有线程共享)的, 而在另一些方面是属于单个线程层面的, 以下对其进行汇总:

- 信号的系统默认行为是属于进程层面。8.3 小节介绍到, 每一个信号都有其对应的系统默认动作, 当进程中的任一线程收到任何一个未经处理(忽略或捕获)的信号时, 会执行该信号的默认操作, 信号的默认操作通常是停止或终止进程。
- 信号处理函数属于进程层面。进程中的所有线程共享程序中所注册的信号处理函数;
- 信号的发送既可针对整个进程, 也可针对某个特定的线程。在满足以下三个条件中的任意一个时, 信号的发送针对的是某个线程:
 - 产生了硬件异常相关信号, 譬如 SIGBUS、SIGFPE、SIGILL 和 SIGSEGV 信号; 这些硬件异常信号在某个线程执行指令的过程中产生, 也就是说这些硬件异常信号是由某个线程所引起; 那么在这种情况下, 系统会将信号发送给该线程。
 - 当线程试图对已断开的管道进行写操作时所产生的 SIGPIPE 信号;
 - 由函数 pthread_kill() 或 pthread_sigqueue() 所发出的信号, 稍后介绍这两个函数; 这些函数允许线程向同一进程下的其它线程发送一个指定的信号。

除了以上提到的三种情况之外, 其它机制产生的信号均属于进程层面, 譬如其它进程调用 kill() 或 sigqueue() 所发送的信号; 用户在终端按下 Ctrl+C、Ctrl+\、Ctrl+Z 向前台进程发送的 SIGINT、SIGQUIT 以及 SIGTSTP 信号。

- 当一个多线程进程接收到一个信号时, 且该信号绑定了信号处理函数时, 内核会任选一个线程来接收这个信号, 意味着由该线程接收信号并调用信号处理函数对其进行处理, 并不是每个线程都会接收到该信号并调用信号处理函数; 这种行为与信号的原始语义是保持一致的, 让进程对单个信号接收重复处理多次是没有意义的。
- 信号掩码其实是属于线程层面的, 也就是说信号掩码是针对每个线程而言。8.9 小节向大家介绍了信号掩码的概念, 并介绍了 sigprocmask() 函数, 通过 sigprocmask() 可以设置进程的信号掩码, 事实上, 信号掩码是并不是针对整个进程来说, 而是针对线程, 对于一个多线程应用程序来说, 并不存在一个作用于整个进程范围内的信号掩码(管理进程中的所有线程); 那么在多线程环境下, 各个线程可以调用 pthread_sigmask() 函数来设置它们各自的信号掩码, 譬如设置线程可以接收哪些信号、不接收哪些信号, 各线程可独立阻止或放行各种信号。
- 针对整个进程所挂起的信号, 以及针对每个线程所挂起的信号, 内核都会分别进行维护、记录。8.11.1 小节介绍到, 调用 sigpending() 会返回进程中所有被挂起的信号, 事实上, sigpending() 会返回针对整个进程所挂起的信号, 以及针对每个线程所挂起的信号的并集。

(2)、线程的信号掩码

对于一个单线程程序来说, 使用 sigprocmask() 函数设置进程的信号掩码, 在多线程环境下, 使用 pthread_sigmask() 函数来设置各个线程的信号掩码, 其函数原型如下所示:

```
#include <signal.h>
```

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

pthread_sigmask() 函数就像 sigprocmask() 一样, 不同之处在于它在多线程程序中使用, 所以 pthread_sigmask() 函数的用法与 sigprocmask() 完全一样, 这里就不再重述!

每个刚创建的线程, 会从其创建者处继承信号掩码, 这个新的线程可以调用 pthread_sigmask() 函数来改变它的信号掩码。

(3)、向线程发送信号

调用 kill() 或 sigqueue() 所发送的信号都是针对整个进程来说的, 它属于进程层面, 具体该目标进程中的哪一个线程会去处理信号, 由内核进行选择。事实上, 在多线程程序中, 可以通过 pthread_kill() 向同一进程中的某个指定线程发送信号, 其函数原型如下所示:

```
#include <signal.h>
```

```
int pthread_kill(pthread_t thread, int sig);
```

参数 thread, 也就是线程 ID, 用于指定同一进程中的某个线程, 调用 pthread_kill() 将向参数 thread 指定的线程发送信号 sig。

如果参数 sig 为 0, 则不发送信号, 但仍会执行错误检查。函数调用成功返回 0, 失败将返回一个错误编号, 不会发送信号。

除了 pthread_kill() 函数外, 还可以调用 pthread_sigqueue() 函数; pthread_sigqueue() 函数执行与 sigqueue 类似的任务, 但它不是向进程发送信号, 而是向同一进程中的某个指定的线程发送信号。其函数原型如下所示:

```
#include <signal.h>
```

```
#include <pthread.h>
```

```
int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

参数 thread 为线程 ID, 指定接收信号的目标线程 (目标线程与调用 pthread_sigqueue() 函数的线程是属于同一个进程), 参数 sig 指定要发送的信号, 参数 value 指定伴随数据, 与 sigqueue() 函数中的 value 参数意义相同。

pthread_sigqueue() 函数的参数的含义与 sigqueue() 函数中对应参数相同意义相同。它俩的唯一区别在于, sigqueue() 函数发送的信号针对的是整个进程, 而 pthread_sigqueue() 函数发送的信号针对的是某个线程。

(4)、异步信号安全函数

应用程序中涉及信号处理函数时必须非常小心, 因为信号处理函数可能会在程序执行的任意时间点被调用, 从而打断主程序。接下来介绍一个概念---异步信号安全函数 (async-signal-safe function)。

前面介绍了线程安全函数, 作为线程安全函数可以被多个线程同时调用, 每次都能得到预期的结果, 但是这里有前提条件, 那就是没有信号处理函数参与; 换句话说, 线程安全函数不能在信号处理函数中被调用, 否则就不能保证它一定是安全的。所以就出现了异步信号安全函数。

异步信号安全函数指的是可以在信号处理函数中可以被安全调用的线程安全函数, 所以它比线程安全函数的要求更为严格! 可重入函数满足这个要求, 所以可重入函数一定是异步信号安全函数。而线程安全函数则不一定是异步信号安全函数了。

举个例子, 下面列举出来的一个函数是线程安全函数:

```
static pthread_mutex_t mutex;
```

```
static int glob = 0;
```

```
static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mutex); //互斥锁上锁

        local = glob;
        local++;
        glob = local;

        pthread_mutex_unlock(&mutex); //互斥锁解锁
    }
}
```

该函数虽然对全局变量进行读写操作，但是在访问全局变量时进行了加锁，避免了引发竞争冒险；它是一个线程安全函数，假设线程 1 正在执行函数 `func`，刚刚获得锁（也就是刚刚对互斥锁上锁），而这时进程收到信号，并分派给线程 1 处理，线程 1 接着跳转去执行信号处理函数，不巧的是，信号处理函数中也调用了 `func()` 函数，同样它也去获取锁，由于此时锁处于锁住状态，所以信号处理函数中调用 `func()` 获取锁将会陷入休眠、等待锁的释放。这时线程 1 就会陷入死锁状态，线程 1 无法执行，锁无法释放；如果其它线程也调用 `func()`，那它们也会陷入休眠、如此将会导致整个程序陷入死锁！

通过上面的分析，可知，涉及到信号处理函数时要非常小心。之所以涉及到信号处理函数时会出现安全问题，笔者认为主要原因在以下两个方面：

- 信号是异步的，信号可能会在任何时间点中断主程序的运行，跳转到信号处理函数处执行，从而形成一个新的执行流（信号处理函数执行流）。
- 信号处理函数执行流与线程执行流存在一些区别，信号处理函数所产生的执行流是由执行信号处理函数的线程所触发的，它俩是在同一个线程中，属于同一个线程执行流。

在异步信号安全函数、可重入函数以及线程安全函数三者中，可重入函数的要求是最严格的，所以通常会说可重入函数一定是线程安全函数、也一定是异步信号安全函数。通常对于上面所列举出的线程安全函数 `func()`，如果想将其实现为异步信号安全函数，可以在获取锁之前通过设置信号掩码，在锁期间禁止接收该信号，也就是说将函数实现为不可被信号中断。经过这样处理之后，函数 `func()` 就是一个异步信号安全函数了。

Linux 标准 C 库和系统调用中以下函数被认为是异步信号安全函数：

<code>_Exit()</code>	<code>_exit()</code>	<code>abort()</code>	<code>accept()</code>
<code>access()</code>	<code>aio_error()</code>	<code>aio_return()</code>	<code>aio_suspend()</code>
<code>alarm()</code>	<code>bind()</code>	<code>cfgetispeed()</code>	<code>cfgetospeed()</code>
<code>cfsetispeed()</code>	<code>cfsetospeed()</code>	<code>chdir()</code>	<code>chmod()</code>
<code>chown()</code>	<code>clock_gettime()</code>	<code>close()</code>	<code>connect()</code>
<code>creat()</code>	<code>dup()</code>	<code>dup2()</code>	<code>execle()</code>
<code>execve()</code>	<code>fchmod()</code>	<code>fchown()</code>	<code>fcntl()</code>
<code>fdatasync()</code>	<code>fork()</code>	<code>execl()</code>	<code>fstat()</code>
<code>fsync()</code>	<code>ftruncate()</code>	<code>getegid()</code>	<code>geteuid()</code>
<code>getgid()</code>	<code>getgroups()</code>	<code>getpeername()</code>	<code>getpgrp()</code>

getpid()	getppid()	getsockname()	getsockopt()
getuid()	kill()	link()	listen()
lseek()	lstat()	mkdir()	mkfifo()
open()	execv()	pause()	pipe()
poll()	posix_trace_event()	pselect()	raise()
read()	readlink()	recv()	recvfrom()
recvmsg()	rename()	rmdir()	select()
sem_post()	send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()	setsockopt()
setuid()	shutdown()	sigaction()	sigaddset()
sigdelset()	sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()	sigprocmask()
sigqueue()	sigset()	sigsuspend()	sleep()
socketatmark()	socket()	socketpair()	stat()
symlink()	faccessat()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()	tcsendbreak()
tcsetattr()	tcsetpgrp()	time()	timer_getoverrun()
timer_gettime()	timer_settime()	times()	umask()
uname()	unlink()	utime()	wait()
waitpid()	write()	fchmodat()	fchownat()
fexecve()	fstatat()	futimens()	linkat()
mkdirat()	mkfifoat()	mknod()	mknodat()
openat()	readlinkat()	renameat()	symlinkat()
unlinkat()	utimensat()	utimes()	fchdir()
pthread_kill()	pthread_self()	pthread_sigmask()	

表 11.11.1 异步信号安全函数

上表所列举出的这些函数被认为是异步信号安全函数,可以通过 man 手册查询,执行命令"man 7 signal",如下所示:

```

Async-signal-safe functions
A signal handler function must be very careful, since processing elsewhere may be interrupted at some arbitrary point in the execution
cept of "safe function". If a signal interrupts the execution of an unsafe function, and handler calls an unsafe function, then the be
POSIX.1-2004 (also known as POSIX.1-2001 Technical Corrigendum 2) requires an implementation to guarantee that the following functions
handler:

    _Exit()
    _exit()
    abort()
    accept()
    access()
    aio_error()
    aio_return()
    aio_suspend()
    alarm()
    bind()
    cfgetispeed()
    cfgetospeed()
    cfsetispeed()
    cfsetospeed()
    chdir()
    chmod()
    chown()
    clock_gettime()
    close()
    connect()
    creat()
    dup()
    dup2()
    execl()
    execve()
    fchmod()
    fchown()
    fcntl()
    fdatsync()
    fork()

```

图 11.11.1 异步信号安全函数

大家可以通过对比 man 手册查询到的这些异步信号安全函数, 来确定自己调用的库函数或系统调用是不是异步信号安全函数, 这里需要说, 在本书的示例代码中, 并没有完全按照安全性要求, 在信号处理函数中使用异步信号安全函数, 譬如在本书中的示例代码中, 信号处理函数中调用了 `printf()` 用于打印信息, 事实上这个函数是一个非异步信号安全函数, 当然在一个实际的项目应用程序当中不能这么用, 但是本书只是为了方便输出打印信息而已。

所以对于一个安全的信号处理函数来说, 需要做到以下几点:

- 首先确保信号处理函数本身的代码是可重入的, 且只能调用异步信号安全函数;
- 当主程序执行不安全函数或是去操作信号处理函数也可能会更新的全局数据结构时, 要阻塞信号的传递。

关于异步信号安全函数就给大家介绍这么多, 多线程环境下涉及到信号处理时尤其要注意这些问题。

(5)、多线程环境下信号的处理

第十二章 线程同步

本章来聊一聊线程同步这个话题, 对于一个单线程进程来说, 它不需要处理线程同步的问题, 所以线程同步是在多线程环境下可能需要注意的一个问题。线程的主要优势在于, 资源的共享性, 譬如通过全局变量来实现信息共享, 不过这种便捷的共享是有代价的, 那就是多个线程并发访问共享数据所导致的数据不一致的问题。

本章来学习如何使用线程同步机制来避免这样的问题!

本章将会讨论如下主题内容。

- 为什么需要线程同步;
- 线程同步之互斥锁;
- 线程同步之信号量;
- 线程同步之条件变量;
- 线程同步之读写锁。

12.1 为什么需要线程同步?

线程同步是为了对共享资源的访问进行保护。这里说的共享资源指的是多个线程都会进行访问的资源,譬如定义了一个全局变量 `a`, 线程 1 访问了变量 `a`、同样在线程 2 中也访问了变量 `a`, 那么此时变量 `a` 就是多个线程间的共享资源, 大家都要访问它。

保护的目的是为了解决数据一致性的问题。当然什么情况下才会出现数据一致性的问题, 根据不同的情况进行区分: 如果每个线程访问的变量都是其它线程不会读取和修改的 (譬如线程函数内定义的局部变量或者只有一个线程访问的全局变量), 那么就不存在数据一致性的问题; 同样, 如果变量是只读的, 多个线程同时读取该变量也不会有数据一致性的问题; 但是, 当一个线程可以修改的变量, 其它的线程也可以读取或者修改的时候, 这个时候就存在数据一致性的问题, 需要对这些线程进行同步操作, 确保它们在访问变量的存储内容时不会访问到无效的值。

出现数据一致性问题其本质在于进程中的多个线程对共享资源的并发访问 (同时访问)。前面给大家介绍了, 进程中的多个线程间是并发执行的, 每个线程都是系统调用的基本单元, 参与到系统调度队列中; 对于多个线程间的共享资源, 并发执行会导致对共享资源的并发访问, 并发访问所带来的问题就是竞争 (如果多个线程同时对共享资源进行访问就表示存在竞争, 跟现实生活当中的竞争有一定的相似之处, 譬如一个队伍当中需要选出一名队长, 现在有两个人在候选名单中, 那么意味着这两个人就存在竞争关系), 并发访问就可能会出现数据一致性问题, 所以需要解决这个问题; 要防止并发访问共享资源, 那么就需要对共享资源的访问进行保护, 防止出现并发访问共享资源。

当一个线程修改变量时, 其它的线程在读取这个变量时可能会看到不一致的值, 图 12.1.1 描述了两个线程读写相同变量 (共享变量、共享资源) 的假设例子。在这个例子当中, 线程 A 读取变量的值, 然后再给这个变量赋予一个新的值, 但写操作需要 2 个时钟周期 (这里只是假设); 当线程 B 在这两个写周期中间读取了这个变量, 它就会得到不一致的值, 这就出现了数据不一致的问题。

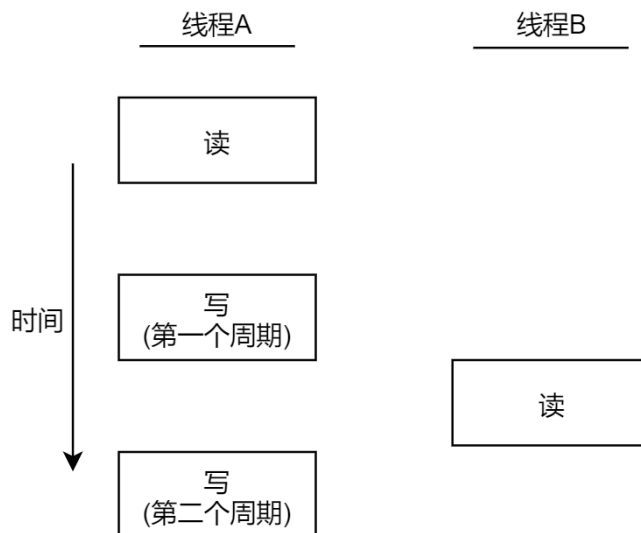


图 12.1.1 多线程并发访问数据不一致

我们可以编写一个简单地代码对此文件进行测试, 示例代码 12.1.1 展示了在 2 个线程在常规方式下访问共享资源, 这里的共享资源指的就是静态全局变量 `g_count`。该程序创建了两个线程, 且均执行同一个函数, 该函数执行一个循环, 重复以下步骤: 将全局变量 `g_count` 复制到本地变量 `l_count` 变量中, 然后递增 `l_count`, 再把 `l_count` 复制回 `g_count`, 以此不断增加全局变量 `g_count` 的值。因为 `l_count` 是分配于线程栈中的自动变量 (函数内定义的局部变量), 所以每个线程都有一份。循环重复的次数要么由命令行参数指定, 要么去默认值 1000 万次, 循环结束之后线程终止, 主线程回收两个线程之后, 再将全局变量 `g_count` 的值打印出来。


```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
        l_count = g_count;
        l_count++;
        g_count = l_count;
    }

    return (void *)0;
}

static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000; //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
```

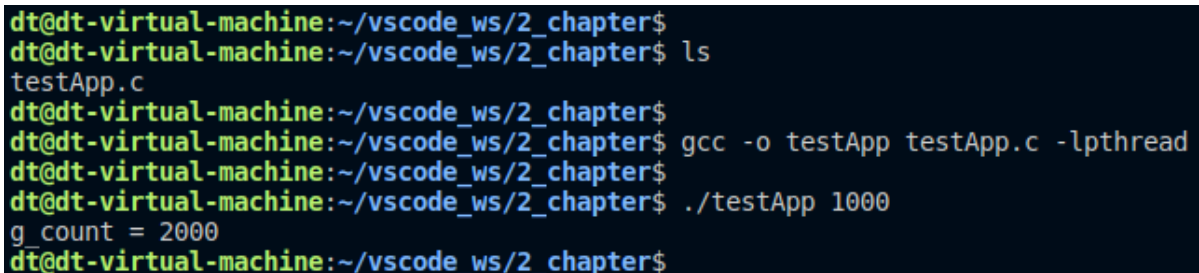
```
if (ret) {
    fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
    exit(-1);
}

/* 等待线程结束 */
ret = pthread_join(tid1, NULL);
if (ret) {
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}

ret = pthread_join(tid2, NULL);
if (ret) {
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}

/* 打印结果 */
printf("g_count = %d\n", g_count);
exit(0);
}
```

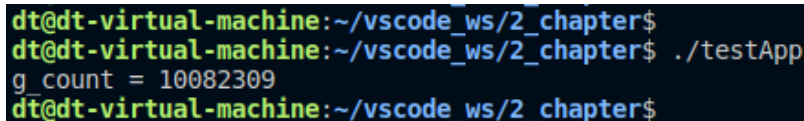
编译代码,进行测试,首先执行代码,传入参数 1000,也就是让每个线程对全局变量 g_count 递增 1000 次,如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 1000
g_count = 2000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.1.2 1000 次测试结果

都打印结果看,得到了我们想象中的结果,每个线程递增 1000 次,最后的数值就是 2000;接着我们把递增次数加大,采用默认值 1000 万次,如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
g_count = 10082309
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.1.3 1000 万次测试结果

可以发现,结果竟然不是我们想看到的样子,执行到最后,应该是 2000 万才对,这里其实就出现图 12.1.1 中所示的问题,数据不一致。

如何解决对共享资源的并发访问出现数据不一致的问题?

为了解决图 12.1.1 中数据不一致的问题,就得需要 Linux 提供的一些方法,也就是接下来将要向大家介绍的线程同步技术,来实现同一时间只允许一个线程访问该变量,防止出现并发访问的情况、消除数据不

一致的问题, 图 12.1.4 描述了这种同步操作, 从图中可知, 线程 A 和线程 B 都不会同时访问这个变量, 当线程 A 需要修改变量的值时, 必须等到写操作完成之后 (不能打断它的操作), 才运行线程 B 去读取。

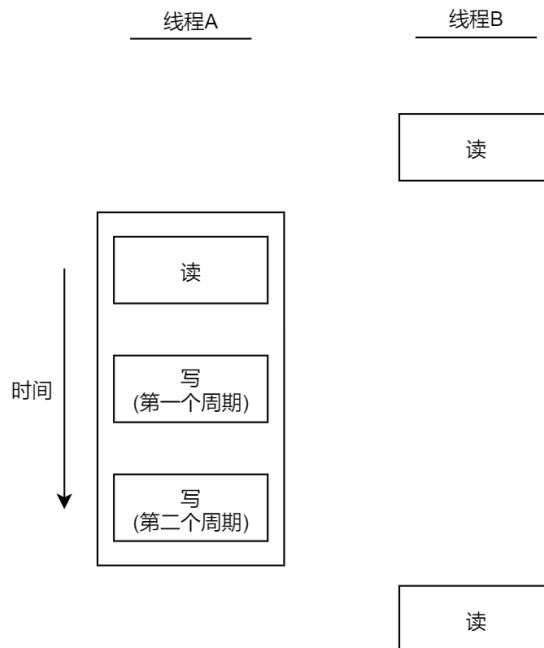


图 12.1.4 线程同步访问变量

线程的主要优势在于, 资源的共享性, 譬如通过全局变量来实现信息共享。不过这种便捷的共享是有代价的, 必须确保多个线程不会同时修改同一变量、或者某一线程不会读取正由其它线程修改的变量, 也就是必须确保不会出现对共享资源的并发访问。Linux 系统提供了多种用于实现线程同步的机制, 常见的方法有: 互斥锁、条件变量、自旋锁以及读写锁等, 下面将向大家一一进行介绍。

12.2 互斥锁

互斥锁 (mutex) 又叫互斥量, 从本质上说是一把锁, 在访问共享资源之前对互斥锁进行上锁, 在访问完成后释放互斥锁 (解锁); 对互斥锁进行上锁之后, 任何其它试图再次对互斥锁进行加锁的线程都会被阻塞, 直到当前线程释放互斥锁。如果释放互斥锁时有一个以上的线程阻塞, 那么这些阻塞的线程会被唤醒, 它们都会尝试对互斥锁进行加锁, 当有一个线程成功对互斥锁上锁之后, 其它线程就不能再次上锁了, 只能再次陷入阻塞, 等待下一次解锁。

举一个非常简单容易理解的例子, 就拿卫生间 (共享资源) 来说, 当来了一个人 (线程) 看到卫生间没人, 然后它进去了、并且从里边把门锁住 (互斥锁上锁) 了; 此时又来了两个人 (线程), 它们也想进卫生间方便, 发生此时门打不开 (互斥锁上锁失败), 因为里边有人, 所以此时它们只能等待 (陷入阻塞); 当里边的人方便完了之后 (访问共享资源完成), 把锁 (互斥锁解锁) 打开从里边出来, 此时外边有两个人在等, 当然它们都迫不及待想要进去 (尝试对互斥锁进行上锁), 自然两个人只能进去一个, 进去的人再次把门锁住, 另外一个人只能继续等待它出来。

在我们的程序设计当中, 只有将所有线程访问共享资源都设计成相同的数据访问规则, 互斥锁才能正常工作。如果允许其中的某个线程在没有得到锁的情况下也可以访问共享资源, 那么即使其它的线程在使用共享资源前都申请锁, 也还是会出现数据不一致的问题。

互斥锁使用 `pthread_mutex_t` 数据类型表示, 在使用互斥锁之前, 必须首先对它进行初始化操作, 可以使用两种方式对互斥锁进行初始化操作。

12.2.1 互斥锁初始化

1、使用 PTHREAD_MUTEX_INITIALIZER 宏初始化互斥锁

互斥锁使用 `pthread_mutex_t` 数据类型表示, `pthread_mutex_t` 其实是一个结构体类型, 而宏 `PTHREAD_MUTEX_INITIALIZER` 其实是一个对结构体赋值操作的封装, 如下所示:

```
# define PTHREAD_MUTEX_INITIALIZER \  
{ { 0, 0, 0, 0, 0, __PTHREAD_SPINS, { 0, 0 } } }
```

所以由此可知, 使用 `PTHREAD_MUTEX_INITIALIZER` 宏初始化互斥锁的操作如下:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

`PTHREAD_MUTEX_INITIALIZER` 宏已经携带了互斥锁的默认属性。

2、使用 `pthread_mutex_init()` 函数初始化互斥锁

使用 `PTHREAD_MUTEX_INITIALIZER` 宏只适用于在定义的时候就直接进行初始化, 对于其它情况则不能使用这种方式, 譬如先定义互斥锁, 后再进行初始化, 或者在堆中动态分配的互斥锁, 譬如使用 `malloc()` 函数申请分配的互斥锁对象, 那么在这些情况下, 可以使用 `pthread_mutex_init()` 函数对互斥锁进行初始化, 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

使用该函数需要包含头文件 `<pthread.h>`。

函数参数和返回值含义如下:

mutex: 参数 `mutex` 是一个 `pthread_mutex_t` 类型指针, 指向需要进行初始化操作的互斥锁对象;

attr: 参数 `attr` 是一个 `pthread_mutexattr_t` 类型指针, 指向一个 `pthread_mutexattr_t` 类型对象, 该对象用于定义互斥锁的属性 (在 12.2.6 小节中介绍), 若将参数 `attr` 设置为 `NULL`, 则表示将互斥锁的属性设置为默认值, 在这种情况下其实就等价于 `PTHREAD_MUTEX_INITIALIZER` 这种方式初始化, 而不同之处在于, 使用宏不进行错误检查。

返回值: 成功返回 0; 失败将返回一个非 0 的错误码。

Tips: 注意, 当在 Ubuntu 系统下执行 "man 3 pthread_mutex_init" 命令时提示找不到该函数, 并不是 Linux 下没有这个函数, 而是该函数相关的 man 手册帮助信息没有被安装, 这时我们只需执行 "sudo apt-get install manpages-posix-dev" 安装即可。

使用 `pthread_mutex_init()` 函数对互斥锁进行初始化示例:

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

或者:

```
pthread_mutex_t *mutex = malloc(sizeof(pthread_mutex_t));  
pthread_mutex_init(mutex, NULL);
```

12.2.2 互斥锁加锁和解锁

互斥锁初始化之后, 处于一个未锁定状态, 调用函数 `pthread_mutex_lock()` 可以对互斥锁加锁、获取互斥锁, 而调用函数 `pthread_mutex_unlock()` 可以对互斥锁解锁、释放互斥锁。其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

使用这些函数需要包含头文件<pthread.h>, 参数 `mutex` 指向互斥锁对象; `pthread_mutex_lock()` 和 `pthread_mutex_unlock()` 在调用成功时返回 0; 失败将返回一个非 0 值的错误码。

调用 `pthread_mutex_lock()` 函数对互斥锁进行上锁, 如果互斥锁处于未锁定状态, 则此次调用会上锁成功, 函数调用将立马返回; 如果互斥锁此时已经被其它线程锁定了, 那么调用 `pthread_mutex_lock()` 会一直阻塞, 直到该互斥锁被解锁, 到那时, 调用将锁定互斥锁并返回。

调用 `pthread_mutex_unlock()` 函数将已经处于锁定状态的互斥锁进行解锁。以下行为均属错误:

- 对处于未锁定状态的互斥锁进行解锁操作;
- 解锁由其它线程锁定的互斥锁。

如果有多个线程处于阻塞状态等待互斥锁被解锁, 当互斥锁被当前锁定它的线程调用 `pthread_mutex_unlock()` 函数解锁后, 这些等待着的线程都会有机会对互斥锁上锁, 但无法判断究竟哪个线程会如愿以偿!

使用示例

使用互斥锁的方式将示例代码 12.1.1 进行修改, 修改之后如示例代码 12.2.1 所示, 使用了一个互斥锁来保护对全局变量 `g_count` 的访问。

示例代码 12.2.1 使用互斥锁保护全局变量的访问

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static pthread_mutex_t mutex;
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mutex); //互斥锁上锁

        l_count = g_count;
        l_count++;
        g_count = l_count;

        pthread_mutex_unlock(&mutex); //互斥锁解锁
    }

    return (void *)0;
}
```

```
static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000; //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid2, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 打印结果 */
    printf("g_count = %d\n", g_count);
}
```

```

    exit(0);
}

```

在测试运行, 使用默认值 1000 万次, 如下所示:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
g_count = 20000000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 12.2.1 测试结果

可以看到确实得到了我们想看到的正确结果, 每次对 `g_count` 的累加总是能够保持正确, 但是在运行程序的过程中, 明显会感觉到锁消耗的时间会比较长, 这就涉及到性能的问题了, 后续会介绍!

12.2.3 pthread_mutex_trylock()函数

当互斥锁已经被其它线程锁住时, 调用 `pthread_mutex_lock()`函数会被阻塞, 直到互斥锁解锁; 如果线程不希望被阻塞, 可以使用 `pthread_mutex_trylock()`函数; 调用 `pthread_mutex_trylock()`函数尝试对互斥锁进行加锁, 如果互斥锁处于未锁住状态, 那么调用 `pthread_mutex_trylock()`将会锁住互斥锁并立马返回, 如果互斥锁已经被其它线程锁住, 调用 `pthread_mutex_trylock()`加锁失败, 但不会阻塞, 而是返回错误码 `EBUSY`。

其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

参数 `mutex` 指向目标互斥锁, 成功返回 0, 失败返回一个非 0 值的错误码, 如果目标互斥锁已经被其它线程锁住, 则调用失败返回 `EBUSY`。

使用示例

对示例代码 12.2.1 进行修改, 使用 `pthread_mutex_trylock()`替换 `pthread_mutex_lock()`。

示例代码 12.2.2 以非阻塞方式对互斥锁进行加锁

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static pthread_mutex_t mutex;
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {

```

```
while(pthread_mutex_trylock(&mutex)); //以非阻塞方式上锁

l_count = g_count;
l_count++;
g_count = l_count;

pthread_mutex_unlock(&mutex); //互斥锁解锁
}

return (void *)0;
}

static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 1000000; //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
```



```
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}

ret = pthread_join(tid2, NULL);
if (ret) {
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}

/* 打印结果 */
printf("g_count = %d\n", g_count);
exit(0);
}
```

整个执行结果跟使用 `pthread_mutex_lock()` 效果是一样的, 大家可以自己测试。

12.2.4 销毁互斥锁

当不再需要互斥锁时, 应该将其销毁, 通过调用 `pthread_mutex_destroy()` 函数来销毁互斥锁, 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

使用该函数需要包含头文件 `<pthread.h>`, 参数 `mutex` 指向目标互斥锁; 同样在调用成功情况下返回 0, 失败返回一个非 0 值的错误码。

- 不能销毁还没有解锁的互斥锁, 否则将会出现错误;
- 没有初始化的互斥锁也不能销毁。

被 `pthread_mutex_destroy()` 销毁之后的互斥锁, 就不能再对它进行上锁和解锁了, 需要再次调用 `pthread_mutex_init()` 对互斥锁进行初始化之后才能使用。

使用示例

对示例代码 12.2.1 进行修改, 在进程退出之前, 使用 `pthread_mutex_destroy()` 函数销毁互斥锁。

示例代码 12.2.3 销毁互斥锁

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static pthread_mutex_t mutex;
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
```

```
int l_count, j;

for (j = 0; j < loops; j++) {
    pthread_mutex_lock(&mutex); //互斥锁上锁

    l_count = g_count;
    l_count++;
    g_count = l_count;

    pthread_mutex_unlock(&mutex); //互斥锁解锁
}

return (void *)0;
}

static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000; //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }
}
```

```
/* 等待线程结束 */
ret = pthread_join(tid1, NULL);
if (ret) {
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}

ret = pthread_join(tid2, NULL);
if (ret) {
    fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
    exit(-1);
}

/* 打印结果 */
printf("g_count = %d\n", g_count);

/* 销毁互斥锁 */
pthread_mutex_destroy(&mutex);
exit(0);
}
```

12.2.5 互斥锁死锁

试想一下, 如果一个线程试图对同一个互斥锁加锁两次, 会出现什么情况? 情况就是该线程会陷入死锁状态, 一直被阻塞永远出不来; 这就是出现死锁的一种情况, 除此之外, 使用互斥锁还有其它很多种方式也能产生死锁。

有时, 一个线程需要同时访问两个或更多不同的共享资源, 而每个资源又由不同的互斥锁管理。当超过一个线程对同一组互斥锁 (两个或两个以上的互斥锁) 进行加锁时, 就有可能发生死锁; 譬如, 程序中使用一个以上的互斥锁, 如果允许一个线程一直占有第一个互斥锁, 并且在试图锁住第二个互斥锁时处于阻塞状态, 但是拥有第二个互斥锁的线程也在试图锁住第一个互斥锁。因为两个线程都在相互请求另一个线程拥有的资源, 所以这两个线程都无法向前运行, 会被一直阻塞, 于是就产生了死锁。如下示例代码中所示:

```
// 线程 A
pthread_mutex_lock(mutex1);
pthread_mutex_lock(mutex2);

// 线程 B
pthread_mutex_lock(mutex2);
pthread_mutex_lock(mutex1);
```

这就好比是 C 语言中两个头文件相互包含的关系, 那肯定编译报错!

在我们的程序当中, 如果用到了多个互斥锁, 要避免此类死锁的问题, 最简单的方式就是定义互斥锁的层级关系, 当多个线程对一组互斥锁操作时, 总是应该按照相同的顺序对该组互斥锁进行锁定。譬如在上述场景中, 如果两个线程总是先锁定 `mutex1` 在锁定 `mutex2`, 死锁就不会出现。有时, 互斥锁之间的层级关系逻辑不够清晰, 即使是这样, 依然可以设计出所有线程都必须遵循的强制层级顺序。

但有时候,应用程序的结构使得对互斥锁进行排序是很困难的,程序复杂、其中所涉及到的互斥锁以及共享资源比较多,程序设计实在无法按照相同的顺序对一组互斥锁进行锁定,那么就必须采用另外的方法。譬如使用 `pthread_mutex_trylock()` 以不阻塞的方式尝试对互斥锁进行加锁,在这种方案中,线程先使用函数 `pthread_mutex_lock()` 锁定第一个互斥锁,然后使用 `pthread_mutex_trylock()` 来锁定其余的互斥锁。如果任一 `pthread_mutex_trylock()` 调用失败(返回 `EBUSY`),那么该线程释放所有互斥锁,可以经过一段时间之后从头再试。与第一种按照层级关系来避免死锁的方法变比,这种方法效率要低一些,因为可能需要经历多次循环。

解决互斥锁死锁的问题还有很多方法,笔者也没详细地去学习过,当大家在实际编程应用中需要用到这些知识再去查阅相关资料、书籍进行学习。

使用示例

想了半天没有什么比较好的例子,暂时先歇下!

12.2.6 互斥锁的属性

如前所述,调用 `pthread_mutex_init()` 函数初始化互斥锁时可以设置互斥锁的属性,通过参数 `attr` 指定。参数 `attr` 指向一个 `pthread_mutexattr_t` 类型对象,该对象对互斥锁的属性进行定义,当然,如果将参数 `attr` 设置为 `NULL`,则表示将互斥锁属性设置为默认值。关于互斥锁的属性本书不打算深入讨论互斥锁属性的细节,也不会将 `pthread_mutexattr_t` 类型中定义的属性一一列出。

如果不使用默认属性,在调用 `pthread_mutex_init()` 函数时,参数 `attr` 必须要指向一个 `pthread_mutexattr_t` 对象,而不能使用 `NULL`。当定义 `pthread_mutexattr_t` 对象之后,需要使用 `pthread_mutexattr_init()` 函数对该对象进行初始化操作,当对象不再使用时,需要使用 `pthread_mutexattr_destroy()` 将其销毁,函数原型如下所示:

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

参数 `attr` 指向需要进行初始化的 `pthread_mutexattr_t` 对象,调用成功返回 0,失败将返回非 0 值的错误码。

`pthread_mutexattr_init()` 函数将使用默认的互斥锁属性初始化参数 `attr` 指向的 `pthread_mutexattr_t` 对象。关于互斥锁的属性比较多,譬如进程共享属性、健壮属性、类型属性等等,本书并不会一一给大家进行介绍,本小节讨论下类型属性,其它的暂时不去解释了。

互斥锁的类型属性控制着互斥锁的锁定特性,一共有 4 中类型:

- **PTHREAD_MUTEX_NORMAL:** 一种标准的互斥锁类型,不做任何的错误检查或死锁检测。如果线程试图对已经由自己锁定的互斥锁再次进行加锁,则发生死锁;互斥锁处于未锁定状态,或者已由其它线程锁定,对其解锁会导致不确定结果。
- **PTHREAD_MUTEX_ERRORCHECK:** 此类互斥锁会提供错误检查。譬如这三种情况都会导致返回错误:线程试图对已经由自己锁定的互斥锁再次进行加锁(同一线程对同一互斥锁加锁两次),返回错误;线程对由其它线程锁定的互斥锁进行解锁,返回错误;线程对处于未锁定状态的互斥锁进行解锁,返回错误。这类互斥锁运行起来比较慢,因为它需要做错误检查,不过可将其作为调试工具,以发现程序哪里违反了互斥锁使用的基本原则。
- **PTHREAD_MUTEX_RECURSIVE:** 此类互斥锁允许同一线程在互斥锁解锁之前对该互斥锁进行多次加锁,然后维护互斥锁加锁的次数,把这种互斥锁称为递归互斥锁,但是如果解锁次数不等于

加速次数, 则是不会释放锁的; 所以, 如果对一个递归互斥锁加锁两次, 然后解锁一次, 那么这个互斥锁依然处于锁定状态, 对它再次进行解锁之前不会释放该锁。

- **PTHREAD_MUTEX_DEFAULT**: 此类互斥锁提供默认的行为和特性。使用宏 **PTHREAD_MUTEX_INITIALIZER** 初始化的互斥锁, 或者调用参数 **arg** 为 **NULL** 的 **pthread_mutexattr_init()** 函数所创建的互斥锁, 都属于此类型。此类锁意在为互斥锁的实现保留最大灵活性, Linux 上, **PTHREAD_MUTEX_DEFAULT** 类型互斥锁的行为与 **PTHREAD_MUTEX_NORMAL** 类型相仿。

可以使用 **pthread_mutexattr_gettype()** 函数得到互斥锁的类型属性, 使用 **pthread_mutexattr_settype()** 修改/设置互斥锁类型属性, 其函数原型如下所示:

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

使用这些函数需要包含头文件 **<pthread.h>**, 参数 **attr** 指向 **pthread_mutexattr_t** 类型对象; 对于 **pthread_mutexattr_gettype()** 函数, 函数调用成功会将互斥锁类型属性保存在参数 **type** 所指向的内存中, 通过它返回出来; 而对于 **pthread_mutexattr_settype()** 函数, 会将参数 **attr** 指向的 **pthread_mutexattr_t** 对象的类型属性设置为参数 **type** 指定的类型。使用方式如下:

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

/* 初始化互斥锁属性对象 */
pthread_mutexattr_init(&attr);

/* 将类型属性设置为 PTHREAD_MUTEX_NORMAL */
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);

/* 初始化互斥锁 */
pthread_mutex_init(&mutex, &attr);

.....

/* 使用完之后 */
pthread_mutexattr_destroy(&attr);
pthread_mutex_destroy(&mutex);
```

12.3 条件变量

本小节讨论第二种线程同步的方法---条件变量。

条件变量是线程可用的另一种同步机制。条件变量用于自动阻塞线程, 知道某个特定事件发生或某个条件满足为止, 通常情况下, 条件变量是和互斥锁一起搭配使用的。使用条件变量主要包括两个动作:

- 一个线程等待某个条件满足而被阻塞;
- 另一个线程中, 条件满足时发出“信号”。

为了说明这个问题, 来看一个没有使用条件变量的例子, 生产者---消费者模式, 生产者这边负责生产产品、而消费者负责消费产品, 对于消费者来说, 没有产品的时候只能等待产品出来, 有产品就使用它。

这里我们使用一个变量来表示这个产品,生产者生产一件产品变量加 1,消费者消费一次变量减 1,示例代码如下所示:

示例代码 12.3.1 生产者---消费者示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static pthread_mutex_t mutex;
static int g_avail = 0;

/* 消费者线程 */
static void *consumer_thread(void *arg)
{
    for (;;) {
        pthread_mutex_lock(&mutex); // 上锁

        while (g_avail > 0)
            g_avail--; // 消费

        pthread_mutex_unlock(&mutex); // 解锁
    }

    return (void *)0;
}

/* 主线程 (生产者) */
int main(int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, consumer_thread, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }
}
```

```
for (;;) {
    pthread_mutex_lock(&mutex); // 上锁
    g_avail++; // 生产
    pthread_mutex_unlock(&mutex); // 解锁
}

exit(0);
}
```

此代码中, 主线程作为“生产者”, 新创建的线程作为“消费者”, 运行之后它们都回处于死循环中, 所以代码中没有加入销毁互斥锁、等待回收新线程相关的代码, 进程终止时会自动被处理。

上述代码虽然可行, 但由于新线程中会不停的循环检查全局变量 `g_avail` 是否大于 0, 故而造成 CPU 资源的浪费。采用条件变量这一问题就可以迎刃而解! 条件变量允许一个线程休眠(阻塞等待)直至获取到另一个线程的通知(收到信号)再去执行自己的操作, 譬如上述代码中, 当条件 `g_avail > 0` 不成立时, 消费者线程会进入休眠状态, 而生产者生成产品后 (`g_avail++`, 此时 `g_avail` 将会大于 0), 向处于等待状态的线程发出“信号”, 而其它线程收到“信号”之后, 便会被唤醒!

Tips: 这里提到的信号并不是第八章内容所指的信号, 需要区分开来!

前面说到, 条件变量通常搭配互斥锁来使用, 是因为条件的检测是在互斥锁的保护下进行的, 也就是说条件本身是由互斥锁保护的, 线程在改变条件状态之前必须首先锁住互斥锁, 不然就可能引发线程不安全的问题。

12.3.1 条件变量初始化

条件变量使用 `pthread_cond_t` 数据类型来表示, 类似于互斥锁, 在使用条件变量之前必须对其进行初始化。初始化方式同样也有两种: 使用宏 `PTHREAD_COND_INITIALIZER` 或者使用函数 `pthread_cond_init()`, 使用宏的初始化方法与互斥锁的初始化宏一样, 这里就不再重述! 譬如:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

`pthread_cond_init()` 函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

同样, 使用这些函数需要包含头文件 `<pthread.h>`, 使用 `pthread_cond_init()` 函数初始化条件变量, 当不再使用时, 使用 `pthread_cond_destroy()` 销毁条件变量。

参数 `cond` 指向 `pthread_cond_t` 条件变量对象, 对于 `pthread_cond_init()` 函数, 类似于互斥锁, 在初始化条件变量时设置条件变量的属性, 参数 `attr` 指向一个 `pthread_condattr_t` 类型对象, `pthread_condattr_t` 数据类型用于描述条件变量的属性。可将参数 `attr` 设置为 `NULL`, 表示使用属性的默认值来初始化条件变量, 与使用 `PTHREAD_COND_INITIALIZER` 宏相同。

函数调用成功返回 0, 失败将返回一个非 0 值的错误码。

对于初始化与销毁操作, 有以下问题需要注意:

- 在使用条件变量之前必须对条件变量进行初始化操作, 使用 `PTHREAD_COND_INITIALIZER` 宏或者函数 `pthread_cond_init()` 都行;
- 对已经初始化的条件变量再次进行初始化, 将可能会导致未定义行为;
- 对没有进行初始化的条件变量进行销毁, 也将可能会导致未定义行为;
- 对某个条件变量而言, 仅当没有任何线程等待它时, 将其销毁才是最安全的;

- 经 `pthread_cond_destroy()` 销毁的条件变量, 可以再次调用 `pthread_cond_init()` 对其进行重新初始化。

12.3.2 通知和等待条件变量

条件变量的主要操作便是发送信号 (signal) 和等待。发送信号操作即是通知一个或多个处于等待状态的线程, 某个共享变量的状态已经改变, 这些处于等待状态的线程收到通知之后便会被唤醒, 唤醒之后再检查条件是否满足。等待操作是指在收到一个通知前一直处于阻塞状态。

函数 `pthread_cond_signal()` 和 `pthread_cond_broadcast()` 均可向指定的条件变量发送信号, 通知一个或多个处于等待状态的线程。调用 `pthread_cond_wait()` 函数是线程阻塞, 直到收到条件变量的通知。

`pthread_cond_signal()` 和 `pthread_cond_broadcast()` 函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

使用这些函数需要包含头文件 `<pthread.h>`, 参数 `cond` 指向目标条件变量, 向该条件变量发送信号。调用成功返回 0; 失败将返回一个非 0 值的错误码。

`pthread_cond_signal()` 和 `pthread_cond_broadcast()` 的区别在于: 二者对阻塞于 `pthread_cond_wait()` 的多个线程对应的处理方式不同, `pthread_cond_signal()` 函数至少能唤醒一个线程, 而 `pthread_cond_broadcast()` 函数则能唤醒所有线程。使用 `pthread_cond_broadcast()` 函数总能产生正确的结果, 唤醒所有等待状态的线程, 但函数 `pthread_cond_signal()` 会更为高效, 因为它只需确保至少唤醒一个线程即可, 所以如果我们的程序当中, 只有一个处于等待状态的线程, 使用 `pthread_cond_signal()` 更好, 具体使用哪个函数根据实际情况进行选择!

`pthread_cond_wait()` 函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

当程序当中使用条件变量, 当判断某个条件不满足时, 调用 `pthread_cond_wait()` 函数将线程设置为等待状态 (阻塞)。**`pthread_cond_wait()` 函数包含两个参数:**

cond: 指向需要等待的条件变量, 目标条件变量;

mutex: 参数 `mutex` 是一个 `pthread_mutex_t` 类型指针, 指向一个互斥锁对象; 前面开头便给大家介绍了, 条件变量通常是和互斥锁一起使用, 因为条件的检测 (条件检测通常是需要访问共享资源的) 是在互斥锁的保护下进行的, 也就是说条件本身是由互斥锁保护的。

返回值: 调用成功返回 0; 失败将返回一个非 0 值的错误码。

在 `pthread_cond_wait()` 函数内部会对参数 `mutex` 所指定的互斥锁进行操作, 通常情况下, 条件判断以及 `pthread_cond_wait()` 函数调用均在互斥锁的保护下, 也就是说, 在此之前线程已经对互斥锁加锁了。调用 `pthread_cond_wait()` 函数时, 调用者把互斥锁传递给函数, 函数会自动把调用线程放到等待条件的线程列表上, 然后将互斥锁解锁; 当 `pthread_cond_wait()` 被唤醒返回时, 会再次锁住互斥锁。

注意注意的是, 条件变量并不保存状态信息, 只是传递应用程序状态信息的一种通讯机制。如果调用 `pthread_cond_signal()` 和 `pthread_cond_broadcast()` 向指定条件变量发送信号时, 若无任何线程等待该条件变量, 这个信号也就会不了了之。

当调用 `pthread_cond_broadcast()` 同时唤醒所有线程时, 互斥锁也只能被某一线程锁住, 其它线程获取锁失败又会陷入阻塞。

使用示例

使用条件变量对示例代码 12.3.1 进行修改, 当消费者线程没有产品可消费时, 让它处于等待状态, 知道生产者把产品生产出来; 当生产者把产品生产出来之后, 再去通知消费者。

示例代码 12.3.2 使用条件变量和互斥锁实现线程同步

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static pthread_mutex_t mutex;    //定义互斥锁
static pthread_cond_t cond;     //定义条件变量
static int g_avail = 0;         //全局共享资源

/* 消费者线程 */
static void *consumer_thread(void *arg)
{
    for (;;) {
        pthread_mutex_lock(&mutex); //上锁

        while (0 >= g_avail)
            pthread_cond_wait(&cond, &mutex); //等待条件满足

        while (0 < g_avail)
            g_avail--; //消费

        pthread_mutex_unlock(&mutex); //解锁
    }

    return (void *)0;
}

/* 主线程 (生产者) */
int main(int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    /* 初始化互斥锁和条件变量 */
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, consumer_thread, NULL);
```

```
if (ret) {
    fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
    exit(-1);
}

for (;;) {
    pthread_mutex_lock(&mutex); // 上锁
    g_avail++; // 生产
    pthread_mutex_unlock(&mutex); // 解锁
    pthread_cond_signal(&cond); // 向条件变量发送信号
}

exit(0);
}
```

全局变量 `g_avail` 作为主线程和新线程之间的共享资源，两个线程在访问它们之间首先会对互斥锁进行上锁，消费者线程中，当判断没有产品可被消费时 (`g_avail <= 0`)，调用 `pthread_cond_wait()` 使得线程陷入等待状态，等待条件变量，等待生产者制造产品；调用 `pthread_cond_wait()` 后线程阻塞并解锁互斥锁；而在生产者线程中，它的任务是生产产品（使用 `g_avail++` 来模拟），产品生产完成之后，调用 `pthread_mutex_unlock()` 将互斥锁解锁，并调用 `pthread_cond_signal()` 向条件变量发送信号；这将会唤醒处于等待该条件变量的消费者线程，唤醒之后再次自动获取互斥锁，然后再对产品进行消费 (`g_avai--` 模拟)。

12.3.3 条件变量的判断条件

使用条件变量，都会有与之相关的判断条件，通常情况下，会涉及到一个或多个共享变量。譬如在示例代码 12.3.2 中，与条件变量相关的判断是 (`0 >= g_avail`)。细心的读者会发现，在这份示例代码中，我们使用了 `while` 循环、而不是 `if` 语句，来控制对 `pthread_cond_wait()` 的调用，这是为何呢？

必须使用 `while` 循环，而不是 `if` 语句，这是一种通用的设计原则：当线程从 `pthread_cond_wait()` 返回时，并不能确定判断条件的状态，应该立即重新检查判断条件，如果条件不满足，那就继续休眠等待。

从 `pthread_cond_wait()` 返回后，并不能确定判断条件是真还是假，其理由如下：

- 当有多于一个线程在等待条件变量时，任何线程都有可能率先醒来获取互斥锁，率先醒来获取到互斥锁的线程可能会对共享变量进行修改，进而改变判断条件的状态。譬如示例代码 12.3.2 中，如果有两个或更多个消费者线程，当其中一个消费者线程从 `pthread_cond_wait()` 返回后，它会将全局共享变量 `g_avail` 的值变成 0，导致判断条件的状态由真变成假。
- 可能会发出虚假的通知。

12.3.4 条件变量的属性

如前所述，调用 `pthread_cond_init()` 函数初始化条件变量时，可以设置条件变量的属性，通过参数 `attr` 指定。参数 `attr` 指向一个 `pthread_condattr_t` 类型对象，该对象对条件变量的属性进行定义，当然，如果将参数 `attr` 设置为 `NULL`，表示使用默认值来初始化条件变量属性。

关于条件变量的属性本书不打算深入讨论，条件变量包括两个属性：进程共享属性和时钟属性。每个属性都提供了相应的 `get` 方法和 `set` 方法，各位读者如果有兴趣，可自行查阅资料学习，本书不再介绍！

12.4 自旋锁

自旋锁与互斥锁很相似,从本质上说也是一把锁,在访问共享资源之前对自旋锁进行上锁,在访问完成后释放自旋锁(解锁);事实上,从实现方式上来说,互斥锁是基于自旋锁来实现的,所以自旋锁相较于互斥锁更加底层。

如果在获取自旋锁时,自旋锁处于未锁定状态,那么将立即获得锁(对自旋锁上锁);如果在获取自旋锁时,自旋锁已经处于锁定状态了,那么获取锁操作将会在原地“自旋”,直到该自旋锁的持有者释放了锁。由此介绍可知,自旋锁与互斥锁相似,但是互斥锁在无法获取到锁时会让线程陷入阻塞等待状态;而自旋锁在无法获取到锁时,将会在原地“自旋”等待。“自旋”其实就是调用者一直在循环查看该自旋锁的持有者是否已经释放了锁,“自旋”一词因此得名。

自旋锁的不足之处在于:自旋锁一直占用的 CPU,它在未获得锁的情况下,一直处于运行状态(自旋),所以占着 CPU,如果不能在很短的时间内获取锁,这无疑会使 CPU 效率降低。

试图对同一自旋锁加锁两次必然会导致死锁,而试图对同一互斥锁加锁两次不一定会导致死锁,原因在于互斥锁有不同的类型,当设置为 `PTHREAD_MUTEX_ERRORCHECK` 类型时,会进行错误检查,第二次加锁会返回错误,所以不会进入死锁状态。

因此我们要谨慎使用自旋锁,自旋锁通常用于以下情况:需要保护的代码段执行时间很短,这样就会使得持有锁的线程会很快释放锁,而“自旋”等待的线程也只需等待很短的时间;在这种情况下就比较适合使用自旋锁,效率高!

综上所述,再来总结下自旋锁与互斥锁之间的区别:

- 实现方式上的区别:互斥锁是基于自旋锁而实现的,所以自旋锁相较于互斥锁更加底层;
- 开销上的区别:获取不到互斥锁会陷入阻塞状态(休眠),直到获取到锁时被唤醒;而获取不到自旋锁会在原地“自旋”,直到获取到锁;休眠与唤醒开销是很大的,所以互斥锁的开销要远高于自旋锁、自旋锁的效率远高于互斥锁;但如果长时间的“自旋”等待,会使得 CPU 使用效率降低,故自旋锁不适用于等待时间比较长的情况。
- 使用场景的区别:自旋锁在用户态应用程序中使用的比较少,通常在内核代码中使用比较多;因为自旋锁可以在中断服务函数中使用,而互斥锁则不行,在执行中断服务函数时要求不能休眠、不能被抢占(内核中使用自旋锁会自动禁止抢占),一旦休眠意味着执行中断服务函数时主动交出了 CPU 使用权,休眠结束时无法返回到中断服务函数中,这样就会导致死锁!

12.4.1 自旋锁初始化

自旋锁使用 `pthread_spinlock_t` 数据类型表示,当定义自旋锁后,需要使用 `pthread_spin_init()` 函数对其进行初始化,当不再使用自旋锁时,调用 `pthread_spin_destroy()` 函数将其销毁,其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

使用这两个函数需要包含头文件 `<pthread.h>`。

参数 `lock` 指向了需要进行初始化或销毁的自旋锁对象,参数 `pshared` 表示自旋锁的进程共享属性,可以取值如下:

- **PTHREAD_PROCESS_SHARED**: 共享自旋锁。该自旋锁可以在多个进程中的线程之间共享;
- **PTHREAD_PROCESS_PRIVATE**: 私有自旋锁。只有本进程内的线程才能够使用该自旋锁。

这两个函数在调用成功的情况下返回 0;失败将返回一个非 0 值的错误码。

12.4.2 自旋锁加锁和解锁

可以使用 `pthread_spin_lock()` 函数或 `pthread_spin_trylock()` 函数对自旋锁进行加锁, 前者在未获取到锁时一直“自旋”; 对于后者, 如果未能获取到锁, 就立刻返回错误, 错误码为 `EBUSY`。不管以何种方式加锁, 自旋锁都可以使用 `pthread_spin_unlock()` 函数对自旋锁进行解锁。其函数原型如下所示:

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

使用这些函数需要包含头文件 `<pthread.h>`。

参数 `lock` 指向自旋锁对象, 调用成功返回 0, 失败将返回一个非 0 值的错误码。

如果自旋锁处于未锁定状态, 调用 `pthread_spin_lock()` 会将其锁定 (上锁), 如果其它线程已经将自旋锁锁住了, 那本次调用将会“自旋”等待; 如果试图对同一自旋锁加锁两次必然会导致死锁。

使用示例

对示例代码 12.2.1 进行修改, 使用自旋锁替换互斥锁来实现线程同步, 对共享资源的访问进行保护。

示例代码 12.4.1 使用自旋锁实现线程同步

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

static pthread_spinlock_t spin; // 定义自旋锁
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
        pthread_spin_lock(&spin); // 自旋锁上锁

        l_count = g_count;
        l_count++;
        g_count = l_count;

        pthread_spin_unlock(&spin); // 自旋锁解锁
    }

    return (void *)0;
}
```

```
static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000; //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化自旋锁(私有) */
    pthread_spin_init(&spin, PTHREAD_PROCESS_PRIVATE);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid2, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 打印结果 */
}
```

```
printf("g_count = %d\n", g_count);

/* 销毁自旋锁 */
pthread_spin_destroy(&spin);
exit(0);
}
```

运行结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
g_count = 20000000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.4.1 测试结果

将互斥锁替换为自旋锁之后, 测试结果打印也是没有问题的, 并且通过对比可以发现, 替换为自旋锁之后, 程序运行所耗费的时间明显变短了, 说明自旋锁确实比互斥锁效率要高, 但是一定要注意自旋锁所适用的场景。

12.5 读写锁

互斥锁或自旋锁要么是加锁状态、要么是不加锁状态, 而且一次只有一个线程可以对其加锁。读写锁有 3 种状态: 读模式下的加锁状态 (以下简称读加锁状态)、写模式下的加锁状态 (以下简称写加锁状态) 和 不加锁状态 (见), 一次只有一个线程可以占有写模式的读写锁, 但是可以有多个线程同时占有读模式的读写锁。因此可知, 读写锁比互斥锁具有更高的并行性!

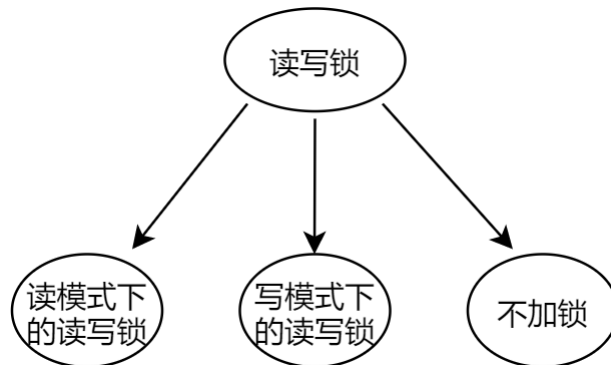


图 12.5.1 读写锁

读写锁有如下两个规则:

- 当读写锁处于写加锁状态时, 在这个锁被解锁之前, 所有试图对这个锁进行加锁操作 (不管是读模式加锁还是以写模式加锁) 的线程都会被阻塞。
- 当读写锁处于读加锁状态时, 所有试图以读模式对它进行加锁的线程都可以加锁成功; 但是任何以写模式对它进行加锁的线程都会被阻塞, 直到所有持有读模式锁的线程释放它们的锁为止。

虽然各操作系统对读写锁的实现各不相同, 但当读写锁处于读模式加锁状态, 而这时有一个线程试图以写模式获取锁时, 该线程会被阻塞; 而如果另一线程以读模式获取锁, 则会成功获取到锁, 对共享资源进行读操作。

所以,读写锁非常适合于对共享数据读的次数远大于写的次数的情况。当读写锁处于写模式加锁状态时,它所保护的数据可以被安全的修改,因为一次只有一个线程可以在写模式下拥有这个锁;当读写锁处于读模式加锁状态时,它所保护的数据就可以被多个获取读模式锁的线程读取。所以在应用程序当中,使用读写锁实现线程同步,当线程需要对共享数据进行读操作时,需要先获取读模式锁(对读模式锁进行加锁),当读取操作完成之后再释放读模式锁(对读模式锁进行解锁);当线程需要对共享数据进行写操作时,需要先获取到写模式锁,当写操作完成之后再释放写模式锁。

读写锁也叫做共享互斥锁。当读写锁是读模式锁住时,就可以说成是共享模式锁住。当它是写模式锁住时,就可以说成是互斥模式锁住。

12.5.1 读写锁初始化

与互斥锁、自旋锁类似,在使用读写锁之前也必须对读写锁进行初始化操作,读写锁使用 `pthread_rwlock_t` 数据类型表示,读写锁的初始化可以使用宏 `PTHREAD_RWLOCK_INITIALIZER` 或者函数 `pthread_rwlock_init()`,其初始化方式与互斥锁相同,譬如使用宏 `PTHREAD_RWLOCK_INITIALIZER` 进行初始化必须在定义读写锁时就对其进行初始化:

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

对于其它方式可以使用 `pthread_rwlock_init()` 函数对其进行初始化,当读写锁不再使用时,需要调用 `pthread_rwlock_destroy()` 函数将其销毁,其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
```

使用这两个函数同样需要包含头文件 `<pthread.h>`,调用成功返回 0,失败将返回一个非 0 值的错误码。

参数 `rwlock` 指向需要进行初始化或销毁的读写锁对象。对于 `pthread_rwlock_init()` 函数,参数 `attr` 是一个 `pthread_rwlockattr_t *` 类型指针,指向 `pthread_rwlockattr_t` 对象。`pthread_rwlockattr_t` 数据类型定义了读写锁的属性(在 12.5.3 小节中介绍),若将参数 `attr` 设置为 `NULL`,则表示将读写锁的属性设置为默认值,在这种情况下其实就等价于 `PTHREAD_RWLOCK_INITIALIZER` 这种方式初始化,而不同之处在于,使用宏不进行错误检查。

当读写锁不再使用时,需要调用 `pthread_rwlock_destroy()` 函数将其销毁。

读写锁初始化使用示例:

```
pthread_rwlock_t rwlock;
```

```
pthread_rwlock_init(&rwlock, NULL);
```

```
.....
```

```
pthread_rwlock_destroy(&rwlock);
```

12.5.2 读写锁上锁和解锁

以读模式对读写锁进行上锁,需要调用 `pthread_rwlock_rdlock()` 函数;以写模式对读写锁进行上锁,需要调用 `pthread_rwlock_wrlock()` 函数。不管是以任何方式锁住读写锁,均可以调用 `pthread_rwlock_unlock()` 函数解锁,其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

使用这些函数需要包含头文件<pthread.h>, 参数 rwlock 指向读写锁对象。调用成功返回 0, 失败返回一个非 0 值的错误码。

当读写锁处于写模式加锁状态时, 其它线程调用 pthread_rwlock_rdlock()或 pthread_rwlock_wrlock()函数均会获取锁失败, 从而陷入阻塞等待状态; 当读写锁处于读模式加锁状态时, 其它线程调用 pthread_rwlock_rdlock()函数可以成功获取到锁, 如果调用 pthread_rwlock_wrlock()函数则不能获取到锁, 从而陷入阻塞等待状态。

如果线程不希望被阻塞, 可以调用 pthread_rwlock_tryrdlock()和 pthread_rwlock_trywrlock()来尝试加锁, 如果不可以获取锁时。这两个函数都会立马返回错误, 错误码为 EBUSY。其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

参数 rwlock 指向需要加锁的读写锁, 加锁成功返回 0, 加锁失败则返回 EBUSY。

使用示例

示例代码 12.5.1 演示了使用读写锁来实现线程同步, 全局变量 g_count 作为线程间的共享变量, 主线程中创建了 5 个读取 g_count 变量的线程, 它们使用同一个函数 read_thread, 这 5 个线程仅仅对 g_count 变量进行读取, 并将其打印出来, 连带打印线程的编号 (1~5); 主线程中还创建了 5 个写 g_count 变量的线程, 它们使用同一个函数 write_thread, write_thread 函数中会将 g_count 变量的值进行累加, 循环 10 次, 每次将 g_count 变量的值在原来的基础上增加 20, 并将其打印出来, 连带打印线程的编号 (1~5)。

示例代码 12.5.1 使用读写锁实现线程同步

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <pthread.h>
```

```
#include <unistd.h>
```

```
#include <string.h>
```

```
static pthread_rwlock_t rwlock; //定义读写锁
```

```
static int g_count = 0;
```

```
static void *read_thread(void *arg)
```

```
{
```

```
    int number = *((int *)arg);
```

```
    int j;
```

```
    for (j = 0; j < 10; j++) {
```

```
        pthread_rwlock_rdlock(&rwlock); //以读模式获取锁
```

```
        printf("读线程<%d>, g_count=%d\n", number+1, g_count);
```

```
        pthread_rwlock_unlock(&rwlock); //解锁
```

```
        sleep(1);
```

```
    }
```



```
    return (void *)0;
}

static void *write_thread(void *arg)
{
    int number = *((int *)arg);
    int j;

    for (j = 0; j < 10; j++) {
        pthread_rwlock_wrlock(&rwlock); //以写模式获取锁
        printf("写线程<%d>, g_count=%d\n", number+1, g_count+=20);
        pthread_rwlock_unlock(&rwlock); //解锁
        sleep(1);
    }

    return (void *)0;
}

static int nums[5] = {0, 1, 2, 3, 4};
int main(int argc, char *argv[])
{
    pthread_t tid[10];
    int j;

    /* 对读写锁进行初始化 */
    pthread_rwlock_init(&rwlock, NULL);

    /* 创建 5 个读 g_count 变量的线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j], NULL, read_thread, &nums[j]);

    /* 创建 5 个写 g_count 变量的线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j+5], NULL, write_thread, &nums[j]);

    /* 等待线程结束 */
    for (j = 0; j < 10; j++)
        pthread_join(tid[j], NULL); //回收线程

    /* 销毁自旋锁 */
    pthread_rwlock_destroy(&rwlock);
    exit(0);
}
```

编译测试, 其打印结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
读线程<2>, g_count=0
读线程<3>, g_count=0
读线程<4>, g_count=0
读线程<5>, g_count=0
读线程<1>, g_count=0
写线程<2>, g_count=20
写线程<3>, g_count=40
写线程<4>, g_count=60
写线程<5>, g_count=80
写线程<1>, g_count=100
读线程<4>, g_count=100
读线程<3>, g_count=100
读线程<5>, g_count=100
读线程<1>, g_count=100
读线程<2>, g_count=100
写线程<3>, g_count=120
写线程<5>, g_count=140
写线程<1>, g_count=160
写线程<2>, g_count=180
写线程<4>, g_count=200
```

图 12.5.2 测试结果

在这个例子中, 我们演示了读写锁的使用, 但仅作为演示使用, 在实际的应用编程中, 需要根据应用场景来选择是否使用读写锁。

12.5.3 读写锁的属性

读写锁与互斥锁类似, 也是有属性的, 读写锁的属性使用 `pthread_rwlockattr_t` 数据类型来表示, 当定义 `pthread_rwlockattr_t` 对象时, 需要使用 `pthread_rwlockattr_init()` 函数对其进行初始化操作, 初始化会将 `pthread_rwlockattr_t` 对象定义的各个读写锁属性初始化为默认值; 当不再使用 `pthread_rwlockattr_t` 对象时, 需要调用 `pthread_rwlockattr_destroy()` 函数将其销毁, 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

参数 `attr` 指向需要进行初始化或销毁的 `pthread_rwlockattr_t` 对象; 函数调用成功返回 0, 失败将返回一个非 0 值的错误码。

读写锁只有一个属性, 那便是进程共享属性, 它与互斥锁以及自旋锁的进程共享属性相同。Linux 下提供了相应的函数用于设置或获取读写锁的共享属性。函数 `pthread_rwlockattr_getpshared()` 用于从 `pthread_rwlockattr_t` 对象中获取共享属性, 函数 `pthread_rwlockattr_setpshared()` 用于设置 `pthread_rwlockattr_t` 对象中的共享属性, 其函数原型如下所示:

```
#include <pthread.h>
```

```
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);
```

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

函数 `pthread_rwlockattr_getpshared()` 参数和返回值:

attr: 指向 `pthread_rwlockattr_t` 对象;

pshared: 调用 `pthread_rwlockattr_getpshared()` 获取共享属性, 将其保存在参数 `pshared` 所指向的内存中;

返回值: 成功返回 0, 失败将返回一个非 0 值的错误码。

函数 `pthread_rwlockattr_setpshared()` 参数和返回值:

attr: 指向 pthread_rwlockattr_t 对象;

pshared: 调用 pthread_rwlockattr_setpshared() 设置读写锁的共享属性, 将其设置为参数 pshared 指定的值。参数 pshared 可取值如下:

- **PTHREAD_PROCESS_SHARED:** 共享读写锁。该读写锁可以在多个进程中的线程之间共享;
- **PTHREAD_PROCESS_PRIVATE:** 私有读写锁。只有本进程内的线程才能够使用该读写锁, 这是读写锁共享属性的默认值。

返回值: 调用成功的情况下返回 0; 失败将返回一个非 0 值的错误码。

使用方式如下:

```
pthread_rwlock_t rwlock;    //定义读写锁
pthread_rwlockattr_t attr;  //定义读写锁属性

/* 初始化读写锁属性对象 */
pthread_rwlockattr_init(&attr);

/* 将进程共享属性设置为 PTHREAD_PROCESS_PRIVATE */
pthread_rwlockattr_setpshared(&attr, PTHREAD_PROCESS_PRIVATE);

/* 初始化读写锁 */
pthread_rwlock_init(&rwlock, &attr);

.....

/* 使用完之后 */
pthread_rwlock_destroy(&rwlock);    //销毁读写锁
pthread_rwlockattr_destroy(&attr);  //销毁读写锁属性对象
```

12.6 总结

本章介绍了线程同步的几种不同的方法, 包括互斥锁、条件变量、自旋锁以及读写锁, 当然, 除此之外, 线程同步的方法其实还有很多, 譬如信号量、屏障等等, 如果大家有兴趣可以自己查阅相关书籍进行学习。在实际应用开发当中, 用的最多的还是互斥锁和条件变量, 当然具体使用哪一种线程同步方法还是得根据场景来进行选择, 方能达到事半功倍的效果!

12.7 练习

第十三章 高级 I/O

本章再次回到文件 I/O 相关话题的讨论, 将会介绍文件 I/O 当中的一些高级用法, 以应对不同应用场合的需求, 主要包括: 非阻塞 I/O、I/O 多路复用、异步 I/O、存储映射 I/O 以及文件锁, 我们统统把它们放到本章《高级 I/O》中进行讨论、学习。

本章将会讨论如下主题内容。

- 阻塞 I/O 与非阻塞 I/O;
- 阻塞 I/O 所带来的困境;
- 非阻塞 I/O 以轮训方式访问多个设备;
- 何为 I/O 多路复用以及原理;
- 何为异步 I/O 以及原理;
- 存储映射 I/O;
- 文件加锁。

13.1 非阻塞 I/O

关于“阻塞”一词前面已经给大家多次提到，阻塞其实就是进入了休眠状态，交出了 CPU 控制权。前面所学习过的函数，譬如 `wait()`、`pause()`、`sleep()` 等函数都会进入阻塞，本小节来聊一聊关于阻塞式 I/O 与非阻塞式 I/O。

阻塞式 I/O 顾名思义就是对文件的 I/O 操作（读写操作）是阻塞式的，非阻塞式 I/O 同理就是对文件的 I/O 操作是非阻塞的。这样说大家可能不太明白，这里举个例子，譬如对于某些文件类型（读管道文件、网络设备文件和字符设备文件），当对文件进行读操作时，如果数据未准备好、文件当前无数据可读，那么读操作可能会使调用者阻塞，直到有数据可读时才会被唤醒，这就是阻塞式 I/O 常见的一种表现；如果是非阻塞式 I/O，即使没有数据可读，也不会被阻塞、而是会立马返回错误！

普通文件的读写操作是不会阻塞的，不管读写多少个字节数据，`read()` 或 `write()` 一定会在有限的时间内返回，所以普通文件一定是以非阻塞的方式进行 I/O 操作，这是普通文件本质上决定的；但是对于某些文件类型，譬如上面所介绍的管道文件、设备文件等，它们既可以使用阻塞式 I/O 操作，也可以使用非阻塞式 I/O 进行操作。

13.1.1 阻塞 I/O 与非阻塞 I/O 读文件

本小节我们将分别演示使用阻塞式 I/O 和非阻塞式 I/O 对文件进行读操作，在调用 `open()` 函数打开文件时，为参数 `flags` 指定 `O_NONBLOCK` 标志，`open()` 调用成功后，后续的 I/O 操作将以非阻塞式方式进行；这就是非阻塞 I/O 的打开方式，如果未指定 `O_NONBLOCK` 标志，则默认使用阻塞式 I/O 进行操作。

对于普通文件来说，指定与未指定 `O_NONBLOCK` 标志对其是没有影响，普通文件的读写操作是不会阻塞的，它总是以非阻塞的方式进行 I/O 操作，这是普通文件本质上决定的，前面已经给大家进行了说明。

本小节我们将以读取鼠标为例，使用两种 I/O 方式进行读取，来进行对比，鼠标是一种输入设备，其对应的设备文件在 `/dev/input/` 目录下，如下所示：

```
dt@dt-virtual-machine:/dev/input$ pwd
/dev/input
dt@dt-virtual-machine:/dev/input$
dt@dt-virtual-machine:/dev/input$ ls -lh
总用量 0
drwxr-xr-x 2 root root    80 1月 26 09:55 by-id
drwxr-xr-x 2 root root   140 1月 26 09:55 by-path
crw-rw---- 1 root input 13, 64 1月 26 09:55 event0
crw-rw---- 1 root input 13, 65 1月 26 09:55 event1
crw-rw---- 1 root input 13, 66 1月 26 09:55 event2
crw-rw---- 1 root input 13, 67 1月 26 09:55 event3
crw-rw---- 1 root input 13, 68 1月 26 09:55 event4
crw-rw---- 1 root input 13, 63 1月 26 09:55 mice
crw-rw---- 1 root input 13, 32 1月 26 09:55 mouse0
crw-rw---- 1 root input 13, 33 1月 26 09:55 mouse1
crw-rw---- 1 root input 13, 34 1月 26 09:55 mouse2
dt@dt-virtual-machine:/dev/input$
```

图 13.1.1 输入设备对应的设备文件

通常情况下是 `mouseX`（X 表示序号 0、1、2），但也不一定，也有可能是 `eventX`，如何确定到底是哪个设备文件，可以通过对设备文件进行读取来判断，譬如使用 `od` 命令：

```
sudo od -x /dev/input/event3
```

Tips: 需要添加 `sudo`，在 Ubuntu 系统下，普通用户是无法对设备文件进行读取或写入操作。

当执行命令之后，移动鼠标或按下鼠标、松开鼠标都会在终端打印出相应的数据，如下所示：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ sudo od -x /dev/input/event3
0000000 688b 607a 0000 0000 37df 000b 0000 0000
0000020 0003 0000 6e29 0000 688b 607a 0000 0000
0000040 37df 000b 0000 0000 0003 0001 4fab 0000
0000060 688b 607a 0000 0000 37df 000b 0000 0000
0000100 0000 0000 0000 0000 688b 607a 0000 0000
0000120 59bf 000b 0000 0000 0003 0000 6f18 0000
0000140 688b 607a 0000 0000 59bf 000b 0000 0000
0000160 0000 0000 0000 0000 688b 607a 0000 0000
0000200 96a7 000b 0000 0000 0003 0000 726f 0000
0000220 688b 607a 0000 0000 96a7 000b 0000 0000
0000240 0003 0001 4dbc 0000 688b 607a 0000 0000
0000260 96a7 000b 0000 0000 0000 0000 0000 0000
0000300 688b 607a 0000 0000 b895 000b 0000 0000
0000320 0003 0000 73c5 0000 688b 607a 0000 0000
0000340 b895 000b 0000 0000 0003 0001 4ce8 0000
0000360 688b 607a 0000 0000 b895 000b 0000 0000
0000400 0000 0000 0000 0000 688b 607a 0000 0000
0000420 da7b 000b 0000 0000 0003 0000 74d7 0000
0000440 688b 607a 0000 0000 da7b 000b 0000 0000
0000460 0003 0001 4c5b 0000 688b 607a 0000 0000
0000500 da7b 000b 0000 0000 0000 0000 0000 0000
0000520 688b 607a 0000 0000 fd38 000b 0000 0000
0000540 0003 0000 7560 0000 688b 607a 0000 0000
0000560 fd38 000b 0000 0000 0003 0001 4c14 0000
```

图 13.1.2 读取鼠标打印信息

如果没有打印信息,那么这个设备文件就不是鼠标对应的设备文件,那么就换一个设备文件再次测试,这样就会帮助你找到鼠标设备文件。笔者使用的 Ubuntu 系统,对应的鼠标设备文件是/dev/input/event3。接下来我们编写一个测试程序,使用阻塞式 I/O 读取鼠标。

示例代码 13.1.1 演示了以阻塞方式读取鼠标,调用 open()函数打开鼠标设备文件"/dev/input/event3",以只读方式打开,没有指定 O_NONBLOCK 标志,说明使用的是阻塞式 I/O;程序中只调用了一次 read()读取鼠标。

示例代码 13.1.1 阻塞式 I/O 读取鼠标数据

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开文件 */
    fd = open("/dev/input/event3", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }
}
```

```

}

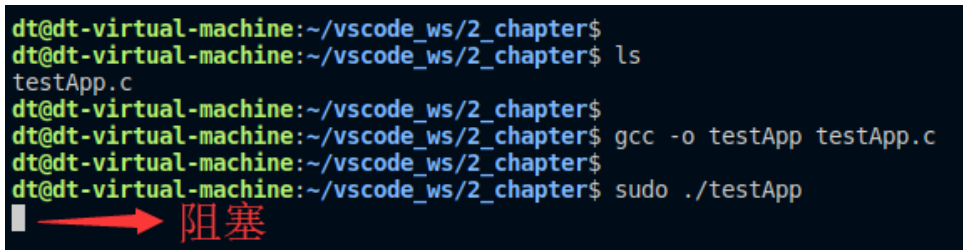
/* 读文件 */
memset(buf, 0, sizeof(buf));
ret = read(fd, buf, sizeof(buf));
if (0 > ret) {
    perror("read error");
    close(fd);
    exit(-1);
}

printf("成功读取<%d>个字节数据\n", ret);

/* 关闭文件 */
close(fd);
exit(0);
}

```

编译上述示例代码进行测试:



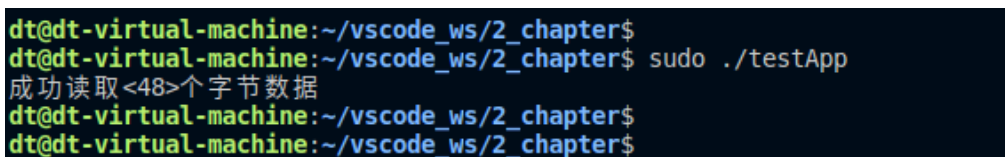
```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
█ → 阻塞

```

图 13.1.3 阻塞

执行程序之后,发现程序没有立即结束,而是一直占用了终端,没有输出信息,原因在于调用 `read()` 之后进入了阻塞状态,因为当前鼠标没有数据可读;如果此时我们移动鼠标、或者按下鼠标上的任何一个按键,阻塞会结束, `read()` 会成功读取到数据并返回,如下所示:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.1.4 移动鼠标

打印信息提示,此次 `read` 成功读取了 48 个字节,程序当中我们明明要求读取的是 100 个字节,为什么这里只读取到了 48 个字节?关于这个问题将会在第二篇内容当中进行介绍,这里暂时先不去理会这个问题。

接下来,我们将示例代码 13.1.1 修改成非阻塞式 I/O,如下所示:

示例代码 13.1.2 非阻塞式 I/O 读取鼠标数据

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

```

```
#include <string.h>

int main(void)
{
    char buf[100];
    int fd, ret;

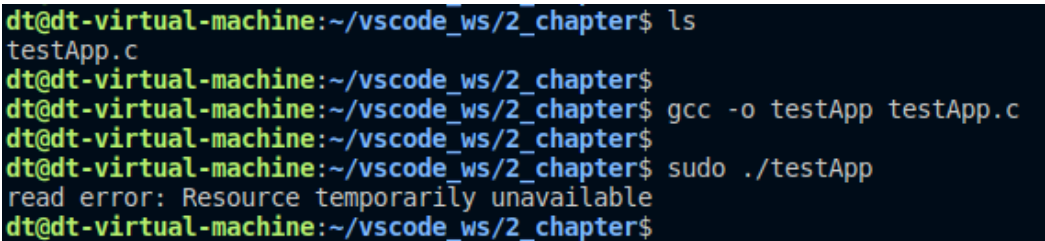
    /* 打开文件 */
    fd = open("/dev/input/event3", O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 读文件 */
    memset(buf, 0, sizeof(buf));
    ret = read(fd, buf, sizeof(buf));
    if (0 > ret) {
        perror("read error");
        close(fd);
        exit(-1);
    }

    printf("成功读取<%d>个字节数据\n", ret);

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

修改方法很简单, 只需在调用 `open()` 函数时指定 `O_NONBLOCK` 标志即可, 对上述示例代码进行编译测试:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
read error: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.1.5 非阻塞

执行程序之后, 程序立马就结束了, 并且调用 `read()` 返回错误, 提示信息为 "Resource temporarily unavailable", 意思就是说资源暂时不可用; 原因在于调用 `read()` 时, 如果鼠标并没有移动或者被按下 (没有发生输入事件), 是没有数据可读, 故而导致失败返回, 这就是非阻塞 I/O。

可以对示例代码 13.1.2 进行修改, 使用轮训方式不断地去读取, 直到鼠标有数据可读, `read()` 将会成功返回:


```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开文件 */
    fd = open("/dev/input/event3", O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 读文件 */
    memset(buf, 0, sizeof(buf));
    for (;;) {
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret) {
            printf("成功读取<%d>个字节数据\n", ret);
            close(fd);
            exit(0);
        }
    }
}
```

具体的执行的效果便不再演示了，各位读者自己动手试试。

13.1.2 阻塞 I/O 的优点与缺点

当对文件进行读取操作时，如果文件当前无数据可读，那么阻塞式 I/O 会将调用者应用程序挂起、进入休眠阻塞状态，直到有数据可读时才会解除阻塞；而对于非阻塞 I/O，应用程序不会被挂起，而是会立即返回，它要么一直轮训等待，直到数据可读，要么直接放弃！

所以阻塞式 I/O 的优点在于能够提升 CPU 的处理效率，当自身条件不满足时，进入阻塞状态，交出 CPU 资源，将 CPU 资源让给别人使用；而非阻塞式则是抓紧利用 CPU 资源，譬如不断地去轮训，这样就会导致该程序占用了非常高的 CPU 使用率！

执行示例代码 13.1.3 对应的程序时，通过 top 命令可以发现该程序的占用了非常高的 CPU 使用率，如下所示：

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
115034	root	20	0	4220	640	576	R	99.7	0.0	0:05.59	testApp
22997	dt	20	0	14.999g	673092	69024	S	7.9	16.8	110:21.32	code
22342	dt	20	0	2450320	61632	29976	S	1.0	1.5	138:39.01	compiz
22972	dt	20	0	619940	103516	24700	S	1.0	2.6	40:45.44	code
21824	root	20	0	499880	40124	15192	S	0.7	1.0	40:25.07	Xorg
22940	dt	20	0	4951048	103596	42420	S	0.7	2.6	5:53.89	code
8	root	20	0	0	0	0	I	0.3	0.0	20:32.31	rcu_sched
115035	dt	20	0	43672	3768	3044	R	0.3	0.1	0:00.04	top
1	root	20	0	185416	4676	3220	S	0.0	0.1	0:44.53	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:01.50	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
7	root	20	0	0	0	0	S	0.0	0.0	0:01.79	ksoftirqd/0
9	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_bh
10	root	rt	0	0	0	0	S	0.0	0.0	0:00.05	migration/0
11	root	rt	0	0	0	0	S	0.0	0.0	0:09.46	watchdog/0

图 13.1.6 CPU 占用率

其 CPU 占用率几乎达到了 100%，在一个系统当中，一个进程的 CPU 占用率这么高是一件非常危险的事情。而示例代码 13.1.1 这种阻塞式方式，其 CPU 占用率几乎为 0，所以就本章的所举例子来说，阻塞式 I/O 绝地要优于非阻塞式 I/O，那既然如此，我们为何还要介绍非阻塞式 I/O 呢？下一小节我们将通过一个例子给大家介绍，阻塞式 I/O 的困境！

13.1.3 使用非阻塞 I/O 实现并发读取

上一小节给大家所举的例子当中，只读取了鼠标的的数据，如果要在程序当中同时读取鼠标和键盘，那又该如何呢？本小节我们将分别演示使用阻塞式 I/O 和非阻塞式 I/O 同时读取鼠标和键盘；同理键盘也是一种输入类设备，但是键盘是标准输入设备 `stdin`，进程会自动从父进程中继承标准输入、标准输出以及标准错误，标准输入设备对应的文件描述符为 0，所以在程序当中直接使用即可，不需要再调用 `open` 打开。

首先我们使用阻塞式方式同时读取鼠标和键盘，示例代码如下所示：

示例代码 13.1.4 阻塞式同时读取鼠标和键盘

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define MOUSE "/dev/input/event3"

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }
}
```

```
/* 读鼠标 */
memset(buf, 0, sizeof(buf));
ret = read(fd, buf, sizeof(buf));
printf("鼠标: 成功读取<%d>个字节数据\n", ret);

/* 读键盘 */
memset(buf, 0, sizeof(buf));
ret = read(0, buf, sizeof(buf));
printf("键盘: 成功读取<%d>个字节数据\n", ret);

/* 关闭文件 */
close(fd);
exit(0);
}
```

上述程序中先读了鼠标, 在接着读键盘, 所以由此可知, 在实际测试当中, 需要先动鼠标在按键盘 (按下键盘上的按键、按完之后按下回车), 这样才能既成功读取鼠标、又成功读取键盘, 程序才能够顺利运行结束。因为 read 此时是阻塞式读取, 先读取了鼠标, 没有数据可读将会一直被阻塞, 后面的读取键盘将得不到执行。

这就是阻塞式 I/O 的一个困境, 无法实现并发读取 (同时读取), 主要原因在于阻塞, 那如何解决这个问题呢? 当然大家可能会想到使用多线程, 一个线程读取鼠标、另一个线程读取键盘, 亦或者创建一个子进程, 父进程读取鼠标、子进程读取键盘等方法, 当然这些方法自然可以解决, 但不是我们要学习的重点。

既然阻塞 I/O 存在这样一个困境, 那我们可以使用非阻塞式 I/O 解决它, 将示例代码 13.1.4 修改为非阻塞式方式同时读取鼠标和键盘。使用 open() 打开得到的文件描述符, 调用 open() 时指定 O_NONBLOCK 标志将其设置为非阻塞式 I/O; 因为标准输入文件描述符 (键盘) 是从其父进程进程而来, 并不是在我们的程序中调用 open() 打开得到的, 那如何将标准输入设置为非阻塞 I/O, 可以使用 3.10.1 小节中给大家介绍的 fcntl() 函数, 具体使用方法在该小节中已有详细介绍, 这里不再重述! 可通过如下代码将标准输入 (键盘) 设置为非阻塞方式:

```
int flag;

flag = fcntl(0, F_GETFL);          //先获取原来的 flag
flag |= O_NONBLOCK;               //将 O_NONBLOCK 标志添加到 flag
fcntl(0, F_SETFL, flag);          //重新设置 flag
```

示例代码 13.1.5 演示了以非阻塞方式同时读取鼠标和键盘。

示例代码 13.1.5 非阻塞式方式同时读取鼠标和键盘

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define MOUSE          "/dev/input/event3"
```

```
int main(void)
{
    char buf[100];
    int fd, ret, flag;

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将键盘设置为非阻塞方式 */
    flag = fcntl(0, F_GETFL); //先获取原来的 flag
    flag |= O_NONBLOCK; //将 O_NONBLOCK 标准添加到 flag
    fcntl(0, F_SETFL, flag); //重新设置 flag

    for (;;) {
        /* 读鼠标 */
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret)
            printf("鼠标: 成功读取<%d>个字节数据\n", ret);

        /* 读键盘 */
        ret = read(0, buf, sizeof(buf));
        if (0 < ret)
            printf("键盘: 成功读取<%d>个字节数据\n", ret);
    }

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

将读取鼠标和读取键盘操作放入到一个循环中,通过轮训方式来实现并发读取鼠标和键盘,对上述代码进行编译,测试结果:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
das
键盘: 成功读取<4>个字节数据
dasd
键盘: 成功读取<5>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
das
键盘: 成功读取<4>个字节数据
das
键盘: 成功读取<4>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
das
键盘: 成功读取<4>个字节数据

```

图 13.1.7 测试结果

这样就解决了示例代码 13.1.4 所出现的问题, 不管是先动鼠标还是先按键盘都可以成功读取到相应数据。

虽然使用非阻塞 I/O 方式解决了示例代码 13.1.4 出现的问题, 但由于程序当中使用轮训方式, 故而会使得该程序的 CPU 占用率特别高, 终归还是不太安全, 会对整个系统产生很大的副作用, 如何解决这样的问题呢? 我们将在下一小节向大家介绍。

13.2 I/O 多路复用

上一小节虽然使用非阻塞式 I/O 解决了阻塞式 I/O 情况下并发读取文件所出现的问题, 但依然不够完美, 使得程序的 CPU 占用率特别高。解决这个问题, 就要用到本小节将要介绍的 I/O 多路复用方法。

13.2.1 何为 I/O 多路复用

I/O 多路复用 (IO multiplexing) 它通过一种机制, 可以监视多个文件描述符, 一旦某个文件描述符 (也就是某个文件) 可以执行 I/O 操作时, 能够通知应用程序进行相应的读写操作。I/O 多路复用技术是为了解决: 在并发式 I/O 场景中进程或线程阻塞到某个 I/O 系统调用而出现的, 使进程不阻塞于某个特定的 I/O 系统调用。

由此可知, I/O 多路复用一般用于并发式的非阻塞 I/O, 也就是多路非阻塞 I/O, 譬如程序中既要读取鼠标、又要读取键盘, 多路读取。

我们可以采用两个功能几乎相同的系统调用来执行 I/O 多路复用操作, 分别是系统调用 `select()` 和 `poll()`。这两个函数基本是一样的, 细节特征上存在些许差别!

I/O 多路复用存在一个非常明显的特征: 外部阻塞式, 内部监视多路 I/O。

13.2.2 select() 函数介绍

系统调用 `select()` 可用于执行 I/O 多路复用操作, 调用 `select()` 会一直阻塞, 直到某一个或多个文件描述符成为就绪态 (可以读或写)。其函数原型如下所示:

```
#include <sys/select.h>
```

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

使用该函数需要包含头文件 `<sys/select.h>`。

可以看出 `select()` 函数的参数比较多, 其中参数 `readfds`、`writelfds` 以及 `exceptfds` 都是 `fd_set` 类型指针, 指向一个 `fd_set` 类型对象, `fd_set` 数据类型是一个文件描述符的集合体, 所以参数 `readfds`、`writelfds` 以及 `exceptfds` 都是指向文件描述符集合的指针, 这些参数按照如下方式使用:

- `readfds` 是用来检测读是否就绪 (是否可读) 的文件描述符集合;
- `writelfds` 是用来检测写是否就绪 (是否可写) 的文件描述符集合;
- `exceptfds` 是用来检测异常情况是否发生的文件描述符集合。

Tips: 异常情况并不是在文件描述符上出现了一些错误。

`fd_set` 数据类型是以位掩码的形式来实现的, 但是, 我们并不需要关心这些细节、无需关心该结构体成员信息, 因为 Linux 提供了四个宏用于对 `fd_set` 类型对象进行操作, 所有关于文件描述符集合的操作都是通过这四个宏来完成的: `FD_CLR()`、`FD_ISSET()`、`FD_SET()`、`FD_ZERO()`, 稍后介绍!

如果对 `readfds`、`writelfds` 以及 `exceptfds` 中的某些事件不感兴趣, 可将其设置为 `NULL`, 这表示对相应条件不关心。如果这三个参数都设置为 `NULL`, 则可以将 `select()` 当做一个类似于 `sleep()` 休眠的函数来使用, 通过 `select()` 函数的最后一个参数 `timeout` 来设置休眠时间。

`select()` 函数的第一个参数 `nfds` 通常表示最大文件描述符编号值加 1, 考虑 `readfds`、`writelfds` 以及 `exceptfds` 这三个文件描述符集合, 在 3 个描述符集中找出最大描述符编号值, 然后加 1, 这就是参数 `nfds`。

`select()` 函数的最后一个参数 `timeout` 可用于设定 `select()` 阻塞的时间上限, 控制 `select` 的阻塞行为, 可将 `timeout` 参数设置为 `NULL`, 表示 `select()` 将会一直阻塞、直到某一个或多个文件描述符成为就绪态; 也可将其指向一个 `struct timeval` 结构体对象, 该结构体在示例代码 5.6.3 有详细介绍, 这里不再重述!

如果参数 `timeout` 指向的 `struct timeval` 结构体对象中的两个成员变量都为 0, 那么此时 `select()` 函数不会阻塞, 它只是简单地轮训指定的文件描述符集合, 看看其中是否有就绪的文件描述符并立刻返回。否则, 参数 `timeout` 将为 `select()` 指定一个等待 (阻塞) 时间的上限值, 如果在阻塞期间内, 文件描述符集合中的某一个或多个文件描述符成为就绪态, 将会结束阻塞并返回; 如果超过了阻塞时间的上限值, `select()` 函数将会返回!

`select()` 函数将阻塞知道有以下事情发生:

- `readfds`、`writelfds` 或 `exceptfds` 指定的文件描述符中至少有一个称为就绪态;
- 该调用被信号处理函数中断;
- 参数 `timeout` 中指定的时间上限已经超时。

FD_CLR()、FD_ISSET()、FD_SET()、FD_ZERO()

文件描述符集合的所有操作都可以通过这四个宏来完成, 这些宏定义如下所示:

```
#include <sys/select.h>
```

```
void FD_CLR(int fd, fd_set *set);
```

```
int  FD_ISSET(int fd, fd_set *set);
```

```
void FD_SET(int fd, fd_set *set);
```

```
void FD_ZERO(fd_set *set);
```

这些宏按照如下方式工作:

- `FD_ZERO()` 将参数 `set` 所指向的集合初始化为空;
- `FD_SET()` 将文件描述符 `fd` 添加到参数 `set` 所指向的集合中;
- `FD_CLR()` 将文件描述符 `fd` 从参数 `set` 所指向的集合中移除;
- 如果文件描述符 `fd` 是参数 `set` 所指向的集合中的成员, 则 `FD_ISSET()` 返回 `true`, 否则返回 `false`。

文件描述符集合有一个最大容量限制, 有常量 `FD_SETSIZE` 来决定, 在 Linux 系统下, 该常量的值为 1024。在定义一个文件描述符集合之后, 必须用 `FD_ZERO()` 宏将其进行初始化操作, 然后再向集合中添加我们关心的各个文件描述符, 例如:

```
fd_set fset;           //定义文件描述符集合
```

```
FD_ZERO(&fset);       //将集合初始化为空
FD_SET(3, &fset);     //向集合中添加文件描述符 3
FD_SET(4, &fset);     //向集合中添加文件描述符 4
FD_SET(5, &fset);     //向集合中添加文件描述符 5
```

在调用 `select()` 函数之后, `select()` 函数内部会修改 `readfds`、`writfds`、`exceptfds` 这些集合, 当 `select()` 函数返回时, 它们包含的就是已处于就绪态的文件描述符集合了。譬如在调用 `select()` 函数之前, `readfds` 所指向的集合中包含了 3、4、5 这三个文件描述符, 当调用 `select()` 函数之后, 假设 `select()` 返回时, 只有文件描述符 4 已经处于就绪态了, 那么此时 `readfds` 指向的集合中就只包含了文件描述符 4。所以由此可知, 如果要在循环中重复调用 `select()`, 我们必须保证每次都要重新初始化并设置 `readfds`、`writfds`、`exceptfds` 这些集合。

select() 函数的返回值

`select()` 函数有三种可能的返回值, 会返回如下三种情况中的一种:

- 返回 -1 表示有错误发生, 并且会设置 `errno`。可能的错误码包括 `EBADF`、`EINTR`、`EINVAL`、`EINVAL` 以及 `ENOMEM`, `EBADF` 表示 `readfds`、`writfds` 或 `exceptfds` 中有一个文件描述符是非法的; `EINTR` 表示该函数被信号处理函数中断了, 其它错误大家可以自己去看, 在 `man` 手册都有相信的记录。
- 返回 0 表示在任何文件描述符成为就绪态之前 `select()` 调用已经超时, 在这种情况下, `readfds`, `writfds` 以及 `exceptfds` 所指向的文件描述符集合都会被清空。
- 返回一个正整数表示有一个或多个文件描述符已达到就绪态。返回值表示处于就绪态的文件描述符的个数, 在这种情况下, 每个返回的文件描述符集合都需要检查, 通过 `FD_ISSET()` 宏进行检查, 以此找出发生的 I/O 事件是什么。如果同一个文件描述符在 `readfds`, `writfds` 以及 `exceptfds` 中同时被指定, 且它多于多个 I/O 事件都处于就绪态的话, 那么就会被统计多次, 换句话说, `select()` 返回三个集合中被标记为就绪态的文件描述符的总数。

使用示例

示例代码 13.2.1 演示了使用 `select()` 函数来实现 I/O 多路复用操作, 同时读取键盘和鼠标。程序中将鼠标和键盘配置为非阻塞 I/O 方式, 本程序对数据进行了 5 次读取, 通过 `while` 循环来实现。由于在 `while` 循环中会重复调用 `select()` 函数, 所以每次调用之前需要对 `rdfds` 进行初始化以及添加鼠标和键盘对应的文件描述符。

该程序中, `select()` 函数的参数 `timeout` 被设置为 `NULL`, 并且我们只关心鼠标或键盘是否有数据可读, 所以将参数 `writfds` 和 `exceptfds` 也设置为 `NULL`。执行 `select()` 函数时, 如果鼠标和键盘均无数据可读, 则 `select()` 调用会陷入阻塞, 直到发生输入事件 (鼠标移动、键盘上的按键按下或松开) 才会返回。

示例代码 13.2.1 使用 `select` 实现同时读取键盘和鼠标

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/select.h>

#define MOUSE        "/dev/input/event3"
```

```
int main(void)
{
    char buf[100];
    int fd, ret = 0, flag;
    fd_set rdfs;
    int loops = 5;

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将键盘设置为非阻塞方式 */
    flag = fcntl(0, F_GETFL); //先获取原来的 flag
    flag |= O_NONBLOCK; //将 O_NONBLOCK 标准添加到 flag
    fcntl(0, F_SETFL, flag); //重新设置 flag

    /* 同时读取键盘和鼠标 */
    while (loops--) {
        FD_ZERO(&rdfs);
        FD_SET(0, &rdfs); //添加键盘
        FD_SET(fd, &rdfs); //添加鼠标

        ret = select(fd + 1, &rdfs, NULL, NULL, NULL);
        if (0 > ret) {
            perror("select error");
            goto out;
        }
        else if (0 == ret) {
            fprintf(stderr, "select timeout.\n");
            continue;
        }

        /* 检查键盘是否为就绪态 */
        if(FD_ISSET(0, &rdfs)) {
            ret = read(0, buf, sizeof(buf));
            if (0 < ret)
                printf("键盘: 成功读取<%d>个字节数据\n", ret);
        }

        /* 检查鼠标是否为就绪态 */
    }
}
```



```

if(FD_ISSET(fd, &rdfs)) {
    ret = read(fd, buf, sizeof(buf));
    if (0 < ret)
        printf("鼠标: 成功读取<%d>个字节数据\n", ret);
    }
}

```

out:

```

/* 关闭文件 */
close(fd);
exit(ret);
}

```

程序中分析 select()函数的返回值 ret, 只有当 ret 大于 0 时才表示有文件描述符处于就绪态, 并将这些处于就绪态的文件描述符通过 rdfs 集合返回出来, 程序中使用 FD_ISSET()宏检查返回的 rdfs 集合中是否包含鼠标文件描述符以及键盘文件描述符, 如果包含则表示可以读取数据了。

编译运行:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码:
das
select ret = 1
键盘: 成功读取<4>个字节数据
das
select ret = 1
键盘: 成功读取<4>个字节数据
select ret = 1
鼠标: 成功读取<48>个字节数据
select ret = 1
鼠标: 成功读取<48>个字节数据
sa
select ret = 1
键盘: 成功读取<3>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.2.1 测试结果

示例代码 13.2.1 将鼠标和键盘都设置为了非阻塞 I/O 方式, 其实设置为阻塞 I/O 方式也是可以的, 因为 select()返回时意味着此时数据是可读取的, 所以以非阻塞和阻塞两种方式读取数据均不会发生阻塞。

13.2.3 poll()函数介绍

系统调用 poll()与 select()函数很相似, 但函数接口有所不同。在 select()函数中, 我们提供三个 fd_set 集合, 在每个集合中添加我们关心的文件描述符; 而在 poll()函数中, 则需要构造一个 struct pollfd 类型的数组, 每个数组元素指定一个文件描述符以及我们对该文件描述符所关心的条件 (数据可读、可写或异常情况)。poll()函数原型如下所示:

```
#include <poll.h>
```

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

使用该函数需要包含头文件<poll.h>。

函数参数含义如下:

fds: 指向一个 struct pollfd 类型的数组, 数组中的每个元素都会指定一个文件描述符以及我们对该文件描述符所关心的条件, 稍后介绍 struct pollfd 结构体类型。

nfds: 参数 nfds 指定了 fds 数组中的元素个数, 数据类型 nfds_t 实际为无符号整形。

timeout: 该参数与 select() 函数的 timeout 参数相似, 用于决定 poll() 函数的阻塞行为, 具体用法如下:

- 如果 timeout 等于 -1, 则 poll() 会一直阻塞 (与 select() 函数的 timeout 等于 NULL 相同), 直到 fds 数组中列出的文件描述符有一个达到就绪态或者捕获到一个信号时返回。
- 如果 timeout 等于 0, poll() 不会阻塞, 只是执行一次检查看看哪个文件描述符处于就绪态。
- 如果 timeout 大于 0, 则表示设置 poll() 函数阻塞时间的上限值, 意味着 poll() 函数最多阻塞 timeout 毫秒, 直到 fds 数组中列出的文件描述符有一个达到就绪态或者捕获到一个信号为止。

struct pollfd 结构体

struct pollfd 结构体如下所示:

示例代码 13.2.2 struct pollfd 结构体

```
struct pollfd {
    int    fd;           /* file descriptor */
    short events;       /* requested events */
    short revents;      /* returned events */
};
```

fd 是一个文件描述符, struct pollfd 结构体中的 events 和 revents 都是位掩码, 调用者初始化 events 来指定需要为文件描述符 fd 做检查的事件。当 poll() 函数返回时, revents 变量由 poll() 函数内部进行设置, 用于说明文件描述符 fd 发生了哪些事件 (注意, poll() 没有更改 events 变量), 我们可以对 revents 进行检查, 判断文件描述符 fd 发生了什么事情。

应将每个数组元素的 events 成员设置为表 13.2.1 中所示的一个或几个标志, 多个标志通过位或运算符 (|) 组合起来, 通过这些值告诉内核我们关心的是该文件描述符的哪些事件。同样, 返回时, revents 变量由内核设置为表 13.2.1 中所示的一个或几个标志。

表 13.2.1 poll 的 events 和 revents 标志

标志名	输入至 events	从 revents 得到结果	说明
POLLIN	●	●	有数据可以读取
POLLRDNORM	●	●	相等于 POLLIN
POLLRDBAND	●	●	可以读取优先级数据 (Linux 上通常不使用)
POLLPRI	●	●	可读取高优先级数据
POLLRDHUP	●	●	对端套接字关闭
POLLOUT	●	●	可写入数据
POLLWRNORM	●	●	相等于 POLLOUT
POLLWRBAND	●	●	优先级数据可写入
POLLERR		●	有错误发生
POLLHUP		●	出现挂断
POLLNVAL		●	文件描述符未打开
POLLMSG			Linux 中不使用

表 13.2.1 中第一组标志 (POLLIN、POLLRDNORM、POLLRDBAND、POLLPRI、POLLRDHUP) 与数据可读相关; 第二组标志 (POLLOUT、POLLWRNORM、POLLWRBAND) 与可写数据相关; 而第三组

标志 (POLLERR、POLLHUP、POLLNVAL) 是设定在 `revents` 变量中用来返回有关文件描述符的附加信息, 如果在 `events` 变量中指定了这三个标志, 则会被忽略。

如果我们对某个文件描述符上的事件不感兴趣, 则可将 `events` 变量设置为 0; 另外, 将 `fd` 变量设置为文件描述符的负值 (取文件描述符 `fd` 的相反数 `-fd`), 将导致对应的 `events` 变量被 `poll()` 忽略, 并且 `revents` 变量将总是返回 0, 这两种方法都用来关闭对某个文件描述符的检查。

在实际应用编程中, 一般用的最多的还是 `POLLIN` 和 `POLLOUT`。对于其它标志这里不再进行介绍了, 后面章节内容中, 如果需要使用时再给大家介绍!

poll()函数返回值

`poll()`函数返回值含义与 `select()`函数的返回值是一样的, 有如下几种情况:

- 返回 -1 表示有错误发生, 并且会设置 `errno`。
- 返回 0 表示该调用在任意一个文件描述符成为就绪态之前就超时了。
- 返回一个正整数表示有一个或多个文件描述符处于就绪态了, 返回值表示 `fds` 数组中返回的 `revents` 变量不为 0 的 `struct pollfd` 对象的数量。

使用示例

示例代码 13.2.3 演示了使用 `poll()`函数来实现 I/O 多路复用操作, 同时读取键盘和鼠标。其实就是将示例代码 13.2.1 进行了修改, 使用 `poll` 替换 `select`。

示例代码 13.2.3 使用 poll 实现同时读取鼠标和键盘

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <poll.h>

#define MOUSE        "/dev/input/event3"

int main(void)
{
    char buf[100];
    int fd, ret = 0, flag;
    int loops = 5;
    struct pollfd fds[2];

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将键盘设置为非阻塞方式 */
```

```
flag = fcntl(0, F_GETFL);    //先获取原来的 flag
flag |= O_NONBLOCK;         //将 O_NONBLOCK 标准添加到 flag
fcntl(0, F_SETFL, flag);    //重新设置 flag

/* 同时读取键盘和鼠标 */
fds[0].fd = 0;
fds[0].events = POLLIN; //只关心数据可读
fds[0].revents = 0;
fds[1].fd = fd;
fds[1].events = POLLIN; //只关心数据可读
fds[1].revents = 0;

while (loops--) {
    ret = poll(fds, 2, -1);
    if (0 > ret) {
        perror("poll error");
        goto out;
    }
    else if (0 == ret) {
        fprintf(stderr, "poll timeout.\n");
        continue;
    }

    /* 检查键盘是否为就绪态 */
    if(fds[0].revents & POLLIN) {
        ret = read(0, buf, sizeof(buf));
        if (0 < ret)
            printf("键盘: 成功读取<%d>个字节数据\n", ret);
    }

    /* 检查鼠标是否为就绪态 */
    if(fds[1].revents & POLLIN) {
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret)
            printf("鼠标: 成功读取<%d>个字节数据\n", ret);
    }
}

out:
/* 关闭文件 */
close(fd);
exit(ret);
}
```

struct pollfd 结构体的 events 变量和 revents 变量都是位掩码, 所以可以使用 "revents & POLLIN" 按位与的方式来检查是否发生了相应的 POLLIN 事件, 判断鼠标或键盘数据是否可读。测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码:
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
dasdasdasd
键盘: 成功读取<11>个字节数据
hedasdas
键盘: 成功读取<9>个字节数据
鼠标: 成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.2.2 测试结果

13.2.4 总结

在使用 select() 或 poll() 时需要注意一个问题, 当监测到某一个或多个文件描述符成为就绪态 (可以读或写) 时, 需要执行相应的 I/O 操作, 以清除该状态, 否则该状态将会一直存在; 譬如示例代码 13.2.1 中, 调用 select() 函数监测鼠标和键盘这两个文件描述符, 当 select() 返回时, 通过 FD_ISSET() 宏判断文件描述符上是否可执行 I/O 操作; 如果可以执行 I/O 操作时, 应在应用程序中对该文件描述符执行 I/O 操作, 以清除文件描述符的就绪态, 如果不清除就绪态, 那么该状态将会一直存在, 那么下一次调用 select() 时, 文件描述符已经处于就绪态了, 将直接返回。

同理对于 poll() 函数来说亦是如此, 譬如示例代码 13.2.3, 当 poll() 成功返回时, 检查文件描述符是否称为就绪态, 如果文件描述符上可执行 I/O 操作时, 也需要对文件描述符执行 I/O 操作, 以清除就绪状态。

13.3 异步 IO

在 I/O 多路复用中, 进程通过系统调用 select() 或 poll() 来主动查询文件描述符上是否可以执行 I/O 操作。而在异步 I/O 中, 当文件描述符上可以执行 I/O 操作时, 进程可以请求内核为自己发送一个信号。之后进程就可以执行任何其它的任务直到文件描述符可以执行 I/O 操作为止, 此时内核会发送信号给进程。所以要使用异步 I/O, 还得结合前面所学习的信号相关的内容, 所以异步 I/O 通常也称为信号驱动 I/O。

要使用异步 I/O, 程序需要按照如下步骤来执行:

- 通过指定 O_NONBLOCK 标志使能非阻塞 I/O。
- 通过指定 O_ASYNC 标志使能异步 I/O。
- 设置异步 I/O 事件的接收进程。也就是当文件描述符上可执行 I/O 操作时会发送信号通知该进程, 通常将调用进程设置为异步 I/O 事件的接收进程。
- 为内核发送的通知信号注册一个信号处理函数。默认情况下, 异步 I/O 的通知信号是 SIGIO, 所以内核会给进程发送信号 SIGIO。在 8.2 小节中简单地提到过该信号。
- 以上步骤完成之后, 进程就可以执行其它任务了, 当 I/O 操作就绪时, 内核会向进程发送一个 SIGIO 信号, 当进程接收到信号时, 会执行预先注册好的信号处理函数, 我们就可以在信号处理函数中进行 I/O 操作。

O_ASYNC 标志

O_ASYNC 标志可用于使能文件描述符的异步 I/O 事件, 当文件描述符可执行 I/O 操作时, 内核会向异步 I/O 事件的接收进程发送 SIGIO 信号(默认情况下)。在 2.3 小节介绍 open() 函数时, 给大家提到过该标志, 但并未介绍该标志的作用, 该标志主要用于异步 I/O。

需要注意的是: 在调用 open() 时无法通过指定 O_ASYNC 标志来使能异步 I/O, 但可以使用 fcntl() 函数添加 O_ASYNC 标志使能异步 I/O, 譬如:

```
int flag;

flag = fcntl(0, F_GETFL);           //先获取原来的 flag
flag |= O_ASYNC;                   //将 O_ASYNC 标志添加到 flag
fcntl(fd, F_SETFL, flag);          //重新设置 flag
```

设置异步 I/O 事件的接收进程

为文件描述符设置异步 I/O 事件的接收进程, 也就是设置异步 I/O 的所有者。同样也是通过 fcntl() 函数进行设置, 操作命令 cmd 设置为 F_SETOWN, 第三个参数传入接收进程的进程 ID (PID), 通常将调用进程的 PID 传入, 譬如:

```
fcntl(fd, F_SETOWN, getpid());
```

注册 SIGIO 信号的处理函数

通过 signal() 或 sigaction() 函数为 SIGIO 信号注册一个信号处理函数, 当进程接收到内核发送过来的 SIGIO 信号时, 会执行该处理函数, 所以我们应该在处理函数当中执行相应的 I/O 操作。

使用示例

示例代码 13.3.1 演示了以异步 I/O 方式读取鼠标, 当进程接收到 SIGIO 信号时, 执行信号处理函数 sigio_handler(), 在该函数中调用 read() 读取鼠标数据。

示例代码 13.3.1 以异步 I/O 方式读取鼠标

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

#define MOUSE          "/dev/input/event3"
static int fd;

static void sigio_handler(int sig)
{
    static int loops = 5;
    char buf[100] = {0};
    int ret;

    if(SIGIO != sig)
        return;
```

```
ret = read(fd, buf, sizeof(buf));
if (0 < ret)
    printf("鼠标: 成功读取<%d>个字节数据\n", ret);

loops--;
if (0 >= loops) {
    close(fd);
    exit(0);
}
}

int main(void)
{
    int flag;

    /* 打开鼠标设备文件<使能非阻塞 I/O> */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 使能异步 I/O */
    flag = fcntl(fd, F_GETFL);
    flag |= O_ASYNC;
    fcntl(fd, F_SETFL, flag);

    /* 设置异步 I/O 的所有者 */
    fcntl(fd, F_SETOWN, getpid());

    /* 为 SIGIO 信号注册信号处理函数 */
    signal(SIGIO, sigio_handler);

    for (;;)
        sleep(1);
}
```

代码比较简单，这里我们进行编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.3.1 测试结果

13.4 优化异步 I/O

上一小节介绍了异步 I/O 的原理以及使用方法, 在一个需要同时检查大量文件描述符(譬如数千个)的应用程序中, 例如某种类型的网络服务端程序, 与 `select()` 和 `poll()` 相比, 异步 I/O 能够提供显著的性能优势。之所以如此, 原因在于: 对于异步 I/O, 内核可以“记住”要检查的文件描述符, 且仅当这些文件描述符上可执行 I/O 操作时, 内核才会向应用程序发送信号。

而对于 `select()` 或 `poll()` 函数来说, 内部实现原理其实是通过轮训的方式来检查多个文件描述符是否可执行 I/O 操作, 所以, 当需要检查的文件描述符数量较多时, 随之也将会消耗大量的 CPU 资源来实现轮训检查操作。当需要检查的文件描述符并不是很多时, 使用 `select()` 或 `poll()` 是一种非常不错的方案!

Tips: 当需要检查大量文件描述符时, 可以使用 `epoll` 解决 `select()` 或 `poll()` 性能低的问题, 本书并不会介绍 `epoll` 相关内容, 如果读者有兴趣可以自行查阅书籍进行学习。在性能表现上, `epoll` 与异步 I/O 方式相似, 但是 `epoll` 有一些胜过异步 I/O 的优点。

不管是异步 I/O、还是 `epoll`, 在需要检查大量文件描述符的应用程序当中, 在这种情况下, 它们的性能相比于 `select()` 或 `poll()` 有着显著的优势!

本小节将对上一小节所讲述的异步 I/O 进行优化, 既然要对其进行优化, 那必然存在着一些缺陷, 如下所示:

- 默认的异步 I/O 通知信号 `SIGIO` 是非排队信号。`SIGIO` 信号是标准信号(非实时信号、不可靠信号), 所以它不支持信号排队机制, 譬如当前正在执行 `SIGIO` 信号的处理函数, 此时内核又发送多次 `SIGIO` 信号给进程, 这些信号将会被阻塞, 只有当信号处理函数执行完毕之后才会传递给进程, 并且只能传递一次, 而其它后续的信号都会丢失。
- 无法得知文件描述符发生了什么事情。在示例代码 13.3.1 的信号处理函数 `sigio_handler()` 中, 直接调用了 `read()` 函数读取鼠标, 而并未判断文件描述符是否处于可读就绪态, 事实上, 示例代码 13.3.1 这种异步 I/O 方式并未告知应用程序文件描述符上发生了什么事情, 是可读取还是可写入亦或者发生异常等。

所以本小节我们将会针对以上列举出的两个缺陷进行优化。

13.4.1 使用实时信号替换默认信号 `SIGIO`

`SIGIO` 作为异步 I/O 通知的默认信号, 是一个非实时信号, 我们可以设置不使用默认信号, 指定一个实时信号作为异步 I/O 通知信号, 如何指定呢? 同样也是使用 `fcntl()` 函数进行设置, 调用函数时将操作命令 `cmd` 参数设置为 `F_SETSIG`, 第三个参数 `arg` 指定一个实时信号编号即可, 表示将该信号作为异步 I/O 通知信号, 譬如:

```
fcntl(fd, F_SETSIG, SIGRTMIN);
```


上述代码指定了 SIGRTMIN 实时信号作为文件描述符 fd 的异步 I/O 通知信号, 而不再使用默认的 SIGIO 信号。当文件描述符 fd 可执行 I/O 操作时, 内核会发送实时信号 SIGRTMIN 给调用进程。

如果第三个参数 arg 设置为 0, 则表示指定 SIGIO 信号作为异步 I/O 通知信号, 也就是回到了默认状态。

13.4.2 使用 sigaction() 函数注册信号处理函数

在应用程序当中需要为实时信号注册信号处理函数, 使用 sigaction 函数进行注册, 并为 sa_flags 参数指定 SA_SIGINFO, 表示使用 sa_sigaction 指向的函数作为信号处理函数, 而不使用 sa_handler 指向的函数。因为 sa_sigaction 指向的函数作为信号处理函数提供了更多的参数, 可以获取到更多信息, 函数定义参考示例代码 8.4.2 中关于 struct sigaction 结构体的描述。

函数参数中包括一个 siginfo_t 指针, 指向 siginfo_t 类型对象, 当触发信号时该对象由内核构建。siginfo_t 结构体中提供了很多信息, 我们可以在信号处理函数中使用这些信息, 具体定义请参考示例代码 8.4.3, 就对于异步 I/O 事件而言, 传递给信号处理函数的 siginfo_t 结构体中与之相关的字段如下:

- si_signo: 引发处理函数被调用的信号。这个值与信号处理函数的第一个参数一致。
- si_fd: 表示发生异步 I/O 事件的文件描述符;
- si_code: 表示文件描述符 si_fd 发生了什么事情, 读就绪态、写就绪态或者是异常事件等。该字段中可能出现的值以及它们对应的描述信息参见表 13.4.1。
- si_band: 是一个位掩码, 其中包含的值与系统调用 poll() 中返回的 revents 字段中的值相同。如表 13.4.1 所示, si_code 中可能出现的值与 si_band 中的位掩码有着一一对应关系。

表 13.4.1 siginfo_t 结构体中的 si_code 和 si_band 的可能值

si_code	si_band 掩码值	描述/说明
POLL_IN	POLLIN POLLRDNORM	可读取数据
POLL_OUT	POLLOUT POLLWRNORM POLLWRBAND	可写入数据
POLL_MSG	POLLIN POLLRDNORM POLLMSG	不使用
POLL_ERR	POLLERR	I/O 错误
POLL_PRI	POLLPRI POLLRDNORM	可读取高优先级数据
POLL_HUP	POLLHUP POLLERR	出现宕机

所以, 由此可知, 可以在信号处理函数中通过对比 siginfo_t 结构体的 si_code 变量来检查文件描述符发生了什么事情, 以采取相应的 I/O 操作。

13.4.3 使用示例

通过 13.4.1 小节和 13.4.2 小节的学习, 我们已经知道了如何针对 13.4 小节开头提出的异步 I/O 存在的两个缺陷进行优化。示例代码 13.4.1 是对示例代码 13.3.1 进行了优化, 使用实时信号+sigaction 解决: 默认异步 I/O 通知信号 SIGIO 可能存在丢失以及信号处理函数中无法判断文件描述符所发生的 I/O 事件这两个问题。

调用 sigaction() 注册信号处理函数时, sa_flags 指定了 SA_SIGINFO, 所以将使用 sa_sigaction 指向的函数 io_handler 作为信号处理函数, io_handler 共有 3 个参数, 参数 sig 等于引发信号处理函数被调用的信号值, 参数 info 附加了很多信息, 前面已有介绍, 这里不再重述。

示例代码 13.4.1 读取鼠标--优化异步 I/O

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <signal.h>

#define MOUSE          "/dev/input/event3"
static int fd;

static void io_handler(int sig,
                      siginfo_t *info,
                      void *context)
{
    static int loops = 5;
    char buf[100] = {0};
    int ret;

    if(SIGRTMIN != sig)
        return;

    /* 判断鼠标是否可读 */
    if (POLL_IN == info->si_code) {
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret)
            printf("鼠标: 成功读取<%d>个字节数据\n", ret);

        loops--;
        if (0 >= loops) {
            close(fd);
            exit(0);
        }
    }
}

int main(void)
{
    struct sigaction act;
    int flag;

    /* 打开鼠标设备文件<使能非阻塞 I/O> */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }
}
```

```

/* 使能异步 I/O */
flag = fcntl(fd, F_GETFL);
flag |= O_ASYNC;
fcntl(fd, F_SETFL, flag);

/* 设置异步 I/O 的所有者 */
fcntl(fd, F_SETOWN, getpid());

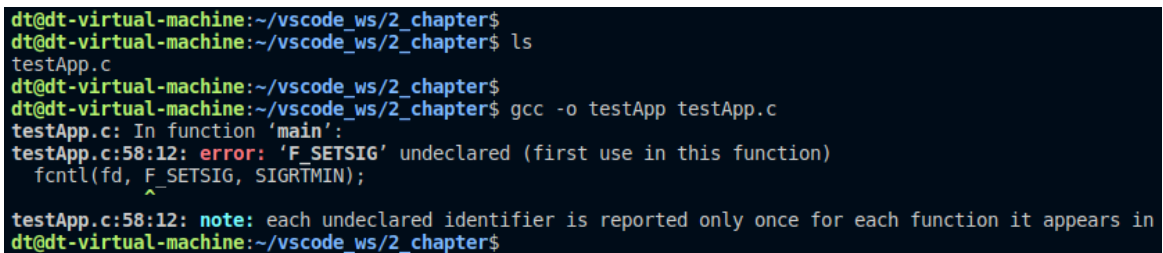
/* 指定实时信号 SIGRTMIN 作为异步 I/O 通知信号 */
fcntl(fd, F_SETSIG, SIGRTMIN);

/* 为实时信号 SIGRTMIN 注册信号处理函数 */
act.sa_sigaction = io_handler;
act.sa_flags = SA_SIGINFO;
sigemptyset(&act.sa_mask);
sigaction(SIGRTMIN, &act, NULL);

for (;;)
    sleep(1);
}

```

对上述示例代码进行编译时, 出现了一些报错信息, 如下所示:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
testApp.c: In function 'main':
testApp.c:58:12: error: 'F_SETSIG' undeclared (first use in this function)
    fcntl(fd, F_SETSIG, SIGRTMIN);
               ^
testApp.c:58:12: note: each undeclared identifier is reported only once for each function it appears in
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.4.1 编译报错

报错提示没有定义 F_SETSIG, 确实如此, 我们需要定义了 _GNU_SOURCE 宏之后才能使用 F_SETSIG, 这个宏在 4.9.3 小节向大家介绍过, 这里不再重述!

这里笔者选择直接在源文件中使用 #define 定义 _GNU_SOURCE 宏, 如下所示:

```

#define _GNU_SOURCE //在源文件开头定义_GNU_SOURCE 宏

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>

```

再次进行编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -D_GNU_SOURCE -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码:
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
鼠标: 成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.4.2 测试结果

13.5 存储映射 I/O

存储映射 I/O (memory-mapped I/O) 是一种基于内存区域的高级 I/O 操作, 它能将一个文件映射到进程地址空间中的一块内存区域中, 当从这段内存中读数据时, 就相当于读文件中的数据 (对文件进行 read 操作), 将数据写入这段内存时, 则相当于将数据直接写入文件中 (对文件进行 write 操作)。这样就可以在不使用基本 I/O 操作函数 read() 和 write() 的情况下执行 I/O 操作。

13.5.1 mmap() 和 munmap() 函数

为了实现存储映射 I/O 这一功能, 我们需要告诉内核将一个给定的文件映射到进程地址空间中的一块内存区域中, 这由系统调用 mmap() 来实现。其函数原型如下所示:

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

使用该函数需要包含头文件 <sys/mman.h>。

函数参数和返回值含义如下:

addr: 参数 addr 用于指定映射到内存区域的起始地址。通常将其设置为 NULL, 这表示由系统选择该映射区的起始地址, 这是最常见的设置方式; 如果参数 addr 不为 NULL, 则表示由自己指定映射区的起始地址, 此函数的返回值是该映射区的起始地址。

length: 参数 length 指定映射长度, 表示将文件中的多大部分映射到内存区域中, 以字节为单位, 譬如 length=1024 * 4, 表示将文件的 4K 字节大小映射到内存区域中。

offset: 文件映射的偏移量, 通常将其设置为 0, 表示从文件头部开始映射; 所以参数 offset 和参数 length 就确定了文件的起始位置和长度, 将文件的这部分映射到内存区域中, 如图 13.5.1 所示。

fd: 文件描述符, 指定要映射到内存区域中的文件。

prot: 参数 prot 指定了映射区的保护要求, 可取值如下:

- **PROT_EXEC:** 映射区可执行;
- **PROT_READ:** 映射区可读;
- **PROT_WRITE:** 映射区可写;
- **PROT_NONE:** 映射区不可访问。

可将 prot 指定为 PROT_NONE, 也可将其设置为 PROT_EXEC、PROT_READ、PROT_WRITE 中一个或多个 (通过按位或运算符任意组合)。对指定映射区的保护要求不能超过文件 open() 时的访问权限, 譬如, 文件是以只读权限方式打开的, 那么对映射区的不能指定为 PROT_WRITE。

flags: 参数 flags 可影响映射区的多种属性, 参数 flags 必须要指定以下两种标志之一:

- **MAP_SHARED:** 此标志指定当对映射区写入数据时, 数据会写入到文件中, 也就是会将写入到映射区中的数据更新到文件中, 并且允许其它进程共享。
- **MAP_PRIVATE:** 此标志指定当对映射区写入数据时, 会创建映射文件的一个私人副本 (copy-on-write), 对映射区的任何操作都不会更新到文件中, 仅仅只是对文件副本进行读写。

除此之外, 还可将以下标志中的 0 个或多个组合到参数 flags 中, 通过按位或运算符进行组合:

- **MAP_FIXED:** 在未指定该标志的情况下, 如果参数 addr 不等于 NULL, 表示由调用者自己指定映射区的起始地址, 但这只是一种建议、而非强制, 所以内核并不会保证使用参数 addr 指定的值作为映射区的起始地址; 如果指定了 MAP_FIXED 标志, 则表示要求必须使用参数 addr 指定的值作为起始地址, 如果使用指定值无法成功建立映射时, 则放弃! 通常, 不建议使用此标志, 因为这不利于移植。
- **MAP_ANONYMOUS:** 建立匿名映射, 此时会忽略参数 fd 和 offset, 不涉及文件, 而且映射区域无法和其它进程共享。
- **MAP_ANON:** 与 MAP_ANONYMOUS 标志同义, 不建议使用。
- **MAP_DENYWRITE:** 该标志被忽略。
- **MAP_EXECUTABLE:** 该标志被忽略。
- **MAP_FILE:** 兼容性标志, 已被忽略。
- **MAP_LOCKED:** 对映射区域进行上锁。

除了以上标志之外, 还有其它一些标志, 这里便不再介绍, 可通过 man 手册进行查看。在众多标志当中, 通常情况下, 参数 flags 中只指定了 MAP_SHARED。

返回值: 成功情况下, 函数的返回值便是映射区的起始地址; 发生错误时, 返回(void *)-1, 通常使用 MAP_FAILED 来表示, 并且会设置 errno 来指示错误原因。

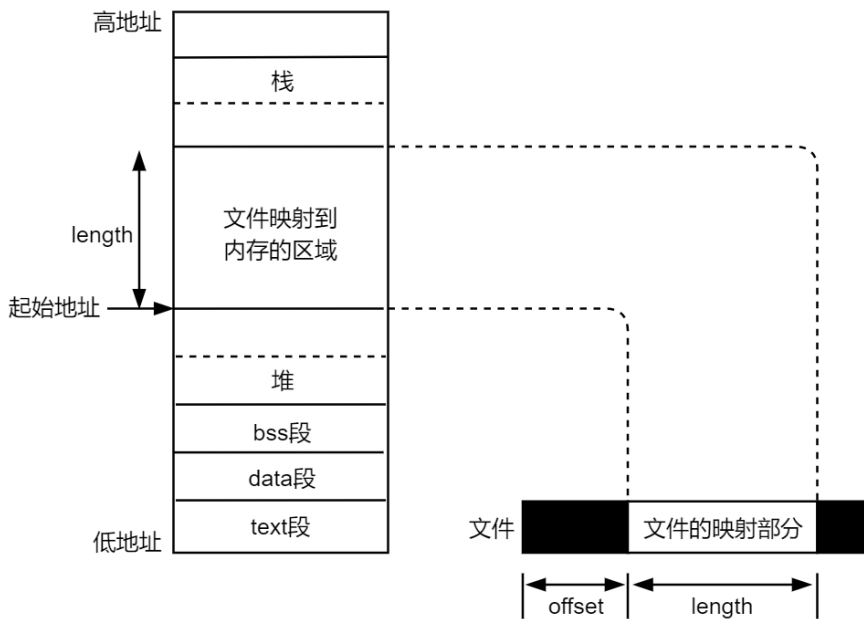


图 13.5.1 存储映射 I/O 示意图

对于 mmap()函数, 参数 addr 和 offset 在不为 NULL 和 0 的情况下, addr 和 offset 的值通常被要求是系统页大小的整数倍, 可通过 sysconf()函数获取页大小, 如下所示 (以字节为单位):

```
sysconf(_SC_PAGE_SIZE)
或
sysconf(_SC_PAGESIZE)
```

虽然对 `addr` 和 `offset` 有这种限制, 但对于参数 `length` 长度来说, 却没有这种要求, 如果映射区的长度不是页长度的整数倍时, 会怎么样呢? 对于这个问题的答案, 我们首先需要了解到, 对于 `mmap()` 函数来说, 当文件成功被映射到内存区域时, 这段内存区域 (映射区) 的大小通常是页大小的整数倍, 即使参数 `length` 并不是页大小的整数倍。如果文件大小为 96 个字节, 我们调用 `mmap()` 时参数 `length` 也是设置为 96, 假设系统页大小为 4096 字节 (4K), 则系统通常会提供 4096 个字节的映射区, 其中后 4000 个字节会被设置为 0, 可以修改后面的这 4000 个字节, 但是并不会影响到文件。但如果访问 4000 个字节后面的内存区域, 将会导致异常情况发生, 产生 `SIGBUS` 信号。

对于参数 `length` 任需要注意, 参数 `length` 的值不能大于文件大小, 即文件被映射的部分不能超出文件。

与映射区相关的两个信号

- **SIGSEGV:** 如果映射区被 `mmap()` 指定成了只读的, 那么进程试图将数据写入到该映射区时, 将会产生 `SIGSEGV` 信号, 此信号由内核发送给进程。在第八章中给大家介绍过该信号, 该信号的系统默认操作是终止进程、并生成核心可用于调试的核心转储文件。
- **SIGBUS:** 如果映射区的某个部分在访问时已不存在, 则会产生 `SIGBUS` 信号。例如, 调用 `mmap()` 进行映射时, 将参数 `length` 设置为文件长度, 但在访问映射区之前, 另一个进程已将该文件截断 (譬如调用 `ftruncate()` 函数进行截断), 此时如果进程试图访问对应于该文件已截去部分的映射区, 进程将会受到内核发送过来的 `SIGBUS` 信号, 同样, 该信号的系统默认操作是终止进程、并生成核心可用于调试的核心转储文件。

`munmap()` 解除映射

通过 `open()` 打开文件, 需要使用 `close()` 将其关闭; 同理, 通过 `mmap()` 将文件映射到进程地址空间中的一块内存区域中, 当不再需要时, 必须解除映射, 使用 `munmap()` 解除映射关系, 其函数原型如下所示:

```
#include <sys/mman.h>
```

```
int munmap(void *addr, size_t length);
```

同样, 使用该函数需要包含头文件 `<sys/mman.h>`。

`munmap()` 系统调用解除指定地址范围内的映射, 参数 `addr` 指定待解除映射地址范围的起始地址, 它必须是系统页大小的整数倍; 参数 `length` 是一个非负整数, 指定了待解除映射区域的大小 (字节数), 被解除映射的区域对应的大小也必须是系统页大小的整数倍, 即使参数 `length` 并不等于系统页大小的整数倍, 与 `mmap()` 函数相似。

需要注意的是, 当进程终止时也会自动解除映射 (如果程序中没有显式调用 `munmap()`), 但调用 `close()` 关闭文件时并不会解除映射。

通常将参数 `addr` 设置为 `mmap()` 函数的返回值, 将参数 `length` 设置为 `mmap()` 函数的参数 `length`, 表示解除整个由 `mmap()` 函数所创建的映射。

使用示例

通过以上介绍, 接下来我们编写一个简单地示例代码, 使用存储映射 I/O 进行文件复制。

示例代码 13.5.1 演示了使用存储映射 I/O 实现文件复制操作, 将源文件中的内容全部复制到另一个目标文件中, 其效果类似于 `cp` 命令。

示例代码 13.5.1 使用存储映射 I/O 复制文件

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int srcfd, dstfd;
    void *srcaddr;
    void *dstaddr;
    int ret;
    struct stat sbuf;

    if (3 != argc) {
        fprintf(stderr, "usage: %s <srcfile> <dstfile>\n", argv[0]);
        exit(-1);
    }

    /* 打开源文件 */
    srcfd = open(argv[1], O_RDONLY);
    if (-1 == srcfd) {
        perror("open error");
        exit(-1);
    }

    /* 打开目标文件 */
    dstfd = open(argv[2], O_RDWR |
                 O_CREAT | O_TRUNC, 0664);
    if (-1 == dstfd) {
        perror("open error");
        ret = -1;
        goto out1;
    }

    /* 获取源文件的大小 */
    fstat(srcfd, &sbuf);

    /* 设置目标文件的大小 */
    ftruncate(dstfd, sbuf.st_size);

    /* 将源文件映射到内存区域中 */
    srcaddr = mmap(NULL, sbuf.st_size,
                  PROT_READ, MAP_SHARED, srcfd, 0);
    if (MAP_FAILED == srcaddr) {
```

```
    perror("mmap error");
    ret = -1;
    goto out2;
}

/* 将目标文件映射到内存区域中 */
dstaddr = mmap(NULL, sbuf.st_size,
               PROT_WRITE, MAP_SHARED, dstfd, 0);
if (MAP_FAILED == dstaddr) {
    perror("mmap error");
    ret = -1;
    goto out3;
}

/* 将源文件中的内容复制到目标文件中 */
memcpy(dstaddr, srcaddr, sbuf.st_size);

/* 程序退出前清理工作 */
out4:
/* 解除目标文件映射 */
munmap(dstaddr, sbuf.st_size);
out3:
/* 解除源文件映射 */
munmap(srcaddr, sbuf.st_size);
out2:
/* 关闭目标文件 */
close(dstfd);
out1:
/* 关闭源文件并退出 */
close(srcfd);
exit(ret);
}
```

当执行程序的时候, 将源文件和目标文件传递给应用程序, 该程序首先会将源文件和目标文件打开, 源文件以只读方式打开, 而目标文件以可读、可写方式打开, 如果目标文件不存在则创建它, 并且将文件的大小截断为 0。

然后使用 `fstat()` 函数获取源文件的大小, 接着调用 `ftruncate()` 函数设置目标文件的大小与源文件大小保持一致。

然后对源文件和目标文件分别调用 `mmap()`, 将文件映射到内存当中; 对于源文件, 调用 `mmap()` 时将参数 `prot` 指定为 `PROT_READ`, 表示对它的映射区会进行读取操作; 对于目标文件, 调用 `mmap()` 时将参数 `prot` 指定为 `PROT_WRITE`, 表示对它的映射区会进行写入操作。最后调用 `memcpy()` 将源文件映射区中的内容复制到目标文件映射区中, 完成文件的复制操作。

接下来我们进行测试, 笔者使用当前目录下的 `srcfile` 作为源文件, `dstfile` 作为目标文件, 先看看源文件 `srcfile` 的内容, 如下所示:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
srcfile testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat srcfile
Linux高级I/O之存储映射I/O测试
使用存储映射I/O实现文件复制
将源文件中的内容复制到目标文件中
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.5.2 源文件中的内容

目标文件 dstfile 并不存在，我们需要在程序中进行创建，编译程序、运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
srcfile testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./srcfile ./dstfile
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
dstfile srcfile testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat dstfile
Linux高级I/O之存储映射I/O测试
使用存储映射I/O实现文件复制
将源文件中的内容复制到目标文件中
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.5.3 测试结果

由打印信息可知，程序运行完之后，生成了目标文件 dstfile，使用 cat 命令查看其内容与源文件 srcfile 相同，本测试程序成功实现了文件复制功能！

13.5.2 mprotect()函数

使用系统调用 mprotect()可以更改一个现有映射区的保护要求，其函数原型如下所示：

```
#include <sys/mman.h>
```

```
int mprotect(void *addr, size_t len, int prot);
```

使用该函数，同样需要包含头文件<sys/mman.h>。

参数 prot 的取值与 mmap()函数的 prot 参数的一样，mprotect()函数会将指定地址范围的保护要求更改为参数 prot 所指定的类型，参数 addr 指定该地址范围的起始地址，addr 的值必须是系统页大小的整数倍；参数 len 指定该地址范围的大小。

mprotect()函数调用成功返回 0；失败将返回-1，并且会设置 errno 来只是错误原因。

13.5.3 msync()函数

在第四章中提到过，read()和 write()系统调用在操作磁盘文件时不会直接发起磁盘访问（读写磁盘硬件），而是仅仅在用户空间缓冲区和内核缓冲区之间复制数据，在后续的某个时刻，内核会将其缓冲区中的数据写入（刷新至）磁盘中，所以由此可知，调用 write()写入到磁盘文件中的数据并不会立马写入磁盘，而是会先缓存在内核缓冲区中，所以就会出现 write()操作与磁盘操作并不同步，也就是数据不同步。

对于存储 I/O 来说亦是如此，写入到文件映射区中的数据也不会立马刷新至磁盘设备中，而是会在我们将数据写入到映射区之后的某个时刻将映射区中的数据写入磁盘中。所以会导致映射区中的内容与磁盘文件中的内容不同步。我们可以调用 msync()函数将映射区中的数据刷写、更新至磁盘文件中（同步操作），系统调用 msync()类似于 fsync()函数，不过 msync()作用于映射区。该函数原型如下所示：

```
#include <sys/mman.h>
```

```
int msync(void *addr, size_t length, int flags);
```

使用该函数，同样需要包含头文件<sys/mman.h>。

参数 `addr` 和 `length` 指定了需同步的内存区域的起始地址和大小。对于参数 `addr` 来说，同样也要求必须是系统页大小的整数倍，也就是与系统页大小对齐。譬如，调用 `msync()` 时，将 `addr` 设置为 `mmap()` 函数的返回值，将 `length` 设置为 `mmap()` 函数的 `length` 参数，将对文件的整个映射区进行同步操作。

参数 `flags` 应指定为 `MS_ASYNC` 和 `MS_SYNC` 两个标志之一，除此之外，还可以根据需求选择是否指定 `MS_INVALIDATE` 标志，作为一个可选标志。

- **MS_ASYNC**: 以异步方式进行同步操作。调用 `msync()` 函数之后，并不会等待数据完全写入磁盘之后才返回。
- **MS_SYNC**: 以同步方式进行同步操作。调用 `msync()` 函数之后，需等待数据全部写入磁盘之后才返回。
- **MS_INVALIDATE**: 是一个可选标志，请求使同一文件的其它映射无效（以便可以用刚写入的新值更新它们）。

`msync()` 函数在调用成功情况下返回 0；失败将返回 -1、并设置 `errno`。

`munmap()` 函数并不影响被映射的文件，也就是说，当调用 `munmap()` 解除映射时并不会将映射区中的内容写到磁盘文件中。如果 `mmap()` 指定了 `MAP_SHARED` 标志，对于文件的更新，会在我们将数据写入到映射区之后的某个时刻将映射区中的数据更新到磁盘文件中，由内核根据虚拟存储算法自动进行。

如果 `mmap()` 指定了 `MAP_PRIVATE` 标志，在解除映射之后，进程对映射区的修改将会丢弃！

13.5.4 普通 I/O 与存储映射 I/O 比较

通过前面的介绍，相信大家对存储映射 I/O 之间有了一个新的认识，本小节我们再来对普通 I/O 方式和存储映射 I/O 做一个简单的总结。

普通 I/O 方式的缺点

普通 I/O 方式一般是通过调用 `read()` 和 `write()` 函数来实现对文件的读写，使用 `read()` 和 `write()` 读写文件时，函数经过层层调用后，才能够最终操作到文件，中间涉及到很多的函数调用过程，数据需要在不同的缓存间倒腾，效率会比较低。同样使用标准 I/O（库函数 `fread()`、`fwrite()`）也是如此，本身标准 I/O 就是对普通 I/O 的一种封装。

那既然效率较低，为啥还要使用这种方式呢？原因在于，只有当数据量比较大时，效率的影响才会比较明显，如果数据量比较小，影响并不大，使用普通的 I/O 方式还是非常方便的。

存储映射 I/O 的优点

存储映射 I/O 的实质其实是共享，与 IPC 之内存共享很相似。譬如执行一个文件复制操作来说，对于普通 I/O 方式，首先需要将源文件中的数据读取出来存放在一个应用层缓冲区中，接着再将缓冲区中的数据写入到目标文件中，如下所示：

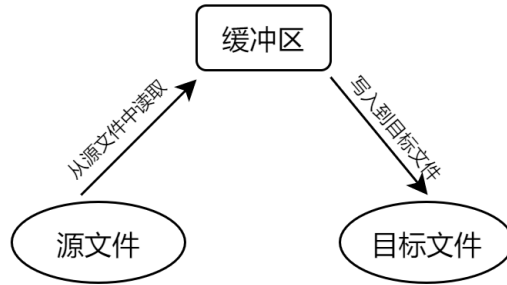


图 13.5.4 普通 I/O 实现文件复制示例图

而对于存储映射 I/O 来说, 由于源文件和目标文件都已映射到了应用层的内存区域中, 所以直接操作映射区来实现文件复制, 如下所示:

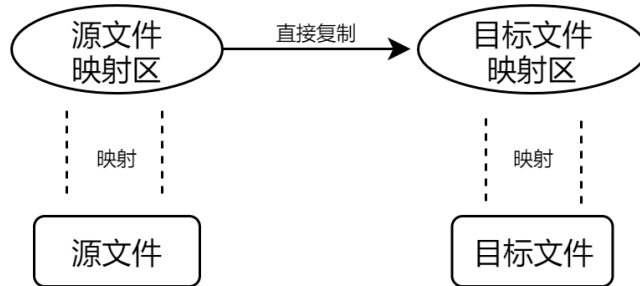


图 13.5.5 存储映射 I/O 实现文件复制

首先非常直观的一点就是, 使用存储映射 I/O 减少了数据的复制操作, 所以在效率上会比普通 I/O 要高, 其次上面也讲了, 普通 I/O 中间涉及到了很多的函数调用过程, 这些都会导致普通 I/O 在效率上会比存储映射 I/O 要低。

前面提到存储映射 I/O 的实质其实是共享, 如何理解共享呢? 其实非常简单, 我们知道, 应用层与内核层是不能直接进行交互的, 必须要通过操作系统提供的系统调用或库函数来与内核进行数据交互, 包括操作硬件。通过存储映射 I/O 将文件直接映射到应用程序地址空间中的一块内存区域中, 也就是映射区; 直接将磁盘文件直接与映射区关联起来, 不用调用 read()、write() 系统调用, 直接对映射区进行读写操作即可操作磁盘上的文件, 而磁盘文件中的数据也可反应到映射区中, 这就是一种共享, 可以认为映射区就是应用层与内核层之间的共享内存。

存储映射 I/O 的不足

存储映射 I/O 方式并不是完美的, 它所映射的文件只能是固定大小, 因为文件所映射的区域已经在调用 mmap() 函数时通过 length 参数指定了。另外, 文件映射的内存区域的大小必须是系统页大小的整数倍, 譬如映射文件的大小为 96 字节, 假定系统页大小为 4096 字节, 那么剩余的 4000 字节全部填充为 0, 虽然可以通过映射地址访问剩余的这些字节数据, 但不能在映射文件中反应出来, 由此可知, 使用存储映射 I/O 在进行大数据量操作时比较有效; 对于少量数据, 使用普通 I/O 方式更加方便!

存储映射 I/O 的应用场景

由上面介绍可知, 存储映射 I/O 在处理大量数据时效率高, 对于少量数据处理不是很划算, 所以通常来说, 存储映射 I/O 会在视频图像处理方面用的比较多, 譬如在第二篇内容, 我们将会介绍 Framebuffer 编程, 通俗点说就是 LCD 编程, 就会使用到存储映射 I/O。

13.6 文件锁

现象一下,当两个人同时编辑磁盘中同一份文件时,其后果将会如何呢?在 Linux 系统中,该文件的最后状态通常取决于写该文件的最后一个进程。多个进程同时操作同一文件,很容易导致文件中的数据发生混乱,因为多个进程对文件进行 I/O 操作时,容易产生竞争状态、导致文件中的内容与预想的不一致!

对于有些应用程序,进程有时需要确保只有它自己能够对某一文件进行 I/O 操作,在这段时间内不允许其它进程对该文件进行 I/O 操作。为了向进程提供这种功能, Linux 系统提供了文件锁机制。

前面学习过互斥锁、自旋锁以及读写锁,文件锁与这些锁一样,都是内核提供的锁机制,锁机制实现用于对共享资源的访问进行保护;只不过互斥锁、自旋锁、读写锁与文件锁的应用场景不一样,互斥锁、自旋锁、读写锁主要用在多线程环境下,对共享资源的访问进行保护,做到线程同步。

而文件锁,顾名思义是一种应用于文件的锁机制,当多个进程同时操作同一文件时,我们怎么保证文件数据的正确性,linux 通常采用的方法是对文件上锁,来避免多个进程同时操作同一文件时产生竞争状态。譬如进程对文件进行 I/O 操作时,首先对文件进行上锁,将其锁住,然后再进行读写操作;只要进程没有对文件进行解锁,那么其它的进程将无法对其进行操作;这样就可以保证,文件被锁住期间,只有它(该进程)可以对其进行读写操作。

一个文件既然可以被多个进程同时操作,那说明文件必然是一种共享资源,所以由此可知,归根结底,文件锁也是一种用于对共享资源的访问进行保护的机制,通过对文件上锁,来避免访问共享资源产生竞争状态。

文件锁的分类

文件锁可以分为建议性锁和强制性锁两种:

● 建议性锁

建议性锁本质上是一种协议,程序访问文件之前,先对文件上锁,上锁成功之后再访问文件,这是建议性锁的一种用法;但是如果你的程序不管三七二十一,在没有对文件上锁的情况下直接访问文件,也是可以访问的,并非无法访问文件;如果是这样,那么建议性锁就没有起到任何作用,如果要使得建议性锁起作用,那么大家就要遵守协议,访问文件之前先对文件上锁。这就好比交通信号灯,规定红灯不能通行,绿灯才可以通行,但如果你非要在红灯的时候通行,谁也拦不住你,那么后果将会导致发生交通事故;所以必须要大家共同遵守交通规则,交通信号灯才能起到作用。

● 强制性锁:

强制性锁比较好理解,它是一种强制性的要求,如果进程对文件上了强制性锁,其它的进程在没有获取到文件锁的情况下是无法对文件进行访问的。其本质原因在于,强制性锁会让内核检查每一个 I/O 操作(譬如 read()、write()),验证调用进程是否是该文件锁的拥有者,如果不是将无法访问文件。当一个文件被上锁进行写入操作的时候,内核将阻止其它进程对其进行读写操作。采取强制性锁对性能的影响很大,每次进行读写操作都必须检查文件锁。

在 Linux 系统中,可以调用 flock()、fcntl()以及 lockf()这三个函数对文件上锁,接下来将向大家介绍每个函数的使用方法。

13.6.1 flock()函数加锁

先来学习系统调用 flock(),使用该函数可以对文件加锁或者解锁,但是 flock()函数只能产生建议性锁,其函数原型如下所示:

```
#include <sys/file.h>
```

```
int flock(int fd, int operation);
```

使用该函数需要包含头文件<sys/file.h>。

函数参数和返回值含义如下:

fd: 参数 fd 为文件描述符, 指定需要加锁的文件。

operation: 参数 operation 指定了操作方式, 可以设置为以下值的其中一个:

- **LOCK_SH:** 在 fd 引用的文件上放置一把共享锁。所谓共享, 指的便是多个进程可以拥有对同一个文件的共享锁, 该共享锁可被多个进程同时拥有。
- **LOCK_EX:** 在 fd 引用的文件上放置一把排它锁 (或叫互斥锁)。所谓互斥, 指的便是互斥锁只能同时被一个进程所拥有。
- **LOCK_UN:** 解除文件锁定状态, 解锁、释放锁。

除了以上三个标志外, 还有一个标志:

- **LOCK_NB:** 表示以非阻塞方式获取锁。默认情况下, 调用 flock() 无法获取到文件锁时会阻塞、直到其它进程释放锁为止, 如果不想让程序被阻塞, 可以指定 LOCK_NB 标志, 如果无法获取到锁应立即返回 (错误返回, 并将 errno 设置为 EWOULDBLOCK), 通常与 LOCK_SH 或 LOCK_EX 一起使用, 通过位或运算符组合在一起。

返回值: 成功将返回 0; 失败返回 -1、并会设置 errno,

对于 flock(), 需要注意的是, 同一个文件不会同时具有共享锁和互斥锁。

使用示例

示例代码 13.6.1 演示了使用 flock() 函数对一个文件加锁和解锁 (建议性锁)。程序首先调用 open() 函数将文件打开, 文件路径通过传参的方式传递进来; 文件打开成功之后, 调用 flock() 函数对文件加锁 (非阻塞方式、排它锁), 并打印出 “文件加锁成功” 信息, 如果加锁失败便会打印出 “文件加锁失败” 信息。然后调用 signal 函数为 SIGINT 信号注册了一个信号处理函数, 当进程接收到 SIGINT 信号后会执行 sigint_handler() 函数, 在信号处理函数中对文件进行解锁, 然后终止进程。

示例代码 13.6.1 使用 flock() 对文件加锁/解锁

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <signal.h>

static int fd = -1; //文件描述符

/* 信号处理函数 */
static void sigint_handler(int sig)
{
    if (SIGINT != sig)
        return;

    /* 解锁 */
    flock(fd, LOCK_UN);
    close(fd);
    printf("进程 1: 文件已解锁!\n");
}
```

```
}

int main(int argc, char *argv[])
{
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_WRONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 以非阻塞方式对文件加锁(排它锁) */
    if (-1 == flock(fd, LOCK_EX | LOCK_NB)) {
        perror("进程 1: 文件加锁失败");
        exit(-1);
    }

    printf("进程 1: 文件加锁成功!\n");

    /* 为 SIGINT 信号注册处理函数 */
    signal(SIGINT, sigint_handler);

    for (;;)
        sleep(1);
}
```

加锁成功之后, 程序进入了 for 死循环, 一直持有锁; 此时我们可以执行另一个程序, 如示例代码 13.6.2 所示, 该程序首先也会打开文件, 文件路径通过传参的方式传递进来, 同样在程序中也会调用 flock() 函数对文件加锁 (排它锁、非阻塞方式), 不管加锁成功与否都会执行下面的 I/O 操作, 将数据写入文件、在读取出来并打印。

示例代码 13.6.2 未获取锁情况下读写文件

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <string.h>
```

```
int main(int argc, char *argv[])
{
    char buf[100] = "Hello World!";
    int fd;
    int len;

    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 以非阻塞方式对文件加锁(排它锁) */
    if (-1 == flock(fd, LOCK_EX | LOCK_NB))
        perror("进程 2: 文件加锁失败");
    else
        printf("进程 2: 文件加锁成功!\n");

    /* 写文件 */
    len = strlen(buf);
    if (0 > write(fd, buf, len)) {
        perror("write error");
        exit(-1);
    }
    printf("进程 2: 写入到文件的字符串<%s>\n", buf);

    /* 将文件读写位置移动到文件头 */
    if (0 > lseek(fd, 0x0, SEEK_SET)) {
        perror("lseek error");
        exit(-1);
    }

    /* 读文件 */
    memset(buf, 0x0, sizeof(buf)); //清理 buf
    if (0 > read(fd, buf, len)) {
        perror("read error");
    }
}
```

```

    exit(-1);
}
printf("进程 2: 从文件读取的字符串<%s>\n", buf);

/* 解锁、退出 */
flock(fd, LOCK_UN);
close(fd);
exit(0);
}

```

把示例代码 13.6.1 作为应用程序 1, 把示例代码 13.6.2 作为应用程序 2, 将它们分别编译成不同的可执行文件 testApp1 和 testApp2, 如下所示:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp1 testApp2 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.6.1 两份可执行文件

在进行测试之前, 创建一个测试用的文件 infile, 直接使用 touch 命令创建即可, 首先执行 testApp1 应用程序, 将 infile 文件作为输入文件, 并将其放置在后台运行:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
infile testApp1 testApp2 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp1 ./infile &
[1] 20710
进程1: 文件加锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
  PID TTY          TIME CMD
 6535 pts/21    00:00:02 bash
 20710 pts/21    00:00:00 testApp1
 20713 pts/21    00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.6.2 执行 testApp1

testApp1 会在后台运行, 由 ps 命令可查看到其 pid 为 20710。接着执行 testApp2 应用程序, 传入相同的文件 infile, 如下所示:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp2 ./infile
进程2: 文件加锁失败: Resource temporarily unavailable
进程2: 写入到文件的字符串<Hello World!>
进程2: 从文件读取的字符串<Hello World!>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.6.3 执行 testApp2

从打印信息可知, testApp2 进程对 infile 文件加锁失败, 原因在于锁已经被 testApp1 进程所持有, 所以 testApp2 加锁自然会失败; 但是可以发现虽然加锁失败, 但是 testApp2 对文件的读写操作是没有问题的, 是成功的, 这就是建议性锁的特点; 正确的使用方式是, 在加锁失败之后不要再对文件进行 I/O 操作了, 遵循这个协议。

接着我们向 testApp1 进程发送一个 SIGIO 信号, 让其对文件 infile 解锁, 接着再执行一次 testApp2, 如下所示:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 20710
进程1: 文件已解锁!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp2 infile
进程2: 文件加锁成功!
进程2: 写入到文件的字符串<Hello World!>
进程2: 从文件读取的字符串<Hello World!>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.4 测试结果

使用 kill 命令向 testApp1 进程发送编号为 2 的信号, 也就是 SIGIO 信号, testApp1 接收到信号之后, 对 infile 文件进行解锁、然后退出; 接着再次执行 testApp2 程序, 从打印信息可知, 这次能够成功对 infile 文件加锁了, 读写也是没有问题的。

关于 flock()的几条规则

- 同一进程对文件多次加锁不会导致死锁。当进程调用 flock()对文件加锁成功, 再次调用 flock()对文件(同一文件描述符)加锁, 这样不会导致死锁, 新加的锁会替换旧的锁。譬如调用 flock()对文件加共享锁, 再次调用 flock()对文件加排它锁, 最终文件锁会由共享锁替换为排它锁。
- 文件关闭的时候, 会自动解锁。进程调用 flock()对文件加锁, 如果在未解锁之前将文件关闭, 则会导致文件锁自动解锁, 也就是说, 文件锁会在相应的文件描述符被关闭之后自动释放。同理, 当一个进程终止时, 它所建立的锁将全部释放。
- 一个进程不可以对另一个进程持有的文件锁进行解锁。
- 由 fork()创建的子进程不会继承父进程所创建的锁。这意味着, 若一个进程对文件加锁成功, 然后该进程调用 fork()创建了子进程, 那么对父进程创建的锁而言, 子进程被视为另一个进程, 虽然子进程从父进程继承了其文件描述符, 但不能继承文件锁。这个约束是有道理的, 因为锁的作用就是阻止多个进程同时写同一个文件, 如果子进程通过 fork()继承了父进程的锁, 则父进程和子进程就可以同时写同一个文件了。

除此之外, 当一个文件描述符被复制时(譬如使用 dup()、dup2()或 fcntl()F_DUPFD 操作), 这些通过复制得到的文件描述符和源文件描述符都会引用同一个文件锁, 使用这些文件描述符中的任何一个进行解锁都可以, 如下所示:

```
flock(fd, LOCK_EX); //加锁
new_fd = dup(fd);
flock(new_fd, LOCK_UN); //解锁
```

这段代码先在 fd 上设置一个排它锁, 然后使用 dup()对 fd 进行复制得到新文件描述符 new_fd, 最后通过 new_fd 来解锁, 这样可以解锁成功。但是, 如果不显示的调用一个解锁操作, 只有当所有文件描述符都被关闭之后锁才会被释放。譬如上面的例子中, 如果不调用 flock(new_fd, LOCK_UN)进行解锁, 只有当 fd 和 new_fd 都被关闭之后锁才会自动释放。

关于本小节内容就暂时到这里为止! 接下来我们将学习使用 fcntl()对文件上锁。

13.6.2 fcntl()函数加锁

fcntl()函数在前面章节内容中已经多次用到了, 它是一个多功能文件描述符管理工具箱, 通过配合不同的 cmd 操作命令来实现不同的功能。为了方便述说, 这里再重申一次:

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */);
```

与锁相关的 cmd 为 F_SETLK、F_SETLKW、F_GETLK，第三个参数 flockptr 是一个 struct flock 结构体指针。使用 fcntl() 实现文件锁功能与 flock() 有两个比较大的区别：

- **flock()** 仅支持对整个文件进行加锁/解锁；而 **fcntl()** 可以对文件的某个区域（某部分内容）进行加锁/解锁，可以精确到某一个字节数据。
- **flock()** 仅支持建议性锁类型；而 **fcntl()** 可支持建议性锁和强制性锁两种类型。

我们先来看看 struct flock 结构体，如下所示：

示例代码 13.6.3 struct flock 结构体

```
struct flock {
    ...
    short l_type;        /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;     /* How to interpret l_start: SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;      /* Starting offset for lock */
    off_t l_len;        /* Number of bytes to lock */
    pid_t l_pid;        /* PID of process blocking our lock(set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

对 struct flock 结构体说明如下：

- **l_type**: 所希望的锁类型，可以设置为 F_RDLCK、F_WRLCK 和 F_UNLCK 三种类型之一，F_RDLCK 表示共享性质的读锁，F_WRLCK 表示独占性质的写锁，F_UNLCK 表示解锁一个区域。
- **l_whence** 和 **l_start**: 这两个变量用于指定要加锁或解锁区域的起始字节偏移量，与 2.7 小节所学的 lseek() 函数中的 offset 和 whence 参数相同，这里不再重述，如果忘记了，可以回到 2.7 小节再看看。
- **l_len**: 需要加锁或解锁区域的字节长度。
- **l_pid**: 一个 pid，指向一个进程，表示该进程持有的锁能阻塞当前进程，当 cmd=F_GETLK 时有效。

以上便是对 struct flock 结构体各成员变量的简单介绍，对于加锁和解锁区域的说明，还需要注意以下几项规则：

- 锁区域可以在当前文件末尾处开始或者越过末尾处开始，但是不能在文件起始位置之前开始。
- 若参数 l_len 设置为 0，表示将锁区域扩大到最大范围，也就是说从锁区域的起始位置开始，到文件的最大偏移量处（也就是文件末尾）都处于锁区域范围内。而且是动态的，这意味着不管向该文件追加写了多少数据，它们都处于锁区域范围，起始位置可以是文件的任意位置。
- 如果我们需要对整个文件加锁，可以将 l_whence 和 l_start 设置为指向文件的起始位置，并且指定参数 l_len 等于 0。

两种类型的锁：F_RDLCK 和 F_WRLCK

上面我们提到了两种类型的锁，分别为共享性读锁（F_RDLCK）和独占性写锁（F_WRLCK）。基本的规则与 12.5 小节所介绍的线程同步读写锁很相似，任意多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定的字节上只能有一个进程有一把独占写锁，进一步而言，如果在一个给定的字节上已经有一把或多把读锁，则不能在该字节上加写锁；如果在一个字节上已经有一把独占性写锁，则不能再对它加任何锁（包括读锁和写锁），下图显示了这些兼容性规则：

		请求	
		读锁	写锁
当前区域	无锁	允许	允许
	有一把或多把读锁	允许	拒绝
	有一把写锁	拒绝	拒绝

图 13.6.5 不同类型锁彼此之间的兼容性

如果一个进程对文件的某个区域已经上了一把锁, 后来该进程又试图在该区域再加一把锁, 那么通常新加的锁将替换旧的锁。譬如, 若某一进程在文件的 100~200 字节区间有一把写锁, 然后又试图在 100~200 字节区间再加一把读锁, 那么该请求将会成功执行, 原来的写锁会替换为读锁。

还需要注意另外一个问题, 当对文件的某一区域加读锁时, 调用进程必须对该文件有读权限, 譬如 `open()` 时 `flags` 参数指定了 `O_RDONLY` 或 `O_RDWR`; 当对文件的某一区域加写锁时, 调用进程必须对该文件有写权限, 譬如 `open()` 时 `flags` 参数指定了 `O_WRONLY` 或 `O_RDWR`。

F_SETLTK、F_SETLKW 和 F_GETLTK

我们来看看与文件锁相关的三个 `cmd` 它们的作用:

- **F_GETLTK:** 这种用法一般用于测试, 测试调用进程对文件加一把由参数 `flockptr` 指向的 `struct flock` 对象所描述的锁是否会加锁成功。如果加锁不成功, 意味着该文件的这部分区域已经存在一把锁, 并且由另一进程所持有, 并且调用进程加的锁与现有锁之间存在排斥关系, 现有锁会阻止调用进程想要加的锁, 并且现有锁的信息将会重写参数 `flockptr` 指向的对象信息。如果不存在这种情况, 也就是说 `flockptr` 指向的 `struct flock` 对象所描述的锁会加锁成功, 则除了将 `struct flock` 对象的 `l_type` 修改为 `F_UNLCK` 之外, 结构体中的其它信息保持不变。
- **F_SETLTK:** 对文件添加由 `flockptr` 指向的 `struct flock` 对象所描述的锁。譬如试图对文件的某一区域加读锁 (`l_type` 等于 `F_RDLCK`) 或写锁 (`l_type` 等于 `F_WRLCK`), 如果加锁失败, 那么 `fcntl()` 将立即出错返回, 此时将 `errno` 设置为 `EACCES` 或 `EAGAIN`。也可用于清除由 `flockptr` 指向的 `struct flock` 对象所描述的锁 (`l_type` 等于 `F_UNLCK`)。
- **F_SETLKW:** 此命令是 `F_SETLTK` 的阻塞版本 (命令名中的 `W` 表示等待 `wait`), 如果所请求的读锁或写锁因另一个进程当前已经对所请求区域的某部分进行了加锁, 而导致请求失败, 那么调用进程将会进入阻塞状态。只有当请求的锁可用时, 进程才会被唤醒。

`F_GETLTK` 命令一般很少用, 事先用 `F_GETLTK` 命令测试是否能够对文件加锁, 然后再用 `F_SETLTK` 或 `F_SETLKW` 命令对文件加锁, 但这两者并不是原子操作, 所以即使测试结果表明可以加锁成功, 但是在使用 `F_SETLTK` 或 `F_SETLKW` 命令对文件加锁之前也有可能被其它进程锁住。

使用示例与测试

示例代码 13.6.4 演示了使用 `fcntl()` 对文件加锁和解锁的操作。需要加锁的文件通过外部传参传入, 先调用 `open()` 函数以只写方式打开文件; 接着对 `struct flock` 类型对象 `lock` 进行填充, `l_type` 设置为 `F_WRLCK` 表示加一个写锁, 通过 `l_whence` 和 `l_start` 两个变量将加锁区域的起始位置设置为文件头部, 接着将 `l_len` 设置为 0 表示对整个文件加锁。

示例代码 13.6.4 使用 `fcntl()` 对文件加锁/解锁使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    struct flock lock = {0};
    int fd = -1;
    char buf[] = "Hello World!";

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_WRONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 对文件加锁 */
    lock.l_type = F_WRLCK; //独占性写锁
    lock.l_whence = SEEK_SET; //文件头部
    lock.l_start = 0; //偏移量为0
    lock.l_len = 0;
    if (-1 == fcntl(fd, F_SETLK, &lock)) {
        perror("加锁失败");
        exit(-1);
    }

    printf("对文件加锁成功!\n");

    /* 对文件进行写操作 */
    if (0 > write(fd, buf, strlen(buf))) {
        perror("write error");
        exit(-1);
    }

    /* 解锁 */
}
```

```
lock.l_type = F_UNLCK; //解锁
fcntl(fd, F_SETLK, &lock);

/* 退出 */
close(fd);
exit(0);
}
```

整个代码很简单, 比较容易理解, 具体执行的结果就不再给大家演示了。

一个进程可以对同一个文件的不同区域进行加锁, 当然这两个区域不能有重叠的情况。示例代码 13.6.5 演示了一个进程对同一文件的两个不同区域分别加读锁和写锁, 对文件的 100~200 字节区间加了一个写锁, 对文件的 400~500 字节区间加了一个读锁。

示例代码 13.6.5 对文件的不同区域进行加锁

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    struct flock wr_lock = {0};
    struct flock rd_lock = {0};
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件大小截断为 1024 字节 */
    ftruncate(fd, 1024);

    /* 对 100~200 字节区间加写锁 */
    wr_lock.l_type = F_WRLCK;
```

```
wr_lock.l_whence = SEEK_SET;
wr_lock.l_start = 100;
wr_lock.l_len = 100;
if (-1 == fcntl(fd, F_SETLK, &wr_lock)) {
    perror("加写锁失败");
    exit(-1);
}

printf("加写锁成功!\n");

/* 对 400~500 字节区间加读锁 */
rd_lock.l_type = F_RDLCK;
rd_lock.l_whence = SEEK_SET;
rd_lock.l_start = 400;
rd_lock.l_len = 100;
if (-1 == fcntl(fd, F_SETLK, &rd_lock)) {
    perror("加读锁失败");
    exit(-1);
}

printf("加读锁成功!\n");

/* 对文件进行 I/O 操作 */
// .....
// .....

/* 解锁 */
wr_lock.l_type = F_UNLCK; //写锁解锁
fcntl(fd, F_SETLK, &wr_lock);

rd_lock.l_type = F_UNLCK; //读锁解锁
fcntl(fd, F_SETLK, &rd_lock);

/* 退出 */
close(fd);
exit(0);
}
```

如果两个区域出现了重叠,譬如 100~200 字节区间和 150~250 字节区间,150~200 就是它们的重叠部分,一个进程对同一文件的相同区域不可能同时加两把锁,新加的锁会把旧的锁替换掉,譬如先对 100~200 字节区间加写锁、再对 150~250 字节区间加读锁,那么 150~200 字节区间最终是读锁控制的,关于这个问题,大家可以自己去验证、测试。

接下来对读锁和写锁彼此之间的兼容性进行测试,使用示例代码 13.6.6 测试读锁的共享性。

示例代码 13.6.6 读锁的共享性测试

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    struct flock lock = {0};
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件大小截断为 1024 字节 */
    ftruncate(fd, 1024);

    /* 对 400~500 字节区间加读锁 */
    lock.l_type = F_RDLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 400;
    lock.l_len = 100;
    if (-1 == fcntl(fd, F_SETLK, &lock)) {
        perror("加读锁失败");
        exit(-1);
    }

    printf("加读锁成功!\n");
    for (;;)
        sleep(1);
}
```

首先运行上述示例代码, 程序加读锁之后会进入死循环, 进程一直在运行着、持有读锁。接着多次运行上述示例代码, 启动多个进程加读锁, 测试结果如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
infile testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[1] 38277
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[2] 38278
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[3] 38279
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[4] 38280
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
  PID TTY          TIME CMD
  6535 pts/21    00:00:05 bash
  38277 pts/21    00:00:00 testApp
  38278 pts/21    00:00:00 testApp
  38279 pts/21    00:00:00 testApp
  38280 pts/21    00:00:00 testApp
  38283 pts/21    00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.6 读锁共享性测试

从打印信息可以发现, 多个进程对同一文件的相同区域都可以加读锁, 说明读锁是共享性的。由于程序是放置在后台运行的, 测试完毕之后, 可以使用 `kill` 命令将这些进程杀死, 或者直接关闭当前终端, 重新启动新的终端。

使用示例代码 13.6.7 测试写锁的独占性。

示例代码 13.6.7 写锁的独占性测试

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    struct flock lock = {0};
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
```



```

    perror("open error");
    exit(-1);
}

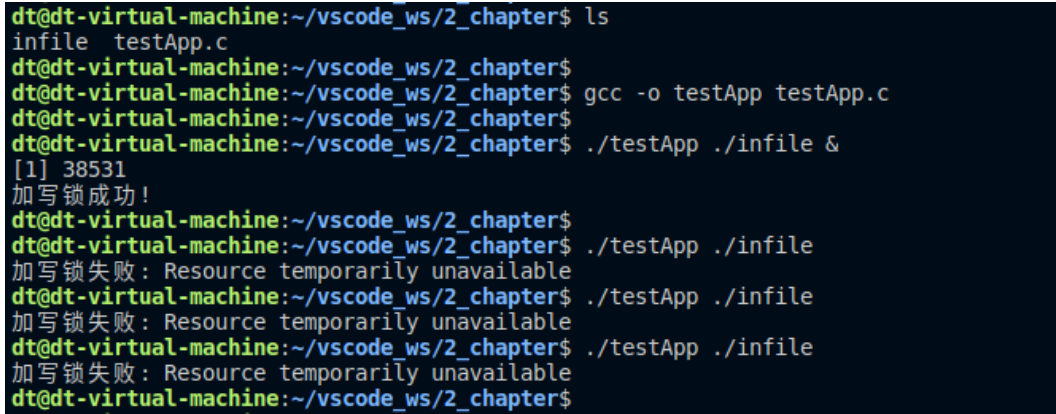
/* 将文件大小截断为 1024 字节 */
ftruncate(fd, 1024);

/* 对 400~500 字节区间加写锁 */
lock.l_type = F_WRLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 400;
lock.l_len = 100;
if (-1 == fcntl(fd, F_SETLK, &lock)) {
    perror("加写锁失败");
    exit(-1);
}

printf("加写锁成功!\n");
for (;;)
    sleep(1);
}

```

测试方法与读锁测试方法一样，如下所示：



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
infile testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[1] 38531
加写锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile
加写锁失败: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile
加写锁失败: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile
加写锁失败: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 13.6.7 写锁的独占性测试

由打印信息可知，但第一次启动的进程对文件加写锁之后，后面再启动进程对同一文件的相同区域加写锁发现都会失败，所以由此可知，写锁是独占性的。

几条规则

关于使用 `fcntl()` 创建锁的几条规则与 `flock()` 相似，如下所示：

- 文件关闭的时候，会自动解锁。
- 一个进程不可以对另一个进程持有的文件锁进行解锁。
- 由 `fork()` 创建的子进程不会继承父进程所创建的锁。

除此之外，当一个文件描述符被复制时（譬如使用 `dup()`、`dup2()` 或 `fcntl()` `F_DUPFD` 操作），这些通过复制得到的文件描述符和源文件描述符都会引用同一个文件锁，使用这些文件描述符中的任何一个进行解锁都可以，这点与 `flock()` 是一样的，如下所示：

```
lock.l_type = F_RDLCK;
fcntl(fd, F_SETLK, &lock);//加锁
```

```
new_fd = dup(fd);
```

```
lock.l_type = F_UNLCK;
fcntl(new_fd, F_SETLK, &lock);//解锁
```

这段代码先在 fd 上设置一个读锁, 然后使用 dup() 对 fd 进行复制得到新文件描述符 new_fd, 最后通过 new_fd 来解锁, 这样可以解锁成功。如果不显示的调用一个解锁操作, 任何一个文件描述符被关闭之后锁都会自动释放, 那么这点与 flock() 是不同的。譬如上面的例子中, 如果不调用 flock(new_fd, LOCK_UN) 进行解锁, 当 fd 或 new_fd 两个文件描述符中的任何一个被关闭之后锁都会自动释放。

建议性锁和强制性锁

前面我们提到了 fcntl() 支持强制性锁和建议性锁, 但是一般不建议使用强制性锁, 所以大部分情况下使用的都是建议性锁, 那如何使能强制性锁呢?

对于一个特定的文件, 开启它的强制性锁机制其实非常简单, 主要跟文件的权限位有关系, 在 5.5 小节对文件的权限进行了比较详细的介绍, 这里不再重述! 如果要开启强制性锁机制, 需要设置文件的 Set-Group-ID (S_ISGID) 位为 1, 并且禁止文件的组用户执行权限 (S_IXGRP), 也就是将其设置为 0。

但是, 有些 Linux/Unix 发行版系统并不支持强制性锁机制, 可以通过示例代码 13.6.8 进行测试。

示例代码 13.6.8 测试系统是否支持强制性锁机制

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    struct stat sbuf = {0};
    int fd = -1;
    pid_t pid;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0664);
    if (-1 == fd) {
        perror("open error");
    }
}
```

```
        exit(-1);
    }

    /* 写入一行字符串 */
    if (12 != write(fd, "Hello World!", 12)) {
        perror("write error");
        exit(-1);
    }

    /* 开启强制性锁机制 */
    if (0 > fstat(fd, &sbuf)) { //获取文件属性
        perror("fstat error");
        exit(-1);
    }
    if (0 > fchmod(fd, (sbuf.st_mode & ~S_IXGRP
        | S_ISGID)) {
        perror("fchmod error");
        exit(-1);
    }

    /* fork 创建子进程 */
    if (0 > (pid = fork())) //出错
        perror("fork error");
    else if (0 < pid) { //父进程
        struct flock lock = {0};

        /* 对整个文件加写锁 */
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 0;
        lock.l_len = 0;
        if (0 > fcntl(fd, F_SETLK, &lock))
            perror("父进程: 加写锁失败");
        else
            printf("父进程: 加写锁成功!\n");

        printf("~~~~~\n");
        if (0 > wait(NULL))
            perror("wait error");
    }
    else { //子进程
        struct flock lock = {0};
        int flag;
```

```
char buf[20] = {0};

sleep(1); //休眠 1 秒钟, 让父进程先运行

/* 设置为非阻塞方式 */
flag = fcntl(fd, F_GETFL);
flag |= O_NONBLOCK;
fcntl(fd, F_SETFL, flag);

/* 对整个文件加读锁 */
lock.l_type = F_RDLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;
if (-1 == fcntl(fd, F_SETLK, &lock))
    perror("子进程: 加读锁失败");
else
    printf("子进程: 加读锁成功!\n");

/* 读文件 */
if (0 > lseek(fd, 0, SEEK_SET))
    perror("lseek error");
if (0 > read(fd, buf, 12))
    perror("子进程: read error");
else
    printf("子进程: read OK, buf = %s\n", buf);
}

exit(0);
}
```

此程序首先创建了一个文件, 文件路径通过传参的方式传递给应用程序, 如果不存在该文件则创建它。接着向文件中写入数据, 开启文件的强制性锁机制。接下来程序调用 `fork()` 创建了一个子进程, 在父进程分支中, 对文件的所有区域加了一把独占性质的写锁, 接着调用 `wait()` 等到回收子进程; 在子进程分支中先是休眠了一秒钟以保证父进程先执行, 子进程将文件设置为非阻塞方式, 这里大家可能会有疑问? 普通文件不都是非阻塞的吗? 这里为什么要设置非阻塞呢? 并不是多此一举, 原因在于这里涉及到了强制性锁的问题, 在强制性锁机制下, 如果文件被进程添加了强制性写锁, 其它进程读或写该文件将会被阻塞, 所以我们需要显式设置为非阻塞方式。

设置为非阻塞之后, 子进程试图对文件设置一把读锁, 接着子进程将文件读、写位置移动到文件头, 并试图 `read` 读该文件。

由于父进程已经对文件设置了写锁, 子进程试图对文件设置读锁时, 将会失败; 子进程在没有获取到读锁的情况下, 调用 `read()` 读取文件将会出现两种情况: 如果系统支持强制性锁机制, 那么 `read()` 将会失败; 如果系统不支持强制性锁机制, `read()` 将会成功!

接下来我们进行测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp infile
父进程: 加写锁成功!
~~~~~
子进程: 加读锁失败: Resource temporarily unavailable
子进程: read OK, buf = Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.8 Ubuntu 系统下测试结果

从打印信息可以发现,父进程设置了写锁的情况下,子进程再次对其设置读锁是不成功的,也就是子进程没有获取到读锁,但是读文件却是成功的,由此可知,我们测试所使用的 Ubuntu 系统不支持强制性锁机制。

13.6.3 lockf()函数加锁

lockf()函数是一个库函数,其内部是基于 fcntl()来实现的,所以 lockf()是对 fcntl 锁的一种封装,具体的使用方法这里便不再介绍。

13.7 小结

本章向大家介绍了几种高级 I/O 功能,非阻塞 I/O、I/O 多路复用、异步 I/O、存储映射 I/O、以及文件锁,其中有许多功能,我们将会在后面的提高篇和进阶篇章实例中使用到。

- 非阻塞 I/O: 进程向文件发起 I/O 操作,使其不会被阻塞。
- I/O 多路复用: select()和 poll()函数。
- 异步 I/O: 当文件描述符上可以执行 I/O 操作时,内核会向进程发送信号通知它。
- 存储映射 I/O: mmap()函数。
- 文件锁: flock()、fcntl()以及 lockf()函数。

第十四章 本篇总结

经过十几章内容的学习,终于结束了入门篇章内容的学习,大家辛苦了!同样也祝贺大家在这个过程中学习到了很多的知识内容,但是学完了并不代表你能够真正地运用起来,在实际的应用编程中,需要能够根据不同的应用场景使用不同的编程技巧来解决它!所以需要大家对前面学习过的编程技巧加以练习,能够独立完成每章的编程练习、并理解它!

对本篇学习的内容,笔者将对此做一个简单地回顾与总结:

- 第一章 应用编程概念:本章介绍了何为应用编程,与逻辑编程、驱动编程有什么区别。
- 第二章 文件 I/O 基础:文件 I/O 作为 Linux 最基本、最重要的编程技巧,必然要掌握于心!
- 第三章 深入探究文件 I/O:本章带大家深入了解了文件 I/O 中的一些细节,譬如文件的管理方式、错误返回的处理、空洞文件、O_APPEND 和 O_TRUNC 标志、原子操作与竞争冒险等等。
- 第四章 标准 I/O 库:本章介绍了标准 I/O 库,使用标准 I/O 库函数对文件进行 I/O 操作、标准 I/O 库函数与普通 I/O read()/write()之间的区别、标准 I/O 库的缓冲与文件 I/O 的内核缓冲等。
- 第五章 文件属性与目录:本章介绍了文件相关的特性以及相关属性,譬如文件类型、文件属性、文件访问权限、文件时间戳、符号链接与硬链接等。
- 第六章 字符串处理:字符串处理在几乎所有的编程语言中都是一个绕不开的话题,本章介绍了 C 库函数中提供的一些用于处理字符串相关的函数以及正则表达式。
- 第七章 系统信息与系统资源:本章介绍了用于获取系统信息相关的函数接口,譬如系统信息 sysinfo()、系统时间日期、proc 文件系统等,以及系统资源的使用,譬如申请堆内存。
- 第八章 信号:在很多应用程序当中,都会存在处理异步事件这种需求,而信号提供了一种处理异步事件的方法,本章向大家介绍了 Linux 下的信号相关的内容,包括:信号的概念、信号的分类、进程对信号的处理、发送信号、信号掩码、实时信号等。
- 第九章 进程:本章介绍了进程相关的内容,包括:进程的概念、fork()创建子进程、父子进程间的文件共享、进程的诞生与终止、进程状态与进程关系、守护进程等。
- 第十章 进程间通信:本章内容向大家介绍 Linux 下提供的进程间通信的手段,用于在多进程的环境下,在一些中小型的程序设计中,多进程的设计其实很少用到,主要用在一些大型项目中,本章以了解为主,在实际编程中需要用到再去深入学习即可!
- 第十一章 线程:本章介绍了线程相关的内容,多线程编程在实际的 Linux 应用项目中占了很大一部分,所以多线程是大家必须要掌握的一个编程技巧。
- 第十二章 线程同步:涉及到线程,那就必然绕不开线程同步,本章介绍了用于实现线程同步的几种不同的方式以及它们的原理,譬如互斥锁、条件变量、自旋锁以及读写锁等,不同的方式适用于不同的场景,需要根据应用场景来选择!
- 第十三章 高级 I/O:本章介绍了文件 I/O 当中的一些高级用法,包括:非阻塞 I/O、I/O 多路复用、异步 I/O、存储映射 I/O 以及文件锁。

本书中关于入门篇的内容就到这里为止了,接下来将会进入到提高篇内容的学习,大家加油!

第二篇 提高篇

在结束了长达十几章内容的入门篇学习之后,从这里开始将进入到提高篇章内容的学习。在入门篇章内容中所涉及到的所有测试例程我们都是在 Ubuntu 系统下运行验证的,当然对于嵌入式 Linux 系统来说,这些测试程序同样也是没有问题的、是可以直接移植过去使用的,使用嵌入式硬件平台对应的交叉编译工具编译应用程序即可在嵌入式 Linux 系统上运行。

在嵌入式 Linux 系统中,我们编写的应用程序通常需要与硬件设备进行交互、操控硬件,譬如点亮开发板上的一颗 LED 灯、获取按键输入数据、在 LCD 屏上显示摄像头采集的图像、应用程序向串口发送数据或采集串口数据、网络编程等,那么本篇我们将会以正点原子的 ALPHA/Mini I.MX6U 开发板为例,向大家介绍如何编写应用程序控制开发板上的各种硬件外设;Linux 系统下,一切皆文件,也包括各种硬件设备,所以在 Linux 系统下,各种硬件设备是以文件的形式呈现给用户层,应用程序通过对文件的 I/O 操作来控制硬件设备;所以本篇所用到的编程知识均是入门篇中学习过的内容,所以大家不用担心学不会,因为本篇没什么新的编程知识点,只是针对不同硬件我们应该如何去编程操控它,这个将是本篇学习的重点!

本篇虽以正点原子的 ALPHA/Mini I.MX6U 开发板为例进行讲解,但对于其它嵌入式 Linux 开发板来说,同样也是适用的,对于应用编程不需要关注底层硬件驱动的实现,所以自然不受具体硬件驱动实现的影响。OK,废话不多说,我们开始正式学习本篇内容,大家加油!

Tips: 本篇将以正点原子 ALPHA/Mini I.MX6U 开发板出厂系统进行测试!!!

第十五章 点亮 LED

对于一款学习型开发板来说, 永远都绕不开 LED 这个小小的设备, 基本上每块板子都至少会有一颗 LED 小灯, 对于我们的 ALPHA/Mini I.MX6U 开发板来说同样也是如此。

ALPHA/Mini I.MX6U 开发板 (包括核心板和底板) 上一共有 3 颗 LED 小灯, 当仅有一颗 LED 能够被用户所控制, 其它两颗均作为电源指示灯而存在, 用户对其不可控制; LED 通常是由 GPIO 所控制的, 本章我们来学习如何编写应用程序控制 LED 灯的亮灭。

本章将会讨论如下主题内容。

- 应用层控制外设的两种不同的方式;
- sysfs 文件系统介绍;
- 如何控制 LED, LED 相关属性文件介绍;

15.1 应用层操控硬件的两种方式

在 Linux 系统下，一切皆文件！应用层如何操控底层硬件，同样也是通过文件 I/O 的方式来实现，前面我们给大家介绍了设备文件，包括字符设备文件和块设备文件，为啥叫设备文件？大家有没有想过这个问题呢？其实设备文件便是各种硬件设备向应用层提供的一个接口，应用层通过对设备文件的 I/O 操作来操控硬件设备，譬如 LCD 显示屏、串口、按键、摄像头等等，所以设备文件其实是与硬件设备相互对应的。设备文件通常在 /dev/ 目录下，我们也把 /dev 目录下的文件称为设备节点。

设备节点并不是操控硬件设备的唯一途径，除此之外，我们还可以通过 sysfs 文件系统对硬件设备进行操控，接下来将进行介绍！

15.1.1 sysfs 文件系统

简单的说，sysfs 是一个基于内存的文件系统，同 devfs、proc 文件系统一样，称为虚拟文件系统；它的作用是将内核信息以文件的方式提供给应用层使用。7.7 小节中我们学习过 proc 文件系统，应用层可以通过 proc 文件系统得到系统信息和进程相关信息，与 proc 文件系统类似，sysfs 文件系统的主要功能便是对系统设备进行的管理，它可以产生一个包含所有系统硬件层次的视图。

sysfs 文件系统把连接在系统上的设备和总线组织成为一个分级的文件、展示设备驱动模型中各组件的层次关系。sysfs 提供了一种机制，可以显式的描述内核对象、对象属性及对象间关系，用来导出内核对象 (kernel object, 譬如一个硬件设备) 的数据、属性到用户空间，以文件目录结构的形式为用户空间提供对这些数据、属性的访问支持。表 15.1.1 描述了内核对象、对象属性及对象间关系在用户空间 sysfs 中的表现：

内核中的组成要素	sysfs 中的表现
内核对象（譬如一个硬件设备）	目录
对象属性（譬如设备属性）	文件
对象关系	链接文件

表 15.1.1 内核对象、对象属性及对象间关系在用户空间的表现

15.1.2 sysfs 与 /sys

sysfs 文件系统挂载在 /sys 目录下，启动 ALPHA/Mini I.MX6U 开发板，进入 Linux 系统（开发板出厂系统）之后，我们进入到 /sys 目录下查看，如下所示：

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# cd /sys/
root@ATK-IMX6U:/sys#
root@ATK-IMX6U:/sys# pwd
/sys
root@ATK-IMX6U:/sys#
root@ATK-IMX6U:/sys# ls
block bus class dev devices firmware fs fsl_otp kernel module power
root@ATK-IMX6U:/sys#
```

图 15.1.1 /sys 目录

上图显示的便是 sysfs 文件系统目录，包括 block、bus、class、dev、devices、firmware、fs、kernel、modules、power 等，每个目录下又有许多文件或子目录，对这些目录的说明如所示：

/sys 下的子目录	说明
/sys/devices	这是系统中所有设备存放的目录，也就是系统中的所有设备在 sysfs 中的呈现、表达，也是 sysfs 管理设备的最重要的目录结构。
/sys/block	块设备的存放目录，这是一个过时的接口，按照 sysfs 的设计理念，系统所有的设备都存放在 /sys/devices 目录下，所

	以/sys/block 目录下的文件通常是链接到/sys/devices 目录下的文件。
/sys/bus	这是系统中的所有设备按照总线类型分类放置的目录结构, /sys/devices 目录下每一种设备都是挂在某种总线下的, 譬如 i2c 设备挂在 I2C 总线下。同样, /sys/bus 目录下的文件通常也是链接到了/sys/devices 目录。
/sys/class	这是系统中的所有设备按照其功能分类放置的目录结构, 同样该目录下的文件也是链接到了/sys/devices 目录。按照设备的功能划分组织在/sys/class 目录下, 譬如/sys/class/leds 目录中存放了所有的 LED 设备, /sys/class/input 目录中存放了所有的输入类设备。
/sys/dev	这是按照设备号的方式放置的目录结构, 同样该目录下的文件也是链接到了/sys/devices 目录。该目录下有很多以主设备号:次设备号 (major:minor) 命名的文件, 这些文件都是链接文件, 链接到/sys/devices 目录下对应的设备。
/sys/firmware	描述了内核中的固件。
/sys/fs	用于描述系统中所有文件系统, 包括文件系统本身和按文件系统分类存放的已挂载点。
/sys/kernel	这里是内核中所有可调参数的位置。
/sys/module	这里有系统中所有模块的信息。
/sys/power	这里是系统中电源选项, 有一些属性可以用于控制整个系统的电源状态。

表 15.1.2 /sys 目录结构

系统中所有的设备(对象)都会在/sys/devices 体现出来, 是 sysfs 文件系统中最重要的目录结构; 而 /sys/bus、/sys/class、/sys/dev 分别将设备按照挂载的总线类型、功能分类以及设备号的形式将设备组织存放在这些目录中, 这些目录下的文件都是链接到了/sys/devices 中。

设备的一些属性、数据通常会通过设备目录下的文件体现出来, 也就是说设备的数据、属性会导出到用户空间, 以文件形式为用户空间提供对这些数据、属性的访问支持, 可以把这些文件称为属性文件; 读这些属性文件就表示读取设备的属性信息, 相反写属性文件就表示对设备的属性进行设置、以控制设备的状态。

15.1.3 总结

这里给大家进行一个总结, 应用层想要对底层硬件进行操控, 通常可以通过两种方式:

- /dev/目录下的设备文件(设备节点);
- /sys/目录下设备的属性文件。

具体使用哪种方式需要根据不同功能类型设备进行选择, 有些设备只能通过设备节点进行操控, 而有些设备只能通过 sysfs 方式进行操控; 当然跟设备驱动具体的实现方式有关, 通常情况下, 一般简单地设备会使用 sysfs 方式操控, 其设备驱动在实现时会将设备的一些属性导出到用户空间 sysfs 文件系统, 以属性文件的形式为用户空间提供对这些数据、属性的访问支持, 譬如 LED、GPIO 等。

但对于一些较复杂的设备通常会使用设备节点的方式, 譬如 LCD 等、触摸屏、摄像头等。

15.1.4 标准接口与非标准接口

Linux 内核中为了尽量降低驱动开发者难度以及接口标准化, 就出现了设备驱动框架的概念; Linux 针对各种常见的设备进行分类, 譬如 LED 类设备、输入类设备、FrameBuffer 类设备、video 类设备、PWM 设备等等, 并为每一种类型的设备设计了一套成熟的、标准的、典型的驱动实现的框架, 这个就叫做设备驱动框架。设备驱动框架为驱动开发和应用层提供了一套统一的接口规范, 譬如对 LED 类设备来说, 内核提供了 LED 设备驱动框架, 驱动工程师编写 LED 驱动时, 使用 LED 驱动框架来开发自己的 LED 驱动程序, 这样做的好处就在于, 能够对上层应用层提供统一、标准化的接口、同时又降低了驱动开发工程师的难度。

编写 LED 驱动程序并不仅仅只能使用内核设计的 LED 设备驱动框架, 不用内核的 LED 驱动框架也是可以开发出 LED 驱动程序的, 但如果你这样写, 使用这个驱动程序注册的 LED 那就不是标准设备了, 因为该驱动程序向应用层提供的接口并不是统一、标准化接口。

除此之外, 还有很多硬件外设, 尤其是嵌入式系统中所使用到的这些硬件外设, 它们可能并不属于 Linux 系统所规划的设备分类当中的任何一种设备类型, 譬如在 Linux 系统中, 有一种设备类型叫杂散/杂项类设备 (misc device), 大家可以想一想为啥叫杂散类设备, 说明这种设备既不属于这种设备类型、又不属于另一种设备类型, 无奈只能把它归为杂项类。

因为一个计算机系统所能够连接、使用的外设实在太多了, 不可能每一种外设都能够精准地分类到某一个设备类型中, 通常把这些无法进行分类的外设就称为杂项设备, 杂项设备驱动程序向应用层提供的接口通常都不是标准化接口、它是一种非标准接口, 具体如何去操控这个设备通常只有驱动工程师知道。所以在嵌入式系统中, 很多硬件外设的驱动程序都是定制的。

15.2 LED 硬件控制方式

ALPHA/Mini I.MX6U 开发板底板上有一颗可被用户控制的 LED 灯, 如下所示:

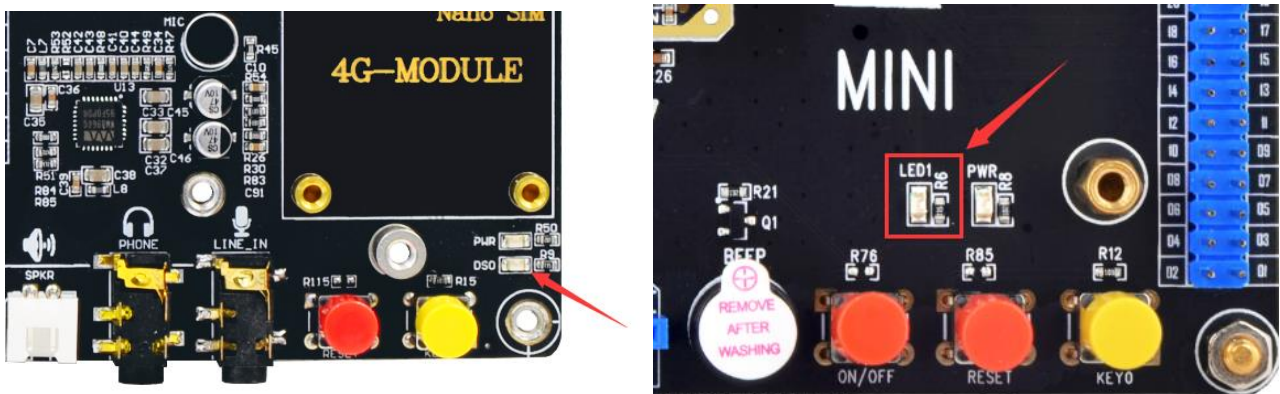


图 15.2.1 用户 LED 灯 (左 ALPHA、右 Mini)

上图中箭头所指的 LED 便是开发板上唯一一个可以被用户所控制的 LED, 另外一颗 LED 则 (名称为 PWR) 是底板上的电源指示灯。

对于 ALPHA/Mini I.MX6U 开发板出厂系统来说, 此 LED 设备使用的是 Linux 内核标准 LED 驱动框架注册而成, 在 /dev 目录下并没有其对应的设备节点, 其实现使用 sysfs 方式控制。进入到 /sys/class/leds 目录下, 如下所示:

```

root@ATK-IMX6U:~# cd /sys/class/leds/
root@ATK-IMX6U:/sys/class/leds#
root@ATK-IMX6U:/sys/class/leds# pwd
/sys/class/leds
root@ATK-IMX6U:/sys/class/leds#
root@ATK-IMX6U:/sys/class/leds# ls
beep mmc0:: sys-led
root@ATK-IMX6U:/sys/class/leds#

```

图 15.2.2 /sys/class/leds 目录

上小节介绍了 /sys/class 目录，系统中的所有设备根据其功能分类组织到了 /sys/class 目录下，所以 /sys/class/leds 目录下便存放了所有的 LED 类设备。从上图可以看到该目录下有一个 sys-led 文件夹，这个便是底板上的用户 LED 设备文件夹，进入到该目录下，如下所示：

```

root@ATK-IMX6U:/sys/class/leds# cd sys-led
root@ATK-IMX6U:/sys/class/leds/sys-led#
root@ATK-IMX6U:/sys/class/leds/sys-led# pwd
/sys/class/leds/sys-led
root@ATK-IMX6U:/sys/class/leds/sys-led#
root@ATK-IMX6U:/sys/class/leds/sys-led# ls
brightness device max_brightness power subsystem trigger uevent
root@ATK-IMX6U:/sys/class/leds/sys-led#
root@ATK-IMX6U:/sys/class/leds/sys-led#

```

图 15.2.3 sys-led 文件夹

这里我们主要关注便是 brightness、max_brightness 以及 trigger 三个文件，这三个文件都是 LED 设备的属性文件：

- **brightness**: 翻译过来就是亮度的意思，该属性文件可读可写；所以这个属性文件是用于设置 LED 的亮度等级或者获取当前 LED 的亮度等级，譬如 brightness 等于 0 表示 LED 灭，brightness 为正整数表示 LED 亮，其值越大、LED 越亮；对于 PWM 控制的 LED 来说，这通常是适用的，因为它存在亮度等级的问题，不同的亮度等级对应不同的占空比，自然 LED 的亮度也是不同的；但对于 GPIO 控制（控制 GPIO 输出高低电平）的 LED 来说，通常不存在亮度等级这样的说法，只有 LED 亮（brightness 等于 0）和 LED 灭（brightness 为非 0 值的正整数）两种状态，ALPHA/Mini I.MX6U 开发板上的这颗 LED 就是如此，所以自然就不存在亮度等级一说，只有亮和灭两种亮度等级。
- **max_brightness**: 该属性文件只能被读取，不能写，用于获取 LED 设备的最大亮度等级。
- **trigger**: 触发模式，该属性文件可读可写，读表示获取 LED 当前的触发模式，写表示设置 LED 的触发模式。不同的触发模式其触发条件不同，LED 设备会根据不同的触发条件自动控制其亮、灭状态，通过 cat 命令查看该属性文件，可获取 LED 支持的所有触发模式以及 LED 当前被设置的触发模式：

```

root@ATK-IMX6U:/sys/class/leds/sys-led# ls
brightness device max_brightness power subsystem trigger uevent
root@ATK-IMX6U:/sys/class/leds/sys-led#
root@ATK-IMX6U:/sys/class/leds/sys-led#
root@ATK-IMX6U:/sys/class/leds/sys-led# cat trigger
none rc-feedback nand-disk mmc0 timer oneshot [heartbeat] backlight gpio
root@ATK-IMX6U:/sys/class/leds/sys-led#

```

图 15.2.4 LED 支持的所有触发模式

方括号 ([heartbeat]) 括起来的表示当前 LED 对应的触发模式，none 表示无触发，常用的触发模式包括 none（无触发）、mmc0（当对 mmc0 设备发起读写操作的时候 LED 会闪烁）、timer（LED 会有规律的一亮一灭，被定时器控制住）、heartbeat（心跳呼吸模式，LED 模仿人的心跳呼吸那样亮灭变化）。

通常系统启动之后,会将板上的一颗 LED 设置为 heartbeat 触发模式,将其作为系统正常运行的指示灯,譬如 ALPHA/Mini I.MX6U 开发板系统启动之后,底板上的用户 LED 就会处于心跳呼吸模式,这个大家自己观察便可知道。

通过上面的介绍,已经知道如何去控制 ALPHA/Mini I.MX6U 开发板底板上的用户 LED 了,譬如通过 echo 命令进行控制:

```
echo timer > trigger    //将 LED 触发模式设置为 timer

echo none > trigger     //将 LED 触发模式设置为 none
echo 1 > brightness    //点亮 LED    echo 0 > brightness//熄灭 LED
```

大家可以自己动手使用 echo 或 cat 命令进行测试、控制 LED 状态;除了使用 echo 或 cat 命令之后,同样我们编写应用程序,使用 write()、read()函数对这些属性文件进行 I/O 操作以达到控制 LED 的效果。

Tips: 命令 cat 读取以及 echo 写入到属性文件中的均是字符串,所以如果在应用程序中通过 write()向属性文件写入数据,同样也要是字符串形式;同理,使用 read()读取的数据也是字符串 ASCII 编码的。

15.3 编写 LED 应用程序

通过上一小节的介绍,我们已经知道了如何控制 LED,接下来编写一个简单地示例代码演示如何控制 LED,测试代码如下所示:

本例程源码对应的路径为: **开发板光盘->11、Linux C 应用编程例程源码->15_led->led.c。**

示例代码 15.3.1 LED 应用程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define LED_TRIGGER    "/sys/class/leds/sys-led/trigger"
#define LED_BRIGHTNESS "/sys/class/leds/sys-led/brightness"
#define USAGE()      fprintf(stderr, "usage:\n" \
                        "    %s <on|off>\n" \
                        "    %s <trigger> <type>\n", argv[0], argv[0])

int main(int argc, char *argv[])
{
    int fd1, fd2;

    /* 校验传参 */
    if (2 > argc) {
        USAGE();
        exit(-1);
    }
}
```

```
/* 打开文件 */
fd1 = open(LED_TRIGGER, O_RDWR);
if (0 > fd1) {
    perror("open error");
    exit(-1);
}

fd2 = open(LED_BRIGHTNESS, O_RDWR);
if (0 > fd2) {
    perror("open error");
    exit(-1);
}

/* 根据传参控制 LED */
if (!strcmp(argv[1], "on")) {
    write(fd1, "none", 4); //先将触发模式设置为 none
    write(fd2, "1", 1); //点亮 LED
}
else if (!strcmp(argv[1], "off")) {
    write(fd1, "none", 4); //先将触发模式设置为 none
    write(fd2, "0", 1); //LED 灭
}
else if (!strcmp(argv[1], "trigger")) {
    if (3 != argc) {
        USAGE();
        exit(-1);
    }

    if (0 > write(fd1, argv[2], strlen(argv[2])))
        perror("write error");
}
else
    USAGE();

exit(0);
}
```

程序中定义了两个宏, LED_TRIGGER 和 LED_BRIGHTNESS, 分别对应/sys/class/leds/sys-led/trigger 和 /sys/class/leds/sys-led/brightness 属性文件, 宏 USAGE()用于打印程序的使用方法; 程序首先会调用 open()函数打开这两个属性文件, 之后判断传入参数指向相应的动作, 传入"on"表示点亮 LED, 先调用 write()将"none"写入到 trigger 属性文件中, 也就是设置为无触发, 接着再向 brightness 属性文件中写入"1"点亮 LED; 传入"off"表示熄灭 LED, 同样也是先调用 write()将"none"写入到 trigger 属性文件设置 LED 为无触发, 接着再向 brightness 属性文件中写入"0"熄灭 LED; 传入"trigger"表示设置 LED 的触发模式, 则需要传入第二个参数, 第二个参数表示需要设置的模式。

整个代码非常简单,接下来对测试代码进行编译,需要注意的时,由于我们是在 ALPHA/Mini I.MX6U 开发板上运行程序,所以需要 I.MX6U 平台对应的交叉编译工具来编译测试代码,这样编译得到的可执行文件才能在开发板上运行。

首先大家需要安装 I.MX6U 硬件平台对应的交叉编译工具,如何安装呢?直接参考“[开发板光盘资料 A-基础资料/【正点原子】I.MX6U 用户快速体验 V1.7.3.pdf](#)”文档中的第四章内容,根据文档的指示安装好交叉编译工具,当然如果你已经在 Ubuntu 系统下安装过了,就不用再次安装了。

安装完成之后,在使用之前先对交叉编译工具的环境进行设置,使用 source 执行安装目录下的 environment-setup-cortexa7hf-neon-poky-linux-gnueabi 脚本文件即可,如下所示:

```
source /opt/fsl-imx-x11/4.1.15-2.1.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

/opt/fsl-imx-x11/4.1.15-2.1.0 便是笔者在 Ubuntu 系统下安装交叉编译工具时对应的安装目录,大家根据自己的情况设置正确的路径。处理完成之后,接下来我们便可以对示例代码 15.3.1 进行编译了:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 15.3.1 交叉编译应用程序

CC 变量其实就是交叉编译工具,如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ echo ${CC}
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 15.3.2 变量 CC

所以 CC 环境变量其实就是 ARM 架构下的 gcc 编译器---交叉编译工具 arm-poky-linux-gnueabi-gcc,后面指定了一些选项,这些选项就不用管了;编译成功之后,会生成可在开发板上运行的可执行文件 testApp,使用 file 命令可以查看 testApp 可执行文件的类型:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ file testApp
testApp: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
4805b9d61e07, not stripped
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 15.3.3 testApp 文件类型

可以看出该文件是一个 32 位 ARM 架构下的可执行文件。

15.4 在开发板上测试

启动开发板进入 Linux 系统,将上小节编译得到的可执行文件 testApp 拷贝到开发板根文件系统中,譬如拷贝到开发板 Linux 系统的家目录下,如下图所示:

```
root@ATK-IMX6U:~# pwd
/home/root
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver  shell  testApp
root@ATK-IMX6U:~#
```




图 15.4.1 testApp 拷贝到开发板

拷贝方法很多, 推荐大家使用 scp 命令, 这里就不再介绍了。

接下来执行 testApp 程序测试:

```
./testApp on      # 点亮 LED
./testApp off     # 熄灭 LED
./testApp trigger heartbeat # 将 LED 触发模式设置为 heartbeat
```

```
root@ATK-IMX6U:~# ./testApp on
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp off
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp trigger heartbeat
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 15.4.2 测试结果

查看 LED 状态是否与程序执行的效果一致!

第十六章 GPIO 应用编程

本章介绍应用层如何控制 GPIO, 譬如控制 GPIO 输出高电平、或输出低电平。

本章将会讨论如下主题内容。

- 应用层 GPIO 编程介绍;
- GPIO 输出测试;
- GPIO 输入测试;
- GPIO 中断测试;

16.1 应用层如何操控 GPIO

与 LED 设备一样, GPIO 同样也是通过 sysfs 方式进行操控, 进入到 `/sys/class/gpio` 目录下, 如下所示:

```
root@ATK-IMX6U:~# cd /sys/class/gpio/
root@ATK-IMX6U:/sys/class/gpio#
root@ATK-IMX6U:/sys/class/gpio# pwd
/sys/class/gpio
root@ATK-IMX6U:/sys/class/gpio#
root@ATK-IMX6U:/sys/class/gpio# ls
export  gpiochip0  gpiochip128  gpiochip32  gpiochip64  gpiochip96  unexport
root@ATK-IMX6U:/sys/class/gpio#
root@ATK-IMX6U:/sys/class/gpio#
```

图 16.1.1 `/sys/class/gpio` 目录

可以看到该目录下包含两个文件 `export`、`unexport` 以及 5 个 `gpiochipX` (X 等于 0、32、64、96、128) 命名的文件夹。

- **gpiochipX:** 当前 SoC 所包含的 GPIO 控制器, 我们知道 I.MX6UL/I.MX6ULL 一共包含了 5 个 GPIO 控制器, 分别为 GPIO1、GPIO2、GPIO3、GPIO4、GPIO5, 在这里分别对应 `gpiochip0`、`gpiochip32`、`gpiochip64`、`gpiochip96`、`gpiochip128` 这 5 个文件夹, 每一个 `gpiochipX` 文件夹用来管理一组 GPIO。随便进入到其中某个目录下, 可以看到这些目录下包含了如下文件:

```
root@ATK-IMX6U:/sys/class/gpio#
root@ATK-IMX6U:/sys/class/gpio# cd gpiochip0/
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
root@ATK-IMX6U:/sys/class/gpio/gpiochip0# ls
base  device  label  ngpio  power  subsystem  uevent
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
```

图 16.1.2 `gpiochip0` 目录下的文件

在这个目录我们主要关注的是 `base`、`label`、`ngpio` 这三个属性文件, 这三个属性文件均是只读、不可写。

base: 与 `gpiochipX` 中的 X 相同, 表示该控制器所管理的这组 GPIO 引脚中最小的编号。每一个 GPIO 引脚都会有一个对应的编号, Linux 下通过这个编号来操控对应的 GPIO 引脚。

```
root@ATK-IMX6U:/sys/class/gpio/gpiochip0# cat base
0
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
```

图 16.1.3 `base` 属性文件

label: 该组 GPIO 对应的标签, 也就是名字。

```
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
root@ATK-IMX6U:/sys/class/gpio/gpiochip0# cat label
209c000.gpio
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
root@ATK-IMX6U:/sys/class/gpio/gpiochip0#
```

图 16.1.4 `label` 属性文件

ngpio: 该控制器所管理的 GPIO 引脚的数量 (所以引脚编号范围是: `base ~ base+ngpio-1`)。

```
root@ATK-IMX6U:/sys/class/gpio/gpiochip128# cat ngpio
32
root@ATK-IMX6U:/sys/class/gpio/gpiochip128#
```

图 16.1.5 `ngpio` 属性文件

对于给定的一个 GPIO 引脚, 如何计算它在 sysfs 中对应的编号呢? 其实非常简单, 譬如给定一个 GPIO 引脚为 `GPIO4_IO16`, 那它对应的编号是多少呢? 首先我们要确定 `GPIO4` 对应于 `gpiochip96`, 该组 GPIO 引

脚的最小编号是 96 (对应于 GPIO4_IO0)，所以 GPIO4_IO16 对应的编号自然是 $96 + 16 = 112$ ；同理 GPIO3_IO20 对应的编号是 $64 + 20 = 84$ 。

- **export**: 用于将指定编号的 GPIO 引脚导出。在使用 GPIO 引脚之前，需要将其导出，导出成功之后才能使用它。注意 export 文件是只写文件，不能读取，将一个指定的编号写入到 export 文件中即可将对应的 GPIO 引脚导出，譬如：

```
echo 0 > export      # 导出编号为 0 的 GPIO 引脚（对于 I.MX6UL/I.MX6ULL 来说，也就是 GPIO1_IO0）
```

```
root@ATK-IMX6U:/sys/class/gpio#
root@ATK-IMX6U:/sys/class/gpio# echo 0 > export
root@ATK-IMX6U:/sys/class/gpio#
```

图 16.1.6 导出成功

导出成功之后会发现在 `/sys/class/gpio` 目录下生成了一个名为 `gpio0` 的文件夹（`gpioX`，`X` 表示对应的编号），如图 16.1.7 所示。这个文件夹就是导出来的 GPIO 引脚对应的文件夹，用于管理、控制该 GPIO 引脚，稍后再给大家介绍。

```
root@ATK-IMX6U:/sys/class/gpio#
root@ATK-IMX6U:/sys/class/gpio# ls
export gpio0 gpiochip0 gpiochip128 gpiochip32 gpiochip64 gpiochip96 unexport
root@ATK-IMX6U:/sys/class/gpio#
```

图 16.1.7 GPIO 引脚对应的文件夹

- **unexport**: 将导出的 GPIO 引脚删除。当使用完 GPIO 引脚之后，我们需要将导出的引脚删除，同样该文件也是只写文件、不可读，譬如：

```
echo 0 > unexport    # 删除导出的编号为 0 的 GPIO 引脚
```

删除成功之后，之前生成的 `gpio0` 文件夹就会消失！

以上就给大家介绍了 `/sys/class/gpio` 目录下的所有文件和文件夹，控制 GPIO 引脚主要是通过 export 导出之后所生成的 `gpioX`（`X` 表示对应的编号）文件夹，在该文件夹目录下存在一些属性文件可用于控制 GPIO 引脚的输入、输出以及输出的电平状态等。

Tips: 需要注意的是，并不是所有 GPIO 引脚都可以成功导出，如果对应的 GPIO 已经在内核中被使用了，那便无法成功导出，打印如下信息：

```
root@ATK-IMX6U:/sys/class/gpio# echo 3 > export
-sh: echo: write error: Device or resource busy
root@ATK-IMX6U:/sys/class/gpio#
```

图 16.1.8 导出失败

那也就是意味着该引脚已经被内核使用了，譬如某个驱动使用了该引脚，那么将无法导出成功！

gpioX

将指定的编号写入到 export 文件中，可以导出指定编号的 GPIO 引脚，导出成功之后会在 `/sys/class/gpio` 目录下生成对应的 `gpioX`（`X` 表示 GPIO 的编号）文件夹，以前面所生成的 `gpio0` 为例，进入到 `gpio0` 目录，该目录下的文件如下所示：

```
root@ATK-IMX6U:/sys/class/gpio/gpio0# pwd
/sys/class/gpio/gpio0
root@ATK-IMX6U:/sys/class/gpio/gpio0#
root@ATK-IMX6U:/sys/class/gpio/gpio0# ls
active_low device direction edge power subsystem uevent value
root@ATK-IMX6U:/sys/class/gpio/gpio0#
```

图 16.1.9 gpioX 文件夹

我们主要关心的文件是 `active_low`、`direction`、`edge` 以及 `value` 这四个属性文件，接下来分别介绍这四个属性文件的作用：

- **direction:** 配置 GPIO 引脚为输入或输出模式。该文件可读、可写，读表示查看 GPIO 当前是输入还是输出模式，写表示将 GPIO 配置为输入或输出模式；读取或写入操作可取的值为"out"（输出模式）和"in"（输入模式），如下所示：

```
root@ATK-IMX6U:/sys/class/gpio/gpio0# cat direction
in
root@ATK-IMX6U:/sys/class/gpio/gpio0#
root@ATK-IMX6U:/sys/class/gpio/gpio0# echo "out" > direction
root@ATK-IMX6U:/sys/class/gpio/gpio0# cat direction
out
root@ATK-IMX6U:/sys/class/gpio/gpio0#
```

图 16.1.10 direction 属性文件

- **value:** 在 GPIO 配置为输出模式下，向 value 文件写入"0"控制 GPIO 引脚输出低电平，写入"1"则控制 GPIO 引脚输出高电平。在输入模式下，读取 value 文件获取 GPIO 引脚当前的输入电平状态。譬如：

```
# 获取 GPIO 引脚的输入电平状态
echo "in" > direction
cat value
```

```
# 控制 GPIO 引脚输出高电平
echo "out" > direction
echo "1" > value
```

- **active_low:** 这个属性文件用于控制极性，可读可写，默认情况下为 0，譬如：

```
# active_low 等于 0 时
echo "0" > active_low
echo "out" > direction
echo "1" > value      #输出高
echo "0" > value      #输出低
```

```
# active_low 等于 1 时
$ echo "1" > active_low
$ echo "out" > direction
$ echo "1" > value      #输出低
$ echo "0" > value      #输出高
```

由此看出，active_low 的作用已经非常明显了，对于输入模式来说也同样适用。

- **edge:** 控制中断的触发模式，该文件可读可写。在配置 GPIO 引脚的中断触发模式之前，需将其设置为输入模式：

```
非中断引脚: echo "none" > edge
上升沿触发: echo "rising" > edge
下降沿触发: echo "falling" > edge
边沿触发: echo "both" > edge
```

当引脚被配置为中断后可以使用 poll()函数监听引脚的电平状态变化，在后面的示例中将向大家介绍。

16.2 GPIO 应用编程之输出

上一小节已经向大家介绍了如何通过 sysfs 方式控制开发板上的 GPIO 引脚，本小节我们编写一个简单地测试程序，控制开发板上的某一个 GPIO 输出高、低不同的电平状态，其示例代码如下所示：

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->16_gpio->gpio_out.c](#)。

示例代码 16.2.1 控制 GPIO 输出高低电平

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

static char gpio_path[100];

static int gpio_config(const char *attr, const char *val)
{
    char file_path[100];
    int len;
    int fd;

    sprintf(file_path, "%s/%s", gpio_path, attr);
    if (0 > (fd = open(file_path, O_WRONLY))) {
        perror("open error");
        return fd;
    }

    len = strlen(val);
    if (len != write(fd, val, len)) {
        perror("write error");
        close(fd);
        return -1;
    }

    close(fd); //关闭文件
    return 0;
}

int main(int argc, char *argv[])
{
    /* 校验传参 */
    if (3 != argc) {
        fprintf(stderr, "usage: %s <gpio> <value>\n", argv[0]);
        exit(-1);
    }
}
```

```
/* 判断指定编号的 GPIO 是否导出 */
sprintf(gpio_path, "/sys/class/gpio/gpio%s", argv[1]);

if (access(gpio_path, F_OK)) { //如果目录不存在 则需要导出

    int fd;
    int len;

    if (0 > (fd = open("/sys/class/gpio/export", O_WRONLY))) {
        perror("open error");
        exit(-1);
    }

    len = strlen(argv[1]);
    if (len != write(fd, argv[1], len)) { //导出 gpio
        perror("write error");
        close(fd);
        exit(-1);
    }

    close(fd); //关闭文件
}

/* 配置为输出模式 */
if (gpio_config("direction", "out"))
    exit(-1);

/* 极性设置 */
if (gpio_config("active_low", "0"))
    exit(-1);

/* 控制 GPIO 输出高低电平 */
if (gpio_config("value", argv[2]))
    exit(-1);

/* 退出程序 */
exit(0);
}
```

执行程序时需要传入两个参数, argv[1]指定 GPIO 的编号、argv[2]指定输出电平状态 (0 表示低电平、1 表示高电平)。

上述代码中首先使用 access()函数判断指定编号的 GPIO 引脚是否已经导出,也就是判断相应的 gpioX 目录是否存在,如果不存在则表示未导出,则通过"/sys/class/gpio/export"文件将其导出;导出之后先配置了 GPIO 引脚为输出模式,也就是向 direction 文件中写入"out";接着再配置极性,通过向 active_low 文件中写

入"0"（不用配置也可以）；最后再控制 GPIO 引脚输出相应的电平状态，通过对 value 属性文件写入"1"或"0"来使其输出高电平或低电平。

使用交叉编译工具编译应用程序，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 16.2.1 编译应用程序

16.3 GPIO 应用编程之输入

本小节我们编写一个读取 GPIO 电平状态的测试程序，其示例代码如下所示：

本例程源码对应的路径为：[开发板光盘->11、Linux C 应用编程例程源码->16_gpio->gpio_in.c](#)。

示例代码 16.3.1 读取 GPIO 电平状态

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

static char gpio_path[100];

static int gpio_config(const char *attr, const char *val)
{
    char file_path[100];
    int len;
    int fd;

    sprintf(file_path, "%s/%s", gpio_path, attr);
    if (0 > (fd = open(file_path, O_WRONLY))) {
        perror("open error");
        return fd;
    }

    len = strlen(val);
    if (len != write(fd, val, len)) {
        perror("write error");
        close(fd);
        return -1;
    }
}
```

```
    }

    close(fd); //关闭文件
    return 0;
}

int main(int argc, char *argv[])
{
    char file_path[100];
    char val;
    int fd;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <gpio>\n", argv[0]);
        exit(-1);
    }

    /* 判断指定编号的 GPIO 是否导出 */
    sprintf(gpio_path, "/sys/class/gpio/gpio%s", argv[1]);

    if (access(gpio_path, F_OK)) { //如果目录不存在 则需要导出

        int len;

        if (0 > (fd = open("/sys/class/gpio/export", O_WRONLY))) {
            perror("open error");
            exit(-1);
        }

        len = strlen(argv[1]);
        if (len != write(fd, argv[1], len)) { //导出 gpio
            perror("write error");
            close(fd);
            exit(-1);
        }

        close(fd); //关闭文件
    }

    /* 配置为输入模式 */
    if (gpio_config("direction", "in"))
        exit(-1);
}
```



```
/* 极性设置 */
if (gpio_config("active_low", "0"))
    exit(-1);

/* 配置为非中断方式 */
if (gpio_config("edge", "none"))
    exit(-1);

/* 读取 GPIO 电平状态 */
sprintf(file_path, "%s/%s", gpio_path, "value");

if (0 > (fd = open(file_path, O_RDONLY))) {
    perror("open error");
    exit(-1);
}

if (0 > read(fd, &val, 1)) {
    perror("read error");
    close(fd);
    exit(-1);
}

printf("value: %c\n", val);

/* 退出程序 */
close(fd);
exit(0);
}
```

执行程序时需要传入一个参数, `argv[1]`指定要读取电平状态的 GPIO 对应的编号。

上述代码中首先使用 `access()`函数判断指定编号的 GPIO 引脚是否已经导出, 若未导出, 则通过 `/sys/class/gpio/export`文件将其导出; 导出之后先配置了 GPIO 引脚为输入模式, 也就是向 `direction` 文件中写入 `"in"`; 接着再配置极性、设置 GPIO 引脚为非中断模式 (向 `edge` 属性文件中写入 `"none"`)。

最后打开 `value` 属性文件, 读取 GPIO 的电平状态并将其打印出来。

使用交叉编译工具编译应用程序, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 16.3.1 编译示例代码

16.4 GPIO 应用编程之中断

在应用层可以将 GPIO 配置为中断触发模式, 譬如将 GPIO 配置为上升沿触发、下降沿触发或者边沿触发, 本小节我们来编写一个测试程序, 将 GPIO 配置为边沿触发模式并监测中断触发状态。其示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->16_gpio->gpio_intr.c](#)。

示例代码 16.4.1 监测 GPIO 中断触发

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <poll.h>

static char gpio_path[100];

static int gpio_config(const char *attr, const char *val)
{
    char file_path[100];
    int len;
    int fd;

    sprintf(file_path, "%s/%s", gpio_path, attr);
    if (0 > (fd = open(file_path, O_WRONLY))) {
        perror("open error");
        return fd;
    }

    len = strlen(val);
    if (len != write(fd, val, len)) {
        perror("write error");
        return -1;
    }

    close(fd); //关闭文件
    return 0;
}

int main(int argc, char *argv[])
{
    struct pollfd pfd;
```

```
char file_path[100];
int ret;
char val;

/* 校验传参 */
if (2 != argc) {
    fprintf(stderr, "usage: %s <gpio>\n", argv[0]);
    exit(-1);
}

/* 判断指定编号的 GPIO 是否导出 */
sprintf(gpio_path, "/sys/class/gpio/gpio%s", argv[1]);

if (access(gpio_path, F_OK)) { //如果目录不存在 则需要导出

    int len;
    int fd;

    if (0 > (fd = open("/sys/class/gpio/export", O_WRONLY))) {
        perror("open error");
        exit(-1);
    }

    len = strlen(argv[1]);
    if (len != write(fd, argv[1], len)) { //导出 gpio
        perror("write error");
        exit(-1);
    }

    close(fd); //关闭文件
}

/* 配置为输入模式 */
if (gpio_config("direction", "in"))
    exit(-1);

/* 极性设置 */
if (gpio_config("active_low", "0"))
    exit(-1);

/* 配置中断触发方式: 上升沿和下降沿 */
if (gpio_config("edge", "both"))
    exit(-1);
```

```
/* 打开 value 属性文件 */
sprintf(file_path, "%s/%s", gpio_path, "value");

if (0 > (pfd.fd = open(file_path, O_RDONLY))) {
    perror("open error");
    exit(-1);
}

/* 调用 poll */
pfd.events = POLLPRI; //只关心高优先级数据可读 (中断)

read(pfd.fd, &val, 1); //先读取一次清除状态
for (;;) {

    ret = poll(&pfd, 1, -1); //调用 poll
    if (0 > ret) {
        perror("poll error");
        exit(-1);
    }
    else if (0 == ret) {
        fprintf(stderr, "poll timeout.\n");
        continue;
    }

    /* 校验高优先级数据是否可读 */
    if(pfd.revents & POLLPRI) {
        if (0 > lseek(pfd.fd, 0, SEEK_SET)) { //将读位置移动到头部
            perror("lseek error");
            exit(-1);
        }

        if (0 > read(pfd.fd, &val, 1)) {
            perror("read error");
            exit(-1);
        }

        printf("GPIO 中断触发<value=%c>\n", val);
    }
}

/* 退出程序 */
exit(0);
```

}

执行程序时需要传入一个参数, `argv[1]`指定要读取电平状态的 GPIO 对应的编号。

上述代码中首先使用 `access()`函数判断指定编号的 GPIO 引脚是否已经导出, 若未导出, 则通过 `"/sys/class/gpio/export"`文件将其导出。

对 GPIO 进行配置: 配置为输入模式、配置极性、将触发方式配置为边沿触发。

打开 `value` 属性文件, 获取到文件描述符, 接着使用 `poll()`函数对 `value` 的文件描述符进行监视, 这里为什么要使用 `poll()`监视、而不是直接对文件描述符进行读取操作? 这里简单的描述一下。

13.2.3 小节给大家详细介绍了 `poll()`函数, 这里不再重述! `poll()`函数可以监视一个或多个文件描述符上的 I/O 状态变化, 譬如 `POLLIN`、`POLLOUT`、`POLLERR`、`POLLPRI` 等, 其中 `POLLIN` 和 `POLLOUT` 表示普通优先级数据可读、可写, 而 `POLLPRI` 表示有高优先级数据可读取, 中断就是一种高优先级事件, 当中断触发时表示有高优先级数据可被读取。当然, 除此之外还可使用 13.4 小节所介绍的异步 I/O 方式来监视 GPIO 中断触发。

使用交叉编译工具编译应用程序, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 16.4.1 编译应用程序

16.5 在开发板上测试

前面我们编写了 3 个测试程序, 并编译得到了对应的可执行文件, 本小节一个一个进行测试。在测试之前, 选择一个测试引脚, 这里笔者以板子上的 `GPIO1_IO01` 引脚为例, 该引脚在底板上已经引出, 如下所示:

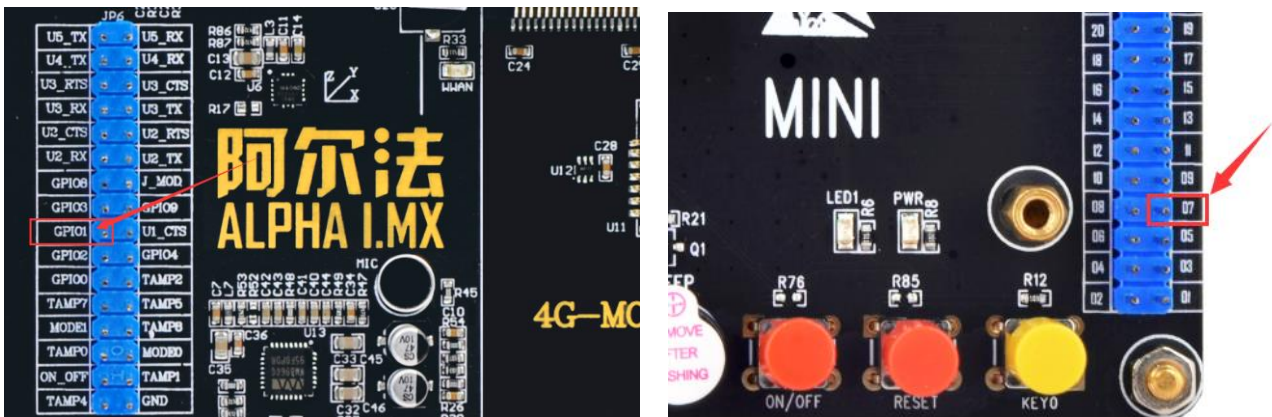


图 16.5.1 GPIO1_IO01 引脚 (左图为 ALPHA 板、右图为 Mini 板)

Mini 开发板可以通过背面丝印标注的名称或原理图进行确认。

16.5.1 GPIO 输出测试

将示例代码 16.2.1 编译的到的可执行文件拷贝到开发板 Linux 系统用户家目录下, 执行该应用程序控制开发板上的 `GPIO1_IO01` 引脚输出高或低电平:

```
./testApp 1 1 #控制 GPIO1_IO01 输出高电平
```

```
./testApp 1 0          #控制 GPIO1_IO01 输出低电平
```

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls  
driver  shell  testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./testApp 1 1  
root@ATK-IMX6U:~# ./testApp 1 0  
root@ATK-IMX6U:~# ./testApp 1 1  
root@ATK-IMX6U:~#
```

图 16.5.2 控制 GPIO 输出的电平状态

执行相应的命令后, 可以使用万用表或者连接一个 LED 小灯进行检验, 以验证实验结果!

16.5.2 GPIO 输入测试

将示例代码 16.3.1 编译的到的可执行文件拷贝到开发板 Linux 系统用户家目录下, 执行该应用程序以读取 GPIO1_IO01 引脚此时的电平状态, 是高电平还是低电平?

首先通过杜邦线将 GPIO1_IO01 引脚连接到板子上的 3.3V 电源引脚上, 接着执行命令读取 GPIO 电平状态:

```
root@ATK-IMX6U:~# ls  
driver  shell  testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./testApp 1  
value: 1  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#
```

图 16.5.3 高电平读取

打印出的 value 等于 1, 表示读取到 GPIO 的电平确实是高电平; 接着将 GPIO1_IO01 引脚连接到板子上的 GND 引脚上, 执行命令:

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls  
driver  shell  testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./testApp 1  
value: 0  
root@ATK-IMX6U:~#
```

图 16.5.4 低电平读取

打印出的 value 等于 0, 表示读取到 GPIO 的电平确实是低电平; 测试结果与实际相符合!

16.5.3 GPIO 中断测试

将示例代码 16.4.1 编译的到的可执行文件拷贝到开发板 Linux 系统用户家目录下, 执行该应用程序可以监测 GPIO 的中断触发。

执行应用程序监测 GPIO1_IO01 引脚的中断触发情况, 如下所示:

```
./testApp 1          # 监测 GPIO1_IO01 引脚中断触发
```

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./testApp 1  
GPIO中断触发<value=0>  
GPIO中断触发<value=1>  
GPIO中断触发<value=0>  
GPIO中断触发<value=1>  
GPIO中断触发<value=0>  
GPIO中断触发<value=1>  
GPIO中断触发<value=0>  
GPIO中断触发<value=1>
```

图 16.5.5 中断触发测试结果

当执行命令之后,我们可以使用杜邦线将 GPIO1_IO01 引脚连接到 GND 或 3.3V 电源引脚上,来回切换,使得 GPIO1_IO01 引脚的电平状态发生由高到低或由低到高的状态变化,以验证 GPIO 中断的边沿触发情况;当发生中断时,终端将会打印相应的信息,如上图所示。

Tips: 测试完成后按 Ctrl+C 退出程序!

第十七章 输入设备应用编程

本章学习输入设备的应用编程,首先要知道什么是输入设备?输入设备其实就是能够产生输入事件的设备就称为输入设备,常见的输入设备包括鼠标、键盘、触摸屏、按钮等等,它们都能够产生输入事件,产生输入数据给计算机系统。

对于输入设备的应用编程其主要是获取输入设备上报的数据、输入设备当前状态等,譬如获取触摸屏当前触摸点的 X、Y 轴位置信息以及触摸屏当前处于按下还是松开状态。

本章将会讨论如下主题内容。

- 什么是输入设备;
- 如何读取输入设备的数据;
- 如何解析从输入设备中获取到的数据;
- 按键、触摸屏设备如何解析数据、应用编程。

17.1 输入类设备编程介绍

17.1.1 什么是输入设备

先来了解什么是输入设备（也称为 input 设备），常见的输入设备有鼠标、键盘、触摸屏、遥控器、电脑画图板等，用户通过输入设备与系统进行交互。

17.1.2 input 子系统

由上面的介绍可知，输入设备种类非常多，每种设备上报的数据类型又不一样，那么 Linux 系统如何管理呢？Linux 系统为了统一管理这些输入设备，实现了一套能够兼容所有输入设备的框架，那么这个框架就是 input 子系统。驱动开发人员基于 input 子系统开发输入设备的驱动程序，input 子系统可以屏蔽硬件的差异，向应用层提供一套统一的接口。

基于 input 子系统注册成功的输入设备，都会在 `/dev/input` 目录下生成对应的设备节点（设备文件），设备节点名称通常为 `eventX`（X 表示一个数字编号 0、1、2、3 等），譬如 `/dev/input/event0`、`/dev/input/event1`、`/dev/input/event2` 等，通过读取这些设备节点可以获取输入设备上报的数据。

17.1.3 读取数据的流程

如果我们要读取触摸屏的数据，假设触摸屏设备对应的设备节点为 `/dev/input/event0`，那么数据读取流程如下：

- ①、应用程序打开 `/dev/input/event0` 设备文件；
- ②、应用程序发起读操作（譬如调用 `read`），如果没有数据可读则会进入休眠（阻塞 I/O 情况下）；
- ③、当有数据可读时，应用程序会被唤醒，读操作获取到数据返回；
- ④、应用程序对读取到的数据进行解析。

当无数据可读时，程序会进入休眠状态（也就是阻塞），譬如应用程序读触摸屏数据，如果当前并没有去触碰触摸屏，自然是无数据可读；当我们用手指触摸触摸屏或者在屏上滑动时，此时就会产生触摸数据、应用程序就有数据可读了，应用程序会被唤醒，成功读取到数据。那么对于其它输入设备亦是如此，无数据可读时应用程序会进入休眠状态（阻塞式 I/O 方式下），当有数据可读时才会被唤醒。

17.1.4 应用程序如何解析数据

首先我们要知道，应用程序打开输入设备对应的设备文件，向其发起读操作，那么这个读操作获取到的是什么样的数据呢？其实每一次 `read` 操作获取的都是一个 `struct input_event` 结构体类型数据，该结构体定义在 `<linux/input.h>` 头文件中，它的定义如下：

示例代码 17.1.1 `struct input_event` 结构体

```
struct input_event {  
    struct timeval time;  
    __u16 type;  
    __u16 code;  
    __s32 value;  
};
```

结构体中的 `time` 成员变量是一个 `struct timeval` 类型的变量，该结构体在前面给大家介绍过，内核会记录每个上报的事件其发生的时间，并通过变量 `time` 返回给应用程序。时间参数通常不是那么重要，而其它 3 个成员变量 `type`、`code`、`value` 更为重要。

- **type:** type 用于描述发生了哪一种类型的事件（对事件的分类），Linux 系统所支持的输入事件类型如下所示：

```

/*
 * Event types
 */

#define EV_SYN          0x00      //同步类事件，用于同步事件
#define EV_KEY          0x01      //按键类事件
#define EV_REL          0x02      //相对位移类事件(譬如鼠标)
#define EV_ABS          0x03      //绝对位移类事件(譬如触摸屏)
#define EV_MSC          0x04      //其它杂类事件
#define EV_SW           0x05
#define EV_LED          0x11
#define EV_SND          0x12
#define EV_REP          0x14
#define EV_FF           0x15
#define EV_PWR          0x16
#define EV_FF_STATUS    0x17
#define EV_MAX          0x1f
#define EV_CNT          (EV_MAX+1)

```

以上这些宏定义也是在<linux/input.h>头文件中，所以在应用程序中需要包含该头文件：一种输入设备通常可以产生多种不同类型的事件，譬如点击鼠标按键（左键、右键，或鼠标上的其它按键）时会上报按键类事件，移动鼠标时则会上报相对位移类事件。

- **code:** code 表示该类事件中的哪一个具体事件，以上列举的每一种事件类型中都包含了一系列具体事件，譬如一个键盘上通常有很多按键，譬如字母 A、B、C、D 或者数字 1、2、3、4 等，而 code 变量则告知应用程序是哪一个按键发生了输入事件。每一种事件类型都包含多种不同的事件，譬如按键类事件：

```

#define KEY_RESERVED    0
#define KEY_ESC         1          //ESC 键
#define KEY_1           2          //数字 1 键
#define KEY_2           3          //数字 2 键
#define KEY_TAB         15         //TAB 键
#define KEY_Q           16         //字母 Q 键
#define KEY_W           17         //字母 W 键
#define KEY_E           18         //字母 E 键
#define KEY_R           19         //字母 R 键
.....

```

相对位移事件

```

#define REL_X           0x00      //X 轴
#define REL_Y           0x01      //Y 轴
#define REL_Z           0x02      //Z 轴
#define REL_RX          0x03
#define REL_RY          0x04

```

```
#define REL_RZ          0x05
#define REL_HWHEEL     0x06
#define REL_DIAL       0x07
#define REL_WHEEL     0x08
#define REL_MISC      0x09
#define REL_MAX       0x0f
#define REL_CNT       (REL_MAX+1)
```

绝对位移事件

触摸屏设备是一种绝对位移设备,它能够产生绝对位移事件;譬如对于触摸屏来说,一个触摸点所包含的信息可能有多种,譬如触摸点的 X 轴坐标、Y 轴坐标、Z 轴坐标、按压力大小以及接触面积等,所以 code 变量告知应用程序当前上报的是触摸点的哪一种信息(X 坐标还是 Y 坐标、亦或者其它);绝对位移事件如下:

```
#define ABS_X          0x00      //X 轴
#define ABS_Y          0x01      //Y 轴
#define ABS_Z          0x02      //Z 轴
#define ABS_RX        0x03
#define ABS_RY        0x04
#define ABS_RZ        0x05
#define ABS_THROTTLE   0x06
#define ABS_RUDDER     0x07
#define ABS_WHEEL     0x08
#define ABS_GAS       0x09
#define ABS_BRAKE     0x0a
#define ABS_HAT0X     0x10
#define ABS_HAT0Y     0x11
#define ABS_HAT1X     0x12
#define ABS_HAT1Y     0x13
#define ABS_HAT2X     0x14
#define ABS_HAT2Y     0x15
#define ABS_HAT3X     0x16
#define ABS_HAT3Y     0x17
#define ABS_PRESSURE   0x18
#define ABS_DISTANCE   0x19
#define ABS_TILT_X    0x1a
#define ABS_TILT_Y    0x1b
#define ABS_TOOL_WIDTH 0x1c
.....
```

除了以上列举出来的之外,还有很多,大家可以自己浏览<linux/input.h>头文件(这些宏其实是定义在 input-event-codes.h 头文件中,该头文件被<linux/input.h>所包含了),关于这些具体的事件,后面再给大家进行介绍。

- **value:** 内核每次上报事件都会向应用层发送一个数据 value,对 value 值的解释随着 code 的变化而变化。譬如对于按键事件(type=1)来说,如果 code=2(键盘上的数字键 1,也就是 KEY_1),那么如果 value 等于 1,则表示 KEY_1 键按下;value 等于 0 表示 KEY_1 键松开,如果 value 等于 2

则表示 KEY_1 键长按。再比如,在绝对位移事件中(`type=3`),如果 `code=0`(触摸点 X 坐标 ABS_X),那么 `value` 值就等于触摸点的 X 轴坐标值;同理,如果 `code=1`(触摸点 Y 坐标 ABS_Y),此时 `value` 值便等于触摸点的 Y 轴坐标值;所以对 `value` 值的解释需要根据不同的 `code` 值而定!

数据同步

上面我们提到了同步事件类型 `EV_SYN`,同步事件用于实现同步操作、告知接收者本轮上报的数据已经完整。应用程序读取输入设备上报的数据时,一次 `read` 操作只能读取一个 `struct input_event` 类型数据,譬如对于触摸屏来说,一个触摸点的信息包含了 X 坐标、Y 坐标以及其它信息,对于这样情况,应用程序需要执行多次 `read` 操作才能把一个触摸点的信息全部读取出来,这样才能得到触摸点的完整信息。

那么应用程序如何得知本轮已经读取到完整的数据了呢?其实这就是通过同步事件来实现的,内核将本轮需要上报、发送给接收者的数据全部上报完毕后,接着会上报一个同步事件,以告知应用程序本轮数据已经完整、可以进行同步了。

同步类事件中也包含了多种不同的事件,如下所示:

```
/*
 * Synchronization events.
 */

#define SYN_REPORT      0
#define SYN_CONFIG      1
#define SYN_MT_REPORT   2
#define SYN_DROPPED     3
#define SYN_MAX         0xf
#define SYN_CNT         (SYN_MAX+1)
```

所以的输入设备都需要上报同步事件,上报的同步事件通常是 `SYN_REPORT`,而 `value` 值通常为 0。

17.2 读取 `struct input_event` 数据

根据前面的介绍可知,对输入设备调用 `read()` 会读取到一个 `struct input_event` 类型数据,本小节编写一个简单地应用程序,将读取到的 `struct input_event` 类型数据中的每一个元素打印出来、并对它们进行解析。

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->17_input->read_input.c](#)。

示例代码 17.2.1 读取 `struct input_event` 类型数据

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/input.h>

int main(int argc, char *argv[])
{
    struct input_event in_ev = {0};
    int fd = -1;
```

```
/* 校验传参 */
if (2 != argc) {
    fprintf(stderr, "usage: %s <input-dev>\n", argv[0]);
    exit(-1);
}

/* 打开文件 */
if (0 > (fd = open(argv[1], O_RDONLY))) {
    perror("open error");
    exit(-1);
}

for (;;) {

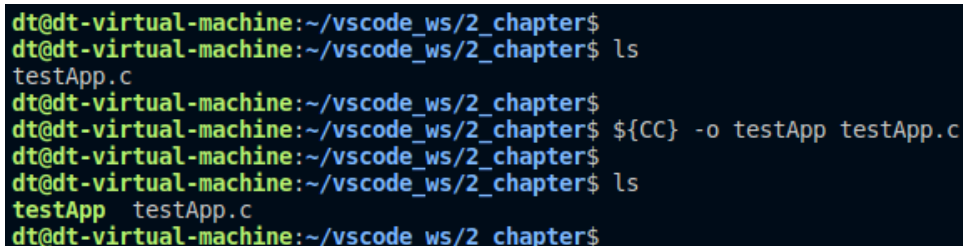
    /* 循环读取数据 */
    if (sizeof(struct input_event) !=
        read(fd, &in_ev, sizeof(struct input_event))) {
        perror("read error");
        exit(-1);
    }

    printf("type:%d code:%d value:%d\n",
           in_ev.type, in_ev.code, in_ev.value);
}
}
```

执行程序时需要传入参数，这个参数就是对应的输入设备的设备节点（设备文件），程序中会对传参进行校验。程序中首先调用 `open()` 函数打开设备文件，之后在 `for` 循环中调用 `read()` 函数读取文件，将读取到的数据存放在 `struct input_event` 结构体对象中，之后将结构体对象中的各个成员变量打印出来。注意，程序中使用了阻塞式 I/O 方式读取设备文件，所以当无数据可读时 `read` 调用会被阻塞，知道有数据可读时才会被唤醒！

Tips: 设备文件不同于普通文件，读写设备文件之前无需设置读写位置偏移量。

使用交叉编译工具编译上述代码得到可执行文件 `testApp`:



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 17.2.1 编译示例代码

17.3 在开发板上验证

ALPHA 和 Mini 开发板上都有一个用户按键 `KEY0`，它就是一个典型的输入设备，如下图所示：

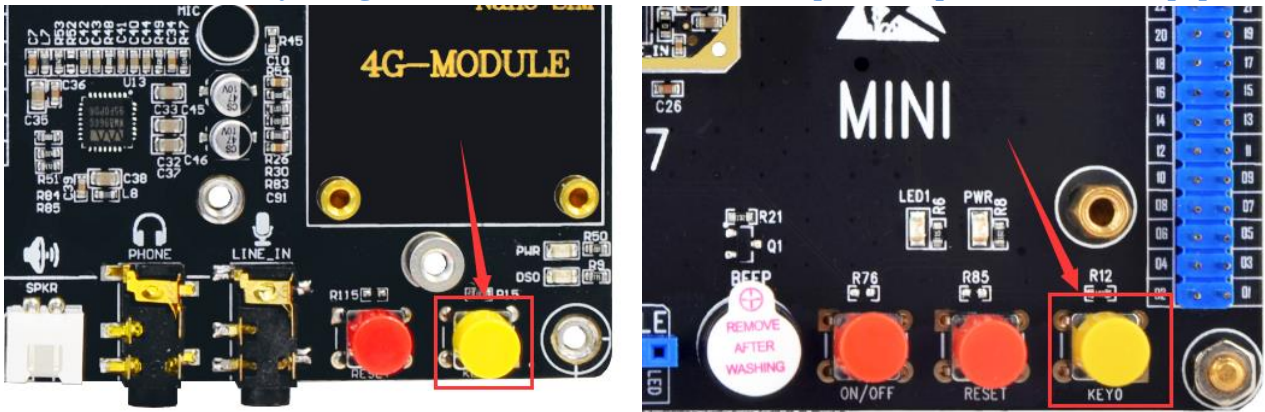


图 17.3.1 用户按键 KEY0 (左图为 ALPHA、右图为 Mini)

该按键是提供给用户使用的一个 GPIO 按键，在出厂系统中，该按键驱动基于 input 子系统而实现，所以在/dev/input 目录下存在 KEY0 的设备节点，具体是哪个设备节点，可以使用 13.1.1 小节介绍的方法进行判断，这里不再重述！也可以通过查看/proc/bus/input/devices 文件得知，查看该文件可以获取到系统中注册的所有输入设备相关的信息，如下所示：

```

root@ATK-IMX6U:~# cat /proc/bus/input/devices
I: Bus=0019 Vendor=0000 Product=0000 Version=0000
N: Name="20cc000.snvs:snvs-powerkey" 设备名字
P: Phys=snvs-pwrkey/input0
S: Sysfs=/devices/platform/soc/2000000.aips-bus/20cc000.snvs/20cc000.snvs:snvs-powerkey/input/input0
U: Uniq=
H: Handlers=kbd event0 设备节点event0
B: PROP=0
B: EV=3
B: KEY=100000 0 0 0

I: Bus=0018 Vendor=0000 Product=0000 Version=0000
N: Name="EP0820M09" 设备名字      触摸屏
P: Phys=
S: Sysfs=/devices/platform/soc/2100000.aips-bus/21a4000.i2c/i2c-1/1-0038/input/input1
U: Uniq=
H: Handlers=mouse0 event1 设备节点event1
B: PROP=0
B: EV=b
B: KEY=400 0 0 0 0 0 0 0 0 0
B: ABS=2608000 3

I: Bus=0019 Vendor=0001 Product=0001 Version=0100
N: Name="gpio_keys@0" 设备名字      GPIO按键KEY0
P: Phys=gpio-keys/input0
S: Sysfs=/devices/platform/gpio_keys@0/input/input2
U: Uniq=
H: Handlers=kbd event2 设备节点event2
B: PROP=0
B: EV=100003
B: KEY=40000 0 0 0

root@ATK-IMX6U:~#

```

图 17.3.2 查看/proc/bus/input/devices 文件

接下来我们使用这个按键进行测试，执行下面的命令：

```

root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp /dev/input/event2
type:1 code:114 value:1
type:0 code:0 value:0
type:1 code:114 value:0
type:0 code:0 value:0
type:1 code:114 value:1
type:0 code:0 value:0
type:1 code:114 value:0
type:0 code:0 value:0
type:1 code:114 value:1
type:0 code:0 value:0
type:1 code:114 value:0
type:0 code:0 value:0

```

图 17.3.3 测试

程序运行后, 执行按下 KEY0、松开 KEY0 等操作, 终端将会打印出相应的信息, 如上图所示。

第一行中 type 等于 1, 表示上报的是按键事件 EV_KEY, code=114, 打开 input-event-codes.h 头文件进行查找, 可以发现 cpde=114 对应的是键盘上的 KEY_VOLUMEDOWN 按键, 这个是 ALPHA/Mini 开发板出厂系统已经配置好的。而 value=1 表示按键按下, 所以整个第一行的意思就是按键 KEY_VOLUMEDOWN 被按下。

第二行, 表示上报了 EV_SYN 同步类事件 (type=0) 中的 SYN_REPORT 事件 (code=0), 表示本轮数据已经完整、报告同步。

第三行, type 等于 1, 表示按键类事件, code 等于 114, value 等于 0, 所以表示按键 KEY_VOLUMEDOWN 被松开。

第四行, 又上报了同步事件。

所以整个上面 4 行的打印信息就是开发板上的 KEY0 按键被按下以及松开这个过程, 内核所上报的事件以及发送给应用层的数据 value。

我们试试长按按键 KEY0, 按住不放, 如下所示:

```

type:1 code:114 value:1
type:0 code:0 value:0
type:1 code:114 value:0
type:0 code:0 value:0
type:1 code:114 value:1
type:0 code:0 value:0
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1
type:1 code:114 value:2
type:0 code:0 value:1

```

图 17.3.4 长按按键 KEY0

可以看到上报按键事件时, 对应的 value 等于 2, 表示长按状态。

17.4 按键应用编程

本小节编写一个应用程序, 获取按键状态, 判断按键当前是按下、松开或长按状态。从上面打印的信息可知, 对于按键来说, 它的事件上报流程如下所示:

以字母 A 键为例

KEY_A //上报 KEY_A 事件

SYN_REPORT //同步

如果是按下, 则上报 KEY_A 事件时, value=1; 如果是松开, 则 value=0; 如果是长按, 则 value=2。
接下来编写按键应用程序, 读取按键状态并将结果打印出来, 代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->17_input->read_key.c](#)。

示例代码 17.4.1 按键应用编程

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/input.h>

int main(int argc, char *argv[])
{
    struct input_event in_ev = {0};
    int fd = -1;
    int value = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <input-dev>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    if (0 > (fd = open(argv[1], O_RDONLY))) {
        perror("open error");
        exit(-1);
    }

    for (;;) {

        /* 循环读取数据 */
        if (sizeof(struct input_event) !=
            read(fd, &in_ev, sizeof(struct input_event))) {
            perror("read error");
            exit(-1);
        }

        if (EV_KEY == in_ev.type) { //按键事件
```



```

switch (in_ev.value) {
case 0:
    printf("code<%d>: 松开\n", in_ev.code);
    break;
case 1:
    printf("code<%d>: 按下\n", in_ev.code);
    break;
case 2:
    printf("code<%d>: 长按\n", in_ev.code);
    break;
}
}
}
}

```

在 for 循环中, 调用 read() 读取输入设备上报的数据, 当按键按下或松开 (以及长按) 动作发生时, read() 会读取到输入设备上报的数据, 首先判断此次上报的事件是否是按键类事件 (EV_KEY), 如果是按键类事件、接着根据 value 值来判断按键当前的状态是松开、按下还是长按。

将上述代码进行编译:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 17.4.1 编译示例代码

将编译得到的可执行文件拷贝开发板 Linux 系统的 /home/root 目录下。

首先我们来测试开发板的 KEY0 按键, 执行应用程序:

`./testApp /dev/input/event2` # 测试开发板上的 KEY0

```

root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp /dev/input/event2
code<114>: 按下
code<114>: 松开
code<114>: 按下
code<114>: 松开
code<114>: 按下
code<114>: 长按
code<114>: 长按
code<114>: 长按
code<114>: 长按
code<114>: 长按
code<114>: 松开
code<114>: 按下
code<114>: 松开

```

图 17.4.2 测试 KEY0

运行程序之后, 按下 KEY0 或松开 KEY0 以及长按情况下, 终端会打印出相应的信息, 如上图所示。code=114 (KEY_VOLUMEDOWN 按键)。

除了测试开发板上的 KEY0 按键之外,我们还可以测试键盘上的按键,首先找到一个 USB 键盘连接到开发板的 USB HOST 接口上,当键盘插入之后,终端将会打印出相应的驱动加载信息:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# [ 583.472627] usb 1-1: new low-speed USB device number 2 using ci_hdrc
[ 583.652677] input: Logitech USB Keyboard as /devices/platform/soc/2100000.aips-bus/2184200.usb/ci_hdrc.1/usb1/1-1/1-1:1.0/0003:046D:C31C.0001/input/input3
[ 583.743088] hid-generic 0003:046D:C31C.0001: input: USB HID v1.10 Keyboard [Logitech USB Keyboard] on usb-ci_hdrc.1-1/input0
[ 583.771627] input: Logitech USB Keyboard as /devices/platform/soc/2100000.aips-bus/2184200.usb/ci_hdrc.1/usb1/1-1/1-1:1.1/0003:046D:C31C.0002/input/input4
[ 583.843074] hid-generic 0003:046D:C31C.0002: input: USB HID v1.10 Device [Logitech USB Keyboard] on usb-ci_hdrc.1-1/input1
root@ATK-IMX6U:~#
```

图 17.4.3 插入 USB 键盘终端打印信息

驱动加载成功之后,可以查看下该键盘设备对应的设备节点,使用命令"cat /proc/bus/input/devices",在打印信息中找到键盘设备的信息:

```
I: Bus=0003 Vendor=046d Product=c31c Version=0110
N: Name="Logitech USB Keyboard"
P: Phys=usb-ci_hdrc.1-1/input0
S: Sysfs=/devices/platform/soc/2100000.aips-bus/2184200.usb/ci_hdrc.1/usb1/1-1/1-1:1.0/0003:046D:C31C.0001/input/input3
U: Uniq=
H: Handlers=sysrq kbd event3
B: PROP=0
B: EV=120013
B: KEY=10000 7 ff9f207a c14057ff febeffdf ffefffff ffffffff ffffffff
B: MSC=10
B: LED=1f
```

图 17.4.4 USB 键盘设备信息

譬如笔者使用的是一个罗技的 USB 键盘"Logitech USB Keyboard",对应的设备节点为/dev/input/event3,运行测试程序并按下、松开键盘上的按键:

```
root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp /dev/input/event3
code<30>: 按下 A键按下、松开
code<30>: 松开
code<48>: 按下 B键按下、松开
code<48>: 松开
code<46>: 按下 C键按下、松开
code<46>: 松开
code<32>: 按下 D键按下、松开
code<32>: 松开
code<28>: 按下 回车键按下、松开
code<28>: 松开
code<57>: 按下 空格键按下、松开
code<57>: 松开
```

图 17.4.5 测试 USB 键盘

大家可以根据 code 值查询到对应的按键(通过 input-event-codes.h 头文件),譬如 code=30 对应的是键盘上的字母 A 键,code=48 对应的是字母 B 键。

17.5 触摸屏应用编程

本小节编写触摸屏应用程序,获取触摸屏的坐标信息并将其打印出来。

17.5.1 解析触摸屏设备上报的数据

触摸屏设备是一个绝对位移设备,可以上报绝对位移事件,绝对位移事件如下:

```
#define ABS_X      0x00    //X 轴坐标
#define ABS_Y      0x01    //Y 轴坐标
#define ABS_Z      0x02    //Z 轴坐标
#define ABS_RX     0x03
#define ABS_RY     0x04
```

```

#define ABS_RZ          0x05
#define ABS_THROTTLE    0x06
#define ABS_RUDDER      0x07
#define ABS_WHEEL       0x08
#define ABS_GAS         0x09
#define ABS_BRAKE       0x0a
#define ABS_HAT0X       0x10
#define ABS_HAT0Y       0x11
#define ABS_HAT1X       0x12
#define ABS_HAT1Y       0x13
#define ABS_HAT2X       0x14
#define ABS_HAT2Y       0x15
#define ABS_HAT3X       0x16
#define ABS_HAT3Y       0x17
#define ABS_PRESSURE    0x18    //按压力
#define ABS_DISTANCE    0x19
#define ABS_TILT_X      0x1a
#define ABS_TILT_Y      0x1b
#define ABS_TOOL_WIDTH  0x1c

#define ABS_VOLUME      0x20

#define ABS_MISC        0x28

#define ABS_MT_SLOT      0x2f /* MT slot being modified */
#define ABS_MT_TOUCH_MAJOR 0x30 /* Major axis of touching ellipse */
#define ABS_MT_TOUCH_MINOR 0x31 /* Minor axis (omit if circular) */
#define ABS_MT_WIDTH_MAJOR 0x32 /* Major axis of approaching ellipse */
#define ABS_MT_WIDTH_MINOR 0x33 /* Minor axis (omit if circular) */
#define ABS_MT_ORIENTATION 0x34 /* Ellipse orientation */
#define ABS_MT_POSITION_X 0x35 /* Center X touch position */ //X 轴坐标
#define ABS_MT_POSITION_Y 0x36 /* Center Y touch position */ //Y 轴坐标
#define ABS_MT_TOOL_TYPE 0x37 /* Type of touching device */
#define ABS_MT_BLOB_ID 0x38 /* Group a set of packets as a blob */
#define ABS_MT_TRACKING_ID 0x39 /* Unique ID of initiated contact */ ID
#define ABS_MT_PRESSURE 0x3a /* Pressure on contact area */ //按压力
#define ABS_MT_DISTANCE 0x3b /* Contact hover distance */
#define ABS_MT_TOOL_X 0x3c /* Center X tool position */
#define ABS_MT_TOOL_Y 0x3d /* Center Y tool position */

#define ABS_MAX          0x3f
#define ABS_CNT          (ABS_MAX+1)

```

单点触摸和多点触摸

触摸屏分为多点触摸设备和单点触摸设备。单点触摸设备只支持单点触摸，一轮（笔者把一个同步事件称为一轮）完整的数据只包含一个触摸点信息；单点触摸设备以 ABS_XXX 事件承载、上报触摸点的信息，譬如 ABS_X（value 值对应的是 X 轴坐标值）、ABS_Y（value 值对应的是 Y 轴坐标值）等绝对位移事件，而有些设备可能还支持 Z 轴坐标（通过 ABS_Z 事件上报、value 值对应的便是 Z 轴坐标值）、按压力大小（通过 ABS_PRESSURE 事件上报、value 值对应的便是按压力大小）以及接触面积等属性。大部分的单点触摸设备都会上报 ABS_X 和 ABS_Y 事件，而其它绝对位移事件则根据具体的设备以及驱动的实现而定！

多点触摸设备可支持多点触摸，譬如 ALPHA/Mini 开发板配套使用的 4.3 寸、7 寸等触摸屏均支持多点触摸，对于多点触摸设备，一轮完整的数据可能包含多个触摸点信息。多点触摸设备则是以 ABS_MT_XXX（MT 是 Multi-touch，意思为：多点触摸）事件承载、上报触摸点的信息，如 ABS_MT_POSITION_X（X 轴坐标）、ABS_MT_POSITION_Y（Y 轴坐标）等绝对位移事件。

触摸屏设备除了上报绝对位移事件之外，还可以上报按键类事件和同步类事件。同步事件很好理解，因为几乎每一个输入设备都会上报同步事件、告知应用层本轮数据是否完整；当手指点击触摸屏或手指从触摸屏离开时，此时就会上报按键类事件，用于描述按下触摸屏和松开触摸屏；具体的按键事件为 BTN_TOUCH（code=0x14a，也就是 330），当然，手指在触摸屏上滑动不会上报 BTN_TOUCH 事件。

Tips: BTN_TOUCH 事件不支持长按状态，故其 value 不会等于 2。对于多点触摸设备来说，只有第一个点按下时上报 BTN_TOUCH 事件 value=1、当最后一个点离开触摸屏时上报 BTN_TOUCH 事件 value=0。

单点触摸设备--事件上报的顺序

通过上面的介绍可知，单点触摸设备事件上报的流程大概如下所示：

```
# 点击触摸屏时
BTN_TOUCH
ABS_X
ABS_Y
SYN_REPORT

# 滑动
ABS_X
ABS_Y
SYN_REPORT

# 松开
BTN_TOUCH
SYN_REPORT
```

以上列举出只是一个大致流程，实际上对于不同的触摸屏设备，能够获取到的信息量大小是不相同的，譬如某设备只能读取到触摸点的 X 和 Y 坐标、而另一设备却能读取 X、Y 坐标以及按压力大小、触点面积等信息，总之这些数据都会在 SYN_REPORT 同步事件之前上报给应用层。

当手指点击触摸屏时，首先上报 BTN_TOUCH 事件，此时 value=1，表示按下；接着上报 ABS_X、ABS_Y 事件将 X、Y 轴坐标数据发送给应用层；数据上报完成接着上报一个同步事件 SYN_REPORT，表示此次触摸点信息已经完整。

当手指在触摸屏上滑动时，并不会上报 BTN_TOUCH 事件，因为滑动过程并未发生按下、松开这种动作。

当松开时, 首先上报了 `BTN_TOUCH` 事件, 此时 `value=0`, 表示手指已经松开了触摸屏, 接着上报一个同步事件 `SYN_REPORT`。

以上就是单点触摸设备事件上报的一个大致流程, 接下来看看多点触摸设备。

多点触摸设备--事件上报的顺序

多点触摸设备上报的一轮完整数据中可能包含多个触摸点的信息, 譬如 5 点触摸设备, 如果 5 个手指同时在触摸屏上滑动, 那么硬件就会更新 5 个触摸点的信息, 内核需要把这 5 个触摸点的信息上报给应用层。

在 Linux 内核中, 多点触摸设备使用多点触摸 (MT) 协议上报各个触摸点的数据, MT 协议分为两种类型: Type A 和 Type B, Type A 协议实际使用中用的比较少, 几乎处于淘汰的边缘, 这里就不再给大家介绍了, 我们重点来看看 Type B 协议。

MT 协议之 Type B 协议

Type B 协议适用于能够追踪并区分触摸点的设备, 开发板配套使用的触摸屏都属于这类设备。Type B 协议的重点是通过 `ABS_MT_SLOT` 事件上报各个触摸点信息的更新!

能够追踪并区分触摸点的设备通常在硬件上能够区分不同的触摸点, 譬如对于一个 5 点触摸设备来说, 硬件能够为每一个识别到的触摸点与一个 slot 进行关联, 这个 slot 就是一个编号, 触摸点 0、触摸点 1、触摸点 2 等。底层驱动向应用层上报 `ABS_MT_SLOT` 事件, 此事件会告诉接收者当前正在更新的是哪个触摸点的数据, `ABS_MT_SLOT` 事件中对应的 `value` 数据存放的便是一个 slot、以告知应用层当前正在更新 slot 关联的触摸点对应的信息。

每个识别出来的触摸点分配一个 slot, 与该 slot 关联起来, 利用这个 slot 来传递对应触点的变化。除了 `ABS_MT_SLOT` 事件之外, Type B 协议还会使用到 `ABS_MT_TRACKING_ID` 事件, `ABS_MT_TRACKING_ID` 事件则用于触摸点的创建、替换和销毁工作, `ABS_MT_TRACKING_ID` 事件携带的数据 `value` 表示一个 ID, 一个非负数的 ID ($ID \geq 0$) 表示一个有效的触摸点, 如果 ID 等于 -1 表示该触摸点已经不存在、被移除了; 一个以前不存在的 ID 表示这是一个新的触摸点。

Type B 协议可以减少发送到用户空间的数据, 只有发生了变更的数据才会上报, 譬如某个触摸点发生了移动, 但仅仅只改变了 X 轴坐标、而未改变 Y 轴坐标, 那么内核只会将改变后的 X 坐标值通过 `ABS_MT_POSITION_X` 事件发送给应用层。

以上关于 Type B 协议就给大家介绍这么多, 为了帮助大家理解, 笔者把 Type B 协议下多点触摸设备上报数据的流程列举如下:

```
ABS_MT_SLOT 0
ABS_MT_TRACKING_ID 10
ABS_MT_POSITION_X
ABS_MT_POSITION_Y
ABS_MT_SLOT 1
ABS_MT_TRACKING_ID 11
ABS_MT_POSITION_X
ABS_MT_POSITION_Y
SYN_REPORT
```

单看这个可能大家看不懂, 接下来我们打印触摸屏的数据一个一个进行分析。

Tips: 大家可能会对 slot 和 ID 这两个概念有点混乱, 这里笔者将自己的理解告知大家; slot 是硬件上的一个概念、而 ID 则可认为是软件上的一个概念; 对于一个多点触摸设备来说, 它最大支持的触摸点数是确定的, 譬如 5 个触摸设备, 最多支持 5 个触摸点; 每一个触摸点在硬件上它有一个区分的编号, 譬如触摸点 0、触摸点 1、触摸点 2 等, 这个编号就是一个 slot (通常从 0 开始); 如何给识别到的触点分配一个 slot 呢

(触点与 slot 关联)? 通常是按照时间先后顺序来的, 譬如第一根手指先触碰到触摸屏, 那第一根手指就对应触摸点 0 (slot=0), 接着第二根手指触碰到触摸屏则对应触摸点 1 (slot=1) 以此类推! 这个通常是硬件所支持的。

而 ID 可认为是软件上的一个概念, 它也用于区分不同的触摸点, 但是它跟 slot 不同, 不是同一层级的概念; 举个例子, 譬如一根手指触碰到触摸屏之后拿开, 然后再次触碰到触摸屏, 这个过程中, 假设只有这一根手指进行触碰操作, 那么两次触应对应都是触摸点 0 (slot=0), 这个无疑义! 但从触摸点的生命周期来看, 它们是同一个触摸点吗? 答案肯定不是, 为啥呢? 手指从触摸屏上离开后, 该触摸点就消失了、被删除了, 该触摸点的生命周期也就到此结束了, 所以它们自然是不同的触摸点, 所以它们的 ID 是不同的。

触摸屏上报数据分析

首先在测试触摸屏之前, 需要保证开发板上已经连接了 LCD 屏, ALPHA/Mini I.MX6U 开发板出厂系统配套支持多种不同分辨率的 LCD 屏, 包括 4.3 寸 480*272、4.3 寸 800*480、7 寸 800*480、7 寸 1024*600 以及 10.1 寸 1280*800, 在启动开发板之前需要将 LCD 屏通过软排线连接到开发板的 LCD 接口, 开发板连接好 LCD 屏之后上电启动开发板、运行出厂系统。

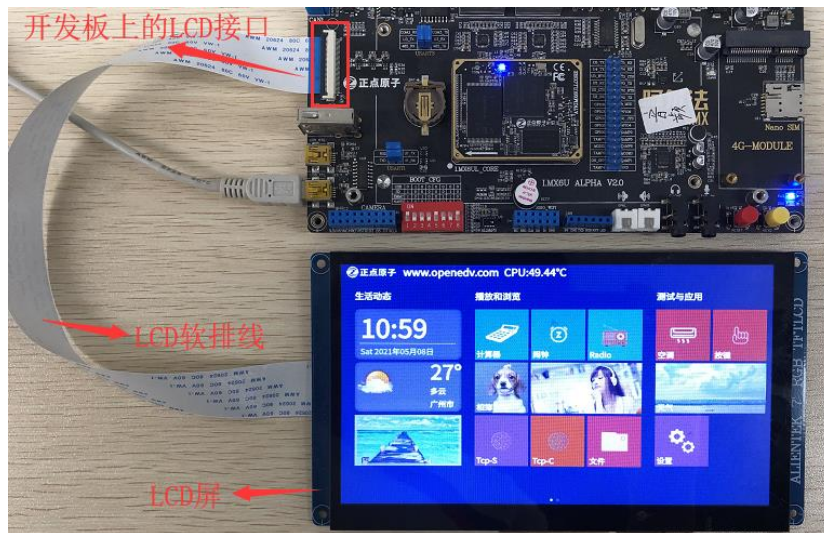


图 17.5.1 连接 LCD 屏 (ALPHA 板)

触摸屏与 LCD 液晶屏面板是粘合在一起的, 也就是说触摸屏是直接贴在了 LCD 液晶屏上面, 直接在 LCD 屏上触摸、滑动操作即可。为了测试方便, 可以将出厂系统的 GUI 应用程序退出, 如何退出呢? 点击屏幕进入设置页面, 可以看到在该页面下有一个退出按钮选项, 直接点击即可!

使用命令 "cat /proc/bus/input/devices", 确定触摸屏对应的设备节点, 如下所示:

```
I: Bus=0018 Vendor=dead Product=beef Version=28bb
N: Name="goodix-ts"
P: Phys=input/ts
S: Sysfs=/devices/virtual/input/input1
U: Uniq=
H: Handlers=mouse0 event1
B: PROP=2
B: EV=b
B: KEY=400 0 0 0 0 0 0 0 0 0
B: ABS=11000003
```

图 17.5.2 触摸屏输入设备的信息 (这里是以 4.3 寸 800*480 屏为例)

笔者使用的是开发板配套的 4.3 寸 800*480 LCD 屏, 如果各位读者使用的是其它屏, 那么查看到的名称可能不是 "goodix-ts"。执行示例代码 17.2.1 对应的可执行文件, 一个手指点击触摸屏先不松开, 终端将会打印如下信息:

```

root@ATK-IMX6U:~# ./testApp /dev/input/event1
type:3 code:57 value:78
type:3 code:53 value:372
type:3 code:54 value:381
type:1 code:330 value:1
type:3 code:0 value:372
type:3 code:1 value:381
type:0 code:0 value:0

```

图 17.5.3 单击不放

首先第一行上报了绝对位移事件 EV_ABS (type=3) 中的 ABS_MT_TRACKING_ID (code=57) 事件, 并且 value 值等于 78, 也就是 ID, 这个 ID 是一个非负数, 所以表示这是一个新的触摸点被创建, 也就意味着触摸屏上产生了一个新的触摸点 (手指按下)。

第二行上报了绝对位移事件 EV_ABS (type=3) 中的 ABS_MT_POSITION_X (code=53) 事件, 其 value 对应的便是触摸点的 X 坐标; 第三行上报了 ABS_MT_POSITION_Y (code=54) 事件, 其 value 值对应的便是触摸点 Y 坐标, 所以由此可知该触摸点的坐标为(372, 381)。

第四行上报了按键类事件 EV_KEY (type=1) 中的 BTN_TOUCH (code=330), value 值等于 1, 表示这是触摸屏上最先产生的触摸点 (slot=0、也就是触摸点 0)。

第五行和第六行分别上报了绝对位移事件 EV_ABS (type=3) 中的 ABS_X (code=0) 和 ABS_Y (code=1), 其 value 分别对应的是触摸点的 X 坐标和 Y 坐标。多点触摸设备也会通过 ABS_X、ABS_Y 事件上报触摸点的 X、Y 坐标, 但通常只有触摸点 0 支持, 所以可以把多点触摸设备当成单点触摸设备来使用。

最后一行上报了同步类事件 EV_SYN (type=0) 中的 SYN_REPORT (code=0) 事件, 表示此次触摸点的信息全部上报完毕。

在第一个触摸点的基础上, 增加第二个触摸点, 打印信息如下所示:

```

root@ATK-IMX6U:~# ./testApp /dev/input/event1
type:3 code:57 value:78
type:3 code:53 value:372
type:3 code:54 value:381
type:1 code:330 value:1
type:3 code:0 value:372
type:3 code:1 value:381
type:0 code:0 value:0
type:3 code:47 value:1
type:3 code:57 value:79
type:3 code:53 value:233
type:3 code:54 value:348
type:0 code:0 value:0

```

图 17.5.4 增加触摸点

1~7 行不再解释, 第八行上报了绝对位移事件 EV_ABS (type=3) 中的 ABS_MT_SLOT 事件 (code=47), 表示目前要更新 slot=1 所关联的触摸点 (也就是触摸点 1) 对应的信息。

第九行上报了绝对位移事件 EV_ABS (type=3) 中的 ABS_MT_TRACKING_ID 事件 (code=57), ID=79, 这是之前没有出现过的 ID, 表示这是一个新的触摸点。

第十、十一行分别上报了 ABS_MT_POSITION_X 和 ABS_MT_POSITION_Y 事件。

最后一行上报同步事件 (type=0、code=0), 告知应用层数据完整。

当手指松开时, 触摸点就会被销毁, 上报 ABS_MT_TRACKING_ID 事件, 并将 value 设置为 -1 (ID), 如下所示:

```
type:3 code:57 value:-1
type:0 code:0 value:0
type:3 code:47 value:0
type:3 code:57 value:-1
type:1 code:330 value:0
type:0 code:0 value:0
```

图 17.5.5 触摸点被销毁

关于触摸屏数据的分析就给大家介绍这么多,不管是键盘也好、或者是鼠标、触摸屏,都可以像上面那样将输入设备的数据直接打印出来,然后自己再去分析,确定该输入设备上报事件的规则和流程,把这些弄懂之后再去编写程序验证结果。下面我们将自己动手编写单点触摸以及多点触摸应用程序,读取触摸点的坐标信息。

17.5.2 获取触摸屏的信息

本小节介绍如何获取触摸屏设备的信息,譬如触摸屏支持的最大触摸点数、触摸屏 X、Y 坐标的范围等。通过 `ioctl()` 函数可以获取到这些信息,3.10.2 小节给大家介绍过该函数, `ioctl()` 是一个文件 I/O 操作的杂物箱,可以处理的事情非常杂、不统一,一般用于操作特殊文件或设备文件,为了方便讲解,再次把 `ioctl()` 函数的原型列出:

```
#include <sys/ioctl.h>
```

```
int ioctl(int fd, unsigned long request, ...);
```

第一个参数 `fd` 对应文件描述符;第二个参数 `request` 与具体要操作的对象有关,没有统一值,表示向文件描述符请求相应的操作,也就是请求指令;此函数是一个可变参函数,第三个参数需要根据 `request` 参数来决定,配合 `request` 来使用。

先来看下输入设备的 `ioctl()` 该怎么用,在 `input.h` 头文件有这样一些宏定义,如下所示:

示例代码 17.5.1 EVIOCG 相关的宏定义

```
#define EVIOCGVERSION      _IOR('E', 0x01, int)          /* get driver version */
#define EVIOCGID          _IOR('E', 0x02, struct input_id) /* get device ID */
#define EVIOCGREP         _IOR('E', 0x03, unsigned int[2]) /* get repeat settings */
#define EVIOCSREP         _IOW('E', 0x03, unsigned int[2]) /* set repeat settings */

#define EVIOCGKEYCODE      _IOR('E', 0x04, unsigned int[2]) /* get keycode */
#define EVIOCGKEYCODE_V2  _IOR('E', 0x04, struct input_keymap_entry)
#define EVIOCSKEYCODE      _IOW('E', 0x04, unsigned int[2]) /* set keycode */
#define EVIOCSKEYCODE_V2  _IOW('E', 0x04, struct input_keymap_entry)

#define EVIOCGNAME(len)    _IOC(_IOC_READ, 'E', 0x06, len) /* get device name */
#define EVIOCGPHYS(len)    _IOC(_IOC_READ, 'E', 0x07, len) /* get physical location */
#define EVIOCGUNIQ(len)    _IOC(_IOC_READ, 'E', 0x08, len) /* get unique identifier */
#define EVIOCGPROP(len)    _IOC(_IOC_READ, 'E', 0x09, len) /* get device properties */

/**
 * EVIOCGMTSLOTS(len) - get MT slot values
 * @len: size of the data buffer in bytes
 */
```



```

* The ioctl buffer argument should be binary equivalent to
*
* struct input_mt_request_layout {
*   __u32 code;
*   __s32 values[num_slots];
* };
*
* where num_slots is the (arbitrary) number of MT slots to extract.
*
* The ioctl size argument (len) is the size of the buffer, which
* should satisfy len = (num_slots + 1) * sizeof(__s32). If len is
* too small to fit all available slots, the first num_slots are
* returned.
*
* Before the call, code is set to the wanted ABS_MT event type. On
* return, values[] is filled with the slot values for the specified
* ABS_MT code.
*
* If the request code is not an ABS_MT value, -EINVAL is returned.
*/
#define EVIOCGMTSLOTS(len)    _IOC(_IOC_READ, 'E', 0x0a, len)

#define EVIOCGKEY(len)        _IOC(_IOC_READ, 'E', 0x18, len)    /* get global key state */
#define EVIOCGLED(len)        _IOC(_IOC_READ, 'E', 0x19, len)    /* get all LEDs */
#define EVIOCGSND(len)        _IOC(_IOC_READ, 'E', 0x1a, len)    /* get all sounds status */
#define EVIOCGSW(len)         _IOC(_IOC_READ, 'E', 0x1b, len)    /* get all switch states */

#define EVIOCGBIT(ev,len)     _IOC(_IOC_READ, 'E', 0x20 + (ev), len) /* get event bits */
#define EVIOCGABS(abs)        _IOR('E', 0x40 + (abs), struct input_absinfo) /* get abs value/limits */
#define EVIOCSABS(abs)        _IOW('E', 0xc0 + (abs), struct input_absinfo) /* set abs value/limits */

#define EVIOCSFF               _IOW('E', 0x80, struct ff_effect) /* send a force effect to a force feedback device */
#define EVIOCRMFF              _IOW('E', 0x81, int) /* Erase a force effect */
#define EVIOCGEFFECTS          _IOR('E', 0x84, int) /* Report number of effects playable at
the same time */

#define EVIOCGRAB              _IOW('E', 0x90, int) /* Grab/Release device */
#define EVIOCREVOKE            _IOW('E', 0x91, int) /* Revoke device access */

```

每一个宏定义后面都有相应的注释，对于 input 输入设备，对其执行 ioctl() 操作需要使用这些宏，不同的宏表示不同请求指令；譬如使用 EVIOCGNAME 宏获取设备名称，使用方式如下：

```

char name[100];
ioctl(fd, EVIOCGNAME(sizeof(name)), name);

```

EVIOCNAME(len)就表示用于接收字符串数据的缓冲区大小,而此时 ioctl()函数的第三个参数需要传入一个缓冲区的地址,该缓冲区用于存放设备名称对应的字符串数据。

EVIOCG (get) 开头的表示获取信息, EVIOC S (set) 开头表示设置;这里暂且不管其它宏,重点来看看 EVIOCGABS(abs)宏,这个宏也是通常使用最多的,如下所示:

```
#define EVIOCGABS(abs) _IOR('E', 0x40 + (abs), struct input_absinfo)
```

通过这个宏可以获取到触摸屏 slot (slot<0>表示触摸点 0、slot<1>表示触摸点 1、slot<2>表示触摸点 2,以此类推!)的取值范围,可以看到使用该宏需要传入一个 abs 参数,该参数表示为一个 ABS_XXX 绝对位移事件,譬如 EVIOCGABS(ABS_MT_SLOT)表示获取触摸屏的 slot 信息,此时 ioctl()函数的第三个参数是一个 struct input_absinfo *的指针,指向一个 struct input_absinfo 对象,调用 ioctl()会将获取到的信息写入到 struct input_absinfo 对象中。struct input_absinfo 结构体如下所示:

示例代码 17.5.2 struct input_absinfo 结构体

```
struct input_absinfo {
    __s32 value;           //最新的报告值
    __s32 minimum;       //最小值
    __s32 maximum;       //最大值
    __s32 fuzz;
    __s32 flat;
    __s32 resolution;
};
```

获取触摸屏支持的最大触摸点数:

```
struct input_absinfo info;
int max_slots; //最大触摸点数

if (0 > ioctl(fd, EVIOCGABS(ABS_MT_SLOT), &info))
    perror("ioctl error");
```

```
max_slots = info.maximum + 1 - info.minimum;
```

其它宏定义不再介绍,读者可以自行测试。

获取触摸屏支持的最大触摸点数

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->17_input->read_slot.c](#)。

示例代码 17.5.3 读取触摸屏支持的最大触摸点数

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/input.h>

int main(int argc, char *argv[])
{
    struct input_absinfo info;
    int fd = -1;
```

```
int max_slots;

/* 校验传参 */
if (2 != argc) {
    fprintf(stderr, "usage: %s <input-dev>\n", argv[0]);
    exit(EXIT_FAILURE);
}

/* 打开文件 */
if (0 > (fd = open(argv[1], O_RDONLY))) {
    perror("open error");
    exit(EXIT_FAILURE);
}

/* 获取 slot 信息 */
if (0 > ioctl(fd, EVIOCGABS(ABS_MT_SLOT), &info)) {
    perror("ioctl error");
    close(fd);
    exit(EXIT_FAILURE);
}

max_slots = info.maximum + 1 - info.minimum;
printf("max_slots: %d\n", max_slots);

/* 关闭、退出 */
close(fd);
exit(EXIT_SUCCESS);
}
```

编译示例代码, 将其拷贝到开发板 Linux 系统的用户家目录下, 执行该应用程序:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp /dev/input/event1
max_slots: 5
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 17.5.6 运行结果

所以从打印结果可知, 我们这个屏是一个 5 点触摸屏。

17.5.3 单点触摸应用程序

通过上面的详细介绍, 详细大家应该知道如何去编写一个触摸屏的应用程序了, 本小节我们编写一个单点触摸应用程序, 获取一个触摸点的坐标信息, 并将其打印出来。

ALPHA/Mini 开发板配套使用的触摸屏均支持多点触摸, 这里我们把它当成单点触摸设备来使用, 编写一个程序读取一个触摸点, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->17_input->read_ts.c](#)。

示例代码 17.5.4 单点触摸应用程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/input.h>

int main(int argc, char *argv[])
{
    struct input_event in_ev;
    int x, y; //触摸点 x 和 y 坐标
    int down; //用于记录 BTN_TOUCH 事件的 value,1 表示按下,0 表示松开,-1 表示移动
    int valid; //用于记录数据是否有效(我们关注的信息发生更新表示有效,1 表示有效,0 表示无效)
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <input-dev>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 打开文件 */
    if (0 > (fd = open(argv[1], O_RDONLY))) {
        perror("open error");
        exit(EXIT_FAILURE);
    }

    x = y = 0; //初始化 x 和 y 坐标值
    down = -1; //初始化<移动>
    valid = 0; //初始化<无效>
    for (;;) {

        /* 循环读取数据 */
        if (sizeof(struct input_event) !=
            read(fd, &in_ev, sizeof(struct input_event))) {
            perror("read error");
            exit(EXIT_FAILURE);
        }
    }
}
```

```
switch (in_ev.type) {
case EV_KEY:    //按键事件
    if (BTN_TOUCH == in_ev.code) {
        down = in_ev.value;
        valid = 1;
    }
    break;

case EV_ABS:    //绝对位移事件
    switch (in_ev.code) {
case ABS_X: //X 坐标
        x = in_ev.value;
        valid = 1;
        break;
case ABS_Y: //Y 坐标
        y = in_ev.value;
        valid = 1;
        break;
    }
    break;

case EV_SYN:    //同步事件
    if (SYN_REPORT == in_ev.code) {
        if (valid) { //判断是否有效
            switch (down) { //判断状态
            case 1:
                printf("按下(%d, %d)\n", x, y);
                break;
            case 0:
                printf("松开\n");
                break;
            case -1:
                printf("移动(%d, %d)\n", x, y);
                break;
            }

            valid = 0; //重置 valid
            down = -1; //重置 down
        }
    }
    break;
}
```

```

}
}

```

程序中首先校验传参, 通过传参的方式将触摸屏设备文件路径传入到程序中, main()函数中定义了 4 个变量:

(1)、变量 x 表示触摸点的 X 坐标;

(2)、变量 y 表示触摸点的 Y 坐标;

(3)、变量 down 表示手指状态时候按下、松开还是滑动, down=1 表示手指按下、down=0 表示手指松开、down=-1 表示手指滑动;

(4)、变量 valid 表示数据是否有效, valid=1 表示有效、valid=0 表示无效; 有效指的是我们检测的信息发生了更改, 譬如程序中只检测了手指的按下、松开动作以及坐标值的变化。

接着调用 open()打开触摸屏设备文件得到文件描述符 fd; 在 for 循环之前, 首先对 x、y、down、valid 这 4 个变量进行初始化操作。在 for 循环读取触摸屏上报的数据, 将读取到的数据存放在 struct input_event 数据结构中。在 switch...case 语句中对读取到的数据进行解析, 获取 BTN_TOUCH 事件的 value 数据, 判断触摸屏是按下还是松开状态, 获取 ABS_X 和 ABS_Y 事件的 value 变量, 得到触摸点的 X 轴坐标和 Y 轴坐标。

当上报同步事件时, 表示数据已经完整, 接着对我们得到的数据进行分析、打印坐标信息。

编译应用程序:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 17.5.7 编译应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下, 准备进行测试。

笔者使用的是 4.3 寸 800*480 LCD 屏, 执行单点触摸应用程序, 程序执行之后, 接着用一个手指按下触摸屏、松开以及滑动操作, 串口终端将会打印出相应的信息:

```

root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp /dev/input/event1
按下 (612, 291)
松开
按下 (634, 282)
移动 (632, 279)
移动 (631, 277)
移动 (631, 276)
移动 (630, 275)
移动 (630, 274)
松开
按下 (379, 321)
松开
按下 (360, 313)
松开
按下 (358, 309)
松开
按下 (360, 312)
移动 (359, 309)
移动 (358, 307)

```

图 17.5.8 单点触摸应用程序测试结果

当手指点击触摸屏时会打印"按下(X, Y)",松开时会打印"松开",手指在触摸屏上滑动时会打印"移动(X, Y)"等信息。大家可以自己动手测试,对代码不理解的,可以对照测试结果进行对比。

17.5.4 多点触摸应用程序

介绍完单点触摸应用程序之后,再来看看多点触摸应用程序该如何编写,前面已经详细给大家介绍了多点触摸设备的事件上报流程。

本例程源码对应的路径为: **开发板光盘->11、Linux C 应用编程例程源码->17_input->read_mt.c。**

示例代码 17.5.5 多点触摸应用程序

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : read_mt.c
作者 : 邓涛
版本 : V1.0
描述 : 触摸屏多点触摸应用程序示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/6/15 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>
#include <linux/input.h>

/* 用于描述 MT 多点触摸每一个触摸点的信息 */
struct ts_mt {
    int x;        //X 坐标
    int y;        //Y 坐标
    int id;       //对应 ABS_MT_TRACKING_ID
    int valid;    //数据有效标志位(=1 表示触摸点信息发生更新)
};

/* 一个触摸点的 x 坐标和 y 坐标 */
struct tp_xy {
    int x;
    int y;
};

```

```
static int ts_read(const int fd, const int max_slots,
                 struct ts_mt *mt)
{
    struct input_event in_ev;
    static int slot = 0; //用于保存上一个 slot
    static struct tp_xy xy[12] = {0}; //用于保存上一次的 x 和 y 坐标值,假设触摸屏支持的最大触摸点数不会超过 12
    int i;

    /* 对缓冲区初始化操作 */
    memset(mt, 0x0, max_slots * sizeof(struct ts_mt)); //清零
    for (i = 0; i < max_slots; i++)
        mt[i].id = -2; //将 id 初始化为-2, id=-1 表示触摸点删除, id>=0 表示创建

    for (;;) {

        if (sizeof(struct input_event) !=
            read(fd, &in_ev, sizeof(struct input_event))) {
            perror("read error");
            return -1;
        }

        switch (in_ev.type) {
        case EV_ABS:
            switch (in_ev.code) {
            case ABS_MT_SLOT:
                slot = in_ev.value;
                break;
            case ABS_MT_POSITION_X:
                xy[slot].x = in_ev.value;
                mt[slot].valid = 1;
                break;
            case ABS_MT_POSITION_Y:
                xy[slot].y = in_ev.value;
                mt[slot].valid = 1;
                break;
            case ABS_MT_TRACKING_ID:
                mt[slot].id = in_ev.value;
                mt[slot].valid = 1;
                break;
            }
            break;

            //case EV_KEY://按键事件对单点触摸应用比较有用

```



```
        // break;
        case EV_SYN:
            if (SYN_REPORT == in_ev.code) {
                for (i = 0; i < max_slots; i++) {
                    mt[i].x = xy[i].x;
                    mt[i].y = xy[i].y;
                }
            }
            return 0;
        }
    }
}

int main(int argc, char *argv[])
{
    struct input_absinfo slot;
    struct ts_mt *mt = NULL;
    int max_slots;
    int fd;
    int i;

    /* 参数校验 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <input_dev>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDONLY);
    if (0 > fd) {
        perror("open error");
        exit(EXIT_FAILURE);
    }

    /* 获取触摸屏支持的最大触摸点数 */
    if (0 > ioctl(fd, EVIOCGABS(ABS_MT_SLOT), &slot)) {
        perror("ioctl error");
        close(fd);
        exit(EXIT_FAILURE);
    }

    max_slots = slot.maximum + 1 - slot.minimum;
    printf("max_slots: %d\n", max_slots);
}
```

```
/* 申请内存空间并清零 */
mt = calloc(max_slots, sizeof(struct ts_mt));

/* 读数据 */
for (;;) {

    if (0 > ts_read(fd, max_slots, mt))
        break;

    for (i = 0; i < max_slots; i++) {
        if (mt[i].valid) { //判断每一个触摸点信息是否发生更新（关注的信息发生更新）

            if (0 <= mt[i].id)
                printf("slot<%d>, 按下(%d, %d)\n", i, mt[i].x, mt[i].y);
            else if (-1 == mt[i].id)
                printf("slot<%d>, 松开\n", i);
            else
                printf("slot<%d>, 移动(%d, %d)\n", i, mt[i].x, mt[i].y);
        }
    }
}

/* 关闭设备、退出 */
close(fd);
free(mt);
exit(EXIT_FAILURE);
}
```

示例代码中申明了 `struct ts_mt` 数据结构，用于描述多点触摸情况下每一个触摸点的信息。

首先来看下 `main()` 函数，定义了 `max_slots` 变量，用于指定触摸屏设备的支持的最大触摸点数，通过：
`ioctl(fd, EVIOCGABS(ABS_MT_SLOT), &slot)`

获取到触摸屏该信息。

接着根据 `max_slots` 变量的值，为 `mt` 指针申请内存：

```
mt = calloc(max_slots, sizeof(struct ts_mt));
```

`for(;;)` 循环中调用 `ts_read()` 函数，该函数是自定义函数，用于获取触摸屏上报的数据，第一个参数表示文件描述符 `fd`、第二个参数表示触摸屏支持的最大触摸点数、第三个参数则是 `struct ts_mt` 数组，`ts_read()` 函数会将获取到的数据存放在数组中，`mt[0]` 表示 `slot<0>` 数据、`mt[1]` 表示 `slot<1>` 的数据依次类推！

在内部的 `for` 循环中，则对获取到的数据进行分析，判断数据是否有效，并根据 `id` 判断手指的动作，在单点触摸应用程序中，我们是通过 `BTN_TOUCH` 事件来判断手指的动作；而在多点触摸应用中，我们需要通过 `id` 来判断多个手指的动作。

关于自定义函数 `ts_read()` 就不再介绍了，代码的注释已经描述很清楚了！

接着编译应用程序，将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下，执行应用程序，接着可以用多个手指触摸触摸屏、松开、滑动等操作：

```
root@ATK-IMX6U:~# ./testApp /dev/input/event1
max_slots: 5
slot<0>, 按下 (728, 415)
slot<1>, 按下 (271, 364)
slot<2>, 按下 (661, 284)
slot<3>, 按下 (403, 276)
slot<4>, 按下 (157, 371)
slot<3>, 松开
slot<4>, 松开
slot<3>, 按下 (405, 275)
slot<3>, 移动 (402, 274)
slot<0>, 移动 (728, 417)
slot<3>, 移动 (401, 274)
slot<3>, 移动 (400, 274)
slot<3>, 移动 (399, 274)
slot<3>, 移动 (398, 274)
slot<3>, 移动 (397, 274)
slot<3>, 移动 (396, 274)
slot<3>, 移动 (395, 274)
slot<3>, 移动 (394, 274)
slot<3>, 松开
slot<2>, 移动 (661, 285)
slot<2>, 松开
slot<0>, 松开
slot<1>, 松开
```

图 17.5.9 多点触摸应用程序测试结果

每一个不同的 slot 表示不同的触摸点，譬如 slot<0>表示触摸点 0、slot<1>表示触摸点 1 以此类推！

17.6 鼠标应用编程

本小节是笔者留给各位读者的一个作业，交给大家去完成，通过本章内容的介绍相信大家可以独立完成，ALPHA/Mini 开发板出厂系统支持 USB 鼠标，直接将一个 USB 鼠标插入到开发板的 USB HOST 接口即可，在终端会打印驱动加载信息。

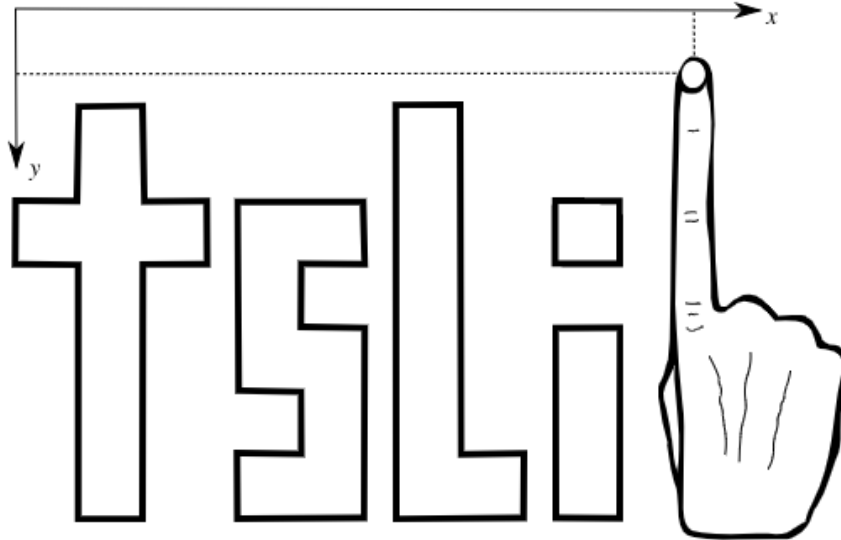
第十八章 使用 tslib 库

上一章我们学习了如何编写触摸屏应用程序，包括单点触摸和多点触摸，主要是对读取到的 `struct input_event` 类型数据进行剖析，得到各个触摸点的坐标。本章向大家介绍 `tslib` 库，这是 Linux 系统下，专门为触摸屏开发的应用层函数库，本章我们将学习如何基于 `tslib` 库编写触摸屏应用程序。

本章将会讨论如下主题内容。

- tslib 简介；
- tslib 移植；
- tslib 库函数的使用介绍；
- 基于 tslib 库函数编写触摸屏应用程序。

18.1 tslib 简介



C library for filtering touchscreen events

图 18.1.1 tslib

tslib 是专门为触摸屏设备所开发的 Linux 应用层函数库，并且是开源，也就意味着我们可以直接获取到 tslib 的源代码，下一小节将向大家介绍如何获取到 tslib 的源代码。

tslib 为触摸屏驱动和应用层之间的适配层，它把应用程序中读取触摸屏 `struct input_event` 类型数据（这是输入设备上报给应用层的原始数据）并进行解析的操作过程进行了封装，向使用者提供了封装好的 API 接口。tslib 从触摸屏中获得原始的坐标数据，并通过一系列的去噪、去抖、坐标变换等操作，来去除噪声并将原始的触摸屏坐标转换为相应的屏幕坐标。

tslib 有一个配置文件 `ts.conf`，该配置文件中提供了一些配置参数、用户可以对其进行修改，具体的配置信息稍后介绍！

tslib 可以作为 Qt 的触摸屏输入插件，为 Qt 提供触摸输入支持，如果在嵌入式 Linux 硬件平台下开发过 Qt 应用程序的读者应该知道；当然，并不是只有 tslib 才能作为 Qt 的插件、为其提供触摸输入支持，还有很多插件都可以，只不过大部分都会选择使用 tslib。

关于 tslib 就介绍这么多，接下来看看如何将 tslib 库移植到我们的开发板平台上。

18.2 tslib 移植

18.2.1 下载 tslib 源码

首先下载 tslib 源码包，进入到 tslib 的 git 仓库下载源码 <https://github.com/libts/tslib/releases>，如下：

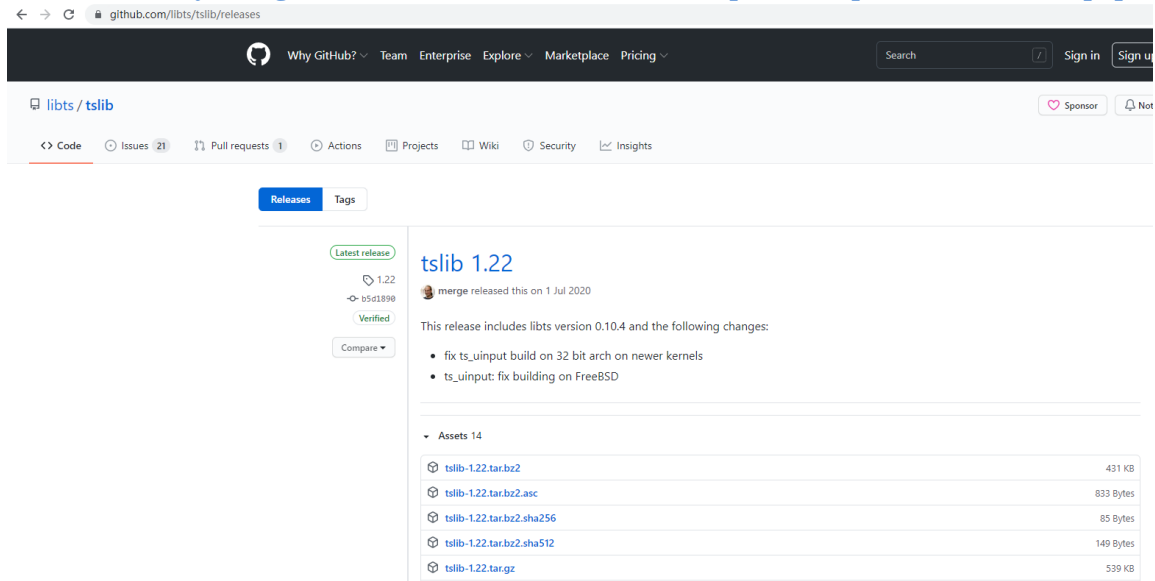


图 18.2.1 tslib git 链接

ALPHA/Mini 开发板出厂系统中已经移植了 tslib，并且版本为 1.16，可以在开发板执行 `ts_finddev` 命令查看到它的版本信息，如下所示：

```
root@ATK-IMX6U:~# ts_finddev
          tslib
tslib 1.16 / libts ABI version 0 (0x000901)
release-date: 2018-04-19

Usage: ts_finddev device_name wait_for_sec
device_name - tdevice to probe, example /dev/input/event0
wait_for_sec - wait seconds for touch event, if 0 - don't wait!
Return codes:
    0 - timeout expired without receiving event.
        But this maybe is TouchScreen.
   -1 - this is NOT TouchScreen device!
    1 - this is TouchScreen for shure!
root@ATK-IMX6U:~#
```

图 18.2.2 查看 tslib 版本

所以为了统一，我们也下载 1.16 版本的 tslib，往下翻找到 1.16 版本的下载链接：

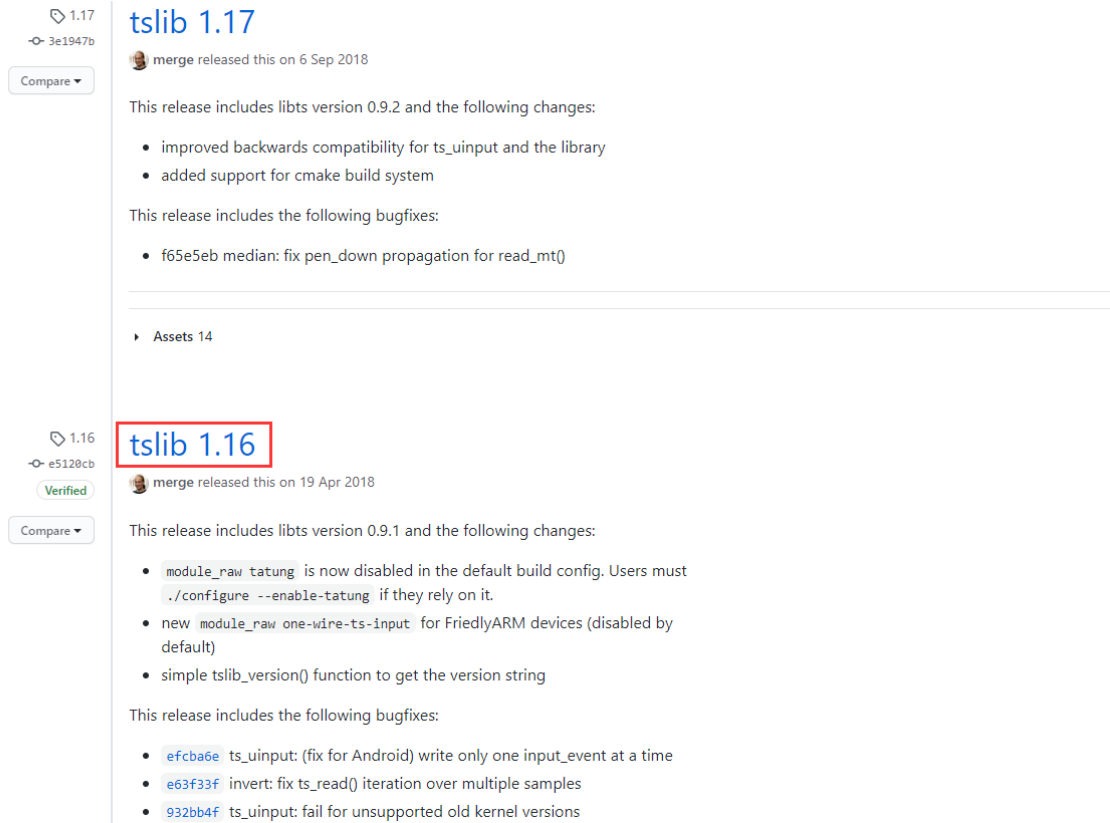


图 18.2.3 1.16 版本

点击红框字样进入下载页面:

tslib-1.16.tar.bz2	402 KB
tslib-1.16.tar.bz2.asc	833 Bytes
tslib-1.16.tar.bz2.sha256	85 Bytes
tslib-1.16.tar.bz2.sha512	149 Bytes
tslib-1.16.tar.gz	507 KB
tslib-1.16.tar.gz.asc	833 Bytes
tslib-1.16.tar.gz.sha256	84 Bytes
tslib-1.16.tar.gz.sha512	148 Bytes
tslib-1.16.tar.xz	326 KB
tslib-1.16.tar.xz.asc	833 Bytes
tslib-1.16.tar.xz.sha256	84 Bytes
tslib-1.16.tar.xz.sha512	148 Bytes
Source code (zip)	
Source code (tar.gz)	

图 18.2.4 tslib 源码包

推荐下载 tar.bz2 或 tar.gz 格式压缩包, 或者 tar.xz 压缩包, 这里笔者下载 tar.gz 格式的压缩包文件, 点击文字即可下载。



图 18.2.5 tslib-1.16.tar.gz

18.2.2 编译 tslib 源码

将 tslib-1.16.tar.gz 源码包拷贝到 Ubuntu 系统的用户家目录下:

```
dt@dt-virtual-machine:~$ ls -l
总用量 560
drwxrwxr-x 2 dt dt 4096 1月 19 19:59 eclipse_ws
-rw-r--r-- 1 dt dt 8980 12月 19 2020 examples.desktop
-rw-r--r-- 1 dt dt 518727 6月 24 18:50 tslib-1.16.tar.gz
drwxrwxr-x 5 dt dt 4096 4月 6 18:13 vscode_ws
drwxr-xr-x 2 dt dt 4096 12月 19 2020 公共的
drwxr-xr-x 2 dt dt 4096 12月 19 2020 模板
drwxr-xr-x 2 dt dt 4096 12月 19 2020 视频
drwxr-xr-x 2 dt dt 4096 12月 19 2020 图片
drwxr-xr-x 2 dt dt 4096 12月 19 2020 文档
drwxr-xr-x 2 dt dt 4096 12月 19 2020 下载
drwxr-xr-x 2 dt dt 4096 12月 19 2020 音乐
drwxr-xr-x 2 dt dt 4096 6月 3 15:49 桌面
dt@dt-virtual-machine:~$
```

图 18.2.6 将 tslib 源码拷贝到 Ubuntu 系统

将其解压到当前目录下:

```
tar -xzf tslib-1.16.tar.gz

dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ tar -xzf tslib-1.16.tar.gz
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  tslib-1.16  tslib-1.16.tar.gz  vscode_ws
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 18.2.7 解压 tslib 压缩包

解压之后会生成 tslib-1.16 目录, 在家目录下创建一个 tools 目录, 然后在 tools 目录下创建 tslib 目录, 等会编译 tslib 库的时候将安装目录指定到这里, 如下所示:

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ mkdir tools
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd tools/
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
dt@dt-virtual-machine:~/tools$ mkdir tslib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
tslib
dt@dt-virtual-machine:~/tools$
```

图 18.2.8 创建 tslib 目录

进入到 tslib-1.16 目录, 准备进行编译 tslib 源码:

```
dt@dt-virtual-machine:~/tools$ cd ../
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  tools  tslib-1.16  tslib-1.16.tar.gz  vscode_ws  公共的  模板  视
dt@dt-virtual-machine:~$ cd tslib-1.16/
dt@dt-virtual-machine:~/tslib-1.16$ ls
acinclude.m4  AUTHORS  ChangeLog  config.guess  config.sub  configure.ac  depcomp  etc
aclocal.m4  autogen.sh  compile  config.h.in  configure  COPYING  doc  INSTALL
dt@dt-virtual-machine:~/tslib-1.16$
```

图 18.2.9 tslib 源码

接下来进行编译, 整个源码的编译分为 3 个步骤:

- 首先第一步是配置工程;
- 第二步是编译工程;
- 第三步是安装, 将编译得到的库文件、可执行文件等安装到一个指定的目录下。

首先在配置工程之前, 先对交叉编译工具的环境进行设置, 使用 source 执行交叉编译工具安装目录下的 environment-setup-cortexa7hf-neon-poky-linux-gnueabi 脚本文件:

```
source /opt/fsl-ixm-x11/4.1.15-2.1.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

执行下面这条命令对 tslib 源码工程进行配置:

```
./configure --host=arm-poky-linux-gnueabi --prefix=/home/dt/tools/tslib/
```

至于工程是如何配置的, 大家可以执行 ./configure --help 查看它的配置选项以及含义, --host 选项用于指定交叉编译得到的库文件是运行在哪个平台, 通常将 --host 设置为交叉编译器名称的前缀, 譬如 arm-poky-linux-gnueabi-gcc 前缀就是 arm-poky-linux-gnueabi; --prefix 选项则用于指定库文件的安装路径, 我们将安装路径设置为之前在家目录下创建的 tools/tslib 目录。

```
dt@dt-virtual-machine:~/tslib-1.16$ ./configure --host=arm-poky-linux-gnueabi --prefix=/home/dt/tools/tslib/
configure: loading site script /opt/fsl-ixm-x11/4.1.15-2.1.0/site-config-cortexa7hf-neon-poky-linux-gnueabi
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for arm-poky-linux-gnueabi-strip... arm-poky-linux-gnueabi-strip
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether make supports nested variables... (cached) yes
checking build system type... x86_64-pc-linux-gnu
checking host system type... arm-poky-linux-gnueabi
checking for arm-poky-linux-gnueabi-gcc... arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixm-x11/4.1.15-2.1.0/ortexta7hf-neon-poky-linux-gnueabi
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... yes
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixm-x11/4.1.15-2.1.0/ortexta7hf-neon-poky-linux-gnueabi accepts -g... yes
checking for arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixm-x11/4.1.15-2.1.0/ortexta7hf-neon-poky-linux-gnueabi option to accept ISO C89... none needed
checking whether arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixm-x11/4.1.15-2.1.0/ortexta7hf-neon-poky-linux-gnueabi understands -c and -o together... yes
checking for style of include used by make... GNU
checking dependency style of arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixm-x11/4.1.15-2.1.0/ortexta7hf-neon-poky-linux-gnueabi... none
checking how to run the C preprocessor... arm-poky-linux-gnueabi-gcc -E -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixm-x11/4.1.15-2.1.0/ortexta7hf-neon-poky-linux-gnueabi
checking whether the C compiler supports -fvisibility=hidden... yes
```

图 18.2.10 配置工程

接着编译工程, 直接执行 make:

```
make
```



```
dt@dt-virtual-machine:~/tslib-1.16$  
dt@dt-virtual-machine:~/tslib-1.16$ cd /home/dt/tools/tslib/  
dt@dt-virtual-machine:~/tools/tslib$  
dt@dt-virtual-machine:~/tools/tslib$ ls  
bin  etc  include  lib  share  
dt@dt-virtual-machine:~/tools/tslib$  
dt@dt-virtual-machine:~/tools/tslib$  
dt@dt-virtual-machine:~/tools/tslib$
```

图 18.2.13 tslib 安装目录下的文件夹

bin 目录

bin 目录下有一些 tslib 提供的小工具，可以用于测试触摸屏，如下所示：

```
dt@dt-virtual-machine:~/tools/tslib$ cd bin/  
dt@dt-virtual-machine:~/tools/tslib/bin$ ls  
ts_calibrate  ts_finddev  ts_harvest  ts_print  ts_print_mt  ts_print_raw  ts_test  ts_test_mt  ts_uinput  ts_verify  
dt@dt-virtual-machine:~/tools/tslib/bin$  
dt@dt-virtual-machine:~/tools/tslib/bin$
```

图 18.2.14 bin 目录下的文件

etc 目录

etc 目录下有一个配置文件 ts.conf，前面给大家提到过，

```
dt@dt-virtual-machine:~/tools/tslib/bin$  
dt@dt-virtual-machine:~/tools/tslib/bin$ cd ../etc/  
dt@dt-virtual-machine:~/tools/tslib/etc$  
dt@dt-virtual-machine:~/tools/tslib/etc$ ls  
ts.conf  
dt@dt-virtual-machine:~/tools/tslib/etc$  
dt@dt-virtual-machine:~/tools/tslib/etc$
```

图 18.2.15 配置文件 ts.conf

打开 ts.conf 文件看看它有哪些配置选项：

```

# Access plugins
#####

# Uncomment if you wish to use the linux input layer event interface
module_raw input

# For other driver modules, see the ts.conf man page

# Filter plugins
#####

# Uncomment if first or last samples are unreliable
# module skip nhead=1 ntail=1

# Uncomment if needed for devices that measure pressure
module pthres pmin=1

# Uncomment if needed
# module debounce drop_threshold=40

# Uncomment if needed to filter spikes
# module median depth=5

# Uncomment to enable smoothing of fraction N/D
# module iir N=6 D=10

# Uncomment if needed
# module lowpass factor=0.1 threshold=1

# Uncomment if needed to filter noise samples
module dejitter delta=100

# Uncomment and adjust if you need to invert an axis or both
# module invert x0=800 y0=480

# Uncomment to use ts_calibrate's settings
module linear
~

```

图 18.2.16 ts.conf 文件的内容

module_raw input: 取消注释, 使能支持 input 输入事件;

module pthres pmin=1: 如果我们的设备支持按压力大小测试, 那么可以把它的注释取消, pmin 用于调节按压力灵敏度, 默认就是等于 1。

module dejitter delta=100: tslib 提供了触摸屏去噪算法插件, 如果需要过滤噪声样本, 取消注释, 默认参数 delta=100。

module linear: tslib 提供了触摸屏坐标变换的功能, 譬如将 X、Y 坐标互换、坐标旋转等之类, 如果我们需要实现坐标变换, 可以把注释去掉。

这里就不去改动了, 直接使用默认的配置就行了。

include 目录

include 目录下只有一个头文件 tslib.h, 该头文件中包含了一些结构体数据结构以及 API 接口的申明, 使用 tslib 提供的 API 就需要包含该头文件。

lib 目录

lib 目录下包含了编译 tslib 源码所得到的库文件, 默认这些都是动态库文件, 也可以通过配置 tslib 工程使其生成静态库文件; ts 目录下存放的是一些插件库。

```

dt@dt-virtual-machine:~/tools/tslib/lib$
dt@dt-virtual-machine:~/tools/tslib/lib$ ls
libts.la libts.so libts.so.0 libts.so.0.9.1 pkgconfig ts
dt@dt-virtual-machine:~/tools/tslib/lib$
dt@dt-virtual-machine:~/tools/tslib/lib$

```

图 18.2.17 lib 目录

share 目录

可以忽略!

18.2.4 在开发板上测试 tslib

移植的最后一步就是把 tslib 安装目录下的库文件、etc 下的配置文件以及编译得到的测试工具拷贝到开发板 Linux 系统目录下, 由于开发板出厂系统中已经移植了 tslib 库, 所以我们这里就不用拷贝了。如果大家是自己做的根文件系统, 并没有移植 tslib, 那么就需要把这些库、可执行文件以及配置文件拷贝到根文件系统中, 那怎么去拷贝? 这里简单地提一下:

- 将安装目录 bin/目录下的所有可执行文件拷贝到开发板/usr/bin 目录下;
- 将安装目录 etc/目录下的配置文件 ts.conf 拷贝到开发板/etc 目录下;
- 将安装目录 lib/目录下的所有库文件拷贝到开发板/usr/lib 目录下。

将安装目录下的测试工具、库文件以及配置文件拷贝到开发板之后, 接着需要配置一些环境变量, 因为 tslib 工作的时候它需要依赖于一些环境变量, 譬如它会通过读取环境变量来得知 ts.conf 配置文件、库文件的路径以及我们要测试的触摸屏对应的设备节点等。

```
export TSLIB_CONSOLEDEVICE=none
export TSLIB_FBDEVICE=/dev/fb0
export TSLIB_TSDEVICE=/dev/input/event1
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_PLUGINDIR=/usr/lib/ts
```

TSLIB_CONSOLEDEVICE: 用于配置控制台设备文件名, 直接配置为 none 即可!

TSLIB_FBDEVICE: 用于配置显示设备的名称, tslib 提供了手指触摸画线的测试工具, 需要在 LCD 上显示, 所以这里需要指定一个显示设备的设备节点。

TSLIB_TSDEVICE: 用于配置触摸屏对应的设备节点, 根据实际情况配置。

TSLIB_CONFFILE: 用于配置 ts.conf 文件的所在路径。

TSLIB_PLUGINDIR: 用于配置插件所在路径。

```
root@ATK-IMX6U:~# export TSLIB_CONSOLEDEVICE=none
root@ATK-IMX6U:~# export TSLIB_FBDEVICE=/dev/fb0
root@ATK-IMX6U:~# export TSLIB_TSDEVICE=/dev/input/event1
root@ATK-IMX6U:~# export TSLIB_CONFFILE=/etc/ts.conf
root@ATK-IMX6U:~# export TSLIB_PLUGINDIR=/usr/lib/ts
root@ATK-IMX6U:~#
```

图 18.2.18 配置环境变量

如果想每次启动系统都能生效, 可以把这些命令放置在/etc/profile 脚本中执行; 出厂系统中已经配置好了, 无需用户进行配置。

接着我们使用 tslib 提供的测试工具测试触摸屏, 它提供了单点触摸测试工具 (ts_print、ts_test) 和多点触摸测试工具 (ts_print_mt、ts_test_mt), ts_print 和 ts_print_mt 可以在终端打印触摸点信息, 而 ts_test 和 ts_test_mt 则支持在 LCD 上画线。

执行 ts_print 命令:

```

root@ATK-IMX6U:~# ts_print
1624623180.204525:    482    264    255
1624623180.277369:    482    264     0
1624623180.734769:    650    231    255
1624623180.891575:    650    231     0
1624623181.189959:    683    237    255
1624623181.346493:    683    237     0
1624623181.538596:    725    223    255
1624623181.632435:    725    223     0
1624623181.888321:    707    287    255
1624623182.079135:    707    287     0
1624623182.268629:    738    209    255
1624623182.415403:    738    209     0
1624623182.628841:    650    298    255
1624623182.775159:    650    298     0
1624623183.031088:    692    259    255

```

图 18.2.19 ts_print 测试工具

执行 `ts_print` 命令之后, 在触摸屏上滑动、或按下、松开触摸屏将会在终端打印出相应的信息。同理, `ts_print_mt` 也是如此, 不过它支持多点触摸, 可以打印多个触摸点的信息:

```

root@ATK-IMX6U:~# ts_print_mt
libts 000901 opened device /dev/input/event1
sample 0 - 1624624008.548093 - (slot 0) 881 270 255
sample 0 - 1624624008.782456 - (slot 1) 533 333 255
sample 0 - 1624624009.339811 - (slot 2) 616 184 255
sample 0 - 1624624010.539823 - (slot 3) 764 190 255
sample 0 - 1624624011.202071 - (slot 3) 764 190 0
sample 0 - 1624624011.981750 - (slot 3) 758 190 255
sample 0 - 1624624012.096598 - (slot 3) 758 190 0
sample 0 - 1624624012.128223 - (slot 2) 616 184 0
sample 0 - 1624624012.201831 - (slot 2) 613 196 255
sample 0 - 1624624012.369453 - (slot 2) 613 196 0
sample 0 - 1624624012.591067 - (slot 2) 614 191 255
sample 0 - 1624624012.790881 - (slot 2) 612 191 255
sample 0 - 1624624012.801440 - (slot 2) 610 192 255
sample 0 - 1624624012.811907 - (slot 2) 609 193 255
sample 0 - 1624624012.822547 - (slot 2) 607 195 255
sample 0 - 1624624012.832997 - (slot 2) 605 197 255
sample 0 - 1624624012.854344 - (slot 2) 603 199 255
sample 0 - 1624624012.980324 - (slot 1) 533 333 255
sample 0 - 1624624012.990241 - (slot 1) 533 334 0
sample 0 - 1624624013.064696 - (slot 2) 603 200 255

```

图 18.2.20 ts_print_mt 测试工具

`ts_test` 和 `ts_test_mt` 支持触摸屏画线操作, 这里就不再给演示了, 自己去测试即可! 如果大家想看这些测试工具的源码实现, 可以在 `tslib` 源码中找到, 具体路径为 `tslib` 源码目录下的 `tests` 文件夹中:

```

dt@dt-virtual-machine:~$ cd tslib-1.16/tests/
dt@dt-virtual-machine:~/tslib-1.16/tests$
dt@dt-virtual-machine:~/tslib-1.16/tests$ ls
COPYING      font_8x16.c  Makefile     testutils.c  ts_calibrate_common.c  ts_finddev  ts_harvest.o  ts_print_mt.o  ts_test      ts_test_mt_sdl.c
fbutils-bsd.c  font_8x16.o  Makefile.am  testutils.h  ts_calibrate_common.o  ts_finddev.c  ts_print      ts_print.o    ts_test.c    ts_test.o
fbutils.h      font_8x8.c  Makefile.in  testutils.o  ts_calibrate.h         ts_finddev.o  ts_print.c    ts_print_raw  ts_test_mt   ts_verify
fbutils-linux.c font_8x8.o  sdutils.c    ts_calibrate  ts_calibrate.o         ts_harvest    ts_print_mt   ts_print_raw.c  ts_test_mt.c  ts_verify.c
fbutils-linux.o font.h       sdutils.h    ts_calibrate.c  ts_calibrate_sdl.c     ts_harvest.c  ts_print_mt.c  ts_print_raw.o  ts_test_mt.o  ts_verify.o
dt@dt-virtual-machine:~/tslib-1.16/tests$

```

图 18.2.21 tests 文件夹

譬如 `ts_test` 程序对应的源码实现为 `ts_test.c`, 不管它怎么做最终都是落实到上一章给大家介绍的内容中。

18.3 tslib 库函数介绍

本小节介绍如何使用 `tslib` 提供的 API 接口来编写触摸屏应用程序, 使用 `tslib` 库函数需要在我们的应用程序中包含 `tslib` 的头文件 `tslib.h`, 使用 `tslib` 编程其实非常简单, 基本步骤如下所示:

第一步打开触摸屏设备:

第二步配置触摸屏设备;
第三步读取触摸屏数据。

18.3.1 打开触摸屏设备

使用 tslib 提供的库函数 `ts_open` 打开触摸屏设备, 其函数原型如下所示:

```
#include "tslib.h"
```

```
struct tsdev *ts_open(const char *dev_name, int nonblock);
```

参数 `dev_name` 指定了触摸屏的设备节点; 参数 `nonblock` 表示是否以非阻塞方式打开触摸屏设备, 如果 `nonblock` 等于 0 表示阻塞方式, 如果为非 0 值则表示以非阻塞方式打开。

调用成功返回一个 `struct tsdev *` 指针, 指向触摸屏设备句柄; 如果打开设备失败, 将返回 `NULL`。

除了使用 `ts_open()` 打开设备外, 还可以使用 `ts_setup()` 函数, 其函数原型如下所示:

```
#include "tslib.h"
```

```
struct tsdev *ts_setup(const char *dev_name, int nonblock)
```

参数 `dev_name` 指定触摸屏的设备节点, 与 `ts_open()` 函数中的 `dev_name` 参数意义相同; 但对于 `ts_setup()` 来说, 参数 `dev_name` 可以设置为 `NULL`, 当 `dev_name` 设置为 `NULL` 时, `ts_setup()` 函数内部会读取 `TSLIB_TSDEVICE` 环境变量, 获取该环境变量的内容以得知触摸屏的设备节点。

参数 `nonblock` 的意义与 `ts_open()` 函数的 `nonblock` 参数相同。

`ts_setup()` 相比 `ts_open()`, 除了打开触摸屏设备外, 还对触摸屏设备进行了配置, 也就是接下来说到的第二步操作。

关闭触摸屏设备使用 `ts_close()` 函数:

```
int ts_close(struct tsdev *);
```

18.3.2 配置触摸屏设备

调用 `ts_config()` 函数进行配置, 其函数原型如下所示:

```
#include "tslib.h"
```

```
int ts_config(struct tsdev *ts)
```

参数 `ts` 指向触摸屏句柄。

成功返回 0, 失败返回 -1。

所谓配置其实指的就是解析 `ts.conf` 文件中的配置信息, 加载相应的插件。

18.3.3 读取触摸屏数据

读取触摸屏数据使用 `ts_read()` 或 `ts_read_mt()` 函数, 区别在于 `ts_read` 用于读取单点触摸数据, 而 `ts_read_mt` 则用于读取多点触摸数据, 其函数原型如下所示:

```
#include "tslib.h"
```

```
int ts_read(struct tsdev *ts, struct ts_sample *samp, int nr)
```

```
int ts_read_mt(struct tsdev *ts, struct ts_sample_mt **samp, int max_slots, int nr)
```

参数 `ts` 指向一个触摸屏设备句柄, 参数 `nr` 表示对一个触摸点的采样数, 设置为 1 即可!

ts_read_mt()函数有一个 max_slots 参数,表示触摸屏支持的最大触摸点数,应用程序可以通过调用 ioctl()函数来获取触摸屏支持的最大触摸点数以及触摸屏坐标的最大分辨率等信息,稍后向大家介绍。

ts_read()函数的 samp 参数是一个 struct ts_sample *类型的指针,指向一个 struct ts_sample 对象,struct ts_sample 数据结构描述了触摸点的信息;调用 ts_read()函数获取到的数据会存放在 samp 指针所指向的内存中。struct ts_sample 结构体内容如下所示:

示例代码 18.3.1 struct ts_sample 结构体

```
struct ts_sample {
    int          x;           //X 坐标
    int          y;           //Y 坐标
    unsigned int pressure;    //按压力大小
    struct timeval tv;        //时间
};
```

ts_read_mt()函数的 samp 参数是一个 struct ts_sample_mt **类型的指针,多点触摸应用程序,每一个触摸点的信息使用 struct ts_sample_mt 数据结构来描述;一个触摸点的数据使用一个 struct ts_sample_mt 对象来装载,将它们组织成一个 struct ts_sample_mt 数组,调用 ts_read_mt()时,将数组地址赋值给 samp 参数。

struct ts_sample 结构体内容如下所示:

示例代码 18.3.2 struct ts_sample_mt 结构体

```
struct ts_sample_mt {
    /* ABS_MT_* event codes. linux/include/uapi/linux/input-event-codes.h
     * has the definitions.
     */
    int          x;           //X 坐标
    int          y;           //Y 坐标
    unsigned int pressure;    //按压力大小
    int          slot;        //触摸点 slot
    int          tracking_id;  //ID

    int          tool_type;
    int          tool_x;
    int          tool_y;
    unsigned int touch_major;
    unsigned int width_major;
    unsigned int touch_minor;
    unsigned int width_minor;
    int          orientation;
    int          distance;
    int          blob_id;

    struct timeval tv;        //时间

    /* BTN_TOUCH state */
    short        pen_down;    //BTN_TOUCH 的状态
};
```



```

/* valid is set != 0 if this sample
 * contains new data; see below for the
 * bits that get set.
 * valid is set to 0 otherwise
 */
short          valid;          //此次样本是否有效标志 触摸点数据是否发生更新
};

```

18.4 基于 tslib 编写触摸屏应用程序

通过对 tslib 库函数的简单介绍, 本小节来编写基于 tslib 的触摸屏应用程序, 包括单点触摸应用程序和多点触摸应用程序。

18.4.1 单点触摸应用程序

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->18_tslib->ts_read.c](#)。

示例代码 18.4.1 tslib 单点触摸程序

```

#include <stdio.h>
#include <stdlib.h>
#include <tslib.h>          //包含 tslib.h 头文件

int main(int argc, char *argv[])
{
    struct tsdev *ts = NULL;
    struct ts_sample samp;
    int pressure = 0; //用于保存上一次的按压力,初始为 0,表示松开

    /* 打开并配置触摸屏设备 */
    ts = ts_setup(NULL, 0);
    if (NULL == ts) {
        fprintf(stderr, "ts_setup error");
        exit(EXIT_FAILURE);
    }

    /* 读数据 */
    for (;;) {

        if (0 > ts_read(ts, &samp, 1)) {
            fprintf(stderr, "ts_read error");
            ts_close(ts);
            exit(EXIT_FAILURE);
        }

        if (samp.pressure) { //按压力>0
            if (pressure) //若上一次的按压力>0

```

```

        printf("移动(%d, %d)\n", samp.x, samp.y);
    else
        printf("按下(%d, %d)\n", samp.x, samp.y);
    }
    else
        printf("松开\n");//打印坐标

    pressure = samp.pressure;
}

ts_close(ts);
exit(EXIT_SUCCESS);
}

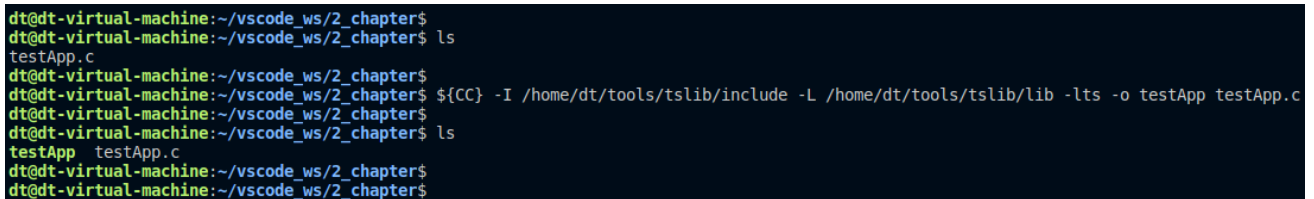
```

代码非常简单, 就不再解释了, 直接打开、配置设备, 接着读取数据即可! 通过判断按压力大小确定触摸的状态, 如果按压力等于 0 则表示手指已经松开; 按压力大于 0, 则需根据上一次的按压力是否大于 0 来判断。

读取数据出错时, `ts_read()` 返回一个负数。

接下来编译应用程序, 编译代码时, 需要通过交叉编译器来指定头文件、库文件的路径以及动态链接库文件名:

```
{CC} -I /home/dt/tools/tslib/include -L /home/dt/tools/tslib/lib -lts -o testApp testApp.c
```



```

dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -I /home/dt/tools/tslib/include -L /home/dt/tools/tslib/lib -lts -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 18.4.1 编译单点触摸应用程序

-I 选项指定头文件的路径, 也就是指定 `tslib` 安装目录下的 `include` 目录, 如果不指定头文件路径, 编译时将会找不到 `tslib.h` 头文件; -L 选项用于指定库文件的路径, 也就是指定 `tslib` 安装目录下的 `lib` 目录; 我们将 `tslib` 编译成了动态库文件, 以库文件的形式提供, 编译时需要链接到这些库文件; 而 -l 选项则用于指定链接库 (也可写成 -l ts, 也就是 `libts.so` 库文件, Linux 中, 动态库文件的命名方式为 `lib+名字+.so`)。

将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下, 执行应用程序, 进行测试:

```
root@ATK-IMX6U:~# ./testApp
按下 (210, 53)
松开
按下 (250, 41)
松开
按下 (401, 40)
移动 (399, 38)
移动 (398, 37)
移动 (398, 36)
移动 (397, 35)
移动 (396, 34)
移动 (395, 34)
移动 (395, 33)
移动 (394, 33)
移动 (394, 32)
移动 (393, 32)
移动 (393, 32)
移动 (392, 31)
移动 (391, 31)
移动 (391, 30)
移动 (390, 30)
移动 (390, 29)
移动 (389, 29)
松开
```

图 18.4.2 tslib 单点触摸应用程序测试

18.4.2 多点触摸应用程序

本小节我们来写基于 tslib 的多点触摸应用程序，使用 `ts_read_mt()` 函数读取多点触摸数据。

本例程源码对应的路径为：[开发板光盘->11、Linux C 应用编程例程源码->18_tslib->ts_read_mt.c](#)。

示例代码 18.4.2 tslib 多点触摸应用程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <linux/input.h>
#include <tslib.h>

int main(int argc, char *argv[])
{
    struct tsdev *ts = NULL;
    struct ts_sample_mt *mt_ptr = NULL;
    struct input_absinfo slot;
    int max_slots;
    unsigned int pressure[12] = {0}; //用于保存每一个触摸点上一次的按压力,初始为 0,表示松开
    int i;

    /* 打开并配置触摸屏设备 */
    ts = ts_setup(NULL, 0);
    if (NULL == ts) {
        fprintf(stderr, "ts_setup error");
        exit(EXIT_FAILURE);
    }
}
```

```
}

/* 获取触摸屏支持的最大触摸点数 */
if (0 > ioctl(ts_fd(ts), EVIOCGABS(ABS_MT_SLOT), &slot)) {
    perror("ioctl error");
    ts_close(ts);
    exit(EXIT_FAILURE);
}

max_slots = slot.maximum + 1 - slot.minimum;
printf("max_slots: %d\n", max_slots);

/* 内存分配 */
mt_ptr = calloc(max_slots, sizeof(struct ts_sample_mt));

/* 读数据 */
for (;;) {

    if (0 > ts_read_mt(ts, &mt_ptr, max_slots, 1)) {
        perror("ts_read_mt error");
        ts_close(ts);
        free(mt_ptr);
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < max_slots; i++) {

        if (mt_ptr[i].valid) { //有效表示有更新!
            if (mt_ptr[i].pressure) { //如果按压力>0
                if (pressure[mt_ptr[i].slot]) //如果上一次的按压力>0
                    printf("slot<%d>, 移动(%d, %d)\n", mt_ptr[i].slot, mt_ptr[i].x, mt_ptr[i].y);
                else
                    printf("slot<%d>, 按下(%d, %d)\n", mt_ptr[i].slot, mt_ptr[i].x, mt_ptr[i].y);
            }
            else
                printf("slot<%d>, 松开\n", mt_ptr[i].slot);

            pressure[mt_ptr[i].slot] = mt_ptr[i].pressure;
        }
    }
}

/* 关闭设备、释放内存、退出 */
```

```

    ts_close(ts);
    free(mt_ptr);
    exit(EXIT_SUCCESS);
}

```

整个思路与单点触摸应用程序相同, 关注 for() 循环内部, 通过 ts_read_mt() 函数读取触摸点数据, 将这些数据存放在 mt_ptr 数组中, 接着在 for() 循环中判断每一个触摸点数据是否有效, 有效则表示该触摸点信息发生更新。

编译应用程序:

```

${CC} -I /home/dt/tools/tslib/include -L /home/dt/tools/tslib/lib -lts -o testApp testApp.c

```

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -I /home/dt/tools/tslib/include -L /home/dt/tools/tslib/lib -lts -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 18.4.3 编译多点触摸应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下, 执行程序:

```

root@ATK-IMX6U:~# ./testApp
max_slots: 5
slot<0>, 按下 (655, 362)
slot<1>, 按下 (201, 249)
slot<1>, 松开
slot<1>, 按下 (201, 250)
slot<1>, 松开
slot<1>, 按下 (200, 247)
slot<2>, 按下 (621, 224)
slot<1>, 移动 (201, 247)
slot<1>, 移动 (203, 247)
slot<2>, 移动 (621, 223)
slot<0>, 移动 (655, 361)
slot<1>, 移动 (204, 247)
slot<2>, 移动 (622, 223)
slot<1>, 移动 (205, 247)
slot<1>, 移动 (206, 247)
slot<1>, 移动 (207, 247)
slot<1>, 移动 (208, 247)
slot<2>, 移动 (623, 223)
slot<0>, 移动 (655, 361)
slot<0>, 移动 (655, 361)
slot<2>, 移动 (624, 223)
slot<2>, 移动 (625, 223)
slot<2>, 移动 (626, 222)
slot<2>, 松开
slot<0>, 松开
slot<1>, 移动 (209, 247)
slot<1>, 移动 (208, 247)
slot<1>, 移动 (208, 247)
slot<1>, 移动 (207, 247)
slot<1>, 松开

```

图 18.4.4 多点触摸应用程序测试结果

slot<0>表示触摸点 0, slot<1>表示触摸点 1, 以此类推, 大家自己去测试, 没什么可说的!

第十九章 FrameBuffer 应用编程

本章学习 Linux 下的 Framebuffer 应用编程, 通过对本章内容的学习, 大家将会了解到 Framebuffer 设备究竟是什么? 以及如何编写应用程序来操控 FrameBuffer 设备。

本章将会讨论如下主题。

- 什么是 Framebuffer 设备?
- LCD 显示的基本原理;
- 使用存储映射 I/O 方式编写 LCD 应用程序。
- 在 LCD 上打点、画线;
- BMP 图片格式详解;
- 在 LCD 上显示图片;

19.1 什么是 FrameBuffer

Frame 是帧的意思, buffer 是缓冲的意思, 所以 Framebuffer 就是帧缓冲, 这意味着 Framebuffer 就是一块内存, 里面保存着一帧图像。帧缓冲 (framebuffer) 是 Linux 系统中的一种显示驱动接口, 它将显示设备 (譬如 LCD) 进行抽象、屏蔽了不同显示设备硬件的实现, 对应用层抽象为一块显示内存 (显存), 它允许上层应用程序直接对显示缓冲区进行读写操作, 而用户不必关心物理显存的位置等具体细节, 这些都由 Framebuffer 设备驱动来完成。

所以在 Linux 系统中, 显示设备被称为 FrameBuffer 设备 (帧缓冲设备), 所以 LCD 显示屏自然而然就是 FrameBuffer 设备。FrameBuffer 设备对应的设备文件为 /dev/fbX (X 为数字, 0、1、2、3 等), Linux 下可支持多个 FrameBuffer 设备, 最多可达 32 个, 分别为 /dev/fb0 到 /dev/fb31, 开发板出厂系统中, /dev/fb0 设备节点便是 LCD 屏。

应用程序读写 /dev/fbX 就相当于读写显示设备的显示缓冲区 (显存), 譬如 LCD 的分辨率是 800*480, 每一个像素点的颜色用 24 位 (譬如 RGB888) 来表示, 那么这个显示缓冲区的大小就是 $800 \times 480 \times 24 / 8 = 1152000$ 个字节。譬如执行下面这条命令将 LCD 清屏, 也就是将其填充为黑色 (假设 LCD 对应的设备节点是 /dev/fb0, 分辨率为 800*480, RGB888 格式):

```
dd if=/dev/zero of=/dev/fb0 bs=1024 count=1125
```

这条命令的作用就是将 1125x1024 个字节数据全部写入到 LCD 显存中, 并且这些数据都是 0x0。

19.2 LCD 的基础知识

关于 LCD 相关的基础知识, 本书不再介绍, 开发板配套提供的驱动教程中已经有过详细的介绍, 除此之外, 网络上也能找到相关内容。

19.3 LCD 应用编程介绍

本小节介绍如何对 FrameBuffer 设备 (譬如 LCD) 进行应用编程, 通过上面的介绍, 相信大家应该已经知道如何操作 LCD 显示设备了, 应用程序通过对 LCD 设备节点 /dev/fb0 (假设 LCD 对应的设备节点是 /dev/fb0) 进行 I/O 操作即可实现对 LCD 的显示控制, 实质就相当于读写了 LCD 的显存, 而显存是 LCD 的显示缓冲区, LCD 硬件会从显存中读取数据显示到 LCD 液晶面板上。

在应用程序中, 操作 /dev/fbX 的一般步骤如下:

- ①、首先打开 /dev/fbX 设备文件。
- ②、使用 ioctl() 函数获取到当前显示设备的参数信息, 譬如屏幕的分辨率大小、像素格式, 根据屏幕参数计算显示缓冲区的大小。
- ③、通过存储映射 I/O 方式将屏幕的显示缓冲区映射到用户空间 (mmap)。
- ④、映射成功后就可以直接读写屏幕的显示缓冲区, 进行绘图或图片显示等操作了。
- ⑤、完成显示后, 调用 munmap() 取消映射、并调用 close() 关闭设备文件。

从上面介绍的操作步骤来看, LCD 的应用编程还是非常简单的, 这些知识点都是在前面的入门篇中给大家介绍过。

19.3.1 使用 ioctl() 获取屏幕参数信息

当打开 LCD 设备文件之后, 需要先获取到 LCD 屏幕的参数信息, 譬如 LCD 的 X 轴分辨率、Y 轴分辨率以及像素格式等信息, 通过这些参数计算出 LCD 显示缓冲区的大小。

通过 ioctl() 函数来获取屏幕参数信息, 对于 Framebuffer 设备来说, 常用的 request 包括 FBIOPUT_VSCREENINFO、FBIOPUT_VSCREENINFO、FBIOPUT_VSCREENINFO。

- **FBIOGET_VSCREENINFO**: 表示获取 FrameBuffer 设备的可变参数信息, 可变参数信息使用 struct fb_var_screeninfo 结构体来描述, 所以此时 ioctl() 需要有第三个参数, 它是一个 struct fb_var_screeninfo * 指针, 指向 struct fb_var_screeninfo 类型对象, 调用 ioctl() 会将 LCD 屏的可变参数信息保存在 struct fb_var_screeninfo 类型对象中, 如下所示:

```
struct fb_var_screeninfo fb_var;
```

```
ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
```

- **FBIOPUT_VSCREENINFO**: 表示设置 FrameBuffer 设备的可变参数信息, 既然是可变参数, 那说明应用层可对其进行修改、重新配置, 当然前提条件是底层驱动支持这些参数的动态调整, 譬如在我们的 Windows 系统中, 用户可以修改屏幕的显示分辨率, 这就是一种动态调整。同样此时 ioctl() 需要有第三个参数, 也是一个 struct fb_var_screeninfo * 指针, 指向 struct fb_var_screeninfo 类型对象, 表示用 struct fb_var_screeninfo 对象中填充的数据设置 LCD, 如下所示:

```
struct fb_var_screeninfo fb_var = {0};
```

```
/* 对 fb_var 进行数据填充 */
```

```
.....
```

```
.....
```

```
/* 设置可变参数信息 */
```

```
ioctl(fd, FBIOPUT_VSCREENINFO, &fb_var);
```

- **FBIOGET_FSCREENINFO**: 表示获取 FrameBuffer 设备的固定参数信息, 既然是固定参数, 那就意味着应用程序不可修改。固定参数信息使用 struct fb_fix_screeninfo 结构体来描述, 所以此时 ioctl() 需要有第三个参数, 它是一个 struct fb_fix_screeninfo * 指针, 指向 struct fb_fix_screeninfo 类型对象, 调用 ioctl() 会将 LCD 的固定参数信息保存在 struct fb_fix_screeninfo 对象中, 如下所示:

```
struct fb_fix_screeninfo fb_fix;
```

```
ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);
```

上面所提到的三个宏定义 FBIOGET_VSCREENINFO、FBIOPUT_VSCREENINFO、FBIOGET_FSCREENINFO 以及 2 个数据结构 struct fb_var_screeninfo 和 struct fb_fix_screeninfo 都定义在 <linux/fb.h> 头文件中, 所以在我们的应用程序中需要包含该头文件。

```
#define FBIOGET_VSCREENINFO 0x4600
```

```
#define FBIOPUT_VSCREENINFO 0x4601
```

```
#define FBIOGET_FSCREENINFO 0x4602
```

struct fb_var_screeninfo 结构体

struct fb_var_screeninfo 结构体内容如下所示:

示例代码 19.3.1 struct fb_var_screeninfo 结构体

```
struct fb_var_screeninfo {
    __u32 xres;           /* 可视区域, 一行有多少个像素点, X 分辨率 */
    __u32 yres;           /* 可视区域, 一列有多少个像素点, Y 分辨率 */
    __u32 xres_virtual;   /* 虚拟区域, 一行有多少个像素点 */
    __u32 yres_virtual;   /* 虚拟区域, 一列有多少个像素点 */
    __u32 xoffset;        /* 虚拟到可见屏幕之间的行偏移 */
    __u32 yoffset;        /* 虚拟到可见屏幕之间的列偏移 */
};
```



```

__u32 bits_per_pixel; /* 每个像素点使用多少个 bit 来描述, 也就是像素深度 bpp */
__u32 grayscale;     /* =0 表示彩色, =1 表示灰度, >1 表示 FOURCC 颜色 */

/* 用于描述 R、G、B 三种颜色分量分别用多少位来表示以及它们各自的偏移量 */
struct fb_bitfield red;      /* Red 颜色分量色域偏移 */
struct fb_bitfield green;   /* Green 颜色分量色域偏移 */
struct fb_bitfield blue;    /* Blue 颜色分量色域偏移 */
struct fb_bitfield transp;  /* 透明度分量色域偏移 */

__u32 nonstd;          /* nonstd 等于 0, 表示标准像素格式; 不等于 0 则表示非标准像素格式 */
__u32 activate;

__u32 height;         /* 用来描述 LCD 屏显示图像的高度 (以毫米为单位) */
__u32 width;          /* 用来描述 LCD 屏显示图像的宽度 (以毫米为单位) */

__u32 accel_flags;

/* 以下这些变量表示时序参数 */
__u32 pixclock;       /* pixel clock in ps (pico seconds) */
__u32 left_margin;   /* time from sync to picture */
__u32 right_margin;  /* time from picture to sync */
__u32 upper_margin;  /* time from sync to picture */
__u32 lower_margin;

__u32 hsync_len;     /* length of horizontal sync */
__u32 vsync_len;     /* length of vertical sync */
__u32 sync;          /* see FB_SYNC_* */
__u32 vmode;         /* see FB_VMODE_* */
__u32 rotate;        /* angle we rotate counter clockwise */
__u32 colorspace;    /* colorspace for FOURCC-based modes */
__u32 reserved[4];   /* Reserved for future compatibility */
};

```

通过 `xres`、`yres` 获取到屏幕的水平分辨率和垂直分辨率, `bits_per_pixel` 表示像素深度 `bpp`, 即每一个像素点使用多少个 `bit` 位来描述它的颜色, 通过 `xres * yres * bits_per_pixel / 8` 计算可得到整个显示缓存区的大小。

`red`、`green`、`blue` 描述了 `RGB` 颜色值中 `R`、`G`、`B` 三种颜色通道分别使用多少 `bit` 来表示以及它们各自的偏移量, 通过 `red`、`green`、`blue` 变量可知道 `LCD` 的 `RGB` 像素格式, 譬如 `RGB888` 还是 `RGB565`, 亦或者是 `BGR888`、`BGR565` 等。结构体 `fb_bitfield` 结构体如下所示:

示例代码 19.3.2 struct fb_bitfield 结构体

```

struct fb_bitfield {
    __u32 offset; /* 偏移量 */
    __u32 length; /* 长度 */
    __u32 msb_right; /* != 0 : Most significant bit is right */
};

```

};

struct fb_fix_screeninfo 结构体

struct fb_fix_screeninfo 结构体内容如下所示:

示例代码 19.3.3 struct fb_fix_screeninfo 结构体

```

struct fb_fix_screeninfo {
    char id[16];           /* 字符串形式的标识符 */
    unsigned long smem_start; /* 显存的起始地址 (物理地址) */

    __u32 smem_len;       /* 显存的长度 */
    __u32 type;
    __u32 type_aux;
    __u32 visual;
    __u16 xpanstep;
    __u16 ypanstep;
    __u16 ywrapstep;
    __u32 line_length;    /* 一行的字节数 */
    unsigned long mmio_start; /* Start of Memory Mapped I/O(physical address) */
    __u32 mmio_len;       /* Length of Memory Mapped I/O */
    __u32 accel;          /* Indicate to driver which specific chip/card we have */
    __u16 capabilities;
    __u16 reserved[2];
};

```

smem_start 表示显存的起始地址, 这是一个物理地址, 当然在应用层无法直接使用; smem_len 表示显存的长度, 这个长度并不一定等于 LCD 实际的显存大小。line_length 表示屏幕的一行像素点有多少个字节, 通常可以使用 line_length * yres 来得到屏幕显示缓冲区的大小。

通过上面介绍, 接下来我们编写一个示例代码, 获取 LCD 屏幕的参数信息, 示例代码如下所示:

本例程源码对应的路径为: **开发板光盘->11、Linux C 应用编程例程源码->19_lcd->lcd_info.c。**

示例代码 19.3.4 获取屏幕的参数信息

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/fb.h>

int main(int argc, char *argv[])
{
    struct fb_fix_screeninfo fb_fix;
    struct fb_var_screeninfo fb_var;
    int fd;
}

```

```
/* 打开 framebuffer 设备 */
if (0 > (fd = open("/dev/fb0", O_WRONLY))) {
    perror("open error");
    exit(-1);
}

/* 获取参数信息 */
ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);
printf("分辨率: %d*%d\n"
       "像素深度 bpp: %d\n"
       "一行的字节数: %d\n"
       "像素格式: R<%d %d> G<%d %d> B<%d %d>\n",
       fb_var.xres, fb_var.yres, fb_var.bits_per_pixel,
       fb_fix.line_length,
       fb_var.red.offset, fb_var.red.length,
       fb_var.green.offset, fb_var.green.length,
       fb_var.blue.offset, fb_var.blue.length);

/* 关闭设备文件退出程序 */
close(fd);
exit(0);
}
```

首先打开 LCD 设备文件，开发板出厂系统，LCD 对应的设备文件为/dev/fb0；打开设备文件之后得到文件描述符 fd，接着使用 ioctl()函数获取 LCD 的可变参数信息和固定参数信息，并将这些信息打印出来。

在测试之前，需将 LCD 屏通过软排线连接到开发板（掉电情况下连接），连接好之后启动开发板。

使用交叉编译工具编译上述示例代码，将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下，并直接运行它，如下所示：

```
root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp
分辨率: 800*480
像素深度bpp: 16
一行的字节数: 1600
像素格式: R<11 5> G<5 6> B<0 5>
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 19.3.1 获取到屏幕参数

笔者使用的是 7 寸 800*480 RGB 屏，与上图打印显示的分辨率 800*480 是相符的；像素深度为 16，也就意味着一个像素点的颜色值将使用 16bit（也就是 2 个字节）来表示；一行的字节数为 1600，一行共有 800 个像素点，每个像素点使用 16bit 来描述，一共就是 $800*16/8=1600$ 个字节数据，这也是没问题的。

打印出像素格式为 R<11 5>G<5 6>B<0 5>，分别表示 R、G、B 三种颜色分量对应的偏移量和长度，第一个数字表示偏移量，第二个参数为长度，从打印的结果可知，16bit 颜色值中高 5 位表示 R 颜色通道、中间 6 位表示 G 颜色通道、低 5 位表示 B 颜色通道，所以这是一个 RGB565 格式的显示设备。

Tips: 正点原子的 RGB LCD 屏幕, 包括 4.3 寸 800*480、4.3 寸 480*272、7 寸 800*480、7 寸 1024*600 以及 10.1 寸 1280*800 硬件上均支持 RGB888, 但 ALPHA/Mini I.MX6U 开发板出厂系统中, LCD 驱动程序将其实现为一个 RGB565 格式的显示设备, 用户可修改设备树使其支持 RGB888, 或者通过 ioctl 修改。

前面我们提到可以通过 ioctl() 去设置 LCD 的可变参数, 使用 FBIOPUT_VSCREENINFO 宏, 但不太建议大家去改这些参数, 如果 FrameBuffer 驱动程序支持不够完善, 改完之后可能会出现一些问题! 这里就不再演示了。

19.3.2 使用 mmap() 将显示缓冲区映射到用户空间

在入门篇 13.5 小节中给大家介绍了存储映射 I/O 这种高级 I/O 方式, 它的一个非常经典的使用场景便是用在 Framebuffer 应用编程中。通过 mmap() 将显示器的显示缓冲区 (显存) 映射到进程的地址空间中, 这样应用程序便可直接对显示缓冲区进行读写操作。

为什么这里需要使用存储映射 I/O 这种方式呢? 其实使用普通的 I/O 方式 (譬如直接 read、write) 也是可以的, 只是, 当数据量比较大时, 普通 I/O 方式效率较低。假设某一显示器的分辨率为 1920 * 1080, 像素格式为 ARGB8888, 针对该显示器, 刷一帧图像的数据量为 $1920 \times 1080 \times 32 / 8 = 8294400$ 个字节 (约等于 8MB), 这还只是一帧的图像数据, 而对于显示器来说, 显示的图像往往是动态改变的, 意味着图像数据会被不断更新。

在这种情况下, 数据量是比较庞大的, 使用普通 I/O 方式必然导致效率低下, 所以才会采用存储映射 I/O 方式。

19.4 LCD 应用编程练习之 LCD 基本操作

本小节编写应用程序, 在 LCD 上实现画点 (俗称打点)、画线、画矩形等基本 LCD 操作, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->19_lcd->lcd_test.c](#)。

示例代码 19.4.1 LCD 画点、画线、画矩形操作

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : lcd_test.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

```
描述 : FrameBuffer 应用程序示例代码
```

```
其他 : 无
```

```
论坛 : www.openedv.com
```

```
日志 : 初版 V1.0 2021/6/15 邓涛创建
```

```
*****/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
```

```

#include <linux/fb.h>

#define argb8888_to_rgb565(color)  ({ \
    unsigned int temp = (color); \
    ((temp & 0xF80000UL) >> 8) | \
    ((temp & 0xFC00UL) >> 5) | \
    ((temp & 0xF8UL) >> 3); \
})

static int width;           //LCD X 分辨率
static int height;         //LCD Y 分辨率
static unsigned short *screen_base = NULL; //映射后的显存基地址

/*****
 * 函数名称:  lcd_draw_point
 * 功能描述:  打点
 * 输入参数:  x, y, color
 * 返回值:  无
 *****/
static void lcd_draw_point(unsigned int x, unsigned int y, unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值

    /* 对传入参数的校验 */
    if (x >= width)
        x = width - 1;
    if (y >= height)
        y = height - 1;

    /* 填充颜色 */
    screen_base[y * width + x] = rgb565_color;
}

/*****
 * 函数名称:  lcd_draw_line
 * 功能描述:  画线 (水平或垂直线)
 * 输入参数:  x, y, dir, length, color
 * 返回值:  无
 *****/
static void lcd_draw_line(unsigned int x, unsigned int y, int dir,
                          unsigned int length, unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值

```

```

unsigned int end;
unsigned long temp;

/* 对传入参数的校验 */
if (x >= width)
    x = width - 1;
if (y >= height)
    y = height - 1;

/* 填充颜色 */
temp = y * width + x; //定位到起点
if (dir) { //水平线
    end = x + length - 1;
    if (end >= width)
        end = width - 1;

    for ( ; x <= end; x++, temp++)
        screen_base[temp] = rgb565_color;
}
else { //垂直线
    end = y + length - 1;
    if (end >= height)
        end = height - 1;

    for ( ; y <= end; y++, temp += width)
        screen_base[temp] = rgb565_color;
}
}

/*****
* 函数名称: lcd_draw_rectangle
* 功能描述: 画矩形
* 输入参数: start_x, end_x, start_y, end_y, color
* 返回值: 无
*****/
static void lcd_draw_rectangle(unsigned int start_x, unsigned int end_x,
                               unsigned int start_y, unsigned int end_y,
                               unsigned int color)
{
    int x_len = end_x - start_x + 1;
    int y_len = end_y - start_y + 1;

    lcd_draw_line(start_x, start_y, 1, x_len, color); //上边

```

```

    lcd_draw_line(start_x, end_y, 1, x_len, color); //下边
    lcd_draw_line(start_x, start_y + 1, 0, y_len, color); //左边
    lcd_draw_line(end_x, start_y + 1, 0, y_len, color); //右边
}

/*****
* 函数名称: lcd_fill
* 功能描述: 将一个矩形区域填充为参数 color 所指定的颜色
* 输入参数: start_x, end_x, start_y, end_y, color
* 返回值: 无
*****/
static void lcd_fill(unsigned int start_x, unsigned int end_x,
                    unsigned int start_y, unsigned int end_y,
                    unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值
    unsigned long temp;
    unsigned int x;

    /* 对传入参数的校验 */
    if (end_x >= width)
        end_x = width - 1;
    if (end_y >= height)
        end_y = height - 1;

    /* 填充颜色 */
    temp = start_y * width; //定位到起点行首
    for (; start_y <= end_y; start_y++, temp+=width) {

        for (x = start_x; x <= end_x; x++)
            screen_base[temp + x] = rgb565_color;
    }
}

int main(int argc, char *argv[])
{
    struct fb_fix_screeninfo fb_fix;
    struct fb_var_screeninfo fb_var;
    unsigned int screen_size;
    int fd;

    /* 打开 framebuffer 设备 */
    if (0 > (fd = open("/dev/fb0", O_RDWR))) {

```

```
    perror("open error");
    exit(EXIT_FAILURE);
}

/* 获取参数信息 */
ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

screen_size = fb_fix.line_length * fb_var.yres;
width = fb_var.xres;
height = fb_var.yres;

/* 将显示缓冲区映射到进程地址空间 */
screen_base = mmap(NULL, screen_size, PROT_WRITE, MAP_SHARED, fd, 0);
if (MAP_FAILED == (void *)screen_base) {
    perror("mmap error");
    close(fd);
    exit(EXIT_FAILURE);
}

/* 画正方形方块 */
int w = height * 0.25; // 方块的宽度为 1/4 屏幕高度
lcd_fill(0, width-1, 0, height-1, 0x0); // 清屏 (屏幕显示黑色)
lcd_fill(0, w, 0, w, 0xFF0000); // 红色方块
lcd_fill(width-w, width-1, 0, w, 0xFF00); // 绿色方块
lcd_fill(0, w, height-w, height-1, 0xFF); // 蓝色方块
lcd_fill(width-w, width-1, height-w, height-1, 0xFFFF00); // 黄色方块

/* 画线: 十字交叉线 */
lcd_draw_line(0, height * 0.5, 1, width, 0xFFFFFFFF); // 白色线
lcd_draw_line(width * 0.5, 0, 0, height, 0xFFFFFFFF); // 白色线

/* 画矩形 */
unsigned int s_x, s_y, e_x, e_y;
s_x = 0.25 * width;
s_y = w;
e_x = width - s_x;
e_y = height - s_y;

for (; (s_x <= e_x) && (s_y <= e_y);
      s_x+=5, s_y+=5, e_x-=5, e_y-=5)
    lcd_draw_rectangle(s_x, e_x, s_y, e_y, 0xFFFFFFFF);
```



```

/* 退出 */
munmap(screen_base, screen_size); //取消映射
close(fd); //关闭文件
exit(EXIT_SUCCESS); //退出进程
}

```

在示例代码中定义了一个宏 `argb8888_to_rgb565`，用于实现将 `unsigned int` 类型的颜色（也就是 ARGB8888 颜色）转换为 RGB565 颜色。

程序中自定义了 4 个函数：

lcd_draw_point: 用于实现画点、打点操作，参数 `x` 和 `y` 指定像素点的位置，参数 `color` 表示颜色。

lcd_draw_line: 用于实现画线操作，参数 `x` 和 `y` 指定线的起始位置；参数 `dir` 表示方向，水平方向(`dir!=0`) 还是垂直方向 (`dir=0`)，不支持斜线画法，画斜线需要一些算法去操作，这不是本章内容需要去关注的知识点；参数 `length` 表示线的长度，以像素为单位；参数 `color` 表示线条的颜色。

lcd_draw_rectangle: 用于实现画矩形操作，参数 `start_x` 和 `start_y` 指定矩形左上角的位置；参数 `end_x` 和 `end_y` 指定矩形右下角的位置；参数 `color` 指定矩形 4 个边的线条颜色。

lcd_fill: 将一个指定的矩形区域填充为参数 `color` 指定的颜色，参数 `start_x` 和 `start_y` 指定矩形左上角的位置；参数 `end_x` 和 `end_y` 指定矩形右下角的位置；参数 `color` 指定矩形区域填充的颜色。

具体代码的实现各位读者自己去看，非常简单，来看下 `main()` 中做了哪些事情：

- 首先调用 `open()` 打开 LCD 设备文件得到文件描述符 `fd`；
- 接着使用 `ioctl` 函数获取 LCD 的可变参数信息和固定参数信息，通过得到的信息计算 LCD 显存大小、得到 LCD 屏幕的分辨率，从图 19.3.1 可知，ALPHA/Mini I.MX6U 开发板出厂系统将 LCD 实现为一个 RGB565 显示设备，所以程序中自定义的 4 个函数在操作 LCD 像素点时、都是以 RGB565 的格式写入颜色值。
- 接着使用 `mmap` 建立映射；
- 映射成功之后就可以在应用层直接操作 LCD 显存了，调用自定义的函数在 LCD 上画线、画矩形、画方块；
- 操作完成之后，调用 `munmap` 取消映射，调用 `close` 关闭 LCD 设备文件，退出程序。

编译应用程序：

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 19.4.1 编译 LCD 测试程序

将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下，执行应用程序（在测试之前，先将出厂系统对应的 Qt GUI 应用程序退出）：

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls  
driver  shell  testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#
```

图 19.4.2 执行应用程序

此时 LCD 屏上将会显示程序中绘制的方块、矩形、以及线条:

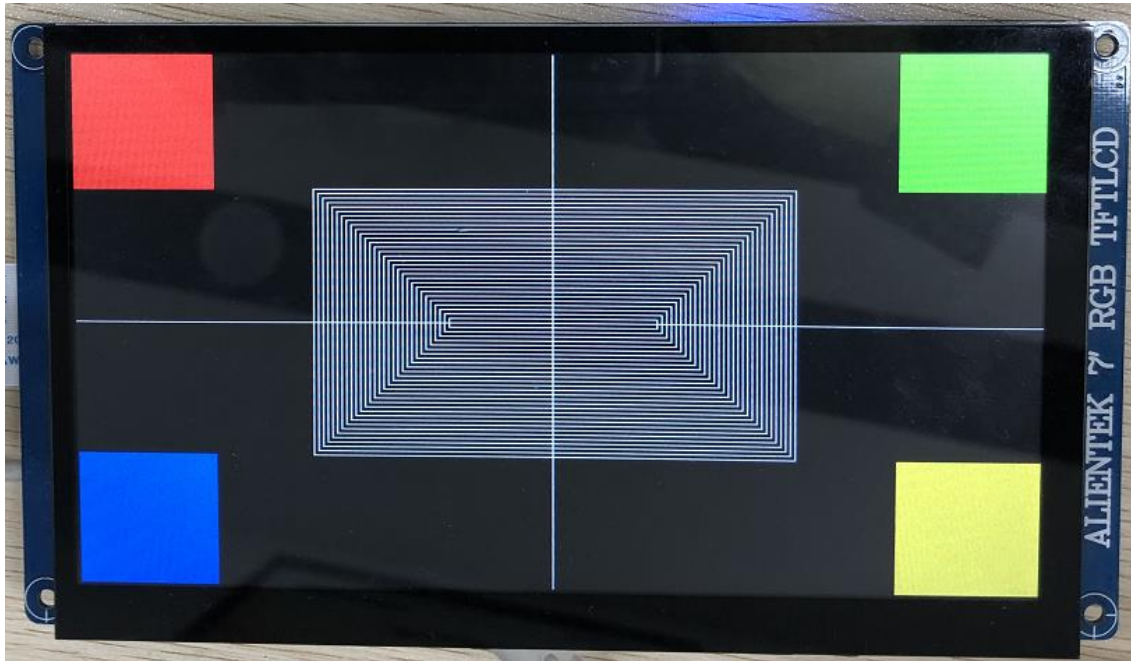


图 19.4.3 LCD 显示的效果

忽略手机拍摄的问题, 实际效果各位读者运行程序便知。

19.5 LCD 应用编程练习之显示 BMP 图片

本小节介绍如何在 LCD 上显示一张 BMP 图片, 在编写程序之前, 首先需要对 BMP 格式图片进行简单地介绍。

19.5.1 BMP 图像介绍

我们常用的图片格式有很多, 一般最常用的有三种: JPEG (或 JPG)、PNG、BMP 和 GIF。其中 JPEG (或 JPG)、PNG 以及 BMP 都是静态图片, 而 GIF 则可以实现动态图片。在本小节实验中, 我们选择使用 BMP 图片格式。

BMP (全称 Bitmap) 是 Window 操作系统中的标准图像文件格式, 文件后缀名为 “.bmp”, 使用非常广。它采用位映射存储格式, 除了图像深度可选以外, 图像数据没有进行任何压缩, 因此, BMP 图像文件所占用的空间很大, 但是没有失真、并且解析 BMP 图像简单。

BMP 文件的图像深度可选 1bit、4bit、8bit、16bit、24bit 以及 32bit, 典型的 BMP 图像文件由四部分组成:

- ①、BMP 文件头 (BMP file header), 它包含 BMP 文件的格式、大小、位图数据的偏移量等信息;
- ②、位图信息头 (bitmap information), 它包含位图信息头大小、图像的尺寸、图像大小、位平面数、压缩方式以及颜色索引等信息;

③、调色板 (color palette)，这部分是可选的，如果使用索引来表示图像，调色板就是索引与其对应颜色的映射表；

④、位图数据 (bitmap data)，也就是图像数据。

BMP 文件头、位图信息头、调色板和位图数据，总结如下表所示：

数据段名称	大小 (Byte)	说明
bmp 文件头 (bmp file header)	14	包含 BMP 文件的格式、大小、到位图数据的偏移量等信息
位图信息头 (bitmap information)	通常为 40 或 56 字节	包含位图信息头大小、图像的尺寸、图像大小、位平面数、压缩方式以及颜色索引等信息；
调色板 (color palette)	由颜色索引数决定	可选，如果使用索引来表示图像的颜色，则调色板就是索引与其对应颜色的映射表；
位图数据 (bitmap data)	由图像尺寸决定	图像数据

表 19.5.1 BMP 图像各数据段说明

一般常见的图像都是以 16 位 (R、G、B 三种颜色分别使用 5bit、6bit、5bit 来表示)、24 位 (R、G、B 三种颜色都使用 8bit 来表示) 色图像为主，我们称这样的图像为真彩色图像，真彩色图像是不需要调色板的，即位图信息头后面紧跟的就是位图数据了。

对某些 BMP 位图文件说并非如此，譬如 16 色位图、256 色位图，它们需要使用到调色板，具体调色板如何使用，我们不关心，本节我们将会以 16 位色 (RGB565) BMP 图像为例。

以一张 16 位 BMP 图像为例 (如何的到 16 位色 BMP 图像，后面向大家介绍)，如下图所示：



图 19.5.1 16 位 BMP 示例图片

首先在 Windows 下查看该图片的属性，如下所示：

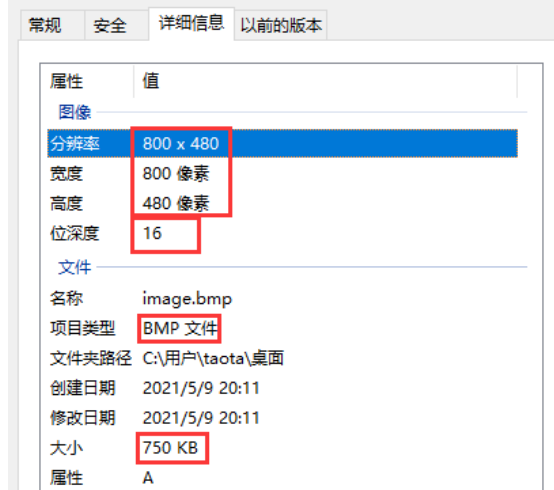


图 19.5.2 示例图片属性

可以看到该图片的分辨率为 800*480，位深度为 16bit，每个像素点使用 16 位表示，也就是 RGB565。为了向大家介绍 BMP 文件结构，接下来使用十六进制查看工具将 image.bmp 文件打开，文件头部分的内容如下所示：

image.bmp	Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
	00000000	42	4D	48	B8	0B	00	00	00	00	00	46	00	00	00	38	00	BMH, F 8
	00000016	00	00	20	03	00	00	E0	01	00	00	01	00	10	00	03	00	à
	00000032	00	00	02	B8	0B	00	C2	0E	00	00	C2	0E	00	00	00	00	, Å Å
	00000048	00	00	00	00	00	00	F8	00	00	E0	07	00	00	1F	00		ø à
	00000064	00	00	00	00	00	00	48	60	68	60	78	00	60	00	58		H`h`x`X
	00000080	00	68	00	68	00	68	00	70	00	90	00	80	00	80	00	98	h h p e e ~
	00000096	00	88	00	70	A0	80	C0	90	20	88	00	88	00	90	00	90	^ p eÀ ^ ^
	00000112	00	90	20	B0	40	C0	00	A8	CF	99	BC	F5	57	FE	80	D8	°@À "iP"õwpeø
	00000128	00	A8	00	A0	00	98	20	98	60	A8	20	A0	40	A0	60	B8	" " " " " " " "
	00000144	A0	C0	60	C0	21	C0	E1	C0	2D	E4	D5	FD	54	EC	65	E2	À`À!ÀÀÀ-äÖYIeá
	00000160	40	B0	00	A0	00	A0	00	A8	00	A8	00	A0	00	98	00	A8	@° " " " " " "
	00000176	00	B8	00	A8	00	A0	00	A0	00	98	00	90	60	90	40	80	, " " " " " " @e
	00000192	00	88	00	90	00	98	20	A8	60	A8	20	98	00	88	00	90	^ " " " " " " ^

图 19.5.3 image.bmp 文件的十六进制数据

一、bmp 文件头

Windows 下为 bmp 文件头定义了如下结构体：

```
typedef struct tagBITMAPFILEHEADER
{
    UINT16 bfType;
    DWORD bfSize;
    UINT16 bfReserved1;
    UINT16 bfReserved2;
    DWORD bfOffBits;
} BITMAPFILEHEADER;
```

结构体中每一个成员说明如下：

变量名	地址偏移	大小	作用
bfType	00H	2 bytes	说明 bmp 文件的类型，可取值为： ①BM – Windows ②BA – OS/2 Bitmap Array ③CI – OS/2 Color Icon

			④CP – OS/2 Color Pointer ⑤IC – OS/2 Icon ⑥PT – OS/2 Pointer
bfSize	02H	4 bytes	说明该文件的大小，以字节为单位。
bfReserved1	06H	2 bytes	保留字段，必须设置为0。
bfReserved2	08H	2 bytes	保留字段，必须设置为0。
bfOffBits	0AH	4 bytes	说明从文件起始位置到图像数据之间的字节偏移量。这个参数非常有用，因为位图信息头和调色板的长度会根据不同的情况而变化，所以我们可以用这个偏移量迅速从文件中找到图像数据的偏移地址。

表 19.5.2 bmp 文件头成员说明

从上面的描述信息，再来对照文件数据：

image.bmp	Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI	ASCII
	00000000	42	4D	48	B8	0B	00	00	00	00	00	4E	00	00	00	00	38	BM	, F 8
	00000016	00	00	20	03	00	00	E0	01	00	00	01	00	10	00	03	00		à
	00000032	00	00	02	B8	0B	00	C2	0E	00	00	C2	0E	00	00	00	00		À Á
	00000048	00	00	00	00	00	00	F8	00	00	E0	07	00	00	1F	00			ø à
	00000064	00	00	00	00	00	00	48	60	68	60	78	00	60	00	58			H`h`x`X
	00000080	00	68	00	68	00	68	00	70	00	90	00	80	00	80	00	98		h h h p € € ~
	00000096	00	88	00	70	A0	80	C0	90	20	88	00	88	00	90	00	90		^ p €À ^ ^
	00000112	00	90	20	B0	40	C0	00	A8	CF	99	BC	F5	57	FE	80	D8		°@À ..I"4δWpεø

图 19.5.4 bmp 文件头数据

00~01H: 0x42、0x4D 对应的 ASCII 字符分别为 B、M，表示这是 Windows 所支持的位图格式，该字段必须是“BM”才是 Windows 位图文件。

02~05H: 对应于文件大小，0x000BB848=768072 字节，与 image.bmp 文件大小是相符的。

06~09H: 保留字段。

0A~0D: 0x00000046=70，即从文件头部开始到位图数据需要偏移 70 个字节。

bmp 文件头的大小固定为 14 个字节。

二、位图信息头

同样，Windows 下为位图信息头定义了如下结构体：

```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG biXPelsPerMeter;
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

结构体中每一个成员说明如下：

变量名	地址偏移	大小	作用
biSize	0EH	4 bytes	位图信息头大小。
biWidth	12H	4 bytes	图像的宽度, 以像素为单位。
biHeight	16H	4 bytes	图像的高度, 以像素为单位。 注意, 这个值除了用于描述图像的高度之外, 它还有另外一个用途, 用于指明该图像是倒向的位图、还是正向的位图。 如果该值是一个正数, 说明是倒向的位图; 如果该值是一个负数, 则说明是正向的位图。 一般情况下, BMP 图像都是倒向的位图, 也就是该值是一个正数。
biPlanes	1AH	2 bytes	色彩平面数, 该值总被设置为 1。
biBitCount	1CH	2 bytes	像素深度, 指明一个像素点需要多少个 bit 数据来描述, 其值可为 1、4、8、16、24、32
biCompression	1EH	4 bytes	说明图像数据的压缩类型, 取值范围如下: ①0 - RGB 方式 ②1 - 8bpp 的 RLE 方式, 只用于 8bit 位图 ③2 - 4bpp 的 RLE 方式, 只用于 4bit 位图 ④3 - Bit-fields 方式 ⑤4 - 仅用于打印机 ⑥5 - 仅用于打印机
biSizeImage	22H	4 bytes	说明图像的大小, 以字节为单位, 当压缩类型为 BI_RGB 时, 可设置为 0。
biXPelsPerMeter	26H	4 bytes	水平分辨率, 用像素/米来表示, 有符号整数。
biYPelsPerMeter	2AH	4 bytes	垂直分辨率, 用像素/米来表示, 有符号整数。
biClrUsed	2EH	4 bytes	说明位图实际使用的彩色表中的颜色索引数。
biClrImportant	32H	4 bytes	说明对图像显示有重要影响的颜色索引的数目, 如果是 0, 则表示都重要。

表 19.5.3 位图信息头成员说明

从上面的描述信息, 再来对照文件数据:

image.bmp	Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
	00000000	42	4D	48	B8	0B	00	00	00	00	00	46	00	00	00	38	00	BMH, F 8
	00000016	00	00	20	03	00	00	E0	01	00	00	01	00	10	00	03	00	à
	00000032	00	00	02	B8	0B	00	C2	0E	00	00	C2	0E	00	00	00	00	, Å Å
	00000048	00	00	00	00	00	00	00	F8	00	00	E0	07	00	00	1F	00	ø à
	00000064	00	00	00	00	00	00	48	60	68	60	78	00	60	00	58		H`h`x`X
	00000080	00	68	00	68	00	68	00	70	00	90	00	80	00	80	00	98	h h h p € € ~
	00000096	00	88	00	70	A0	80	C0	90	20	88	00	88	00	90	00	90	^ p €À ^ ^
	00000112	00	90	20	B0	40	C0	00	A8	CF	99	BC	F5	57	FE	80	D8	°@À "i=4&WpEØ

图 19.5.5 位图信息头数据

0E~11H: 0x00000038=56, 这说明这个位图信息头的大小为 56 个字节。

12~15H: 0x00000320=800, 图像宽度为 800 个像素, 与文件属性一致。

16~19H: 0x000001E0=480, 图像高度为 480 个像素, 与文件属性一致; 这个数是一个正数, 说明是一个倒向的位图, 什么是正向的位图、什么是倒向的位图, 说的是图像数据的排列问题; 如果是正向的位图,

图像数据是按照图像的左上角到右下角方式排列的, 水平方向从左到右, 垂直方向从上到下。倒向的位图, 图像数据则是按照图像的左下角到右上角方式排列的, 水平方向依然从左到右, 垂直方向改为从下到上。

1A~1BH: 0x0001=1, 这个值总为 1。

1C~1DH: 0x0010=16, 表示每个像素占 16 个 bit。

1E~21H: 0x00000003=3, bit-fields 方式。

22~25H: 0x000BB802=768002, 图像的大小, 注意图像的大小并不是 BMP 文件的大小, 而是图像数据的大小。

26~29H: 0x00000EC2=3778, 水平分辨率为 3778 像素/米。

2A~2DH: 0x00000EC2=3778, 垂直分辨率为 3778 像素/米。

2E~31H: 0x00000000=0, 本位图未使用调色板。

32~35H: 0x00000000=0。

只有压缩方式选项被设置为 bit-fields (0x3) 时, 位图信息头的大小才会等于 56 字节, 否则, 为 40 字节。56 个字节相比于 40 个字节, 多出了 16 个字节, 那么多出的 16 个字节数据描述了什么信息呢? 稍后再给大家介绍。

三、调色板

调色板是单色、16 色、256 色位图图像文件所持有的, 如果是 16 位、24 位以及 32 位位图文件, 则 BMP 文件组成部分中不包含调色板, 关于调色板这里不过多介绍, 有兴趣可以自己去了解。

四、位图数据

位图数据其实就是图像的数据, 对于 24 位位图, 使用 3 个字节数据来表示一个像素点的颜色, 对于 16 位位图, 使用 2 个字节数据来表示一个像素点的颜色, 同理, 32 位位图则使用 4 个字节来描述。

BMP 位图分为正向的位图和倒向的位图, 主要区别在于图像数据存储的排列方式, 前面已经给大家解释的比较清楚了, 如下如所示 (左边对应的是正向位图, 右边对应的则是倒向位图):

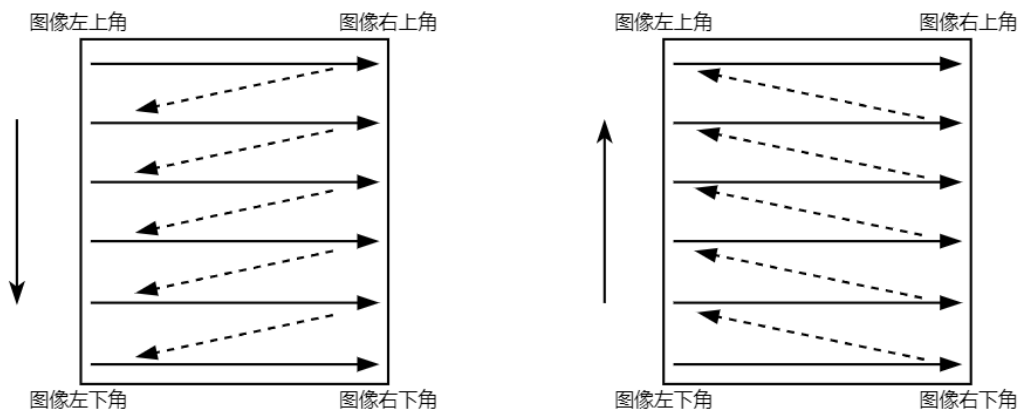


图 19.5.6 正向位图和倒向位图

所以正向位图先存储图像的第一行数据, 从左到右依次存放, 接着存放第二行, 依次这样; 而倒向位图, 则先存储图像的最后一行 (倒数第一行) 数据, 也是从左到右依次存放, 接着倒数二行, 依次这样。

RGB 和 Bit-Fields

当图像中引用的色彩超过 256 种时, 就需要 16bpp 或更高 bpp 的位图 (24 位、32 位)。调色板不适合 bpp 较大的位图, 因此 16bpp 及以上的位图都不使用调色板, 不使用调色板的位图图像有两种编码格式: RGB 和 Bit-Fields (下称 BF)。

RGB 编码格式是一种均分的思想, 使 Red、Green、Blue 三种颜色信息容量一样大, 譬如 24bpp-RGB, 它通常只有这一种编码格式, 在 24bits 中, 低 8 位表示 Blue 分量; 中 8 为表示 Green 分量; 高 8 位表示 Red 分量。

而在 32bpp-RGB 中, 低 24 位的编码方式与 24bpp 位图相同, 最高 8 位用来表示透明度 Alpha 分量。32bpp 的位图尺寸太大, 一般只有在图像处理的中间过程中使用。对于需要半透过效果的图像, 更好的选择是 PNG 格式。

BF 编码格式与 RGB 不同, 它利用位域操作, 人为地确定 RGB 三分量所包含的信息容量。位图信息头介绍中提到, 当压缩方式选项置为 BF 时, 位图信息头大小比平时多出 16 字节, 这 16 个字节实际上是 4 个 32bit 的位域掩码, 按照先后顺序, 它们分别是 R、G、B、A 四个分量的位域掩码, 当然如果没有 Alpha 分量, 则 Alpha 掩码没有实际意义。

位域掩码的作用是指出 R、G、B 三种颜色信息容量的大小, 分别使用多少个 bit 数据来表示, 以及三种颜色分量的位置偏移量。譬如对于 16 位色的 RGB565 图像, 通常使用 BF 编码格式, 同样这也是 BF 编码格式最著名和最普遍的应用之一, 它的 R、G 和 B 分量的位域掩码分别是 0xF800、0x07E0 和 0x001F, 也就是 R 通道使用 2 个字节中的高 5 位表示, G 通道使用 2 个字节中的中间 6 位表示。而 B 通道则使用 2 个字节中的最低 5 位表示, 如下图所示:

image.bmp	Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00000000	42	4D	48	B8	0B	00	00	00	00	00	00	46	00	00	00	38	00	BMH, F 8
00000016	00	00	20	03	00	00	E0	01	00	00	01	00	10	00	03	00	00	à
00000032	00	00	02	B8	0B	00	C2	0E	00	00	C2	0E	00	00	00	00	00	, Å Å
00000048	00	00	00	00	00	00	00	F8	00	00	E0	07	00	00	1F	00	00	ø à
00000064	00	00	00	00	00	00	00	48	60	68	60	78	00	60	00	58	00	H`h`x`X
00000080	00	68	00	68	00	68	00	70	00	90	00	80	00	80	00	98	00	h h h p € € ~
00000096	00	88	00	70	A0	80	C0	90	20	88	00	88	00	90	00	90	00	^ p €À ^ ^

图 19.5.7 R、G、B、A 四个分量的位域掩码

关于 BMP 图像文件的格式就给大家介绍这么多, 后面的程序代码中将不会再做解释!

如何得到 16 位色 RGB565 格式 BMP 图像?

在 Windows 下我们转换得到的 BMP 位图通常是 24 位色的 RGB888 格式图像, 那如何得到 RGB565 格式 BMP 位图呢? 当然这个方法很多, 这里笔者向大家介绍一种方法就是通过 Photoshop 软件来得到 RGB565 格式的 BMP 位图。

首先, 找一张图片, 图片格式无所谓, 只要 Photoshop 软件能打开即可; 确定图片之后, 我们启动 Photoshop 软件, 并且使用 Photoshop 软件打开这张图片, 打开之后点击菜单栏中的文件--->存储为, 接着出现如下界面:

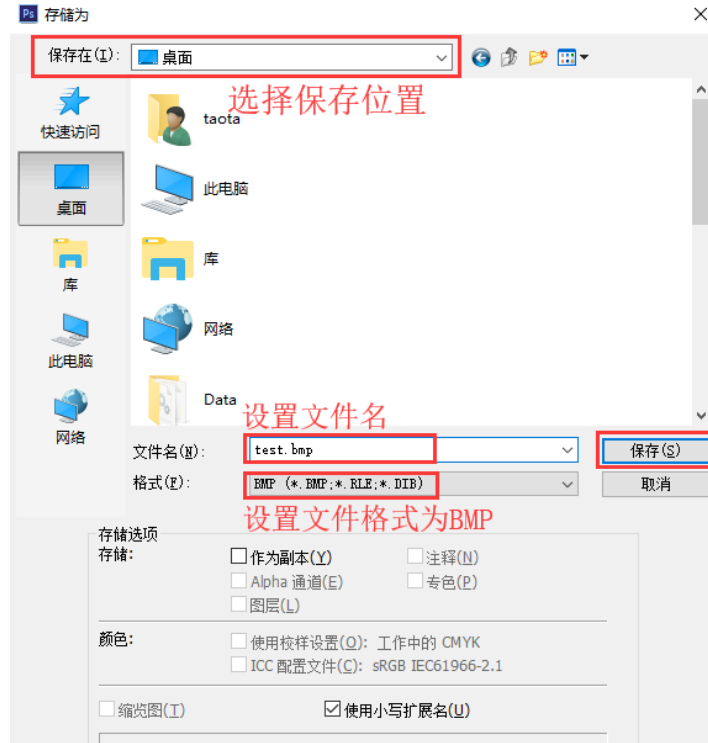


图 19.5.8 设置文件名和文件格式

在这个界面中, 首先选择文件保存的路径, 然后设置文件名以及文件格式, 选择文件格式为 BMP 格式, 之后点击保存, 如下:



图 19.5.9 BMP 选项

点击选择 16 位色图, 接着点击高级模式按钮:



图 19.5.10 BMP 高级模式

点击选择 RGB565，接着点击确定按钮即可，这样就得到 16 位色 RGB565 格式的 BMP 图像。

19.5.2 在 LCD 上显示 BMP 图像

通过上小节对 BMP 图像的介绍之后，相信大家对 BMP 文件的格式已经非常了解了，那么本小节我们将编写一个示例代码，在 LCD 上显示一张指定的 BMP 图像，示例代码笔者已经完成了，如下所示。

本例程源码对应的路径为：[开发板光盘->11、Linux C 应用编程例程源码->19_lcd->bmp_show.c](#)。

示例代码 19.5.1 显示 BMP 图像

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : show_bmp.c
作者 : 邓涛
版本 : V1.0
描述 : FrameBuffer 应用程序示例代码之显示 BMP 图像
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/6/15 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include <linux/fb.h>
#include <sys/mman.h>

/**** BMP 文件头数据结构 ****/
typedef struct {

```

```

    unsigned char type[2];        //文件类型
    unsigned int size;           //文件大小
    unsigned short reserved1;    //保留字段 1
    unsigned short reserved2;    //保留字段 2
    unsigned int offset;         //到位图数据的偏移量
} __attribute__((packed)) bmp_file_header;

/**** 位图信息头数据结构 ****/
typedef struct {
    unsigned int size;           //位图信息头大小
    int width;                   //图像宽度
    int height;                  //图像高度
    unsigned short planes;       //位面数
    unsigned short bpp;          //像素深度
    unsigned int compression;    //压缩方式
    unsigned int image_size;     //图像大小
    int x_pels_per_meter;        //像素/米
    int y_pels_per_meter;        //像素/米
    unsigned int clr_used;
    unsigned int clr_important;
} __attribute__((packed)) bmp_info_header;

/**** 静态全局变量 ****/
static int width;                //LCD X 分辨率
static int height;              //LCD Y 分辨率
static unsigned short *screen_base = NULL; //映射后的显存基地址
static unsigned long line_length; //LCD 一行的长度 (字节为单位)

/*****
* 函数名称: show_bmp_image
* 功能描述: 在 LCD 上显示指定的 BMP 图片
* 输入参数: 文件路径
* 返回值: 成功返回 0, 失败返回 -1
*****/
static int show_bmp_image(const char *path)
{
    bmp_file_header file_h;
    bmp_info_header info_h;
    unsigned short *line_buf = NULL; //行缓冲区
    unsigned long line_bytes; //BMP 图像一行的字节的大小
    unsigned int min_h, min_bytes;
    int fd = -1;
    int j;

```

```
/* 打开文件 */
if (0 > (fd = open(path, O_RDONLY))) {
    perror("open error");
    return -1;
}

/* 读取 BMP 文件头 */
if (sizeof(bmp_file_header) !=
    read(fd, &file_h, sizeof(bmp_file_header))) {
    perror("read error");
    close(fd);
    return -1;
}

if (0 != memcmp(file_h.type, "BM", 2)) {
    fprintf(stderr, "it's not a BMP file\n");
    close(fd);
    return -1;
}

/* 读取位图信息头 */
if (sizeof(bmp_info_header) !=
    read(fd, &info_h, sizeof(bmp_info_header))) {
    perror("read error");
    close(fd);
    return -1;
}

/* 打印信息 */
printf("文件大小: %d\n"
       "位图数据的偏移量: %d\n"
       "位图信息头大小: %d\n"
       "图像分辨率: %d*%d\n"
       "像素深度: %d\n", file_h.size, file_h.offset,
       info_h.size, info_h.width, info_h.height,
       info_h.bpp);

/* 将文件读写位置移动到图像数据开始处 */
if (-1 == lseek(fd, file_h.offset, SEEK_SET)) {
    perror("lseek error");
    close(fd);
    return -1;
}
```

```

}

/* 申请一个 buf、暂存 bmp 图像的一行数据 */
line_bytes = info_h.width * info_h.bpp / 8;
line_buf = malloc(line_bytes);
if (NULL == line_buf) {
    fprintf(stderr, "malloc error\n");
    close(fd);
    return -1;
}

if (line_length > line_bytes)
    min_bytes = line_bytes;
else
    min_bytes = line_length;

**** 读取图像数据显示到 LCD ****
/*****
* 为了软件处理上方便, 这个示例代码便不去做兼容性设计了
* 如果你想做兼容, 可能需要判断传入的 BMP 图像是 565 还是 888
* 如何判断呢? 文档里边说的很清楚了
* 我们默认传入的 bmp 图像是 RGB565 格式
*****/
if (0 < info_h.height) { //倒向位图
    if (info_h.height > height) {
        min_h = height;
        lseek(fd, (info_h.height - height) * line_bytes, SEEK_CUR);
        screen_base += width * (height - 1); //定位到屏幕左下角位置
    }
    else {
        min_h = info_h.height;
        screen_base += width * (info_h.height - 1); //定位到....不知怎么描述 懂的人自然懂!
    }

    for (j = min_h; j > 0; screen_base -= width, j--) {
        read(fd, line_buf, line_bytes); //读取出图像数据
        memcpy(screen_base, line_buf, min_bytes); //刷入 LCD 显存
    }
}
else { //正向位图
    int temp = 0 - info_h.height; //负数转成正数
    if (temp > height)
        min_h = height;

```

```
else
    min_h = temp;

for (j = 0; j < min_h; j++, screen_base += width) {
    read(fd, line_buf, line_bytes);
    memcpy(screen_base, line_buf, min_bytes);
}
}

/* 关闭文件、函数返回 */
close(fd);
free(line_buf);
return 0;
}

int main(int argc, char *argv[])
{
    struct fb_fix_screeninfo fb_fix;
    struct fb_var_screeninfo fb_var;
    unsigned int screen_size;
    int fd;

    /* 传参校验 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <bmp_file>\n", argv[0]);
        exit(-1);
    }

    /* 打开 framebuffer 设备 */
    if (0 > (fd = open("/dev/fb0", O_RDWR))) {
        perror("open error");
        exit(EXIT_FAILURE);
    }

    /* 获取参数信息 */
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

    screen_size = fb_fix.line_length * fb_var.yres;
    line_length = fb_fix.line_length;
    width = fb_var.xres;
    height = fb_var.yres;
```

```
/* 将显示缓冲区映射到进程地址空间 */
screen_base = mmap(NULL, screen_size, PROT_WRITE, MAP_SHARED, fd, 0);
if (MAP_FAILED == (void *)screen_base) {
    perror("mmap error");
    close(fd);
    exit(EXIT_FAILURE);
}

/* 显示 BMP 图片 */
memset(screen_base, 0xFF, screen_size);
show_bmp_image(argv[1]);

/* 退出 */
munmap(screen_base, screen_size); //取消映射
close(fd); //关闭文件
exit(EXIT_SUCCESS); //退出进程
}
```

代码中有两个自定义结构体 `bmp_file_header` 和 `bmp_info_header`, 描述 `bmp` 文件头的数据结构 `bmp_file_header`、以及描述位图信息头的数据结构 `bmp_info_header`。

当执行程序时候, 需要传入参数, 指定一个 `bmp` 文件。`main()`函数中会调用 `show_bmp_image()`函数在 LCD 上显示 `bmp` 图像, `show_bmp_image()`函数的参数为 `bmp` 文件路径, 在 `show_bmp_image()`函数中首先会打开指定路径的 `bmp` 文件, 得到对应的文件描述符 `fd`, 接着调用 `read()`函数读取 `bmp` 文件头和位图信息头。

获取到信息之后使用 `printf` 将其打印出来, 接着使用 `lseek()`函数将文件的读写位置移动到图像数据起始位置处, 也就是 `bmp_file_header` 结构体中的 `offset` 变量指定的地址偏移量。

通过 `info_h.height` 判断该 `BMP` 位图是正向的位图还是倒向的位图, 它们的处理方式不一样, 这些代码自己去看, 笔者不好去解释, 毕竟这只是文字描述的形式, 不太好表述! 代码只是一种参考, 自己能够独立写出来才是硬道理!

关于本示例代码就介绍这么多, 接下来使用交叉编译工具编译上述示例代码, 如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 19.5.11 编译示例代码

19.5.3 在开发板上测试

将上小节编译得到的可执行文件 `testApp` 以及测试使用的 `bmp` 图像文件拷贝到开发板 Linux 系统的用户家目录下:

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls  
driver image.bmp shell testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#
```

图 19.5.12 将 bmp 图像和测试程序拷贝到开发板

接着执行测试程序（在测试之前，先将出厂系统对应的 Qt GUI 应用程序退出）：

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./testApp image.bmp  
文件大小: 768072  
位图数据的偏移量: 70  
位图信息头大小: 56  
图像分辨率: 800*480  
像素深度: 16  
root@ATK-IMX6U:~#
```

图 19.5.13 执行测试程序

此时 LCD 屏上会显示 image.bmp 图像。

如下所示：



图 19.5.14 LCD 显示结果

忽略手机拍摄的问题，由于周围物体以及光线导致上图显示的结果与实际 LCD 显示的图像存在差异，image.bmp 原图如下所示：



图 19.5.15 image.bmp 原图

本章内容到此结束!

第二十章 在 LCD 上显示 jpeg 图像

我们常用的图片格式有很多,一般最常用的有三种:JPEG(或JPG)、PNG、BMP。上一章给大家介绍了如何在 LCD 上显示 BMP 图片,详细介绍了 BMP 图像的格式: BMP 图像虽然没有失真、并且解析简单,但是由于图像数据没有进行任何压缩,因此, BMP 图像文件所占用的存储空间很大,不适合存储在磁盘设备中。

而 JPEG(或 JPG)、PNG 则是经过压缩处理的图像格式,将图像数据进行压缩编码,大大降低了图像文件的大小,适合存储在磁盘设备中,所以很常用。本章我们就来学习如何在 LCD 屏上显示 jpeg 图像,下一章将向大家介绍如何在 LCD 屏上显示 png 图像。

本章将会讨论如下主题。

- JPEG 简介;
- libjpeg 库简介;
- libjpeg 库移植;
- 使用 libjpeg 库函数对 JPEG 图像进行解码;

20.1 JPEG 简介

JPEG (Joint Photographic Experts Group) 是由国际标准组织为静态图像所建立的第一个国际数字图像压缩标准, 也是至今一直在使用的、应用最广的图像压缩标准。

JPEG 由于可以提供有损压缩, 因此压缩比可以达到其他传统压缩算法无法比拟的程度; JPEG 虽然是有损压缩, 但这个损失的部分是人的视觉不容易察觉到的部分, 它充分利用了人眼对计算机色彩中的高频信息部分不敏感的特点, 来大大节省了需要处理的数据信息。

JPEG 压缩文件通常以 .jpg 或 .jpeg 作为文件后缀名, 关于 JPEG 压缩标准就给大家介绍这么多, 这些内容都是笔者从网络上截取下来的, 对此感兴趣的读者可以自行从网络上查阅这些信息。

20.2 libjpeg 简介

JPEG 压缩标准使用了一套压缩算法对原始图像数据进行了压缩得到 .jpg 或 .jpeg 图像文件, 如果要在 LCD 上显示 .jpg 或 .jpeg 图像文件, 则需要对其进行解压缩、以得到图像的原始数据, 譬如 RGB 数据。

既然压缩过程使用了算法, 那对 .jpg 或 .jpeg 图像文件进行解压同样也需要算法来处理, 当然, 笔者并不会教大家如何编写解压算法, 这些算法的实现也是很复杂的, 笔者肯定不会, 自然教不了大家! 但是, 我们可以使用别人写好的库、调用别人写好的库函数来解压 .jpg 或 .jpeg 图像文件, 也就是本小节要向大家介绍的 libjpeg 库。

libjpeg 是一个完全用 C 语言编写的函数库, 包含了 JPEG 解码 (解压缩)、JPEG 编码 (创建压缩) 和其他的 JPEG 功能的实现。可以使用 libjpeg 库对 .jpg 或 .jpeg 压缩文件进行解压或者生成 .jpg 或 .jpeg 压缩文件。

libjpeg 是一个开源 C 语言库, 我们获取到它的源代码。

20.3 libjpeg 移植

20.3.1 下载源码包

首先, 打开 <http://www.ijg.org/files/> 链接地址, 如下所示:

README	3 KB	Mon Jan 04 22:33 2016	
T-REC-T.871-201105-I!!PDF-E.pdf	198 KB	Tue Oct 21 13:22 2014	Portable Document Format
TIFFTechNote2.txt.gz	13 KB	Thu Apr 17 03:02 2008	Plain Text
Wallace.JPEG.pdf	78 KB	Fri May 11 23:29 2012	Portable Document Format
jdosabi.zip	2 KB	Thu Apr 17 03:02 2008	Zip Compressed Data
jfif3.pdf	17 KB	Tue Oct 21 14:47 2014	Portable Document Format
jpeg.documents.gz	2 KB	Thu Apr 17 03:02 2008	
jpegaltui.v9d.tar.gz	14 KB	Sat Jan 11 12:03 2020	Unix Tape Archive
jpegaltui9d.zip	18 KB	Sat Jan 11 11:09 2020	Zip Compressed Data
jpegsrc6b.zip	745 KB	Sat Apr 20 13:47 2002	Zip Compressed Data
jpegsrc7.zip	1,009 KB	Sat Jun 27 10:42 2009	Zip Compressed Data
jpegsrc8.zip	1,013 KB	Sun Jan 10 10:10 2010	Zip Compressed Data
jpegsrc8a.zip	1,014 KB	Sun Feb 28 11:19 2010	Zip Compressed Data
jpegsrc8b.zip	1,018 KB	Sun May 16 10:19 2010	Zip Compressed Data
jpegsrc8c.zip	1,032 KB	Sun Jan 16 10:17 2011	Zip Compressed Data
jpegsrc8d.zip	1,037 KB	Sun Jan 15 10:34 2012	Zip Compressed Data
jpegsrc9.zip	1,029 KB	Sun Jan 13 10:24 2013	Zip Compressed Data
jpegsrc9a.zip	1,042 KB	Sun Jan 19 10:26 2014	Zip Compressed Data
jpegsrc9b.zip	1,065 KB	Sun Jan 17 10:46 2016	Zip Compressed Data
jpegsrc9c.zip	1,071 KB	Sun Jan 14 10:10 2018	Zip Compressed Data
jpegsrc9d.zip	1,108 KB	Sun Jan 12 10:07 2020	Zip Compressed Data
jpegsrc.v6a.tar.gz	527 KB	Sun Mar 10 14:19 2019	Unix Tape Archive
jpegsrc.v6b.tar.gz	599 KB	Sun May 28 18:39 2006	Unix Tape Archive
jpegsrc.v7.tar.gz	938 KB	Sat Jun 27 10:18 2009	Unix Tape Archive
jpegsrc.v8.tar.gz	940 KB	Sun Jan 10 10:38 2010	Unix Tape Archive
jpegsrc.v8a.tar.gz	940 KB	Sun Feb 28 11:15 2010	Unix Tape Archive
jpegsrc.v8b.tar.gz	943 KB	Sun May 16 10:14 2010	Unix Tape Archive
jpegsrc.v8c.tar.gz	964 KB	Sun Jan 16 10:11 2011	Unix Tape Archive
jpegsrc.v8d.tar.gz	969 KB	Sun Jan 15 10:25 2012	Unix Tape Archive
jpegsrc.v9.tar.gz	965 KB	Sun Jan 13 10:15 2013	Unix Tape Archive
jpegsrc.v9a.tar.gz	977 KB	Sun Jan 19 10:18 2014	Unix Tape Archive
jpegsrc.v9b.tar.gz	999 KB	Sun Jan 17 10:38 2016	Unix Tape Archive
jpegsrc.v9c.tar.gz	1,004 KB	Sun Jan 14 11:48 2018	Unix Tape Archive
jpegsrc.v9d.tar.gz	1,046 KB	Sun Jan 12 10:49 2020	Unix Tape Archive
pm.errata.gz	2 KB	Thu Apr 17 03:02 2008	

图 20.3.1 libjpeg 下载链接页面

目前最新的一个版本是 v9d, 对应的年份为 2020 年, 这里我们选择一个适中的版本, 笔者以 v9b 为例, 对应的文件名为 `jpegsrc.v9b.tar.gz`, 点击该文件即可下载。

其实开发板出厂系统中已经移植了 libjpeg 库, 但是版本太旧了! 所以这里我们选择重新移植。

下载后如下所示:

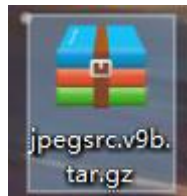


图 20.3.2 jpegsrc.v9b.tar.gz 源码包

20.3.2 编译源码

将 `jpegsrc.v9b.tar.gz` 压缩包文件拷贝到 Ubuntu 系统用户家目录下, 如下所示:

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  jpegsrc.v9b.tar.gz  tools  vscode_ws
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 20.3.3 将压缩包文件拷贝到 Ubuntu 系统

执行命令解压:

```
tar -xzf jpegsrc.v9b.tar.gz
```

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ tar -xzf jpegsrc.v9b.tar.gz
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  jpeg-9b  jpegsrc.v9b.tar.gz  tools  vscode_ws
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 20.3.4 将 jpegsrc.v9b.tar.gz 文件解压

解压成功之后会生成 jpeg-9b 文件夹，也就是 libjpeg 源码文件夹。

编译之前，在家目录下的 tools 文件夹中创建一个名为 jpeg 的文件夹，该目录作为 libjpeg 库的安装目录：

```
dt@dt-virtual-machine:~$ cd tools/
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
tslib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ mkdir jpeg
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
jpeg  tslib
dt@dt-virtual-machine:~/tools$
```

图 20.3.5 创建 jpeg 文件夹

回到家目录下，进入到 libjpeg 源码目录 jpeg-9b 中，该目录下包含的内容如下所示：

```
dt@dt-virtual-machine:~/jpeg-9b$ pwd
/home/dt/jpeg-9b
dt@dt-virtual-machine:~/jpeg-9b$
dt@dt-virtual-machine:~/jpeg-9b$ ls
aclocal.m4      configure.ac    jcdctmgr.c     jconfig.st     jdcoefct.c     jerror.h       jmemnobs.c     makcjpeg.st    Makefile.am
ar-lib         depcomp       jchuff.c      jconfig.txt    jdcclor.c     jfdctflt.c    jmemsys.h      makdjpeg.st    makefile.ansi
cderror.h     djpeg.l      jcinit.c     jconfig.vc     jdct.h        jfdctfst.c    jmorecfg.h    makeasln.v10  makefile.b32
cdjpeg.c     djpeg.c      jcmainct.c   jconfig.vms   jddctmgr.c    jfdctint.c    jpegint.h     makeasln.v10  makefile.bcc
cdjpeg.h     example.c    jcmarker.c   jconfig.wat   jdhuft.c     jidctflt.c    jpeglib.h     makecddep.vc6  makefile.dj
change.log   filelist.txt jcmaster.c   jcparm.c     jdinput.c     jidctfst.c    jpegtran.1    makecdsp.vc6  Makefile.in
cjpeg.l      install-sh   jcomapi.c    jcprept.c     jdmainct.c    jidctint.c    jpegtran.c    makecfil.v10  makefile.manx
cjpeg.c     install.txt  jconfig.bcc  jcsample.c    jdmarker.c    jinclude.h    jquant1.c     makecmak.vc6  makefile.mc6
ckconfig.c   jaricom.c   jconfig.cfg  jctrans.c     jdmaster.c    jmemansi.c    jquant2.c     makecvcx.v10  makefile.mms
coderrules.txt jcapimin.c  jconfig.dj   jdapimin.c    jdmerge.c     jmemdos.aasm  jutils.c      makeddep.vc6  makefile.sas
compile     jcapistd.c  jconfig.mac  jdapistd.c    jdpostct.c    jmemdos.c     jversion.h    makeddsp.vc6  makefile.unix
config.guess jcarith.c   jconfig.manx jdarith.c     jdsample.c    jmemmac.c     libjpeg.map   makedfil.v10  makefile.vc
config.sub   jccoefct.c  jconfig.mc6  jdatadst.c    jdtrans.c     jmemmgr.c     libjpeg.txt   makedmak.vc6  makefile.vms
configure    jccolor.c   jconfig.sas  jdatasrc.c    jerror.c      jmemname.c    ltmain.sh     makedvcx.v10  makefile.wat
dt@dt-virtual-machine:~/jpeg-9b$
```

图 20.3.6 libjpeg 源码目录下的文件

接下来对 libjpeg 源码进行交叉编译，跟编译 tslib 时步骤一样，包含三个步骤：

- 配置工程；
- 编译工程；
- 安装；

一套流程下来非常地快！没有任何难点。在此之前，先对交叉编译工具的环境进行初始化，使用 source 执行交叉编译工具安装目录下的 environment-setup-cortexa7hf-neon-poky-linux-gnueabi 脚本文件（如果已经初始化过了，那就不用再进行初始化了）：

```
source /opt/fsl-imx-x11/4.1.15-2.1.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

执行下面这条命令对 libjpeg 工程进行配置：

```
./configure --host=arm-poky-linux-gnueabi --prefix=/home/dt/tools/jpeg/
```

大家可以执行 `./configure --help` 查看它的配置选项以及含义, `--host` 选项用于指定交叉编译得到的库文件是运行在哪个平台, 通常将 `--host` 设置为交叉编译器名称的前缀, 譬如 `arm-poky-linux-gnueabi-gcc` 前缀就是 `arm-poky-linux-gnueabi`; `--prefix` 选项则用于指定库文件的安装路径, 将家目录下的 `tools/jpeg` 目录作为 `libjpeg` 的安装目录。

```
dt@dt-virtual-machine:~/jpeg-9b$
dt@dt-virtual-machine:~/jpeg-9b$ ./configure --host=arm-poky-linux-gnueabi --prefix=/home/dt/tools/jpeg/
configure: loading site script /opt/fsl-ixl-x11/4.1.15-2.1.0/site-config-cortexa7hf-neon-poky-linux-gnueabi
checking build system type... x86_64-unknown-linux-gnu
checking host system type... arm-poky-linux-gnueabi
checking target system type... arm-poky-linux-gnueabi
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for arm-poky-linux-gnueabi-strip... arm-poky-linux-gnueabi-strip
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether make supports nested variables... (cached) yes
checking whether to enable maintainer-specific portions of Makefiles... no
checking for arm-poky-linux-gnueabi-gcc... arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... yes
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/usr/arm-poky-linux-gnueabi accepts -g... yes
```

图 20.3.7 配置工程

接着执行 `make` 命令编译工程:

`make`

```
dt@dt-virtual-machine:~/jpeg-9b$ make
make all-am
make[1]: Entering directory '/home/dt/jpeg-9b'
CC jaricom.lo
CC jcapimin.lo
CC jcapistd.lo
CC jcarith.lo
CC jccoefct.lo
CC jccolor.lo
CC jcdctmgr.lo
CC jchuff.lo
CC jcinit.lo
CC jcmainct.lo
CC jcmarker.lo
CC jcmaster.lo
CC jcomapi.lo
CC jcparam.lo
CC jcpresct.lo
CC jcsample.lo
CC jctrans.lo
CC jdapimin.lo
CC jdapistd.lo
CC jdarith.lo
CC jdatadst.lo
CC jdatasrc.lo
```

图 20.3.8 编译工程

编译完成之后, 执行命令安装 `libjpeg`:

`make install`

```
dt@dt-virtual-machine:~/jpeg-9b$ make install
make[1]: Entering directory '/home/dt/jpeg-9b'
/bin/mkdir -p '/home/dt/tools/jpeg/lib'
/bin/bash ./libtool --mode=install /usr/bin/install -c libjpeg.la '/home/dt/tools/jpeg/lib'
libtool: install: /usr/bin/install -c .libs/libjpeg.so.9.2.0 /home/dt/tools/jpeg/lib/libjpeg.so.9.2.0
libtool: install: (cd /home/dt/tools/jpeg/lib && { ln -s -f libjpeg.so.9.2.0 libjpeg.so.9 || { rm -f l
libtool: install: (cd /home/dt/tools/jpeg/lib && { ln -s -f libjpeg.so.9.2.0 libjpeg.so || { rm -f lib
libtool: install: /usr/bin/install -c .libs/libjpeg.lai /home/dt/tools/jpeg/lib/libjpeg.la
libtool: install: /usr/bin/install -c .libs/libjpeg.a /home/dt/tools/jpeg/lib/libjpeg.a
libtool: install: chmod 644 /home/dt/tools/jpeg/lib/libjpeg.a
libtool: install: arm-poky-linux-gnueabi-ranlib /home/dt/tools/jpeg/lib/libjpeg.a
libtool: finish: PATH="/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin:/opt/fsl-im
2.1.0/sysroots/x86_64-pokysdk-linux/usr/sbin:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-lin
imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/./x86_64-pokysdk-linux/bin:/opt/fsl-imx-x1
fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-uclibc:/opt/fsl-imx-x11/
8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
lib
ldconfig: /home/dt/tools/jpeg/lib/libjpeg.so.9 is for unknown machine 40.
ldconfig: /home/dt/tools/jpeg/lib/libjpeg.so.9.2.0 is for unknown machine 40.
ldconfig: /home/dt/tools/jpeg/lib/libjpeg.so is for unknown machine 40.
-----
Libraries have been installed in:
  /home/dt/tools/jpeg/lib
If you ever happen to want to link against installed libraries
```

图 20.3.9 安装

命令执行完成之后，我们的 libjpeg 也就安装成功了！

20.3.3 安装目录下的文件夹介绍

进入到 libjpeg 安装目录：

```
dt@dt-virtual-machine:~/jpeg-9b$
dt@dt-virtual-machine:~/jpeg-9b$ cd ~/tools/jpeg/
dt@dt-virtual-machine:~/tools/jpeg$
dt@dt-virtual-machine:~/tools/jpeg$ pwd
/home/dt/tools/jpeg
dt@dt-virtual-machine:~/tools/jpeg$ ls
bin include lib share
dt@dt-virtual-machine:~/tools/jpeg$
```

图 20.3.10 安装目录下的文件夹

与 tslib 库安装目录下的包含的文件夹基本相同（除了没有 etc 目录），bin 目录下包含一些测试工具；include 目录下包含头文件；lib 目录下包含动态链接库文件。

进入到 include 目录下：

```
dt@dt-virtual-machine:~/tools/jpeg$ cd include/
dt@dt-virtual-machine:~/tools/jpeg/include$
dt@dt-virtual-machine:~/tools/jpeg/include$ pwd
/home/dt/tools/jpeg/include
dt@dt-virtual-machine:~/tools/jpeg/include$ ls
jconfig.h jerror.h jmorecfg.h jpeglib.h
dt@dt-virtual-machine:~/tools/jpeg/include$
```

图 20.3.11 头文件

在这个目录下包含了 4 个头文件，在应用程序中，我们只需包含 jpeglib.h 头文件即可！

进入到 lib 目录下：

```
dt@dt-virtual-machine:~/tools/jpeg$
dt@dt-virtual-machine:~/tools/jpeg$ cd lib/
dt@dt-virtual-machine:~/tools/jpeg/lib$
dt@dt-virtual-machine:~/tools/jpeg/lib$ pwd
/home/dt/tools/jpeg/lib
dt@dt-virtual-machine:~/tools/jpeg/lib$
dt@dt-virtual-machine:~/tools/jpeg/lib$ ls
libjpeg.a libjpeg.la libjpeg.so libjpeg.so.9 libjpeg.so.9.2.0
dt@dt-virtual-machine:~/tools/jpeg/lib$
```

图 20.3.12 库文件

libjpeg.so 和 libjpeg.so.9 都是符号链接, 指向 libjpeg.so.9.2.0。

20.3.4 移植到开发板

开发板出厂系统已经移植了 libjpeg 库, 前面给大家提到过, 只是移植的版本太低了, 所以这里不打算使用出厂系统移植的 libjpeg 库, 而使用 20.3.2 小节交叉编译好的 libjpeg 库。

进入到 libjpeg 安装目录下, 将 bin 目录下的所有测试工具拷贝到开发板 Linux 系统/usr/bin 目录; 将 lib 目录下的所有库文件拷贝到开发板 Linux 系统/usr/lib 目录。

拷贝 lib 目录下的库文件时, 需要注意符号链接的问题, 不能破坏原有的符号链接; 可以将 lib 目录下的所有文件打包成压缩包的形式, 譬如进入到 lib 目录, 执行命令:

```
tar -czf lib.tar.gz ./*
```

```
dt@dt-virtual-machine:~/tools/jpeg$ cd lib/
dt@dt-virtual-machine:~/tools/jpeg/lib$
dt@dt-virtual-machine:~/tools/jpeg/lib$ pwd
/home/dt/tools/jpeg/lib
dt@dt-virtual-machine:~/tools/jpeg/lib$ ls
libjpeg.a libjpeg.la libjpeg.so libjpeg.so.9 libjpeg.so.9.2.0
dt@dt-virtual-machine:~/tools/jpeg/lib$
dt@dt-virtual-machine:~/tools/jpeg/lib$ tar -czf lib.tar.gz ./*
dt@dt-virtual-machine:~/tools/jpeg/lib$
dt@dt-virtual-machine:~/tools/jpeg/lib$ ls
libjpeg.a libjpeg.la libjpeg.so libjpeg.so.9 libjpeg.so.9.2.0 lib.tar.gz
dt@dt-virtual-machine:~/tools/jpeg/lib$
```

图 20.3.13 将 lib 目录下的所有文件打包

再将 lib.tar.gz 压缩文件拷贝到开发板 Linux 的用户家目录下, 在解压之前, 将开发板出厂系统中已经移植的 libjpeg 库删除, 执行命令:

```
rm -rf /usr/lib/libjpeg.*
```

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# rm -rf /usr/lib/libjpeg.*
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 20.3.14 删除出厂系统原有的 libjpeg 库

Tips: 注意! 当出厂系统原有的 libjpeg 库被删除后, 将会导致开发板下次启动后, 出厂系统的 Qt GUI 应用程序会出现一些问题, 原本显示图片的位置变成了空白, 显示不出来了! 原因在于 Qt 程序处理图片 (对 jpeg 图片解码) 时, 它的底层使用到了 libjpeg 库, 而现在我们将出厂系统原有的 libjpeg 库给删除了, 自然就会导致 Qt GUI 应用程序中图片显示不出来 (无法对 jpeg 图片进行解码)!

这个跟具体的 libjpeg 版本绑定起来的, 即使我们将 20.3.2 小节编译得到的库文件拷贝到 /usr/lib 目录下, 也是无济于事, 因为版本不同, 这里大家知道就行。

接着我们将 lib.tar.gz 压缩文件解压到开发板 Linux 系统/usr/lib 目录下:


```
tar -xzf lib.tar.gz -C /usr/lib
```

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# pwd
/home/root
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver lib.tar.gz shell
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# tar -xzf lib.tar.gz -C /usr/lib
root@ATK-IMX6U:~#
```

图 20.3.15 将 lib.tar.gz 解压到/usr/lib 目录

解压成功之后,接着执行 libjpeg 提供的测试工具,看看我们移植成功没:

```
jpeg --help
```

```
root@ATK-IMX6U:~# jpeg --help
usage: jpeg [switches] [inputfile]
Switches (names may be abbreviated):
  -colors N          Reduce image to no more than N colors
  -fast             Fast, low-quality processing
  -grayscale        Force grayscale output
  -rgb              Force RGB output
  -scale M/N        Scale output image by fraction M/N, eg, 1/8
  -bmp              Select BMP output format (Windows style)
  -gif              Select GIF output format
  -os2              Select BMP output format (OS/2 style)
  -pnm              Select PBMPLUS (PPM/PGM) output format (default)
  -targa            Select Targa output format
Switches for advanced users:
  -dct int          Use integer DCT method (default)
  -dct fast         Use fast integer DCT (less accurate)
  -dct float        Use floating-point DCT method
  -dither fs        Use F-S dithering (default)
  -dither none      Don't use dithering in quantization
  -dither ordered   Use ordered dither (medium speed, quality)
  -map FILE         Map to colors used in named image file
  -nosmooth         Don't use high-quality upsampling
  -onepass          Use 1-pass quantization (fast, low quality)
  -maxmemory N     Maximum memory to use (in kbytes)
  -outfile name    Specify name for output file
  -verbose or -debug Emit debug output
root@ATK-IMX6U:~#
```

图 20.3.16 测试 libjpeg 移植是否成功

jpeg 是编译 libjpeg 源码得到的测试工具(在 libjpeg 安装目录下的 lib 目录中),当执行命令之后,能够成功打印出这些信息就表示我们的移植成功了!

20.4 libjpeg 使用说明

libjpeg 提供 JPEG 解码、JPEG 编码和其他的 JPEG 功能的实现,本小节我们只给大家介绍如何使用 libjpeg 提供的库函数对 .jpg/.jpeg 进行解码(解压),得到 RGB 数据。

首先,使用 libjpeg 库需要在我们的应用程序中包含它的头文件 jpeglib.h,该头文件包含了一些结构体数据结构以及 API 接口的申明。先来看看解码操作的过程:

- (1)、创建 jpeg 解码对象;
- (2)、指定解码数据源;
- (3)、读取图像信息;
- (4)、设置解码参数;
- (5)、开始解码;
- (6)、读取解码后的数据;
- (7)、解码完毕;

(8)、释放/销毁解码对象。

以上便是整个解码操作的过程, 用 `libjpeg` 库解码 `jpeg` 数据的时候, 最重要的一个数据结构为 `struct jpeg_decompress_struct` 结构体, 该数据结构记录着 `jpeg` 数据的详细信息, 也保存着解码之后输出数据的详细信息。除此之外, 还需要定义一个用于处理错误的对象, 错误处理对象是一个 `struct jpeg_error_mgr` 结构体变量。

```
struct jpeg_decompress_struct cinfo;
struct jpeg_error_mgr jerr;
```

以上就定义了 JPEG 解码对象和错误处理对象。

20.4.1 错误处理

使用 `libjpeg` 库函数的时候难免会产生错误, 所以我们在使用 `libjpeg` 解码之前, 首先要做好错误处理。在 `libjpeg` 库中, 实现了默认错误处理函数, 当错误发生时, 譬如如果内存不足、文件格式不对等, 则会 `libjpeg` 实现的默认错误处理函数, 默认错误处理函数将会调用 `exit()` 结束整个进程; 当然, 我们可以修改错误处理的方式, `libjpeg` 提供了接口让用户可以注册一个自定义错误处理函数。

错误处理对象使用 `struct jpeg_error_mgr` 结构体描述, 该结构体内容如下所示:

示例代码 20.4.1 `struct jpeg_error_mgr` 结构体

```
/* Error handler object */
struct jpeg_error_mgr {
    /* Error exit handler: does not return to caller */
    JMETHOD(noreturn_t, error_exit, (j_common_ptr cinfo));
    /* Conditionally emit a trace or warning message */
    JMETHOD(void, emit_message, (j_common_ptr cinfo, int msg_level));
    /* Routine that actually outputs a trace or error message */
    JMETHOD(void, output_message, (j_common_ptr cinfo));
    /* Format a message string for the most recent JPEG error or message */
    JMETHOD(void, format_message, (j_common_ptr cinfo, char * buffer));
#define JMSG_LENGTH_MAX 200 /* recommended size of format_message buffer */
    /* Reset error state variables at start of a new image */
    JMETHOD(void, reset_error_mgr, (j_common_ptr cinfo));

    /* The message ID code and any parameters are saved here.
     * A message can have one string parameter or up to 8 int parameters.
     */
    int msg_code;
#define JMSG_STR_PARM_MAX 80
    union {
        int i[8];
        char s[JMSG_STR_PARM_MAX];
    } msg_parm;

    /* Standard state variables for error facility */
    int trace_level; /* max msg_level that will be displayed */
};
```

```

/* For recoverable corrupt-data errors, we emit a warning message,
 * but keep going unless emit_message chooses to abort.  emit_message
 * should count warnings in num_warnings.  The surrounding application
 * can check for bad data by seeing if num_warnings is nonzero at the
 * end of processing.
 */
long num_warnings;          /* number of corrupt-data warnings */

/* These fields point to the table(s) of error message strings.
 * An application can change the table pointer to switch to a different
 * message list (typically, to change the language in which errors are
 * reported).  Some applications may wish to add additional error codes
 * that will be handled by the JPEG library error mechanism; the second
 * table pointer is used for this purpose.
 *
 * First table includes all errors generated by JPEG library itself.
 * Error code 0 is reserved for a "no such error string" message.
 */
const char * const * jpeg_message_table; /* Library errors */
int last_jpeg_message; /* Table contains strings 0..last_jpeg_message */
/* Second table can be added by application (see cjpeg/djpeg for example).
 * It contains strings numbered first_addon_message..last_addon_message.
 */
const char * const * addon_message_table; /* Non-library errors */
int first_addon_message; /* code for first string in addon table */
int last_addon_message; /* code for last string in addon table */
};

```

error_exit 函数指针便指向了错误处理函数。使用 libjpeg 库函数 jpeg_std_error() 会将 libjpeg 错误处理设置为默认处理方式。如下所示:

```
//初始化错误处理对象、并将其与解压对象绑定
```

```
cinfo.err = jpeg_std_error(&jerr);
```

如果我们要修改默认的错误处理函数, 可这样操作:

```
void my_error_exit(struct jpeg_decompress_struct *cinfo)
```

```
{
    /* ... */
}
```

```
cinfo.err.error_exit = my_error_exit;
```

20.4.2 创建解码对象

要使用 libjpeg 解码 jpeg 数据, 这步是必须要做的。

```
jpeg_create_decompress(&cinfo);
```

在创建解码对象之后, 如果解码结束或者解码出错时, 需要调用 `jpeg_destroy_decompress` 销毁/释放解码对象, 否则将会内存泄漏。

20.4.3 设置数据源

也就是设置需要进行解码的 jpeg 文件, 使用 `jpeg_stdio_src()` 函数设置数据源:

```
FILE *jpeg_file = NULL;

//打开.jpeg/.jpg 图像文件
jpeg_file = fopen("./image.jpg", "r"); //只读方式打开
if (NULL == jpeg_file) {
    perror("fopen error");
    return -1;
}

//指定图像文件
jpeg_stdio_src(&cinfo, jpeg_file);
```

待解码的 jpeg 文件使用标准 I/O 方式 `fopen` 将其打开。除此之外, jpeg 数据源还可以来自内存中、而不一定的是文件流。

20.4.4 读取 jpeg 文件的头信息

这个和创建解码对象一样, 是必须要调用的, 是约定, 没什么好说的。因为在解码之前, 需要读取 jpeg 文件的头部信息, 以获取该文件的信息, 这些获取到的信息会直接赋值给 `cinfo` 对象的某些成员变量。

```
jpeg_read_header(&cinfo, TRUE);
```

调用 `jpeg_read_header()` 后, 可以得到 jpeg 图像的一些信息, 譬如 jpeg 图像的宽度、高度、颜色通道数以及 `colorspace` 等, 这些信息会赋值给 `cinfo` 对象中的相应成员变量, 如下所示:

```
cinfo.image_width      //jpeg 图像宽度
cinfo.image_height    //jpeg 图像高度
cinfo.num_components  //颜色通道数
cinfo.jpeg_color_space //jpeg 图像的颜色空间
```

支持的颜色包括如下几种:

```
/* Known color spaces. */
typedef enum {
    JCS_UNKNOWN,      /* error/unspecified */
    JCS_GRAYSCALE,    /* monochrome */
    JCS_RGB,          /* red/green/blue, standard RGB (sRGB) */
    JCS_YCbCr,        /* Y/Cb/Cr (also known as YUV), standard YCC */
    JCS_CMYK,         /* C/M/Y/K */
    JCS_YCCK,         /* Y/Cb/Cr/K */
    JCS_BG_RGB,       /* big gamut red/green/blue, bg-sRGB */
    JCS_BG_YCC        /* big gamut Y/Cb/Cr, bg-sYCC */
} J_COLOR_SPACE;
```

20.4.5 设置解码处理参数

在进行解码之前,我们可以对一些解码参数进行设置,这些参数都有一个默认值,调用 `jpeg_read_header()` 函数后,这些参数被设置成相应的默认值。

直接对 `cinfo` 对象的成员变量进行修改即可,这里介绍两个比较有代表性的解码处理参数:

- 输出的颜色 (`cinfo.out_color_space`): 默认配置为 RGB 颜色,也就是 JCS_RGB;
- 图像缩放操作 (`cinfo.scale_num` 和 `cinfo.scale_denom`): `libjpeg` 可以设置解码出来的图像的大小,也就是与原图的比例。使用 `scale_num` 和 `scale_denom` 两个参数,解出来的图像大小就是 `scale_num/scale_denom`, JPEG 当前仅支持 1/1、1/2、1/4、和 1/8 这几种缩小比例。默认是 1/1,也就是保持原图大小。譬如要将输出图像设置为原图的 1/2 大小,可进行如下设置:

```
cinfo.scale_num=1;
cinfo.scale_denom=2;
```

20.4.6 开始解码

经过前面的参数设置,我们可以开始解码了,调用 `jpeg_start_decompress()` 函数:

```
jpeg_start_decompress(&cinfo);
```

在完成解压缩操作后,会将解压后的图像信息填充至 `cinfo` 结构中。譬如,输出图像宽度 `cinfo.output_width`,输出图像高度 `cinfo.output_height`,每个像素中的颜色通道数 `cinfo.output_components` (比如灰度为 1,全彩色 RGB888 为 3) 等。

一般情况下,这些参数是在 `jpeg_start_decompress` 后才被填充到 `cinfo` 中的,如果希望在调用 `jpeg_start_decompress` 之前就获得这些参数,可以通过调用 `jpeg_calc_output_dimensions()` 的方法来实现。

20.4.7 读取数据

接下来就可以读取解码后的数据了,数据是按照行读取的,解码后的数据按照从左到右、从上到下的顺序存储,每个像素点对应的各颜色或灰度通道数据是依次存储,譬如一个 24-bit RGB 真彩色的图像中,一行的数据存储模式为 B,G,R,B,G,R,B,G,R,...。

`libjpeg` 默认解码得到的图像数据是 BGR888 格式,即 R 颜色在低 8 位、而 B 颜色在高 8 位。可以定义一个 BGR888 颜色类型,如下所示:

```
typedef struct bgr888_color {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} __attribute__((packed)) bgr888_t;
```

每次读取一行数据,计算每行数据需要的空间大小,比如 RGB 图像就是宽度×3 (24-bit RGB 真彩色一个像素 3 个字节),灰度图就是宽度×1 (一个像素 1 个字节)。

```
bgr888_t *line_buf = malloc(cinfo.output_width * cinfo.output_components);
```

以上我们分配了一个行缓冲区,它的大小为 `cinfo.output_width * cinfo.output_components`,也就是输出图像的宽度乘上每一个像素的字节大小。我们除了使用 `malloc` 分配缓冲区外,还可以使用 `libjpeg` 的内存管理器来分配缓冲区,这个不再介绍!

缓冲区分配好之后,接着可以调用 `jpeg_read_scanlines()` 来读取数据,`jpeg_read_scanlines()` 可以指定一次读多少行,但是目前该函数还只能支持一次只读 1 行;函数如下所示:

```
jpeg_read_scanlines(&cinfo, &buf, 1);
```

1 表示每次读取的行数,通常都是将其设置为 1。

`cinfo.output_scanline` 表示接下来要读取的行对应的索引值, 初始化为 0 (表示第一行)、1 表示第二行等, 每读取一行数据, 该变量就会加 1, 所以我们可以通过下面这种循环方式依次读取解码后的所有数据:

```
while(cinfo.output_scanline < cinfo.output_height)
{
    jpeg_read_scanlines(&cinfo, buffer, 1);
    //do something
}
```

20.4.8 结束解码

解码完毕之后调用 `jpeg_finish_decompress()` 函数:

```
jpeg_finish_decompress(&cinfo);
```

20.4.9 释放/销毁解码对象

当解码完成之后, 我们需要调用 `jpeg_destroy_decompress()` 函数销毁/释放解码对象:

```
jpeg_destroy_decompress(&cinfo);
```

20.5 libjpeg 应用编程

通过上小节的介绍, 我们已经知道了如何使用 `libjpeg` 提供的库函数来解码 `.jpg/.jpeg` 图像, 本小节进行实战, 对一个指定的 `jpeg` 图像进行解码, 显示在 LCD 屏上, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->20_libjpeg->show_jpeg_image.c](#)。

示例代码 20.5.1 libjpeg 应用程序示例代码

```
/******
```

Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.

文件名 : `show_jpeg_image.c`

作者 : 邓涛

版本 : V1.0

描述 : `libjpeg` 使用实战

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/6/15 邓涛创建

```
*****/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include <linux/fb.h>
#include <sys/mman.h>
```

```
#include <jpeglib.h>

typedef struct bgr888_color {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} __attribute__((packed)) bgr888_t;

static int width;           //LCD X 分辨率
static int height;         //LCD Y 分辨率
static unsigned short *screen_base = NULL; //映射后的显存基地址
static unsigned long line_length; //LCD 一行的长度 (字节为单位)
static unsigned int bpp; //像素深度 bpp

static int show_jpeg_image(const char *path)
{
    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;
    FILE *jpeg_file = NULL;
    bgr888_t *jpeg_line_buf = NULL; //行缓冲区:用于存储从 jpeg 文件中解压出来的一行图像数据
    unsigned short *fb_line_buf = NULL; //行缓冲区:用于存储写入到 LCD 显存的一行数据
    unsigned int min_h, min_w;
    unsigned int valid_bytes;
    int i;

    //绑定默认错误处理函数
    cinfo.err = jpeg_std_error(&jerr);

    //打开.jpeg/.jpg 图像文件
    jpeg_file = fopen(path, "r"); //只读方式打开
    if (NULL == jpeg_file) {
        perror("fopen error");
        return -1;
    }

    //创建 JPEG 解码对象
    jpeg_create_decompress(&cinfo);

    //指定图像文件
    jpeg_stdio_src(&cinfo, jpeg_file);

    //读取图像信息
    jpeg_read_header(&cinfo, TRUE);
```

```
printf("jpeg 图像大小: %d*%d\n", cinfo.image_width, cinfo.image_height);

//设置解码参数
cinfo.out_color_space = JCS_RGB;//默认就是 JCS_RGB
//cinfo.scale_num = 1;
//cinfo.scale_denom = 2;

//开始解码图像
jpeg_start_decompress(&cinfo);

//为缓冲区分配内存空间
jpeg_line_buf = malloc(cinfo.output_components * cinfo.output_width);
fb_line_buf = malloc(line_length);

//判断图像和 LCD 屏那个的分辨率更低
if (cinfo.output_width > width)
    min_w = width;
else
    min_w = cinfo.output_width;

if (cinfo.output_height > height)
    min_h = height;
else
    min_h = cinfo.output_height;

//读取数据
valid_bytes = min_w * bpp / 8;//一行的有效字节数 表示真正写入到 LCD 显存的一行数据的大小
while (cinfo.output_scanline < min_h) {

    jpeg_read_scanlines(&cinfo, (unsigned char **)&jpeg_line_buf, 1);//每次读取一行数据

    //将读取到的 BGR888 数据转为 RGB565
    for (i = 0; i < min_w; i++)
        fb_line_buf[i] = ((jpeg_line_buf[i].red & 0xF8) << 8) |
            ((jpeg_line_buf[i].green & 0xFC) << 3) |
            ((jpeg_line_buf[i].blue & 0xF8) >> 3);

    memcpy(screen_base, fb_line_buf, valid_bytes);
    screen_base += width;//+width 定位到 LCD 下一行显存地址的起点
}

//解码完成
jpeg_finish_decompress(&cinfo); //完成解码
```



```
jpeg_destroy_decompress(&cinfo); //销毁 JPEG 解码对象、释放资源

//关闭文件、释放内存
fclose(jpeg_file);
free(fb_line_buf);
free(jpeg_line_buf);
return 0;
}

int main(int argc, char *argv[])
{
    struct fb_fix_screeninfo fb_fix;
    struct fb_var_screeninfo fb_var;
    unsigned int screen_size;
    int fd;

    /* 传参校验 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <jpeg_file>\n", argv[0]);
        exit(-1);
    }

    /* 打开 framebuffer 设备 */
    if (0 > (fd = open("/dev/fb0", O_RDWR))) {
        perror("open error");
        exit(EXIT_FAILURE);
    }

    /* 获取参数信息 */
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

    line_length = fb_fix.line_length;
    bpp = fb_var.bits_per_pixel;
    screen_size = line_length * fb_var.yres;
    width = fb_var.xres;
    height = fb_var.yres;

    /* 将显示缓冲区映射到进程地址空间 */
    screen_base = mmap(NULL, screen_size, PROT_WRITE, MAP_SHARED, fd, 0);
    if (MAP_FAILED == (void *)screen_base) {
        perror("mmap error");
        close(fd);
    }
}
```

```

        exit(EXIT_FAILURE);
    }

    /* 显示 BMP 图片 */
    memset(screen_base, 0xFF, screen_size);
    show_jpeg_image(argv[1]);

    /* 退出 */
    munmap(screen_base, screen_size); //取消映射
    close(fd); //关闭文件
    exit(EXIT_SUCCESS); //退出进程
}

```

代码就不再讲解了，前面的内容看懂了，代码自然就能看懂！

在 while 循环中，通过 jpeg_read_scanlines() 每次读取一行数据，注意，jpeg_read_scanlines() 函数的第二个参数是一个 unsigned char ** 类型指针。读取到数据之后，需要将其转为 RGB565 格式，因为我们这个开发板出厂系统，LCD 是 RGB565 格式的显示设备，这个转化非常简单，没什么可说的，懂的人自然懂！

编译上述代码：

```
{CC} -o testApp testApp.c -I /home/dt/tools/jpeg/include -L /home/dt/tools/jpeg/lib -ljpeg
```

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c -I /home/dt/tools/jpeg/include -L /home/dt/tools/jpeg/lib -ljpeg
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 20.5.1 编译示例代码

编译的时候需要指定头文件的路径、库文件的路径以及需要链接的库文件，与编译 tslib 应用程序是一样的道理。

将编译得到的可执行文件和一个 .jpg/.jpeg 图像文件拷贝到开发板 Linux 系统的用户家目录下，执行测试程序：

```

root@ATK-IMX6U:~# ls
driver image.jpg shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp image.jpg
jpeg图像大小: 853*480
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#

```

图 20.5.2 执行应用程序

此时 LCD 屏上便会显示这张图片，如下所示（执行测试程序之前，建议关闭出厂系统的 Qt GUI 应用程序）：



图 20.5.3 LCD 显示效果

20.6 总结

关于本章的内容就向大家介绍这么多, `libjpeg` 除了 JPEG 解码功能外, 还可以实现 JPEG 编码以及其它一些 JPEG 功能, 大家可以自己去学习、去摸索一下, 笔者不可能把所有 API 都给你讲一遍, 这是不现实的, 譬如后面会给大家介绍音频应用编程, 用到了 `alsa-lib` 库, 这个库估计包含了几百个 API, 你说我会一个一个给你讲吗? 所以这是不可能的事情, 大家应该学习的是一种方法, 在原有内容的基础上进行扩展, 学习更多的用法, 而不仅限于本书中的这些内容。

`libjpeg` 提供的 API 其实并不是很多, 大家可以打开它的头文件 `jpeglib.h`, 大致去浏览一下, 其实从它函数的命名上可以看出它的一个大致作用, 再结合注释信息基本可以确定函数的功能, 除此之外, 这些函数库都会提供一些示例代码供用户参考。笔者也曾尝试找了找 `libjpeg` 官方的帮助文档, 但是很遗憾未能找到! 不知是官方没有出帮助文档还是笔者找的方法不对, 总之, 笔者确实没找到, 如果有哪位读者找到了, 那么希望可以通知到笔者, 我会把它的链接地址写入本书, 供读者查阅!

OK, 那本章内容到此结束! 大家加油!

第二十一章 在 LCD 上显示 png 图片

上一章介绍了如何使用 libjpeg 库对 jpeg 图像进行解码、并显示到 LCD 屏上,除了 jpeg 图像之外, png 图像也很常见,那本章我们就来学习如何对 png 图像进行解码、并显示到 LCD 屏上。

本章将会讨论如下主题。

- PNG 简介;
- libpng 库简介;
- libpng 库移植;
- 使用 libpng 库函数对 PNG 图像进行解码;

21.1 PNG 简介

以下的这些内容都是从网络上截取下来的。

PNG (便携式网络图形格式 Portable Network Graphic Format, 简称 PNG) 是一种采用无损压缩算法的位图格式, 其设计目的是试图替代 GIF 和 TIFF 文件, 同时增加一些 GIF 文件所不具备的特性。PNG 使用从 LZ77 派生的无损数据压缩算法, 它压缩比高, 生成文件体积小, 并且支持透明效果, 所以被广泛使用。

特点

- **无损压缩:** PNG 文件采用 LZ77 算法的派生算法进行压缩, 其结果是获得高的压缩比, 不损失数据。它利用特殊的编码方法标记重复出现的数据, 因而对图像的颜色没有影响, 也不可能产生颜色的损失, 这样就可以重复保存而不降低图像质量。
- **体积小:** 在保证图片清晰、逼真、不失真的前提下, PNG 使用从 LZ77 派生的无损数据压缩算法, 它压缩比高, 生成文件体积小;
- **索引彩色模式:** PNG-8 格式与 GIF 图像类似, 同样采用 8 位调色板将 RGB 彩色图像转换为索引彩色图像。图像中保存的不再是各个像素的彩色信息, 而是从图像中挑选出来的具有代表性的颜色编号, 每一编号对应一种颜色, 图像的数据量也因此减少, 这对彩色图像的传播非常有利。
- **更优化的网络传输显示:** PNG 图像在浏览器上采用流式浏览, 即使经过交错处理的图像会在完全下载之前提供浏览者一个基本的图像内容, 然后再逐渐清晰起来。它允许连续读出和写入图像数据, 这个特性很适合于在通信过程中显示和生成图像。
- **支持透明效果:** PNG 可以为原图像定义 256 个透明层次, 使得彩色图像的边缘能与任何背景平滑地融合, 从而彻底地消除锯齿边缘。这种功能是 GIF 和 JPEG 没有的。

关于 PNG 格式就介绍这么多。

21.2 libpng 简介

对于 png 图像, 我们可以使用 libpng 库对其进行解码, 跟 libjpeg 一样, 它也是一套免费、开源的 C 语言函数库, 支持对 png 图像文件解码、编码等功能。

21.3 zlib 移植

zlib 其实是一套包含了数据压缩算法的函数库, 此函数库为自由软件, 是一套免费、开源的 C 语言函数库, 所以我们可以获取到它源代码。

libpng 依赖于 zlib 库, 所以要想移植 libpng 先得移植 zlib 库才可以, zlib 也好、libpng 也好, 其实移植过程非常简单, 无非就是下载源码、编译源码这样的一些工作, 那本小节就向大家介绍如何移植 zlib。

在移植之前, 先给大家说明一下, 我们的开发板出厂系统都是已经移植好了这些库, 其实是可以直接使用的, 但是作为学习, 必须要自己亲自把这些库给移植到开发板, 这是非常重要的!

21.3.1 下载源码包

我们可以进入到 <https://www.zlib.net/fossils/> 这个链接地址下载 zlib 源码包:

Index of /fossils

Name	Last modified	Size	Description
Parent Directory	-	-	-
OBSOLETE	2017-01-15 11:23	267	
zlib-0.8.tar.gz	1995-04-29 10:53	59K	
zlib-0.9.tar.gz	1995-05-01 10:36	62K	
zlib-0.71.tar.gz	1995-04-14 14:40	57K	
zlib-0.79.tar.gz	1995-04-28 11:21	59K	
zlib-0.91.tar.gz	1995-05-02 10:57	62K	
zlib-0.92.tar.gz	1995-05-03 10:34	62K	
zlib-0.93.tar.gz	1995-06-25 01:06	62K	
zlib-0.94.tar.gz	1995-08-13 02:48	66K	
zlib-0.95.tar.gz	1995-08-13 02:48	67K	
zlib-0.99.tar.gz	1996-01-29 12:53	76K	
zlib-1.0-pre.tar.gz	1996-01-15 13:06	70K	
zlib-1.0.1.tar.gz	1996-05-21 10:40	79K	
zlib-1.0.2.tar.gz	1996-05-23 10:11	83K	
zlib-1.0.4.tar.gz	1996-07-24 06:55	84K	
zlib-1.0.5.tar.gz	1998-01-03 17:24	85K	
zlib-1.0.6.tar.gz	1998-01-19 14:28	111K	
zlib-1.0.7.tar.gz	1998-01-20 10:43	113K	
zlib-1.0.8.tar.gz	1998-01-27 11:46	116K	
zlib-1.0.9.tar.gz	1998-02-17 09:33	117K	
zlib-1.1.0.tar.gz	1998-02-24 07:44	118K	
zlib-1.1.1.tar.gz	1998-02-27 13:38	119K	
zlib-1.1.2.tar.gz	1998-03-19 08:25	145K	

图 21.3.1 zlib 源码下载链接

往下翻, 找到一个合适的版本, 这里我们就选择 1.2.10 版本的 zlib:

zlib-1.2.4.5.tar.gz	2010-04-18 12:47	532K
zlib-1.2.4.tar.gz	2010-03-14 18:09	528K
zlib-1.2.5.1.tar.gz	2011-09-10 16:18	536K
zlib-1.2.5.2.tar.gz	2011-12-18 11:45	546K
zlib-1.2.5.3.tar.gz	2012-01-15 20:01	542K
zlib-1.2.5.tar.gz	2010-04-19 21:15	532K
zlib-1.2.6.1.tar.gz	2012-02-12 15:25	544K
zlib-1.2.6.tar.gz	2012-01-29 11:17	544K
zlib-1.2.7.1.tar.gz	2013-03-24 22:52	560K
zlib-1.2.7.2.tar.gz	2013-04-13 18:19	560K
zlib-1.2.7.3.tar.gz	2013-04-13 21:20	560K
zlib-1.2.7.tar.gz	2012-05-02 21:21	547K
zlib-1.2.8.tar.gz	2013-04-28 17:25	558K
zlib-1.2.9.tar.gz	2017-01-01 01:24	593K
zlib-1.2.10.tar.gz	2017-01-02 19:30	593K
zlib-1.2.11.tar.gz	2017-01-15 10:36	593K

图 21.3.2 选择一个版本

点击文件名就可以下载了, 下载成功之后就会得到.tar.gz 格式的压缩文件:

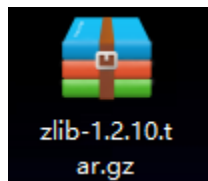
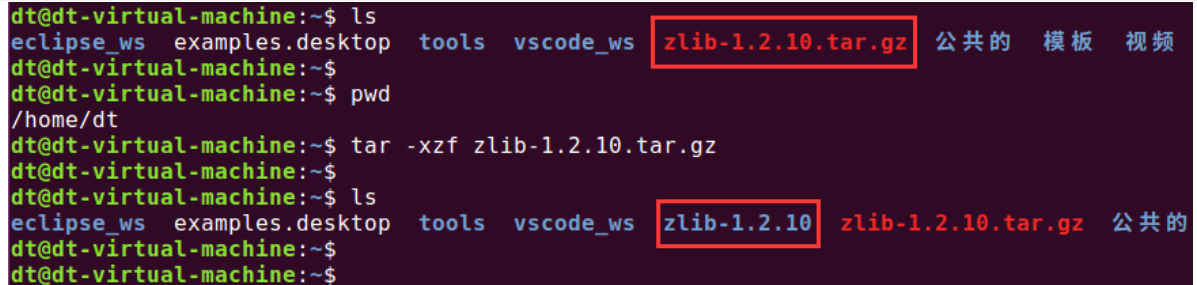


图 21.3.3 zlib-1.2.10.tar.gz 文件

21.3.2 编译源码

将下载的 zlib-1.2.10.tar.gz 压缩文件拷贝到 Ubuntu 系统的用户家目录下, 然后将其解压开:

```
tar -xzf zlib-1.2.10.tar.gz
```

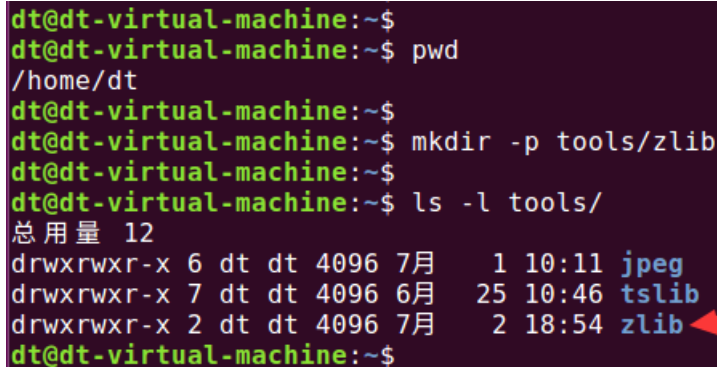


```
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  tools  vscode_ws  zlib-1.2.10.tar.gz  公共的  模板  视频
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$ tar -xzf zlib-1.2.10.tar.gz
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  tools  vscode_ws  zlib-1.2.10  zlib-1.2.10.tar.gz  公共的
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 21.3.4 将 zlib 压缩文件解压

解压之后就会得到 zlib-1.2.10 文件夹, 这就是 zlib 的源代码目录。

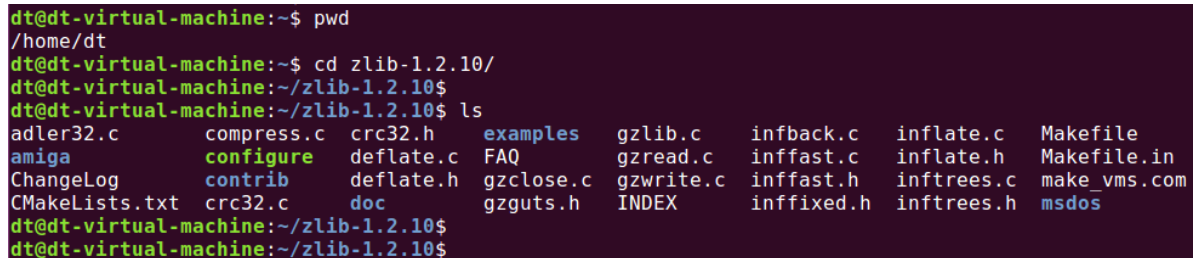
在编译 zlib 之前, 我们先在 tools 目录下创建一个名为 zlib 的文件夹, 作为 zlib 库的安装目录:



```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ mkdir -p tools/zlib
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls -l tools/
总用量 12
drwxrwxr-x 6 dt dt 4096 7月  1 10:11 jpeg
drwxrwxr-x 7 dt dt 4096 6月 25 10:46 tslib
drwxrwxr-x 2 dt dt 4096 7月  2 18:54 zlib
dt@dt-virtual-machine:~$
```

图 21.3.5 创建 zlib 文件夹

接着我们进入到 zlib 的源码目录 zlib-1.2.10, 如下所示:



```
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$ cd zlib-1.2.10/
dt@dt-virtual-machine:~/zlib-1.2.10$
dt@dt-virtual-machine:~/zlib-1.2.10$ ls
adler32.c      compress.c    crc32.h      examples     gzlib.c      inffast.c    inflate.c    Makefile
amiga         configure    deflate.c    FAQ          gzread.c     inffast.c    inflate.h    Makefile.in
ChangeLog     contrib      deflate.h    gzclose.c   gzwrite.c    inffast.h    inftrees.c  make_vms.com
CMakeLists.txt  crc32.c     doc         gzguts.h    INDEX        inffixed.h  inftrees.h  msdos
dt@dt-virtual-machine:~/zlib-1.2.10$
dt@dt-virtual-machine:~/zlib-1.2.10$
```

图 21.3.6 zlib 源码目录下的文件

同样也是执行三部曲: 配置、编译、安装, 一套流程下来就 OK 了!

在此之前, 先对交叉编译工具的环境进行初始化, 使用 source 执行交叉编译工具安装目录下的 environment-setup-cortexa7hf-neon-poky-linux-gnueabi 脚本文件 (如果已经初始化过了, 那就不用再进行初始化了):

```
source /opt/fsl-imx-x11/4.1.15-2.1.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

执行下面这条命令对 zlib 工程进行配置:

```
./configure --prefix=/home/dt/tools/zlib/
```

--prefix 选项指定 zlib 库的安装目录, 将家目录下的 tools/zlib 作为 zlib 库的安装目录。

```
dt@dt-virtual-machine:~/zlib-1.2.10$
dt@dt-virtual-machine:~/zlib-1.2.10$ ./configure --prefix=/home/dt/tools/zlib/
Checking for shared library support...
Building shared library libz.so.1.2.10 with arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 -mfpu=neon -mfloat-abi=hard
Checking for size t... Yes.
Checking for off64 t... Yes.
Checking for fseeko... Yes.
Checking for strerror... Yes.
Checking for unistd.h... Yes.
Checking for stdarg.h... Yes.
Checking whether to use vs[n]printf() or s[n]printf()... using vs[n]printf().
Checking for vsnprintf() in stdio.h... Yes.
Checking for return value of vsnprintf()... Yes.
Checking for attribute(visibility) support... Yes.
dt@dt-virtual-machine:~/zlib-1.2.10$
```

图 21.3.7 配置 zlib 源码工程

配置完成之后, 直接 make 编译:

make

```
dt@dt-virtual-machine:~/zlib-1.2.10$
dt@dt-virtual-machine:~/zlib-1.2.10$ make
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -I. -c -o example.o test/example.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o adler32.o adler32.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o crc32.o crc32.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o deflate.o deflate.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o infback.o infback.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o inffast.o inffast.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o inflate.o inflate.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o inftrees.o inftrees.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o trees.o trees.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o zutil.o zutil.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o compress.o compress.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-
tools/zlib/include -D_LARGEFILE64_SOURCE=1 -DHAVE_HIDDEN -c -o uncompr.o uncompr.c
```

图 21.3.8 make 编译 zlib

编译完成之后, 接着执行 make install 安装即可!

make install

```
dt@dt-virtual-machine:~/zlib-1.2.10$ make install
rm -f /home/dt/tools/zlib//lib/libz.a
cp libz.a /home/dt/tools/zlib//lib
chmod 644 /home/dt/tools/zlib//lib/libz.a
cp libz.so.1.2.10 /home/dt/tools/zlib//lib
chmod 755 /home/dt/tools/zlib//lib/libz.so.1.2.10
rm -f /home/dt/tools/zlib//share/man/man3/zlib.3
cp zlib.3 /home/dt/tools/zlib//share/man/man3
chmod 644 /home/dt/tools/zlib//share/man/man3/zlib.3
rm -f /home/dt/tools/zlib//lib/pkgconfig/zlib.pc
cp zlib.pc /home/dt/tools/zlib//lib/pkgconfig
chmod 644 /home/dt/tools/zlib//lib/pkgconfig/zlib.pc
rm -f /home/dt/tools/zlib//include/zlib.h /home/dt/tools/zlib//include/zconf.h
cp zlib.h zconf.h /home/dt/tools/zlib//include
chmod 644 /home/dt/tools/zlib//include/zlib.h /home/dt/tools/zlib//include/zconf.h
dt@dt-virtual-machine:~/zlib-1.2.10$
```

图 21.3.9

21.3.3 安装目录下的文件夹介绍

进入到 zlib 库的安装目录:


```
dt@dt-virtual-machine:~/zlib-1.2.10$  
dt@dt-virtual-machine:~/zlib-1.2.10$ cd ../tools/zlib/  
dt@dt-virtual-machine:~/tools/zlib$ pwd  
/home/dt/tools/zlib  
dt@dt-virtual-machine:~/tools/zlib$  
dt@dt-virtual-machine:~/tools/zlib$ ls  
include lib share  
dt@dt-virtual-machine:~/tools/zlib$
```

图 21.3.10 zlib 安装目录下的文件夹

头文件目录 include 以及库文件目录 lib。

至此, zlib 库就已经编译好了, 接下来我们需要把编译得到的库文件拷贝到开发板。

21.3.4 移植到开发板

进入到 zlib 安装目录下, 将 lib 目录下的所有动态链接库文件拷贝到开发板 Linux 系统/usr/lib 目录; 注意在拷贝之前, 需要先将出厂系统中原有的 zlib 库文件删除, 在开发板 Linux 系统下执行命令:

```
rm -rf /usr/lib/libz.* /lib/libz.*
```

删除之后, 再将我们编译得到的 zlib 库文件拷贝到开发板/usr/lib 目录, 拷贝库文件时, 需要注意符号链接的问题, 不能破坏原有的符号链接。

拷贝过去之后, 开发板/usr/lib 目录下就应该存在这些库文件, 如下所示:

```
root@ATK-IMX6U:~# ls -l /usr/lib/libz.*  
-rw-r--r-- 1 1000 tracing 146362 Jul 2 2021 /usr/lib/libz.a  
lrwxrwxrwx 1 1000 tracing 14 Jul 2 2021 /usr/lib/libz.so -> libz.so.1.2.10  
lrwxrwxrwx 1 1000 tracing 14 Jul 2 2021 /usr/lib/libz.so.1 -> libz.so.1.2.10  
-rwxr-xr-x 1 1000 tracing 145036 Jul 2 2021 /usr/lib/libz.so.1.2.10  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#
```

图 21.3.11 开发板/usr/lib 目录下的 zlib 库文件

21.4 libpng 移植

移植好 zlib 库之后, 接着我们开始移植 libpng。

21.4.1 下载源码包

首先下载 libpng 源码包, 进入 <https://github.com/glennrp/libpng/releases> 链接地址, 如下:

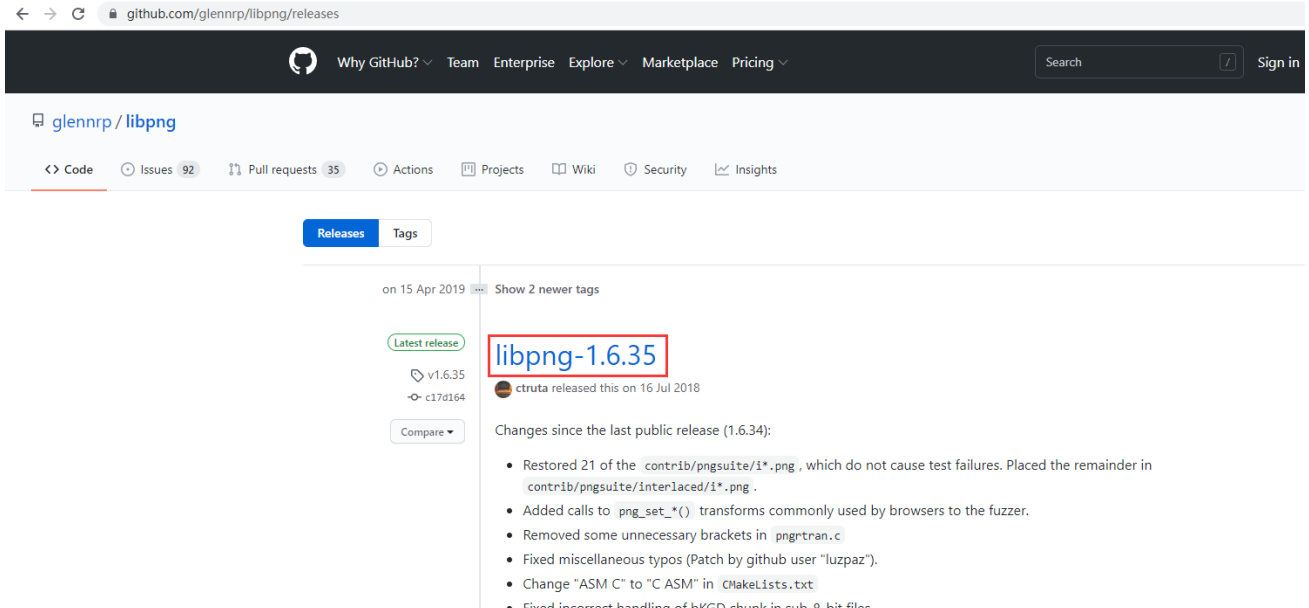


图 21.4.1 libpng 源码下载链接

我们直接下载这个最新的版本 1.6.35，点击下面的 Source Code(tar.gz)压缩文件进行下载：

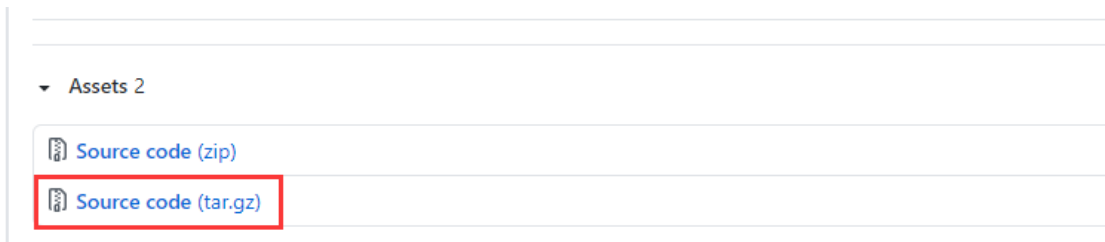


图 21.4.2 选择 1.6.35 版本 tar.gz 压缩文件

下载完成之后，就会得到 libpng 的源码包：

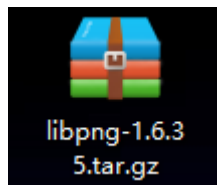


图 21.4.3 libpng 源码包

21.4.2 编译源码

将下载的 libpng-1.6.35.tar.gz 压缩包文件拷贝到 Ubuntu 系统的用户家目录下，接着将其解压：

```
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  libpng-1.6.35.tar.gz  tools  vscode_ws  公共
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ tar -xzf libpng-1.6.35.tar.gz
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  libpng-1.6.35  libpng-1.6.35.tar.gz  tools
dt@dt-virtual-machine:~$
```

图 21.4.4 将 libpng-1.6.35.tar.gz 解压

解压之后得到 libpng-1.6.35 文件夹, 这便是 libpng 的源码目录。

在编译 libpng 之前, 先在 tools 目录下创建一个名为 png 的文件夹, 作为 libpng 库的安装目录:

```
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  libpng-1.6.35  libpng-1.6.35.tar.gz  tools
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ mkdir -p tools/png
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls tools/
jpeg  png  tslib  zlib
dt@dt-virtual-machine:~$
```

图 21.4.5 创建 png 目录

接着我们进入到 libpng 源码目录下, 同样也是执行三部曲: 配置、编译、安装, 一套流程下来就 OK 了!

在此之前, 先对交叉编译工具的环境进行初始化, 使用 source 执行交叉编译工具安装目录下的 environment-setup-cortexa7hf-neon-poky-linux-gnueabi 脚本文件 (如果已经初始化过了, 那就不要再进行初始化了):

```
source /opt/fsl-imx-x11/4.1.15-2.1.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

libpng 依赖于 zlib 库, 前面我们已经将 zlib 库编译成功了, 但是我们得告知编译器 zlib 库的安装目录, 这样编译器才能找到 zlib 的库文件以及头文件, 编译 libpng 的时才不会报错。

执行下面这三条命令, 将 zlib 库安装目录下的 include 和 lib 路径导出到环境变量:

```
export LDFLAGS="${LDFLAGS} -L/home/dt/tools/zlib/lib"
export CFLAGS="${CFLAGS} -I/home/dt/tools/zlib/include"
export CPPFLAGS="${CPPFLAGS} -I/home/dt/tools/zlib/include"
```

```
dt@dt-virtual-machine:~/libpng-1.6.35$
dt@dt-virtual-machine:~/libpng-1.6.35$ export LDFLAGS="${LDFLAGS} -L/home/dt/tools/zlib/lib"
dt@dt-virtual-machine:~/libpng-1.6.35$ export CFLAGS="${CFLAGS} -I/home/dt/tools/zlib/include"
dt@dt-virtual-machine:~/libpng-1.6.35$ export CPPFLAGS="${CPPFLAGS} -I/home/dt/tools/zlib/include"
dt@dt-virtual-machine:~/libpng-1.6.35$
```

图 21.4.6 导出环境变量

接着执行下面这条命令对 libpng 源码工程进行配置:

```
./configure --prefix=/home/dt/tools/png --host=arm-poky-linux-gnueabi
```

--prefix 选项指定 libpng 的安装目录, 将家目录下的 tools/png 作为 libpng 的安装目录。

```
dt@dt-virtual-machine:~/libpng-1.6.35$ ./configure --prefix=/home/dt/tools/png --host=arm-poky-linux-gnueabi
configure: loading site script /opt/fsl-ixml/4.1.15-2.1.0/site-config-cortexa7hf-neon-poky-linux-gnueabi
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for arm-poky-linux-gnueabi-strip... arm-poky-linux-gnueabi-strip
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking whether to enable maintainer-specific portions of Makefiles... no
checking for arm-poky-linux-gnueabi-gcc... arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=
texas7hf-neon-poky-linux-gnueabi
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... yes
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=
gnueabi accepts -g... yes
checking for arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt
fsl-ixml option to accept ISO C89... none needed
checking whether arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=
gnueabi understands -c and -o together... yes
checking whether make supports the include directive... yes (GNU style)
checking dependency style of arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=
gnueabi... gcc3
checking dependency style of arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=
gnueabi... gcc3
```

图 21.4.7 配置 libpng 源码工程

接着执行 make 进行编译:

```
make
```

```
dt@dt-virtual-machine:~/libpng-1.6.35$ make
rm -f pnglibconf.c pnglibconf.tf[45]
mawk -f ./scripts/options.awk out=pnglibconf.tf4 version=search\
./pngconf.h ./scripts/pnglibconf.dfa\
./pngusr.dfa 1>&2
mawk -f ./scripts/options.awk out=pnglibconf.tf5 pnglibconf.tf4 1>&2
rm pnglibconf.tf4
mv pnglibconf.tf5 pnglibconf.c
rm -f pnglibconf.out pnglibconf.tf[12]
test -d scripts || mkdir scripts || test -d scripts
arm-poky-linux-gnueabi-gcc -E -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-
ixml -I. \
-I/home/dt/tools/zlib/include -DPNGLIB_LIBNAME='PNG16_0' -DPNGLIB_VERSION='1.6.35' -DSYMBOL_PREFIX=' -DP
nf.tf1
mawk -f "./scripts/dfn.awk" out="pnglibconf.tf2" pnglibconf.tf1 1>&2
rm -f pnglibconf.tf1
mv pnglibconf.tf2 pnglibconf.out
rm -f pnglibconf.h
cp pnglibconf.out pnglibconf.h
make all-am
make[1]: Entering directory '/home/dt/libpng-1.6.35'
debase=`echo contrib/tools/pngfix.o | sed 's|^[^/]*$|.deps/&;s|\.\o$||'`; \
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-ixml
-I. -I/home/dt/tools/zlib/include -I/home/dt/tools/zlib/include -MT contrib/tools/pngfix.o -MD -MP -M
mv -f $debase.Tpo $debase.Po
:>pngprefix.h
debase=`echo png.lo | sed 's|^[^/]*$|.deps/&;s|\.\lo$||'`; \
```

图 21.4.8 make 编译

最后执行 make install 安装即可!

```
make install
```

```

dt@dt-virtual-machine:~/libpng-1.6.35$ make install
make install-am
make[1]: Entering directory '/home/dt/libpng-1.6.35'
make[2]: Entering directory '/home/dt/libpng-1.6.35'
/bin/mkdir -p '/home/dt/tools/png/lib'
/bin/bash ./libtool --mode=install /usr/bin/install -c libpng16.la '/home/dt/tools/png/lib'
libtool: install: /usr/bin/install -c .libs/libpng16.so.16.35.0 /home/dt/tools/png/lib/libpng16.so.16.35.0
libtool: install: (cd /home/dt/tools/png/lib && { ln -s -f libpng16.so.16.35.0 libpng16.so.16 || { rm -f libpng16.
libtool: install: (cd /home/dt/tools/png/lib && { ln -s -f libpng16.so.16.35.0 libpng16.so || { rm -f libpng16.
libtool: install: /usr/bin/install -c .libs/libpng16.lai /home/dt/tools/png/lib/libpng16.la
libtool: install: /usr/bin/install -c .libs/libpng16.a /home/dt/tools/png/lib/libpng16.a
libtool: install: chmod 644 /home/dt/tools/png/lib/libpng16.a
libtool: install: arm-poky-linux-gnueabi-ranlib /home/dt/tools/png/lib/libpng16.a
libtool: finish: PATH="/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin:/opt/fsl-imx-x11/4.1
2.1.0/sysroots/x86_64-pokysdk-linux/usr/sbin:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/bin:/o
imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/./x86_64-pokysdk-linux/bin:/opt/fsl-imx-x11/4.1.15-2
fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-uclibc:/opt/fsl-imx-x11/4.1.15-2
8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/
ib
ldconfig: /home/dt/tools/png/lib/libpng16.so is for unknown machine 40.

ldconfig: /home/dt/tools/png/lib/libpng16.so.16 is for unknown machine 40.

ldconfig: /home/dt/tools/png/lib/libpng16.so.16.35.0 is for unknown machine 40.

-----
Libraries have been installed in:
/home/dt/tools/png/lib

```

图 21.4.9 make install 安装

21.4.3 安装目录下的文件夹介绍

进入到 libpng 安装目录:

```

dt@dt-virtual-machine:~$ cd tools/png/
dt@dt-virtual-machine:~/tools/png$
dt@dt-virtual-machine:~/tools/png$ pwd
/home/dt/tools/png
dt@dt-virtual-machine:~/tools/png$
dt@dt-virtual-machine:~/tools/png$ ls
bin include lib share
dt@dt-virtual-machine:~/tools/png$

```

图 21.4.10 libpng 安装目录下的文件夹

同样包含了 bin、include、lib 这些目录。

21.4.4 移植到开发板

进入到 libpng 安装目录, 将 bin 目录下的所有测试工具拷贝到开发板 Linux 系统/usr/bin 目录; 将 lib 目录下的所有库文件拷贝到 Linux 系统/usr/lib 目录, 注意在拷贝之前, 先将开发板出厂系统中已经移植好的 libpng 库文件删除, 执行下面这条命令:

```
rm -rf /lib/libpng* /usr/lib/libpng*
```

删除之后, 再将编译得到的 libpng 库文件拷贝到开发板/usr/lib 目录, 拷贝库文件时, 需要注意符号链接的问题, 不能破坏原有的符号链接。

拷贝过去之后, 开发板/usr/lib 目录下就应该存在这些库文件, 如下所示:

```

root@ATK-IMX6U:~# ls -l /usr/lib/libpng*
-rw-r--r-- 1 1000 tracing 350982 Jul 3 2021 /usr/lib/libpng16.a
-rwxr-xr-x 1 1000 tracing 971 Jul 3 2021 /usr/lib/libpng16.la
lrwxrwxrwx 1 1000 tracing 19 Jul 3 2021 /usr/lib/libpng16.so -> libpng16.so.16.35.0
lrwxrwxrwx 1 1000 tracing 19 Jul 3 2021 /usr/lib/libpng16.so.16 -> libpng16.so.16.35.0
-rwxr-xr-x 1 1000 tracing 307236 Jul 3 2021 /usr/lib/libpng16.so.16.35.0
lrwxrwxrwx 1 1000 tracing 10 Jul 3 2021 /usr/lib/libpng.a -> libpng16.a
lrwxrwxrwx 1 1000 tracing 11 Jul 3 2021 /usr/lib/libpng.la -> libpng16.la
lrwxrwxrwx 1 1000 tracing 11 Jul 3 2021 /usr/lib/libpng.so -> libpng16.so
root@ATK-IMX6U:~#

```

图 21.4.11 开发板/usr/lib 目录下的 libpng 相关库文件

21.5 libpng 使用说明

本小节向大家简单地介绍如何使用 libpng 对 png 图像进行解码, libpng 除了解码功能之外, 还包含编码功能, 也就是创建 png 压缩文件, 当然, 这个笔者就不再介绍了。libpng 官方提供一份非常详细地使用文档, 笔者也是参考了这份文档给大家进行介绍的, 这份文档的链接地址如下:

<http://www.libpng.org/pub/png/libpng-1.4.0-manual.pdf>

<http://www.libpng.org/pub/png/libpng-manual.txt>

这两份文档的内容是一样的, 第一份是 pdf 文档、第二份是 txt 文档, 如果大家想更加深入的了解、学习, 那么可以查阅这份文档。

21.5.1 libpng 的数据结构

首先, 使用 libpng 库需要包含它的头文件 <png.h>。png.h 头文件中包含了 API、数据结构的申明, libpng 中有两个很重要的数据结构体: png_struct 和 png_info。

png_struct 作为 libpng 库函数内部使用的一个数据结构体, 除了作为传递给每个 libpng 库函数调用的第一个变量外, 在大多数情况下不会被用户所使用。使用 libpng 之前, 需要创建一个 png_struct 对象并对其进行初始化操作, 该对象由 libpng 库内部使用, 调用 libpng 库函数时, 通常需要把这个对象作为参数传入。

png_info 数据结构体描述了 png 图像的信息, 在以前旧的版本中, 用户可以直接访问 png_info 对象中的成员, 譬如查看图像的宽、高、像素深度、修改解码参数等; 然而, 这往往会导致出现一些问题, 因此新的版本中专门开发了一组 png_info 对象的访问接口: get 方法 png_get_XXX 和 set 方法 png_set_XXX, 建议大家通过 API 来访问这些成员。

21.5.2 创建和初始化 png_struct 对象

首先第一步是创建 png_struct 对象、并对其进行初始化操作, 使用 png_create_read_struct() 函数创建一个 png_struct 对象、并完成初始化操作, read 表示我们需要创建的是一个用于 png 解码的 png_struct 对象; 同理可以使用 png_create_write_struct() 创建一个用于 png 编码的 png_struct 对象。

png_create_read_struct 函数原型如下所示:

```

png_structp png_create_read_struct(png_const_charp user_png_ver, png_voidp error_ptr, png_error_ptr error_fn,
png_error_ptr warn_fn);

```

它的是返回值是一个 png_structp 指针, 指向一个 png_struct 对象; 所以 png_create_read_struct() 函数创建 png_struct 对象之后, 会返回一个指针给调用者, 该指针指向所创建的 png_struct 对象。但如果创建对象失败, 则会返回 NULL, 所以调用者可以通过判断返回值是否为 NULL 来确定 png_create_read_struct() 函数执行是否成功!

该函数有 4 个参数, 第一个参数 user_png_ver 指的是 libpng 的版本信息, 通常将其设置为 PNG_LIBPNG_VER_STRING, 这是 png.h 头文件中定义的一个宏, 其内容便是 libpng 的版本号信息, 如下:

```

#define PNG_LIBPNG_VER_STRING "1.6.35"

```

创建、初始化 `png_struct` 对象时,调用者可以指定自定义的错误处理函数和自定义的警告处理函数,通过参数 `error_fn` 指向自定义的错误处理函数、通过参数 `warn_fn` 指向自定义的警告处理函数,而参数 `error_ptr` 表示传递给这些函数所使用的数据结构的指针;当然也可将它们设置为 `NULL`,表示使用 `libpng` 默认的错误处理函数以及警告函数。使用示例如下:

```
png_structp png_ptr = NULL;
png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
if (!png_ptr)
    return -1;
```

21.5.3 创建和初始化 `png_info` 对象

`png_info` 数据结构体描述了 `png` 图像的信息,同样也需要创建 `png_info` 对象,调用 `png_create_info_struct()` 函数创建一个 `png_info` 对象,其函数原型如下所示:

```
png_infop png_create_info_struct(png_const_structp png_ptr);
```

该函数返回一个 `png_infop` 指针,指向一个 `png_info` 对象,所以 `png_create_info_struct()` 函数创建 `png_info` 对象之后,会将它的指针返回给调用者;如果创建失败,则会返回 `NULL`,所以调用者可以通过判断返回值是否为 `NULL` 来确定函数调用是否成功!

该函数有一个参数,需要传入一个 `png_struct` 对象的指针,内部会将它们之间建立关联,当销毁 `png_struct` 对象时,也可将 `png_info` 对象销毁。使用示例如下:

```
png_infop info_ptr = NULL;
info_ptr = png_create_info_struct(png_const_structp png_ptr);
if (NULL == info_ptr) {
    png_destroy_read_struct(&png_ptr, NULL, NULL);
    return -1;
}
```

`png_destroy_read_struct()` 函数用于销毁 `png_struct` 对象的函数,后面再给大家介绍。

21.5.4 设置错误返回点

调用 `png_create_read_struct()` 函数创建 `png_struct` 对象时,调用者可以指定一个自定义的错误处理函数,当 `libpng` 工作发生错误时,它就会执行这个错误处理函数;但如果调用者并未指定自定义的错误处理函数,那么 `libpng` 将会使用默认的错误处理函数,其实默认的错误处理函数会执行一个跳转动作,跳转到程序中的某一个位置,我们把这个位置称为错误返回点。

这样,当调用者未指定自定义错误处理函数时,当 `libpng` 遇到错误时,它会执行默认错误处理函数,而默认错误处理函数会跳转到错误返回点,通常这个错误返回点就是在我们程序中的某个位置,我们期望 `libpng` 发生错误时能够回到我们的程序中,为什么要这样做呢?因为发生错误时不能直接终止退出,而需要执行释放、销毁等清理工作,譬如前面创建的 `png_struct` 和 `png_info` 对象,需要销毁,避免内存泄漏。

那如何在我们的程序中设置错误返回点呢?在此之前,笔者需要向大家介绍两个库函数: `setjmp` 和 `longjmp`。

setjmp 和 longjmp

在 C 语言中,在一个函数中执行跳转,我们可以使用 `goto` 语句,笔者也经常使用 `goto` 语句,尤其是在开发驱动程序时;但 `goto` 语句只能在一个函数内部进行跳转,不能跨越函数,譬如从 `func1()` 函数跳转到 `func2()` 函数,如果想要实现这种跨越函数间的跳转,在 Linux 下,我们可以使用库函数 `setjmp` 和 `longjmp`。

setjmp 函数用于设置跳转点, 也就是跳转位置; longjmp 执行跳转, 那么它会跳转到 setjmp 函数所设置的跳转点, 来看看这两个函数的原型:

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

可以看到 setjmp 和 longjmp 函数都有一个 env 参数, 这是一个 jmp_buf 类型的参数, jmp_buf 是一种特殊类型, 当调用 setjmp() 时, 它会把当前进程环境的各种信息保存到 env 参数中, 而调用 longjmp() 也必须指定相同的参数, 这样才可跳转到 setjmp 所设置的跳转点。

从编程角度来看, 调用 longjmp() 函数后, 看起来就和第二次调用 setjmp() 返回时完全一样, 可以通过检查 setjmp() 函数的返回值, 来区分 setjmp() 是初次调用返回还是第二次“返回”, 初始调用返回值为 0, 后续“伪”返回的返回值为 longjmp() 调用中参数 val 所指定的任意值, 通过对 val 参数使用不同的值, 可以区分出程序中跳转到同一位置的多个不同的起跳位置。

所以, 通常情况下, 调用 longjmp() 时, 不会将参数 val 设置为 0, 这样将会导致无法区分 setjmp() 是初次返回还是后续的“伪”返回, 这里大家要注意!

好, 那么关于这两个函数就向大家介绍这么多, 我们来看一个例子:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->21_libpng->setjmp.c](#)。

示例代码 21.5.1 setjmp/longjmp 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

static jmp_buf buf;

static void hello(void)
{
    printf("hello world!\n");
    longjmp(buf, 1);
    printf("Nice to meet you!\n");
}

int main(void)
{
    if(0 == setjmp(buf)) {
        printf("First return\n");
        hello();
    }
    else
        printf("Second return\n");

    exit(0);
}
```

我们直接在 Ubuntu 系统下编译运行, 运行结果如下所示:


```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
First return
hello world!
Second return
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 21.5.1 测试程序运行结果

打印结果就不再分析了,上面给大家讲解地非常清楚了。

libpng 设置错误返回点

libpng 库默认也使用 setjmp/longjmp 这两个库函数组合来处理发生错误时的跳转,当 libpng 遇到错误时,执行默认错误处理函数,默认错误处理函数会调用 longjmp()来进行跳转,所以我们需要使用 setjmp()来为 libpng 设置一个错误返回点。设置方法如下:

```
/* 设置错误返回点 */
if (setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
    return -1;
}
```

png_jmpbuf()函数可以获取到 png_struct 对象中的 jmp_buf 变量,那么后续 libpng 库调用 longjmp 执行跳转时也是使用这个变量。我们可以在错误返回点执行一些清理工作。

21.5.5 指定数据源

也就是指定需要进行解码的 png 图像,通常可以使用多种方式来指定数据源,譬如文件输入流、内存中的数据流等,这里笔者以文件输入流为例。

libpng 提供了 png_init_io()函数, png_init_io()可以指定数据源,该数据源以文件输入流的方式提供,来看看函数原型:

```
png_init_io(png_structrp png_ptr, png_FILE_p fp);
```

第一个参数是 png_ptr,指向 png_struct 对象;而第二个参数 fp 则是一个 png_FILE_p 类型指针,其实就是标准 I/O 中的 FILE *指针。所以由此可知,我们需要先使用 fopen()函数将 png 文件打开,然后得到指向该文件的 FILE *类型指针。

使用示例如下:

```
FILE *png_file = NULL;

/* 打开 png 文件 */
png_file = fopen("image.png", "r"); //以只读方式打开
if (NULL == png_file) {
    perror("fopen error");
    return -1;
}

/* 指定数据源 */
png_init_io(png_ptr, png_file);
```

21.5.6 读取 png 图像数据并解码

从 png 文件中读取数据并进行解码, 将解码后的图像数据存放在内存中, 待用户读取。关于这一步的操作, libpng 提供了两种方式去处理: high-level 接口处理和 low-level 接口处理。其实 high-level 只是对 low-level 方式进行了一个封装, 使用 high-level 接口非常方便只需一个函数即可, 但缺点是灵活性不高、被限定了; 而 low-level 接口恰好相反, 灵活性高、但需要用户调用多个 API; 所以具体使用哪种方式要看你的需求。

high-level 接口

通常在满足以下两个条件时使用 high-level 接口:

- 用户的内存空间足够大, 可以一次性存放整个 png 文件解码后的数据;
- 数据输出格式限定为 libpng 预定义的数据转换格式。

在满足以上两个条件时, 可以使用 high-level 接口, libpng 预定义数据转换类型包括:

libpng 预定义转换类型	说明
PNG_TRANSFORM_IDENTITY	No transformation
PNG_TRANSFORM_STRIP_16	Strip 16-bit samples to 8 bits
PNG_TRANSFORM_STRIP_ALPHA	Discard the alpha channel
PNG_TRANSFORM_PACKING	Expand 1, 2 and 4-bit samples to bytes
PNG_TRANSFORM_PACKSWAP	Change order of packed pixels to LSB first
PNG_TRANSFORM_EXPAND	Perform set_expand()
PNG_TRANSFORM_INVERT_MONO	Invert monochrome images
PNG_TRANSFORM_SHIFT	Normalize pixels to the sBIT depth
PNG_TRANSFORM_BGR	Flip RGB to BGR, RGBA to BGRA
PNG_TRANSFORM_SWAP_ALPHA	Flip RGBA to ARGB or GA to AG
PNG_TRANSFORM_INVERT_ALPHA	Change alpha from opacity to transparency
PNG_TRANSFORM_SWAP_ENDIAN	Byte-swap 16-bit samples
PNG_TRANSFORM_GRAY_TO_RGB	Expand grayscale samples to RGB (or GA to RGBA)

表 21.5.1 libpng 预定义的数据转换类型

后面的注释说明大家自己去翻译, 我怕翻译错了, 把你们带入坑!

这些转换当中, 还不包括背景颜色设置(透明图)、伽马变换、抖动和填充物等, 使用 high-level 接口只能使用以上这些预定义的转换类型, 而其它的配置则保持默认。

high-level 接口只需要使用一个函数 `png_read_png()`, 调用该函数将一次性把整个 png 文件的图像数据解码出来、将解码后的数据存放在内存中, 如下所示:

```
png_read_png(png_structrp png_ptr, png_inforp info_ptr, int transforms, png_voidp params);
```

第一个参数 `png_ptr` 为指向 `png_struct` 对象的指针, 第二个参数 `info_ptr` 为指向 `png_info` 对象的指针; 而第三个参数 `transforms` 为整型参数, 取值为上表所列出的 libpng 预定义的数据转换类型, 可以使用 `or` (C 语言的或 | 运算符) 组合多个转换类型。使用示例如下:

```
png_read_png(png_ptr, info_ptr, PNG_TRANSFORM_STRIP_ALPHA, NULL);
```

该函数相当于调用一系列 low-level 函数(下文将会介绍), 调用顺序如下所示:

- (1)、调用 `png_read_info` 函数获得 png 图像信息;
- (2)、根据参数 `transforms` 所指定的转换类型对数据输出转换格式进行设置;
- (3)、调用 `png_read_image` 一次性把整个 png 文件的图像数据解码出来、并将解码后的数据存放在内存中。

(4)、调用 `png_read_end` 结束解码。

low-level 接口

使用 low-level 接口, 需要用户将函数 `png_read_png()` 所做的事情一步一步执行:

a)、读取 png 图像的信息

首先我们要调用 `png_read_info()` 函数获取 png 图像的信息:

```
png_read_info(png_ptr, info_ptr);
```

该函数会把 png 图像的信息读入到 `info_ptr` 指向的 `png_info` 对象中。

b)、查询图像的信息

前面提到 `png_read_info()` 函数会把 png 图像的信息读入到 `png_info` 对象中, 接下来我们可以调用 libpng 提供的 API 查询这些信息。

```
unsigned int width = png_get_image_width(png_ptr, info_ptr); //获取 png 图像的宽度
unsigned int height = png_get_image_height(png_ptr, info_ptr); //获取 png 图像的高度
unsigned char depth = png_get_bit_depth(png_ptr, info_ptr); //获取 png 图像的位深度
unsigned char color_type = png_get_color_type(png_ptr, info_ptr); //获取 png 图像的颜色类型
```

`color type` 在 `png.h` 头文件中定义, 如下所示:

```
/* These describe the color_type field in png_info. */
/* color type masks */
#define PNG_COLOR_MASK_PALETTE 1
#define PNG_COLOR_MASK_COLOR 2
#define PNG_COLOR_MASK_ALPHA 4

/* color types. Note that not all combinations are legal */
#define PNG_COLOR_TYPE_GRAY 0
#define PNG_COLOR_TYPE_PALETTE (PNG_COLOR_MASK_COLOR | PNG_COLOR_MASK_PALETTE)
#define PNG_COLOR_TYPE_RGB (PNG_COLOR_MASK_COLOR)
#define PNG_COLOR_TYPE_RGB_ALPHA (PNG_COLOR_MASK_COLOR | PNG_COLOR_MASK_ALPHA)
#define PNG_COLOR_TYPE_GRAY_ALPHA (PNG_COLOR_MASK_ALPHA)
/* aliases */
#define PNG_COLOR_TYPE_RGBA PNG_COLOR_TYPE_RGB_ALPHA
#define PNG_COLOR_TYPE_GA PNG_COLOR_TYPE_GRAY_ALPHA
```

c)、设置解码输出参数 (转换参数)

这步非常重要, 用户可以指定数据输出转换的格式, 比如 `RGB888`, `BGR888`、`ARGB8888` 等数据输出格式, libpng 提供了很多 `set` 方法 (`png_set_xxxxx` 函数) 来实现这些设置, 例如如下代码:

```
unsigned char depth = png_get_bit_depth(png_ptr, info_ptr);
unsigned char color_type = png_get_color_type(png_ptr, info_ptr);

if (16 == depth)
    png_set_strip_16(png_ptr); //将 16 位深度转为 8 位深度

if (8 > depth)
    png_set_expand(png_ptr); //如果位深小于 8, 则扩展为 24-bit RGB
```

```
if (PNG_COLOR_TYPE_GRAY_ALPHA == color_type)
```

```
    png_set_gray_to_rgb(png_ptr); //如果是灰度图, 则转为 RGB
```

关于这些函数的作用和使用方法, 大家可以打开 libpng 的头文件 png.h 进行查看, 每个函数它都有相应的注释信息以及参数列表。如上我们列举了几个 png_set_xxx 转换函数, 这种转换函数还很多, 这里便不再一一进行介绍, 具体请查看 libpng 的使用手册以了解他们的作用。

虽然 libpng 提供了很多转换函数, 可以调用它们对数据的输出格式进行设置, 但是用户的需求是往往无限的, 很多输出格式 libpng 并不是原生支持的, 譬如 YUV565、RGB565、YUYV 等, 为了解决这样的问题, libpng 允许用户设置自定义转换函数, 可以让用户注册自定义转换函数给 libpng 库, libpng 库对输出数据进行转换时, 会调用用户注册的自定义转换函数进行转换。

调用者通过 png_set_read_user_transform_fn() 函数向 libpng 注册一个自定义转换函数, 另外调用者还可以通过 png_set_user_transform_info() 函数告诉 libpng 自定义转换函数的用户自定义数据结构和输出数据的详细信息, 比如颜色深度、颜色通道 (channel) 等等。关于这些内容, 大家自己去查阅 libpng 的使用帮助文档。

d)、更新 png 数据的详细信息

经过前面的设置之后, 信息肯定会有一些变化, 我们需要调用 png_read_update_info 函数更新信息:

```
png_read_update_info(png_ptr, info_ptr);
```

该函数将会更新保存在 info_ptr 指向的 png_info 对象中的图像信息。

e)、读取 png 数据并解码

前面设置完成之后, 接下来便可对 png 文件的数据进行解码了。调用 png_read_image() 函数可以一次性把整个 png 文件的图像数据解码出来、并将解码后的数据存放在用户提供的内存区域中, 使用示例如下:

```
png_read_image(png_ptr, row_pointers);
```

该函数无返回值, 参数 png_ptr 指向 png_struct 对象; 第二个参数 row_pointers 是一个 png_bytepp 类型的指针变量, 也就是 unsigned char **, 是一个指针数组, 如下所示:

```
png_bytep row_pointers[height];
```

调用该函数, 需要调用者提供足够大的内存空间, 可以保存整个图像的数据, 这个内存空间的大小通常是解码后数据的总大小; 调用者分配内存空间后, 需要传入指向每一行的指针数组, 如下所示:

```
png_bytep row_pointers[height] = {0};
```

```
size_t rowbytes = png_get_rowbytes(png_ptr, info_ptr); //获取每一行数据的字节大小
```

```
int row;
```

```
/* 为每一行数据分配一个缓冲区 */
```

```
for (row = 0; row < height; row++)
```

```
    row_pointers[row] = png_malloc(png_ptr, rowbytes);
```

```
png_read_image(png_ptr, row_pointers);
```

Tips: png_malloc() 函数是 libpng 提供的一个 API, 其实就等价于库函数 malloc。

除了 png_read_image() 函数之外, 我们也可以调用 png_read_rows() 一次解码 1 行或多行数据、并将解码后的数据存放在用于提供的内存区域中, 譬如:

```
size_t rowbytes = png_get_rowbytes(png_ptr, info_ptr); //获取每一行数据的字节大小
```

```
png_bytep row_buf = png_malloc(png_ptr, rowbytes); //分配缓冲、用于存储一行数据
```

```
int row;
```

```
for (row = 0; row < height; row++) {
```

```

png_read_rows(png_ptr, &row_buf, NULL, 1); //每次读取、解码一行数据(最后一个数字 1 表示每次 1 行)

/* 对这一行数据进行处理: 譬如刷入 LCD 显存进行显示 */
do_something();
}

```

png_read_rows 会自动跳转处理下一行数据。

由此可知, 在 low-level 接口, 调用 png_read_image()或 png_read_rows()函数都需要向 libpng 提供用于存放数据的内存区域。但是在 high-level 接口中, 调用 png_read_png()时我们并不需要自己分配缓冲区, png_read_png()函数内部会自动分配一块缓冲区, 那我们如何获取到它分配的缓冲区呢? 通过 png_get_rows()函数得到, 下小节介绍。

f)、png_read_end()结束读取、解码

当整个 png 文件的数据已经读取、解码完成之后, 我们可以调用 png_read_end()结束, 代码如下:

```
png_read_end(png_ptr, info_ptr);
```

21.5.7 读取解码后的数据

解码完成之后, 我们便可以去获取解码后的数据了, 要么那它们做进一步的处理、要么直接刷入显存显示到 LCD 上; 对于 low-level 方式, 存放图像数据的缓冲区是由调用者分配的, 所以直接从缓冲区中获取数据即可!

对于 high-level 方式, 存放图像数据的缓冲区是由 png_read_png()函数内部所分配的, 并将缓冲区与 png_struct 对象之间建立了关联, 我们可以通过 png_get_rows()函数获取到指向每一行数据缓冲区的指针数组, 如下所示:

```
png_bytepp row_pointers = NULL;
```

```
row_pointers = png_get_rows(png_ptr, info_ptr); //获取到指向每一行数据缓冲区的指针数组
```

当我们销毁 png_struct 对象时, 由 png_read_png()所分配的缓冲区也会被释放归还给操作系统。

21.5.8 结束销毁对象

调用 png_destroy_read_struct()销毁 png_struct 对象, 该函数原型如下所示:

```
void png_destroy_read_struct(png_structpp png_ptr_ptr, png_infopp info_ptr_ptr, png_infopp end_info_ptr_ptr);
```

使用方法如下:

```
png_destroy_read_struct(png_ptr, info_ptr, NULL);
```

21.6 libpng 应用编程

经过上一小节的介绍后, 相信大家已经知道了如何使用 libpng 库对 png 图像进行解码, 本小节我们将进行实战, 使用 libpng 库对一张指定的 png 图像进行解码, 并在 LCD 上显示图像。示例代码如下所示:

本例程源码对应的路径为: **开发板光盘->11、Linux C 应用编程例程源码->21_libpng->show_png_image.c。**

示例代码 21.6.1 libpng 应用编程示例代码

```

/*****

```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : show_png_image.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

描述 : libpng 使用实战

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/6/15 邓涛创建

```

*****/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <png.h>

static int width;           //LCD X 分辨率
static int height;         //LCD Y 分辨率
static unsigned short *screen_base = NULL; //映射后的显存基地址
static unsigned long line_length; //LCD 一行的长度 (字节为单位)
static unsigned int bpp;    //像素深度 bpp

static int show_png_image(const char *path)
{
    png_structp png_ptr = NULL;
    png_infop info_ptr = NULL;
    FILE *png_file = NULL;
    unsigned short *fb_line_buf = NULL; //行缓冲区:用于存储写入到 LCD 显存的一行数据
    unsigned int min_h, min_w;
    unsigned int valid_bytes;
    unsigned int image_h, image_w;
    png_bytepp row_pointers = NULL;
    int i, j, k;

    /* 打开 png 文件 */
    png_file = fopen(path, "r"); //以只读方式打开
    if (NULL == png_file) {
        perror("fopen error");
        return -1;
    }
}

```

```
/* 分配和初始化 png_ptr、info_ptr */
png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
if (!png_ptr) {
    fclose(png_file);
    return -1;
}

info_ptr = png_create_info_struct(png_ptr);
if (!info_ptr) {
    png_destroy_read_struct(&png_ptr, NULL, NULL);
    fclose(png_file);
    return -1;
}

/* 设置错误返回点 */
if (setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
    fclose(png_file);
    return -1;
}

/* 指定数据源 */
png_init_io(png_ptr, png_file);

/* 读取 png 文件 */
png_read_png(png_ptr, info_ptr, PNG_TRANSFORM_STRIP_ALPHA, NULL);
image_h = png_get_image_height(png_ptr, info_ptr);
image_w = png_get_image_width(png_ptr, info_ptr);
printf("分辨率: %d*%d\n", image_w, image_h);

/* 判断是不是 RGB888 */
if ((8 != png_get_bit_depth(png_ptr, info_ptr)) &&
    (PNG_COLOR_TYPE_RGB != png_get_color_type(png_ptr, info_ptr))) {
    printf("Error: Not 8bit depth or not RGB color");
    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
    fclose(png_file);
    return -1;
}

/* 判断图像和 LCD 屏那个的分辨率更低 */
if (image_w > width)
    min_w = width;
else
```

```
min_w = image_w;

if (image_h > height)
    min_h = height;
else
    min_h = image_h;

valid_bytes = min_w * bpp / 8;

/* 读取解码后的数据 */
fb_line_buf = malloc(valid_bytes);
row_pointers = png_get_rows(png_ptr, info_ptr); //获取数据

unsigned int temp = min_w * 3; //RGB888 一个像素 3 个 bit 位
for(i = 0; i < min_h; i++) {

    // RGB888 转为 RGB565
    for(j = k = 0; j < temp; j += 3, k++)
        fb_line_buf[k] = ((row_pointers[i][j] & 0xF8) << 8) |
            ((row_pointers[i][j+1] & 0xFC) << 3) |
            ((row_pointers[i][j+2] & 0xF8) >> 3);

    memcpy(screen_base, fb_line_buf, valid_bytes); //将一行数据刷入显存
    screen_base += width; //定位到显存下一行
}

/* 结束、销毁/释放内存 */
png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
free(fb_line_buf);
fclose(png_file);
return 0;
}

int main(int argc, char *argv[])
{
    struct fb_fix_screeninfo fb_fix;
    struct fb_var_screeninfo fb_var;
    unsigned int screen_size;
    int fd;

    /* 传参校验 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <png_file>\n", argv[0]);
    }
}
```



```
        exit(-1);
    }

    /* 打开 framebuffer 设备 */
    if (0 > (fd = open("/dev/fb0", O_RDWR))) {
        perror("open error");
        exit(EXIT_FAILURE);
    }

    /* 获取参数信息 */
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

    line_length = fb_fix.line_length;
    bpp = fb_var.bits_per_pixel;
    screen_size = line_length * fb_var.yres;
    width = fb_var.xres;
    height = fb_var.yres;

    /* 将显示缓冲区映射到进程地址空间 */
    screen_base = mmap(NULL, screen_size, PROT_WRITE, MAP_SHARED, fd, 0);
    if (MAP_FAILED == (void *)screen_base) {
        perror("mmap error");
        close(fd);
        exit(EXIT_FAILURE);
    }

    /* 显示 BMP 图片 */
    memset(screen_base, 0xFF, screen_size); // 屏幕刷白
    show_png_image(argv[1]);

    /* 退出 */
    munmap(screen_base, screen_size); // 取消映射
    close(fd); // 关闭文件
    exit(EXIT_SUCCESS); // 退出进程
}
```

代码不再进行讲解, 示例代码中所使用到的函数都已经给大家介绍过, 上述示例代码使用的是 high-level 接口处理方式, 直接调用了 `png_read_png`, 一次性把整个 png 文件的数据解码出来, 由于得到的数据是 RGB888 格式, 所以我们需要将其转为 RGB565, 转换完成之后将其刷入到显存中。

接下来我们编译示例代码, 这里要注意下, 使用交叉编译器编译代码时, 需要指定 `libpng` 库和 `zlib` 库, 如下所示:

```
{CC} -o testApp testApp.c -I/home/dt/tools/png/include -L/home/dt/tools/png/lib -L/home/dt/tools/zlib/lib -lpng -lz
```

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ gcc -o testApp testApp.c -I/home/dt/tools/png/include -L/home/dt/tools/png/lib -L/home/dt/tools/zlib/lib -lpng -lz
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
```

图 21.6.1 编译示例代码

使用-I选项指定 libpng 的头文件（也就是安装目录下的 include 目录），不需要指定 zlib 的头文件；使用了两次-L选项，分别指定了 libpng 和 zlib 的库目录（也就是安装目录下的 lib 目录）；再使用-l选项指定需要链接的库（z 表示 libz.so、png 表示 libpng.so）。

将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下，并准备一个 png 文件，接着执行测试程序（执行测试程序前，先关闭出厂系统的 Qt GUI 应用程序）：

```
root@ATK-IMX6U:~# ls
driver image.png shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp image.png
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: cHRM chunk does not match sRGB
png图像分辨率: 853*480
root@ATK-IMX6U:~#
```

图 21.6.2 执行测试程序

可以看到打印出了一些警告信息，原因是新版本的 libpng 增强了检查，发出了警告；不过这并不影响我们的使用，可以忽略。

此时开发板 LCD 上会显示我们指定的 png 图像，如下所示：



图 21.6.3 LCD 显示 png 图像

本章内容到此结束！

第二十二章 LCD 横屏切换为竖屏

之前在技术交流群里边有人提到了 LCD 横屏切换为竖屏的问题,笔者觉得还是很有必要给大家讲一下,所以这里单独做一章内容来讲一讲怎么样实现 LCD 横屏切换为竖屏,其实个人觉得还是非常简单地。首先给大家普及一个基本的知识点,这种横屏、竖屏的切换与驱动程序无关,是应用层需要去解决的一个问题!

本章将会讨论如下主题。

- 横屏显示如何切换为竖屏显示;
- 编写代码验证;

22.1 横屏显示如何切换为竖屏显示

开发板配套使用的这些 LCD 屏都是横屏显示的, 包括正点原子 4.3 寸 480*272、4.3 寸 800*480、7 寸 800*480、7 寸 1024*600 以及 10.1 寸 1280*800 等这些 RGB LCD 屏; LCD 屏正向放置情况下 (以 800*480 分辨率为例), 它的左上角就是坐标(0, 0)、左下角坐标是(0, 480-1)、右上角坐标是(800-1, 0)、右下角坐标是(800-1, 480-1), 如下所示:

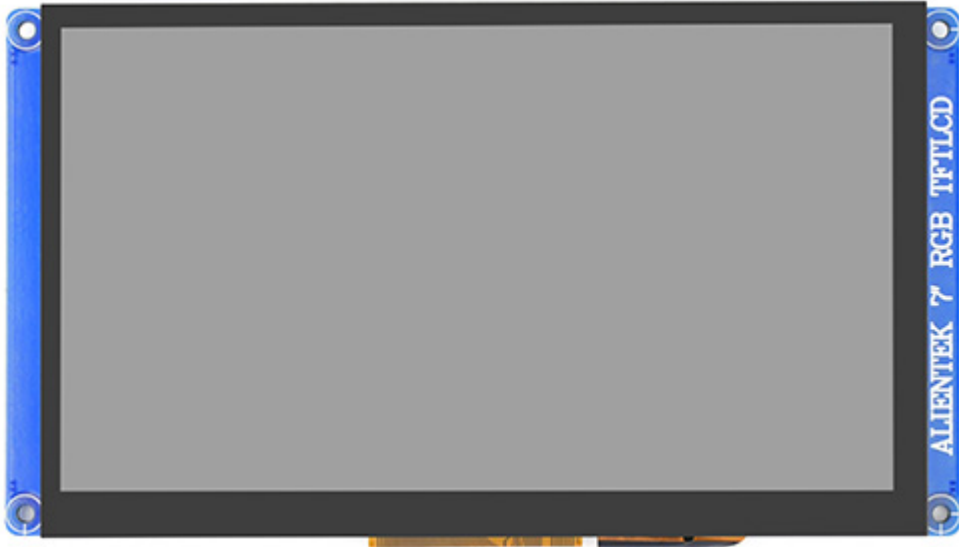


图 22.1.1 LCD 屏正向放置

这是硬件上固定的, 它是一种不可修改的硬件属性, 譬如你不能对 LCD 硬件进行配置, 将屏幕左下角设置为起点(0, 0), 这是不可以的; 像素点的排列顺序是从左到右、从上到下, 我们对 LCD 上不同像素点进行操作时, 需要找到该像素点对应的显存地址, 同样也是基于这种标准来的; 假设显存基地址为(unsigned char *)base, 那么定位一个(x, y)坐标像素点对应的地址的公式为 $base + (y * width + x) * pix_bytes$, 其中 pix_bytes 表示一个像素点使用 pix_bytes 个字节来描述。

示意图如下所示:

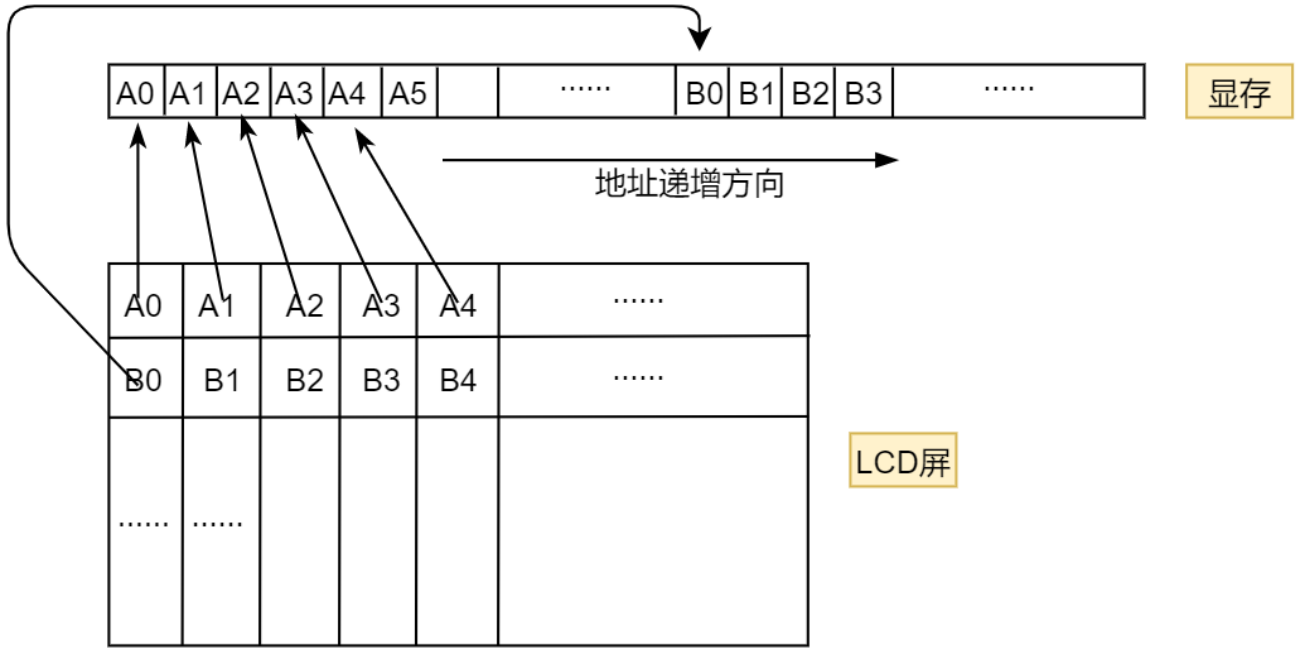


图 22.1.2 LCD 像素点与显存对应关系示意图

上图已经很直观、明了的说明了 LCD 屏上各个像素点与显存空间的对应关系。

但是在很多的应用场合中，往往需要以竖屏的方式来显示画面，譬如手机就是一个很好的例子，相信大家手机都是竖屏方式显示的；甚至还有一些电子产品既能支持横屏也能支持竖屏显示，当然这是针对应用程序而言。

那我们的应用程序中如何将 LCD 屏修改为竖屏显示呢？其实原理上非常简单，我相信大家都可以想到，图 22.1.1 所示屏幕，如果我们要将其作为竖屏显示，譬如在应用程序中将左下角作为起点(0, 0)，那么左上角对应就是(480-1, 0)、右下角对应就是(0, 800-1)、右上角对应就是(480-1, 800-1)，如下图所示：

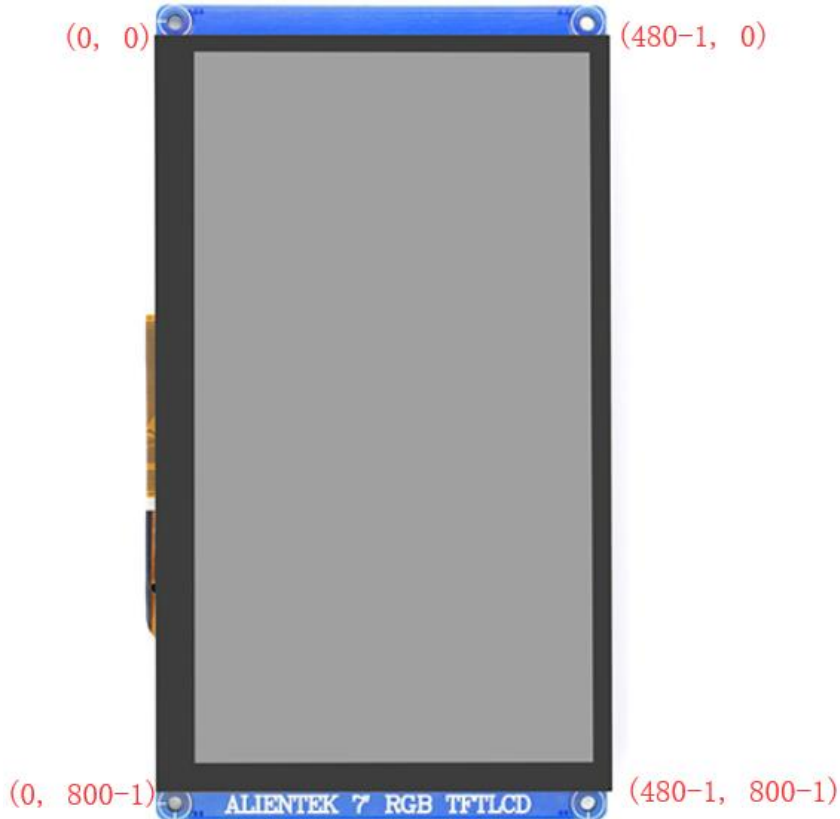


图 22.1.3 竖屏方式坐标分布

以上便是竖屏显示情况下, 其中的一种坐标分布情况, 当然这是应用程序认为的一种坐标分布, 对于 LCD 硬件来说, 实际物理上的起点坐标依然是图 22.1.1 中左上角的位置。

那么在上图中竖屏这种情况下, 应用程序的坐标对应的像素点, 它的显存地址就不能使用 $\text{base} + (\text{y} * \text{width} + \text{x}) * \text{pix_bytes}$ 公式进行计算了; 譬如上图竖屏方式下, 起点坐标(0, 0)对应的实际物理坐标是(0, 480-1), 同理它的显存地址也是通过实际物理坐标(0, 480-1)这个坐标计算而来、而不是通过(0, 0)计算。

在上图中竖屏方式下, 应用程序的(x, y)坐标点对应的显存地址可通过如下公式进行计算:

```
base + ((height - 1 - x) * width + y) * pix_bytes;
```

公式中的 x 和 y 分别表示竖屏方式下的(x, y)坐标, 当然这个公式仅适用于上图这种竖屏方式; 你也可以把图 22.1.3 旋转 180 度倒过来, 同样也是竖屏, 这种情况就不能用上面这条公式了。公式推导非常简单, 没什么可解释的。

22.2 编写示例代码

通过上小节的介绍, 我们已经知道了如何将横屏切换为竖屏显示, 本小节我们将对示例代码 19.4.1 进行修改, 将其修改为竖屏显示, 示例代码笔者已经给出, 如下所示。

本例程源码对应的路径为: [开发板光盘 ->11](#)、[Linux C 应用编程例程源码 ->22_lcd_vertical_display->lcd_vertical_display.c](#)。

示例代码 22.2.1 LCD 横屏切换为竖屏显示

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : lcd_vertical_display.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

描述 : FrameBuffer 应用编程之横屏切换为竖屏显示

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/6/15 邓涛创建

*****/

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/fb.h>

#define argb8888_to_rgb565(color)  ({ \
    unsigned int temp = (color); \
    ((temp & 0xF8000UL) >> 8) | \
    ((temp & 0xFC00UL) >> 5) | \
    ((temp & 0xF8UL) >> 3); \
})

static int lcd_width;           //LCD X 分辨率
static int lcd_height;         //LCD Y 分辨率
static int lcd_max_y;         //LCD Y 坐标最大值
static int user_width;         //竖屏模式下 X 分辨率
static int user_height;        //竖屏模式下 Y 分辨率
static unsigned short *screen_base = NULL; //映射后的显存基地址

/*****
 * 函数名称: lcd_draw_point
 * 功能描述: 打点
 * 输入参数: x, y, color
 * 返回值: 无
 *****/
static void lcd_draw_point(unsigned int x, unsigned int y, unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值

    /* 对传入参数的校验 */
    if (x >= user_width)
        x = user_width - 1;

```



```

    if (y >= user_height)
        y = user_height - 1;

    /* 填充颜色 */
    screen_base[(lcd_max_y-x) * lcd_width + y] = rgb565_color;
}

/*****
* 函数名称: lcd_draw_line
* 功能描述: 画线 (水平或垂直线)
* 输入参数: x, y, dir, length, color
* 返回值: 无
*****/
static void lcd_draw_line(unsigned int x, unsigned int y, int dir,
                          unsigned int length, unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值
    unsigned int end;
    unsigned long temp;

    /* 对传入参数的校验 */
    if (x >= user_width)
        x = user_width - 1;
    if (y >= user_height)
        y = user_height - 1;

    /* 填充颜色 */
    temp = (lcd_max_y-x) * lcd_width + y;
    if (dir) { //水平线
        end = x + length - 1;
        if (end >= user_width)
            end = user_width - 1;

        for (; x <= end; x++, temp -= lcd_width)
            screen_base[temp] = rgb565_color;
    }
    else { //垂直线
        end = y + length - 1;
        if (end >= user_height)
            end = user_height - 1;

        for (; y <= end; y++, temp++)
            screen_base[temp] = rgb565_color;
    }
}

```

```
    }
}

/*****
* 函数名称: lcd_draw_rectangle
* 功能描述: 画矩形
* 输入参数: start_x, end_x, start_y, end_y, color
* 返回值: 无
*****/
static void lcd_draw_rectangle(unsigned int start_x, unsigned int end_x,
                               unsigned int start_y, unsigned int end_y,
                               unsigned int color)
{
    int x_len = end_x - start_x + 1;
    int y_len = end_y - start_y - 1;

    lcd_draw_line(start_x, start_y, 1, x_len, color); //上边
    lcd_draw_line(start_x, end_y, 1, x_len, color); //下边
    lcd_draw_line(start_x, start_y + 1, 0, y_len, color); //左边
    lcd_draw_line(end_x, start_y + 1, 0, y_len, color); //右边
}

/*****
* 函数名称: lcd_fill
* 功能描述: 将一个矩形区域填充为参数 color 所指定的颜色
* 输入参数: start_x, end_x, start_y, end_y, color
* 返回值: 无
*****/
static void lcd_fill(unsigned int start_x, unsigned int end_x,
                    unsigned int start_y, unsigned int end_y,
                    unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值
    unsigned long temp;
    unsigned long step_size_count;
    int x;

    /* 对传入参数的校验 */
    if (end_x >= user_width)
        end_x = user_width - 1;
    if (end_y >= user_height)
        end_y = user_height - 1;
}
```

```
/* 填充颜色 */
temp = (lcd_max_y-start_x) * lcd_width + start_y;
for ( ; start_y <= end_y; start_y++, temp++) {

    step_size_count = 0;
    for (x = start_x; x <= end_x; x++, step_size_count += lcd_width)
        screen_base[temp - step_size_count] = rgb565_color;
}
}

int main(int argc, char *argv[])
{
    struct fb_fix_screeninfo fb_fix;
    struct fb_var_screeninfo fb_var;
    unsigned int screen_size;
    int fd;

    /* 打开 framebuffer 设备 */
    if (0 > (fd = open("/dev/fb0", O_RDWR))) {
        perror("open error");
        exit(EXIT_FAILURE);
    }

    /* 获取参数信息 */
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

    screen_size = fb_fix.line_length * fb_var.yres;
    lcd_width = fb_var.xres;
    lcd_height = fb_var.yres;
    lcd_max_y = lcd_height - 1;
    user_width = fb_var.yres;
    user_height = fb_var.xres;

    /* 将显示缓冲区映射到进程地址空间 */
    screen_base = mmap(NULL, screen_size, PROT_WRITE, MAP_SHARED, fd, 0);
    if (MAP_FAILED == (void *)screen_base) {
        perror("mmap error");
        close(fd);
        exit(EXIT_FAILURE);
    }

    /* 画正方形方块 */
}
```

```

int w = user_height * 0.25; // 方块的宽度为 1/4 屏幕高度
lcd_fill(0, user_width-1, 0, user_height-1, 0x0); // 清屏 (屏幕显示黑色)
lcd_fill(0, w, 0, w, 0xFF0000); // 红色方块
lcd_fill(user_width-w, user_width-1, 0, w, 0xFF00); // 绿色方块
lcd_fill(0, w, user_height-w, user_height-1, 0xFF); // 蓝色方块
lcd_fill(user_width-w, user_width-1, user_height-w, user_height-1, 0xFFFF00); // 黄色方块

/* 画线: 十字交叉线 */
lcd_draw_line(0, user_height * 0.5, 1, user_width, 0xFFFFFFFF); // 白色水平线
lcd_draw_line(user_width * 0.5, 0, 0, user_height, 0xFFFFFFFF); // 白色垂直线

/* 画矩形 */
unsigned int s_x, s_y, e_x, e_y;
s_x = 0.25 * user_width;
s_y = w;
e_x = user_width - s_x;
e_y = user_height - s_y;

for ( ; (s_x <= e_x) && (s_y <= e_y);
      s_x+=5, s_y+=5, e_x-=5, e_y-=5)
    lcd_draw_rectangle(s_x, e_x, s_y, e_y, 0xFFFFFFFF);

/* 退出 */
munmap(screen_base, screen_size); // 取消映射
close(fd); // 关闭文件
exit(EXIT_SUCCESS); // 退出进程
}

```

示例代码中自定义的 4 个函数: `lcd_draw_point()`、`lcd_draw_line()`、`lcd_draw_rectangle()`、`lcd_fill()` 都是基于竖屏显示方式进行定义的, 我们传入的坐标都是竖屏坐标, 函数内部会将其转为实际的物理坐标, 然后操作写对应的显存地址。

代码就不在讲解了, 没什么好说, 主要是一些转换上的逻辑代码, 理解了上小节所介绍的内容, 这个代码是不难理解的。

接着我们编译示例代码, 将编译得到的可执行文件拷贝到开发板 Linux 系统家目录下, 执行测试程序, 此时 LCD 显示效果如下所示:

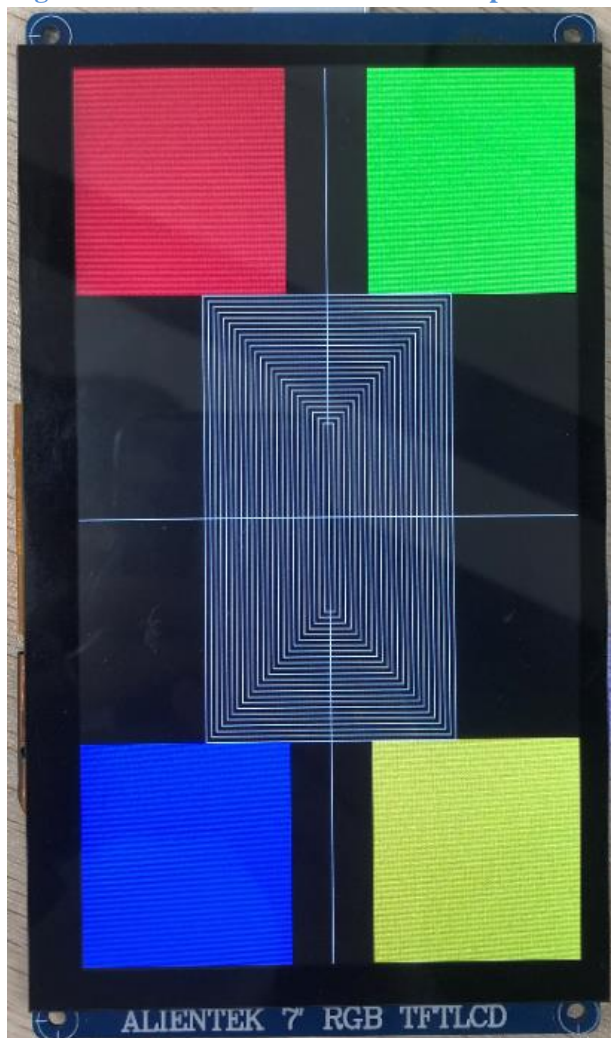


图 22.2.1 LCD 显示效果

大家可以与图 19.4.3 进行对比,画面的显示的效果,一个是横向显示、另一个是竖向显示。

第二十三章 在 LCD 上显示字符

前面几个章节向大家介绍了如何在 LCD 屏上显示图像，本章我们就来学习下，如何在 LCD 屏上显示字符，譬如数字、字母以及中文字符等，相信很多读者都已经迫不及待的想要了解了。OK，废话不多说直接开干！

本章将会讨论如下主题。

- 使用原始的方式：自己取模显示字符
- 使用 freetype 访问字体文件；
- freetype 简介；
- freetype 移植；
- freetype 的使用介绍。

23.1 原始方式: 取模显示字符

LCD 显示屏是由 $width * height$ 个像素点构成的, 显示字符, 一个非常容易想到的方法便是对字符取模, 然后在 LCD 屏上打点显示字符; 如果大家以前学习过单片机, 想必接触过一些显示屏, 譬如 oled、或者其它一些点阵式的显示屏, 其实这些显示屏显示字符的原理都是一样的, 如下所示:

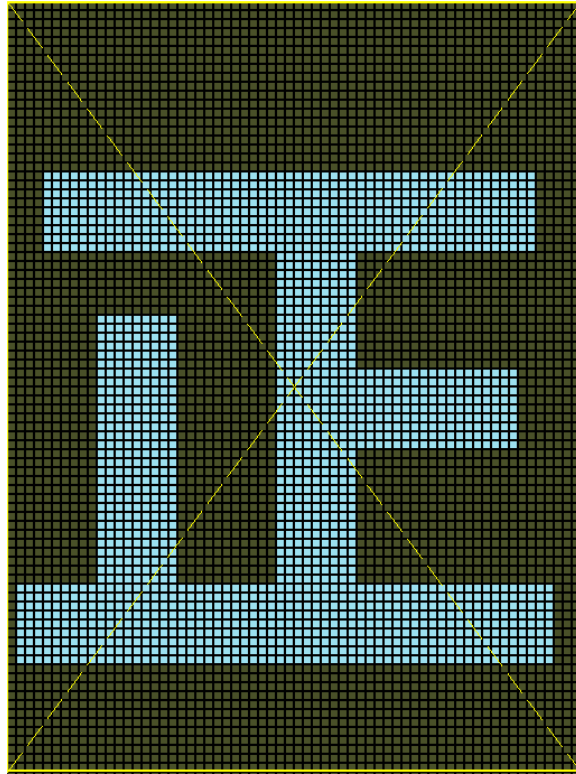


图 23.1.1 字符点阵图

我们可以通过一些字符取模软件获取到字符的子模; 所谓子模, 其实就是一个二维数组, 用于表示字符点阵中, 哪些小方块应该要填充颜色、哪些小方块不填充颜色。譬如上图“正”字符点阵, 这是一个宽度为 64 (64 个小方块)、高度为 86 (86 个小方块) 的字符点阵, 我们会使用一个二维数组来表示这个字符点阵:

```
unsigned char arr[86][64];
```

也就是一个 86 行 64 列的 `unsigned char` 类型数组, 数组存储的其实就是字符的位图数据, 字符点阵中的每一个小方块对应一个 bit 位, 因为一行一共有 64 个小方块, 也就对应 8 个字节 ($8 * 8 = 64$); 将填充颜色的方块使用 1 表示、不填充颜色的方块使用 0 来表示, 所以一个小方块刚好可以使用一个 bit 位来描述。

以上给大家简单地介绍了字符点阵的问题, 相信绝大部分读者都知道这些基础的东西, 其实本不太想讲这些内容, 但是考虑到可能有些读者确实就真的没接触这些, 所以还是简单地提一下。

我们编写一个简单的程序去测试下, 网上有很多的这种字符取模的小软件, 大家可以找一下, 我们用这个取模软件, 获取几个字符的子模, 然后在我们的 LCD 屏上去显示这些字符。

编写应用程序

本例程源码对应的路径为: 开发板光盘->11、Linux C 应用编程例程源码->23_freetype->show_char.c。

示例代码 23.1.1 以取模打点方式在 LCD 上显示字符

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : show_character.c
```

```
作者 : 邓涛
```



```

{0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0x7F,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x20,0x00},
{0x00,0x01,0x00,0x00,0xC0,0x00,0x7C,0x00},
{0x00,0x0F,0x00,0x03,0xF8,0x00,0x7F,0x00},
{0x80,0x3F,0xF0,0x03,0xFF,0x81,0xFF,0x00},
{0x80,0xFF,0xFC,0x03,0xFE,0x81,0xFF,0x00},
{0xC0,0x7F,0xFC,0x07,0xFE,0x03,0xFF,0x01},
{0xC0,0x7F,0xF8,0x07,0xFC,0x03,0xFF,0x01},
{0xE0,0x3F,0xF8,0x0F,0xFC,0x07,0xFE,0x01},
{0xE0,0x3F,0xF0,0x0F,0xFC,0x07,0xFE,0x03},
{0xF0,0x1F,0xF0,0x0F,0xF8,0x07,0xFE,0x03},
{0xF0,0x1F,0xF0,0x1F,0xF8,0x0F,0xFC,0x07},
{0xF8,0x0F,0xE0,0x1F,0xF8,0x0F,0xFC,0x07},
{0xF8,0x0F,0xE0,0x1F,0xF0,0x0F,0xF8,0x0F},
{0xFC,0x07,0xE0,0x3F,0xF0,0x1F,0xF8,0x0F},
{0xFC,0x07,0xE0,0x3F,0xF0,0x1F,0xF0,0x1F},
{0xFC,0x03,0xC0,0x0F,0xE0,0x03,0xF0,0x07},
{0xF0,0x03,0xC0,0x00,0x60,0x00,0xE0,0x01},
{0xE0,0x01,0x00,0x00,0x00,0x00,0x20,0x00},
{0x80,0x01,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*"点",1*/
};

```

```

static unsigned char ch_char3[86][8] = {
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},

```

```
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x1F},
{0x80,0x7F,0x00,0x80,0xFF,0x00,0x00,0x00},
{0x80,0x7F,0x00,0xC0,0x7F,0x00,0x00,0x00},
{0x80,0x7F,0x00,0xC0,0x7F,0x00,0x00,0x00},
{0x80,0x7F,0x00,0xC0,0x3F,0x00,0x00,0x00},
{0x80,0x7F,0x00,0xE0,0x3F,0x00,0x00,0x00},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0x07,0x00,0x00,0xFE,0x01},
{0x80,0x7F,0xF8,0x07,0x00,0x00,0xFE,0x01},
{0x80,0x7F,0xF8,0x07,0x00,0x00,0xFE,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0x80,0x7F,0xF8,0x07,0x00,0x00,0xFE,0x01},
{0x80,0x7F,0xF8,0x07,0x00,0x00,0xFE,0x01},
{0x80,0x7F,0xF8,0x07,0x00,0x00,0xFE,0x01},
{0xC0,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0xC0,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0xC0,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0xC0,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
```

```

{0xC0,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0xC0,0x7F,0xF8,0xFF,0xFF,0xFF,0xFF,0x01},
{0xC0,0x3F,0x00,0x00,0xFC,0x07,0x06,0x00},
{0xC0,0x3F,0x80,0x00,0xFC,0x07,0x07,0x00},
{0xC0,0x3F,0x80,0x03,0xFC,0x87,0x0F,0x00},
{0xE0,0x3F,0xC0,0x07,0xFC,0xE7,0x1F,0x00},
{0xE0,0x3F,0xE0,0x1F,0xFC,0xF7,0x3F,0x00},
{0xE0,0x3F,0xF0,0x3F,0xFC,0xE7,0x7F,0x00},
{0xE0,0x1F,0xF0,0x3F,0xFC,0xE7,0xFF,0x00},
{0xF0,0x1F,0xF8,0x1F,0xFC,0xC7,0xFF,0x00},
{0xF0,0x1F,0xFC,0x0F,0xFC,0x87,0xFF,0x01},
{0xF0,0x1F,0xFE,0x0F,0xFC,0x07,0xFF,0x03},
{0xF0,0x1F,0xFF,0x07,0xFC,0x07,0xFF,0x07},
{0xF8,0x8F,0xFF,0x03,0xFC,0x07,0xFE,0x0F},
{0xF8,0x8F,0xFF,0x01,0xFE,0x07,0xFC,0x0F},
{0xFC,0xCF,0xFF,0xFF,0xFF,0x07,0xF8,0x1F},
{0xFC,0xE7,0xFF,0xFE,0xFF,0x07,0xF8,0x0F},
{0xFE,0xC7,0x7F,0xFE,0xFF,0x03,0xF0,0x07},
{0xFE,0x83,0x3F,0xFC,0xFF,0x03,0xE0,0x03},
{0xF8,0x03,0x1F,0xFC,0xFF,0x03,0xE0,0x00},
{0xF0,0x03,0x0E,0xFC,0xFF,0x01,0x40,0x00},
{0xE0,0x01,0x04,0xFC,0xFF,0x00,0x00,0x00},
{0xC0,0x00,0x00,0xFC,0x3F,0x00,0x00,0x00},
{0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*"原",2*/
};

```

```

static unsigned char ch_char4[86][8] = {
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},

```



```

{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x1F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xF8,0x0F,0x00,0x00,0x00},
{0x00,0x00,0x00,0xFC,0x0F,0x00,0x00,0x00},
{0x00,0x10,0x00,0xFE,0x0F,0x00,0x00,0x00},
{0x00,0xF0,0xFF,0xFF,0x0F,0x00,0x00,0x00},
{0x00,0xF0,0xFF,0xFF,0x0F,0x00,0x00,0x00},
{0x00,0xE0,0xFF,0xFF,0x0F,0x00,0x00,0x00},
{0x00,0xE0,0xFF,0xFF,0x07,0x00,0x00,0x00},
{0x00,0xE0,0xFF,0xFF,0x07,0x00,0x00,0x00},
{0x00,0xE0,0xFF,0xFF,0x03,0x00,0x00,0x00},
{0x00,0xC0,0xFF,0xFF,0x01,0x00,0x00,0x00},
{0x00,0xC0,0xFF,0xFF,0x00,0x00,0x00,0x00},
{0x00,0xC0,0xFF,0x3F,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00},/*"子",3*/
};

#define argb8888_to_rgb565(color)  ({ \
    unsigned int temp = (color); \
    ((temp & 0xF8000UL) >> 8) | \
    ((temp & 0xFC00UL) >> 5) | \
    ((temp & 0xF8UL) >> 3); \
}

```



```

    })

/*****
* 函数名称: lcd_draw_character
* 功能描述: 在 LCD 屏指定位置处(x, y)画字符, 参数 color 指定字符的颜色
            指针 ch 指向字符对应的子模数组、参数 w、h 分别表示字符的宽度和高度
* 输入参数: color
* 返回值: 无
*****/
static void lcd_draw_character(unsigned int x, unsigned int y,
                               const unsigned char *ch, unsigned int w,
                               unsigned int h, unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值
    unsigned long temp;
    unsigned int end_x, end_y;
    int j;
    int columns;

    /**
    计算出二维数组有多少列
    参数 w 表示的是字符的宽度, 1 个宽度表示的是 1 个 bit 位
    并不是一个字节, 这里要注意, 如果宽度不是 byte 单位的整数倍
    通常会补零
    **/
    columns = w / 8;    //1byte=8bit
    if (0 != w % 8) columns++;

    /* 对参数进行限定 */
    if (w < 1 || h < 1) return;
    if (x >= width || y >= height) return;

    /* 计算出结束坐标位置 */
    end_x = x + w - 1;
    end_y = y + h - 1;

    /* 对结束坐标位置进行限定 */
    if (end_x >= width)
        end_x = width - 1;
    if (end_y >= height)
        end_y = height - 1;

    /* 计算有效宽度 */

```

```
h = end_y - y + 1;
w = end_x - x + 1;

/* 打点 */
temp = y * width + x; //定位到起点
for (y = 0; y < h; y++, temp += width) {

    for (x = 0, j = 0; x < w; ) {

        if (*(ch + y * columns + j) & (0x1 << (x % 8)))
            screen_base[temp + x] = rgb565_color;
        x++;
        if (0 == x % 8) j++;
    }
}

int main(void)
{
    struct fb_var_screeninfo fb_var = {0};
    struct fb_fix_screeninfo fb_fix = {0};
    unsigned long screen_size;
    int fd;

    /* 打开 framebuffer 设备 */
    fd = open(FB_DEV, O_RDWR);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", FB_DEV, strerror(errno));
        exit(EXIT_FAILURE);
    }

    /* 获取 framebuffer 设备信息 */
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

    screen_size = fb_fix.line_length * fb_var.yres;
    width = fb_var.xres;
    height = fb_var.yres;

    /* 内存映射 */
    screen_base = mmap(NULL, screen_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (MAP_FAILED == (void *)screen_base) {
        perror("mmap error");
    }
}
```

```
    close(fd);
    exit(EXIT_FAILURE);
}

/* LCD 背景刷白 */
memset(screen_base, 0xFF, screen_size);

/* 显示字符 */
int x = width * 0.5 - 128;
int y = height * 0.5 - 43;
lcd_draw_character(x, y, (unsigned char *)ch_char1, 64, 86, 0xFF00FF);
lcd_draw_character(x + 64, y, (unsigned char *)ch_char2, 64, 86, 0xFF00FF);
lcd_draw_character(x + 128, y, (unsigned char *)ch_char3, 64, 86, 0xFF00FF);
lcd_draw_character(x + 192, y, (unsigned char *)ch_char4, 64, 86, 0xFF00FF);

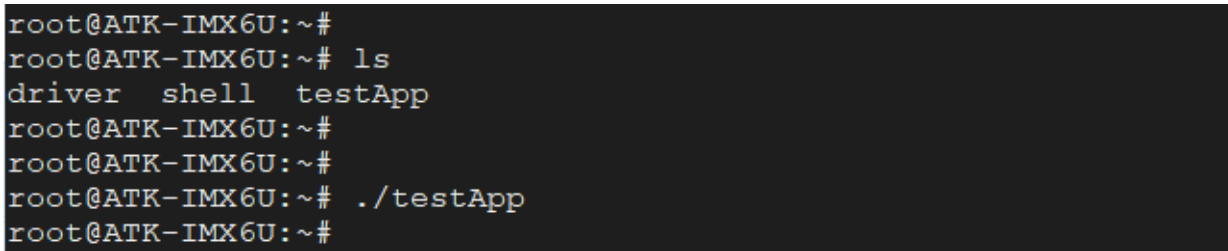
/* 退出程序 */
munmap(screen_base, screen_size);
close(fd);
return 0;
}
```

程序中定义了 4 个二维数组 `ch_char1`、`ch_char2`、`ch_char3`、`ch_char4`，这 4 个二维数组便是 4 个中文字符“正点原子”的子模，字符宽度为 64、高度为 86。这种字符取模软件就不给大家介绍了，网上很多这种软件、用法方法通常也非常简单，我们在取模的时候，选择从左到右、从上到下这种方式进行取模，上述示例代码便是按照这种方式进行解析的。

调用自定义函数 `lcd_draw_character` 在 LCD 上显示字符、并且是居中显示。

`lcd_draw_character` 函数中会子模数组进行解析，代码也没什么难度，就不讲解了。

编译上述示例代码，将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下，执行该测试程序：



```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver  shell  testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp
root@ATK-IMX6U:~#
```

图 23.1.2 执行测试程序

此时开发板 LCD 屏上会显示“正点原子”四个中文字符，如下图所示：

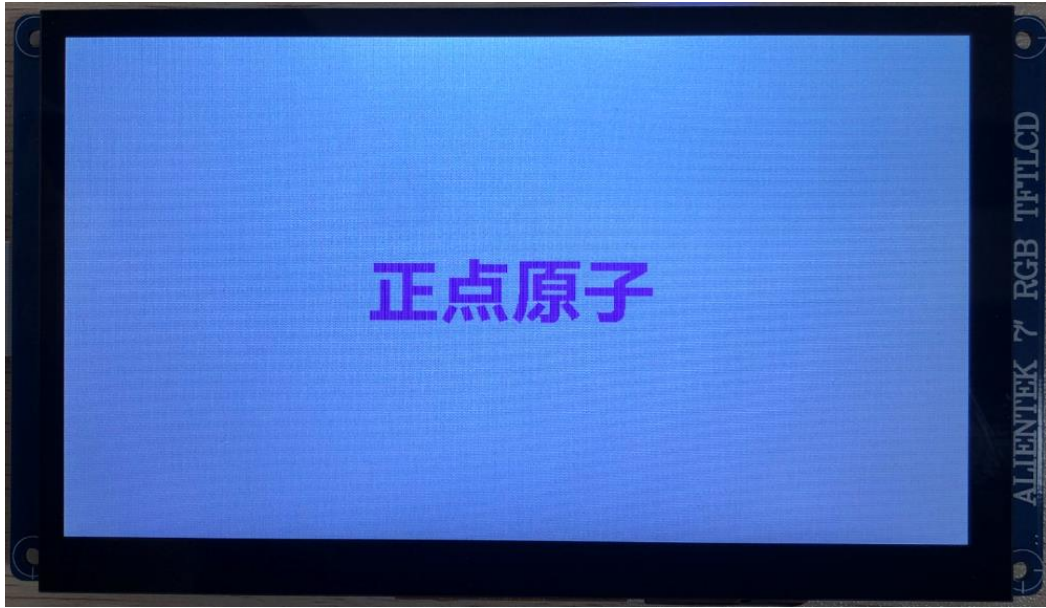


图 23.1.3 LCD 屏显示中文字符

使用这种方式还是非常简单地，自己取模、自己写函数打点显示，但是很 low！好歹咱们玩的也是操作系统，实际的应用项目一般肯定不会这么干，除非你的程序就是显示那么几个固定的字符。那既然如此，为什么还要给大家介绍这种方式呢？一来是让大家理解字符显示的原理；二来，一些实际的应用项目中可能确实就是这么做的，不同的项目它的需求不同，不能一概而论！

操作系统中通常都会有很多的字体文件，譬如 Windows 系统“C:\Windows\Fonts”目录下就有很多的字体文件，如下所示：

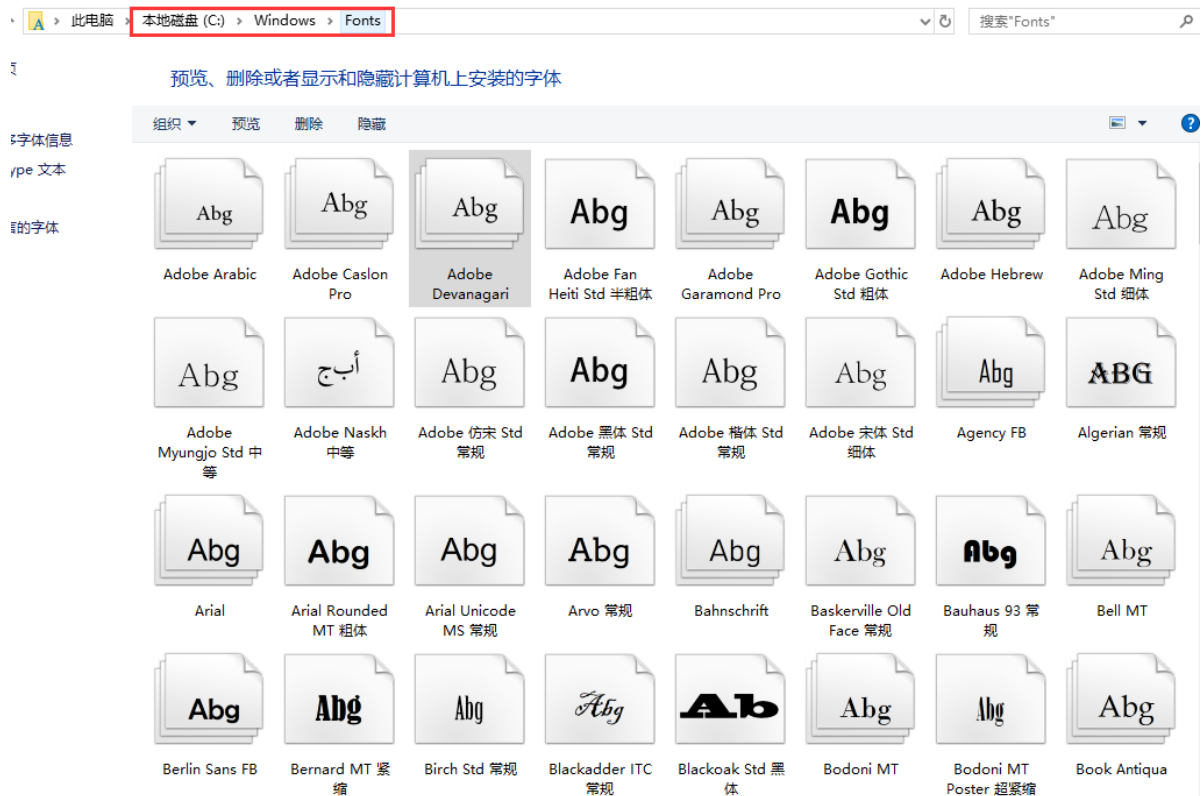


图 23.1.4 Windows 系统中的字体文件

字体文件的格式也有很多种, 譬如 otf、ttf、ttc 等, 这里就不给大家列举了, 有兴趣的读者自己百度一下; Linux 系统中, 字体文件通常会放在 /usr/share/fonts 目录下, 有了字体文件之后, 我们就不需要再对字符进行取模了, 它们已经编码进了字体文件中, 我们只需要解析字体文件、访问字体文件, 从字体文件中读取出字符的位图数据即可!

当然, 这些复杂的解析过程并不需要我们自己去实现, 有很多开源的字体引擎可以帮助我们来处理这些复杂的解析过程, 譬如 freetype 库, 下小节将向大家介绍如何使用 freetype 访问字体文件。

23.2 freetype 简介

FreeType 一个完全免费 (开源) 的软件字体引擎库, 设计小巧、高效、高度可定制且可移植, 它提供了统一的接口来访问多种不同格式的字体文件。它提供了一个简单、易于使用且统一的接口来访问字体文件的内容, 从而大大简化了这些任务。

请注意, “FreeType” 也称为 “FreeType 2”, 以区别于旧的、已弃用的 “FreeType 1” 库, FreeType 1 库已经不再维护和支持了。

23.3 freetype 移植

本小节我们来移植 FreeType, 将 FreeType 移植到我们的开发板根文件系统中; 事实上, 开发板出厂系统已经移植好了这些库, 但是版本稍微太低了, 作为学习者, 我们应该要学会自己动手移植, 这个是很重要的工作; 从上几个章节的学习内容可知, 我们很多的编程工作都是基于别人写好的库、调用库函数来实现应用程序的功能, 因为 Linux 软件生态做得非常好, 有很多免费、开源的库是可以使用的, 不用自己去捣鼓这些东西; 所以作为一个嵌入式 Linux 软件开发人员, 学会移植就显得很重要了。

23.3.1 下载 FreeType 源码

开发板出厂系统中, FreeType 的版本为 2.6, 这个版本稍微有点低, 我们选择移植 2.8 版本的 FreeType。进入到 <https://download.savannah.gnu.org/releases/freetype/> 链接地址, 如下所示:

Index of /releases/freetype/

File Name ↓	File Size ↓	Date ↓
Parent directory/	-	-
freetype-2.10.0.tar.bz2	3M	15-Mar-2019 07:28
freetype-2.10.0.tar.gz	3M	15-Mar-2019 07:27
freetype-2.10.0.tar.gz.sig	181	15-Mar-2019 07:30
freetype-2.10.1.tar.gz	3M	01-Jul-2019 21:12
freetype-2.10.1.tar.gz.sig	195	01-Jul-2019 21:13
freetype-2.10.1.tar.xz	2M	01-Jul-2019 21:12
freetype-2.10.1.tar.xz.sig	195	01-Jul-2019 21:13
freetype-2.10.2.tar.gz	3M	09-May-2020 05:20
freetype-2.10.2.tar.gz.sig	195	09-May-2020 05:21
freetype-2.10.2.tar.xz	2M	09-May-2020 05:20
freetype-2.10.2.tar.xz.sig	195	09-May-2020 05:21
freetype-2.10.3.tar.gz	3M	10-Oct-2020 16:36
freetype-2.10.3.tar.gz.sig	195	10-Oct-2020 16:38
freetype-2.10.3.tar.xz	2M	10-Oct-2020 16:36
freetype-2.10.3.tar.xz.sig	195	10-Oct-2020 16:38
freetype-2.10.4.tar.gz	3M	20-Oct-2020 05:23
freetype-2.10.4.tar.gz.sig	195	20-Oct-2020 05:24
freetype-2.10.4.tar.xz	2M	20-Oct-2020 05:23
freetype-2.10.4.tar.xz.sig	195	20-Oct-2020 05:24
freetype-2.4.0.tar.bz2	1M	12-Jul-2010 20:18
freetype-2.4.0.tar.bz2.sig	190	12-Jul-2010 20:19
freetype-2.4.0.tar.gz	2M	12-Jul-2010 20:17
freetype-2.4.0.tar.gz.sig	190	12-Jul-2010 20:19
freetype-2.4.1.tar.bz2	1M	18-Jul-2010 04:30

图 23.3.1 FreeType 源码下载地址

往下翻找到 2.8 版本, 选择 `freetype-2.8.tar.gz` 压缩文件:

freetype-2.8.1.tar.bz2.sig	181
freetype-2.8.1.tar.gz	2M
freetype-2.8.1.tar.gz.sig	181
freetype-2.8.tar.bz2	2M
freetype-2.8.tar.bz2.sig	181
freetype-2.8.tar.gz	2M
freetype-2.8.tar.gz.sig	181
freetype-2.9.1.tar.bz2	2M

图 23.3.2 找到 2.8 版本

点击 `freetype-2.8.tar.gz` 下载源码, 下载完成后我们将会得到 2.8 版本的 FreeType 源码包:

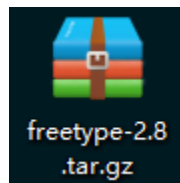


图 23.3.3 freetype-2.8.tar.gz

大家要自己动手下载, 开发板资料盘中是没有提供这些源码包的, 包括前面几个章节介绍的库, 譬如 `libpng`、`libjpeg` 以及 `tslib` 等, 都没有提供它们的源码压缩文件。

23.3.2 交叉编译 FreeType 源码

将下载好的 `freetype-2.8.tar.gz` 压缩文件拷贝到 Ubuntu 系统的用户家目录下, 如下所示:

```
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  freetype-2.8.tar.gz  tools  vscode_ws
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 23.3.4 将源码包拷贝到 Ubuntu 系统

在 tools 目录下创建一个名为 freetype 的目录，把它作为 FreeType 的安装目录：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ mkdir ~/tools/freetype
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls ~/tools/
freetype  jpeg  png  tslib  zlib
dt@dt-virtual-machine:~$
```

图 23.3.5 创建 FreeType 的安装目录

执行命令将 freetype-2.8.tar.gz 解压开来：

```
tar -xzf freetype-2.8.tar.gz
```

```
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  freetype-2.8.tar.gz  tools  vscode_ws  公共
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ tar -xzf freetype-2.8.tar.gz
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls
eclipse_ws  examples.desktop  freetype-2.8  freetype-2.8.tar.gz  tools
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$
```

图 23.3.6 将 freetype-2.8.tar.gz 压缩文件解压

解压成功之后便会得到 FreeType 的源码目录 freetype-2.8。

进入到 freetype-2.8 目录，老规矩，同样是三部曲：配置、编译、安装。

首先对交叉编译工具的环境进行初始化，前面章节内容已经提过很多次了，使用交叉编译器之前，必须对其环境进行初始化（如果当前终端已经初始化过了，则无需再次进行初始化）。

FreeType 库基于模块化设计，意味着我们可以对其进行裁剪，将不需要的功能模块从配置中移除，减小库文件的体积；除此之外，FreeType 还支持很多配置选项，如果大家想要对 FreeType 做一些自定义配置或者对其进行裁剪，可以参考 FreeType 源码目录下 docs/CUSTOMIZE 文档，该文件对此有比较详细的说明，建议大家看一看，如果有需求的话。docs 目录下还有其它很多的说明文档，也都可以读一读。

这里我们简单地配置一下，打开 include/freetype/config/ftoption.h 文件，如下所示：

```
vi include/freetype/config/ftoption.h
```

```

1  /******
2  /*
3  /*  ftoption.h
4  /*
5  /*    User-selectable configuration macros (specification only).
6  /*
7  /*  Copyright 1996-2017 by
8  /*  David Turner, Robert Wilhelm, and Werner Lemberg.
9  /*
10 /*  This file is part of the FreeType project, and may only be used,
11 /*  modified, and distributed under the terms of the FreeType project
12 /*  license, LICENSE.TXT.  By continuing to use, modify, or distribute
13 /*  this file you indicate that you have read the license and
14 /*  understand and accept it fully.
15 /*
16 /******
17
18
19 #ifndef FTOPTION_H_
20 #define FTOPTION_H_
21
22
23 #include <ft2build.h>
24
25
26 FT_BEGIN_HEADER
27
28 /******
29 /*
30 /*          USER-SELECTABLE CONFIGURATION MACROS
31 /*
32 /*  This file contains the default configuration macro definitions for
33 /*  a standard build of the FreeType library.  There are three ways to
34 /*  use this file to build project-specific versions of the library:
35 /*
36 /*  - You can modify this file by hand, but this is not recommended in
37 /*  cases where you would like to build several versions of the
38 /*  library from a single source directory.
39 /*
40 /*  - You can put a copy of this file in your build directory, more

```

图 23.3.7 ftoption.h 配置文件

该文件定义了很多的配置宏，我们可以选择使能或禁用这些配置选项，具体配置哪些功能，大家自己去研究，每一个配置宏都有详细地解释说明。这里我们打开以下两个配置宏：

```

#define FT_CONFIG_OPTION_SYSTEM_ZLIB
#define FT_CONFIG_OPTION_USE_PNG

```

大家找到这两个宏，默认情况下，这两个都被注释掉了，所以是没有使能的；把这两个宏的注释去掉，使能这两个配置宏。

第一个配置宏表示使用系统安装的 zlib 库，因为 FreeType 支持 Gzip 压缩文件，会使用到 zlib 库，zlib 之前我们移植好了；第二个配置宏表示支持 PNG bitmap 位图，因为 FreeType 可以加载 PNG 格式的彩色位图字形，需要依赖于 libpng 库，这个库前面我们也是移植好了。

配置好之后，保存、退出 ftoption.h 文件，接着执行如下命令对 FreeType 工程源码进行配置：

```

./configure --prefix=/home/dt/tools/freetype/ --host=arm-poky-linux-gnueabi --with-zlib=yes --with-bzip2=no -
-with-png=yes --with-harfbuzz=no ZLIB_CFLAGS="-I/home/dt/tools/zlib/include -L/home/dt/tools/zlib/lib"
ZLIB_LIBS=-lz LIBPNG_CFLAGS="-I/home/dt/tools/png/include -L/home/dt/tools/png/lib" LIBPNG_LIBS=-
lpng

```

这个配置命令很长，简单地提一下，具体的细节大家可以执行"./configure --help"查看配置帮助信息。

--prefix 选项指定 FreeType 库的安装目录；--host 选项设置为交叉编译器名称的前缀，这两个选项前面几个章节内容都已经给大家详细地解释过。

- with-zlib=yes 表示使用 zlib；
- with-bzip2=no 表示不使用 bzip2 库；
- with-png=yes 表示使用 libpng 库；
- with-harfbuzz=no 表示不使用 harfbuzz 库。

ZLIB_CFLAGS 选项用于指定 zlib 的头文件路径和库文件路径, 根据实际安装路径填写;

ZLIB_LIBS 选项指定链接的 zlib 库的名称;

LIBPNG_CFLAGS 选项用于指定 libpng 的头文件路径和库文件路径, 根据实际安装路径填写;

LIBPNG_LIBS 选项用于指定链接的 libpng 库的名称。

```
dt@dt-virtual-machine:~/freetype-2.8$
dt@dt-virtual-machine:~/freetype-2.8$ ./configure --prefix=/home/dt/tools/freetype/ --host=arm-poky-linux-gnueabi --with-zlib=yes --with-bzip2=no --with-png=yes --with-harfbuzz=no ZLIB_CFLAGS="-I/home/dt/tools/zlib/include -L/home/dt/tools/zlib/lib" ZLIB_LIBS="-lz" LIBPNG_CFLAGS="-I/home/dt/tools/png/include -L/home/dt/tools/png/lib" LIBPNG_LIBS="-lpng
FreeType build system -- automatic system detection

The following settings are used:

platform          unix
compiler          arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-
poky-linux-gnueabi
configuration directory  ./builds/unix
configuration rules     ./builds/unix/unix.mk

If this does not correspond to your system or settings please remove the file
'config.mk' from this directory then read the INSTALL file for help.

Otherwise, simply type `make` again to build the library,
or `make reldoc` to build the API reference (this needs python >= 2.6).

Generating modules list in ./objs/ftmodule.h...
* module: truetype (Windows/Mac font files with extension *.ttf or *.ttc)
* module: type1 (Postscript font files with extension *.pfa or *.pfb)
* module: cff (OpenType fonts with extension *.otf)
* module: cid (Postscript CID-keyed fonts, no known extension)
* module: pfr (PFR/TrueDoc font files with extension *.pfr)
* module: type42 (Type 42 font files with no known extension)
* module: winfnt (Windows bitmap fonts with extension *.fnt or *.fon)
* module: pcf (pcf bitmap fonts)
* module: bdf (bdf bitmap fonts)

checking for working mmap... (cached) yes
checking whether munmap is declared... yes
checking for munmap's first parameter type... void *
checking for memcpy... (cached) yes
checking for memmove... (cached) yes
checking gcc compiler flag -pedantic to assure ANSI C works correctly... ok, add it to XX_ANSIFLAGS
checking gcc compiler flag -ansi to assure ANSI C works correctly... ok, add it to XX_ANSIFLAGS
checking for ZLIB... yes
checking for LIBPNG... yes
configure: creating ./config.status
config.status: creating unix-cc.mk
config.status: creating unix-def.mk
config.status: creating ftconfig.h
config.status: executing libtool commands
configure:

Library configuration:
external zlib: yes (ZLIB_CFLAGS and ZLIB_LIBS)
bzip2: no
libpng: yes (LIBPNG_CFLAGS and LIBPNG_LIBS)
harfbuzz: no

make: Nothing to be done for 'unix'.
dt@dt-virtual-machine:~/freetype-2.8$
```

图 23.3.8 配置 FreeType 工程

配置完成之后接着执行 make 编译, 编译完成之后执行 make install 安装即可!

23.3.3 安装目录下的文件

进入到 FreeType 安装目录下, 如下所示:

```
dt@dt-virtual-machine:~$ pwd
/home/dt
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd ~/tools/freetype/
dt@dt-virtual-machine:~/tools/freetype$ pwd
/home/dt/tools/freetype
dt@dt-virtual-machine:~/tools/freetype$ ls
bin include lib share
dt@dt-virtual-machine:~/tools/freetype$
```

图 23.3.9 FreeType 安装目录下的文件

同样有 bin 目录、include 目录以及 lib 目录, 大家可以自己进入到这些目录下, 浏览下这些目录下的有哪些文件, 对此有个印象。

如果要使用 FreeType 库,我们需要在应用程序源码中包含 `include/freetype2` 目录下的 `ft2build.h` 头文件,除此之外,还需要包含另一个头文件 `FT_FREETYPE_H`,这是一个用宏定义的头文件,其实就是 `include/freetype2/freetype/freetype.h` 头文件。

所以,在我们的应用程序一般是这样写:

```
#include <ft2build.h>
#include FT_FREETYPE_H
```

23.3.4 移植到开发板

接下来将编译得到的动态链接库文件拷贝到开发板 Linux 系统 `/usr/lib` 目录,在拷贝之前,需将 `/usr/lib` 目录下原有的 FreeType 库文件删除掉,执行下面这条命令:

```
rm -rf /usr/lib/libfreetype.*
```

删除之后,再将我们编译得到的库文件拷贝到开发板 `/usr/lib` 目录下,也就是 FreeType 安装目录 `lib` 目录下的所有库文件,拷贝的时候注意符号链接的问题。拷贝完成之后,如下所示:

```
root@ATK-IMX6U:~# ls -l /usr/lib/libfreetype.*
-rw-r--r-- 1 1000 tracing 3904846 Jul 13 2021 /usr/lib/libfreetype.a
-rwxr-xr-x 1 1000 tracing 987 Jul 13 2021 /usr/lib/libfreetype.la
lrwxrwxrwx 1 1000 tracing 21 Jul 13 2021 /usr/lib/libfreetype.so -> libfreetype.so.6.14.0
lrwxrwxrwx 1 1000 tracing 21 Jul 13 2021 /usr/lib/libfreetype.so.6 -> libfreetype.so.6.14.0
-rwxr-xr-x 1 1000 tracing 2500644 Jul 13 2021 /usr/lib/libfreetype.so.6.14.0
root@ATK-IMX6U:~#
```

图 23.3.10 开发板 `/usr/lib` 目录下的 FreeType 库文件

23.4 freetype 库的使用

整个移植工作完成之后,接着简单地介绍下 FreeType 库的使用,FreeType 库支持的功能很多、提供给用户的库函数也很多,所以笔者肯定不会给大家细聊!以介绍性为主。

FreeType 官方也提供了详细地使用帮助文档,以下便是这些文档的链接地址:

<https://www.freetype.org/freetype2/docs/tutorial/step1.html>

<https://www.freetype.org/freetype2/docs/tutorial/step2.html>

<https://www.freetype.org/freetype2/docs/reference/index.html>

以下这个链接是一份中文参考文档,大家可以看一下,笔者也不知道是哪位作者编写的,写的非常详细!

<https://www.doc88.com/p-7178359224563.html?r=1>

在正式介绍 FreeType 库使用之前,需要先了解几个涉及到的概念:

字形 (glyph)

字符图像就叫做字形,一个字符能够有多种不同的字形,可以理解为字形就是字符的一种书写风格,譬如宋体的汉字“国”与微软雅黑的汉字“国”,它们的字形是不同的,也就是它们书写风格是不同;宋体的“国”与微软雅黑的“国”就是两种不同的字形。

字形索引

在字体文件中,通过字形索引找到对应的字形,而字形索引是由字符编码转换而来的,譬如 ASCII 编码、GB2312 编码、BIG5 编码、GBK 编码以及国际标准字符集使用的 Unicode 编码等。对于字符编码,如果还有不了解读者,建议自行查阅相关的书籍。

像素点 (pixel)、点 (point) 以及 dpi

像素点大家都知道,譬如 LCD 分辨率为 800×480 ,那就表示 LCD 水平方向有 800 个像素点、垂直方向有 480 个像素点,所以此 LCD 一共有 800×480 个像素点。

像素点这个概念大家都很熟悉了,也就不再多说;我们再来看下“点”的概念,点 (point) 是一种简单地物理单位,在数字印刷中,一个点 (point) 等于 $1/72$ 英寸 (1 英寸等于 25.4 毫米)。

除此之外,还有一个 dpi 的概念, dpi (dots per inch) 表示每英寸的像素点数,譬如 300×400 dpi 表示在水平方向,每英寸有 300 个像素点、在垂直方向上每英寸有 400 个像素点。通过点数和 dpi 可以计算出像素点数,公式如下:

$$\text{像素点数} = \text{点数} * \text{dpi} / 72$$

譬如,假设某一显示设备水平方向 dpi 为 300,已知水平方向的点数为 50,那么像素点数的计算方式为:

$$50 * 300 / 72 = 208$$

所以可以算出像素点数为 208,因为后面会用到这些概念,所以先给大家简单地说明一下。

字形的布局

以下两张图清晰地描述了字形布局的情况:分为水平布局和垂直布局,分别使用图 23.4.1 和图 23.4.2 来描述布局情况,以下这两张图都是从官方的文档中截取过来的。

水平方向书写文字使用水平布局方式,绝大部分情况下我们一般都是在水平方向上书写文字;垂直方向书写文字使用垂直布局方式,对于汉字来说,垂直方向书写也是比较常见的,很有代表性的就是对联、还有很多古书文字的写法,也都是采用这种垂直书写。

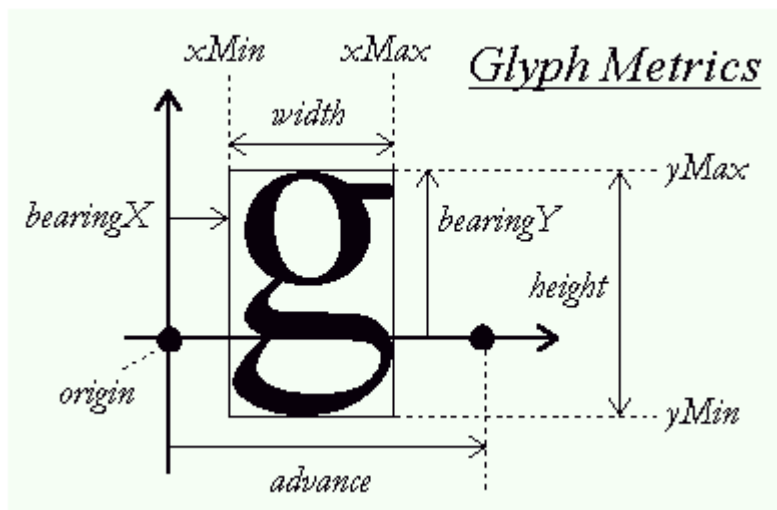


图 23.4.1 水平布局

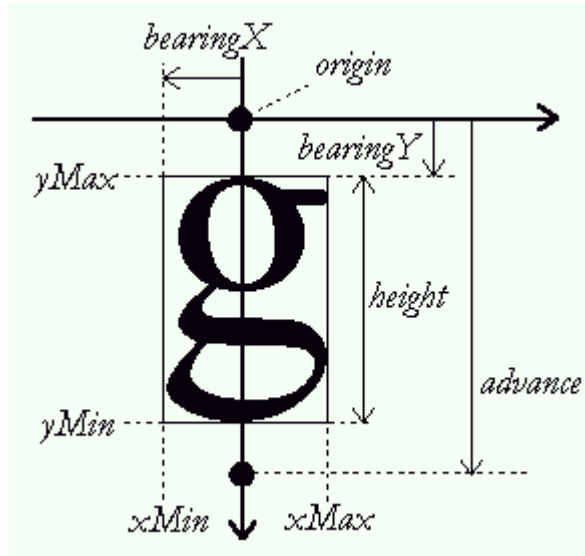


图 23.4.2 垂直布局

- 基准线、原点

从图中可以看到,不管是水平布局还是垂直布局,图中都可以找到一个 `origin` 原点,经过原点的水平线(X轴)和垂直线(Y轴)称为基准线,笔者将其称为水平基准线和垂直基准线。

对于水平布局,垂直基准线在字形的左边,垂直基准线简单地放置在字形上,通过图中所标注的度量数据确定与基准线的位置关系。

对于垂直布局,水平基准线在字形的上方,字形在垂直基准线上居中放置,同样也是通过图中所标注的度量数据确定与基准线的位置关系。

原点、基准线可以用于定位字形,水平布局和垂直布局使用不同的约束来放置字形。

- 字形的宽度和高度

每一个字形都有自己的宽度和高度,图中使用 `width` (宽)和 `height` (高)来表示, `width` 描述了字形轮廓的最左边到最右边的距离;而 `height` 描述了字形轮廓的最上边到最下边的距离。同一种书写风格,不同字符所对应的字形,它们的宽高是不一定相等的,譬如大写 A 和小写 a,宽度和高度明显是不同的;但有些字符的字形宽度和高度是相同的,这个与具体的字符有关!

- `bearingX` 和 `bearingY`

`bearingX` 表示从垂直基准线到字形轮廓最左边的距离。对于水平布局来说,字形在垂直基准线的右侧,所以 `bearingX` 是一个正数;而对于垂直布局来说,字形在垂直基准线上居中放置,所以字形轮廓的最左边通常是在垂直基准线的左侧,所以 `bearingX` 是一个负数。

`bearingY` 则表示从水平基准线到字形轮廓最上边的距离。对于垂直布局来说, `bearingY` 是一个正数,字形处于水平基准线的下方;而对于水平布局来说,如果字形轮廓的最上边在水平基准线的上方,则 `bearingY` 是一个正数、相反则是一个负数。

- `xMin/xMax`、`yMin/yMax`

`xMin` 表示字形轮廓最左边的位置, `xMax` 则表示字形轮廓最右边的位置; `yMin` 表示字形轮廓最下边的位置, `yMax` 则表示字形轮廓最上边的位置,通过这 4 个位置可以构成一个字形的边界框 (bounding box, `bbox`),当然这是一个假象的框子,它尽可能紧密的装入字形。

- `advance`

`advance` 则表示步进宽度,相邻两个原点位置的距离(字间距)。如果是水平布局,则表示相邻的两个原点在水平方向上的距离 (`advanceX`),也就是相邻两条垂直基准线之间的距离;同理,如果是垂直布局,则表示相邻的两个原点在垂直方向上的距离 (`advanceY`),也就是相邻两条水平基准线之间的距离。

以上所提到的这些参数都很重要,大家一定要理解这些参数所表示的意义,绘制字符时,需要以这些参数作为参考值进行对齐显示。

使用 FreeType 访问字体文件,可以从字体文件中获取到字形的位图数据,位图数据存储在一个 buffer 中,buffer 大小为字形的宽*高个字节(字形边界框的宽*高个字节),也就是图 23.4.1 中 width*height 个字节大小,每一个点使用一个字节来表示,当数组中该点对应的数值等于 0,表示该点不填充颜色;当数值大于 0,表示该点需要填充颜色。

字符显示时如何对齐?

平时我们使用文本编辑器编写文字的时候,这些字符都是对齐显示的;譬如在一行文本中,即使包含大小写的英文字母、标点符号、汉字等这些字符,这一行字符显示在屏幕上时、都是对齐显示的;这里说的对齐是按照标准规范进行对齐,譬如逗号", "显示时是靠近下边的、而不是靠近上边显示;双引号“”显示时是靠近上边的、而不是居中显示;这就是笔者认为的字符显示时的对齐规范,你可以认为每一个字符,它都有对应的一个显示规范,是靠近上边显示呢、还是靠近下边显示亦或者是靠近中间显示呢等。

那我们如何保证对齐显示呢?其实就是通过图 23.4.1 中的水平基线和垂直基线,如下图所示:

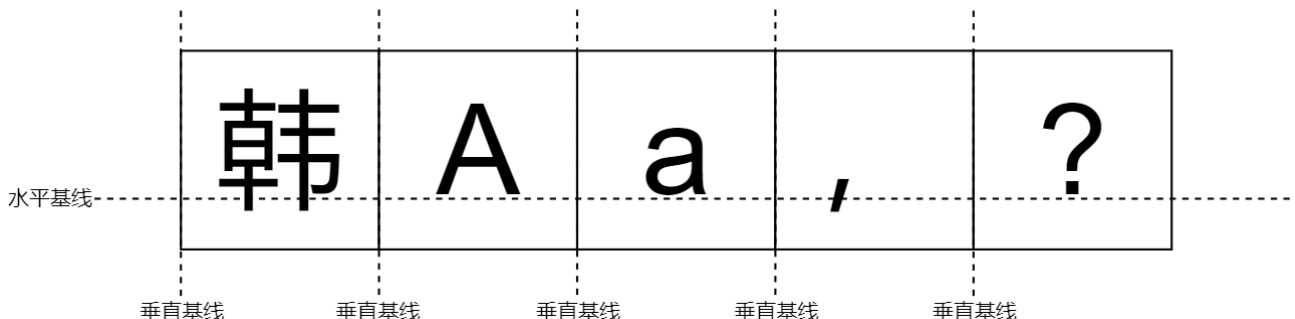


图 23.4.3 通过水平基线和垂直基线保证对齐显示

不同字符对应的字形,水平基线到字形轮廓最上边的距离都是不一样的,譬如图中水平基线到“A”和“a”轮廓最上边的距离明显是不一样的;除此之外,有些字形的轮廓最下边已经在水平基线之下、而有些字形的轮廓最下边却又在水平基线之上。

分析完水平基线之后,我们再来看看垂直基线,每一个字形的垂直基线到字形轮廓最左边的距离也都是不一样的,譬如“韩”和“a”,很明显、它们各自的垂直基线到字形轮廓最左边的距离是不一样的。

水平基线可以作为垂直方向(上下方向)上对齐显示的基准线,而垂直基线可以作为水平方向(左右方向)上对齐显示的基准线;对于水平布局来说,相邻两条垂直基线的距离就是字间距或者叫步进宽度;从一个字形的原点加上一个步进宽度就到了下一个字形的原点。

当我们要在屏幕上画字形的时候,首先要定位到字形的左上角位置,从左上角开始,依次从左到右、从上到下,字形显示的宽度就是字形的宽度 width、字符显示的高度就是字形的高度 height。那如何找到左上角的位置,这个很简单,通过 bearingY 和 bearingX 便可确定。譬如我们将(100, 100)这个位置作为原点,那么,字符显示位置的左上角便是(100+bearingX, 100-bearingY)。

以上就给大家介绍关于字符对齐的问题,大家一定要理解这些内容,如果你理解不了上面的内容,后面的示例代码你可能就看不懂。

23.4.1 初始化 FreeType 库

在使用 FreeType 库函数之前,需要对 FreeType 库进行初始化操作,使用 FT_Init_FreeType()函数完成初始化操作。在调用该函数之前,我们需要定义一个 FT_Library 类型变量,调用 FT_Init_FreeType()函数时将该变量的指针作为参数传递进去;使用示例如下所示:

```
FT_Library library;
FT_Error error;
```

```
error = FT_Init_FreeType(&library);
if (error)
    fprintf(stderr, "Error: failed to initialize FreeType library object\n");
```

FT_Init_FreeType 完成以下操作:

- 它创建了一个 FreeType 库对象, 并将 library 作为库对象的句柄。
- FT_Init_FreeType()调用成功返回 0; 失败将返回一个非零值错误码。

23.4.2 加载 face 对象

应用程序通过调用 FT_New_Face()函数创建一个新的 face 对象, 其实就是加载字体文件, 为啥叫 face (脸), 应该是一种抽象的说法! 一个 face 对象描述了一个特定的字体样式和风格, 譬如"Times New Roman Regular"和"Times New Roman Italic"对应两种不同的 face。

调用 FT_New_Face()函数前, 我们需要定义一个 FT_Face 类型变量, 使用示例如下所示:

```
FT_Library library;    //库对象的句柄
FT_Face face;         //face 对象的句柄
FT_Error error;

FT_Init_FreeType(&library);

error = FT_New_Face(library, "/usr/share/fonts/font.ttf", 0, &face);
if (error) {
    /* 发生错误、进行相关处理 */
}
```

FT_New_Face()函数原型如下所示:

```
FT_Error FT_New_Face(FT_Library library, const char *filepathname,
                    FT_Long face_index, FT_Face *aface);
```

函数参数以及返回值说明如下:

library: 一个 FreeType 库对象的句柄, face 对象从中建立;

filepathname: 字库文件路径名 (一个标准的 C 字符串);

face_index: 某些字体格式允许把几个字体 face 嵌入到同一个文件中, 这个索引指示了你想加载的 face, 其实就是一个下标, 如果这个值太大, 函数将会返回一个错误, 通常把它设置为 0 即可! 想要知道一个字体文件中包含了多少个 face, 只要简单地加载它的第一个 face (把 face_index 设置为 0), 函数调用成功返回后, face->num_faces 的值就指示出了有多少个 face 嵌入在该字体文件中。

aface: 一个指向新建 face 对象的指针, 当失败时其值被设置为 NULL。

返回值: 调用成功返回 0; 失败将返回一个非零值的错误码。

23.4.3 设置字体大小

设置字体的大小有两种方式: FT_Set_Char_Size()和 FT_Set_Pixel_Sizes()。

FT_Set_Pixel_Sizes()函数

调用 FT_Set_Pixel_Sizes()函数设置字体的宽度和高度, 以像素为单位, 使用示例如下所示:

```
FT_Set_Pixel_Sizes(face, 50, 50);
```

第一个参数传入 face 句柄；第二个参数和第三个参数分别指示字体的宽度和高度，以像素为单位；需要注意的是，我们可以将宽度或高度中的任意一个参数设置为 0，那么意味着设置为 0 的参数将会与另一个参数保持相等，如下所示：

```
FT_Set_Pixel_Sizes(face, 50, 0);
```

上面调用 FT_Set_Pixel_Sizes() 函数时，将字体高度设置为 0，也就意味着字体高度将自动等于字体宽度 50。

FT_Set_Char_Size() 函数

调用 FT_Set_Char_Size() 函数设置字体大小，示例如下所示，假设在一个 300x300dpi 的设备上把字体大小设置为 16pt：

```
error = FT_Set_Char_Size(
    face, //face 对象的句柄
    16*64, //以 1/64 点为单位的字体宽度
    16*64, //以 1/64 点为单位的字体高度
    300, //水平方向上每英寸的像素点数
    300); //垂直方向上每英寸的像素点数
```

说明：

- 字体的宽度和高度并不是以像素为单位，而是以 1/64 点（point）为单位表示（也就是 26.6 固定浮点格式），一个点是一个 1/72 英寸的距离。
- 同样也可将宽度或高度其中之一设置为 0，那么意味着设置为 0 的参数将会与另一个参数保持相等。
- dpi 参数设置为 0 时，表示使用默认值 72dpi。

23.4.4 加载字形图像

设置完成之后，接下来就可以加载字符图像了，总共分为 3 步：

a)、获取字符的字形索引

通过 FT_Get_Char_Inde() 函数将字符编码转换为字形索引（glyph index），FreeType 默认使用 UTF-16 编码类型，也就是 Unicode 编码方式，采用 2 个字节来表示一个编码值。

```
XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXX
```

23.5 示例代码

显示单个 ASCII 字符

显示单个中文字符

显示一行字符（一行字符对齐显示）

显示多行字符（一行字符的高度计算）

字体变形（旋转、斜体……）

示例代码 23.5.1 FreeType 使用示例代码

```
/******
```

Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.

文件名：freetype_test.c

作者：邓涛

版本 : V1.0

描述 : FreeType 测试示例代码

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/7/14 邓涛创建

```

*****/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <linux/fb.h>
#include <math.h>          //数学库函数头文件

#include <wchar.h>
#include <ft2build.h>
#include FT_FREETYPE_H

#define FB_DEV          "/dev/fb0"          //LCD 设备节点

#define argb8888_to_rgb565(color)  ({ \
    unsigned int temp = (color); \
    ((temp & 0xF80000UL) >> 8) | \
    ((temp & 0xFC00UL) >> 5) | \
    ((temp & 0xF8UL) >> 3); \
})

static unsigned int width;          //LCD 宽度
static unsigned int height;        //LCD 高度
static unsigned short *screen_base = NULL; //LCD 显存基地址 RGB565
static unsigned long screen_size;
static int fd = -1;

static FT_Library library;
static FT_Face face;

static int fb_dev_init(void)

```



```
{
    struct fb_var_screeninfo fb_var = {0};
    struct fb_fix_screeninfo fb_fix = {0};

    /* 打开 framebuffer 设备 */
    fd = open(FB_DEV, O_RDWR);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", FB_DEV, strerror(errno));
        return -1;
    }

    /* 获取 framebuffer 设备信息 */
    ioctl(fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fd, FBIOGET_FSCREENINFO, &fb_fix);

    screen_size = fb_fix.line_length * fb_var.yres;
    width = fb_var.xres;
    height = fb_var.yres;

    /* 内存映射 */
    screen_base = mmap(NULL, screen_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (MAP_FAILED == (void *)screen_base) {
        perror("mmap error");
        close(fd);
        return -1;
    }

    /* LCD 背景刷成黑色 */
    memset(screen_base, 0xFF, screen_size);
    return 0;
}

static int freetype_init(const char *font, int angle)
{
    FT_Error error;
    FT_Vector pen;
    FT_Matrix matrix;
    float rad;      //旋转角度

    /* FreeType 初始化 */
    FT_Init_FreeType(&library);

    /* 加载 face 对象 */
```

```

error = FT_New_Face(library, font, 0, &face);
if (error) {
    fprintf(stderr, "FT_New_Face error: %d\n", error);
    exit(EXIT_FAILURE);
}

/* 原点坐标 */
pen.x = 0 * 64;
pen.y = 0 * 64;    //原点设置为(0, 0)

/* 2x2 矩阵初始化 */
rad = (1.0 * angle / 180) * M_PI;    // (角度转换为弧度) M_PI 是圆周率
#if 0    //非水平方向
matrix.xx = (FT_Fixed)( cos(rad) * 0x10000L);
matrix.xy = (FT_Fixed)(-sin(rad) * 0x10000L);
matrix.yx = (FT_Fixed)( sin(rad) * 0x10000L);
matrix.yy = (FT_Fixed)( cos(rad) * 0x10000L);
#endif

#if 1    //斜体 水平方向显示的
matrix.xx = (FT_Fixed)( cos(rad) * 0x10000L);
matrix.xy = (FT_Fixed)( sin(rad) * 0x10000L);
matrix.yx = (FT_Fixed)( 0 * 0x10000L);
matrix.yy = (FT_Fixed)( 1 * 0x10000L);
#endif

/* 设置 */
FT_Set_Transform(face, &matrix, &pen);
FT_Set_Pixel_Sizes(face, 50, 0);    //设置字体大小

return 0;
}

static void lcd_draw_character(int x, int y,
    const wchar_t *str, unsigned int color)
{
    unsigned short rgb565_color = argb8888_to_rgb565(color); //得到 RGB565 颜色值
    FT_GlyphSlot slot = face->glyph;
    size_t len = wcslen(str);    //计算字符的个数
    long int temp;
    int n;
    int i, j, p, q;
    int max_x, max_y, start_y, start_x;

```

```
// 循环加载各个字符
for (n = 0; n < len; n++) {

    // 加载字形、转换得到位图数据
    if (FT_Load_Char(face, str[n], FT_LOAD_RENDER))
        continue;

    start_y = y - slot->bitmap_top; //计算字形轮廓上边 y 坐标起点位置 注意是减去 bitmap_top
    if (0 > start_y) { //如果为负数 如何处理? ?
        q = -start_y;
        temp = 0;
        j = 0;
    }
    else { // 正数又该如何处理??
        q = 0;
        temp = width * start_y;
        j = start_y;
    }

    max_y = start_y + slot->bitmap.rows; //计算字形轮廓下边 y 坐标结束位置
    if (max_y > (int)height)
        max_y = height;

    for (; j < max_y; j++, q++, temp += width) {

        start_x = x + slot->bitmap_left; //起点位置要加上左边空余部分长度
        if (0 > start_x) {
            p = -start_x;
            i = 0;
        }
        else {
            p = 0;
            i = start_x;
        }

        max_x = start_x + slot->bitmap.width;
        if (max_x > (int)width)
            max_x = width;

        for (; i < max_x; i++, p++) {

            // 如果数据不为 0, 则表示需要填充颜色
```

```

        if (slot->bitmap.buffer[q * slot->bitmap.width + p])
            screen_base[temp + i] = rgb565_color;
    }
}

//调整到下一个字形的原点
x += slot->advance.x / 64; //26.6 固定浮点格式
y -= slot->advance.y / 64;
}
}

int main(int argc, char *argv[])
{

    /* LCD 初始化 */
    if (fb_dev_init())
        exit(EXIT_FAILURE);

    /* freetype 初始化 */
    if (freetype_init(argv[1], atoi(argv[2])))
        exit(EXIT_FAILURE);

    /* 在 LCD 上显示中文 */
    int y = height * 0.25;
    lcd_draw_character(50, 100, L"路漫漫其修远兮，吾将上下而求索", 0x000000);
    lcd_draw_character(50, y+100, L"莫愁前路无知己，天下谁人不识君", 0x9900FF);
    lcd_draw_character(50, 2*y+100, L"君不见黄河之水天上来，奔流到海不复回", 0xFF0099);
    lcd_draw_character(50, 3*y+100, L"君不见高堂明镜悲白发，朝如青丝暮成雪", 0x9932CC);

    /* 退出程序 */
    FT_Done_Face(face);
    FT_Done_FreeType(library);
    munmap(screen_base, screen_size);
    close(fd);
    exit(EXIT_SUCCESS);
}

```

编译方法:

```

${CC} -o testApp testApp.c -I/home/dt/tools/freetype/include/freetype2 -L/home/dt/tools/freetype/lib -
lfreetype -L/home/dt/tools/zlib/lib -lz -L/home/dt/tools/png/lib -lpng -lm

```

效果图:

路漫漫其修远兮，吾将上下而求索

莫愁前路无知己，天下谁人不识君

君不见黄河之水天上来，奔流到海不复回

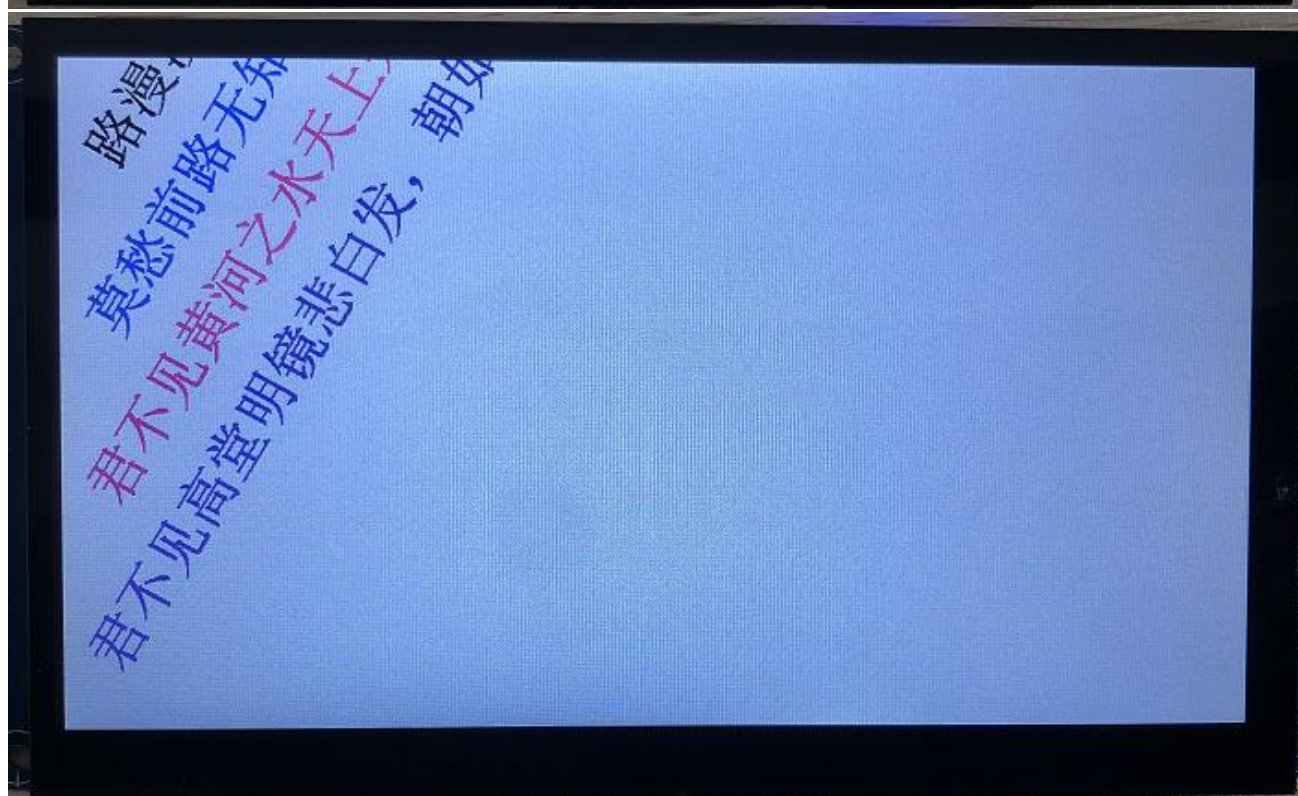
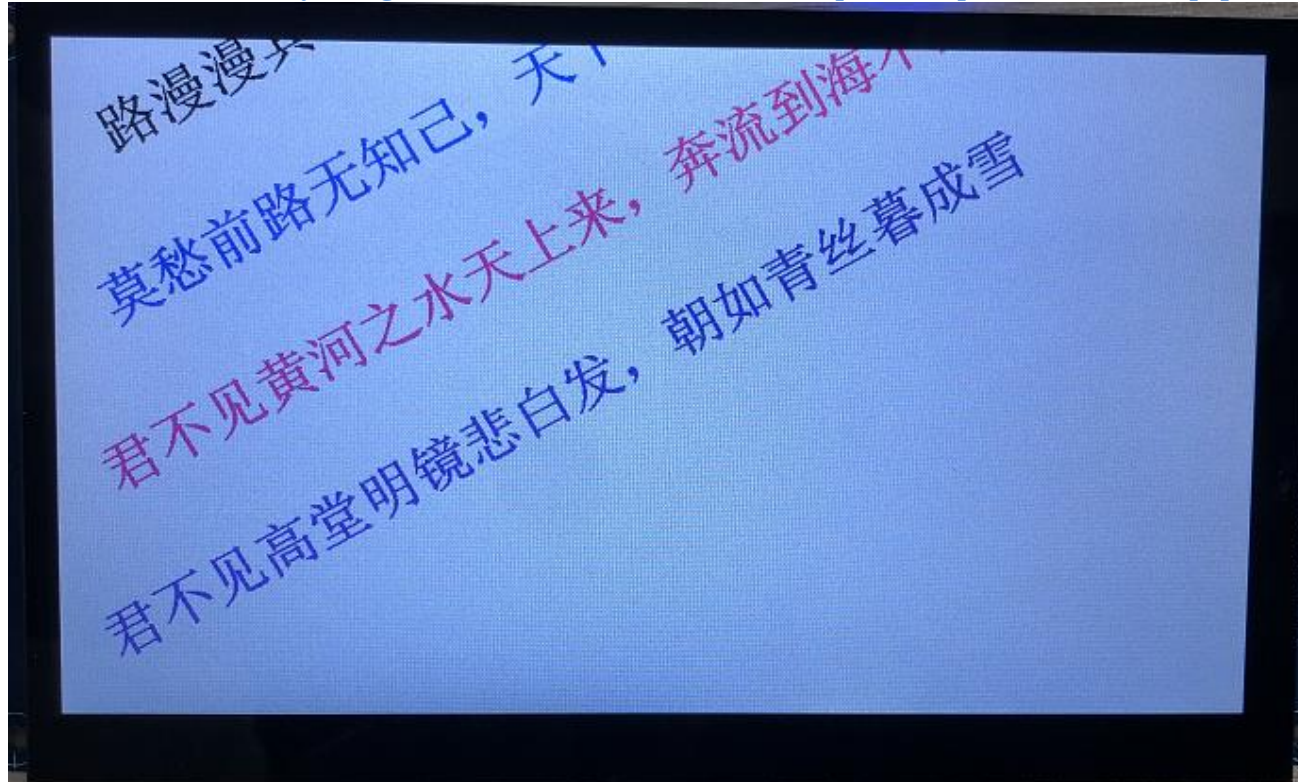
君不见高堂明镜悲白发，朝如青丝暮成雪

路漫漫其修远兮，吾将上下而求索

莫愁前路无知己，天下谁人不识君

君不见黄河之水天上来，奔流到海不复回

君不见高堂明镜悲白发，朝如青丝暮成雪



第二十四章 PWM 应用编程

本章我们将学习如何对开发板上的 PWM 设备进行应用编程。

本章将会讨论如下主题内容。

- 应用层 PWM 编程介绍;
- PWM 测试。

24.1 应用层如何操控 PWM

与 LED 设备一样, PWM 同样也是通过 sysfs 方式进行操控, 进入到 `/sys/class/pwm` 目录下, 如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# cd /sys/class/pwm/
root@ATK-IMX6U:/sys/class/pwm#
root@ATK-IMX6U:/sys/class/pwm# ls
pwmchip0  pwmchip1  pwmchip2  pwmchip3  pwmchip4  pwmchip5  pwmchip6  pwmchip7
root@ATK-IMX6U:/sys/class/pwm#
root@ATK-IMX6U:/sys/class/pwm#
root@ATK-IMX6U:/sys/class/pwm#
root@ATK-IMX6U:/sys/class/pwm#
```

图 24.1.1 /sys/class/pwm 目录下的内容

这里列举出了 8 个以 `pwmchipX` (X 表示数字 0~7) 命名的文件夹, 这八个文件夹其实就对应了 I.MX6U 的 8 个 PWM 控制器, I.MX6U 总共有 8 个 PWM 控制器, 大家可以通过查询 I.MX6U 参考手册得知。

我们随便以其中一个为例, 进入到 `pwmchip0` 目录下:

```
root@ATK-IMX6U:/sys/class/pwm#
root@ATK-IMX6U:/sys/class/pwm# cd pwmchip0/
root@ATK-IMX6U:/sys/class/pwm/pwmchip0#
root@ATK-IMX6U:/sys/class/pwm/pwmchip0# ls
device  export  npwm  power  subsystem  uevent  unexport
root@ATK-IMX6U:/sys/class/pwm/pwmchip0#
root@ATK-IMX6U:/sys/class/pwm/pwmchip0#
```

图 24.1.2 pwmchip0 目录下的内容

在这个目录下我们重点关注的是 `export`、`npwm` 以及 `unexport` 这三个属性文件, 下面一一进行介绍:

- **npwm:** 这是一个只读属性, 读取该文件可以得知该 PWM 控制器下共有几路 PWM 输出, 如下所示:

```
root@ATK-IMX6U:/sys/class/pwm/pwmchip0#
root@ATK-IMX6U:/sys/class/pwm/pwmchip0# cat npwm
1
root@ATK-IMX6U:/sys/class/pwm/pwmchip0#
root@ATK-IMX6U:/sys/class/pwm/pwmchip0#
```

图 24.1.3 读取 npwm 属性文件

I.MX6U 每个 PWM 控制器只有 1 路 PWM 输出, 所以总共有 8 路 PWM, 分别对应 I.MX6U 的 PWM1~PWM8 这 8 路输出 (`pwmchip0` 对应 PWM1, `pwmchip1` 对应 PWM2, 以此类推, 开发板出厂系统中, PWM1 已经被用作 LCD 背光控制了, 应用层不能直接对它进行控制了; 而其它 PWM 均不能使用, 原因在于 I/O 资源不够, 为了满足板子上其它外设对 I/O 引脚的需求, 取舍情况下只能如此!)。

- **export:** 与 GPIO 控制一样, 在使用 PWM 之前, 也需要将其导出, 通过 `export` 属性进行导出, 以下所示:

```
echo 0 > export
```



```

root@ATK-IMX6U:/sys/class/pwm/pwmchip1# pwd
/sys/class/pwm/pwmchip1
root@ATK-IMX6U:/sys/class/pwm/pwmchip1#
root@ATK-IMX6U:/sys/class/pwm/pwmchip1# ls
device export npwm power subsystem uevent unexport
root@ATK-IMX6U:/sys/class/pwm/pwmchip1#
root@ATK-IMX6U:/sys/class/pwm/pwmchip1# echo 0 > export
root@ATK-IMX6U:/sys/class/pwm/pwmchip1#
root@ATK-IMX6U:/sys/class/pwm/pwmchip1# ls
device export npwm power pwm0 subsystem uevent unexport
root@ATK-IMX6U:/sys/class/pwm/pwmchip1#

```

图 24.1.4 导出 PWM

0 表示一个编号, 注意, 每个 PWM 控制器 (pwmchipX) 下, 使用 export 属性文件导出 PWM 时, 编号都是从 0 开始; 因为 I.MX6U 每个控制器都只有一路 PWM, 所以都只能使用编号 0, 如下所示:

```

echo 0 > /sys/class/pwm/pwmchip0/export      #导出 PWM1
echo 0 > /sys/class/pwm/pwmchip1/export      #导出 PWM2
echo 0 > /sys/class/pwm/pwmchip2/export      #导出 PWM3
echo 0 > /sys/class/pwm/pwmchip3/export      #导出 PWM4
echo 0 > /sys/class/pwm/pwmchip4/export      #导出 PWM5
echo 0 > /sys/class/pwm/pwmchip5/export      #导出 PWM6

```

导出成功后会在 pwmchipX (X 表示数字 0~7) 目录下生成一个名为 pwm0 的目录, 如图 24.1.4 所示, 稍后介绍。

- **unexport:** 将导出的 PWM 删除。当使用完 PWM 之后, 我们需要将导出的 PWM 删除, 譬如:

```
echo 0 > unexport
```

写入到 unexport 文件中的编号与写入到 export 文件中的编号是相对应的; 需要注意的是, export 文件和 unexport 文件都是只写的、没有读权限。

如何控制 PWM

通过 export 导出之后, 便会生成 pwm0 这个目录, 我们进入到该目录下看看:

```

root@ATK-IMX6U:/sys/class/pwm/pwmchip1# pwd
/sys/class/pwm/pwmchip1
root@ATK-IMX6U:/sys/class/pwm/pwmchip1#
root@ATK-IMX6U:/sys/class/pwm/pwmchip1# ls
device export npwm power pwm0 subsystem uevent unexport
root@ATK-IMX6U:/sys/class/pwm/pwmchip1#
root@ATK-IMX6U:/sys/class/pwm/pwmchip1# cd pwm0/
root@ATK-IMX6U:/sys/class/pwm/pwmchip1/pwm0#
root@ATK-IMX6U:/sys/class/pwm/pwmchip1/pwm0# ls
duty_cycle enable period polarity power uevent
root@ATK-IMX6U:/sys/class/pwm/pwmchip1/pwm0#

```

图 24.1.5 pwm0 目录下的内容

该目录下也有一些属性文件, 我们重点关注 duty_cycle、enable、period 以及 polarity 这四个属性文件, 接下来一一进行介绍。

- **enable:** 可读可写, 写入"0"表示禁止 PWM; 写入"1"表示使能 PWM。读取该文件获取 PWM 当前是禁止还是使能状态。

```

echo 0 > enable      #禁止 PWM 输出
echo 1 > enable      #使能 PWM 输出

```

通常配置好 PWM 之后, 再使能 PWM。

- **polarity:** 用于设置极性, 可读可写, 可写入的值如下:

"normal": 普通;

"inversed": 反转;

```
echo normal > polarity          #默认极性
```

```
echo inversed > polarity        #极性反转
```

很多 SoC 的 PWM 外设其硬件上并不支持极性配置, 所以对应的驱动程序中并未实现这个接口, 应用层自然也就无法通过 polarity 属性文件对 PWM 极性进行配置, ALPHA/Mini I.MX6U 开发板出厂系统便是如此!

- **period:** 用于配置 PWM 周期, 可读可写; 写入一个字符串数字值, 以 ns (纳秒) 为单位, 譬如配置 PWM 周期为 10us (微秒):

```
echo 10000 > period            #PWM 周期设置为 10us (10 * 1000ns)
```

- **duty_cycle:** 用于配置 PWM 的占空比, 可读可写; 写入一个字符串数字值, 同样也是以 ns 为单位, 譬如:

```
echo 5000 > duty_cycle         #PWM 占空比设置为 5us
```

24.2 编写应用程序

通过上面的介绍, 我们已经知道在应用层如何去使用 PWM 外设了, 本小节我们来编写一个简单的测试代码, 来控制开发板上的 PWM 外设, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->24_pwm->pwm.c](#)。

示例代码 24.2.1 pwm 应用程序示例代码

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : pwm.c
作者 : 邓涛
版本 : V1.0
描述 : PWM 应用程序示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/6/15 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

static char pwm_path[100];

static int pwm_config(const char *attr, const char *val)
{
    char file_path[100];

```

```
int len;
int fd;

sprintf(file_path, "%s/%s", pwm_path, attr);
if (0 > (fd = open(file_path, O_WRONLY))) {
    perror("open error");
    return fd;
}

len = strlen(val);
if (len != write(fd, val, len)) {
    perror("write error");
    close(fd);
    return -1;
}

close(fd); //关闭文件
return 0;
}

int main(int argc, char *argv[])
{
    /* 校验传参 */
    if (4 != argc) {
        fprintf(stderr, "usage: %s <id> <period> <duty>\n",
            argv[0]);
        exit(-1);
    }

    /* 打印配置信息 */
    printf("PWM config: id<%s>, period<%s>, duty<%s>\n",
        argv[1], argv[2],
        argv[3]);

    /* 导出 pwm */
    sprintf(pwm_path, "/sys/class/pwm/pwmchip%s/pwm0", argv[1]);

    if (access(pwm_path, F_OK)) { //如果 pwm0 目录不存在, 则导出

        char temp[100];
        int fd;

        sprintf(temp, "/sys/class/pwm/pwmchip%s/export", argv[1]);
```

```
if (0 > (fd = open(temp, O_WRONLY))) {
    perror("open error");
    exit(-1);
}

if (1 != write(fd, "0", 1)) { //导出 pwm
    perror("write error");
    close(fd);
    exit(-1);
}

close(fd); //关闭文件
}

/* 配置 PWM 周期 */
if (pwm_config("period", argv[2]))
    exit(-1);

/* 配置占空比 */
if (pwm_config("duty_cycle", argv[3]))
    exit(-1);

/* 使能 pwm */
pwm_config("enable", "1");

/* 退出程序 */
exit(0);
}
```

main()函数中, 首先对传参进行校验, 执行该应用程序的时候需要用户传入 3 个参数, 分别是编号 (0、1、2、3 等, 分别表示 I.MX6U 的 PWM1、PWM2、PWM3...)、周期 (以 ns 为单位)、PWM 占空比 (以 ns 为单位)。譬如:

```
./testApp 0 500000 250000
```

接下来需要导出 pwm, 首先使用 access()函数判断 pwm0 目录是否存在, 如果存在表示 pwm 已经导出, 如果不存在, 则表示未导出, 那么就需要通过 export 文件将其导出。

导出成功之后, 接着配置 PWM 周期、占空比, 最后使能 PWM。

编译示例代码:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
testApp testApp.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 24.2.1 编译应用程序

24.3 在开发板上测试

将上小节编译得到的可执行文件拷贝到开发板 Linux 系统/home/root 目录下, 如下所示:

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# pwd  
/home/root  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls  
driver shell testApp  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#
```

图 24.3.1 将测试程序拷贝到开发板/home/root 目录

前面提到了, 开发板出厂系统没法使用 PWM, 如果大家想要测试 PWM, 可以对出厂系统的内核源码进行配置、需修改设备树, 禁用 LCD 和 backlight 背光设备 (status 属性设置为 disabled 即可), 修改完之后重新编译设备树, 用编译得到的设备树镜像文件 (dtb 文件) 替换掉开发板启动文件中的 dtb 文件。也可以参考《I.MX6U 嵌入式 Linux 驱动开发指南》第七十三章内容, 自行配置 PWM。

这里笔者告诉大家一个简单地方法, 不用重新编译设备树文件, 直接把禁用 LCD 和 backlight 背光设备, 将 PWM1 腾出来给我们测试使用, 直接操作呢?

首先我们需要重启开发板, 进入到 u-boot 命令行模式下, 如下:

```

U-Boot 2016.03-ge468cdc (Mar 29 2021 - 15:59:40 +0800)

CPU:   Freescale i.MX6ULL rev1.1 792 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 50C
Reset cause: POR
Board: I.MX6U ALPHA|MINI
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

Display: ATK-LCD-7-800x480 (800x480)
Video: 800x480x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc1(part 0) is current device
Net:   FEC1
Error: FEC1 address not set.

Normal Boot
Hit any key to stop autoboot:  0
=>

```

在倒计时计算前，按任何键停止启动，进入 u-boot 命令行

图 24.3.2 进入到 u-boot 命令行模式

我们要做什么呢？其实就是去修改内核设备树文件，将 LCD 和 backlight 设备的 status 属性修改为“disabled”，禁用这两个设备；怎么修改呢？u-boot 中提供了查看、修改设备树的命令，u-boot 启动时，会将内核设备树（dtb）拷贝到内存中，当拷贝到内存中之后呢，我们就可以去查看或修改设备树了，这里笔者直接把需要执行的命令贴出来，如下所示：

SD/eMMC 启动方式：

```
setenv disable_lcd 'fdt addr ${fdt_addr}; fdt set /backlight status disable; fdt set /soc/aips-bus@02100000/lcdif@021c8000 status disable'
```

下一条命令

```
setenv mmcboot 'echo Booting from mmc ...; run mmcargs; if test ${boot_fdt} = yes || test ${boot_fdt} = try; then if run loadfdt; then run disable_lcd; bootz ${loadaddr} - ${fdt_addr}; else if test ${boot_fdt} = try; then bootz; else echo WARN: Cannot load the DT; fi; fi; else bootz; fi;'
```

NAND 启动方式：

```
setenv disable_lcd 'fdt addr ${fdt_addr}; fdt set /backlight status disable; fdt set /soc/aips-bus@02100000/lcdif@021c8000 status disable'
```

下一条命令

```
setenv bootcmd 'nand read ${loadaddr} 0x620000 0x800000;nand read ${fdt_addr} ${fdt_offset} 0x20000; run disable_lcd; bootz ${loadaddr} - ${fdt_addr}'
```

拷贝时注意格式的问题，分为 SD/eMMC 启动方式和 NAND 启动方式；笔者测试用的开发板是 eMMC 方式启动的，在 u-boot 命令行模式下执行如下命令：

```

Hit any key to stop autoboot:  0
=>
=>
=> setenv disable_lcd 'fdt addr ${fdt_addr}; fdt set /backlight status disable; fdt set /soc/aips-bus@02100000/lcdif@021c8000 status disable'
=>
=> setenv mmcboot 'echo Booting from mmc ...; run mmcargs; if test ${boot_fdt} = yes || test ${boot_fdt} = try; then if run loadfdt; then run di
else if test ${boot_fdt} = try; then bootz; else echo WARN: Cannot load the DT; fi; fi; else bootz; fi;'
=>

```

图 24.3.3 在 u-boot 下执行命令

执行完两条命令后，接着执行 boot 命令启动开发板：

boot

```

=>
=> boot
switch to partitions #0, OK
mmc1(part 0) is current device
switch to partitions #0, OK
mmc1(part 0) is current device
reading boot.scr
** Unable to read file boot.scr **
reading zImage
6785488 bytes read in 220 ms (29.4 MiB/s)
Booting from mmc ...
reading imx6ull-14x14-emmc-7-800x480-c.dtb
39327 bytes read in 18 ms (2.1 MiB/s)
Kernel image @ 0x80800000 [ 0x000000 - 0x6789d0 ]
## Flattened Device Tree blob at 83000000
Booting using the fdt blob at 0x83000000
Using Device Tree in place at 83000000, end 8300c99e

Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.1.15-g14132a2 (alientek@ubuntu) (gcc version 5.3.0 (GCC) ) #1 SMP PREEMPT Mon Jun
[ 0.000000] CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c53c7d

```

图 24.3.4 启动内核

Tips: 这种修改方式只对本次启动生效, 因为我们修改的是内存中的那份设备树文件, 下一次重启开发板时将又恢复到未修改前的状态, 请悉知!

系统启动之后, PWM1 就已经腾出来给我们测试使用了, 此时 LCD 被禁用了!

执行上小节编译得到的可执行文件:

```

root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver  shell  testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp 0 500000 250000
PWM config: id<0>, period<500000>, duty<250000>
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#

```

图 24.3.5 运行测试程序

本实验测试的是 PWM1, 开发板出厂系统已经将 PWM1 输出绑定到了 GPIO1_IO08 引脚(也就是 LCD 背光引脚), 该引脚已经通过开发板上的扩展口引出, 如下所示:



图 24.3.6 GPIO1_IO08 引脚(左图 ALPHA、右图 Mini)

Mini 开发板可以通过背面丝印标注的名称或原理图进行确认。

接下来使用示波器来检测 GPIO1_IO08 引脚输出的 PWM 波形, 如下所示:

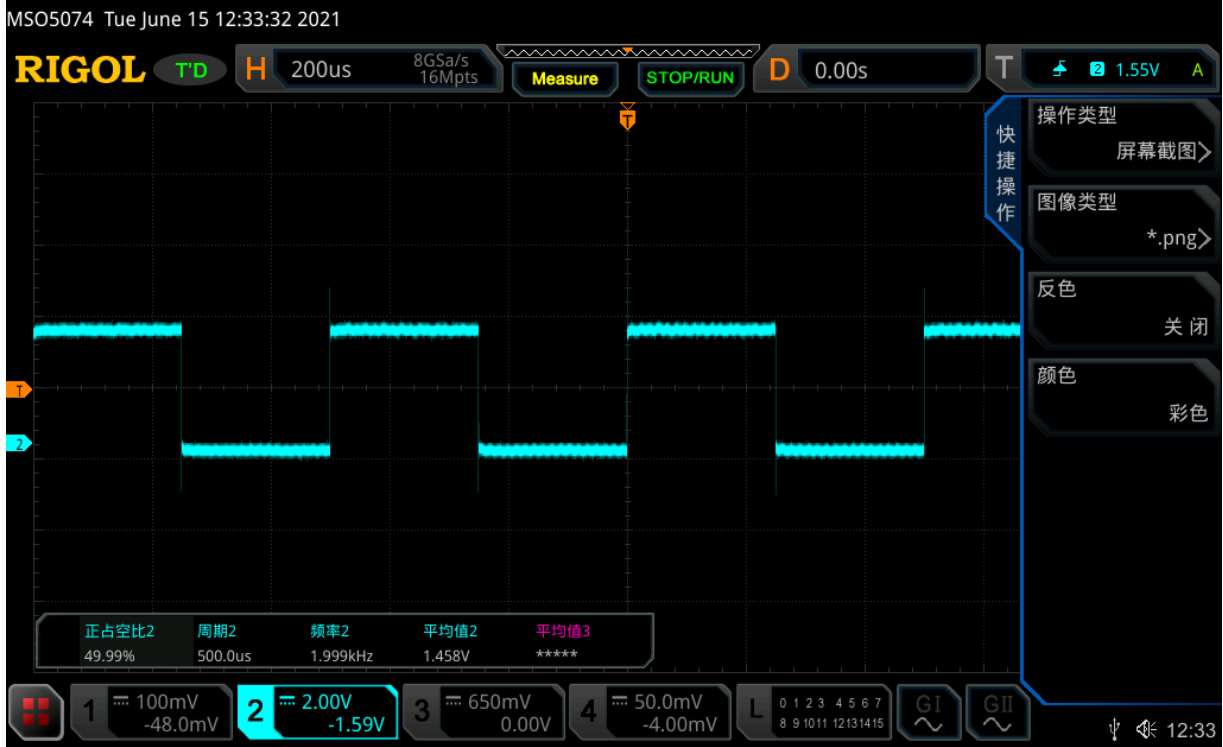


图 24.3.7 PWM 波形

此时 GPIO1_IO08 引脚输出了 PWM 波形, 其周期为 500us (也就是 500000ns), 对应的频率为 2KHz, 占空比为 50%, 与我们配置的情况是一样的。

本章内容到此结束!

第二十五章 V4L2 摄像头应用编程

ALPHA/Mini I.MX6U 开发板配套支持多种不同的摄像头, 包括正点原子的 ov5640 (500W 像素)、ov2640 (200W 像素) 以及 ov7725 (不带 FIFO、30W 像素) 这三款摄像头, 在开发板出厂系统上, 可以使用这些摄像头; 当然, 除此之外我们还可以使用 USB 摄像头, 直接将 USB 摄像头插入到开发板上的 USB 接口即可! 本章我们就来学习 Linux 下的摄像头应用编程。

本章将会讨论如下主题内容。

- V4L2 简介;
- V4L2 设备应用编程介绍;
- 摄像头应用编程实战;

25.1 V4L2 简介

大家可以看到我们本章的标题叫做“V4L2 摄像头应用编程”，那什么是 V4L2 呢？对 Linux 下摄像头驱动程序开发有了解了的读者，应该知道这是什么意思。

V4L2 是 Video for linux two 的简称，是 Linux 内核中视频类设备的一套驱动框架，为视频类设备驱动开发和应用层提供了一套统一的接口规范，那什么是视频类设备呢？一个非常典型的视频类设备就是视频采集设备，譬如各种摄像头；当然还包括其它类型视频类设备，这里就不再介绍了。

使用 V4L2 设备驱动框架注册的设备会在 Linux 系统/dev/目录下生成对应的设备节点文件，设备节点的名称通常为 videoX（X 标准一个数字编号，0、1、2、3.....），每一个 videoX 设备文件就代表一个视频类设备。应用程序通过对 videoX 设备文件进行 I/O 操作来配置、使用设备类设备，下小节将向大家详细介绍！

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls /dev/video*  
/dev/video0 /dev/video1 /dev/video2  
root@ATK-IMX6U:~#
```

图 25.1.1 video 类设备节点

25.2 V4L2 摄像头应用程序

V4L2 设备驱动框架向应用层提供了一套统一、标准的接口规范，应用程序按照该接口规范来进行应用编程，从而使用摄像头。对于摄像头设备来说，其编程模式如下所示：

1. 首先是打开摄像头设备；
2. 查询设备的属性或功能；
3. 设置设备的参数，譬如像素格式、帧大小、帧率；
4. 申请帧缓冲、内存映射；
5. 帧缓冲入队；
6. 开启视频采集；
7. 帧缓冲出队、对采集的数据进行处理；
8. 处理完后，再次将帧缓冲入队，往复；
9. 结束采集。

流程图如下所示：

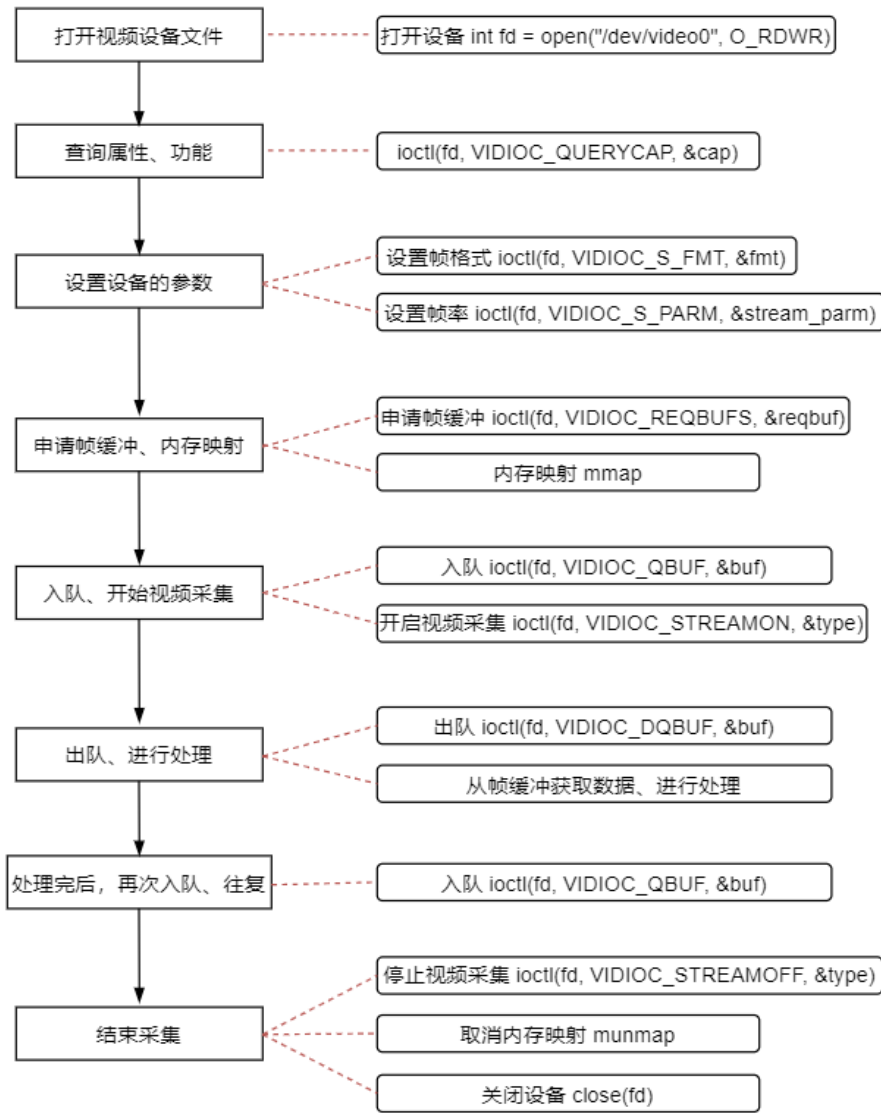


图 25.2.1 摄像头视频采集流程图

从流程图中可以看到，几乎对摄像头的所有操作都是通过 ioctl()来完成，搭配不同的 V4L2 指令 (request 参数) 请求不同的操作，这些指令定义在头文件 linux/videodev2.h 中，在摄像头应用程序代码中，需要包含头文件 linux/videodev2.h，该头文件中申明了很多与摄像头应用编程相关的数据结构以及宏定义，大家可以打开这个头文件看看。

在 videodev2.h 头文件中，定义了很多 ioctl()的指令，以宏定义的形式提供 (VIDIOC_XXX)，如下所示：

```

/*
 * IOCTL CODES FOR VIDEO DEVICES
 *
 */
#define VIDIOC_QUERYCAP      _IOR('V', 0, struct v4l2_capability)
#define VIDIOC_RESERVED     _IO('V', 1)
#define VIDIOC_ENUM_FMT     _IOWR('V', 2, struct v4l2_fmtdesc)
#define VIDIOC_G_FMT        _IOWR('V', 4, struct v4l2_format)
#define VIDIOC_S_FMT        _IOWR('V', 5, struct v4l2_format)
    
```

```

#define VIDIOC_REQBUFS      _IOWR('V', 8, struct v4l2_requestbuffers)
#define VIDIOC_QUERYBUF    _IOWR('V', 9, struct v4l2_buffer)
#define VIDIOC_G_FBUF      _IOR('V', 10, struct v4l2_framebuffer)
#define VIDIOC_S_FBUF      _IOW('V', 11, struct v4l2_framebuffer)
#define VIDIOC_OVERLAY     _IOW('V', 14, int)
#define VIDIOC_QBUF        _IOWR('V', 15, struct v4l2_buffer)
#define VIDIOC_EXPBUF      _IOWR('V', 16, struct v4l2_exportbuffer)
#define VIDIOC_DQBUF       _IOWR('V', 17, struct v4l2_buffer)
#define VIDIOC_STREAMON    _IOW('V', 18, int)
#define VIDIOC_STREAMOFF   _IOW('V', 19, int)
#define VIDIOC_G_PARM      _IOWR('V', 21, struct v4l2_streamparm)
#define VIDIOC_S_PARM      _IOWR('V', 22, struct v4l2_streamparm)
#define VIDIOC_G_STD       _IOR('V', 23, v4l2_std_id)
#define VIDIOC_S_STD       _IOW('V', 24, v4l2_std_id)
#define VIDIOC_ENUMSTD     _IOWR('V', 25, struct v4l2_standard)
#define VIDIOC_ENUMINPUT   _IOWR('V', 26, struct v4l2_input)
#define VIDIOC_G_CTRL      _IOWR('V', 27, struct v4l2_control)
#define VIDIOC_S_CTRL      _IOWR('V', 28, struct v4l2_control)
#define VIDIOC_G_TUNER     _IOWR('V', 29, struct v4l2_tuner)
#define VIDIOC_S_TUNER     _IOW('V', 30, struct v4l2_tuner)
#define VIDIOC_G_AUDIO     _IOR('V', 33, struct v4l2_audio)
#define VIDIOC_S_AUDIO     _IOW('V', 34, struct v4l2_audio)
#define VIDIOC_QUERYCTRL   _IOWR('V', 36, struct v4l2_queryctrl)
#define VIDIOC_QUERYMENU   _IOWR('V', 37, struct v4l2_querymenu)
#define VIDIOC_G_INPUT     _IOR('V', 38, int)
#define VIDIOC_S_INPUT     _IOWR('V', 39, int)
#define VIDIOC_G_EDID      _IOWR('V', 40, struct v4l2_edid)
#define VIDIOC_S_EDID      _IOWR('V', 41, struct v4l2_edid)
#define VIDIOC_G_OUTPUT    _IOR('V', 46, int)
#define VIDIOC_S_OUTPUT    _IOWR('V', 47, int)
#define VIDIOC_ENUMOUTPUT  _IOWR('V', 48, struct v4l2_output)
#define VIDIOC_G_AUDOUT    _IOR('V', 49, struct v4l2_audioout)
#define VIDIOC_S_AUDOUT    _IOW('V', 50, struct v4l2_audioout)
#define VIDIOC_G_MODULATOR _IOWR('V', 54, struct v4l2_modulator)
#define VIDIOC_S_MODULATOR _IOW('V', 55, struct v4l2_modulator)
#define VIDIOC_G_FREQUENCY _IOWR('V', 56, struct v4l2_frequency)
#define VIDIOC_S_FREQUENCY _IOW('V', 57, struct v4l2_frequency)
#define VIDIOC_CROPCAP     _IOWR('V', 58, struct v4l2_cropcap)
#define VIDIOC_G_CROP      _IOWR('V', 59, struct v4l2_crop)
#define VIDIOC_S_CROP      _IOW('V', 60, struct v4l2_crop)
#define VIDIOC_G_JPEGCOMP   _IOR('V', 61, struct v4l2_jpegcompression)
#define VIDIOC_S_JPEGCOMP   _IOW('V', 62, struct v4l2_jpegcompression)
#define VIDIOC_QUERYSTD    _IOR('V', 63, v4l2_std_id)

```

```

#define VIDIOC_TRY_FMT          _IOWR('V', 64, struct v4l2_format)
#define VIDIOC_ENUMAUDIO        _IOWR('V', 65, struct v4l2_audio)
#define VIDIOC_ENUMAUDIOOUT     _IOWR('V', 66, struct v4l2_audioout)
#define VIDIOC_G_PRIORITY       _IOR('V', 67, __u32) /* enum v4l2_priority */
#define VIDIOC_S_PRIORITY       _IOW('V', 68, __u32) /* enum v4l2_priority */
#define VIDIOC_G_SLICED_VBI_CAP _IOWR('V', 69, struct v4l2_sliced_vbi_cap)
#define VIDIOC_LOG_STATUS       _IO('V', 70)
#define VIDIOC_G_EXT_CTRL      _IOWR('V', 71, struct v4l2_ext_controls)
#define VIDIOC_S_EXT_CTRL      _IOWR('V', 72, struct v4l2_ext_controls)
#define VIDIOC_TRY_EXT_CTRL     _IOWR('V', 73, struct v4l2_ext_controls)
#define VIDIOC_ENUM_FRAMESIZES  _IOWR('V', 74, struct v4l2_fmtdesc)
#define VIDIOC_ENUM_FRAMEINTERVALS _IOWR('V', 75, struct v4l2_fmtdesc)
#define VIDIOC_G_ENC_INDEX      _IOR('V', 76, struct v4l2_enc_idx)
#define VIDIOC_ENCODER_CMD      _IOWR('V', 77, struct v4l2_encoder_cmd)
#define VIDIOC_TRY_ENCODER_CMD  _IOWR('V', 78, struct v4l2_encoder_cmd)

```

每一个不同的指令宏就表示向设备请求不同的操作，从上面可以看到，每一个宏后面（_IOWR/_IOR/_IOW）还携带了一个 struct 数据结构体，譬如 struct v4l2_capability、struct v4l2_fmtdesc，这就是调用 ioctl() 时需要传入的第三个参数的类型；调用 ioctl() 前，定义一个该类型变量，调用 ioctl() 时，将变量的指针作为 ioctl() 的第三个参数传入，譬如：

```

struct v4l2_capability cap;
.....
ioctl(fd, VIDIOC_QUERYCAP, &cap);

```

在实际的应用编程中，并不是所有的指令都会用到，针对视频采集类设备，以下笔者列出了一些常用的指令：

V4L2 指令	描述
VIDIOC_QUERYCAP	查询设备的属性/能力/功能
VIDIOC_ENUM_FMT	枚举设备支持的像素格式
VIDIOC_G_FMT	获取设备当前的帧格式信息
VIDIOC_S_FMT	设置帧格式信息
VIDIOC_REQBUFS	申请帧缓冲
VIDIOC_QUERYBUF	查询帧缓冲
VIDIOC_QBUF	帧缓冲入队操作
VIDIOC_DQBUF	帧缓冲出队操作
VIDIOC_STREAMON	开启视频采集
VIDIOC_STREAMOFF	关闭视频采集
VIDIOC_G_PARM	获取设备的一些参数
VIDIOC_S_PARM	设置参数
VIDIOC_TRY_FMT	尝试设置帧格式、用于判断设备是否支持该格式
VIDIOC_ENUM_FRAMESIZES	枚举设备支持的视频采集分辨率
VIDIOC_ENUM_FRAMEINTERVALS	枚举设备支持的视频采集帧率

表 25.2.1 v4l2 摄像头常用的 ioctl 指令

25.2.1 打开摄像头

视频类设备对应的设备节点为/dev/videoX, X 为数字编号, 通常从 0 开始; 摄像头应用编程的第一步便是打开设备, 调用 open 打开, 得到文件描述符 fd, 如下所示:

```
int fd = -1;

/* 打开摄像头 */
fd = open("/dev/video0", O_RDWR);
if (0 > fd) {
    fprintf(stderr, "open error: %s: %s\n", "/dev/video0", strerror(errno));
    return -1;
}
```

打开设备文件时, 需要使用 O_RDWR 指定读权限和写权限。

25.2.2 查询设备的属性/能力/功能

打开设备之后, 接着需要查询设备的属性, 确定该设备是否是一个视频采集类设备、以及其它一些属性, 怎么查询呢? 自然是通过 ioctl() 函数来实现, ioctl() 对于设备文件来说是一个非常重要的系统调用, 凡是涉及到配置设备、获取设备配置等操作都会使用 ioctl 来完成, 在前面章节内容中我们就已经见识过了; 但对于普通文件来说, ioctl() 几乎没什么用。

查询设备的属性, 使用的指令为 VIDIOC_QUERYCAP, 如下所示:

```
ioctl(int fd, VIDIOC_QUERYCAP, struct v4l2_capability *cap);
```

此时通过 ioctl() 将获取到一个 struct v4l2_capability 类型数据, struct v4l2_capability 数据结构描述了设备的一些属性, 结构体定义如下所示:

示例代码 25.2.1 struct v4l2_capability 结构体

```
struct v4l2_capability {
    __u8    driver[16];        /* 驱动的名字 */
    __u8    card[32];         /* 设备的名字 */
    __u8    bus_info[32];     /* 总线的名字 */
    __u32   version;         /* 版本信息 */
    __u32   capabilities;     /* 设备拥有的能力 */
    __u32   device_caps;
    __u32   reserved[3];     /* 保留字段 */
};
```

我们重点关注的是 capabilities 字段, 该字段描述了设备拥有的能力, 该字段的值如下 (可以是以下任意一个值或多个值的位或关系):

```
/* Values for 'capabilities' field */
#define V4L2_CAP_VIDEO_CAPTURE      0x00000001 /* Is a video capture device */
#define V4L2_CAP_VIDEO_OUTPUT      0x00000002 /* Is a video output device */
#define V4L2_CAP_VIDEO_OVERLAY     0x00000004 /* Can do video overlay */
#define V4L2_CAP_VBI_CAPTURE       0x00000010 /* Is a raw VBI capture device */
#define V4L2_CAP_VBI_OUTPUT        0x00000020 /* Is a raw VBI output device */
#define V4L2_CAP_SLICED_VBI_CAPTURE 0x00000040 /* Is a sliced VBI capture device */
#define V4L2_CAP_SLICED_VBI_OUTPUT 0x00000080 /* Is a sliced VBI output device */
```

```

#define V4L2_CAP_RDS_CAPTURE          0x00000100 /* RDS data capture */
#define V4L2_CAP_VIDEO_OUTPUT_OVERLAY 0x00000200 /* Can do video output overlay */
#define V4L2_CAP_HW_FREQ_SEEK        0x00000400 /* Can do hardware frequency seek */
#define V4L2_CAP_RDS_OUTPUT          0x00000800 /* Is an RDS encoder */

/* Is a video capture device that supports multiplanar formats */
#define V4L2_CAP_VIDEO_CAPTURE_MPLANE 0x00001000
/* Is a video output device that supports multiplanar formats */
#define V4L2_CAP_VIDEO_OUTPUT_MPLANE  0x00002000
/* Is a video mem-to-mem device that supports multiplanar formats */
#define V4L2_CAP_VIDEO_M2M_MPLANE     0x00004000
/* Is a video mem-to-mem device */
#define V4L2_CAP_VIDEO_M2M            0x00008000

#define V4L2_CAP_TUNER                 0x00010000 /* has a tuner */
#define V4L2_CAP_AUDIO                 0x00020000 /* has audio support */
#define V4L2_CAP_RADIO                 0x00040000 /* is a radio device */
#define V4L2_CAP_MODULATOR            0x00080000 /* has a modulator */

#define V4L2_CAP_SDR_CAPTURE           0x00100000 /* Is a SDR capture device */
#define V4L2_CAP_EXT_PIX_FORMAT        0x00200000 /* Supports the extended pixel format */
#define V4L2_CAP_SDR_OUTPUT            0x00400000 /* Is a SDR output device */
#define V4L2_CAP_META_CAPTURE          0x00800000 /* Is a metadata capture device */

#define V4L2_CAP_READWRITE              0x01000000 /* read/write systemcalls */
#define V4L2_CAP_ASYNCIO               0x02000000 /* async I/O */
#define V4L2_CAP_STREAMING              0x04000000 /* streaming I/O ioctls */

#define V4L2_CAP_TOUCH                  0x10000000 /* Is a touch device */

#define V4L2_CAP_DEVICE_CAPS            0x80000000 /* sets device capabilities field */

```

这些宏都是在 videodev2.h 头文件中所定义的,大家可以自己去看。对于摄像头设备来说,它的 capabilities 字段必须包含 V4L2_CAP_VIDEO_CAPTURE,表示它支持视频采集功能。所以我们可以通过判断 capabilities 字段是否包含 V4L2_CAP_VIDEO_CAPTURE、来确定它是否是一个摄像头设备,譬如:

```

/* 查询设备功能 */
ioctl(fd, VIDIOC_QUERYCAP, &vcap);

/* 判断是否是视频采集设备 */
if (!(V4L2_CAP_VIDEO_CAPTURE & vcap.capabilities)) {
    fprintf(stderr, "Error: No capture video device!\n");
    return -1;
}

```

25.2.3 设置帧格式、帧率

一个摄像头通常会支持多种不同的像素格式，譬如 RGB、YUYV 以及压缩格式 MJPEG 等，并且还支持多种不同的视频采集分辨率，譬如 640*480、320*240、1280*720 等，除此之外，同一分辨率可能还支持多种不同的视频采集帧率（15fps、30fps）。所以，通常在进行视频采集之前、需要在应用程序中去设置这些参数。

a) 枚举出摄像头支持的所有像素格式：VIDIOC_ENUM_FMT

要设置像素格式，首先得知道该设备支持哪些像素格式，如何得知呢？使用 VIDIOC_ENUM_FMT 指令：

```
ioctl(int fd, VIDIOC_ENUM_FMT, struct v4l2_fmtdesc *fmtdesc);
```

使用 VIDIOC_ENUM_FMT 可以枚举出设备所支持的所有像素格式，调用 ioctl() 需要传入一个 struct v4l2_fmtdesc * 指针，ioctl() 会将获取到的数据写入到 fmtdesc 指针所指向的对象中。struct v4l2_fmtdesc 结构体描述了像素格式相关的信息，我们来看看 struct v4l2_fmtdesc 结构体的定义：

示例代码 25.2.2 struct v4l2_fmtdesc 结构体

```
/*
 * FORMAT ENUMERATION
 */
struct v4l2_fmtdesc {
    __u32      index;           /* Format number */
    __u32      type;           /* enum v4l2_buf_type */
    __u32      flags;
    __u8       description[32]; /* Description string */
    __u32      pixelformat;    /* Format fourcc */
    __u32      reserved[4];
};
```

index 表示编号，在枚举之前，需将其设置为 0，然后每次 ioctl() 调用之后将其值加 1。一次 ioctl() 调用只能得到一种像素格式的信息，如果设备支持多种像素格式，则需要循环调用多次，通过 index 来控制，index 从 0 开始，调用一次 ioctl() 之后加 1，直到 ioctl() 调用失败，表示已经将所有像素格式都枚举出来了；所以 index 就是一个编号，获取 index 编号指定的像素格式。

description 字段是一个简单地描述性字符串，简单描述 pixelformat 像素格式。

pixelformat 字段则是对应的像素格式编号，这是一个无符号 32 位数据，每一种像素格式都会使用一个 u32 类型数据来表示，如下所示：

```
/* RGB formats */
#define V4L2_PIX_FMT_RGB332    v4l2_fourcc('R', 'G', 'B', '1')    /* 8  RGB-3-3-2    */
#define V4L2_PIX_FMT_RGB444    v4l2_fourcc('R', '4', '4', '4')    /* 16  xxxrrrrr ggggbbbb */
#define V4L2_PIX_FMT_ARGB444    v4l2_fourcc('A', 'R', '1', '2')    /* 16  aaaarrrr ggggbbbb */
#define V4L2_PIX_FMT_XRGB444    v4l2_fourcc('X', 'R', '1', '2')    /* 16  xxxrrrrr ggggbbbb */
#define V4L2_PIX_FMT_RGB555    v4l2_fourcc('R', 'G', 'B', 'O')    /* 16  RGB-5-5-5    */
#define V4L2_PIX_FMT_ARGB555    v4l2_fourcc('A', 'R', '1', '5')    /* 16  ARGB-1-5-5-5 */
#define V4L2_PIX_FMT_XRGB555    v4l2_fourcc('X', 'R', '1', '5')    /* 16  XRGB-1-5-5-5 */
#define V4L2_PIX_FMT_RGB565    v4l2_fourcc('R', 'G', 'B', 'P')    /* 16  RGB-5-6-5    */
.....
/* Grey formats */
```



```

#define V4L2_PIX_FMT_GREY      v4l2_fourcc('G', 'R', 'E', 'Y')      /* 8 Greyscale */
#define V4L2_PIX_FMT_Y4       v4l2_fourcc('Y', '0', '4', ' ')    /* 4 Greyscale */
#define V4L2_PIX_FMT_Y6       v4l2_fourcc('Y', '0', '6', ' ')    /* 6 Greyscale */
#define V4L2_PIX_FMT_Y10      v4l2_fourcc('Y', '1', '0', ' ')    /* 10 Greyscale */
.....
/* Luminance+Chrominance formats */
#define V4L2_PIX_FMT_YUYV     v4l2_fourcc('Y', 'U', 'Y', 'V')    /* 16 YUV 4:2:2 */
#define V4L2_PIX_FMT_YYUV     v4l2_fourcc('Y', 'Y', 'U', 'V')    /* 16 YUV 4:2:2 */
#define V4L2_PIX_FMT_YVYU     v4l2_fourcc('Y', 'V', 'Y', 'U')    /* 16 YVU 4:2:2 */
#define V4L2_PIX_FMT_UYVY     v4l2_fourcc('U', 'Y', 'V', 'Y')    /* 16 YUV 4:2:2 */
.....
/* compressed formats */
#define V4L2_PIX_FMT_MJPEG     v4l2_fourcc('M', 'J', 'P', 'G')    /* Motion-JPEG */
#define V4L2_PIX_FMT_JPEG     v4l2_fourcc('J', 'P', 'E', 'G')    /* JFIF JPEG */
#define V4L2_PIX_FMT_DV       v4l2_fourcc('d', 'v', 's', 'd')    /* 1394 */
#define V4L2_PIX_FMT_MPEG     v4l2_fourcc('M', 'P', 'E', 'G')    /* MPEG-1/2/4 Multiplexed */

```

以上列举出来的只是其中一部分，篇幅有限、不能将所有的像素格式都列举出来，大家可以自己查看 videodev2.h 头文件。可以看到后面有一个 v4l2_fourcc 宏，其实就是通过这个宏以及对应的参数合成的一个 u32 类型数据。

type 字段指定类型，表示我们要获取设备的哪种功能对应的像素格式，因为有些设备它可能即支持视频采集功能、又支持视频输出等其它的功能：type 字段可取值如下：

```

enum v4l2_buf_type {
    V4L2_BUF_TYPE_VIDEO_CAPTURE      = 1, //视频采集
    V4L2_BUF_TYPE_VIDEO_OUTPUT       = 2, //视频输出
    V4L2_BUF_TYPE_VIDEO_OVERLAY      = 3,
    V4L2_BUF_TYPE_VBI_CAPTURE        = 4,
    V4L2_BUF_TYPE_VBI_OUTPUT         = 5,
    V4L2_BUF_TYPE_SLICED_VBI_CAPTURE = 6,
    V4L2_BUF_TYPE_SLICED_VBI_OUTPUT  = 7,
    V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY = 8,
    V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE = 9,
    V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE = 10,
    V4L2_BUF_TYPE_SDR_CAPTURE        = 11,
    V4L2_BUF_TYPE_SDR_OUTPUT         = 12,
    V4L2_BUF_TYPE_META_CAPTURE       = 13,
    /* Deprecated, do not use */
    V4L2_BUF_TYPE_PRIVATE            = 0x80,
};

```

type 字段需要在调用 ioctl() 之前设置它的值，对于摄像头，需要将 type 字段设置为 V4L2_BUF_TYPE_VIDEO_CAPTURE，指定我们将要获取的是视频采集的像素格式。

使用示例如下所示：

```
struct v4l2_fmtdesc fmtdesc;
```

```
/* 枚举出摄像头所支持的所有像素格式以及描述信息 */
fmtdesc.index = 0;
fmtdesc.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
while (0 == ioctl(fd, VIDIOC_ENUM_FMT, &fmtdesc)) {

    printf("fmt: %s <0x%x>\n", fmtdesc.description, fmtdesc.pixelformat);
    fmtdesc.index++;
}

```

b)枚举摄像头所支持的所有视频采集分辨率: VIDIOC_ENUM_FRAMESIZES

使用 VIDIOC_ENUM_FRAMESIZES 指令可以枚举出设备所支持的所有视频采集分辨率, 用法如下所示:

```
ioctl(int fd, VIDIOC_ENUM_FRAMESIZES, struct v4l2_frmsizeenum *frmsize);
```

调用 ioctl() 需要传入一个 struct v4l2_frmsizeenum * 指针, ioctl() 会将获取到的数据写入到 frmsize 指针所指向的对象中。struct v4l2_frmsizeenum 结构体描述了视频帧大小相关的信息, 我们来看看 struct v4l2_frmsizeenum 结构体的定义:

示例代码 25.2.3 struct v4l2_frmsizeenum 结构体

```
struct v4l2_frmsizeenum {
    __u32      index;          /* Frame size number */
    __u32      pixel_format;  /* 像素格式 */
    __u32      type;          /* type */

    union {                  /* Frame size */
        struct v4l2_frmsize_discrete discrete;
        struct v4l2_frmsize_stepwise stepwise;
    };

    __u32      reserved[2];   /* Reserved space for future use */
};

struct v4l2_frmsize_discrete {
    __u32      width;         /* Frame width [pixel] */
    __u32      height;       /* Frame height [pixel] */
};

```

index 字段与 struct v4l2_fmtdesc 结构体的 index 字段意义相同, 一个摄像头通常支持多种不同的视频采集分辨率, 一次 ioctl() 调用只能得到一种视频帧大小信息, 如果设备支持多种视频帧大小, 则需要循环调用多次, 通过 index 来控制。

pixel_format 字段指定像素格式, 而 type 字段与 struct v4l2_fmtdesc 结构体的 type 字段意义相同; 在调用 ioctl() 之前, 需要先设置 type 字段与 pixel_format 字段, 确定我们将要枚举的是: 设备的哪种功能、哪种像素格式支持的视频帧大小。

可以看到 struct v4l2_frmsizeenum 结构体中有一个 union 共用体, type=V4L2_BUF_TYPE_VIDEO_CAPTURE 情况下, discrete 生效, 这是一个 struct v4l2_frmsize_discrete 类型变量, 描述了视频帧大小信息(包括视频帧的宽度和高度), 也就是视频采集分辨率大小。

譬如我们要枚举出摄像头 RGB565 像素格式所支持的所有视频帧大小:

```
struct v4l2_frmsizeenum frmsize;
```

```
frmsize.index = 0;
frmsize.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
frmsize.pixel_format = V4L2_PIX_FMT_RGB565;
while (0 == ioctl(fd, VIDIOC_ENUM_FRAMESIZES, &frmsize)) {

    printf("frame_size<%d*%d>\n", frmsize.discrete.width, frmsize.discrete.height);
    frmsize.index++;
}
}
```

c)枚举摄像头所支持的所有视频采集帧率: VIDIOC_ENUM_FRAMEINTERVALS

同一种视频帧大小,摄像头可能会支持多种不同的视频采集帧率,譬如常见的 15fps、30fps、45fps 以及 60fps 等;使用 VIDIOC_ENUM_FRAMEINTERVALS 指令可以枚举出设备所支持的所有帧率,使用方式如下:

```
ioctl(int fd, VIDIOC_ENUM_FRAMEINTERVALS, struct v4l2_fmvalenum *frmival);
```

调用 ioctl()需要传入一个 struct v4l2_fmvalenum *指针,ioctl()会将获取到的数据写入到 frmival 指针所指向的对象中。struct v4l2_fmvalenum 结构体描述了视频帧率相关的信息,我们来看看 struct v4l2_fmvalenum 结构体的定义:

示例代码 25.2.4 struct v4l2_fmvalenum 结构体

```
struct v4l2_fmvalenum {
    __u32      index;          /* Frame format index */
    __u32      pixel_format; /* Pixel format */
    __u32      width;         /* Frame width */
    __u32      height;        /* Frame height */
    __u32      type;          /* type */

    union {                  /* Frame interval */
        struct v4l2_fract      discrete;
        struct v4l2_fmval_stepwise stepwise;
    };

    __u32      reserved[2];   /* Reserved space for future use */
};

struct v4l2_fract {
    __u32      numerator;     //分子
    __u32      denominator;  //分母
};
```

index、type 字段与 struct v4l2_frmsizeenum 结构体的 index、type 字段意义相同。

width、height 字段用于指定视频帧大小, pixel_format 字段指定像素格式。

以上这些字段都是需要在调用 ioctl()之前设置它的值。

可以看到 struct v4l2_fmvalenum 结构体也有一个 union 共用体,当 type=V4L2_BUF_TYPE_VIDEO_CAPTURE 时, discrete 生效,这是一个 struct v4l2_fract 类型变量,描述了视频

帧率信息（一秒钟采集图像的帧数）；struct v4l2_fract 结构体中，numerator 表示分子、denominator 表示分母，使用 numerator / denominator 来表示图像采集的周期（采集一幅图像需要多少秒），所以视频帧率等于 denominator / numerator。

使用示例，譬如，我们要枚举出 RGB565 像素格式下 640*480 帧大小所支持的所有视频采集帧率：

```
struct v4l2_fmvalenum fmval;

fmval.index = 0;
fmval.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmval.pixel_format = V4L2_PIX_FMT_RGB565;
fmval.width = 640;
fmval.height = 480;

while (0 == ioctl(fd, VIDIOC_ENUM_FRAMEINTERVALS, &fmval)) {

    printf("Frame interval<%ffps> ", fmval.discrete.denominator / fmval.discrete.numerator);
    fmval.index++;
}
```

d) 查看或设置当前的格式：VIDIOC_G_FMT、VIDIOC_S_FMT

前面介绍的指令只是枚举设备支持的像素格式、视频帧大小以及视频采集帧率等这些信息，将下来我们将介绍如何设置这些参数。

首先可以使用 VIDIOC_G_FMT 指令查看设备当前的格式，用法如下所示

```
int ioctl(int fd, VIDIOC_G_FMT, struct v4l2_format *fmt);
```

调用 ioctl() 需要传入一个 struct v4l2_format * 指针，ioctl() 会将获取到的数据写入到 fmt 指针所指向的对象中，struct v4l2_format 结构体描述了格式相关的信息。

使用 VIDIOC_S_FMT 指令设置设备的格式，用法如下所示：

```
int ioctl(int fd, VIDIOC_S_FMT, struct v4l2_format *fmt);
```

ioctl() 会使用 fmt 所指对象的数据去设置设备的格式。我们来看看 v4l2_format 结构体的定义：

示例代码 25.2.5 struct v4l2_format 结构体

```
struct v4l2_format {
    __u32    type;
    union {
        struct v4l2_pix_format      pix;      /* V4L2_BUF_TYPE_VIDEO_CAPTURE */
        struct v4l2_pix_format_mplane pix_mp; /* V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE */
        struct v4l2_window          win;      /* V4L2_BUF_TYPE_VIDEO_OVERLAY */
        struct v4l2_vbi_format      vbi;      /* V4L2_BUF_TYPE_VBI_CAPTURE */
        struct v4l2_sliced_vbi_format sliced; /* V4L2_BUF_TYPE_SLICED_VBI_CAPTURE */
        struct v4l2_sdr_format      sdr;      /* V4L2_BUF_TYPE_SDR_CAPTURE */
        struct v4l2_meta_format     meta;     /* V4L2_BUF_TYPE_META_CAPTURE */
        __u8    raw_data[200];             /* user-defined */
    } fmt;
};
```

type 字段依然与前面介绍的结构体中的 type 字段意义相同，不管是获取格式、还是设置格式都需要在调用 ioctl() 函数之前设置它的值。

接下来是一个 union 共用体, 当 type 被设置为 V4L2_BUF_TYPE_VIDEO_CAPTURE 时, pix 变量生效, 它是一个 struct v4l2_pix_format 类型变量, 记录了视频帧格式相关的信息, 如下所示:

示例代码 25.2.6 struct v4l2_pix_format 结构体

```
struct v4l2_pix_format {
    __u32      width;           //视频帧的宽度 (单位: 像素)
    __u32      height;         //视频帧的高度 (单位: 像素)
    __u32      pixelformat;     //像素格式
    __u32      field;          /* enum v4l2_field */
    __u32      bytesperline;    /* for padding, zero if unused */
    __u32      sizeimage;
    __u32      colorspace;     /* enum v4l2_colorspace */
    __u32      priv;           /* private data, depends on pixelformat */
    __u32      flags;          /* format flags (V4L2_PIX_FMT_FLAG_*) */
    union {
        /* enum v4l2_ycbcr_encoding */
        __u32      ycbcr_enc;
        /* enum v4l2_hsv_encoding */
        __u32      hsv_enc;
    };
    __u32      quantization;    /* enum v4l2_quantization */
    __u32      xfer_func;      /* enum v4l2_xfer_func */
};
```

colorspace 字段描述的是一个颜色空间, 可取值如下:

```
enum v4l2_colorspace {
    /*
     * Default colorspace, i.e. let the driver figure it out.
     * Can only be used with video capture.
     */
    V4L2_COLORSPACE_DEFAULT      = 0,

    /* SMPTE 170M: used for broadcast NTSC/PAL SDTV */
    V4L2_COLORSPACE_SMPTE170M   = 1,

    /* Obsolete pre-1998 SMPTE 240M HDTV standard, superseded by Rec 709 */
    V4L2_COLORSPACE_SMPTE240M   = 2,

    /* Rec.709: used for HDTV */
    V4L2_COLORSPACE_REC709      = 3,

    /*
     * Deprecated, do not use. No driver will ever return this. This was
     * based on a misunderstanding of the bt878 datasheet.
     */
};
```

```

V4L2_COLORSPACE_BT878          = 4,

/*
 * NTSC 1953 colorspace. This only makes sense when dealing with
 * really, really old NTSC recordings. Superseded by SMPTE 170M.
 */
V4L2_COLORSPACE_470_SYSTEM_M  = 5,

/*
 * EBU Tech 3213 PAL/SECAM colorspace. This only makes sense when
 * dealing with really old PAL/SECAM recordings. Superseded by
 * SMPTE 170M.
 */
V4L2_COLORSPACE_470_SYSTEM_BG = 6,

/*
 * Effectively shorthand for V4L2_COLORSPACE_SRGB, V4L2_YCBCR_ENC_601
 * and V4L2_QUANTIZATION_FULL_RANGE. To be used for (Motion-)JPEG.
 */
V4L2_COLORSPACE_JPEG          = 7,

/* For RGB colorspaces such as produces by most webcams. */
V4L2_COLORSPACE_SRGB          = 8,

/* AdobeRGB colorspace */
V4L2_COLORSPACE_ADOBERGB     = 9,

/* BT.2020 colorspace, used for UHD TV. */
V4L2_COLORSPACE_BT2020       = 10,

/* Raw colorspace: for RAW unprocessed images */
V4L2_COLORSPACE_RAW          = 11,

/* DCI-P3 colorspace, used by cinema projectors */
V4L2_COLORSPACE_DCI_P3       = 12,
};

```

使用 `VIDIOC_S_FMT` 指令设置格式时，通常不需要用户指定 `colorspace`，底层驱动会根据像素格式 `pixelformat` 来确定对应的 `colorspace`。

例子：获取当前的格式、并设置格式

```

struct v4l2_format fmt;

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (0 > ioctl(fd, VIDIOC_G_FMT, &fmt)) { //获取格式信息

```

```

    perror("ioctl error");
    return -1;
}

printf("width:%d, height:%d format:%d\n", fmt.fmt.pix.width, fmt.fmt.pix.height, fmt.fmt.pix.pixelformat);

fmt.fmt.pix.width = 800;
fmt.fmt.pix.height = 480;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB565;
if (0 > ioctl(fd, VIDIOC_S_FMT, &fmt)) { //设置格式
    perror("ioctl error");
    return -1;
}

```

使用指令 VIDIOC_S_FMT 设置格式时, 实际设置的参数并不一定等于我们指定的参数, 譬如上面我们指定视频帧宽度为 800、高度为 480, 但这个摄像头不一定支持这种视频帧大小, 或者摄像头不支持 V4L2_PIX_FMT_RGB565 这种像素格式; 通常在这种情况下, 底层驱动程序并不会按照我们指定的参数进行设置, 它会对这些参数进行修改, 譬如, 如果摄像头不支持 800*480, 那么底层驱动可能会将其修改为 640*480(假设摄像头支持这种分辨率); 所以, 当 ioctl()调用返回后, 我们还需要检查返回的 struct v4l2_format 类型变量, 以确定我们指定的参数是否已经生效:

```

struct v4l2_format fmt;

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width = 800;
fmt.fmt.pix.height = 480;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB565;
if (0 > ioctl(fd, VIDIOC_S_FMT, &fmt)) { //设置格式
    perror("ioctl error");
    return -1;
}

if (800 != fmt.fmt.pix.width ||
    480 != fmt.fmt.pix.height){
    do_something();
}

if (V4L2_PIX_FMT_RGB565 != fmt.fmt.pix.pixelformat) {
    do_something();
}

```

e) 设置或获取当前的流类型相关参数: VIDIOC_G_PARM、VIDIOC_S_PARM

使用 VIDIOC_G_PARM 指令可以获取设备的流类型相关参数 (Stream type-dependent parameters), 使用方式如下:

```
ioctl(int fd, VIDIOC_G_PARM, struct v4l2_streamparm *streamparm);
```

调用 `ioctl()` 需要传入一个 `struct v4l2_streamparm *` 指针, `ioctl()` 会将获取到的数据写入到 `streamparm` 指针所指向的对象中, `struct v4l2_streamparm` 结构体描述了流类型相关的信息, 具体的内容等会在介绍。

使用 `VIDIOC_S_PARM` 指令设置设备的流类型相关参数, 用法如下所示:

```
ioctl(int fd, VIDIOC_S_PARM, struct v4l2_streamparm *streamparm);
```

`ioctl()` 会使用 `streamparm` 所指对象的数据去设置设备的流类型相关参数。我们来看看 `struct v4l2_streamparm` 结构体的定义:

示例代码 25.2.7 `struct v4l2_streamparm` 结构体

```
struct v4l2_streamparm {
    __u32    type;           /* enum v4l2_buf_type */
    union {
        struct v4l2_captureparm    capture;
        struct v4l2_outputparm     output;
        __u8    raw_data[200]; /* user-defined */
    } parm;
};

struct v4l2_captureparm {
    __u32    capability;    /* Supported modes */
    __u32    capturemode;   /* Current mode */
    struct v4l2_fract    timeperframe; /* Time per frame in seconds */
    __u32    extendedmode; /* Driver-specific extensions */
    __u32    readbuffers;   /* # of buffers for read */
    __u32    reserved[4];
};

struct v4l2_fract {
    __u32    numerator;    /* 分子 */
    __u32    denominator; /* 分母 */
};
```

`type` 字段与前面一样, 不再介绍, 在调用 `ioctl()` 之前需先设置它的值。

当 `type= V4L2_BUF_TYPE_VIDEO_CAPTURE` 时, `union` 共用体中 `capture` 变量生效, 它是一个 `struct v4l2_captureparm` 类型变量, `struct v4l2_captureparm` 结构体描述了摄像头采集相关的一些参数, 譬如视频采集帧率, 上面已经给出了该结构体的定义。

`struct v4l2_captureparm` 结构体中, `capability` 字段表示设备支持的模式有哪些, 可取值如下 (以下任意一个或多个的位或关系):

```
/* Flags for 'capability' and 'capturemode' fields */
#define V4L2_MODE_HIGHQUALITY    0x0001 /* High quality imaging mode 高品质成像模式 */
#define V4L2_CAP_TIMEPERFRAME    0x1000 /* timeperframe field is supported 支持设置 timeperframe 字段 */
```

`capturemode` 则表示当前的模式, 与 `capability` 字段的取值相同。

`timeperframe` 字段是一个 `struct v4l2_fract` 结构体类型变量, 描述了设备视频采集的周期, 前面已经给大家介绍过。使用 `VIDIOC_S_PARM` 可以设置视频采集的周期, 也就是视频采集帧率, 但是很多设备并不支

持应用层设置 `timeperframe` 字段, 只有当 `capability` 字段包含 `V4L2_CAP_TIMEPERFRAME` 时才表示设备支持 `timeperframe` 字段, 这样应用层才可以去设置设备的视频采集帧率。

所以, 在设置之前, 先通过 `VIDIOC_G_PARM` 命令获取到设备的流类型相关参数, 判断 `capability` 字段是否包含 `V4L2_CAP_TIMEPERFRAME`, 如下所示:

```
struct v4l2_streamparm streamparm;

streamparm.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ioctl(v4l2_fd, VIDIOC_G_PARM, &streamparm);

/** 判断是否支持帧率设置 **/
if (V4L2_CAP_TIMEPERFRAME & streamparm.parm.capture.capability) {
    streamparm.parm.capture.timeperframe.numerator = 1;
    streamparm.parm.capture.timeperframe.denominator = 30;//30fps

    if (0 > ioctl(v4l2_fd, VIDIOC_S_PARM, &streamparm)) { //设置参数
        fprintf(stderr, "ioctl error: VIDIOC_S_PARM: %s\n", strerror(errno));
        return -1;
    }
}
else
    fprintf(stderr, "不支持帧率设置");
```

25.2.4 申请帧缓冲、内存映射

读取摄像头数据的方式有两种, 一种是 `read` 方式, 也就是直接通过 `read()` 系统调用读取摄像头采集到的数据; 另一种则是 `streaming` 方式; 25.2.2 小节中介绍了使用 `VIDIOC_QUERYCAP` 指令查询设备的属性, 得到一个 `struct v4l2_capability` 类型数据, 其中 `capabilities` 字段记录了设备拥有的能力, 当该字段包含 `V4L2_CAP_READWRITE` 时, 表示设备支持 `read I/O` 方式读取数据; 当该字段包含 `V4L2_CAP_STREAMING` 时, 表示设备支持 `streaming I/O` 方式; 事实上, 绝大部分设备都支持 `streaming I/O` 方式读取数据, 使用 `streaming I/O` 方式, 我们需要向设备申请帧缓冲, 并将帧缓冲映射到应用程序进程地址空间中。

当完成对设备的配置之后, 接下来就可以去申请帧缓冲了, 帧缓冲顾名思义就是用于存储一帧图像数据的缓冲区, 使用 `VIDIOC_REQBUFS` 指令可申请帧缓冲, 使用方式如下所示:

```
ioctl(int fd, VIDIOC_REQBUFS, struct v4l2_requestbuffers *reqbuf);
```

调用 `ioctl()` 需要传入一个 `struct v4l2_requestbuffers *` 指针, `struct v4l2_requestbuffers` 结构体描述了申请帧缓冲的信息, `ioctl()` 会根据 `reqbuf` 所指对象填充的信息进行申请。我们来看看 `struct v4l2_requestbuffers` 结构体的定义:

示例代码 25.2.8 `struct v4l2_requestbuffers` 结构体

```
/*
 * MEMORY - MAPPING  B U F F E R S
 */
struct v4l2_requestbuffers {
    __u32          count;          //申请帧缓冲的数量
    __u32          type;          /* enum v4l2_buf_type */
    __u32          memory;        /* enum v4l2_memory */
```

```

__u32          reserved[2];
};

```

type 字段与前面所提及到的 type 字段意义相同, 不再介绍, 在调用 ioctl() 之前需先设置它的值。
count 字段用于指定申请帧缓冲的数量。
memory 字段可取值如下:

```

enum v4l2_memory {
    V4L2_MEMORY_MMAP      = 1,
    V4L2_MEMORY_USERPTR  = 2,
    V4L2_MEMORY_OVERLAY  = 3,
    V4L2_MEMORY_DMABUF   = 4,
};

```

通常将 memory 设置为 V4L2_MEMORY_MMAP 即可! 使用示例如下:

```

struct v4l2_requestbuffers reqbuf;

reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.count = 3;          // 申请 3 个帧缓冲
reqbuf.memory = V4L2_MEMORY_MMAP;

if (0 > ioctl(fd, VIDIOC_REQBUFS, &reqbuf)) {
    fprintf(stderr, "ioctl error: VIDIOC_REQBUFS: %s\n", strerror(errno));
    return -1;
}

```

streaming I/O 方式会在内核空间中维护一个帧缓冲队列, 驱动程序会将从摄像头读取的一帧数据写入到队列中的一个帧缓冲, 接着将下一帧数据写入到队列中的下一个帧缓冲; 当应用程序需要读取一帧数据时, 需要从队列中取出一个装满一帧数据的帧缓冲, 这个取出过程就叫做出队; 当应用程序处理完这一帧数据后, 需要再把这个帧缓冲加入到内核的帧缓冲队列中, 这个过程叫做入队! 这个很容易理解, 现实当中都有很多这样的例子, 这里就不再举例了。

所以由此可知, 读取图像数据的过程其实就是一个不断地出队列和入队列的过程, 如下图所示

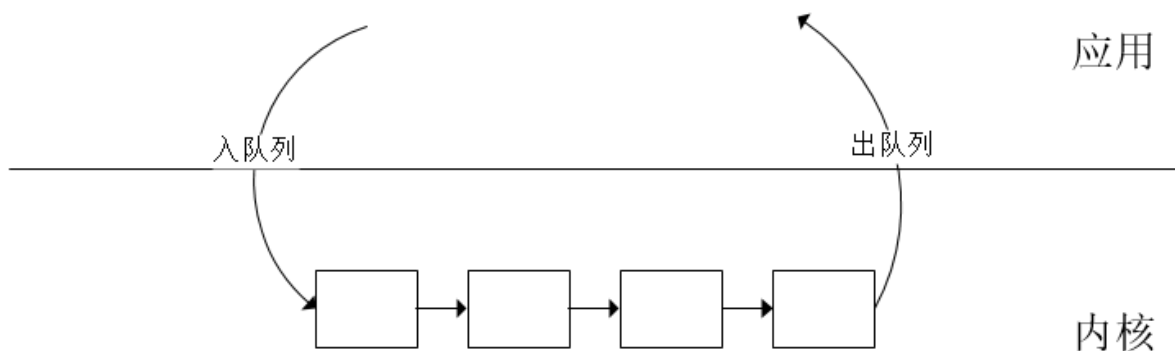


图 25.2.2 应用层读取图像数据的过程

将帧缓冲映射到进程地址空间

使用 VIDIOC_REQBUFS 指令申请帧缓冲, 该缓冲区实质上是由内核所维护的, 应用程序不能直接读取该缓冲区的数据, 我们需要将其映射到用户空间中, 这样, 应用程序读取映射区的数据实际上就是读取内核维护的帧缓冲中的数据。

在映射之前, 需要查询帧缓冲的信息, 譬如帧缓冲的长度、偏移量等信息, 使用 VIDIOC_QUERYBUF 指令查询, 使用方式如下所示:

```
ioctl(int fd, VIDIOC_QUERYBUF, struct v4l2_buffer *buf);
```

调用 ioctl() 需要传入一个 struct v4l2_buffer * 指针, struct v4l2_buffer 结构体描述了帧缓冲的信息, ioctl() 会将获取到的数据写入到 buf 指针所指的对象中。我们来看看 struct v4l2_buffer 结构体的定义:

示例代码 25.2.9 struct v4l2_buffer 结构体

```
struct v4l2_buffer {
    __u32      index;          //buffer 的编号
    __u32      type;          //type
    __u32      bytesused;
    __u32      flags;
    __u32      field;
    struct timeval    timestamp;
    struct v4l2_timecode    timecode;
    __u32      sequence;

    /* memory location */
    __u32      memory;
    union {
        __u32      offset;    //偏移量
        unsigned long    userptr;
        struct v4l2_plane    *planes;
        __s32      fd;
    } m;
    __u32      length;        //buffer 的长度
    __u32      reserved2;
    __u32      reserved;
};
```

index 字段表示一个编号, 申请的多个帧缓冲、每一个帧缓冲都有一个编号, 从 0 开始。一次 ioctl() 调用只能获取指定编号对应的帧缓冲的信息, 所以要获取多个帧缓冲的信息, 需要重复调用多次, 每调用一次 ioctl()、index 加 1, 指向下一个帧缓冲。

type 字段与前面所提及到的 type 字段意义相同, 不再介绍, 在调用 ioctl() 之前需先设置它的值。

memory 字段与 struct v4l2_requestbuffers 结构体的 memory 字段意义相同, 需要在调用 ioctl() 之前设置它的值。

length 字段表示帧缓冲的长度, 而共同体中的 offset 则表示帧缓冲的偏移量, 如何理解这个偏移量? 因为应用程序通过 VIDIOC_REQBUFS 指令申请帧缓冲时, 内核会向操作系统申请一块内存空间作为帧缓冲区, 这块内存空间的大小就等于申请的帧缓冲数量 * 每一个帧缓冲的大小, 每一个帧缓冲对应到这一块内存空间的某一段, 所以它们都有一个地址偏移量。

帧缓冲的数量不要太多了, 尤其是在一些内存比较吃紧的嵌入式系统中, 帧缓冲的数量太多, 势必会占用太多的系统内存。

使用示例, 申请帧缓冲后、调用 mmap() 将帧缓冲映射到用户地址空间:

```
struct v4l2_requestbuffers reqbuf;
struct v4l2_buffer buf;
```

```
void *frm_base[3];

reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.count = 3;      // 申请 3 个帧缓冲
reqbuf.memory = V4L2_MEMORY_MMAP;

/* 申请 3 个帧缓冲 */
if (0 > ioctl(fd, VIDIOC_REQBUFS, &reqbuf) {
    fprintf(stderr, "ioctl error: VIDIOC_REQBUFS: %s\n", strerror(errno));
    return -1;
}

/* 建立内存映射 */
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
for (buf.index = 0; buf.index < 3; buf.index++) {

    ioctl(fd, VIDIOC_QUERYBUF, &buf);
    frm_base[buf.index] = mmap(NULL, buf.length,
                               PROT_READ | PROT_WRITE, MAP_SHARED,
                               fd, buf.m.offset);
    if (MAP_FAILED == frm_base[buf.index]) {
        perror("mmap error");
        return -1;
    }
}
}
```

在上述的示例中,我们会将三个帧缓冲映射到用户空间,并将每一个帧缓冲对应的映射区的起始地址保存在 `frm_base` 数组中,后面读取摄像头采集的数据时,直接读取映射区即可。

25.2.5 入队

使用 `VIDIOC_QBUF` 指令将帧缓冲放入到内核的帧缓冲队列中,使用方式如下:

```
ioctl(int fd, VIDIOC_QBUF, struct v4l2_buffer *buf);
```

调用 `ioctl()` 之前,需要设置 `struct v4l2_buffer` 类型对象的 `memory`、`type` 字段,使用示例如下所示:

将三个帧缓冲放入内核的帧缓冲队列(入队操作)中:

```
struct v4l2_buffer buf;

/* 入队操作 */
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
for (buf.index = 0; buf.index < 3; buf.index++) {

    if (0 > ioctl(fd, VIDIOC_QBUF, &buf) {
        perror("ioctl error");
    }
}
```

```
        return -1;
    }
}
```

25.2.6 开启视频采集

将三个帧缓冲放入到队列中之后,接着便可以打开摄像头、开启图像采集了,使用 VIDIOC_DQBUF 指令开启视频采集,使用方式如下所示:

```
ioctl(int fd, VIDIOC_STREAMON, int *type);    //开启视频采集
ioctl(int fd, VIDIOC_STREAMOFF, int *type);    //停止视频采集
```

type 其实一个 enum v4l2_buf_type *指针,通常用法如下:

```
enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (0 > ioctl(fd, VIDIOC_STREAMON, &type)) {
    perror("ioctl error");
    return -1;
}
```

25.2.7 读取数据、对数据进行处理

开启视频采集之后,接着便可以去读取数据了,前面我们已经说过,直接读取每一个帧缓冲的在用户空间的映射区即可读取到摄像头采集的每一帧图像数据。在读取数据之前,需要将帧缓冲从内核的帧缓冲队列中取出,这个操作叫做帧缓冲出队(有入队自然就有出队),前面已经给大家详细地介绍了这些理论知识。

使用 VIDIOC_DQBUF 指令执行出队操作,使用方式如下:

```
ioctl(int fd, VIDIOC_DQBUF, struct v4l2_buffer *buf);
```

帧缓冲出队之后,接下来便可读取数据了,然后对数据进行处理,譬如将摄像头采集的图像显示到 LCD 屏上;数据处理完成之后,再将帧缓冲入队,将队列中的下一个帧缓冲出队,然后读取数据、处理,这样往复操作。

使用示例如下:

```
struct v4l2_buffer buf;

buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
for (;) {

    for(buf.index = 0; buf.index < 3; buf.index++) {

        ioctl(fd, VIDIOC_DQBUF, &buf);    //出队

        // 读取帧缓冲的映射区、获取一帧数据
        // 处理这一帧数据
        do_something();

        // 数据处理完之后、将当前帧缓冲入队、接着读取下一帧数据
        ioctl(fd, VIDIOC_QBUF, &buf);
```

```
}  
}
```

25.2.8 结束视频采集

如果要结束视频采集, 使用 `VIDIOC_STREAMOFF` 指令, 用法前面已经介绍了。使用示例如下所示:

```
enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
```

```
if (0 > ioctl(fd, VIDIOC_STREAMOFF, &type)) {  
    perror("ioctl error");  
    return -1;  
}
```

25.3 V4L2 摄像头应用编程实战

通过前面的介绍, 我们已经知道如何对摄像头进行应用编程了, 摄像头的应用编程其实并不难, 基本都是按照那样的一套流程下来即可: 打开设备、查询设备、设置格式、申请帧缓冲、内存映射、入队、开启视频采集、出队、对采集到的数据进行处理, 虽然步骤很多, 但是这些操作步骤都是容易理解的, 他并没有让你感觉到很难理解这个步骤, 每一个步骤基本都是通过 `ioctl()` 来实现, 搭配不同请求指令。

本小节我们来编写摄像头应用程序, 笔者希望大家能够自己去独立完成, 通过前面的介绍, 相信大家是能够独立完成的, 可以适当地参考下面笔者提供的示例代码:

本例程源码对应的路径为: [开发板光盘 ->11](#)、[Linux C 应用编程例程源码 ->25_v4l2_camera->v4l2_camera.c](#)。

示例代码 25.3.1 V4L2 摄像头应用编程示例代码

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : v4l2_camera.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

```
描述 : V4L2 摄像头应用编程实战
```

```
其他 : 无
```

```
论坛 : www.openedv.com
```

```
日志 : 初版 V1.0 2021/7/09 邓涛创建
```

```
*****/
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <sys/ioctl.h>  
#include <string.h>  
#include <errno.h>  
#include <sys/mman.h>
```

```
#include <linux/videodev2.h>
#include <linux/fb.h>

#define FB_DEV          "/dev/fb0"          //LCD 设备节点
#define FRAMEBUFFER_COUNT 3                //帧缓冲数量

/**/ 摄像头像素格式及其描述信息 ***/
typedef struct camera_format {
    unsigned char description[32]; //字符串描述信息
    unsigned int pixelformat;      //像素格式
} cam_fmt;

/**/ 描述一个帧缓冲的信息 ***/
typedef struct cam_buf_info {
    unsigned short *start;        //帧缓冲起始地址
    unsigned long length;         //帧缓冲长度
} cam_buf_info;

static int width;                //LCD 宽度
static int height;               //LCD 高度
static unsigned short *screen_base = NULL; //LCD 显存基地址
static int fb_fd = -1;           //LCD 设备文件描述符
static int v4l2_fd = -1;         //摄像头设备文件描述符
static cam_buf_info buf_infos[FRAMEBUFFER_COUNT];
static cam_fmt cam_fmts[10];
static int frm_width, frm_height; //视频帧宽度和高度

static int fb_dev_init(void)
{
    struct fb_var_screeninfo fb_var = {0};
    struct fb_fix_screeninfo fb_fix = {0};
    unsigned long screen_size;

    /* 打开 framebuffer 设备 */
    fb_fd = open(FB_DEV, O_RDWR);
    if (0 > fb_fd) {
        fprintf(stderr, "open error: %s: %s\n", FB_DEV, strerror(errno));
        return -1;
    }

    /* 获取 framebuffer 设备信息 */
    ioctl(fb_fd, FBIOGET_VSCREENINFO, &fb_var);
    ioctl(fb_fd, FBIOGET_FSCREENINFO, &fb_fix);
```

```
screen_size = fb_fix.line_length * fb_var.yres;
width = fb_var.xres;
height = fb_var.yres;

/* 内存映射 */
screen_base = mmap(NULL, screen_size, PROT_READ | PROT_WRITE, MAP_SHARED, fb_fd, 0);
if (MAP_FAILED == (void *)screen_base) {
    perror("mmap error");
    close(fb_fd);
    return -1;
}

/* LCD 背景刷白 */
memset(screen_base, 0xFF, screen_size);
return 0;
}

static int v4l2_dev_init(const char *device)
{
    struct v4l2_capability cap = {0};

    /* 打开摄像头 */
    v4l2_fd = open(device, O_RDWR);
    if (0 > v4l2_fd) {
        fprintf(stderr, "open error: %s: %s\n", device, strerror(errno));
        return -1;
    }

    /* 查询设备功能 */
    ioctl(v4l2_fd, VIDIOC_QUERYCAP, &cap);

    /* 判断是否是视频采集设备 */
    if (!(V4L2_CAP_VIDEO_CAPTURE & cap.capabilities)) {
        fprintf(stderr, "Error: %s: No capture video device!\n", device);
        close(v4l2_fd);
        return -1;
    }

    return 0;
}

static void v4l2_enum_formats(void)
```



```
{
    struct v4l2_fmtdesc fmtdesc = {0};

    /* 枚举摄像头所支持的所有像素格式以及描述信息 */
    fmtdesc.index = 0;
    fmtdesc.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    while (0 == ioctl(v4l2_fd, VIDIOC_ENUM_FMT, &fmtdesc)) {

        // 将枚举出来的格式以及描述信息存放在数组中
        cam_fmts[fmtdesc.index].pixelformat = fmtdesc.pixelformat;
        strcpy(cam_fmts[fmtdesc.index].description, fmtdesc.description);
        fmtdesc.index++;
    }
}

static void v4l2_print_formats(void)
{
    struct v4l2_frmsizeenum frmsize = {0};
    struct v4l2_fmtdesc fmt = {0};
    int i;

    frmsize.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    for (i = 0; cam_fmts[i].pixelformat; i++) {

        printf("format<0x%x>, description<%s>\n", cam_fmts[i].pixelformat,
              cam_fmts[i].description);

        /* 枚举出摄像头所支持的所有视频采集分辨率 */
        frmsize.index = 0;
        frmsize.pixel_format = cam_fmts[i].pixelformat;
        frmsize.pixel_format = cam_fmts[i].pixelformat;
        while (0 == ioctl(v4l2_fd, VIDIOC_ENUM_FRAMESIZES, &frmsize)) {

            printf("size<%d*%d> ",
                  frmsize.discrete.width,
                  frmsize.discrete.height);
            frmsize.index++;

            /* 获取摄像头视频采集帧率 */
            frmsize.index = 0;
            frmsize.width = frmsize.discrete.width;
            frmsize.height = frmsize.discrete.height;
```

```

        while (0 == ioctl(v4l2_fd, VIDIOC_ENUM_FRAMEINTERVALS, &frmival)) {

            printf("<%dfps>", frmival.discrete.denominator /
                frmival.discrete.numerator);
            frmival.index++;
        }
        printf("\n");
    }
    printf("\n");
}

static int v4l2_set_format(void)
{
    struct v4l2_format fmt = {0};
    struct v4l2_streamparm streamparm = {0};

    /* 设置帧格式 */
    fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE; //type 类型
    fmt.fmt.pix.width = width; //视频帧宽度
    fmt.fmt.pix.height = height; //视频帧高度
    fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB565; //像素格式
    if (0 > ioctl(v4l2_fd, VIDIOC_S_FMT, &fmt)) {
        fprintf(stderr, "ioctl error: VIDIOC_S_FMT: %s\n", strerror(errno));
        return -1;
    }

    /*** 判断是否已经设置为我们要求的 RGB565 像素格式
    如果没有设置成功表示该设备不支持 RGB565 像素格式 */
    if (V4L2_PIX_FMT_RGB565 != fmt.fmt.pix.pixelformat) {
        fprintf(stderr, "Error: the device does not support RGB565 format!\n");
        return -1;
    }

    frm_width = fmt.fmt.pix.width; //获取实际的帧宽度
    frm_height = fmt.fmt.pix.height; //获取实际的帧高度
    printf("视频帧大小<%d * %d>\n", frm_width, frm_height);

    /* 获取 streamparm */
    streamparm.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ioctl(v4l2_fd, VIDIOC_G_PARM, &streamparm);

    /*** 判断是否支持帧率设置 ***/

```

```
if (V4L2_CAP_TIMEPERFRAME & streamparm.parm.capture.capability) {
    streamparm.parm.capture.timeperframe.numerator = 1;
    streamparm.parm.capture.timeperframe.denominator = 30;//30fps
    if (0 > ioctl(v4l2_fd, VIDIOC_S_PARM, &streamparm)) {
        fprintf(stderr, "ioctl error: VIDIOC_S_PARM: %s\n", strerror(errno));
        return -1;
    }
}

return 0;
}

static int v4l2_init_buffer(void)
{
    struct v4l2_requestbuffers reqbuf = {0};
    struct v4l2_buffer buf = {0};

    /* 申请帧缓冲 */
    reqbuf.count = FRAMEBUFFER_COUNT;           //帧缓冲的数量
    reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    reqbuf.memory = V4L2_MEMORY_MMAP;
    if (0 > ioctl(v4l2_fd, VIDIOC_REQBUFS, &reqbuf)) {
        fprintf(stderr, "ioctl error: VIDIOC_REQBUFS: %s\n", strerror(errno));
        return -1;
    }

    /* 建立内存映射 */
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    for (buf.index = 0; buf.index < FRAMEBUFFER_COUNT; buf.index++) {

        ioctl(v4l2_fd, VIDIOC_QUERYBUF, &buf);
        buf_infos[buf.index].length = buf.length;
        buf_infos[buf.index].start = mmap(NULL, buf.length,
            PROT_READ | PROT_WRITE, MAP_SHARED,
            v4l2_fd, buf.m.offset);
        if (MAP_FAILED == buf_infos[buf.index].start) {
            perror("mmap error");
            return -1;
        }
    }
}

/* 入队 */
```

```
for (buf.index = 0; buf.index < FRAMEBUFFER_COUNT; buf.index++) {

    if (0 > ioctl(v4l2_fd, VIDIOC_QBUF, &buf)) {
        fprintf(stderr, "ioctl error: VIDIOC_QBUF: %s\n", strerror(errno));
        return -1;
    }
}

return 0;
}

static int v4l2_stream_on(void)
{
    /* 打开摄像头、摄像头开始采集数据 */
    enum v4l2_buf_type type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

    if (0 > ioctl(v4l2_fd, VIDIOC_STREAMON, &type)) {
        fprintf(stderr, "ioctl error: VIDIOC_STREAMON: %s\n", strerror(errno));
        return -1;
    }

    return 0;
}

static void v4l2_read_data(void)
{
    struct v4l2_buffer buf = {0};
    unsigned short *base;
    unsigned short *start;
    int min_w, min_h;
    int j;

    if (width > frm_width)
        min_w = frm_width;
    else
        min_w = width;
    if (height > frm_height)
        min_h = frm_height;
    else
        min_h = height;

    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
```

```
for (;;) {

    for(buf.index = 0; buf.index < FRAMEBUFFER_COUNT; buf.index++) {

        ioctl(v4l2_fd, VIDIOC_DQBUF, &buf);    //出队
        for (j = 0, base=screen_base, start=buf_infos[buf.index].start;
             j < min_h; j++) {

            memcpy(base, start, min_w * 2); //RGB565 一个像素占 2 个字节
            base += width; //LCD 显示指向下一行
            start += frm_width; //指向下一行数据
        }

        // 数据处理完之后、再入队、往复
        ioctl(v4l2_fd, VIDIOC_QBUF, &buf);
    }
}

int main(int argc, char *argv[])
{
    if (2 != argc) {
        fprintf(stderr, "Usage: %s <video_dev>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 初始化 LCD */
    if (fb_dev_init())
        exit(EXIT_FAILURE);

    /* 初始化摄像头 */
    if (v4l2_dev_init(argv[1]))
        exit(EXIT_FAILURE);

    /* 枚举所有格式并打印摄像头支持的分辨率及帧率 */
    v4l2_enum_formats();
    v4l2_print_formats();

    /* 设置格式 */
    if (v4l2_set_format())
        exit(EXIT_FAILURE);

    /* 初始化帧缓冲: 申请、内存映射、入队 */
}
```

```

if (v4l2_init_buffer())
    exit(EXIT_FAILURE);

/* 开启视频采集 */
if (v4l2_stream_on())
    exit(EXIT_FAILURE);

/* 读取数据: 出队 */
v4l2_read_data();          //在函数内循环采集数据、将其显示到 LCD 屏

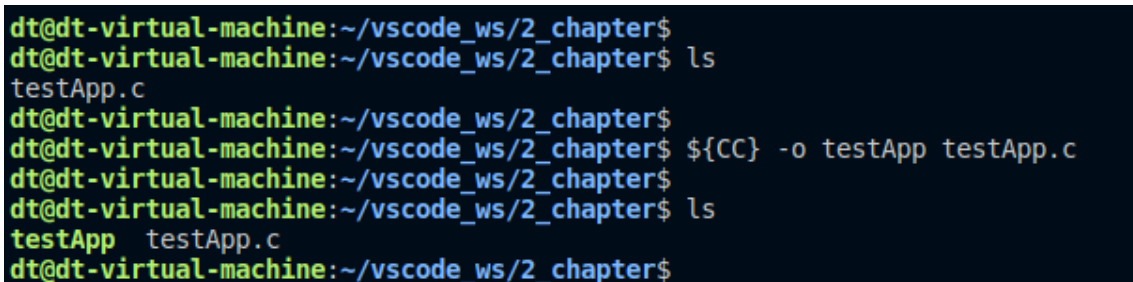
exit(EXIT_SUCCESS);
}

```

上述示例代码中, 会将摄像头采集到的图像数据显示到开发板 LCD 屏上, 我们将摄像头的像素格式设置为 RGB565, 因为这样比较好处理。其它的代码就不给大家介绍了, 没什么可说的, 代码中的注释信息已经描述得很清楚了, 这要是讲视频还可以给扯一扯, 文本形式的话, 有些东西不是那么好描述!

开发板出厂系统支持正点原子的 ov5640、ov7725 (无 FIFO) 以及 ov2640 这几款摄像头, 这几款摄像头都支持 RGB565 像素格式; 当然除此之外, 还可以板子上使用 UVC USB 摄像头, 如果大家身边有这种摄像头, 也可以进行测试, 但是这种 USB 摄像头通常不支持 RGB565 格式, 而更多是 YUYV 格式, 上述代码并不支持 YUYV 格式的处理, 需要大家进行修改, 你得将采集到的 YUYV 数据转为 RGB565 数据, 才能在 LCD 上显示采集到的图像。

接下来编译示例代码:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 25.3.1 编译摄像头应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统的用户家目录下:



```

root@ATK-IMX6U:~# pwd
/home/root
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#

```

图 25.3.2 将可执行文件拷贝到开发板

首先在测试之前, 我们的开发板上得插上一个摄像头, 这里需要注意一下, 前面我们提到开发板出厂系统支持 ov5640、ov7725 以及 ov2640, 这三款摄像头, 但是不能同时生效, 出厂系统默认配置使能的是 ov5640, 如果要使用 ov7725 或 ov2640, 则需修改设备树, 具体如何修改请大家参考“[开发板光盘资料 A-基础资料/【正点原子】I.MX6U 用户快速体验 V1.7.3.pdf](#)”文档中的 3.16 小节。

这里笔者以 ov2640 摄像头为例, 笔者的测试板上已经连接了 ov2640 摄像头, 如下所示:

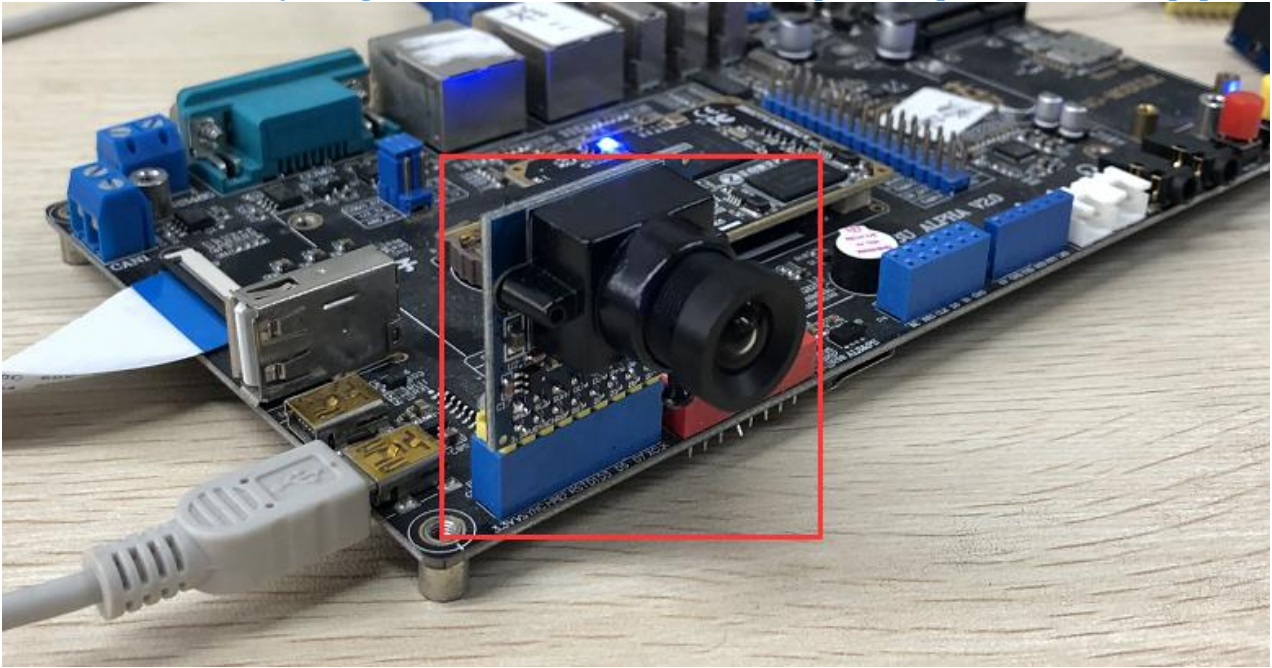


图 25.3.3 摄像头安装方式

其它摄像头的安装方式也是如此，头部朝外，注意一定是在启动之前就安装好了、而不是开发板启动之后再安装，切记！如果是 USB 摄像头，则可在开发板运行状态下，直接将 USB 摄像头插入到开发板上的 USB HOST 接口即可。

接着运行测试程序，我们需要传入一个参数，该参数表示摄像头的对应的设备节点：

```
root@ATK-IMX6U:~# ls
driver  shell  testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp /dev/video1
format<0x56595559>, description<YUYV-16>

format<0x59565955>, description<UYVY-16>

format<0x50424752>, description<RGB565_BE>

format<0x50424752>, description<RGB565_LE>

视频帧大小<800 * 600>
```

图 25.3.4 执行摄像头测试程序

程序运行之后，此时开发板 LCD 屏上将会显示摄像头所采集到的图像，如下所示：



图 25.3.5 LCD 显示摄像头采集的图像

请大家忽略手机拍摄的问题!

原本运行程序之后, 终端会打印出摄像头所支持的像素格式、描述信息以及摄像头所支持的采集分辨率、帧率等信息, 但是从图 25.3.4 中打印信息可知, 程序运行之后只打印了像素格式以及描述信息, 并没有打印分辨率和帧率等信息, 为什么呢? 当然这个不是我们的程序有问题, 而是摄像头的驱动功能不够完善, 底层驱动并没有去实现这些相关的功能, 这里给大家简单地提一下, 免得大家以为程序有问题! 这里笔者换了一个 USB 摄像头, 给大家看下它的打印信息, 如下所示:

```
root@ATK-IMX6U:~# ./testApp /dev/video2
format<0x47504a4d>, description<MJPEG>
size<1280*720> <30fps>
size<160*120> <30fps>
size<176*144> <30fps>
size<320*240> <30fps>
size<352*288> <30fps>
size<424*240> <30fps>
size<640*360> <30fps>
size<960*540> <30fps>
size<1280*720> <30fps>
size<640*480> <30fps>

format<0x56595559>, description<YUV 4:2:2 (YUYV)>
size<640*480> <30fps>
size<160*120> <30fps>
size<176*144> <30fps>
size<320*240> <30fps>
size<352*288> <30fps>
size<424*240> <30fps>
size<640*360> <30fps>
size<960*540> <10fps>
size<1280*720> <10fps>

Error: the device does not support RGB565 format!
root@ATK-IMX6U:~#
```

图 25.3.6 USB 摄像头打印信息

从上图可以看到, 程序打印了摄像头所支持的所有采集分辨率大小以及帧率。

好了, 本章的内容到此结束了, 到此为止, 我们已经学习了很多硬件外设的应用编程知识了, 大家要学会活学活用, 把这些东西用起来, 尝试着做一个综合类的好玩的小项目, 提高自己的应用编程能力, 笔者觉

得这是非常重要,你不要跟着笔者的教程一个一个章节往下走,你得停下来思考、多动动手、在教程的基础上多往外扩展,这样你才能进步!大家加油!

第二十六章 串口应用编程

本小节我们来学习 Linux 下串口应用编程, 串口 (UART) 是一种非常常见的外设, 串口在嵌入式开发领域当中一般作为一种调试手段, 通过串口输出调试打印信息, 或者通过串口发送指令给主机端进行处理; 当然除了作为基本的调试手段之外, 还可以通过串口与其他设备或传感器进行通信, 譬如有些 sensor 就使用了串口通信的方式与主机端进行数据交互。

本章将会讨论如下主题内容。

- 串口应用编程介绍
- 应用编程实战

26.1 串口应用编程介绍

串口全称叫做串行接口，串行接口指的是数据一个一个的按顺序传输，通信线路简单。使用两条线即可实现双向通信，一条用于发送，一条用于接收。串口通信距离远，但是速度相对会低，串口是一种很常用的工业接口。

关于串口的基础知识以及通行原理、通行数据格式等之类的问题，笔者就不给大家介绍了，免得大家嫌啰嗦。串口（UART）在嵌入式 Linux 系统中常作为系统的标准输入、输出设备，系统运行过程产生的打印信息通过串口输出；同理，串口也作为系统的标准输入设备，用户通过串口与 Linux 系统进行交互。

所以串口在 Linux 系统就是一个终端，提到串口，就不得不引出“终端（Terminal）”这个概念了。

26.1.1 终端 Terminal

终端就是处理主机输入、输出的一套设备，它用来显示主机运算的输出，并且接受主机要求的输入。典型的终端包括显示器键盘套件，打印机打字机套件等。其实本质上也就一句话，能接受输入、能显示输出，这就够了，不管到了什么时代，终端始终扮演着人机交互的角色，所谓 Terminal，即机器的边缘！

只要能提供给计算机输入和输出功能，它就是终端，而与其所在的位置无关。

终端的分类

- **本地终端**：例如对于我们的个人 PC 机来说，PC 机连接了显示器、键盘以及鼠标等设备，这样的显示器/键盘组合就是一个本地终端；同样对于开发板来说也是如此，开发板也可以连接一个 LCD 显示器、键盘和鼠标等，同样可以构成本地终端。
- **用串口连接的远程终端**：对于嵌入式 Linux 开发来说，这是最常见的终端—串口终端。譬如我们的开发板通过串口线连接到一个带有显示器和键盘的 PC 机，在 PC 机通过运行一个终端模拟程序，譬如 Windows 超级终端、putty、MobaXterm、SecureCRT 等来获取并显示开发板通过串口发出的数据、同样还可以通过这些终端模拟程序将用户数据通过串口发送给开发板 Linux 系统，系统接收到数据之后便会进行相应的处理、譬如执行某个命令，这就是一种人机交互！
- **基于网络的远程终端**：譬如我们可以通过 ssh、Telnet 这些协议登录到一个远程主机。

以上列举的这些都是终端，前两类称之为物理终端；最后一个称之为伪终端。前两类都是在本地就直接关联了物理设备的，譬如显示器、鼠标键盘、串口等之类的，这种终端叫做物理终端，而第三类在本地则没有关联任何物理设备，注意，不要把物理网卡当成终端关联的物理设备，它们与终端并不直接相关，所以这类不直接关联物理设备的终端叫做伪终端。

终端对应的设备节点

在 Linux 当中，一切皆是文件。当然，终端也不例外，每一个终端在/dev 目录下都有一个对应的设备节点。

- /dev/ttyX（X 是一个数字编号，譬如 0、1、2、3 等）设备节点：ttyX（teletype 的简称）是最令人熟悉的了，在 Linux 中，/dev/ttyX 代表的都是上述提到的本地终端，包括/dev/tty1~dev/tty63 一共 63 个本地终端，也就是连接到本机的键盘显示器可以操作的终端。事实上，这是 Linux 内核在初始化时所生成的 63 个本地终端。如下所示：

```
root@ATK-IMX6U:~# ls /dev/tty*
/dev/tty /dev/tty12 /dev/tty17 /dev/tty21 /dev/tty26 /dev/tty30 /dev/tty35 /dev/tty4 /dev/tty44 /dev/tty49 /dev/tty53 /dev/tty58 /dev/tty62
/dev/tty0 /dev/tty13 /dev/tty18 /dev/tty22 /dev/tty27 /dev/tty31 /dev/tty36 /dev/tty40 /dev/tty45 /dev/tty5 /dev/tty54 /dev/tty59 /dev/tty63
/dev/tty1 /dev/tty14 /dev/tty19 /dev/tty23 /dev/tty28 /dev/tty32 /dev/tty37 /dev/tty41 /dev/tty46 /dev/tty50 /dev/tty55 /dev/tty6 /dev/tty7
/dev/tty10 /dev/tty15 /dev/tty2 /dev/tty24 /dev/tty29 /dev/tty33 /dev/tty38 /dev/tty42 /dev/tty47 /dev/tty51 /dev/tty56 /dev/tty60 /dev/tty8
/dev/tty11 /dev/tty16 /dev/tty20 /dev/tty25 /dev/tty3 /dev/tty34 /dev/tty39 /dev/tty43 /dev/tty48 /dev/tty52 /dev/tty57 /dev/tty61 /dev/tty9
root@ATK-IMX6U:~#
```

图 26.1.1 本地终端设备节点

- /dev/pts/X（X 是一个数字编号，譬如 0、1、2、3 等）设备节点：这类设备节点是伪终端对应的设备节点，也就是说，伪终端对应的设备节点都在/dev/pts 目录下、以数字编号命名。譬如我们通过

ssh 或 Telnet 这些远程登录协议登录到开发板主机, 那么开发板 Linux 系统会在 /dev/pts 目录下生成一个设备节点, 这个设备节点便对应伪终端, 如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls /dev/pts/*
/dev/pts/0
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 26.1.2 伪终端设备节点

- 串口终端设备节点/dev/ttymxcX: 对于 ALPHA/Mini I.MX6U 开发板来说, 有两个串口, 也就是有两个串口终端, 对应两个设备节点, 如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls /dev/ttymxc*
/dev/ttymxc0 /dev/ttymxc2
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 26.1.3 串口终端对应的设备节点

这里为什么是 0 和 2、而不是 0 和 1? 我们知道, I.MX6U SoC 支持 8 个串口外设, 分别是 UART1~UART8; 出厂系统只注册了 2 个串口外设, 分别是 UART1 和 UART3, 所以对应这个数字就是 0 和 2、而不是 0 和 1, 这里了解一下就行!

还需要注意的是, mxc 这个名字不是一定的, 这个名字的命名与驱动有关系 (与硬件平台有关), 如果你换一个硬件平台, 那么它这个串口对应的设备节点就不一定是 mxcX 了; 譬如 ZYNQ 平台, 它的系统中串口对应的设备节点就是 /dev/ttyPSX (X 是一个数字编号), 所以说这个名字它不是统一的, 但是名字前缀都是以 “tty” 开头, 以表明它是一个终端。

在 Linux 系统下, 我们可以使用 who 命令来查看计算机系统当前连接了哪些终端 (一个终端就表示有一个用户使用该计算机), 如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# who
root      ttymxc0      2021-06-23 20:43
root      pts/0        2021-06-23 22:24 (192.168.1.209)
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 26.1.4 查看系统连接了哪些终端

可以看到, 开发板系统当前有两个终端连接到它, 一个就是我们的串口终端, 也就是开发板的 USB 调试串口 (对应/dev/ttymxc0); 另一个则是伪终端, 这是笔者通过 ssh 连接的。

26.1.2 串口应用编程

现在我们已经知道了串口在 Linux 系统中是一种终端设备, 并且在我们的开发板上, 其设备节点为 /dev/ttymxc0 (UART1) 和 /dev/ttymxc2 (UART3)。

其实串口的应用编程也很简单, 无非就是通过 ioctl() 对串口进行配置, 调用 read() 读取串口的数据、调用 write() 向串口写入数据, 是的, 就是这么简单! 但是我们不这么做, 因为 Linux 为上层用户做了一层封装, 将这些 ioctl() 操作封装成了一套标准的 API, 我们就直接使用这一套标准 API 编写自己的串口应用程序即可!

笔者把这一套接口称为 termios API, 这些 API 其实是 C 库函数, 可以使用 man 手册查看到它们的帮助信息; 这里需要注意的是, 这一套接口并不是针对串口开发的, 而是针对所有的终端设备, 串口是一种终端设备, 计算机系统本地连接的鼠标、键盘也是终端设备, 通过 ssh 远程登录连接的伪终端也是终端设备。

要使用 termios API, 需要在我们的应用程序中包含 termios.h 头文件。

26.1.3 struct termios 结构体

对于终端来说, 其应用编程内容无非包括两个方面的内容: 配置和读写; 对于配置来说, 一个很重要的数据结构便是 struct termios 结构体, 该数据结构描述了终端的配置信息, 这些参数能够控制、影响终端的行为、特性, 事实上, 终端设备应用编程 (串口应用编程) 主要就是对这个结构体进行配置。

struct termios 结构体定义如下:

示例代码 26.1.1 struct termios 结构体

```
struct termios
{
    tcflag_t c_iflag;    /* input mode flags */
    tcflag_t c_oflag;    /* output mode flags */
    tcflag_t c_cflag;    /* control mode flags */
    tcflag_t c_lflag;    /* local mode flags */
    cc_t c_line;        /* line discipline */
    cc_t c_cc[NCCS];    /* control characters */
    speed_t c_ispeed;    /* input speed */
    speed_t c_ospeed;    /* output speed */
};
```

如上定义所示, 影响终端的参数按照不同模式分为如下几类:

- 输入模式;
- 输出模式;
- 控制模式;
- 本地模式;
- 线路规程;
- 特殊控制字符;
- 输入速率;
- 输出速率。

接下来, 简单地给大家介绍一下如何去配置这些参数、它们分别表示什么意思。

一、输入模式: c_iflag

输入模式控制输入数据 (终端驱动程序从串口或键盘接收到的字符数据) 在被传递给应用程序之前的处理方式。通过设置 struct termios 结构体中 c_iflag 成员的标志对它们进行控制。所有的标志都被定义为宏, 除 c_iflag 成员外, c_oflag、c_cflag 以及 c_lflag 成员也都采用这种方式进行配置。

可用于 c_iflag 成员的宏如下所示:

IGNBRK	忽略输入终止条件
BRKINT	当检测到输入终止条件时发送 SIGINT 信号
IGNPAR	忽略帧错误和奇偶校验错误
PARMRK	对奇偶校验错误做出标记
INPCK	对接收到的数据执行奇偶校验
ISTRIP	将所有接收到的数据裁剪为 7 比特位、也就是去除第八位
INLCR	将接收到的 NL (换行符) 转换为 CR (回车符)
IGNCR	忽略接收到的 CR (回车符)

ICRNL	将接收到的 CR (回车符) 转换为 NL (换行符)
IUCLC	将接收到的大写字符映射为小写字符
IXON	启动输出软件流控
IXOFF	启动输入软件流控

表 26.1.1 用于 c_iflag 成员的标志

以上所列举出的这些宏, 我们可以通过 man 手册查询到它们的详细描述信息, 执行命令" man 3 termios ", 如下图所示:

```

c_iflag flag constants:

IGNBRK Ignore BREAK condition on input.

BRKINT If IGNBRK is set, a BREAK is ignored. If it is not set but BRKINT is set, then a BREAK causes the input and
controlling terminal of a foreground process group, it will cause a SIGINT to be sent to this foreground p
BREAK reads as a null byte ('\0'), except when PARMRK is set, in which case it reads as the sequence \377 \0

IGNPAR Ignore framing errors and parity errors.

PARMRK If this bit is set, input bytes with parity or framing errors are marked when passed to the program. This b
not set. The way erroneous bytes are marked is with two preceding bytes, \377 and \0. Thus, the program ac
from the terminal. If a valid byte has the value \377, and ISTRIP (see below) is not set, the program might
Therefore, a valid byte \377 is passed to the program as two bytes, \377 \377, in this case.

If neither IGNPAR nor PARMRK is set, read a character with a parity error or framing error as \0.

INPCK Enable input parity checking.

ISTRIP Strip off eighth bit.

INLCR Translate NL to CR on input.

IGNCR Ignore carriage return on input.

ICRNL Translate carriage return to newline on input (unless IGNCR is set).

IUCLC (not in POSIX) Map uppercase characters to lowercase on input.
    
```

图 26.1.5 通过 man 手册查询

二、输出模式: c_oflag

输出模式控制输出字符的处理方式, 即由应用程序发送出去的字符数据在传递到串口或屏幕之前是如何处理的。可用于 c_oflag 成员的宏如下所示:

OPOST	启用输出处理功能, 如果不设置该标志则其他标志都被忽略
OLCUC	将输出字符中的大写字符转换成小写字符
ONLCR	将输出中的换行符 (NL '\n') 转换成回车符 (CR '\r')
OCRNL	将输出中的回车符 (CR '\r') 转换成换行符 (NL '\n')
ONOCR	在第 0 列不输出回车符 (CR)
ONLRET	不输出回车符
OFILL	发送填充字符以提供延时
OFDEL	如果设置该标志, 则表示填充字符为 DEL 字符, 否则为 NULL 字符

表 26.1.2 用于 c_oflag 成员的标志

三、控制模式: c_cflag

控制模式控制终端设备的硬件特性, 譬如对于串口来说, 该字段比较重要, 可设置串口波特率、数据位、校验位、停止位等硬件特性。通过设置 struct termios 结构中 c_cflag 成员的标志对控制模式进行配置。可用于 c_cflag 成员的标志如下所示:

CBAUD	波特率的位掩码
B0	波特率为 0

.....
B1200	1200 波特率
B1800	1800 波特率
B2400	2400 波特率
B4800	4800 波特率
B9600	9600 波特率
B19200	19200 波特率
B38400	38400 波特率
B57600	57600 波特率
B115200	115200 波特率
B230400	230400 波特率
B460800	460800 波特率
B500000	500000 波特率
B576000	576000 波特率
B921600	921600 波特率
B1000000	1000000 波特率
B1152000	1152000 波特率
B1500000	1500000 波特率
B2000000	2000000 波特率
B2500000	2500000 波特率
B3000000	3000000 波特率
.....
CSIZE	数据位的位掩码
CS5	5 个数据位
CS6	6 个数据位
CS7	7 个数据位
CS8	8 个数据位
CSTOPB	2 个停止位, 如果不设置该标志则默认是一个停止位
CREAD	接收使能
PARENB	使能奇偶校验
PARODD	使用奇校验、而不是偶校验
HUPCL	关闭时挂断调制解调器
CLOCAL	忽略调制解调器控制线
CRTSCTS	使能硬件流控

表 26.1.3 用于 c_cflag 成员的标志

在 struct termios 结构体中, 有一个 c_ispeed 成员变量和 c_ospeed 成员变量, 在其它一些系统中, 可能会使用这两个变量来指定串口的波特率; 在 Linux 系统下, 则是使用 CBAUD 位掩码所选择的几个 bit 位来指定串口波特率。事实上, termios API 中提供了 cfgetispeed()和 cfsetispeed()函数分别用于获取和设置串口的波特率。

四、本地模式: c_lflag

本地模式用于控制终端的本地数据处理和工作模式。通过设置 `struct termios` 结构体中 `c_lflag` 成员的标志对本地模式进行配置。可用于 `c_lflag` 成员的标志如下所示:

ISIG	若收到信号字符 (INTR、QUIT 等), 则会产生相应的信号
ICANON	启用规范模式
ECHO	启用输入字符的本地回显功能。当我们在终端输入字符的时候, 字符会显示出来, 这就是回显功能
ECHOE	若设置 ICANON, 则允许退格操作
ECHOK	若设置 ICANON, 则 KILL 字符会删除当前行
ECHONL	若设置 ICANON, 则允许回显换行符
ECHOCTL	若设置 ECHO, 则控制字符 (制表符、换行符等) 会显示成 “^X”, 其中 X 的 ASCII 码等于给相应控制字符的 ASCII 码加上 0x40。例如, 退格字符 (0x08) 会显示为 “^H” (‘H’的 ASCII 码为 0x48)
ECHOPRT	若设置 ICANON 和 IECHO, 则删除字符 (退格符等) 和被删除的字符都会被显示
ECHOKE	若设置 ICANON, 则允许回显在 ECHOE 和 ECHOPRT 中设定的 KILL 字符
NOFLSH	在通常情况下, 当接收到 INTR、QUIT 和 SUSP 控制字符时, 会清空输入和输出队列。如果设置该标志, 则所有的队列不会被清空
TOSTOP	若一个后台进程试图向它的控制终端进行写操作, 则系统向该后台进程的进程组发送 SIGTTOU 信号。该信号通常终止进程的执行
IEXTEN	启用输入处理功能

表 26.1.4 用于 `c_lflag` 成员的标志

五、特殊控制字符: `c_cc`

特殊控制字符是一些字符组合, 如 `Ctrl+C`、`Ctrl+Z` 等, 当用户键入这样的组合键, 终端会采取特殊处理方式。`struct termios` 结构体中 `c_cc` 数组将各种特殊字符映射到对应的支持函数。每个字符位置 (数组下标) 由对应的宏定义的, 如下所示

- VEOF: 文件结尾符 EOF, 对应键为 `Ctrl+D`; 该字符使终端驱动程序将输入行中的全部字符传递给正在读取输入的应用程序。如果文件结尾符是该行的第一个字符, 则用户程序中的 `read` 返回 0, 表示文件结束。
- VEOL: 附加行结尾符 EOL, 对应键为 Carriage return (CR); 作用类似于行结束符。
- VEOL2: 第二行结尾符 EOL2, 对应键为 Line feed (LF);
- VERASE: 删除操作符 ERASE, 对应键为 Backspace (BS); 该字符使终端驱动程序删除输入行中的最后一个字符;
- VINTR: 中断控制字符 INTR, 对应键为 `Ctrl+C`; 该字符使终端驱动程序向与终端相连的进程发送 SIGINT 信号;
- VKILL: 删除行符 KILL, 对应键为 `Ctrl+U`, 该字符使终端驱动程序删除整个输入行;
- VMIN: 在非规范模式下, 指定最少读取的字符数 MIN;
- VQUIT: 退出操作符 QUIT, 对应键为 `Ctrl+Z`; 该字符使终端驱动程序向与终端相连的进程发送 SIGQUIT 信号。
- VSTART: 开始字符 START, 对应键为 `Ctrl+Q`; 重新启动被 STOP 暂停的输出。
- VSTOP: 停止字符 STOP, 对应键为 `Ctrl+S`; 字符作用“截流”, 即阻止向终端的进一步输出。用于支持 XON/XOFF 流控。

- **VSUSP**: 挂起字符 **SUSP**, 对应键为 **Ctrl+Z**; 该字符使终端驱动程序向与终端相连的进程发送 **SIGSUSP** 信号, 用于挂起当前应用程序。
- **VTIME**: 非规范模式下, 指定读取的每个字符之间的超时时间 (以分秒为单位) **TIME**。

在以上所列举的这些宏定义中, **TIME** 和 **MIN** 值只能用于非规范模式, 可用于控制非规范模式下 **read()** 调用的一些行为特性, 后面再向大家介绍。

六、总结说明

上面已经给大家介绍了 **struct termios** 结构体中 **c_iflag** 成员 (输入模式)、**c_oflag** 成员 (输出模式)、**c_cflag** 成员 (控制模式) 以及 **c_lflag** 成员 (本地控制) 这四个参数, 这些参数能够分别控制、影响终端的行为特性。

这里有两个问题需要向大家说明, 首先第一个是关于这些成员变量赋值的问题。

对于这些变量尽量不要直接对其初始化, 而要将其通过“按位与”、“按位或”等操作添加标志或清除某个标志。譬如, 通常不会这样对变量进行初始化:

```
struct termios ter;
```

```
ter.c_iflag = IGNBRK | BRKINT | PARMRK;
```

而是要像下面这样:

```
ter.c_iflag |= (IGNBRK | BRKINT | PARMRK | ISTRIP);
```

说完第一个问题之后, 我们来看看第二个问题。

前面我们介绍了很多的标志, 但是并非所有标志对于实际的终端设备来说都是有效的, 就拿串口终端来说, 串口可以配置波特率、数据位、停止位等这些硬件参数, 但是其它终端是不一定支持这些配置的, 譬如本地终端键盘、显示器, 这些设备它是没有这些硬件概念的。

因为这些终端设备都使用了这一套 **API** 来编程, 然而不同的终端设备, 本身硬件上就存在很大的区别, 所以会导致这些配置参数并不是对所有终端设备都是有效的。在使用过程中也不需要去搞懂所有标志的作用, 事实上, 快速掌握一项技术的核心点才是一种学习能力!

26.1.4 终端的三种工作模式

当 **ICANON** 标志被设置时表示启用终端的规范模式, 什么规范模式? 这里给大家简单地说明一下。

终端有三种工作模式, 分别为规范模式 (**canonical mode**)、非规范模式 (**non-canonical mode**) 和原始模式 (**raw mode**)。通过在 **struct termios** 结构体的 **c_lflag** 成员中设置 **ICANON** 标志来定义终端是以规范模式 (设置 **ICANON** 标志) 还是以非规范模式 (清除 **ICANON** 标志) 工作, 默认情况为规范模式。

在规范模式下, 所有的输入是基于行进行处理的。在用户输入一个行结束符 (回车符、**EOF** 等) 之前, 系统调用 **read()** 函数是读不到用户输入的任何字符的。除了 **EOF** 之外的行结束符 (回车符等) 与普通字符一样会被 **read()** 函数读取到缓冲区中。在规范模式中, 行编辑是可行的, 而且一次 **read()** 调用最多只能读取一行数据。如果在 **read()** 函数中被请求读取的数据字节数小于当前行可读取的字节数, 则 **read()** 函数只会读取被请求的字节数, 剩下的字节下次再被读取。

在非规范模式下, 所有的输入是即时有效的, 不需要用户另外输入行结束符, 而且不可进行行编辑。在非规范模式下, 对参数 **MIN(c_cc[VMIN])** 和 **TIME(c_cc[VTIME])** 的设置决定 **read()** 函数的调用方式。

上一小节给大家提到过, **TIME** 和 **MIN** 的值只能用于非规范模式, 两者结合起来可以控制对输入数据的读取方式。根据 **TIME** 和 **MIN** 的取值不同, 会有以下 4 种不同情况:

- **MIN = 0** 和 **TIME = 0**: 在这种情况下, **read()** 调用总是会立即返回。若有可读数据, 则读取数据并返回被读取的字节数; 否则读取不到任何数据并返回 0。

- **MIN > 0 和 TIME = 0:** 在这种情况下, read()函数会被阻塞,直到有 MIN 个字符可以读取时才返回,返回值是读取的字符数量。到达文件尾时返回 0。
- **MIN = 0 和 TIME > 0:** 在这种情况下,只要有数据可读或者经过 TIME 个十分之一秒的时间,read()函数则立即返回,返回值为被读取的字节数。如果超时并且未读到数据,则 read()函数返回 0。
- **MIN > 0 和 TIME > 0:** 在这种情况下,当有 MIN 个字节可读或者两个输入字符之间的时间间隔超过 TIME 个十分之一秒时,read()函数才返回。因为在输入第一个字符后系统才会启动定时器,所以,在这种情况下,read()函数至少读取一个字节后才返回。

原始模式 (Raw mode)

按照严格意义来讲,原始模式是一种特殊的非规范模式。在原始模式下,所有的输入数据以字节为单位被处理。在这个模式下,终端是不可回显的,并且禁用终端输入和输出字符的所有特殊处理。在我们的应用程序中,可以通过调用 cfmakeraw()函数将终端设置为原始模式。

cfmakeraw()函数内部其实就是对 struct termios 结构体进行了如下配置:

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
    | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

什么时候会使用原始模式? 串口在 Linux 系统下是作为一种终端设备存在,终端通常会对用户的输入、输出数据进行相应的处理,如前所述!

但是串口并不仅仅只扮演着人机交互的角色(数据以字符的形式传输、也就数说传输的数据其实字符对应的 ASCII 编码值);串口本就是一种数据串行传输接口,通过串口可以与其他设备或传感器进行数据传输、通信,譬如很多 sensor 就使用了串口方式与主机端进行数据交互。那么在这种情况下,我们就得使用原始模式,意味着通过串口传输的数据不应进行任何特殊处理、不应将其解析成 ASCII 字符。

26.1.5 打开串口设备

好了,前面已经向大家详细地介绍了 struct termios 结构体以及终端的三种工作模式,为我们接下来的要讲解的内容打下了一个基础。从本小节开始,我们来看看如何编写串口应用程序。

首先第一步便是打开串口设备,使用 open()函数打开串口的设备节点文件,得到文件描述符:

```
int fd;

fd = open("/dev/ttymx2", O_RDWR | O_NOCTTY);
if (0 > fd) {
    perror("open error");
    return -1;
}
```

调用 open()函数时,使用了 O_NOCTTY 标志,该标志用于告知系统/dev/ttymx2 它不会成为进程的控制终端。

26.1.6 获取终端当前的配置参数: tcgetattr()函数

通常,在配置终端之前,我们会先获取到终端当前的配置参数,将其保存到一个 struct termios 结构体对象中,这样可以在之后、很方便地将终端恢复到原来的状态,这也是为了安全起见以及后续的调试方便。

tcgetattr()函数可以获取到串口终端当前的配置参数, tcgetattr 函数原型如下所示(可通过命令"man 3 tcgetattr"查询):

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(int fd, struct termios *termios_p);
```

首先在我们的应用程序中需要包含 termios.h 头文件和 unistd.h 头文件。

第一个参数对应串口终端设备的文件描述符 fd。

调用 tcgetattr 函数之前, 我们需要定义一个 struct termios 结构体变量, 将该变量的指针作为 tcgetattr() 函数的第二个参数传入; tcgetattr()调用成功后, 会将终端当前的配置参数保存到 termios_p 指针所指的對象中。

函数调用成功返回 0; 失败将返回-1, 并且会设置 errno 以告知错误原因。

使用示例如下:

```
struct termios old_cfg;

if (0 > tcgetattr(fd, &old_cfg)) {
    /* 出错处理 */
    do_something();
}
```

26.1.7 对串口终端进行配置

假设我们需要采用原始模式进行串口数据通信。

1)配置串口终端为原始模式

调用<termios.h>头文件中声明的 cfmakeraw()函数可以将终端配置为原始模式:

```
struct termios new_cfg;

memset(&new_cfg, 0x0, sizeof(struct termios));

//配置为原始模式
cfmakeraw(&new_cfg);
```

这个函数没有返回值。

2)接收使能

使能接收功能只需在 struct termios 结构体的 c_cflag 成员中添加 CREAD 标志即可, 如下所示:

```
new_cfg.c_cflag |= CREAD;    //接收使能
```

3)设置串口的波特率

设置波特率有专门的函数, 用户不能直接通过位掩码来操作。设置波特率的主要函数有 cfsetispeed()和 cfsetospeed(), 这两个函数在<termios.h>头文件中声明, 使用方法很简单, 如下所示:

```
cfsetispeed(&new_cfg, B115200);
cfsetospeed(&new_cfg, B115200);
```

B115200 是一个宏, 前面已经给大家介绍了, B115200 表示波特率为 115200。

cfsetispeed()函数设置数据输入波特率,而 cfsetospeed()函数设置数据输出波特率。一般来说,用户需将终端的输入和输出波特率设置成一样的。

除了之外,我们还可以直接使用 cfsetispeed()函数一次性设置输入和输出波特率,该函数也是在<termios.h>头文件中申明,使用方式如下:

```
cfsetispeed(&new_cfg, B115200);
```

这几个函数在成功时返回 0,失败时返回-1。

4)设置数据位大小

与设置波特率不同,设置数据位大小并没有现成可用的函数,我们需要自己通过位掩码来操作、设置数据位大小。设置方法也很简单,首先将 c_cflag 成员中 CSIZE 位掩码所选择的几个 bit 位清零,然后再设置数据位大小,如下所示:

```
new_cfg.c_cflag &= ~CSIZE;
new_cfg.c_cflag |= CS8; //设置为 8 位数据位
```

5)设置奇偶校验位

通过 26.1.3 小节的内容可知,串口的奇偶校验位配置一共涉及到 struct termios 结构体中的两个成员变量: c_cflag 和 c_iflag。首先对于 c_cflag 成员,需要添加 PARENB 标志以使能串口的奇偶校验功能,只有使能奇偶校验功能之后才会对输出数据产生校验位,而对输入数据进行校验检查;同时对于 c_iflag 成员来说,还需要添加 INPCK 标志,这样才能对接收到的数据执行奇偶校验,代码如下所示:

```
//奇校验使能
new_cfg.c_cflag |= (PARODD | PARENB);
new_cfg.c_iflag |= INPCK;

//偶校验使能
new_cfg.c_cflag |= PARENB;
new_cfg.c_cflag &= ~PARODD; /* 清除 PARODD 标志,配置为偶校验 */
new_cfg.c_iflag |= INPCK;

//无校验
new_cfg.c_cflag &= ~PARENB;
new_cfg.c_iflag &= ~INPCK;
```

6)设置停止位

停止位则是通过设置 c_cflag 成员的 CSTOPB 标志而实现的。若停止位为一个比特,则清除 CSTOPB 标志;若停止位为两个,则添加 CSTOPB 标志即可。以下分别是停止位为一个和两个比特时的代码:

```
// 将停止位设置为一个比特
new_cfg.c_cflag &= ~CSTOPB;

// 将停止位设置为 2 个比特
new_cfg.c_cflag |= CSTOPB;
```

7)设置 MIN 和 TIME 的值

如前面所介绍那样,MIN 和 TIME 的取值会影响非规范模式下 read()调用的行为特征,原始模式是一种特殊的非规范模式,所以 MIN 和 TIME 在原始模式下也是有效的。

在对接收字符和等待时间没有特别要求的情况下, 可以将 MIN 和 TIME 设置为 0, 这样则在任何情况下 read()调用都会立即返回, 此时对串口的 read 操作会设置为非阻塞方式, 如下所示:

```
new_cfg.c_cc[VTIME] = 0;
new_cfg.c_cc[VMIN] = 0;
```

26.1.8 缓冲区的处理

我们在使用串口之前, 需要对串口的缓冲区进行处理, 因为在我们使用之前, 其缓冲区中可能已经存在一些数据等待处理或者当前正在进行数据传输、接收, 所以使用之前, 所以需要对此情况进行处理。这时就可以调用<termios.h>中声明的 tcdrain()、tcflow()、tcflush()等函数来处理目前串口缓冲中的数据, 它们的函数原型如下所示:

```
#include <termios.h>
#include <unistd.h>

int tcdrain(int fd);
int tcflush(int fd, int queue_selector);
int tcflow(int fd, int action);
```

调用 tcdrain()函数后会使得应用程序阻塞, 直到串口输出缓冲区中的数据全部发送完毕为止!

调用 tcflow()函数会暂停串口上的数据传输或接收工作, 具体情况取决于参数 action, 参数 action 可取值如下:

- TCOOFF: 暂停数据输出 (输出传输);
- TCOON: 重新启动暂停的输出;
- TCIOFF: 发送 STOP 字符, 停止终端设备向系统发送数据;
- TCION: 发送一个 START 字符, 启动终端设备向系统发送数据;

再来看看 tcflush()函数, 调用该函数会清空输入/输出缓冲区中的数据, 具体情况取决于参数 queue_selector, 参数 queue_selector 可取值如下:

- TCIFLUSH: 对接收到而未被读取的数据进行清空处理;
- TCOFLUSH: 对尚未传输成功的输出数据进行清空处理;
- TCIOFLUSH: 包括前两种功能, 即对尚未处理的输入/输出数据进行清空处理。

以上这三个函数, 调用成功时返回 0; 失败将返回-1、并且会设置 errno 以指示错误类型。

通常我们会选择 tcdrain()或 tcflush()函数来对串口缓冲区进行处理。譬如直接调用 tcdrain()阻塞:

```
tcdrain(fd);
// 或者调用 tcflush()清空缓冲区:
tcflush(fd, TCIOFLUSH);
```

26.1.9 写入配置、使配置生效: tcsetattr()函数

前面已经完成了对 struct termios 结构体各个成员进行配置, 但是配置还未生效, 我们需要将配置参数写入到终端设备 (串口硬件), 使其生效。通过 tcsetattr()函数将配置参数写入到硬件设备, 其函数原型如下所示:

```
#include <termios.h>
#include <unistd.h>

int tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
```

调用该函数会将参数 termios_p 所指 struct termios 对象中的配置参数写入到终端设备中, 使配置生效!

而参数 `optional_actions` 可以指定更改何时生效, 其取值如下:

- `TCSANOW`: 配置立即生效。
- `TCSADRAIN`: 配置在所有写入 `fd` 的输出都传输完毕之后生效。
- `TCSAFLUSH`: 所有已接收但未读取的输入都将在配置生效之前被丢弃。

该函数调用成功时返回 0; 失败将返回 -1, 并设置 `errno` 以指示错误类型。

譬如, 调用 `tcsetattr()` 将配置参数写入设备, 使其立即生效:

```
tcsetattr(fd, TCSANOW, &new_cfg);
```

26.1.10 读写数据: `read()`、`write()`

所有准备工作完成之后, 接着便可以读写数据了, 直接调用 `read()`、`write()` 函数即可!

26.2 串口应用编程实战

通过上一节的介绍, 详细大家已经知道了如何对串口进行应用编程, 其实总的来说还是非常简单地, 本节我们进行编程实战, 在串口终端的原始模式下, 使用串口进行数据传输, 包括通过串口发送数据、以及读取串口接收到的数据, 并将其打印出来。

示例代码笔者已经写好了, 如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->26_uart->uart_test.c](#)。

示例代码 26.2.1 串口数据传输示例代码

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : uart_test.c
作者 : 邓涛
版本 : V1.0
描述 : 串口在原始模式下进行数据传输--应用程序示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/20 邓涛创建
*****/

#define _GNU_SOURCE //在源文件开头定义_GNU_SOURCE 宏
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <termios.h>

typedef struct uart_hardware_cfg {

```

```
    unsigned int baudrate;    /* 波特率 */
    unsigned char dbit;      /* 数据位 */
    char parity;            /* 奇偶校验 */
    unsigned char sbit;     /* 停止位 */
} uart_cfg_t;

static struct termios old_cfg; //用于保存终端的配置参数
static int fd;                //串口终端对应的文件描述符

/**
** 串口初始化操作
** 参数 device 表示串口终端的设备节点
**/
static int uart_init(const char *device)
{
    /* 打开串口终端 */
    fd = open(device, O_RDWR | O_NOCTTY);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", device, strerror(errno));
        return -1;
    }

    /* 获取串口当前的配置参数 */
    if (0 > tcgetattr(fd, &old_cfg)) {
        fprintf(stderr, "tcgetattr error: %s\n", strerror(errno));
        close(fd);
        return -1;
    }

    return 0;
}

/**
** 串口配置
** 参数 cfg 指向一个 uart_cfg_t 结构体对象
**/
static int uart_cfg(const uart_cfg_t *cfg)
{
    struct termios new_cfg = {0}; //将 new_cfg 对象清零
    speed_t speed;

    /* 设置为原始模式 */
    cfmakeraw(&new_cfg);
```

```
/* 使能接收 */
new_cfg.c_cflag |= CREAD;

/* 设置波特率 */
switch (cfg->baudrate) {
case 1200: speed = B1200;
    break;
case 1800: speed = B1800;
    break;
case 2400: speed = B2400;
    break;
case 4800: speed = B4800;
    break;
case 9600: speed = B9600;
    break;
case 19200: speed = B19200;
    break;
case 38400: speed = B38400;
    break;
case 57600: speed = B57600;
    break;
case 115200: speed = B115200;
    break;
case 230400: speed = B230400;
    break;
case 460800: speed = B460800;
    break;
case 500000: speed = B500000;
    break;
default: //默认配置为 115200
    speed = B115200;
    printf("default baud rate: 115200\n");
    break;
}

if (0 > cfsetspeed(&new_cfg, speed)) {
    fprintf(stderr, "cfsetspeed error: %s\n", strerror(errno));
    return -1;
}

/* 设置数据位大小 */
new_cfg.c_cflag &= ~CSIZE; //将数据位相关的比特位清零
```



```
switch (cfg->dbit) {
case 5:
    new_cfg.c_cflag |= CS5;
    break;
case 6:
    new_cfg.c_cflag |= CS6;
    break;
case 7:
    new_cfg.c_cflag |= CS7;
    break;
case 8:
    new_cfg.c_cflag |= CS8;
    break;
default:    //默认数据位大小为 8
    new_cfg.c_cflag |= CS8;
    printf("default data bit size: 8\n");
    break;
}

/* 设置奇偶校验 */
switch (cfg->parity) {
case 'N':    //无校验
    new_cfg.c_cflag &= ~PARENB;
    new_cfg.c_iflag &= ~INPCK;
    break;
case 'O':    //奇校验
    new_cfg.c_cflag |= (PARODD | PARENB);
    new_cfg.c_iflag |= INPCK;
    break;
case 'E':    //偶校验
    new_cfg.c_cflag |= PARENB;
    new_cfg.c_cflag &= ~PARODD; /* 清除 PARODD 标志, 配置为偶校验 */
    new_cfg.c_iflag |= INPCK;
    break;
default:    //默认配置为无校验
    new_cfg.c_cflag &= ~PARENB;
    new_cfg.c_iflag &= ~INPCK;
    printf("default parity: N\n");
    break;
}

/* 设置停止位 */
switch (cfg->sbit) {
```

```
case 1: //1 个停止位
    new_cfg.c_cflag &= ~CSTOPB;
    break;
case 2: //2 个停止位
    new_cfg.c_cflag |= CSTOPB;
    break;
default: //默认配置为 1 个停止位
    new_cfg.c_cflag &= ~CSTOPB;
    printf("default stop bit size: 1\n");
    break;
}

/* 将 MIN 和 TIME 设置为 0 */
new_cfg.c_cc[VTIME] = 0;
new_cfg.c_cc[VMIN] = 0;

/* 清空缓冲区 */
if (0 > tcflush(fd, TCIOFLUSH)) {
    fprintf(stderr, "tcflush error: %s\n", strerror(errno));
    return -1;
}

/* 写入配置、使配置生效 */
if (0 > tcsetattr(fd, TCSANOW, &new_cfg)) {
    fprintf(stderr, "tcsetattr error: %s\n", strerror(errno));
    return -1;
}

/* 配置 OK 退出 */
return 0;
}

/**
--dev=/dev/ttymx2
--brate=115200
--dbit=8
--parity=N
--sbit=1
--type=read
***/
/**
** 打印帮助信息
**/
```

```

static void show_help(const char *app)
{
    printf("Usage: %s [选项]\n"
           "\n 必选选项:\n"
           "  --dev=DEVICE    指定串口终端设备名称, 譬如--dev=/dev/ttymx2\n"
           "  --type=TYPE     指定操作类型, 读串口还是写串口, 譬如--type=read(read 表示读、write 表示写、其它值无效)\n"
           "\n 可选选项:\n"
           "  --brate=SPEED  指定串口波特率, 譬如--brate=115200\n"
           "  --dbit=SIZE    指定串口数据位个数, 譬如--dbit=8(可取值为: 5/6/7/8)\n"
           "  --parity=PARITY 指定串口奇偶校验方式, 譬如--parity=N(N 表示无校验、O 表示奇校验、E 表示偶校验)\n"
           "  --sbit=SIZE    指定串口停止位个数, 譬如--sbit=1(可取值为: 1/2)\n"
           "  --help        查看本程序使用帮助信息\n\n", app);
}

/**
** 信号处理函数, 当串口有数据可读时, 会跳转到该函数执行
**/
static void io_handler(int sig, siginfo_t *info, void *context)
{
    unsigned char buf[10] = {0};
    int ret;
    int n;

    if(SIGRTMIN != sig)
        return;

    /* 判断串口是否有数据可读 */
    if (POLL_IN == info->si_code) {
        ret = read(fd, buf, 8);    //一次最多读 8 个字节数据
        printf("[ ");
        for (n = 0; n < ret; n++)
            printf("0x%hhx ", buf[n]);
        printf("]\n");
    }
}

/**
** 异步 I/O 初始化函数
**/
static void async_io_init(void)
{

```

```
struct sigaction sigatn;
int flag;

/* 使能异步 I/O */
flag = fcntl(fd, F_GETFL); //使能串口的异步 I/O 功能
flag |= O_ASYNC;
fcntl(fd, F_SETFL, flag);

/* 设置异步 I/O 的所有者 */
fcntl(fd, F_SETOWN, getpid());

/* 指定实时信号 SIGRTMIN 作为异步 I/O 通知信号 */
fcntl(fd, F_SETSIG, SIGRTMIN);

/* 为实时信号 SIGRTMIN 注册信号处理函数 */
sigatn.sa_sigaction = io_handler; //当串口有数据可读时, 会跳转到 io_handler 函数
sigatn.sa_flags = SA_SIGINFO;
sigemptyset(&sigatn.sa_mask);
sigaction(SIGRTMIN, &sigatn, NULL);
}

int main(int argc, char *argv[])
{
    uart_cfg_t cfg = {0};
    char *device = NULL;
    int rw_flag = -1;
    unsigned char w_buf[10] = {0x11, 0x22, 0x33, 0x44,
                               0x55, 0x66, 0x77, 0x88}; //通过串口发送出去的数据
    int n;

    /* 解析出参数 */
    for (n = 1; n < argc; n++) {

        if (!strcmp("--dev=", argv[n], 6))
            device = &argv[n][6];
        else if (!strcmp("--brate=", argv[n], 8))
            cfg.baudrate = atoi(&argv[n][8]);
        else if (!strcmp("--dbit=", argv[n], 7))
            cfg.dbit = atoi(&argv[n][7]);
        else if (!strcmp("--parity=", argv[n], 9))
            cfg.parity = argv[n][9];
        else if (!strcmp("--sbit=", argv[n], 7))
            cfg.sbit = atoi(&argv[n][7]);
    }
}
```

```
else if (!strcmp("--type=", argv[n], 7)) {
    if (!strcmp("read", &argv[n][7]))
        rw_flag = 0;        //读
    else if (!strcmp("write", &argv[n][7]))
        rw_flag = 1;        //写
}
else if (!strcmp("--help", argv[n])) {
    show_help(argv[0]); //打印帮助信息
    exit(EXIT_SUCCESS);
}
}

if (NULL == device || -1 == rw_flag) {
    fprintf(stderr, "Error: the device and read|write type must be set!\n");
    show_help(argv[0]);
    exit(EXIT_FAILURE);
}

/* 串口初始化 */
if (uart_init(device))
    exit(EXIT_FAILURE);

/* 串口配置 */
if (uart_cfg(&cfg)) {
    tcsetattr(fd, TCSANOW, &old_cfg); //恢复到之前的配置
    close(fd);
    exit(EXIT_FAILURE);
}

/* 读写串口 */
switch (rw_flag) {
case 0: //读串口数据
    async_io_init(); //我们使用异步 I/O 方式读取串口的数据, 调用该函数去初始化串口的异步 I/O
    for (;;)
        sleep(1); //进入休眠、等待有数据可读, 有数据可读之后就会跳转到 io_handler()函数
    break;
case 1: //向串口写入数据
    for (;;) { //循环向串口写入数据
        write(fd, w_buf, 8); //一次向串口写入 8 个字节
        sleep(1); //间隔 1 秒钟
    }
    break;
}
}
```

```
/* 退出 */
tcsetattr(fd, TCSANOW, &old_cfg); //恢复到之前的配置
close(fd);
exit(EXIT_SUCCESS);
}
```

代码稍稍有点长,不过与串口相关的代码并不是很多,代码中所涉及到的所有内容在前面的章节中都给大家详细介绍过。

首先来看下 main()函数,进入到 main()函数之后有一个 for()循环,这是对用户传参进行了解析,我们这个应用程序设计的时候,允许用户传入相应的参数,譬如用户可以指定串口终端的设备节点、串口波特率、数据位个数、停止位个数、奇偶校验等,具体的使用方法,大家可以看一看 show_help()函数。

接下来调用了 uart_init()函数,这是一个自定义的函数,用于初始化串口,实际上就做了两件事:打开串口终端设备、获取串口终端当前的配置参数,将其保存到 old_cfg 变量中。

接着调用 uart_cfg()函数,这也是一个自定义函数,用于对串口进行配置,包括将串口配置为原始模式、使能串口接收、设置串口波特率、数据位个数、停止位个数、奇偶校验,以及 MIN 和 TIME 值的设置,最后清空缓冲区,将配置参数写入串口设备使其生效,具体的代码大家自己去看,代码的注释都已经写的很清楚了!

最后根据用户传参中, --type 选项所指定类型进行读串口或写串口操作,如果 --type=read 表示本次测试是进行串口读取操作,如果 --type=write 表示本次测试是进行串口写入操作。

对于读取串口数据,程序使用了异步 I/O 的方式读取数据,首先调用 async_io_init()函数对异步 I/O 进行初始化,注册信号处理函数。当检测到有数据可读时,会跳转到信号处理函数 io_handler()执行,在这个函数中读取串口的数据并将其打印出来,这里需要注意的是,本例程一次最多读取 8 个字节数据,如果可读数据大于 8 个字节,多余的数据会在下一次 read()调用时被读取出来。

对于写操作,我们直接调用 write()函数,每隔一秒钟向串口写入 8 个字节数据[0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88]。

好了,这个示例代码就给大家介绍完了,非常简单,没什么难点,相信各位聪明的读者绝对能看懂!如果看不懂?那可能就.....呵呵了!

接下来我们要编译示例代码:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 26.2.1 编译示例代码

26.3 在开发板上进行测试

将上小节编译得到的可执行文件拷贝到开发板 Linux 系统/home/root 目录下,如下所示:

```
root@ATK-IMX6U:~# pwd
/home/root
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 26.3.1 将串口测试程序拷贝到开发板

ALPHA I.MX6U 开发板上共预留出了两个串口，一个 USB 串口（对应 I.MX6U 的 UART1）、一个 RS232/RS485 串口（对应 I.MX6U 的 UART3），如图 26.3.2 和图 26.3.3 所示。

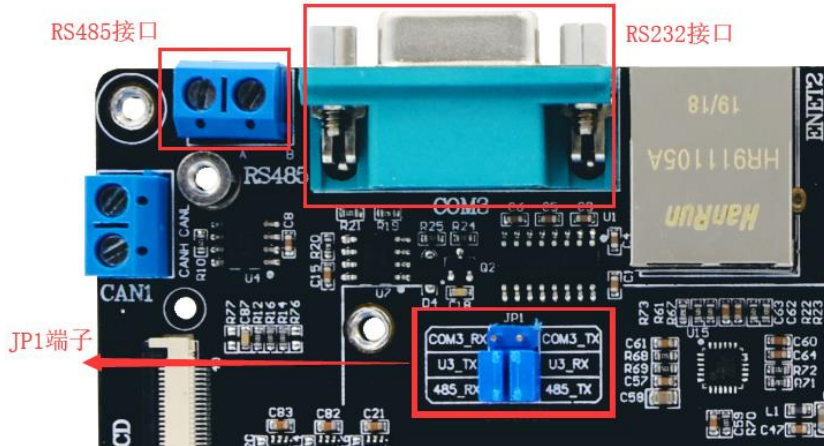


图 26.3.2 RS232/RS485 串口

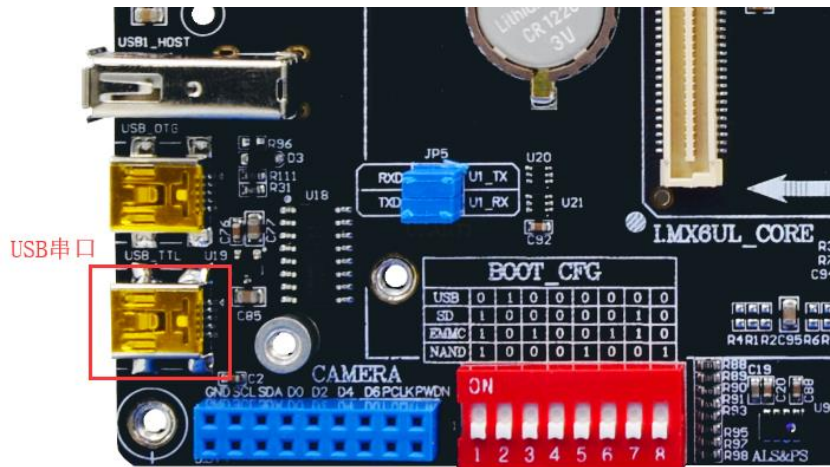


图 26.3.3 USB 串口

注意，板子上的 485 和 232 接口是共用了 I.MX6U 的 UART3，这两个接口无法同时使用，可通过配置底板上的 JP1 端子来使能 RS232 或 RS485 接口，使用跳线帽将每一列上面的两个针脚连接起来，此时 RS232 接口被使能、而 RS485 接口不能使用；如果使用跳线帽将下面两个针脚连接起来，如图 26.3.2 中所示，则此时 RS485 接口被使能、RS232 接口不能使用。

本次测试笔者使用 RS232 串口，注意不能使用 USB 串口进行测试，它是系统的控制台终端。由于 Mini 开发板只有一个 USB 串口，没有 RS232 或 RS485 接口，所以不太好测试，当然并不是说没有办法进行测试；虽然 Mini 板上没有 232 或 485 接口，但是串口用到的 I/O 都已经通过扩展口引出了，你使用一个 USB 转 TTL 模块也是可以测试的。

将板上的 RS232 接口通过<USB 转 RS232>串口线连接到 PC 机。

接下来进行测试, 首先执行如下命令查看测试程序的帮助信息:

```
./testApp --help
```

```
root@ATK-IMX6U:~# ./testApp --help
Usage: ./testApp [选项]

必选选项:
--dev=DEVICE      指定串口终端设备名称, 譬如--dev=/dev/ttyxc2
--type=TYPE       指定操作类型, 读串口还是写串口, 譬如--type=read

可选选项:
--brate=SPEED    指定串口波特率, 譬如--brate=115200
--dbit=SIZE      指定串口数据位个数, 譬如--dbit=8
--parity=PARITY  指定串口奇偶校验方式, 譬如--parity=N(N表示无校验、O表示奇校验、E表示偶校验)
--sbit=SIZE      指定串口停止位个数, 譬如--sbit=1
--help          查看本程序使用帮助信息

root@ATK-IMX6U:~#
```

图 26.3.4 查看测试程序的帮助信息

可选选项表示是可选的, 如果没有指定则会使用默认值!

先进行读测试:

```
./testApp --dev=/dev/ttyxc2 --type=read
```

```
root@ATK-IMX6U:~# ./testApp --dev=/dev/ttyxc2 --type=read
default baud rate: 115200
default data bit size: 8
default parity: N
default stop bit size: 1
█
```

图 26.3.5 读测试

执行测试程序时, 笔者没有指定波特率、数据位个数、停止位个数以及奇偶校验等, 程序将使用默认的配置, 波特率 115200、数据位个数为 8、停止位个数为 1、无校验!

程序执行之后, 在 Windows 下打开串口调试助手上位机软件, 譬如正点原子的 XCOM 串口调试助手:



图 26.3.6 串口调试助手

打开 XCOM 之后, 对其进行配置、并打开串口, 如下所示:



图 26.3.7 配置上位机并打开串口

点击发送按钮向开发板 RS232 串口发送 8 个字节数据[0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88], 此时我们的应用程序便会读取到串口的数据, 这些数据就是 PC 机串口调试助手发送过来的数据, 如下所示:

```

root@ATK-IMX6U:~# ./testApp --dev=/dev/ttymx2 --type=read
default baud rate: 115200
default data bit size: 8
default parity: N
default stop bit size: 1
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]
[ 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 ]

```

图 26.3.8 应用程序读取到串口的数据

测试完读串口后, 我们再来测试向串口写数据, 按 Ctrl+C 结束测试程序, 再次执行测试程序, 本次测试写串口, 如下所示:

```
./testApp --dev=/dev/ttymx2 --type=write
```

```
root@ATK-IMX6U:~# ./testApp --dev=/dev/ttymx2 --type=write
default baud rate: 115200
default data bit size: 8
default parity: N
default stop bit size: 1
```

图 26.3.9 测试写串口

执行测试程序后,测试程序会每隔 1 秒中将 8 个字节数据[0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88]写入到 RS232 串口,此时 PC 端串口调试助手便会接收到这些数据,如下所示:

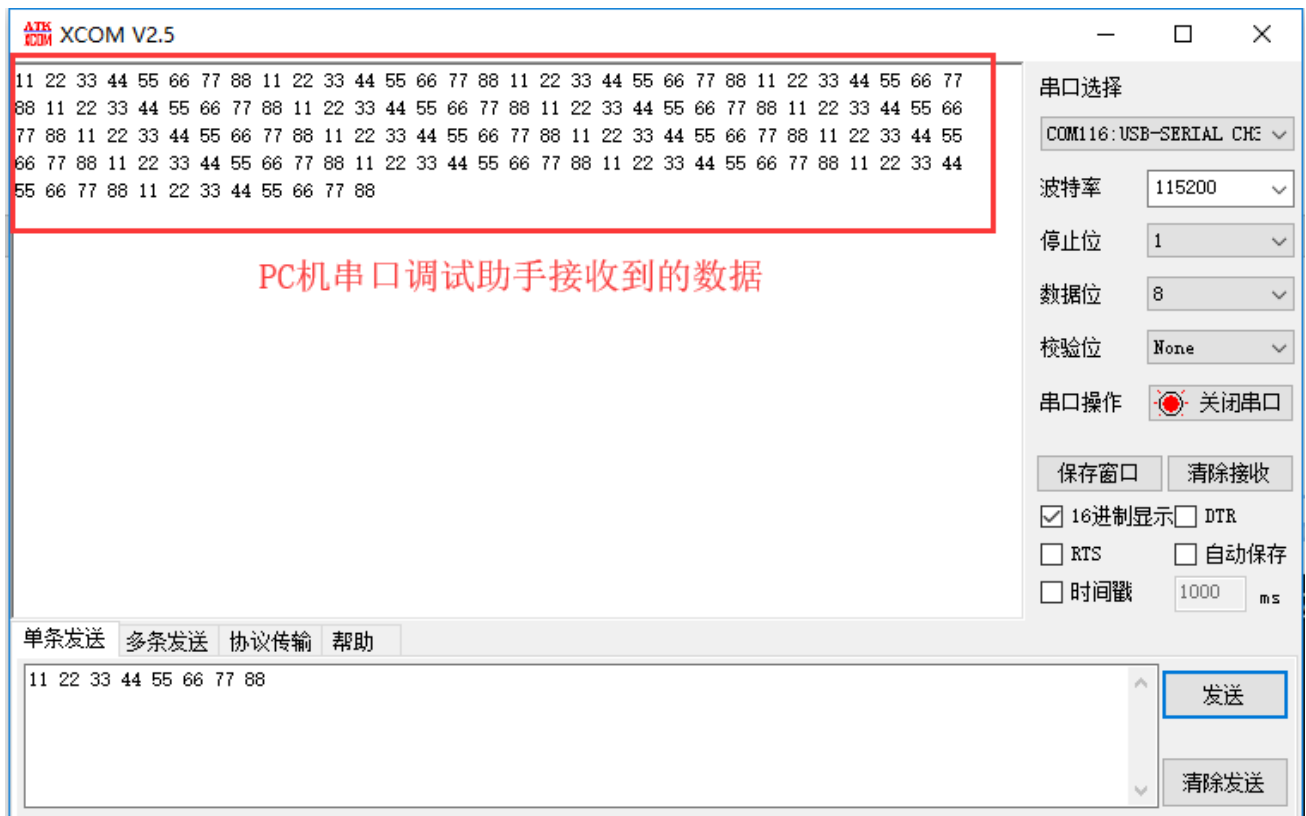


图 26.3.10 串口调试助手接收到开发板 RS232 串口发送过来的数据

本章内容到此结束!

第二十七章 看门狗应用编程

在产品化的嵌入式系统中,为了使系统在异常情况下能自动复位,一般都需要引入看门狗。看门狗其实就是一个可以在一定时间内被复位的计数器。当看门狗启动后,计数器开始自动计数,经过一定时间,如果没有被复位,计数器溢出就会对 CPU 产生一个复位信号使系统重启(俗称“被狗咬”)。系统正常运行时,需要在看门狗允许的时间间隔内对看门狗计数器清零(俗称“喂狗”),不让复位信号产生。如果系统不出问题,程序保证按时“喂狗”,一旦程序跑飞,没有“喂狗”,系统“被咬”复位。

本章我们来学习应用层如何控制看门狗外设,譬如打开看门狗、关闭看门狗以及喂狗等操作。

本章将会讨论如下主题内容。

- 看门狗应用编程介绍;
- 编写程序控制看门狗外设。

27.1 看门狗应用编程介绍

前面已经说到, 看门狗其实就是一个可以在一定时间内被复位/重置的计数器, 一般叫做看门狗计时器(或看门狗定时器); 如果在规定时间内没有复位看门狗计时器, 计数器溢出则会对 CPU 产生一个复位信号使系统重启, 当然有些看门狗也可以只产生中断信号而不会使系统复位。

I.MX6UL/I.MX6ULL SoC 集成了两个看门狗定时器(WDOG): WDOG1 和 WDOG2; WDOG2 用于安全目的, 而 WDOG1 则是一个普通的看门狗, 支持产生中断信号以及复位 CPU。

Linux 系统中所注册的看门狗外设, 都会在/dev/目录下生成对应的设备节点(设备文件), 设备节点名称通常为 watchdogX (X 表示一个数字编号 0、1、2、3 等), 譬如/dev/watchdog0、/dev/watchdog1 等, 通过这些设备节点可以控制看门狗外设。

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls -l /dev/watchdog*
crw----- 1 root root 10, 130 Jun 23 14:37 /dev/watchdog
crw----- 1 root root 251, 0 Jun 23 14:37 /dev/watchdog0
root@ATK-IMX6U:~#
```

图 27.1.1 watchdog0 和 watchdog 设备节点

这个 watchdog0 其实就是 I.MX6U 的 WDOG1 所对应的设备节点, 从图中可知, 除了/dev/watchdog0 之外, 还有一个 watchdog 设备节点, 这个设备节点的名称没有后面的数字编号, 这个又是什么意思呢? 因为系统中可能注册了多个看门狗设备, /dev/watchdog 设备节点则代表系统默认的看门狗设备; 通常这指的就是 watchdog0, 所以, 上图中, /dev/watchdog 其实就等于/dev/watchdog0, 也就意味着它俩代表的是同一个硬件外设。

应用层控制看门狗其实非常简单, 通过 ioctl()函数即可做到! 接下来笔者向大家进行介绍。

首先在我们的应用程序中, 需要包含头文件<linux/watchdog.h>头文件, 该头文件中定义了一些 ioctl 指令宏, 每一个不同的指令宏表示向设备请求不同的操作, 如下所示:

```
#define WDIOC_GETSUPPORT      _IOR(WATCHDOG_IOCTL_BASE, 0, struct watchdog_info)
#define WDIOC_GETSTATUS      _IOR(WATCHDOG_IOCTL_BASE, 1, int)
#define WDIOC_GETBOOTSTATUS  _IOR(WATCHDOG_IOCTL_BASE, 2, int)
#define WDIOC_GETTEMP        _IOR(WATCHDOG_IOCTL_BASE, 3, int)
#define WDIOC_SETOPTIONS     _IOR(WATCHDOG_IOCTL_BASE, 4, int)
#define WDIOC_KEEPAWAKE      _IOR(WATCHDOG_IOCTL_BASE, 5, int)
#define WDIOC_SETTIMEOUT     _IOWR(WATCHDOG_IOCTL_BASE, 6, int)
#define WDIOC_GETTIMEOUT     _IOR(WATCHDOG_IOCTL_BASE, 7, int)
#define WDIOC_SETPRETIMEOUT  _IOWR(WATCHDOG_IOCTL_BASE, 8, int)
#define WDIOC_GETPRETIMEOUT  _IOR(WATCHDOG_IOCTL_BASE, 9, int)
#define WDIOC_GETTIMELEFT    _IOR(WATCHDOG_IOCTL_BASE, 10, int)
```

比较常用指令包括: WDIOC_GETSUPPORT、WDIOC_SETOPTIONS、WDIOC_KEEPAWAKE、WDIOC_SETTIMEOUT、WDIOC_GETTIMEOUT, 说明如下:

ioctl 指令	说明
WDIOC_GETSUPPORT	获取看门狗支持哪些功能
WDIOC_SETOPTIONS	用于开启或关闭看门狗
WDIOC_KEEPAWAKE	喂狗操作
WDIOC_SETTIMEOUT	设置看门狗超时时间
WDIOC_GETTIMEOUT	获取看门狗超时时间

表 27.1.1 看门狗常用指令

27.1.1 打开设备

首先在调用 `ioctl()` 函数之前, 需要先打开看门狗设备得到文件描述符, 如下所示:

```
int fd;

fd = open("/dev/watchdog", "O_RDWR");
if (0 > fd)
    fprintf(stderr, "open error: %s: %s\n", "/dev/watchdog", strerror(errno));
```

27.1.2 获取设备支持哪些功能: `WDIOC_GETSUPPORT`

使用 `WDIOC_GETSUPPORT` 指令获取看门狗设备支持哪些功能, 使用方式如下:

```
ioctl(int fd, WDIOC_GETSUPPORT, struct watchdog_info *info);
```

使用 `WDIOC_GETSUPPORT` 指令可以获取设备的信息, 调用 `ioctl()` 需要传入一个 `struct watchdog_info` *指针, `ioctl()` 会将获取到的数据写入到 `info` 指针所指向的对象中。`struct watchdog_info` 结构体描述了看门狗设备的信息, 我们来看看 `struct watchdog_info` 结构体的定义:

示例代码 27.1.1 `struct watchdog_info` 结构体

```
struct watchdog_info {
    __u32 options;           /* Options the card/driver supports */
    __u32 firmware_version; /* Firmware version of the card */
    __u8  identity[32];     /* Identity of the board */
};
```

`options` 字段记录了设备支持哪些功能或选项;

`firmware_version` 字段记录了设备的固件版本号;

`identity` 字段则是一个描述性的字符串。

我们重点关注的是 `options` 字段, 该字段描述了设备支持哪些功能、选项, 该字段的值如下 (可以是以下任意一个值或多个值的位或关系):

```
#define WDIOF_OVERHEAT      0x0001 /* Reset due to CPU overheat */
#define WDIOF_FANFAULT     0x0002 /* Fan failed */
#define WDIOF_EXTERN1      0x0004 /* External relay 1 */
#define WDIOF_EXTERN2      0x0008 /* External relay 2 */
#define WDIOF_POWERUNDER   0x0010 /* Power bad/power fault */
#define WDIOF_CARDRESET     0x0020 /* Card previously reset the CPU */
#define WDIOF_POWEROVER    0x0040 /* Power over voltage */
#define WDIOF_SETTIMEOUT   0x0080 /* Set timeout (in seconds) */
#define WDIOF_MAGICCLOSE   0x0100 /* Supports magic close char */
#define WDIOF_PRETIMEOUT   0x0200 /* Pretimeout (in seconds), get/set */
#define WDIOF_ALARMONLY    0x0400 /* Watchdog triggers a management or other external alarm
not a reboot */
#define WDIOF_KEEPALIVEPING 0x8000 /* Keep alive ping reply */
```

一般常见的值包括: `WDIOF_SETTIMEOUT`、`WDIOF_KEEPALIVEPING`; `WDIOF_SETTIMEOUT` 表示设备支持设置超时时间; `WDIOF_KEEPALIVEPING` 表示设备支持“喂狗”操作, 也就是重置看门狗计时器。

使用示例如下:

```
struct watchdog_info info;
```

```
if (0 > ioctl(fd, WDIOG_GETSUPPORT, &info)) {
    fprintf(stderr, "ioctl error: WDIOG_GETSUPPORT: %s\n", strerror(errno));
    return -1;
}

printf("identity: %s\n", info.identity);
printf("version: %u\n", firmware_version);

if (0 == (WDIOF_KEEPAALIVEPING & info.options))
    printf("设备不支持喂狗操作\n");
if (0 == (WDIOF_SETTIMEOUT & info.options))
    printf("设备不支持设置超时时间\n");
```

27.1.3 获取/设置超时时间: WDIOG_GETTIMEOUT、WDIOG_SETTIMEOUT

使用 WDIOG_GETTIMEOUT 指令可获取设备当前设置的超时时间, 使用方式如下:

```
ioctl(int fd, WDIOG_GETTIMEOUT, int *timeout);
```

使用 WDIOG_SETTIMEOUT 指令可设置看门狗的超时时间, 使用方式如下:

```
ioctl(int fd, WDIOG_SETTIMEOUT, int *timeout);
```

超时时间是以秒为单位, 设置超时时间时, 不可超过其最大值、否则 ioctl()调用将会失败, 使用示例如下所示:

```
int timeout;

/* 获取超时时间 */
if (0 > ioctl(fd, WDIOG_GETTIMEOUT, &timeout)) {
    fprintf(stderr, "ioctl error: WDIOG_GETTIMEOUT: %s\n", strerror(errno));
    return -1;
}

printf("current timeout: %ds\n", timeout);

/* 设置超时时间 */
timeout = 10; //10 秒钟
if (0 > ioctl(fd, WDIOG_SETTIMEOUT, &timeout)) {
    fprintf(stderr, "ioctl error: WDIOG_SETTIMEOUT: %s\n", strerror(errno));
    return -1;
}
```

27.1.4 开启/关闭看门狗: WDIOG_SETOPTIONS

设置好超时时间之后, 接着便可以开启看门狗计时了, 使用 WDIOG_SETOPTIONS 指令可以开启看门狗计时或停止看门狗计时, 使用方式如下:

```
ioctl(int fd, WDIOG_SETOPTIONS, int *option);
```

option 指针指向一个 int 类型变量, 该变量可取值如下:

```
#define WDIOS_DISABLECARD 0x0001 /* Turn off the watchdog timer */
#define WDIOS_ENABLECARD 0x0002 /* Turn on the watchdog timer */
```

WDIOS_DISABLECARD 表示停止看门狗计时, WDIOS_ENABLECARD 则表示开启看门狗计时。
使用示例如下所示:

```
int option = WDIOS_ENABLECARD; //开启
//int option = WDIOS_DISABLECARD; //停止

if (0 > ioctl(fd, WDIOC_SETOPTIONS, &option)) {
    fprintf(stderr, "ioctl error: WDIOC_SETOPTIONS: %s\n", strerror(errno));
    return -1;
}
```

需要注意的是, 当调用 `open()` 打开看门狗设备的时候, 即使程序中没有开启看门狗计时器, 当 `close()` 关闭设备时, 看门狗会自动启动; 所以, 当打开设备之后, 需要使用 `WDIOC_SETOPTIONS` 指令停止看门狗计时, 等所有设置完成之后再开启看门狗计时器。

27.1.5 喂狗: WDIOC_KEEPLIVE

看门狗计时器启动之后, 我们需要在超时之前, 去“喂狗”, 否则计时器溢出超时将会导致系统复位或产生一个中断信号, 通过 `WDIOC_KEEPLIVE` 指令喂狗, 使用方式如下:

```
ioctl(int fd, WDIOC_KEEPLIVE, NULL);
```

使用示例如下:

```
if (0 > ioctl(fd, WDIOC_KEEPLIVE, NULL)) {
    fprintf(stderr, "ioctl error: WDIOC_KEEPLIVE: %s\n", strerror(errno));
}
```

27.2 看门狗应用编程实战

通过上小节介绍之后, 我们已经知道了如何编写应用程序去控制看门狗外设了, 本小节我们来编写一个简单地看门狗应用程序, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->27_watchdog->watchdog_test.c](#)。

示例代码 27.2.1 看门狗测试程序

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : watchdog_test.c
作者 : 邓涛
版本 : V1.0
描述 : 看门狗应用程序示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/14 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
#include <linux/watchdog.h>

#define WDOG_DEV "/dev/watchdog"

int main(int argc, char *argv[])
{
    struct watchdog_info info;
    int timeout;
    int time;
    int fd;
    int op;

    if (2 != argc) {
        fprintf(stderr, "usage: %s <timeout>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 打开看门狗 */
    fd = open(WDOG_DEV, O_RDWR);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", WDOG_DEV, strerror(errno));
        exit(EXIT_FAILURE);
    }

    /* 打开之后看门狗计时器会开启、先停止它 */
    op = WDIOF_DISABLECARD;
    if (0 > ioctl(fd, WDIOF_SETOPTIONS, &op)) {
        fprintf(stderr, "ioctl error: WDIOF_SETOPTIONS: %s\n", strerror(errno));
        close(fd);
        exit(EXIT_FAILURE);
    }

    timeout = atoi(argv[1]);
    if (1 > timeout)
        timeout = 1;

    /* 设置超时时间 */
```



```

printf("timeout: %ds\n", timeout);
if (0 > ioctl(fd, WDIOC_SETTIMEOUT, &timeout)) {
    fprintf(stderr, "ioctl error: WDIOC_SETTIMEOUT: %s\n", strerror(errno));
    close(fd);
    exit(EXIT_FAILURE);
}

/* 开启看门狗计时器 */
op = WDIOS_ENABLECARD;
if (0 > ioctl(fd, WDIOC_SETOPTIONS, &op)) {
    fprintf(stderr, "ioctl error: WDIOC_SETOPTIONS: %s\n", strerror(errno));
    close(fd);
    exit(EXIT_FAILURE);
}

/* 喂狗 */
time = (timeout * 1000 - 100) * 1000; //喂狗时间设置 us 微秒、在超时时间到来前 100ms 喂狗
for (;) {

    usleep(time);
    ioctl(fd, WDIOC_KEEPAIVE, NULL);
}
}

```

示例代码很简单，首先打开看门狗设备，接着使用 `WDIOC_SETOPTIONS` 指令（`op = WDIOS_DISABLECARD`）先停止看门狗计时器；接着通过 `atoi` 获取到用户传入的超时时间，所以执行该测试程序的时候，需要传入一个参数作为看门狗超时时间。

接着使用 `WDIOC_SETTIMEOUT` 指令设置超时时间，再使用 `WDIOC_SETOPTIONS` 指令（`op = WDIOS_ENABLECARD`）开启看门狗计时器，看门狗开始工作。接着我们需要在超时时间到来之前，去喂狗，喂狗之后，计时器重置，重新计时；不断地喂狗重置计时器、不让他超时、如果一旦超时系统将会复位重启。

编译示例代码：

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 27.2.1 编译示例代码

将编译得到的可执行文件拷贝到开发板 Linux 系统 `/home/root` 目录下，执行测试程序，譬如启动看门狗，设置超时时间为 2 秒钟，如下：

```
root@ATK-IMX6U:~# ls
driver  shell  testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp 2
timeout: 2s
```

图 27.2.2 开启看门狗 2 秒超时时间

执行程序之后, 开门狗计时器就已经启动了, 程序中会不断的喂狗重置计时器, 以保证程序不会重启。

现在我们按 Ctrl+C 结束程序, 结束程序意味着已经停止喂狗了、然后看门狗计时器并没有停止, 这样将会导致计时器溢出、发生复位重启:

```
root@ATK-IMX6U:~# ./testApp 2
timeout: 2s
^C[ 367.921167] watchdog watchdog0: watchdog did not stop!

root@ATK-IMX6U:~#

U-Boot 2016.03-ge468cdc (Mar 29 2021 - 15:59:40 +0800)

CPU:   Freescale i.MX6ULL rev1.1 792 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 50C
Reset cause: POR
Board: I.MX6U ALPHA|MINI
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment
```

重启, 这
是u-boot
启动代码

图 27.2.3 计时器溢出系统重启

当按 Ctrl+C 终止进程后, 内核打印出 “watchdog watchdog0: watchdog did not stop!” 信息, 表示看门狗计时器还正在计时、未停止。

第二十八章 音频应用编程

ALPHA I.MX6U 开发板支持音频, 板上搭载了音频编解码芯片 WM8960, 支持播放以及录音功能!

本章我们来学习 Linux 下的音频应用编程, 音频应用编程相比于前面几个章节所介绍的内容、其难度有所上升, 但是笔者仅向大家介绍 Linux 音频应用编程中的基础知识, 而更多细节、更加深入的内容需要大家自己去学习。

本章将会讨论如下主题内容。

- Linux 下 ALSA 框架概述;
- alsa-lib 库介绍;
- alsa-lib 库移植;
- alsa-lib 库的使用;
- 音频应用编程之播放;
- 音频应用编程之录音。

28.1 ALSA 概述

ALSA 是 Advanced Linux Sound Architecture (高级的 Linux 声音体系) 的缩写, 目前已经成为了 linux 下的主流音频体系架构, 提供了音频和 MIDI 的支持, 替代了原先旧版本中的 OSS (开发声音系统); 学习过 Linux 音频驱动开发的读者肯定知道这个; 事实上, ALSA 是 Linux 系统下一套标准的、先进的音频驱动框架, 那么这套框架的设计本身是比较复杂的, 采用分离、分层思想设计而成, 具体的细节便不给大家介绍了! 作为音频应用编程, 我们不用去研究这个。

在应用层, ALSA 为我们提供了一套标准的 API, 应用程序只需要调用这些 API 就可完成对底层音频硬件设备的控制, 譬如播放、录音等, 这一套 API 称为 alsa-lib。如下图所示:

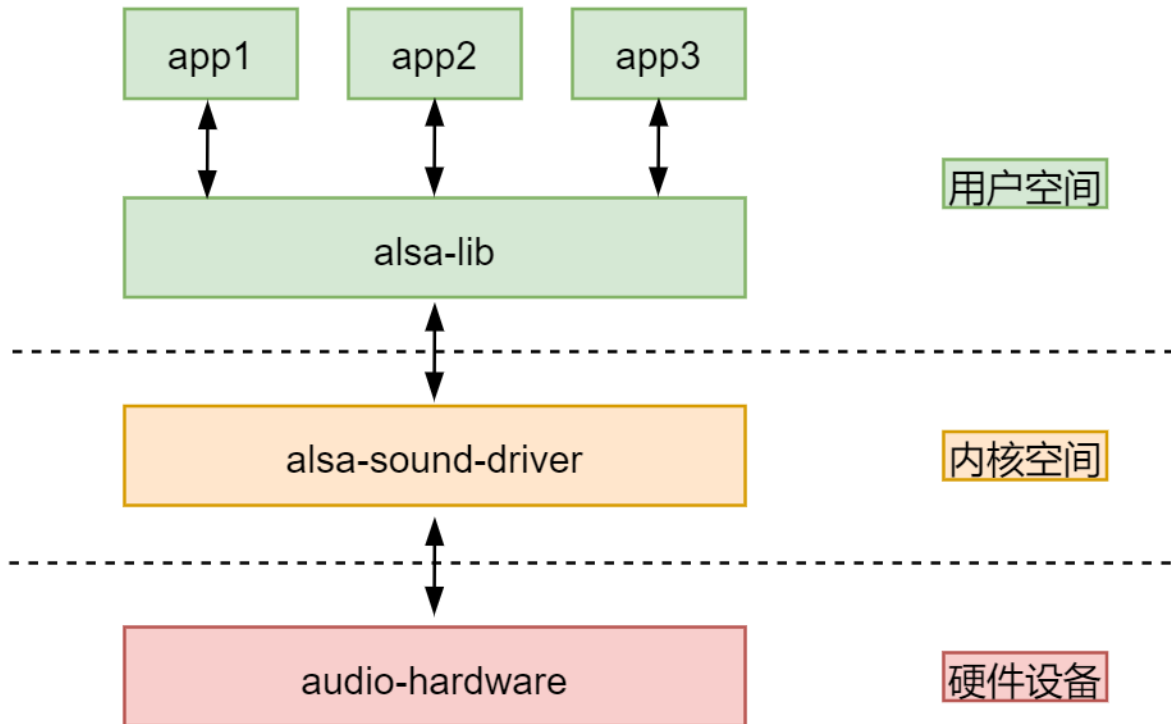


图 28.1.1 alsa 音频示意图

28.2 alsa-lib 简介

如上所述, alsa-lib 是一套 Linux 应用层的 C 语言函数库, 为音频应用程序开发提供了一套统一、标准的接口, 应用程序只需调用这一套 API 即可完成对底层声卡设备的操控, 譬如播放与录音。

用户空间的 alsa-lib 对应用程序提供了统一的 API 接口, 这样可以隐藏驱动层的实现细节, 简化了应用程序的实现难度、无需应用程序开发人员直接去读写音频设备节点。所以本章, 对于我们来说, 学习音频应用编程其实就是学习 alsa-lib 库函数的使用、如何基于 alsa-lib 库函数开发音频应用程序。

ALSA 提供了关于 alsa-lib 的使用说明文档, 其链接地址为: <https://www.alsa-project.org/alsa-doc/alsa-lib/>, 进入到该链接地址后, 如下所示:

ALSA project - the C library reference

Main Page | Related Pages | Modules | Data Structures ▾ | Files ▾ | Examples

Index Preamble and License

Author

Jaroslav Kysela perex@perex.cz
Abramo Bagnara abramo@alsa-project.org
Takashi Iwai tiwai@suse.de
Frank van de Pol fvdpol@coil.demon.nl

Preface

The Advanced Linux Sound Architecture (*ALSA*) comes with a kernel API and a library API. This document describes the library API and how it interfaces with the kernel API.

Documentation License

This documentation is free; you can redistribute it without any restrictions. Modifications or derived work must retain the copyright and list all authors.

This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

API usage

Application programmers should use the library API rather than the kernel API. The library offers 100% of the functionality of the kernel API, but adds major improvements in usability, making the application code driver.

API links

- Page **Control interface** explains the primitive controls API.
- Page **Primitive control plugins** explains the design of primitive control plugins.
- Page **High level control interface** explains the high-level primitive controls API.
- Page **Mixer interface** explains the mixer controls API.
- Page **PCM (digital audio) interface** explains the design of the PCM (digital audio) API.
- Page **PCM (digital audio) plugins** explains the design of PCM (digital audio) plugins.
- Page **PCM External Plugin SDK** explains the external PCM plugin SDK.
- Page **ct_external_plugins** explains the external control plugin SDK.
- Page **RawMidi interface** explains the design of the RawMidi API.
- Page **Timer interface** explains the design of the Timer API.
- Page **Sequencer interface** explains the design of the Sequencer API.
- Page **Use Case Interface** explains the use case API.
- Page **ALSA Topology Interface** explains the DSP topology API.

图 28.2.1 alsa-lib 使用参考手册

alsa-lib 库支持功能比较多, 提供了丰富的 API 接口供应用程序开发人员调用, 根据函数的功能、作用将这些 API 进行了分类, 可以点击上图中 Modules 按钮查看其模块划分, 如下所示:

ALSA project - the C library reference

Main Page	Related Pages	Modules	Data Structures ▾	Files ▾	Examples
-----------	---------------	----------------	-------------------	---------	----------

Modules

Here is a list of all modules:

Global defines and functions	
- Constants for Digital Audio Interfaces	
Input Interface	
Output Interface	
Error handling	
Configuration Interface	
- Control Interface	
- PCM Interface	
RawMidi Interface	
Timer Interface	
Hardware Dependant Interface	
- MIDI Sequencer	
- External PCM plugin SDK	
External Control Plugin SDK	
- Mixer Interface	
Use Case Interface	
Topology Interface	

图 28.2.2 alsa-lib 模块

一个分类就是一个模块 (module)，有些模块下可能该包含了子模块，譬如上图中，模块名称前面有三角箭头的表示该模块包含有子模块。

- Global defines and functions: 包括一些全局的定义，譬如函数、宏等；
- Constants for Digital Audio Interfaces: 数字音频接口相关的常量；
- Input Interface: 输入接口；
- Output Interface: 输出接口；
- Error handling: 错误处理相关接口；
- Configuration Interface: 配置接口；
- Control Interface: 控制接口；
- PCM Interface: PCM 设备接口；
- RawMidi Interface: RawMidi 接口；
- Timer Interface: 定时器接口；
- Hardware Dependant Interface: 硬件相关接口；
- MIDI Sequencer: MIDI 音序器；
- External PCM plugin SDK: 外部 PCM 插件 SDK；
- External Control Plugin SDK: 外部控制插件 SDK；
- Mixer Interface: 混音器接口；
- Use Case Interface: 用例接口；
- Topology Interface: 拓扑接口。

可以看到，alsa-lib 提供的接口确实非常多、模块很多，以上所列举出来的这些模块，很多模块笔者也不是很清楚它们的具体功能、作用，但是本章我们仅涉及到三个模块下的 API 函数，包括：PCM Interface、Error Interface 以及 Mixer Interface。

PCM Interface

PCM Interface, 提供了 PCM 设备相关的操作接口, 譬如打开/关闭 PCM 设备、配置 PCM 设备硬件或软件参数、控制 PCM 设备 (启动、暂停、恢复、写入/读取数据), 该模块下还包含了一些子模块, 如下所示:

▼ PCM Interface
Stream Information
Hardware Parameters
Software Parameters
Access Mask Functions
Format Mask Functions
Subformat Mask Functions
Status Functions
Description Functions
Debug Functions
<u>Direct Access (MMAP) Functions</u>
Helper Functions
Hook Extension
Scope Plugin Extension
Simple setup functions
Deprecated Functions

图 28.2.3 PCM Interface 下的子模块

点击模块名称可以查看到该模块提供的 API 接口有哪些以及相应的函数说明, 这里就不给大家演示了!

Error Interface

该模块提供了关于错误处理相关的接口, 譬如函数调用发生错误时, 可调用该模块下提供的函数打印错误描述信息。

Mixer Interface

提供了关于混音器相关的一系列操作接口, 譬如音量、声道控制、增益等等。

28.3 sound 设备节点

在 Linux 内核设备驱动层、基于 ALSA 音频驱动框架注册的 sound 设备会在 /dev/snd 目录下生成相应的设备节点文件, 譬如 ALPHA I.MX6U 开发板出厂系统 /dev/snd 目录下有如下文件:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls -l /dev/snd/
total 0
drwxr-xr-x 2 root root      60 Jul 27 15:11 by-path
crw-rw---- 1 root audio 116,  0 Jul 27 15:11 controlC0
crw-rw---- 1 root audio 116, 24 Jul 27 15:11 pcmC0D0c
crw-rw---- 1 root audio 116, 16 Jul 27 15:11 pcmC0D0p
crw-rw---- 1 root audio 116, 25 Jul 27 15:11 pcmC0D1c
crw-rw---- 1 root audio 116, 17 Jul 27 15:11 pcmC0D1p
crw-rw---- 1 root audio 116, 33 Jul 27 15:11 timer
root@ATK-IMX6U:~#
```

图 28.3.1 /dev/snd 目录下的文件

Tips: 注意, Mini I.MX6U 开发板出厂系统 /dev/snd 目录下是没有这些文件的, 因为 Mini 板不支持音频、没有板载音频编解码芯片, 所以本章实验例程无法在 Mini 板上进行测试, 请悉知!

从上图可以看到有如下设备文件:

- **controlC0:** 用于声卡控制的设备节点, 譬如通道选择、混音器、麦克风的控制等, C0 表示声卡 0 (card0);
- **pcmC0D0c:** 用于录音的 PCM 设备节点。其中 C0 表示 card0, 也就是声卡 0; 而 D0 表示 device 0, 也就是设备 0; 最后一个字母 c 是 capture 的缩写, 表示录音; 所以 pcmC0D0c 便是系统的声卡 0 中的录音设备 0;
- **pcmC0D0p:** 用于播放 (或叫放音、回放) 的 PCM 设备节点。其中 C0 表示 card0, 也就是声卡 0; 而 D0 表示 device 0, 也就是设备 0; 最后一个字母 p 是 playback 的缩写, 表示播放; 所以 pcmC0D0p 便是系统的声卡 0 中的播放设备 0;
- **pcmC0D1c:** 用于录音的 PCM 设备节点。对应系统的声卡 0 中的录音设备 1;
- **pcmC0D1p:** 用于播放的 PCM 设备节点。对应系统的声卡 0 中的播放设备 1。
- **timer:** 定时器。

本章我们编写的应用程序, 虽然是调用 alsa-lib 库函数去控制底层音频硬件, 但最终也是落实到对 sound 设备节点的 I/O 操作, 只不过 alsa-lib 已经帮我们封装好了。在 Linux 系统的 /proc/asound 目录下, 有很多的文件, 这些文件记录了系统中声卡相关的信息, 如下所示:

```
root@ATK-IMX6U:/proc/asound# cd
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# cd /proc/asound/
root@ATK-IMX6U:/proc/asound#
root@ATK-IMX6U:/proc/asound# pwd
/proc/asound
root@ATK-IMX6U:/proc/asound# ls -l
total 0
dr-xr-xr-x 6 root root 0 Jul 29 12:50 card0
-r--r--r-- 1 root root 0 Jul 29 12:50 cards
-r--r--r-- 1 root root 0 Jul 29 12:50 devices
-r--r--r-- 1 root root 0 Jul 29 12:50 pcm
-r--r--r-- 1 root root 0 Jul 29 12:50 timers
-r--r--r-- 1 root root 0 Jul 29 12:50 version
lrwxrwxrwx 1 root root 5 Jul 29 12:50 wm8960audio -> card0
root@ATK-IMX6U:/proc/asound#
```

图 28.3.2 /proc/asound 目录下的文件

cards:

通过 "cat /proc/asound/cards" 命令、查看 cards 文件的内容, 可列出系统中可用的、注册的声卡, 如下所示:

```
cat /proc/asound/cards
```

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# cat /proc/asound/cards
 0 [wm8960audio ]: wm8960-audio - wm8960-audio
                    wm8960-audio
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 28.3.3 查看系统中注册的所有声卡

我们的阿尔法板子上只有一个声卡 (WM8960 音频编解码器), 所以它的编号为 0, 也就是 card0。系统中注册的所有声卡都会在 /proc/asound/ 目录下存在一个相应的目录, 该目录的命名方式为 cardX (X 表示声卡的编号), 譬如图 28.3.2 中的 card0; card0 目录下记录了声卡 0 相关的信息, 譬如声卡的名字以及声卡注册的 PCM 设备, 如下所示:


```
root@ATK-IMX6U:/proc/asound/card0# pwd
/proc/asound/card0
root@ATK-IMX6U:/proc/asound/card0#
root@ATK-IMX6U:/proc/asound/card0# ls
id pcm0c pcm0p pcm1c pcm1p
root@ATK-IMX6U:/proc/asound/card0#
root@ATK-IMX6U:/proc/asound/card0# cat id
wm8960audio
root@ATK-IMX6U:/proc/asound/card0#
```

图 28.3.4 card0 目录下的文件

devices:

列出系统中所有声卡注册的设备, 包括 control、pcm、timer、seq 等等。如下所示:

```
cat /proc/asound/devices
```

```
root@ATK-IMX6U:/proc/asound# cat /proc/asound/devices
 0: [ 0] : control
16: [ 0- 0]: digital audio playback
17: [ 0- 1]: digital audio playback
24: [ 0- 0]: digital audio capture
25: [ 0- 1]: digital audio capture
33: : timer
root@ATK-IMX6U:/proc/asound#
```

图 28.3.5 列出所有设备

pcm:

列出系统中的所有 PCM 设备, 包括 playback 和 capture:

```
cat /proc/asound/pcm
```

```
root@ATK-IMX6U:/proc/asound# cat /proc/asound/pcm
00-00: HiFi wm8960-hifi-0 : : playback 1 : capture 1
00-01: HiFi-ASRC-FE (*) : : playback 1 : capture 1
root@ATK-IMX6U:/proc/asound#
```

图 28.3.6 列出系统中所有 PCM 设备

28.4 alsa-lib 移植

因为 alsa-lib 是 ALSA 提供的一套 Linux 下的 C 语言函数库, 需要将 alsa-lib 移植到开发板上, 这样基于 alsa-lib 编写的应用程序才能成功运行, 除了移植 alsa-lib 库之外, 通常还需要移植 alsa-utils, alsa-utils 包含了一些用于测试、配置声卡的工具。

事实上, ALPHA I.MX6U 开发板出厂系统中已经移植了 alsa-lib 和 alsa-utils, 本章我们直接使用出厂系统移植好的 alsa-lib 和 alsa-utils 进行测试, 笔者也就不再介绍移植过程了。其实它们的移植方法也非常简单, 如果你想自己尝试移植, 网上有很多参考, 大家可以自己去看看。

alsa-utils 提供了一些用于测试、配置声卡的工具, 譬如 aplay、arecord、alsactl、alsaloop、alsamixer、amixer 等, 在开发板出厂系统上可以直接使用这些工具, 这些应用程序也都是基于 alsa-lib 编写的。

aplay

aplay 是一个用于测试音频播放功能程序, 可以使用 aplay 播放 wav 格式的音频文件, 如下所示:

```

root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver QDesktop-fb shell test.wav
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# aplay test.wav
Playing WAVE 'test.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo

```

图 28.4.1 使用 aplay 播放 wav 音乐

程序运行之后就会开始播放音乐, 因为 ALPHA 开发板支持喇叭和耳机自动切换, 如果不插耳机默认从喇叭播放音乐, 插上耳机以后喇叭就会停止播放, 切换为耳机播放音乐, 这个大家可以自己进行测试。

需要注意的是, aplay 工具只能解析 wav 格式音频文件, 不支持 mp3 格式解码, 所以无法使用 aplay 工具播放 mp3 音频文件。稍后笔者会向大家介绍如何基于 alsa-lib 编写一个简单地音乐播放器, 实现与 aplay 相同的效果。

alsamixer

alsamixer 是一个很重要的工具, 用于配置声卡的混音器, 它是一个字符图形化的配置工具, 直接在开发板串口终端运行 alsamixer 命令, 打开图形化配置界面, 如下所示:

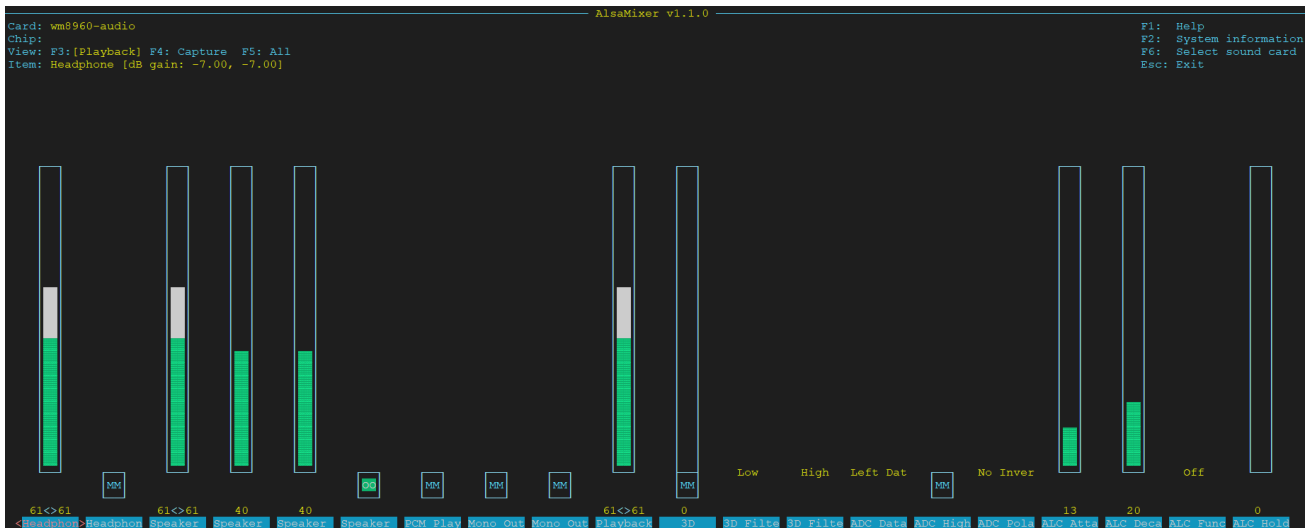


图 28.4.2 alsamixer 界面

alsamixer 可对声卡的混音器进行配置, 左上角 “Card: wm8960-audio” 表示当前配置的声卡为 wm8960-audio, 如果你的系统中注册了多个声卡, 可以按 F6 进行选择。

按下 H 键可查看界面的操作说明, 如下所示:

```

----- Help -----
Esc      Exit
F1 ? H   Help
F2 /     System information
F3       Show playback controls
F4       Show capture controls
F5       Show all controls
Tab      Toggle view mode (F3/F4/F5)
F6 S     Select sound card
L        Redraw screen

Left     Move to the previous control
Right    Move to the next control

Up/Down  Change volume
+ -      Change volume
Page Up/Dn Change volume in big steps
End      Set volume to 0%
0-9      Set volume to 0%-90%
Q W E    Increase left/both/right volumes
Z X C    Decrease left/both/right volumes
B        Balance left and right volumes

M        Toggle mute
< >     Toggle left/right mute

Space    Toggle capture
; '      Toggle left/right capture

Authors:
  Tim Janik
  Jaroslav Kysela <perex@perex.cz>
  Clemens Ladisch <clemens@ladisch.de>

```

图 28.4.3 alsamixer 界面操作说明

不同声卡支持的混音器配置选项是不同的, 这个与具体硬件相关, 需要硬件上的支持! 上图展示的便是开发板 WM8960 声卡所支持的配置项, 包括 Playback 播放和 Capture 录音, 左上角 View 处提示:

View: F3:[Playback] F4: Capture F5: All

表示当前显示的是[Playback]的配置项, 通过 F4 按键切换为 Capture、或按 F5 显示所有配置项。

Tips: 在终端按下 F4 或 F5 按键时, 可能会直接退出配置界面, 这个原因可能是 F4 或 F5 快捷键被其它程序给占用了, 大家可以试试在 Ubuntu 系统下使用 ssh 远程登录开发板, 然后在 Ubuntu ssh 终端执行 alsamixer 程序, 笔者测试 F4、F5 都是正常的。

左上角 Item 处提示:

Item: Headphone [dB gain: -8.00, -8.00]

表示当前选择的是 Headphone 配置项, 可通过键盘上的 LEFT (向左) 和 RIGHT (向右) 按键切换到其它配置项。当用户对配置项进行修改时, 只能修改被选中的配置项, 而中括号[dB gain: -7.00, -7.00]中的内容显示了该配置项当前的配置值。

上图中只是列出了其中一部分, 还有一部分配置项并未显示出来, 可以通过左右按键移动查看到其余配置项。WM8960 声卡所支持的配置项特别多, 包括播放音量、耳机音量、喇叭音量、capture 录音音量、通道使能、ZC、AC、DC、ALC、3D 等, 配置项特别多, 很多配置项笔者也不懂。以下列出了其中一些配置项及其说明:

Headphone: 耳机音量, 使用上(音量增加)、下(音量降低)按键可以调节播放时耳机输出的音量大小, 当然可以通过 Q(左声道音量增加)、Z(左声道音量降低)按键单独调节左声道音量或通过 E(右声道音量增加)、C(右声道音量降低)按键单独调节右声道音量。

Headphone Playback ZC: 耳机播放 ZC(交流), 通过 M 键打开或关闭 ZC。

Speaker: 喇叭播放音量, 音量调节方法与 Headphon 相同。

Speaker AC: 喇叭 ZC, 通过上下按键可调节大小。

Speaker DC: 喇叭 DC, 通过上下按键可调节大小。

Speaker Playback ZC: 喇叭播放 ZC, 通过 M 键打开或关闭 ZC。

Playback: 播放音量, 播放音量作用于喇叭、也能作用于耳机, 能同时控制喇叭和耳机的输出音量。调节方法与 Headphon 相同。

Capture: 采集音量, 也就是录音时的音量大小, 调节方法与 Headphon 相同。

其它的配置项就不再介绍了, 笔者也看不懂, 后面会用到时再给大家解释!

开发板出厂系统中有一个配置文件 `/var/lib/alsa/asound.state`, 这其实就是 WM8960 声卡的配置文件, 每当开发板启动进入系统时会自动读取该文件加载声卡配置; 而每次系统关机时, 又会将声卡当前的配置写入到该文件中进行保存, 以便下一次启动时加载。加载与保存操作其实是通过 `alsactl` 工具完成的, 稍后向大家介绍。

alsactl

配置好声卡之后, 如果直接关机, 下一次重启之后之前的设置都会消失, 必须要重新设置, 所以我们需要对配置进行保存, 如何保存呢? 可通过 `alsactl` 工具完成。

使用 `alsactl` 工具可以将当前声卡的配置保存在一个文件中, 这个文件默认是 `/var/lib/alsa/asound.state`, 譬如使用 `alsactl` 工具将声卡配置保存在该文件中:

```
alsactl -f /var/lib/alsa/asound.state store
```

`-f` 选项指定保存在哪一个文件中, 当然也可以不用指定, 如果不指定则使用 `alsactl` 默认的配置文件的 `/var/lib/alsa/asound.state`, `store` 表示保存配置。保存成功以后就会生成 `/var/lib/alsa/asound.state` 这个文件, `asound.state` 文件中保存了声卡的各种设置信息, 大家可以打开此文件查看里面的内容, 如下所示:

```
state.wm8960audio {
    control.1 {
        iface MIXER
        name 'Capture Volume'
        value.0 54
        value.1 54
        comment {
            access 'read write'
            type INTEGER
            count 2
            range '0 - 63'
            dbmin -1725
            dbmax 3000
            dbvalue.0 2325
            dbvalue.1 2325
        }
    }
    control.2 {
        iface MIXER
        name 'Capture Volume ZC Switch'
        value.0 0
        value.1 0
        comment {
            access 'read write'
            type INTEGER
            count 2
            range '0 - 1'
        }
    }
}
```

图 28.4.4 asound.state 文件部分内容

除了保存配置之外,还可以加载配置,譬如使用/var/lib/alsa/asound.state 文件中的配置信息来配置声卡,可执行如下命令:

```
alsactl -f /var/lib/alsa/asound.state restore
```

restore 表示加载配置,读取/var/lib/alsa/asound.state 文件中的配置信息并对声卡进行设置。关于 alsactl 的详细使用方法,可以执行"alsactl -h"进行查看。

开发板出厂系统每次开机启动时便会自动从/var/lib/alsa/asound.state 文件中读取配置信息并配置声卡,而每次关机时(譬如执行 reset 或 poweroff 命令)又会将声卡当前的配置写入到该文件中进行保存,以便下一次启动时加载。其实也就是在系统启动(或关机)时通过 alsactl 工具加载(或保存)配置。

amixer

amixer 工具也是一个声卡配置工具,与 alsamixer 功能相同,区别在于,alsamixer 是一个基于字符图形化的配置工具、而 amixer 不是图形化配置工具,直接使用命令行配置即可,详细地用法大家可以执行"amixer --help"命令查看,下面笔者简单地提一下该工具怎么用:

执行命令"amixer scontrols"可以查看到有哪些配置项,如下所示:

```
root@ATK-IMX6U:~# amixer scontrols
Simple mixer control 'Headphone',0
Simple mixer control 'Headphone Playback ZC',0
Simple mixer control 'Speaker',0
Simple mixer control 'Speaker AC',0
Simple mixer control 'Speaker DC',0
Simple mixer control 'Speaker Playback ZC',0
Simple mixer control 'PCM Playback -6dB',0
Simple mixer control 'Mono Output Mixer Left',0
Simple mixer control 'Mono Output Mixer Right',0
Simple mixer control 'Playback',0
Simple mixer control 'Capture',0
Simple mixer control '3D',0
Simple mixer control '3D Filter Lower Cut-Off',0
Simple mixer control '3D Filter Upper Cut-Off',0
Simple mixer control 'ADC Data Output Select',0
Simple mixer control 'ADC High Pass Filter',0
Simple mixer control 'ADC PCM',0
Simple mixer control 'ADC Polarity',0
Simple mixer control 'ALC Attack',0
Simple mixer control 'ALC Decay',0
Simple mixer control 'ALC Function',0
Simple mixer control 'ALC Hold Time',0
Simple mixer control 'ALC Max Gain',0
Simple mixer control 'ALC Min Gain',0
Simple mixer control 'ALC Mode',0
Simple mixer control 'ALC Target',0
Simple mixer control 'DAC Deemphasis',0
Simple mixer control 'DAC Polarity',0
Simple mixer control 'Left Boost Mixer LINPUT1',0
Simple mixer control 'Left Boost Mixer LINPUT2',0
Simple mixer control 'Left Boost Mixer LINPUT3',0
Simple mixer control 'Left Input Boost Mixer LINPUT1',0
Simple mixer control 'Left Input Boost Mixer LINPUT2',0
Simple mixer control 'Left Input Boost Mixer LINPUT3',0
Simple mixer control 'Left Input Mixer Boost',0
Simple mixer control 'Left Output Mixer Boost Bypass',0
Simple mixer control 'Left Output Mixer LINPUT3',0
```

图 28.4.5 查看有哪些配置项

从打印信息可知, 这里打印出来的配置项与 alsamixer 配置界面中所看到的配置项是相同的, 那如何进去配置呢? 不同的配置项对应的配置方法(配置值或值类型)是不一样的, 可以先使用命令"amixer scontent"查看配置项的说明, 如下所示:

```
amixer scontent
```

```

root@ATK-IMX6U:~# amixer scontents
Simple mixer control 'Headphone',0
  Capabilities: pvolume
  Playback channels: Front Left - Front Right
  Limits: Playback 0 - 127
  Mono:
  Front Left: Playback 115 [91%] [-6.00dB]
  Front Right: Playback 115 [91%] [-6.00dB]
Simple mixer control 'Headphone Playback ZC',0
  Capabilities: pswitch
  Playback channels: Front Left - Front Right
  Mono:
  Front Left: Playback [on]
  Front Right: Playback [on]
Simple mixer control 'Speaker',0
  Capabilities: pvolume
  Playback channels: Front Left - Front Right
  Limits: Playback 0 - 127
  Mono:
  Front Left: Playback 118 [93%] [-3.00dB]
  Front Right: Playback 118 [93%] [-3.00dB]
Simple mixer control 'Speaker AC',0
  Capabilities: volume volume-joined
  Playback channels: Mono
  Capture channels: Mono
  Limits: 0 - 5
  Mono: 1 [20%]
Simple mixer control 'Speaker DC',0
  Capabilities: volume volume-joined
  Playback channels: Mono
  Capture channels: Mono

```

图 28.4.6 每一个配置项的配置说明

“Headphone”配置项用于设置耳机音量，音量可调节范围为 0-127，当前音量为 115（左右声道都是 115）；有些设置项是 bool 类型，只有 on 和 off 两种状态。

譬如将耳机音量左右声道都设置为 100，可执行如下命令进行设置：

```
amixer sset Headphone 100,100
```

譬如打开或关闭 Headphone Playback ZC：

```
amixer sset "Headphone Playback ZC" off      #关闭 ZC
```

```
amixer sset "Headphone Playback ZC" on      #打开 ZC
```

以上给大家举了两个例子，配置方法还是很简单地！

arecord

arecord 工具是一个用于录音测试的应用程序，这里笔者简单地给大家介绍一下工具的使用方法，详细的使用方法大家可以执行“arecord --help”命令查看帮助信息。譬如使用 arecord 录制一段 10 秒钟的音频，可以执行如下命令：

```
arecord -f cd -d 10 test.wav
```

```

root@ATK-IMX6U:~# arecord -f cd -d 10 test.wav
Recording WAVE 'test.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo

```

图 28.4.7 使用 arecord 工具录音

-f 选项指定音频格式，cd 则表示 cd 级别音频，也就是“16 bit little endian, 44100, stereo”；-d 选项指定音频录制时间长度，单位是秒；test.wav 指定音频数据保存的文件。当录制完成之后，会生成 test.wav 文件，接着我们可以使用 aplay 工具播放这一段音频。

以上给大家介绍了 alsa-utils 提供的几个测试音频、配置声卡的工具，当然，本文也只是进行了简单地介绍，更加详细的使用方法还需要大家自己查看帮助信息。

28.5 编写一个简单地 alsa-lib 应用程序

本小节开始, 我们来学习如何基于 alsa-lib 编写音频应用程序, alsa-lib 提供的库函数也别多, 笔者肯定不会全部给大家介绍, 只介绍基础的使用方法, 关于更加深入、更加详细的使用方法需要大家自己去研究、学习。

对于 alsa-lib 库的使用, ALSA 提供了一些参考资料来帮助应用程序开发人员快速上手 alsa-lib、基于 alsa-lib 进行应用编程, 以下笔者给出了链接:

https://users.suse.com/~mana/alsa090_howto.html

<https://www.alsa-project.org/alsa-doc/alsa-lib/examples.html>

第一份文档向用户介绍了如何使用 alsa-lib 编写简单的音频应用程序, 包括 PCM 播放音频、PCM 录音等, 笔者也是参考了这份文档来编写本章教程, 对应初学者, 建议大家看一看。

第二个链接地址是 ALSA 提供的一些示例代码, 如下所示:

ALSA project - the C library reference

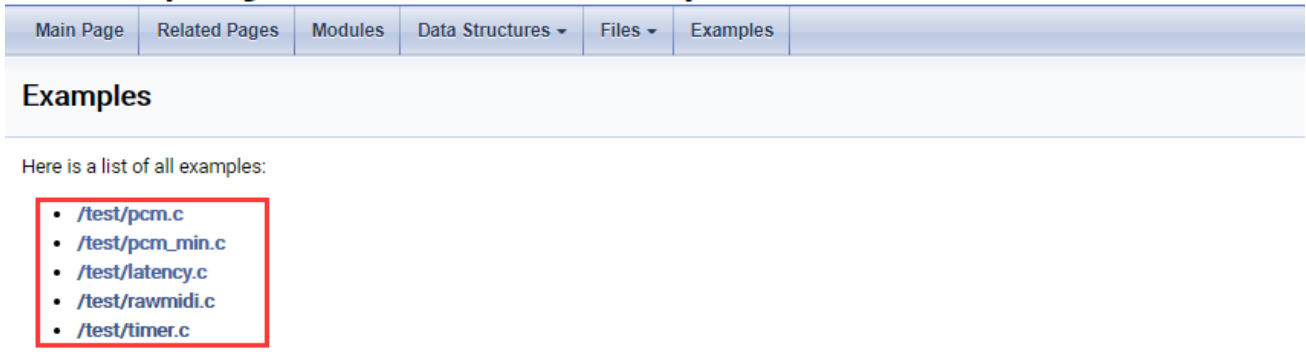


图 28.5.1 ALSA 提供的参考代码

点击对应源文件即可查看源代码。

以上便是 ALSA 提供的帮助文档以及参考代码, 链接地址已经给出了, 大家有兴趣可以看一下。

本小节笔者将向大家介绍如何基于 alsa-lib 编写一个简单地音频应用程序, 譬如播放音乐、录音等; 但在此之前, 首先我们需要先来了解一些基本的概念, 为后面的学习打下一个坚实的基础!

28.5.1 一些基本概念

主要是与音频相关的基本概念, 因为在 alsa-lib 应用编程中会涉及到这些概念, 所以先给大家进行一个简单地介绍。

样本长度 (Sample)

样本是记录音频数据最基本的单元, 样本长度就是采样位数, 也称为位深度 (Bit Depth、Sample Size、Sample Width)。是指计算机在采集和播放声音文件时, 所使用数字声音信号的二进制位数, 或者说每个采样样本所包含的位数 (计算机对每个通道采样量化时数字比特位数), 通常有 8bit、16bit、24bit 等。

声道数 (channel)

分为单声道(Mono)和双声道/立体声(Stereo)。1 表示单声道、2 表示立体声。

帧 (frame)

帧记录了一个声音单元, 其长度为样本长度与声道数的乘积, 一段音频数据就是由若干帧组成的。

把所有声道中的数据加在一起叫做一帧, 对于单声道: 一帧 = 样本长度 * 1; 双声道: 一帧 = 样本长度 * 2。譬如对于样本长度为 16bit 的双声道来说, 一帧的大小等于: $16 * 2 / 8 = 4$ 个字节。

采样率 (Sample rate)

也叫采样频率, 是指每秒钟采样次数, 该次数是针对帧而言。譬如常见的采样率有:

8KHz - 电话所用采样率

22.05KHz - FM 调频广播所用采样率

44.1KHz - 音频 CD, 也常用于 MPEG-1 音频 (VCD、SVCD、MP3) 所用采样率

48KHz - miniDV、数字电视、DVD、DAT、电影和专业音频所用的数字声音所用采样率。

交错模式 (interleaved)

交错模式是一种音频数据的记录方式, 分为交错模式和非交错模式。在交错模式下, 数据以连续帧的形式存放, 即首先记录完帧 1 的左声道样本和右声道样本 (假设为立体声格式), 再记录帧 2 的左声道样本和右声道样本。而在非交错模式下, 首先记录的是一个周期内所有帧的左声道样本, 再记录右声道样本, 数据是以连续通道的方式存储。不过多数情况下, 我们一般都是使用交错模式。

周期 (period)

周期是音频设备处理 (读、写) 数据的单位, 换句话说, 也就是音频设备读写数据的单位是周期, 每一次读或写一个周期的数据, 一个周期包含若干个帧; 譬如周期的大小为 1024 帧, 则表示音频设备进行一次读或写操作的数据量大小为 1024 帧, 假设一帧为 4 个字节, 那么也就是 $1024 * 4 = 4096$ 个字节数据。

一个周期其实就是两次硬件中断之间的帧数, 音频设备每处理 (读或写) 完一个周期的数据就会产生一个中断, 所以两个中断之间相差一个周期, 关于中断的问题, 稍后再向大家介绍!

缓冲区 (buffer)

数据缓冲区, 一个缓冲区包含若干个周期, 所以 buffer 是由若干个周期所组成的一块空间。下面一张图直观地表示了 buffer、period、frame、sample (样本长度) 之间的关系, 假设一个 buffer 包含 4 个周期、而一个周期包含 1024 帧、一帧包含两个样本 (左、右两个声道):

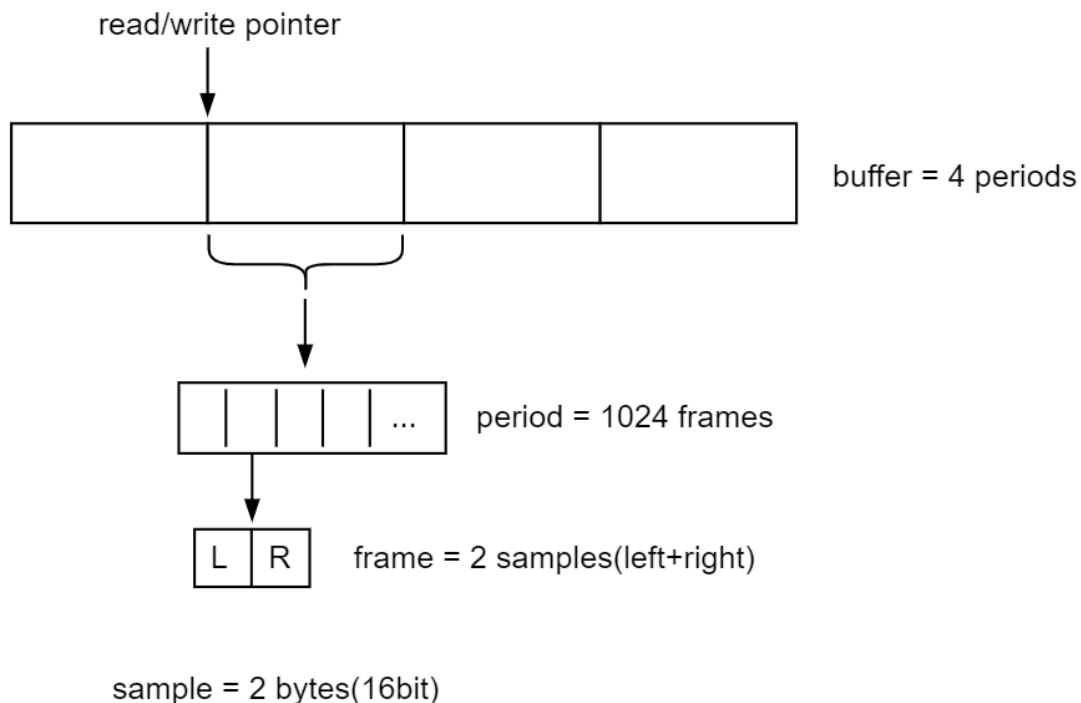


图 28.5.2 buffer/period/frame/sample 之间的关系示例图

音频设备底层驱动程序使用 DMA 来搬运数据, 这个 buffer 中有 4 个 period, 每当 DMA 搬运完一个 period 的数据就会触发一次中断, 因此搬运整个 buffer 中的数据将产生 4 次中断。ALSA 为什么这样做? 直

接把整个 buffer 中的数据一次性搬运过去岂不是更快? 情况并非如此, 我们没有考虑到一个很重要的问题, 那就是延迟; 如果数据缓存区 buffer 很大, 一次传输整个 buffer 中的数据可能会导致不可接受的延迟, 因为一次搬运的数据量越大, 所花费的时间就越长, 那么必然会导致数据从传输开始到发出声音 (以播放为例) 这个过程所经历的时间就会越长, 这就是延迟。为了解决这个问题, ALSA 把缓存区拆分成多个周期, 以周期为传输单元进行传输数据。

所以, 周期不宜设置过大, 周期过大会导致延迟过高; 但周期也不能太小, 周期太小会导致频繁触发中断, 这样会使得 CPU 被频繁中断而无法执行其它的任务, 使得效率降低! 所以, 周期大小要合适, 在延迟可接受的情况下, 尽量设置大一些, 不过这个需要根据实际应用场合而定, 有些应用场合, 可能要求低延迟、实时性高, 但有些应用场合没有这种需求。

数据之间的传输

这里再介绍一下数据之间传输的问题, 这个问题很重要, 大家一定要理解, 这样会更好的帮助我们理解代码、理解代码的逻辑。

● PCM 播放情况下

在播放情况下, buffer 中存放了需要播放的 PCM 音频数据, 由应用程序向 buffer 中写入音频数据, buffer 中的音频数据由 DMA 传输给音频设备进行播放, 所以应用程序向 buffer 写入数据、音频设备从 buffer 读取数据, 这就是 buffer 中数据的传输情况。

图 28.5.2 中标识有 read pointer 和 write pointer 指针, write pointer 指向当前应用程序写 buffer 的位置, read pointer 指向当前音频设备读 buffer 的位置。在数据传输之前 (播放之前), buffer 缓冲区是没有数据的, 此时 write/read pointer 均指向了 buffer 的起始位置, 也就是第一个周期的起始位置, 如下所示:

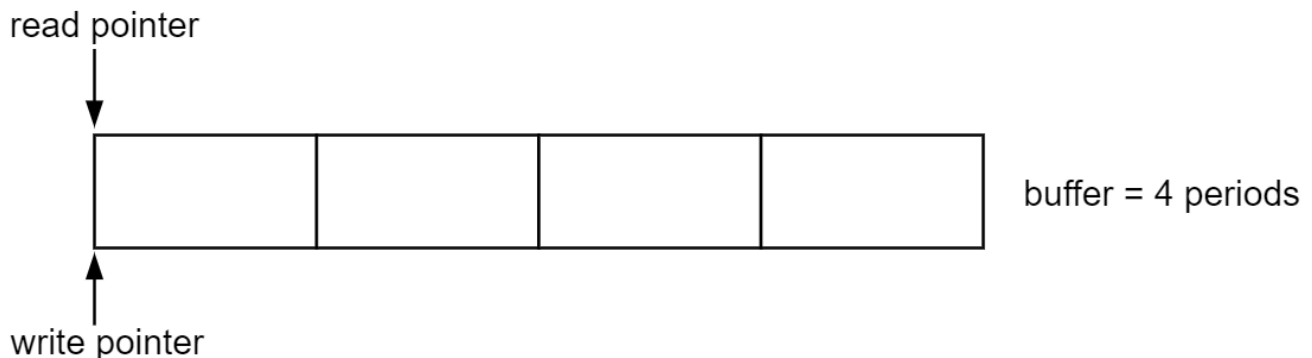


图 28.5.3 pointer 指向 buffer 起始位置

应用程序向 buffer 写入多少帧数据, 则 write pointer 指针向前移动多少帧, 当应用程序向 buffer 中写入一个周期的数据时, write pointer 指针将向前移动一个周期; 接着再写入一个周期, 指针再向前移动一个周期, 以此类推! 当 write pointer 移动到 buffer 末尾时, 又会回到 buffer 的起始位置, 以此循环! 所以由此可知, 这是一个环形缓冲区。

以上是应用程序写 buffer 的一个过程, 接着再来看看音频设备读 buffer (播放) 的过程。在播放开始之前, read pointer 指向了 buffer 的起始位置, 也就是第一个周期的起始位置。音频设备每次只播放一个周期的数据 (读取一个周期), 每一次都是从 read pointer 所指位置开始读取; 每读取一个周期, read pointer 指针向前移动一个周期, 同样, 当 read pointer 指针移动到 buffer 末尾时, 又会回到 buffer 的起始位置, 以此构成一个循环!

应用程序需要向 buffer 中写入音频数据, 音频设备才能读取数据进行播放, 如果 read pointer 所指向的周期并没有填充音频数据, 则无法播放! 当 buffer 数据满时, 应用程序将不能再写入数据, 否则就会覆盖之前的数据, 必须要等待音频设备播放完一个周期, 音频设备每播放完一个周期, 这个周期就变成空闲状态了, 此时应用程序就可以写入一个周期的数据以填充这个空闲周期。

● PCM 录音情况下

在录音情况下, buffer 中存放了音频设备采集到的音频数据(外界模拟声音通过 ADC 转为数字声音), 由音频设备向 buffer 中写入音频数据(DMA 搬运), 而应用程序从 buffer 中读取数据, 所以音频设备向 buffer 写入数据、应用程序从 buffer 读取数据, 这就是录音情况下 buffer 中数据的传输情况。

回到图 28.5.2 中, 此时 write pointer 指向音频设备写 buffer 的位置、read pointer 指向应用程序读 buffer 的位置。在录音开始之前, buffer 缓冲区是没有数据的, 此时 write/read pointer 均指向了 buffer 的起始位置, 也就是第一个周期的起始位置, 如图 28.5.3 中所示。

音频设备向 buffer 写入多少帧数据, 则 write pointer 指针向前移动多少帧, 音频设备每次只采集一个周期, 将采集到的数据写入 buffer 中, 从 write pointer 所指位置开始写入; 当音频设备向 buffer 中写入一个周期的数据时, write pointer 指针将向前移动一个周期; 接着再写入一个周期, 指针再向前移动一个周期, 以此类推! 当 write pointer 移动到 buffer 末尾时, 又会回到 buffer 的起始位置, 以此构成循环!

以上是音频设备写 buffer 的一个过程, 接着再来看看应用程序读 buffer 的过程。在录音开始之前, read pointer 指向了 buffer 的起始位置, 也就是第一个周期的起始位置。同样, 应用程序从 buffer 读取了多少帧数据, 则 read pointer 指针向前移动多少帧; 从 read pointer 所指位置开始读取, 当 read pointer 指针移动到 buffer 末尾时, 又会回到 buffer 的起始位置, 以此构成一个循环!

音频设备需要向 buffer 中写入音频数据, 应用程序才能从 buffer 中读取数据(录音), 如果 read pointer 所指向的周期并没有填充音频数据, 则无法读取! 当 buffer 中没有数据时, 需要等待音频设备向 buffer 中写入数据, 音频设备每次写入一个周期, 当应用程序读取完这个周期的数据后, 这个周期又变成了空闲周期, 需要等待音频设备写入数据。

Over and Under Run

当一个声卡处于工作状态时, 环形缓冲区 buffer 中的数据总是连续地在音频设备和应用程序缓存区间传输, 如下图所示:

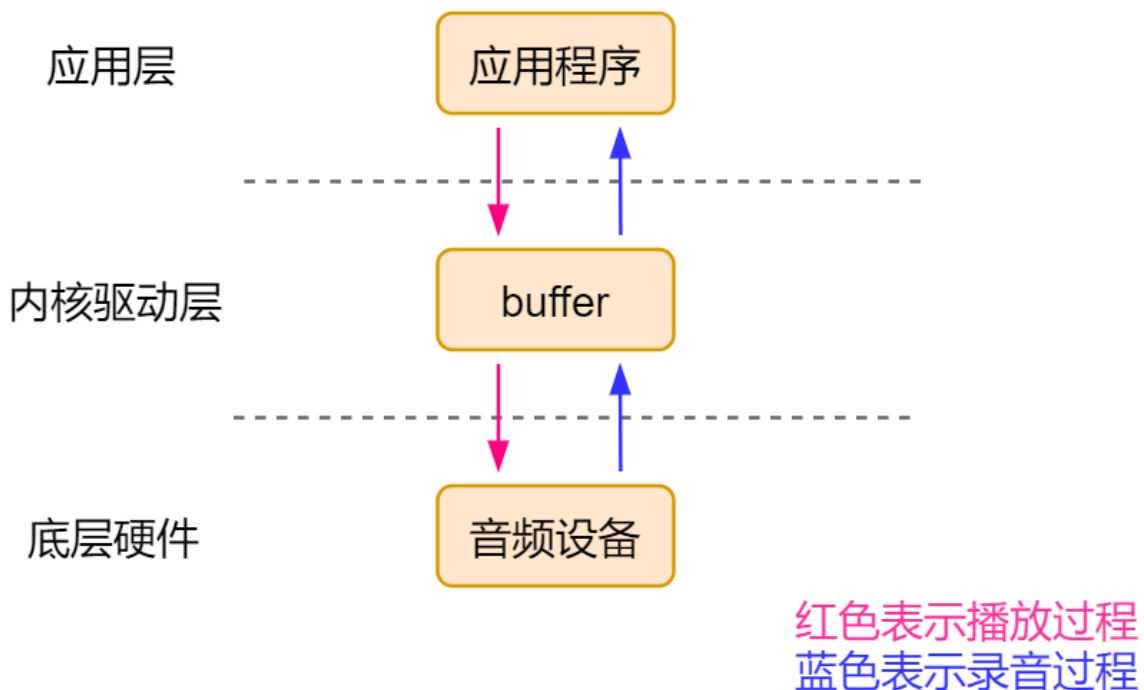


图 28.5.4 buffer 中数据的传输

上图展示了声卡在工作状态下, buffer 中数据的传输情况, 总是连续地在音频设备和应用程序缓存区间传输, 但事情并不总是那么完美、也会出现有例外; 譬如在录音例子中, 如果应用程序读取数据不够快, 环形缓冲区 buffer 中的数据已经被音频设备写满了、而应用程序还未来得及读走, 那么数据将会被覆盖; 这种数据的丢失被称为 overrun。在播放例子中, 如果应用程序写入数据到环形缓冲区 buffer 中的速度不够快,

缓存区将会“饿死”（缓冲区中无数据可播放）；这样的错误被称为 `underrun`（欠载）。在 ALSA 文档中，将这两种情形统称为“`XRUN`”，适当地设计应用程序可以最小化 `XRUN` 并且可以从中恢复过来。

28.5.2 打开 PCM 设备

从本小节开始，将正式介绍如何编写一个音频应用程序，首先我们需要在应用程序中包含 `alsa-lib` 库的头文件 `<alsa/asoundlib.h>`，这样才能在应用程序中调用 `alsa-lib` 库函数以及使用相关宏。

第一步需要打开 PCM 设备，调用函数 `snd_pcm_open()`，该函数原型如下所示：

```
int snd_pcm_open(snd_pcm_t **pcmp, const char *name, snd_pcm_stream_t stream, int mode)
```

该函数一共有 4 个参数，如下所示：

- **pcmp**: `snd_pcm_t` 用于描述一个 PCM 设备，所以一个 `snd_pcm_t` 对象表示一个 PCM 设备；`snd_pcm_open` 函数会打开参数 `name` 所指定的设备，实例化 `snd_pcm_t` 对象，并将对象的指针（也就是 PCM 设备的句柄）通过 `pcmp` 返回出来。
- **name**: 参数 `name` 指定 PCM 设备的名字。`alsa-lib` 库函数中使用逻辑设备名而不是设备文件名，命名方式为“`hw:i,j`”，`i` 表示声卡的卡号，`j` 则表示这块声卡上的设备号；譬如“`hw:0,0`”表示声卡 0 上的 PCM 设备 0，在播放情况下，这其实就对应 `/dev/snd/pcmC0D0p`（如果是录音，则对应 `/dev/snd/pcmC0D0c`）。除了使用“`hw:i,j`”这种方式命名之外，还有其它两种常用的命名方式，譬如“`plughw:i,j`”、“`default`”等，关于这些名字的不同，本章最后再向大家进行简单地介绍，这里暂时先不去理会这个问题。
- **stream**: 参数 `stream` 指定流类型，有两种不同类型：`SND_PCM_STREAM_PLAYBACK` 和 `SND_PCM_STREAM_CAPTURE`；`SND_PCM_STREAM_PLAYBACK` 表示播放，`SND_PCM_STREAM_CAPTURE` 则表示采集。
- **mode**: 最后一个参数 `mode` 指定了 `open` 模式，通常情况下，我们会将其设置为 0，表示默认打开模式，默认情况下使用阻塞方式打开设备；当然，也可将其设置为 `SND_PCM_NONBLOCK`，表示以非阻塞方式打开设备。

设备打开成功，`snd_pcm_open` 函数返回 0；打开失败，返回一个小于 0 的错误编号，可以使用 `alsa-lib` 提供的库函数 `snd_strerror()` 来得到对应的错误描述信息，该函数与 C 库函数 `strerror()` 用法相同。

与 `snd_pcm_open` 相对应的是 `snd_pcm_close()`，函数 `snd_pcm_close()` 用于关闭 PCM 设备，函数原型如下所示：

```
int snd_pcm_close(snd_pcm_t *pcm);
```

使用示例：

调用 `snd_pcm_open()` 函数打开声卡 0 的 PCM 播放设备 0：

```
snd_pcm_t *pcm_handle = NULL;
int ret;

ret = snd_pcm_open(&pcm_handle, "hw:0,0", SND_PCM_STREAM_PLAYBACK, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_open error: %s\n", snd_strerror(ret));
    return -1;
}
```

28.5.3 设置硬件参数

打开 PCM 设备之后,接着我们需要对设备进行设置,包括硬件配置和软件配置。软件配置就不再介绍了,使用默认配置即可!我们主要是对硬件参数进行配置,譬如采样率、声道数、格式、访问类型、period 周期大小、buffer 大小等。

实例化 `snd_pcm_hw_params_t` 对象

alsa-lib 使用 `snd_pcm_hw_params_t` 数据类型来描述 PCM 设备的硬件配置参数,在配置参数之前,我们需要实例化一个 `snd_pcm_hw_params_t` 对象,使用 `snd_pcm_hw_params_malloc` 或 `snd_pcm_hw_params_alloca()` 来实例化一个 `snd_pcm_hw_params_t` 对象,如下所示:

```
snd_pcm_hw_params_t *hwparams = NULL;
```

```
snd_pcm_hw_params_malloc(&hwparams);
```

或

```
snd_pcm_hw_params_alloca(&hwparams);
```

它们之间的区别也就是 C 库函数 `malloc` 和 `alloca` 之间的区别。当然,你也可以直接使用 `malloc()` 或 `alloca()` 来分配一个 `snd_pcm_hw_params_t` 对象,亦或者直接定义全局变量或栈自动变量。与 `snd_pcm_hw_params_malloc/snd_pcm_hw_params_alloca` 相对应的是 `snd_pcm_hw_params_free`,`snd_pcm_hw_params_free()` 函数用于释放 `snd_pcm_hw_params_t` 对象占用的内存空间。函数原型如下所示:

```
void snd_pcm_hw_params_free(snd_pcm_hw_params_t *obj)
```

初始化 `snd_pcm_hw_params_t` 对象

`snd_pcm_hw_params_t` 对象实例化完成之后,接着我们需要对其进行初始化操作,调用 `snd_pcm_hw_params_any()` 对 `snd_pcm_hw_params_t` 对象进行初始化操作,调用该函数会使用 PCM 设备当前的配置参数去初始化 `snd_pcm_hw_params_t` 对象,如下所示:

```
snd_pcm_hw_params_any(pcm_handle, hwparams);
```

第一个参数为 PCM 设备的句柄,第二个参数传入 `snd_pcm_hw_params_t` 对象的指针。

对硬件参数进行设置

alsa-lib 提供了一系列的 `snd_pcm_hw_params_set_xxx` 函数用于设置 PCM 设备的硬件参数,同样也提供了一系列的 `snd_pcm_hw_params_get_xxx` 函数用于获取硬件参数。

(1) 设置 access 访问类型: `snd_pcm_hw_params_set_access()`

调用 `snd_pcm_hw_params_set_access` 设置访问类型,其函数原型如下所示:

```
int snd_pcm_hw_params_set_access(snd_pcm_t *pcm,
    snd_pcm_hw_params_t * params,
    snd_pcm_access_t access
)
```

参数 `access` 指定设备的访问类型,是一个 `snd_pcm_access_t` 类型常量,这是一个枚举类型,如下所示:

```
enum snd_pcm_access_t {
    SND_PCM_ACCESS_MMAP_INTERLEAVED = 0, //mmap access with simple interleaved channels
    SND_PCM_ACCESS_MMAP_NONINTERLEAVED, //mmap access with simple non interleaved
channels
    SND_PCM_ACCESS_MMAP_COMPLEX, //mmap access with complex placement
    SND_PCM_ACCESS_RW_INTERLEAVED, //snd_pcm_readi/snd_pcm_writei access
    SND_PCM_ACCESS_RW_NONINTERLEAVED, //snd_pcm_readn/snd_pcm_writen access
}
```

```
SND_PCM_ACCESS_LAST = SND_PCM_ACCESS_RW_NONINTERLEAVED
};
```

通常，将访问类型设置为 `SND_PCM_ACCESS_RW_INTERLEAVED`，交错访问模式，通过 `snd_pcm_readi/snd_pcm_writei` 对 PCM 设备进行读/写操作。

函数调用成功返回 0；失败将返回一个小于 0 的错误码，可通过 `snd_strerror()` 函数获取错误描述信息。

使用示例：

```
ret = snd_pcm_hw_params_set_access(pcm_handle, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
if (0 > ret)
    fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
```

(2) 设置数据格式: `snd_pcm_hw_params_set_format()`

调用 `snd_pcm_hw_params_set_format()` 函数设置 PCM 设备的数据格式，函数原型如下所示：

```
int snd_pcm_hw_params_set_format(snd_pcm_t *pcm,
    snd_pcm_hw_params_t *params,
    snd_pcm_format_t format
)
```

参数 `format` 指定数据格式，该参数是一个 `snd_pcm_format_t` 类型常量，这是一个枚举类型，如下所示：

```
enum snd_pcm_format_t {
    SND_PCM_FORMAT_UNKNOWN = -1,
    SND_PCM_FORMAT_S8 = 0,
    SND_PCM_FORMAT_U8,
    SND_PCM_FORMAT_S16_LE,
    SND_PCM_FORMAT_S16_BE,
    SND_PCM_FORMAT_U16_LE,
    SND_PCM_FORMAT_U16_BE,
    SND_PCM_FORMAT_S24_LE,
    SND_PCM_FORMAT_S24_BE,
    SND_PCM_FORMAT_U24_LE,
    SND_PCM_FORMAT_U24_BE,
    SND_PCM_FORMAT_S32_LE,
    SND_PCM_FORMAT_S32_BE,
    SND_PCM_FORMAT_U32_LE,
    SND_PCM_FORMAT_U32_BE,
    SND_PCM_FORMAT_FLOAT_LE,
    SND_PCM_FORMAT_FLOAT_BE,
    SND_PCM_FORMAT_FLOAT64_LE,
    SND_PCM_FORMAT_FLOAT64_BE,
    SND_PCM_FORMAT_IEC958_SUBFRAME_LE,
    SND_PCM_FORMAT_IEC958_SUBFRAME_BE,
    SND_PCM_FORMAT_MU_LAW,
    SND_PCM_FORMAT_A_LAW,
    SND_PCM_FORMAT_IMA_ADPCM,
    SND_PCM_FORMAT_MPEG,
    SND_PCM_FORMAT_GSM,
```

```
SND_PCM_FORMAT_S20_LE,
SND_PCM_FORMAT_S20_BE,
SND_PCM_FORMAT_U20_LE,
SND_PCM_FORMAT_U20_BE,
SND_PCM_FORMAT_SPECIAL = 31,
SND_PCM_FORMAT_S24_3LE = 32,
SND_PCM_FORMAT_S24_3BE,
SND_PCM_FORMAT_U24_3LE,
SND_PCM_FORMAT_U24_3BE,
SND_PCM_FORMAT_S20_3LE,
SND_PCM_FORMAT_S20_3BE,
SND_PCM_FORMAT_U20_3LE,
SND_PCM_FORMAT_U20_3BE,
SND_PCM_FORMAT_S18_3LE,
SND_PCM_FORMAT_S18_3BE,
SND_PCM_FORMAT_U18_3LE,
SND_PCM_FORMAT_U18_3BE,
SND_PCM_FORMAT_G723_24,
SND_PCM_FORMAT_G723_24_1B,
SND_PCM_FORMAT_G723_40,
SND_PCM_FORMAT_G723_40_1B,
SND_PCM_FORMAT_DSD_U8,
SND_PCM_FORMAT_DSD_U16_LE,
SND_PCM_FORMAT_DSD_U32_LE,
SND_PCM_FORMAT_DSD_U16_BE,
SND_PCM_FORMAT_DSD_U32_BE,
SND_PCM_FORMAT_LAST = SND_PCM_FORMAT_DSD_U32_BE,
SND_PCM_FORMAT_S16 = SND_PCM_FORMAT_S16_LE,
SND_PCM_FORMAT_U16 = SND_PCM_FORMAT_U16_LE,
SND_PCM_FORMAT_S24 = SND_PCM_FORMAT_S24_LE,
SND_PCM_FORMAT_U24 = SND_PCM_FORMAT_U24_LE,
SND_PCM_FORMAT_S32 = SND_PCM_FORMAT_S32_LE,
SND_PCM_FORMAT_U32 = SND_PCM_FORMAT_U32_LE,
SND_PCM_FORMAT_FLOAT = SND_PCM_FORMAT_FLOAT_LE,
SND_PCM_FORMAT_FLOAT64 = SND_PCM_FORMAT_FLOAT64_LE,
SND_PCM_FORMAT_IEC958_SUBFRAME = SND_PCM_FORMAT_IEC958_SUBFRAME_LE,
SND_PCM_FORMAT_S20 = SND_PCM_FORMAT_S20_LE,
SND_PCM_FORMAT_U20 = SND_PCM_FORMAT_U20_LE
};
```

用的最多的格式是 `SND_PCM_FORMAT_S16_LE`，有符号 16 位、小端模式。当然，音频设备不一定支持用户所指定的格式，在此之前，用户可以调用 `snd_pcm_hw_params_test_format()` 函数测试 PCM 设备是否支持某种格式，如下所示：

```
if (snd_pcm_hw_params_test_format(pcm_handle, hwparams, SND_PCM_FORMAT_S16_LE)) {
```

```
// 返回一个非零值 表示不支持该格式
}
else {
    // 返回 0 表示支持
}
```

(3)设置声道数: `snd_pcm_hw_params_set_channels()`

调用 `snd_pcm_hw_params_set_channels()`函数设置 PCM 设备的声道数, 函数原型如下所示:

```
int snd_pcm_hw_params_set_channels(snd_pcm_t *pcm,
    snd_pcm_hw_params_t *params,
    unsigned int val
)
```

参数 `val` 指定声道数量, `val=2` 表示双声道, 也就是立体声。函数调用成功返回 0, 失败返回小于 0 的错误码。

使用示例:

```
ret = snd_pcm_hw_params_set_channels(pcm_handle, hwparams, 2);
if (0 > ret)
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
```

(4)设置采样率大小: `snd_pcm_hw_params_set_rate()`

调用 `snd_pcm_hw_params_set_rate` 设置采样率大小, 其函数原型如下所示:

```
int snd_pcm_hw_params_set_rate(snd_pcm_t *pcm,
    snd_pcm_hw_params_t *params,
    unsigned int val,
    int dir
)
```

参数 `val` 指定采样率大小, 譬如 44100; 参数 `dir` 用于控制方向, 若 `dir=-1`, 则实际采样率小于参数 `val` 指定的值; `dir=0` 表示实际采样率等于参数 `val`; `dir=1` 表示实际采样率大于参数 `val`。

函数调用成功返回 0; 失败将返回小于 0 的错误码。

使用示例:

```
ret = snd_pcm_hw_params_set_rate(pcm_handle, hwparams, 44100, 0);
if (0 > ret)
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
```

(5)设置周期大小: `snd_pcm_hw_params_set_period_size()`

这里说的周期, 也就是 28.5.1 小节中向大家介绍的周期, 一个周期的大小使用帧来衡量, 譬如一个周期 1024 帧; 调用 `snd_pcm_hw_params_set_period_size()`函数设置周期大小, 其函数原型如下所示:

```
int snd_pcm_hw_params_set_period_size(snd_pcm_t *pcm,
    snd_pcm_hw_params_t *params,
    snd_pcm_uframes_t val,
    int dir
)
```

`alsa-lib` 使用 `snd_pcm_uframes_t` 类型表示帧的数量; 参数 `dir` 与 `snd_pcm_hw_params_set_rate()`函数的 `dir` 参数意义相同。

使用示例 (将周期大小设置为 1024 帧):

```
ret = snd_pcm_hw_params_set_period_size(pcm_handle, hwparams, 1024, 0);
```



```
if (0 > ret)
```

```
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
```

注意, 参数 val 的单位是帧、而不是字节。

(6)设置 buffer 大小: `snd_pcm_hw_params_set_buffer_size()`

调用 `snd_pcm_hw_params_set_buffer_size()` 函数设置 buffer 的大小, 其函数原型如下所示:

```
int snd_pcm_hw_params_set_buffer_size(snd_pcm_t *pcm,
    snd_pcm_hw_params_t *params,
    snd_pcm_uframes_t val
)
```

参数 val 指定 buffer 的大小, 以帧为单位, 通常 buffer 的大小是周期大小的整数倍, 譬如 16 个周期; 但函数 `snd_pcm_hw_params_set_buffer_size()` 是以帧为单位来表示 buffer 的大小, 所以需要转换一下, 譬如将 buffer 大小设置为 16 个周期, 则参数 val 等于 $16 * 1024$ (假设一个周期为 1024 帧) = 16384 帧。

函数调用成功返回 0; 失败返回一个小于 0 的错误码。

使用示例:

```
ret = snd_pcm_hw_params_set_buffer_size(pcm_handle, hwparams, 16*1024);
```

```
if (0 > ret)
```

```
    fprintf(stderr, "snd_pcm_hw_params_set_buffer_size error: %s\n", snd_strerror(ret));
```

除了 `snd_pcm_hw_params_set_buffer_size()` 函数之外, 我们还可以调用 `snd_pcm_hw_params_set_periods()` 函数设置 buffer 大小, 其函数原型如下所示:

```
int snd_pcm_hw_params_set_periods(snd_pcm_t *pcm,
    snd_pcm_hw_params_t *params,
    unsigned int val,
    int dir
)
```

参数 val 指定了 buffer 的大小, 该大小以周期为单位、并不是以帧为单位, 注意区分!

参数 dir 与 `snd_pcm_hw_params_set_rate()` 函数的 dir 参数意义相同。

函数调用成功返回 0; 失败将返回一个小于 0 的错误码。

使用示例:

```
ret = snd_pcm_hw_params_set_periods(pcm_handle, hwparams, 16, 0); //buffer 大小为 16 个周期
```

```
if (0 > ret)
```

```
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
```

(7)安装/加载硬件配置参数: `snd_pcm_hw_params()`

参数设置完成之后, 最后调用 `snd_pcm_hw_params()` 加载/安装配置、将配置参数写入硬件使其生效, 其函数原型如下所示:

```
int snd_pcm_hw_params(snd_pcm_t *pcm, snd_pcm_hw_params_t *params)
```

函数调用成功返回 0, 失败将返回一个小于 0 的错误码。函数 `snd_pcm_hw_params()` 调用之后, 其内部会自动调用 `snd_pcm_prepare()` 函数, PCM 设备的状态被更改为 `SND_PCM_STATE_PREPARED`。

设备有多种不同的状态, `SND_PCM_STATE_PREPARED` 为其中一种, 关于状态的问题, 后面在向大家介绍。调用 `snd_pcm_prepare()` 函数会使得 PCM 设备处于 `SND_PCM_STATE_PREPARED` 状态 (也就是处于一种准备好的状态)。

使用示例:

```
ret = snd_pcm_hw_params(pcm_handle, hwparams);
```

```
if (0 > ret)
```

```
fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
```

28.5.4 读/写数据

接下来就可以进行读/写数据了, 如果是 PCM 播放, 则调用 `snd_pcm_writei()` 函数向播放缓冲区 `buffer` 中写入音频数据; 如果是 PCM 录音, 则调用 `snd_pcm_readi()` 函数从录音缓冲区 `buffer` 中读取数据, 它们的函数原型如下所示:

```
snd_pcm_sframes_t snd_pcm_writei(snd_pcm_t *pcm,
    const void *buffer,
    snd_pcm_uframes_t size
)

snd_pcm_sframes_t snd_pcm_readi(snd_pcm_t *pcm,
    void *buffer,
    snd_pcm_uframes_t size
)
```

参数 `pcm` 为 PCM 设备的句柄; 调用 `snd_pcm_writei()` 函数, 将参数 `buffer` (应用程序的缓冲区) 缓冲区中的数据写入到驱动层的播放环形缓冲区 `buffer` 中, 参数 `size` 指定写入数据的大小, 以帧为单位; 通常情况下, 每次调用 `snd_pcm_writei()` 写入一个周期数据。

调用 `snd_pcm_readi()` 函数, 将从驱动层的录音环形缓冲区 `buffer` 中读取数据到参数 `buffer` 指定的缓冲区中(应用程序的缓冲区), 参数 `size` 指定读取数据的大小, 以帧为单位; 通常情况下, 每次调用 `snd_pcm_readi()` 读取一个周期数据。

Tips: `snd_pcm_writei/snd_pcm_readi` 函数原型中, 参数 `buffer` 指的是应用程序的缓冲区, 不要与驱动层的环形缓冲区搞混了!

`snd_pcm_readi/snd_pcm_writei` 调用成功, 返回实际读取/写入的帧数; 调用失败将返回一个负数错误码。即使调用成功, 实际读取/写入的帧数不一定等于参数 `size` 所指定的帧数, 仅当发生信号或 `XRUN` 时, 返回的帧数可能会小于参数 `size`。

阻塞与非阻塞

调用 `snd_pcm_open()` 打开设备时, 若指定为阻塞方式, 则调用 `snd_pcm_readi/snd_pcm_writei` 以阻塞方式进行读/写。对于 PCM 录音来说, 当 `buffer` 缓冲区中无数据可读时, 调用 `snd_pcm_readi()` 函数将会阻塞, 直到音频设备向 `buffer` 中写入采集到的音频数据; 同理, 对于 PCM 播放来说, 当 `buffer` 缓冲区中的数据满时, 调用 `snd_pcm_writei()` 函数将会阻塞, 直到音频设备从 `buffer` 中读走数据进行播放。

若调用 `snd_pcm_open()` 打开设备时, 指定为非阻塞方式, 则调用 `snd_pcm_readi/snd_pcm_writei` 以非阻塞方式进行读/写。对于 PCM 录音来说, 当 `buffer` 缓冲区中无数据可读时, 调用 `snd_pcm_readi()` 不会阻塞, 而是立即以错误形式返回; 同理, 对于 PCM 播放来说, 当 `buffer` 缓冲区中的数据满时, 调用 `snd_pcm_writei()` 函数也不会阻塞, 而是立即以错误形式返回。

`snd_pcm_readn` 和 `snd_pcm_writen`

`snd_pcm_readi/snd_pcm_writei` 适用于交错模式 (interleaved) 读/写数据, 如果用户设置的访问类型并不是交错模式, 而是非交错模式 (non interleaved), 此时便不可再使用 `snd_pcm_readi/snd_pcm_writei` 进行读写操作了, 而需要使用 `snd_pcm_readn` 和 `snd_pcm_writen` 进行读写。

28.5.5 示例代码之 PCM 播放

通过上小节的一个介绍, 相信大家对 alsa-lib 音频应用编程已经有了基本的认识和理解, 本小节我们来编写一个简单地音乐播放器, 可以播放 WAV 音频文件, 代码笔者已经写好了, 如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->28_alsa-lib->pcm_playback.c](#)。

示例代码 28.5.1 一个简单地 PCM 播放示例程序

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : pcm_playback.c
作者 : 邓涛
版本 : V1.0
描述 : 一个简单地 PCM 播放示例代码--播放 WAV 音频文件
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/20 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <alsa/asoundlib.h>

/*****
宏定义
*****/
#define PCM_PLAYBACK_DEV    "hw:0,0"

/*****
WAV 音频文件解析相关数据结构申明
*****/
typedef struct WAV_RIFF {
    char ChunkID[4];           /* "RIFF" */
    u_int32_t ChunkSize;       /* 从下一个地址开始到文件末尾的总字节数 */
    char Format[4];           /* "WAVE" */
} __attribute__((packed)) RIFF_t;

typedef struct WAV_FMT {
    char Subchunk1ID[4];       /* "fmt " */
    u_int32_t Subchunk1Size;   /* 16 for PCM */
    u_int16_t AudioFormat;      /* PCM = 1 */
    u_int16_t NumChannels;      /* Mono = 1, Stereo = 2, etc. */
    u_int32_t SampleRate;       /* 8000, 44100, etc. */
    u_int32_t ByteRate;         /* = SampleRate * NumChannels * BitsPerSample/8 */
}

```

```

    u_int16_t BlockAlign;          /* = NumChannels * BitsPerSample/8 */
    u_int16_t BitsPerSample;      /* 8bits, 16bits, etc. */
} __attribute__((packed)) FMT_t;
static FMT_t wav_fmt;

typedef struct WAV_DATA {
    char Subchunk2ID[4];          /* "data" */
    u_int32_t Subchunk2Size;      /* data size */
} __attribute__((packed)) DATA_t;

/*****
static 静态全局变量定义
*****/

static snd_pcm_t *pcm = NULL;    //pcm 句柄
static unsigned int buf_bytes;   //应用程序缓冲区的大小（字节为单位）
static void *buf = NULL;        //指向应用程序缓冲区的指针
static int fd = -1;              //指向 WAV 音频文件的文件描述符
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16; //周期数（设备驱动层 buffer 的大小）

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_PLAYBACK_DEV, SND_PCM_STREAM_PLAYBACK, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_PLAYBACK_DEV, snd_strerror(ret));
        return -1;
    }

    /* 实例化 hwparams 对象 */
    snd_pcm_hw_params_malloc(&hwparams);

    /* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
    ret = snd_pcm_hw_params_any(pcm, hwparams);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
        goto err2;
    }
}

```

```
/*  
设置参数  
*/  
/* 设置访问类型: 交错模式 */  
ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);  
if (0 > ret) {  
    fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));  
    goto err2;  
}  
  
/* 设置数据格式: 有符号 16 位、小端模式 */  
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);  
if (0 > ret) {  
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));  
    goto err2;  
}  
  
/* 设置采样率 */  
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, wav_fmt.SampleRate, 0);  
if (0 > ret) {  
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));  
    goto err2;  
}  
  
/* 设置声道数: 双声道 */  
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, wav_fmt.NumChannels);  
if (0 > ret) {  
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));  
    goto err2;  
}  
  
/* 设置周期大小: period_size */  
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);  
if (0 > ret) {  
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));  
    goto err2;  
}  
  
/* 设置周期数 (驱动层 buffer 的大小) : periods */  
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);  
if (0 > ret) {  
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));  
    goto err2;  
}
```

```
    }

    /* 使配置生效 */
    ret = snd_pcm_hw_params(pcm, hwparams);
    snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
        goto err1;
    }

    buf_bytes = period_size * wav_fmt.BlockAlign; //变量赋值, 一个周期的字节大小
    return 0;

err2:
    snd_pcm_hw_params_free(hwparams); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    return -1;
}

static int open_wav_file(const char *file)
{
    RIFF_t wav_riff;
    DATA_t wav_data;
    int ret;

    fd = open(file, O_RDONLY);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", file, strerror(errno));
        return -1;
    }

    /* 读取 RIFF chunk */
    ret = read(fd, &wav_riff, sizeof(RIFF_t));
    if (sizeof(RIFF_t) != ret) {
        if (0 > ret)
            perror("read error");
        else
            fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }
}
```

```
if (strcmp("RIFF", wav_riff.ChunkID, 4) != 0) //校验
    strcmp("WAVE", wav_riff.Format, 4) {
    fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

/* 读取 sub-chunk-fmt */
ret = read(fd, &wav_fmt, sizeof(FMT_t));
if (sizeof(FMT_t) != ret) {
    if (0 > ret)
        perror("read error");
    else
        fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

if (strcmp("fmt ", wav_fmt.Subchunk1ID, 4) != 0) //校验
    fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

/* 打印音频文件的信息 */
printf("<<<<音频文件格式信息>>>>\n\n");
printf("  file name:      %s\n", file);
printf("  Subchunk1Size: %u\n", wav_fmt.Subchunk1Size);
printf("  AudioFormat:    %u\n", wav_fmt.AudioFormat);
printf("  NumChannels:    %u\n", wav_fmt.NumChannels);
printf("  SampleRate:     %u\n", wav_fmt.SampleRate);
printf("  ByteRate:       %u\n", wav_fmt.ByteRate);
printf("  BlockAlign:     %u\n", wav_fmt.BlockAlign);
printf("  BitsPerSample: %u\n\n", wav_fmt.BitsPerSample);

/* sub-chunk-data */
if (0 > lseek(fd, sizeof(RIFF_t) + 8 + wav_fmt.Subchunk1Size,
             SEEK_SET)) {
    perror("lseek error");
    close(fd);
    return -1;
}
```

```
while(sizeof(DATA_t) == read(fd, &wav_data, sizeof(DATA_t))) {

    /* 找到 sub-chunk-data */
    if (!strcmp("data", wav_data.Subchunk2ID, 4))//校验
        return 0;

    if (0 > lseek(fd, wav_data.Subchunk2Size, SEEK_CUR)) {
        perror("lseek error");
        close(fd);
        return -1;
    }
}

fprintf(stderr, "check error: %s\n", file);
return -1;
}

/*****
main 主函数
*****/
int main(int argc, char *argv[])
{
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <audio_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 打开 WAV 音频文件 */
    if (open_wav_file(argv[1]))
        exit(EXIT_FAILURE);

    /* 初始化 PCM Playback 设备 */
    if (snd_pcm_init())
        goto err1;

    /* 申请读缓冲区 */
    buf = malloc(buf_bytes);
    if (NULL == buf) {
        perror("malloc error");
        goto err2;
    }
}
```



```
/* 播放 */
for (;;) {

    memset(buf, 0x00, buf_bytes); //buf 清零
    ret = read(fd, buf, buf_bytes); //从音频文件中读取数据
    if (0 >= ret) // 如果读取出错或文件读取完毕
        goto err3;

    ret = snd_pcm_writei(pcm, buf, period_size);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
        goto err3;
    }
    else if (ret < period_size) { //实际写入的帧数小于指定的帧数
        //此时我们需要调整下音频文件的读位置
        //将读位置向后移动（往回移）(period_size-ret)*frame_bytes 个字节
        //frame_bytes 表示一帧的字节大小
        if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
            perror("lseek error");
            goto err3;
        }
    }
}

err3:
    free(buf); //释放内存
err2:
    snd_pcm_close(pcm); //关闭 pcm 设备
err1:
    close(fd); //关闭打开的音频文件
    exit(EXIT_FAILURE);
}
```

本应用程序实现可以播放 WAV 音频文件，关于 WAV 文件格式的解析，本文档不作说明，WAV 文件格式其实非常简单，大家自己百度了解。

在 main() 函数中，首先对参数进行了校验，执行测试程序需要用户传入一个参数，这个参数用于指定一个需要播放的 WAV 音频文件。接着调用自定义函数 open_wav_file() 对 WAV 文件进行解析，其实也就是对它的头部数据进行校验、解析，获取音频格式信息以及音频数据的位置偏移量。

接着调用自定义函数 snd_pcm_init() 对 PCM 设备进行初始化，在 snd_pcm_init() 函数中，首先调用 alsa-lib 库函数 snd_pcm_open() 打开 PCM 播放设备，接着对 PCM 设备硬件参数进行设置，包括：访问类型、数据格式、采样率、声道数、周期大小以及 buffer 的大小，这些内容前面已经给大家详细介绍过，这里不再重述！

回到 main() 函数, 调用 C 库函数 malloc() 申请分配一个缓冲区, 用于存放从音频文件中读取出来的音频数据。

一切准备好之后, 就可以播放音频了, 在 for 循环中, 首先调用 read() 函数从音频文件中读取音频数据, 每次读取一个周期, 将读取到的数据存放在 buf 指向的缓冲区中, 接着调用 alsa-lib 库函数 snd_pcm_writei() 写入数据进行播放。示例程序中调用 snd_pcm_open() 时使用的是阻塞方式, 当驱动层环形缓冲区 buffer 还未满时, 调用 snd_pcm_writei() 并不会阻塞, 而是会将数据写入到环形缓冲区中、然后返回; 调用一次 snd_pcm_writei() 写入一个周期数据、调用一次再写入一个周期; 当环形缓冲区数据满时, 调用 snd_pcm_writei() 会阻塞, 直到音频设备播放完一个周期、此时会出现一个空闲周期, 接着 snd_pcm_writei() 将数据填充到这个空闲周期后返回。

以上对示例代码进行了一个简单地介绍, 代码本身非常简单, 没什么难点, 代码中注释信息也已经描述地比较清楚了, 相信大家都可以看懂。需要注意, 必须要在源码中包含 alsa-lib 的头文件 <alsa/asoundlib.h>!

编译示例代码

接下来编译上述示例代码, 编译的方法非常简单, 按照以前的惯例, 编译时无非是要指定两个路径 (alsa-lib 头文件所在路径、alsa-lib 库文件所在路径) 以及链接库 (需要链接的库文件名称), 譬如:

```
$(CC) -o testApp testApp.c -Ixxx -Lyyy -lzzz
```

xxx 表示头文件的路径, yyy 表示库文件的路径, zzz 表示链接库。

但是我们并没有自己移植 alsa-lib, 也就意味着我们在 Ubuntu 下并没有移植、安装 alsa-lib, 所以这些路径无法指定。其实, 我们使用的交叉编译工具对应的安装目录下已经安装了 alsa-lib, 进入到交叉编译工具安装目录下的 sysroots/cortexa7hf-neon-poky-linux-gnueabi 目录, 譬如笔者使用的 Ubuntu 系统, 交叉编译工具安装路径为 /opt/fsl-imx-x11/4.1.15-2.1.0。

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi$
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi$ pwd
/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi$ ls
lib  usr
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi$
```

图 28.5.5 cortexa7hf-neon-poky-linux-gnueabi 目录下的文件夹

该目录下有两个目录, lib 和 usr, 这两个目录其实就是 Linux 系统根目录下的 lib 和 usr; 所以 lib 目录下存放了一些链接库文件, usr 目录下包含了 include 和 lib 目录, 分别存放了头文件和链接库文件。usr/include/alsa 目录下存放了 alsa-lib 的头文件, 如下所示:

```
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/usr/include/alsa$
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/usr/include/alsa$ ls
alisp.h      conf.h      error.h      iatomic.h   mixer.h     pcm_extplug.h  pcm_old.h   rawmidi.h   seqmid.h
asounddef.h control_external.h global.h     input.h     output.h     pcm.h         pcm_plugin.h  seq_event.h  seq_midi_event.h
asoundlib.h control.h   hwdep.h     mixer_abst.h pcm_external.h pcm_ioplug.h  pcm_rate.h  seq.h       sound
```

图 28.5.6 alsa-lib 的头文件

我们需要包含的头文件 asoundlib.h 头文件就在该目录下。

usr/lib 目录下包含了 alsa-lib 库文件, 如下所示:

```
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/usr/lib$
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/usr/lib$ ls -l *asound*
-rwxr-xr-x 1 root root 937 11月 17 2020 libasound.la
lrwxrwxrwx 1 root root 18 1月 26 2021 libasound.so -> libasound.so.2.0.0
lrwxrwxrwx 1 root root 18 1月 26 2021 libasound.so.2 -> libasound.so.2.0.0
-rwxr-xr-x 1 root root 836656 11月 17 2020 libasound.so.2.0.0
dt@dt-virtual-machine:/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/usr/lib$
```

图 28.5.7 alsa-lib 库文件

alsa-lib 链接库 libasound.so 就在该目录下。那既然找到了 alsa-lib 的头文件路径和库文件路径, 编译应用程序时直接指定这些路径即可。但我们不需要自己手动指定这些路径, 交叉编译器已经把这些路径添加到它的搜索路径中了, 使用 `echo ${CC}` 查看环境变量 CC 的内容, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ echo ${CC}
arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
```

图 28.5.8 CC 环境变量的内容

其中交叉编译器 `arm-poky-linux-gnueabi-gcc` 有一个选项 `--sysroot`, 它指定了一个路径, 这个路径就是交叉编译工具安装目录下的 `sysroots/cortexa7hf-neon-poky-linux-gnueabi` 目录, `--sysroot` 选项用于设置目标平台的根目录, 设置了平台根目录之后, 当编译应用程序时, 编译器会将根目录下的 `usr/include` 添加到头文件搜索路径中、将根目录下的 `lib` 和 `usr/lib` 添加到库文件搜索路径中。

所以由此可知, 编译应用程序时, 我们只需指定链接库即可, 如下所示:

```
${CC} -o testApp testApp.c -lasound
```

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c -lasound
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 28.5.9 编译应用程序

测试应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统 `/home/root` 目录下, 并拷贝一个 WAV 音频文件到 `/home/root` 目录下, 如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# pwd
/home/root
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver EXO-Overdose.wav QDesktop-fb shell testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 28.5.10 将测试程序和 WAV 音频文件拷贝到开发板家目录

接着进行测试, 在测试之前, 我们还需要对声卡混音器进行配置, 当然, 你也可以不配置, 因为开发板出厂系统中声卡是已经配置好的。这里我们直接使用 `amixer` 工具进行配置, 配置如下:

```
# 打开耳机播放 ZC
amixer sset 'Headphone Playback ZC' on

# 打开喇叭播放 ZC
amixer sset 'Speaker Playback ZC' on
amixer sset 'Speaker AC' 3
amixer sset 'Speaker DC' 3

# 音量设置
amixer sset Headphone 105,105 //耳机音量设置
```

```

amixer sset Playback 230,230 //播放音量设置
amixer sset Speaker 118,118 //喇叭音量设置

# 打开左右声道
amixer sset 'Right Output Mixer PCM' on //打开右声道
amixer sset 'Left Output Mixer PCM' on //打开左声道

```

```

root@ATK-IMX6U:~# amixer sset 'Headphone Playback ZC' on
Simple mixer control 'Headphone Playback ZC',0
  Capabilities: pswitch
  Playback channels: Front Left - Front Right
  Mono:
  Front Left: Playback [on]
  Front Right: Playback [on]
root@ATK-IMX6U:~# amixer sset 'Speaker Playback ZC' on
Simple mixer control 'Speaker Playback ZC',0
  Capabilities: pswitch
  Playback channels: Front Left - Front Right
  Mono:
  Front Left: Playback [on]
  Front Right: Playback [on]
root@ATK-IMX6U:~# amixer sset 'Speaker AC' 3
Simple mixer control 'Speaker AC',0
  Capabilities: volume volume-joined
  Playback channels: Mono
  Capture channels: Mono
  Limits: 0 - 5
  Mono: 3 [60%]
root@ATK-IMX6U:~# amixer sset 'Speaker DC' 3
Simple mixer control 'Speaker DC',0
  Capabilities: volume volume-joined
  Playback channels: Mono
  Capture channels: Mono
  Limits: 0 - 5

```

图 28.5.11 声卡设置

由于篇幅有限,打印信息不能给大家全部截取出来。声音的大小,大家根据情况进行调节。声卡设置完成之后,接着运行测试程序,如下所示:

```

root@ATK-IMX6U:~# ./testApp ./EXO-Overdose.wav
<<<<音频文件格式信息>>>>

file name:      ./EXO-Overdose.wav
Subchunk1Size: 16
AudioFormat:    1
NumChannels:    2
SampleRate:     44100
ByteRate:       176400
BlockAlign:     4
BitsPerSample: 16

```

图 28.5.12 执行测试程序

程序运行之后,对传入的 WAV 文件进行解析,并将其音频格式信息打印出来。此时开发板喇叭便会开始播放音乐,如果连接了耳机,则会通过耳机播放音乐。

28.5.6 示例代码值 PCM 录音

本小节我们来编写一个 PCM 音频录制（录音）的测试程序，示例代码笔者已经给出，如下所示：

本例程源码对应的路径为：[开发板光盘->11、Linux C 应用编程例程源码->28_alsa-lib->pcm_capture.c](#)。

示例代码 28.5.2 一个简单地 PCM 录音示例程序

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : pcm_capture.c
作者 : 邓涛
版本 : V1.0
描述 : 一个简单地 PCM 音频采集示例代码--录音
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/20 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <alsa/asoundlib.h>

/*****
宏定义
*****/
#define PCM_CAPTURE_DEV    "hw:0,0"

/*****
static 静态全局变量定义
*****/
static snd_pcm_t *pcm = NULL;           //pcm 句柄
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16;       //周期数（buffer 的大小）
static unsigned int rate = 44100;       //采样率

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_CAPTURE_DEV, SND_PCM_STREAM_CAPTURE, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",

```

```
        PCM_CAPTURE_DEV, snd_strerror(ret));

    return -1;
}

/* 实例化 hwparams 对象 */
snd_pcm_hw_params_malloc(&hwparams);

/* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
ret = snd_pcm_hw_params_any(pcm, hwparams);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
    goto err2;
}

/*****
  设置参数
  *****/
/* 设置访问类型: 交错模式 */
ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置数据格式: 有符号 16 位、小端模式 */
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, rate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, 2);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}
```

```

    }

    /* 设置周期大小: period_size */
    ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /* 设置周期数 (buffer 的大小): periods */
    ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /* 使配置生效 */
    ret = snd_pcm_hw_params(pcm, hwparams);
    snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
        goto err1;
    }

    return 0;

err2:
    snd_pcm_hw_params_free(hwparams); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    return -1;
}

/*****
main 主函数
*****/
int main(int argc, char *argv[])
{
    unsigned char *buf = NULL;
    unsigned int buf_bytes;
    int fd = -1;
    int ret;

```

```
if (2 != argc) {
    fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
    exit(EXIT_FAILURE);
}

/* 初始化 PCM Capture 设备 */
if (snd_pcm_init())
    exit(EXIT_FAILURE);

/* 申请读缓冲区 */
buf_bytes = period_size * 4;    //字节大小 = 周期大小*帧的字节大小 16 位双声道
buf = malloc(buf_bytes);
if (NULL == buf) {
    perror("malloc error");
    goto err1;
}

/* 打开一个新建文件 */
fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL);
if (0 > fd) {
    fprintf(stderr, "open error: %s: %s\n", argv[1], strerror(errno));
    goto err2;
}

/* 录音 */
for (;;) {

    //memset(buf, 0x00, buf_bytes);    //buf 清零
    ret = snd_pcm_readi(pcm, buf, period_size); //读取 PCM 数据 一个周期
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_readi error: %s\n", snd_strerror(ret));
        goto err3;
    }

    // snd_pcm_readi 的返回值 ret 等于实际读取的帧数 * 4 转为字节数
    ret = write(fd, buf, ret * 4);    //将读取到的数据写入文件中
    if (0 >= ret)
        goto err3;
}

err3:
    close(fd); //关闭文件
err2:
```



```

    free(buf);           //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    exit(EXIT_FAILURE);
}

```

在 main()函数中, 首先对参数进行了校验, 执行测试程序需要用户传入一个参数, 指定输出文件, 因为示例程序中会将录制的音频数据保存到该文件中。

接着调用自定义函数 snd_pcm_init()对 PCM 设备进行初始化, 在 snd_pcm_init()函数中, 首先调用 alsa-lib 库函数 snd_pcm_open()打开 PCM 录音设备, 接着对 PCM 设备硬件参数进行设置, 访问类型设置交错模式 SND_PCM_ACCESS_RW_INTERLEAVED、数据格式设置为 SND_PCM_FORMAT_S16_LE、采样率设置为 44100、双声道、周期大小设置为 1024 帧、buffer 大小设置为 16 个周期。

回到 main()函数, 调用 C 库函数 malloc()申请分配一个缓冲区, 用于存放从驱动层环形缓冲区 buffer 读取出来的音频数据。并打开一个新建文件 (因为使用了 O_CREAT | O_EXCL 标志)。

一切准备好之后, 就可以进行音频录制了, 在 for 循环中, 首先调用 alsa-lib 库函数 snd_pcm_readi()从环形缓冲区中读取音频设备采集到的音频数据, 读取出来之后调用 write()函数将数据写入到文件中。示例程序中调用 snd_pcm_open()时使用的是阻塞方式, 当环形缓冲区 buffer 中有数据可读时, 调用 snd_pcm_readi()并不会阻塞, 而是读取出数据、然后返回; 调用一次 snd_pcm_readi()读取一个周期、调用一次再读取一个周期; 当环形缓冲区为空时, 调用 snd_pcm_readi()会阻塞, 直到音频设备采集到一个周期数据、此时被阻塞 snd_pcm_readi()调用被唤醒、读取这一个周期然后返回。

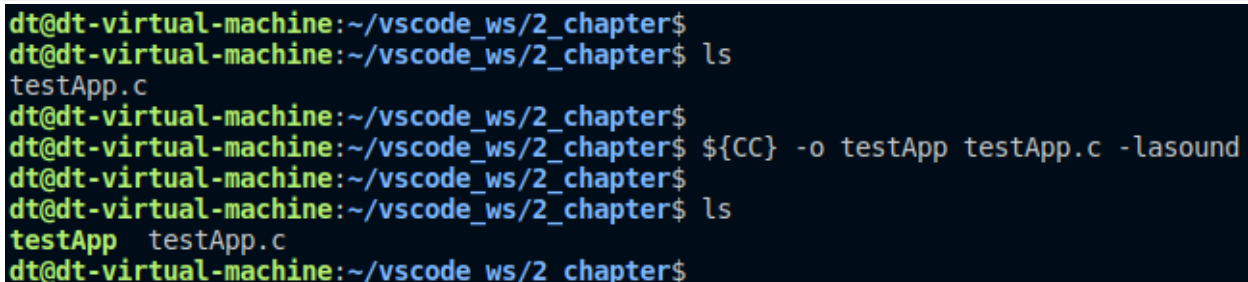
编译示例代码

接下来我们编译示例代码, 如下所示:

```

${CC} -o testApp testApp.c -lasound

```



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c -lasound
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 28.5.13 编译示例代码

测试应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统/home/root 目录下, 在执行测试程序之前, 我们需要对声卡进行配置, 同样使用 amixer 工具进行配置, 如下:

```

amixer sset Capture 58,58           //录制音量大小
amixer sset 'ADC PCM' 200,200      //PCM ADC

# 左声道 Mixer Boost 管理
amixer sset 'Left Input Mixer Boost' off
amixer sset 'Left Boost Mixer LINPUT1' off
amixer sset 'Left Input Boost Mixer LINPUT1' 0
amixer sset 'Left Boost Mixer LINPUT2' off
amixer sset 'Left Input Boost Mixer LINPUT2' 0

```

```

amixer sset 'Left Boost Mixer LINPUT3' off
amixer sset 'Left Input Boost Mixer LINPUT3' 0

# 右声道 Mixer Boost 管理
amixer sset 'Right Input Mixer Boost' on
amixer sset 'Right Boost Mixer RINPUT1' on
amixer sset 'Right Input Boost Mixer RINPUT1' 5
amixer sset 'Right Boost Mixer RINPUT2' on
amixer sset 'Right Input Boost Mixer RINPUT2' 5
amixer sset 'Right Boost Mixer RINPUT3' off
amixer sset 'Right Input Boost Mixer RINPUT3' 0

```

```

root@ATK-IMX6U:~# amixer sset Capture 58,58
Simple mixer control 'Capture',0
  Capabilities: cvolume cswitch
  Capture channels: Front Left - Front Right
  Limits: Capture 0 - 63
  Front Left: Capture 58 [92%] [26.25dB] [on]
  Front Right: Capture 58 [92%] [26.25dB] [on]
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# amixer sset 'ADC PCM' 200,200
Simple mixer control 'ADC PCM',0
  Capabilities: cvolume
  Capture channels: Front Left - Front Right
  Limits: Capture 0 - 255
  Front Left: Capture 200 [78%] [2.50dB]
  Front Right: Capture 200 [78%] [2.50dB]
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# amixer sset 'Left Input Mixer Boost' off
Simple mixer control 'Left Input Mixer Boost',0
  Capabilities: pswitch pswitch-joined
  Playback channels: Mono
  Mono: Playback [off]
root@ATK-IMX6U:~# amixer sset 'Left Boost Mixer LINPUT1' off
Simple mixer control 'Left Boost Mixer LINPUT1',0
  Capabilities: pswitch pswitch-joined
  Playback channels: Mono
  Mono: Playback [off]

```

图 28.5.14 声卡配置 (录音)

左右声道的 Mixer Boost (混音器增强) 为什么要这样去配置? 这个与硬件设计有关系, 我们就不去解释这个了。具体详情可以参考《I.MX6U 嵌入式 Linux 驱动开发指南》文档中音频驱动章节的内容。

接下来, 执行测试程序进行录音, 如下所示:

```

root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver  EXO-Overdose.wav  QDesktop-fb  shell  testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp ./cap.wav

```

图 28.5.15 录音

执行测试程序之后, 就开始录音了, 接着我们可以对着底板上的麦 (MIC) 说话, 板载的 MIC 如下所示:

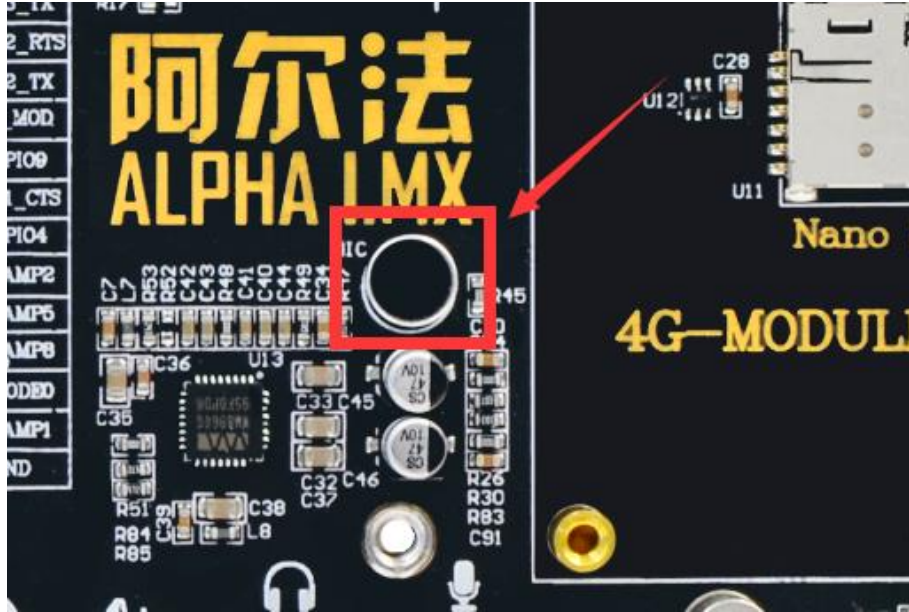


图 28.5.16 板载麦克风

程序就会把我们说的话录进去；如果想要停止录音、只能终止进程，按 Ctrl+C 终止应用程序；此时在当前目录下会生成 cap.wav 音频文件，如下所示：

```

root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
cap.wav driver EXO-Overdose.wav QDesktop-fb shell testApp
root@ATK-IMX6U:~#

```

图 28.5.17 生成 cap.wav 文件

生成的文件是一个纯音频数据的文件，并不是 WAV 格式的文件，因为这个文件没有头部信息，程序中如果检测到该文件不是 WAV 格式文件、会直接退出，所以不能直接使用上小节 28.5.5 的测试程序播放 cap.wav 文件，这里要注意！当然你可以对上小节的示例代码进行修改，也可直接使用 aplay 工具播放这段录制的音频，如下：

```
aplay -f cd cap.wav
```

```

root@ATK-IMX6U:~# aplay -f cd cap.wav
Playing raw data 'cap.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo

```

图 28.5.18 使用 aplay 播放录制的音频

如果录制正常，使用 aplay 播放出来的声音就是我们录制的声音！

LINE_IN 测试

除了麦克风之外，开发板底板上还有一个 LINE_IN 接口，也就是线路输入，如下图所示：

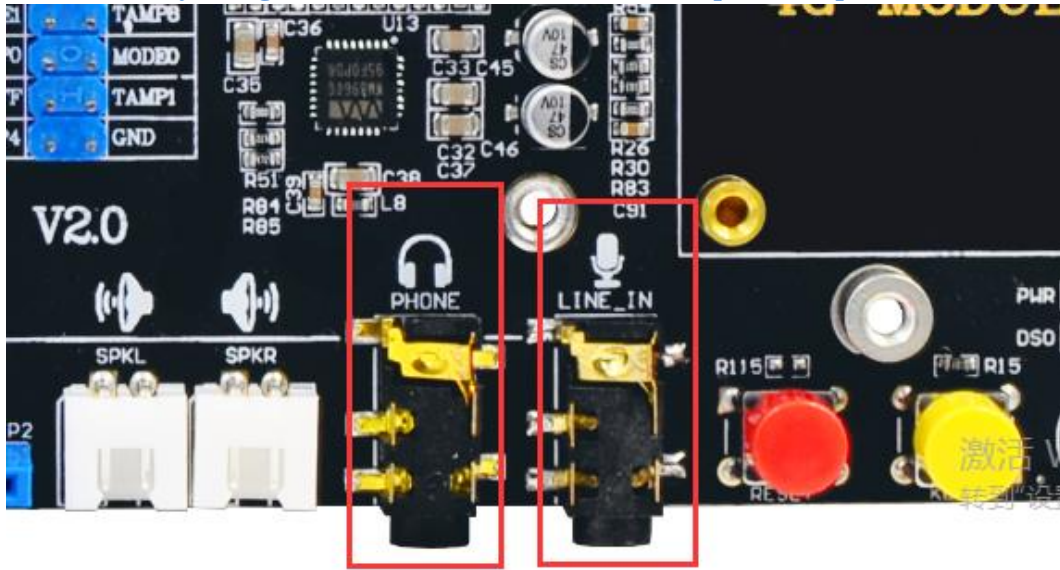


图 28.5.19 LINE_IN 接口

上图中左边的是耳机接口、右边的是 LINE_IN 接口，支持音频输入，我们通过本测试程序对 LINE_IN 接口进行测试，采集 LINE_IN 接口输入的音频。测试时我们使用一根 3.5mm 公对公音频线，一头连接到手机或者电脑、另外一头连接到 LINE_IN 接口上，然后手机或电脑端播放音乐，那么音频数据就会通过 LINE_IN 接口输入到开发板被我们的应用程序采集（录制）。

在测试之前，我们需要对声卡进行配置，如下所示：

```
amixer sset Capture 58,58 //录制音量大小
amixer sset 'ADC PCM' 200,200 //PCM ADC
```

左声道 Mixer Boost 管理

```
amixer sset 'Left Input Mixer Boost' off
amixer sset 'Left Boost Mixer LINPUT1' off
amixer sset 'Left Input Boost Mixer LINPUT1' 0
amixer sset 'Left Boost Mixer LINPUT2' on
amixer sset 'Left Input Boost Mixer LINPUT2' 5
amixer sset 'Left Boost Mixer LINPUT3' off
amixer sset 'Left Input Boost Mixer LINPUT3' 0
```

右声道 Mixer Boost 管理

```
amixer sset 'Right Input Mixer Boost' on
amixer sset 'Right Boost Mixer RINPUT1' off
amixer sset 'Right Input Boost Mixer RINPUT1' 0
amixer sset 'Right Boost Mixer RINPUT2' off
amixer sset 'Right Input Boost Mixer RINPUT2' 0
amixer sset 'Right Boost Mixer RINPUT3' on
amixer sset 'Right Input Boost Mixer RINPUT3' 5
```

配置好之后就可以进行测试了，执行程序之后，手机或电脑端播放音乐，开发板采集从 LINE_IN 接口输入的音频数据，测试方式跟 MIC 麦克风一样，大家自己去测试！

28.6 使用异步方式

上小节中的示例代码 28.5.1 和示例代码 28.5.2 都是采用了同步方式进行读写, 这样会使得应用程序无法做一些其它的事情, 本小节我们来学习如何使用异步方式读写。

其实使用异步方式读写非常简单, 只需要注册异步处理函数即可。

snd_async_add_pcm_handler()函数

alsa-lib 提供了 `snd_async_add_pcm_handler()` 函数用于注册异步处理函数, 其实我们只需要通过这个函数注册一个异步处理函数即可, 其函数原型如下所示:

```
int snd_async_add_pcm_handler(snd_async_handler_t **handler,
    snd_pcm_t *pcm,
    snd_async_callback_t callback,
    void *private_data
)
```

调用该函数需要传入 4 个参数:

- **handler:** 参数 `snd_async_handler_t` 用于描述一个异步处理, 所以一个 `snd_async_handler_t` 对象表示一个异步处理对象; 调用 `snd_async_add_pcm_handler()` 函数会实例化一个 `snd_async_handler_t` 对象, 并将对象的指针 (指针作为异步处理对象的句柄) 通过 `*handler` 返回出来。
- **pcm:** pcm 设备的句柄。
- **callback:** 异步处理函数 (或者叫回调函数), `snd_async_callback_t` 函数指针如下所示:

```
typedef void(*snd_async_callback_t)(snd_async_handler_t *handler)
```

参数 `handler` 也就是异步处理对象的句柄。

- **private_data:** 传递给异步处理函数的私有数据, 私有数据的数据类型, 可以由用户自己定义, 调用 `snd_async_add_pcm_handler()` 函数时, 参数 `private_data` 指向你的私有数据对象。在异步处理函数中便可以获取到私有数据, 调用 `snd_async_handler_get_callback_private()` 函数即可, 如下所示:

```
struct my_private_data *data = snd_async_handler_get_callback_private(handler);
```

关于 `snd_async_add_pcm_handler()` 函数的参数介绍, 就给大家说这么多。当调用该函数之后, 用户传入的 PCM 设备将会与异步处理对象关联起来, 在异步处理函数 `callback` 中可以通过异步处理对象的句柄获取到 PCM 设备的句柄, 通过 `snd_async_handler_get_pcm()` 获取, 如下所示:

```
snd_pcm_t *pcm_handle = snd_async_handler_get_pcm(handler);
```

实现异步 I/O, 应用程序通常需要完成这三件事情:

- 使能异步 I/O;
- 设置异步 I/O 的所有者;
- 注册信号处理函数 (譬如 `SIGIO` 信号或其它实时信号)。

这是内容在 13.3 小节给大家详细介绍过, 这里不再啰嗦! 所以由此可知, `snd_async_add_pcm_handler` 函数中已经帮我们完成这些事情。

使用示例:

```
static void snd_playback_async_callback(snd_async_handler_t *handler)
{
    snd_pcm_t *handle = snd_async_handler_get_pcm(handler); //获取 PCM 句柄

    .....
}
```

```
int main(void)
{
    .....

    snd_async_handler_t *async_handler = NULL;

    /* 注册异步处理函数 */
    ret = snd_async_add_pcm_handler(&async_handler, pcm, snd_playback_async_callback, NULL);
    if (0 > ret)
        fprintf(stderr, "snd_async_add_pcm_handler error: %s\n", snd_strerror(ret));

    .....
}
```

调用 `snd_async_add_pcm_handler()` 注册了异步回调函数 `snd_playback_async_callback()`，当环形缓冲区有空闲的周期可填充数据时（以播放为例），音频设备驱动程序会向应用程序发送信号（SIGIO），接着应用程序便会跳转到 `snd_playback_async_callback()` 函数执行。

而对于录音来说，当环形缓冲区中有数据可读时（譬如音频设备已经录制了一个周期、并将数据写入到了环形缓冲区），驱动程序便会向应用程序发送信号，接着应用程序跳转到回调函数执行。

在播放情况下，通常会先将环形缓冲区填满，当音频设备每播放完一个周期，就会产生一个空闲周期，此时应用程序会接收到信号，进而跳转到异步回调函数中执行。

snd_pcm_avail_update()函数

在异步处理函数中，我们通常会使用到这个函数，在录音情况下，应用程序调用 `snd_pcm_avail_update()` 函数用于获取当前可读取的帧数；在播放情况下，应用程序调用该函数用于获取当前可写入的帧数。换句话说，也就是驱动层环形缓冲区中当前有多少帧数据可读取（录音）或可写入多少帧数据（播放，环形缓冲区未写满时、应用程序才可写入数据）。

该函数原型如下所示：

```
snd_pcm_sframes_t snd_pcm_avail_update(snd_pcm_t *pcm);
```

本小节主要给大家介绍这两个函数，因为后面的示例代码中会使用到。

28.6.1 PCM 播放示例-异步方式

通过上面的介绍，本小节我们来编写一个使用异步方式的 PCM 播放示例程序，直接基于示例代码 28.5.1 进行修改，代码笔者已经写好了，如下所示：

本例程源码对应的路径为：[开发板光盘 ->11](#)、[Linux C 应用编程例程源码 ->28_alsa-lib->pcm_playback_async.c](#)。

示例代码 28.6.1 一个简单地 PCM 播放示例程序（异步方式）

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : pcm_playback_async.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

```
描述 : 一个简单地 PCM 播放示例代码--使用异步方式
```

```
其他 : 无
```

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/7/20 邓涛创建

```

*****/

```

```

#include <stdio.h>

```

```

#include <stdlib.h>

```

```

#include <errno.h>

```

```

#include <string.h>

```

```

#include <alsa/asoundlib.h>

```

```

/*****

```

```

宏定义

```

```

*****/

```

```

#define PCM_PLAYBACK_DEV    "hw:0,0"

```

```

/*****

```

```

WAV 音频文件解析相关数据结构申明

```

```

*****/

```

```

typedef struct WAV_RIFF {

```

```

    char ChunkID[4];           /* "RIFF" */

```

```

    u_int32_t ChunkSize;       /* 从下一个地址开始到文件末尾的总字节数 */

```

```

    char Format[4];           /* "WAVE" */

```

```

} __attribute__((packed)) RIFF_t;

```

```

typedef struct WAV_FMT {

```

```

    char Subchunk1ID[4];       /* "fmt " */

```

```

    u_int32_t Subchunk1Size;    /* 16 for PCM */

```

```

    u_int16_t AudioFormat;      /* PCM = 1 */

```

```

    u_int16_t NumChannels;      /* Mono = 1, Stereo = 2, etc. */

```

```

    u_int32_t SampleRate;       /* 8000, 44100, etc. */

```

```

    u_int32_t ByteRate;         /* = SampleRate * NumChannels * BitsPerSample/8 */

```

```

    u_int16_t BlockAlign;       /* = NumChannels * BitsPerSample/8 */

```

```

    u_int16_t BitsPerSample;    /* 8bits, 16bits, etc. */

```

```

} __attribute__((packed)) FMT_t;

```

```

static FMT_t wav_fmt;

```

```

typedef struct WAV_DATA {

```

```

    char Subchunk2ID[4];       /* "data" */

```

```

    u_int32_t Subchunk2Size;    /* data size */

```

```

} __attribute__((packed)) DATA_t;

```

```

/*****

```

```

static 静态全局变量定义

```

```

*****/

static snd_pcm_t *pcm = NULL;           //pcm 句柄
static unsigned int buf_bytes;         //应用程序缓冲区的大小（字节为单位）
static void *buf = NULL;               //指向应用程序缓冲区的指针
static int fd = -1;                    //指向 WAV 音频文件的文件描述符
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16;      //周期数（设备驱动层 buffer 的大小）

/*****
static 静态函数
*****/

static void snd_playback_async_callback(snd_async_handler_t *handler)
{
    snd_pcm_t *handle = snd_async_handler_get_pcm(handler); //获取 PCM 句柄
    snd_pcm_sframes_t avail;
    int ret;

    avail = snd_pcm_avail_update(handle); //获取环形缓冲区中有多少帧数据需要填充
    while (avail >= period_size) { //我们一次写入一个周期

        memset(buf, 0x00, buf_bytes); //buf 清零
        ret = read(fd, buf, buf_bytes);
        if (0 >= ret)
            goto out;

        ret = snd_pcm_writei(handle, buf, period_size);
        if (0 > ret) {
            fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
            goto out;
        }
        else if (ret < period_size) { //实际写入的帧数小于指定的帧数
            //此时我们需要调整下音频文件的读位置 重新读取没有播放出去的数据
            //将读位置向后移动（往回移）(period_size-ret)*frame_bytes 个字节
            //frame_bytes 表示一帧的字节大小
            if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
                perror("lseek error");
                goto out;
            }
        }
    }

    avail = snd_pcm_avail_update(handle); //再次获取、更新 avail
}

```



```
    return;
out:
    snd_pcm_close(handle); //关闭 pcm 设备
    free(buf);
    close(fd);           //关闭打开的音频文件
    exit(EXIT_FAILURE); //退出程序
}

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    snd_async_handler_t *async_handler = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_PLAYBACK_DEV, SND_PCM_STREAM_PLAYBACK, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_PLAYBACK_DEV, snd_strerror(ret));
        return -1;
    }

    /* 实例化 hwparams 对象 */
    snd_pcm_hw_params_malloc(&hwparams);

    /* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
    ret = snd_pcm_hw_params_any(pcm, hwparams);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /******
    设置参数
    *****/
    /* 设置访问类型: 交错模式 */
    ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /* 设置数据格式: 有符号 16 位、小端模式 */
```

```
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, wav_fmt.SampleRate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, wav_fmt.NumChannels);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期大小: period_size */
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期数 (驱动层环形缓冲区 buffer 的大小): periods */
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 使配置生效 */
ret = snd_pcm_hw_params(pcm, hwparams);
snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
    goto err1;
}

buf_bytes = period_size * wav_fmt.BlockAlign; //变量赋值, 一个周期的字节大小
```

```
/* 注册异步处理函数 */
ret = snd_async_add_pcm_handler(&async_handler, pcm, snd_playback_async_callback, NULL);
if (0 > ret) {
    fprintf(stderr, "snd_async_add_pcm_handler error: %s\n", snd_strerror(ret));
    goto err1;
}

return 0;

err2:
snd_pcm_hw_params_free(hwparams); //释放内存
err1:
snd_pcm_close(pcm); //关闭 pcm 设备
return -1;
}

static int open_wav_file(const char *file)
{
    RIFF_t wav_riff;
    DATA_t wav_data;
    int ret;

    fd = open(file, O_RDONLY);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", file, strerror(errno));
        return -1;
    }

    /* 读取 RIFF chunk */
    ret = read(fd, &wav_riff, sizeof(RIFF_t));
    if (sizeof(RIFF_t) != ret) {
        if (0 > ret)
            perror("read error");
        else
            fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    if (strncmp("RIFF", wav_riff.ChunkID, 4) || //校验
        strncmp("WAVE", wav_riff.Format, 4)) {
        fprintf(stderr, "check error: %s\n", file);
    }
}
```

```
        close(fd);
        return -1;
    }

    /* 读取 sub-chunk-fmt */
    ret = read(fd, &wav_fmt, sizeof(FMT_t));
    if (sizeof(FMT_t) != ret) {
        if (0 > ret)
            perror("read error");
        else
            fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    if (strcmp("fmt ", wav_fmt.Subchunk1ID, 4) != 0) { //校验
        fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    /* 打印音频文件的信息 */
    printf("<<<<音频文件格式信息>>>>\n\n");
    printf("  file name:      %s\n", file);
    printf("  Subchunk1Size: %u\n", wav_fmt.Subchunk1Size);
    printf("  AudioFormat:    %u\n", wav_fmt.AudioFormat);
    printf("  NumChannels:    %u\n", wav_fmt.NumChannels);
    printf("  SampleRate:     %u\n", wav_fmt.SampleRate);
    printf("  ByteRate:       %u\n", wav_fmt.ByteRate);
    printf("  BlockAlign:     %u\n", wav_fmt.BlockAlign);
    printf("  BitsPerSample: %u\n\n", wav_fmt.BitsPerSample);

    /* sub-chunk-data */
    if (0 > lseek(fd, sizeof(RIFF_t) + 8 + wav_fmt.Subchunk1Size,
                SEEK_SET)) {
        perror("lseek error");
        close(fd);
        return -1;
    }

    while(sizeof(DATA_t) == read(fd, &wav_data, sizeof(DATA_t))) {

        /* 找到 sub-chunk-data */
```

```
    if (!strcmp("data", wav_data.Subchunk2ID, 4))//校验
        return 0;

    if (0 > lseek(fd, wav_data.Subchunk2Size, SEEK_CUR)) {
        perror("lseek error");
        close(fd);
        return -1;
    }
}

fprintf(stderr, "check error: %s\n", file);
return -1;
}

/*****
main 主函数
*****/
int main(int argc, char *argv[])
{
    snd_pcm_sframes_t avail;
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <audio_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 打开 WAV 音频文件 */
    if (open_wav_file(argv[1]))
        exit(EXIT_FAILURE);

    /* 初始化 PCM Playback 设备 */
    if (snd_pcm_init())
        goto err1;

    /* 申请读缓冲区 */
    buf = malloc(buf_bytes);
    if (NULL == buf) {
        perror("malloc error");
        goto err2;
    }

    /* 播放: 先将环形缓冲区填满数据 */
```

```
avail = snd_pcm_avail_update(pcm); //获取环形缓冲区中有多少帧数据需要填充
while (avail >= period_size) { //我们一次写入一个周期

    memset(buf, 0x00, buf_bytes); //buf 清零
    ret = read(fd, buf, buf_bytes);
    if (0 >= ret)
        goto err3;

    ret = snd_pcm_writei(pcm, buf, period_size); //向环形缓冲区中写入数据
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
        goto err3;
    }
    else if (ret < period_size) { //实际写入的帧数小于指定的帧数
        //此时我们需要调整下音频文件的读位置
        //将读位置向后移动 (往回移) (period_size-ret)*frame_bytes 个字节
        //frame_bytes 表示一帧的字节大小
        if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
            perror("lseek error");
            goto err3;
        }
    }
}

avail = snd_pcm_avail_update(pcm); //再次获取、更新 avail
}

for (;;) {
    /* 主程序可以做一些其它的事，当环形缓冲区有空闲周期需要写入数据时
     * 音频设备驱动程序会向应用程序发送 SIGIO 信号
     * 接着应用程序跳转到 snd_playback_async_callback()函数执行 */
    //do_something();
    sleep(1);
}

err3:
    free(buf); //释放内存
err2:
    snd_pcm_close(pcm); //关闭 pcm 设备
err1:
    close(fd); //关闭打开的音频文件
    exit(EXIT_FAILURE);
}
```

在 `snd_pcm_init()` 函数中, 我们调用了 `snd_async_add_pcm_handler()` 函数注册了异步回调函数 `snd_playback_async_callback()`, 当可写入数据时, 跳转到 `snd_playback_async_callback()` 函数去执行。

在异步回调函数中, 我们首先调用 `snd_pcm_avail_update()` 获取当前可写入多少帧数据, 然后在 `while()` 循环中调用 `read()` 读取音频文件的数据、接着调用 `snd_pcm_writewi()` 向环形缓冲区写入数据, 每次循环写入一个周期, 直到把缓冲区写满, 然后退出回调函数。

回到 `main()` 函数中, 在进入 `for()` 死循环之前, 我们先将环形缓冲区填满, 执行的代码与回调函数中的代码相同, 这里就不再说明了!

编译示例代码

在 Ubuntu 系统下执行命令, 编译示例代码:

```
${CC} -o testApp testApp.c -lasound
```

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c -lasound
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 28.6.1 编译示例代码

测试应用程序

将上面编译得到的可执行文件拷贝开发板 Linux 系统/home/root 目录下, 然后在开发板上测试, 大家自己去测!

28.6.2 PCM 录音示例-异步方式

本小节编写使用异步方式的 PCM 录音的示例程序, 代码笔者已经写好了, 如下所示:

本例程源码对应的路径为: [开发板光盘 ->11、Linux C 应用编程例程源码 ->28_alsa-lib->pcm_capture_async.c](#)。

示例代码 28.6.2 一个简单地 PCM 录音示例程序 (异步方式)

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : pcm_capture_async.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

```
描述 : 一个简单地 PCM 音频采集示例代码--异步方式
```

```
其他 : 无
```

```
论坛 : www.openedv.com
```

```
日志 : 初版 V1.0 2021/7/20 邓涛创建
```

```
*****/
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

```

#include <alsa/asoundlib.h>

/*****
宏定义
*****/
#define PCM_CAPTURE_DEV    "hw:0,0"

/*****
static 静态全局变量定义
*****/
static snd_pcm_t *pcm = NULL;           //pcm 句柄
static unsigned int buf_bytes;         //应用层缓冲区的大小（字节为单位）
static void *buf = NULL;               //指向应用层缓冲区的指针
static int fd = -1;                    //输出文件的文件描述符
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16;      //周期数（驱动层环形缓冲区的大小）
static unsigned int rate = 44100;     //采样率

/*****
static 静态函数
*****/
static void snd_capture_async_callback(snd_async_handler_t *handler)
{
    snd_pcm_t *handle = snd_async_handler_get_pcm(handler);
    snd_pcm_sframes_t avail;
    int ret;

    avail = snd_pcm_avail_update(handle); //检查有多少帧数据可读
    while (avail >= period_size) { //每次读取一个周期

        //memset(buf, 0x00, buf_bytes); //buf 清零
        ret = snd_pcm_readi(handle, buf, period_size); //读取 PCM 数据 一个周期
        if (0 > ret) {
            fprintf(stderr, "snd_pcm_readi error: %s\n", snd_strerror(ret));
            goto out;
        }

        // snd_pcm_readi 的返回值 ret 等于实际读取的帧数 * 4 转为字节数
        ret = write(fd, buf, ret * 4); //将读取到的数据写入文件中
        if (0 >= ret)
            goto out;

        avail = snd_pcm_avail_update(handle); //再次读取、更新 avail
    }
}

```



```

    }

    return;
out:
    snd_pcm_close(handle); //关闭 pcm 设备
    free(buf);
    close(fd);           //关闭打开的音频文件
    exit(EXIT_FAILURE); //退出程序
}

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    snd_async_handler_t *async_handler = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_CAPTURE_DEV, SND_PCM_STREAM_CAPTURE, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_CAPTURE_DEV, snd_strerror(ret));
        return -1;
    }

    /* 实例化 hwparams 对象 */
    snd_pcm_hw_params_malloc(&hwparams);

    /* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
    ret = snd_pcm_hw_params_any(pcm, hwparams);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /******
    设置参数
    *****/

    /* 设置访问类型: 交错模式 */
    ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
        goto err2;
    }
}

```

```
/* 设置数据格式: 有符号 16 位、小端模式 */
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, rate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, 2);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期大小: period_size */
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期数 (buffer 的大小): periods */
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 使配置生效 */
ret = snd_pcm_hw_params(pcm, hwparams);
snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
    goto err1;
}
```

```
/* 注册异步处理函数 */
ret = snd_async_add_pcm_handler(&async_handler, pcm, snd_capture_async_callback, NULL);
if (0 > ret) {
    fprintf(stderr, "snd_async_add_pcm_handler error: %s\n", snd_strerror(ret));
    goto err1;
}

return 0;

err2:
    snd_pcm_hw_params_free(hwparams); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    return -1;
}

/*****
main 主函数
*****/
int main(int argc, char *argv[])
{
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 初始化 PCM Capture 设备 */
    if (snd_pcm_init())
        exit(EXIT_FAILURE);

    /* 申请读缓冲区 */
    buf_bytes = period_size * 4; //字节大小 = 周期大小*帧的字节大小 16 位双声道
    buf = malloc(buf_bytes);
    if (NULL == buf) {
        perror("malloc error");
        goto err1;
    }

    /* 打开一个新建文件 */
    fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL);
```

```
if (0 > fd) {
    fprintf(stderr, "open error: %s: %s\n", argv[1], strerror(errno));
    goto err2;
}

/* 录音 */
ret = snd_pcm_start(pcm);          //开始录音
if (0 > ret) {
    fprintf(stderr, "snd_pcm_start error: %s\n", snd_strerror(ret));
    goto err3;
}

for (;;) {
    /* 主程序可以做一些其它的事, 当环形缓冲区有数据可读时
     * 音频设备驱动程序会向应用程序发送 SIGIO 信号
     * 接着应用程序跳转到 snd_capture_async_callback()函数执行、读取数据 */
    //do_something();
    sleep(1);
}

err3:
    close(fd); //关闭文件
err2:
    free(buf); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    exit(EXIT_FAILURE);
}
```

这份代码基于示例代码 28.5.2 改写, 使用异步方式读取录制的音频数据。

代码不再解释了, 值得注意的是, 在 main()函数中我们调用了 snd_pcm_start()函数, 这个函数前面没给大家介绍过, 该函数的作用其实如它命名那般, 用于启动 PCM 设备, 譬如在录音情况下, 调用该函数开始录音; 在播放情况下, 调用该函数开始播放。

前面的几个示例代码中, 为啥没有调用该函数呢? 这个问题我们先留着、稍后再给大家介绍!

编译示例代码

执行命令编译示例代码:

```
${CC} -o testApp testApp.c -lasound
```

测试应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统/home/root 目录下, 然后进行测试, 测试方法与示例代码 28.5.2 对应的测试程序相同, 这里不再重述, 大家自己去测!

28.7 使用 poll()函数

上小节我们使用了异步 I/O 方式读写 PCM 设备, 本小节我们来学习如何使用 poll I/O 多路复用来实现读写数据。

28.7.1 使用 poll I/O 多路复用实现读写

I/O 多路复用是一种高级 I/O, 在第一篇 13.2 小节给大家进行了详细地介绍, 可通过 select()或 poll()函数来实现 I/O 多路复用, 本小节我们使用 poll()函数来实现 I/O 多路复用, 接下来将向大家介绍!

获取计数: snd_pcm_poll_descriptors_count

该函数用于获取 PCM 句柄的轮询描述符计数, 其函数原型如下所示:

```
int snd_pcm_poll_descriptors_count(snd_pcm_t *pcm);
```

调用该函数返回 PCM 句柄的轮询描述符计数。

分配 struct pollfd 对象

为每一个轮询描述符分配一个 struct pollfd 对象, 譬如:

```
struct pollfd *pfd = NULL;
int count;

/* 获取 PCM 句柄的轮询描述符计数 */
count = snd_pcm_poll_descriptors_count(pcm);
if (0 >= count) {
    fprintf(stderr, "Invalid poll descriptors count\n");
    return -1;
}

/* 分配内存 */
pfd = calloc(count, sizeof(struct pollfd));
if (NULL == pfd) {
    perror("calloc error");
    return -1;
}
```

填充 struct pollfd: snd_pcm_poll_descriptors

接下来调用 snd_pcm_poll_descriptors()函数对 struct pollfd 对象进行填充(初始化), 其函数原型如下所示:

```
int snd_pcm_poll_descriptors(
    snd_pcm_t *pcm,
    struct pollfd *pfd,
    unsigned int space
);
```

参数 space 表示 pfd 数组中的元素个数。

```
/* 填充 pfd */
ret = snd_pcm_poll_descriptors(pcm, pfd, count);
if (0 > ret)
```

```
return -1;
```

poll+snd_pcm_poll_descriptors_revents

一切准备完成之后, 就可以调用 poll()函数来监视 PCM 设备是否有数据可读或可写, 当有数据可读或可写时, poll()函数返回, 此时我们可以调用 snd_pcm_poll_descriptors_revents()函数获取文件描述符中返回的事件类型, 并与 poll 的 events 标志进行比较, 以确定是否可读或可写, snd_pcm_poll_descriptors_revents()函数原型如下所示:

```
int snd_pcm_poll_descriptors_revents(
    snd_pcm_t *pcm,
    struct pollfd *pfd,
    unsigned int nfds,
    unsigned short *revents
)
```

参数 nfds 表示 pfd 数组中元素的个数, 调用该函数获取文件描述符中返回的事件, 通过参数 revents 返回出来; 注意, 不要直接读取 struct pollfd 对象中的 revents 成员变量, 因为 snd_pcm_poll_descriptors_revents()函数会对 poll()系统调用返回的 revents 掩码进行“分解”以纠正语义(POLLIN = 读取, POLLOUT = 写入)。

使用示例:

```
for (;) {

    ret = poll(pfd, count, -1); //调用 poll
    if (0 > ret) {
        perror("poll error");
        return -1;
    }

    ret = snd_pcm_poll_descriptors_revents(pcm, pfd, count, &revents);
    if (0 > ret)
        return -1;
    if (revents & POLLERR) //发生 I/O 错误
        return -1;
    if (revents & POLLIN) { //表示可读取数据
        // 从 PCM 设备读取数据
    }
    if (revents & POLLOUT) { //表示可写入数据
        // 将数据写入 PCM 设备
    }
}
```

28.7.2 PCM 播放示例代码

对示例代码 28.5.1 进行修改, 使用 poll I/O 多路复用, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘 ->11](#)、[Linux C 应用编程例程源码 ->28_alsa-lib->pcm_playback_poll.c](#)。

示例代码 28.7.1 PCM 播放示例程序—poll

```
*****
```

Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.

文件名 : pcm_playback_poll.c

作者 : 邓涛

版本 : V1.0

描述 : 一个简单地 PCM 播放示例代码--使用 I/O 多路复用 (poll) 写数据

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/7/20 邓涛创建

```

*****/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <poll.h>
#include <alsa/asoundlib.h>

/*****
宏定义
*****/
#define PCM_PLAYBACK_DEV    "hw:0,0"

/*****
WAV 音频文件解析相关数据结构申明
*****/
typedef struct WAV_RIFF {
    char ChunkID[4];           /* "RIFF" */
    u_int32_t ChunkSize;      /* 从下一个地址开始到文件末尾的总字节数 */
    char Format[4];           /* "WAVE" */
} __attribute__((packed)) RIFF_t;

typedef struct WAV_FMT {
    char Subchunk1ID[4];      /* "fmt " */
    u_int32_t Subchunk1Size;  /* 16 for PCM */
    u_int16_t AudioFormat;    /* PCM = 1*/
    u_int16_t NumChannels;    /* Mono = 1, Stereo = 2, etc. */
    u_int32_t SampleRate;     /* 8000, 44100, etc. */
    u_int32_t ByteRate;       /* = SampleRate * NumChannels * BitsPerSample/8 */
    u_int16_t BlockAlign;     /* = NumChannels * BitsPerSample/8 */
    u_int16_t BitsPerSample;  /* 8bits, 16bits, etc. */
} __attribute__((packed)) FMT_t;
static FMT_t wav_fmt;

```

```

typedef struct WAV_DATA {
    char Subchunk2ID[4];           /* "data" */
    u_int32_t Subchunk2Size;      /* data size */
} __attribute__((packed)) DATA_t;

/*****
static 静态全局变量定义
*****/

static snd_pcm_t *pcm = NULL;    //pcm 句柄
static unsigned int buf_bytes;  //应用程序缓冲区的大小（字节为单位）
static void *buf = NULL;        //指向应用程序缓冲区的指针
static int fd = -1;              //指向 WAV 音频文件的文件描述符
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16; //周期数（设备驱动层 buffer 的大小）

static struct pollfd *pfd = NULL;
static int count;

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_PLAYBACK_DEV, SND_PCM_STREAM_PLAYBACK, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_PLAYBACK_DEV, snd_strerror(ret));
        return -1;
    }

    /* 实例化 hwparams 对象 */
    snd_pcm_hw_params_malloc(&hwparams);

    /* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
    ret = snd_pcm_hw_params_any(pcm, hwparams);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /*****
    设置参数
    *****/

```



```
*****/
/* 设置访问类型: 交错模式 */
ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置数据格式: 有符号 16 位、小端模式 */
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, wav_fmt.SampleRate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, wav_fmt.NumChannels);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期大小: period_size */
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期数 (驱动层 buffer 的大小): periods */
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
    goto err2;
}
```

```
/* 使配置生效 */
ret = snd_pcm_hw_params(pcm, hwparams);
snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
    goto err1;
}

buf_bytes = period_size * wav_fmt.BlockAlign; //变量赋值, 一个周期的字节大小
return 0;

err2:
snd_pcm_hw_params_free(hwparams); //释放内存
err1:
snd_pcm_close(pcm); //关闭 pcm 设备
return -1;
}

static int open_wav_file(const char *file)
{
    RIFF_t wav_riff;
    DATA_t wav_data;
    int ret;

    fd = open(file, O_RDONLY);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", file, strerror(errno));
        return -1;
    }

    /* 读取 RIFF chunk */
    ret = read(fd, &wav_riff, sizeof(RIFF_t));
    if (sizeof(RIFF_t) != ret) {
        if (0 > ret)
            perror("read error");
        else
            fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    if (strcmp("RIFF", wav_riff.ChunkID, 4) || //校验
        strcmp("WAVE", wav_riff.Format, 4)) {
```

```
fprintf(stderr, "check error: %s\n", file);
close(fd);
return -1;
}

/* 读取 sub-chunk-fmt */
ret = read(fd, &wav_fmt, sizeof(FMT_t));
if (sizeof(FMT_t) != ret) {
    if (0 > ret)
        perror("read error");
    else
        fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

if (strcmp("fmt ", wav_fmt.Subchunk1ID, 4) != 0) { //校验
    fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

/* 打印音频文件的信息 */
printf("<<<<音频文件格式信息>>>>\n\n");
printf("  file name:      %s\n", file);
printf("  Subchunk1Size: %u\n", wav_fmt.Subchunk1Size);
printf("  AudioFormat:    %u\n", wav_fmt.AudioFormat);
printf("  NumChannels:    %u\n", wav_fmt.NumChannels);
printf("  SampleRate:     %u\n", wav_fmt.SampleRate);
printf("  ByteRate:       %u\n", wav_fmt.ByteRate);
printf("  BlockAlign:     %u\n", wav_fmt.BlockAlign);
printf("  BitsPerSample: %u\n\n", wav_fmt.BitsPerSample);

/* sub-chunk-data */
if (0 > lseek(fd, sizeof(RIFF_t) + 8 + wav_fmt.Subchunk1Size,
             SEEK_SET)) {
    perror("lseek error");
    close(fd);
    return -1;
}

while(sizeof(DATA_t) == read(fd, &wav_data, sizeof(DATA_t))) {
```

```
/* 找到 sub-chunk-data */
if (!strcmp("data", wav_data.Subchunk2ID, 4))//校验
    return 0;

if (0 > lseek(fd, wav_data.Subchunk2Size, SEEK_CUR)) {
    perror("lseek error");
    close(fd);
    return -1;
}
}

fprintf(stderr, "check error: %s\n", file);
return -1;
}

static int snd_pcm_poll_init(void)
{
    int ret;

    /* 获取 PCM 句柄的轮询描述符计数 */
    count = snd_pcm_poll_descriptors_count(pcm);
    if (0 >= count) {
        fprintf(stderr, "Invalid poll descriptors count\n");
        return -1;
    }

    /* 分配内存 */
    pfd = calloc(count, sizeof(struct pollfd));
    if (NULL == pfd) {
        perror("calloc error");
        return -1;
    }

    /* 填充 pfd */
    ret = snd_pcm_poll_descriptors(pcm, pfd, count);
    if (0 > ret)
        return -1;

    return 0;
}

/*****
main 主函数
```

```
*****/
int main(int argc, char *argv[])
{
    unsigned short revents;
    snd_pcm_sframes_t avail;
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <audio_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 打开 WAV 音频文件 */
    if (open_wav_file(argv[1]))
        exit(EXIT_FAILURE);

    /* 初始化 PCM Playback 设备 */
    if (snd_pcm_init())
        goto err1;

    /* 申请读缓冲区 */
    buf = malloc(buf_bytes);
    if (NULL == buf) {
        perror("malloc error");
        goto err2;
    }

    /* I/O 多路复用 poll 初始化 */
    if (snd_pcm_poll_init())
        goto err3;

    for (;;) {

        ret = poll(pfds, count, -1); /*调用 poll
        if (0 > ret) {
            perror("poll error");
            goto err3;
        }

        ret = snd_pcm_poll_descriptors_revents(pcm, pfds, count, &revents);
        if (0 > ret)
            goto err3;
        if (revents & POLLERR)
```

```

    goto err3;
if (revents & POLLOUT) {    //可写数据
    avail = snd_pcm_avail_update(pcm); //获取环形缓冲区中有多少帧数据需要填充
    while (avail >= period_size) { //我们一次写入一个周期

        memset(buf, 0x00, buf_bytes); //buf 清零
        ret = read(fd, buf, buf_bytes);
        if (0 >= ret)
            goto err3;

        ret = snd_pcm_writei(pcm, buf, period_size);
        if (0 > ret) {
            fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
            goto err3;
        }
        else if (ret < period_size) {
            if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
                perror("lseek error");
                goto err3;
            }
        }

        avail = snd_pcm_avail_update(pcm); //再次获取、更新 avail
    }
}

err3:
    free(buf); //释放内存
err2:
    snd_pcm_close(pcm); //关闭 pcm 设备
err1:
    close(fd); //关闭打开的音频文件
    exit(EXIT_FAILURE);
}

```

28.7.3 PCM 录音示例代码

对示例代码 28.5.2 进行修改, 使用 poll I/O 多路复用, 示例代码如下所示:

本例程源码对应的路径为: [开发板光盘 ->11、Linux C 应用编程例程源码 ->28_alsa-lib->pcm_capture_poll.c](#)。

示例代码 28.7.2 PCM 录音示例程序—poll

Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.

文件名 : pcm_capture_poll.c

作者 : 邓涛

版本 : V1.0

描述 : 一个简单地 PCM 音频采集示例代码--使用 I/O 多路复用 (poll) 读数据

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/7/20 邓涛创建

```

*****/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <poll.h>
#include <alsa/asoundlib.h>

/*****
宏定义
*****/
#define PCM_CAPTURE_DEV    "hw:0,0"

/*****
static 静态全局变量定义
*****/
static snd_pcm_t *pcm = NULL;           //pcm 句柄
static snd_pcm_uframes_t period_size = 1024; //周期大小 (单位: 帧)
static unsigned int periods = 16;       //周期数 (buffer 的大小)
static unsigned int rate = 44100;      //采样率

static struct pollfd *pfd = NULL;
static int count;

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_CAPTURE_DEV, SND_PCM_STREAM_CAPTURE, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_CAPTURE_DEV, snd_strerror(ret));
        return -1;
    }
}

```

```
}

/* 实例化 hwparams 对象 */
snd_pcm_hw_params_malloc(&hwparams);

/* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
ret = snd_pcm_hw_params_any(pcm, hwparams);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
    goto err2;
}

/*****
设置参数
*****/
/* 设置访问类型: 交错模式 */
ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置数据格式: 有符号 16 位、小端模式 */
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, rate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, 2);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}
```



```
/* 设置周期大小: period_size */
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期数 (buffer 的大小): periods */
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 使配置生效 */
ret = snd_pcm_hw_params(pcm, hwparams);
snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
    goto err1;
}

return 0;

err2:
snd_pcm_hw_params_free(hwparams); //释放内存
err1:
snd_pcm_close(pcm); //关闭 pcm 设备
return -1;
}

static int snd_pcm_poll_init(void)
{
    int ret;

    /* 获取 PCM 句柄的轮询描述符计数 */
    count = snd_pcm_poll_descriptors_count(pcm);
    if (0 >= count) {
        fprintf(stderr, "Invalid poll descriptors count\n");
        return -1;
    }

    /* 分配内存 */
```

```
    pfdes = calloc(count, sizeof(struct pollfd));
    if (NULL == pfdes) {
        perror("calloc error");
        return -1;
    }

    /* 填充 pfdes */
    ret = snd_pcm_poll_descriptors(pcm, pfdes, count);
    if (0 > ret)
        return -1;

    return 0;
}

/*****
main 主函数
*****/
int main(int argc, char *argv[])
{
    unsigned char *buf = NULL;
    unsigned int buf_bytes;
    unsigned short revents;
    snd_pcm_sframes_t avail;
    int fd = -1;
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 初始化 PCM Capture 设备 */
    if (snd_pcm_init())
        exit(EXIT_FAILURE);

    /* 申请读缓冲区 */
    buf_bytes = period_size * 4;    //字节大小 = 周期大小*帧的字节大小 16 位双声道
    buf = malloc(buf_bytes);
    if (NULL == buf) {
        perror("malloc error");
        goto err1;
    }
}
```

```
/* 打开一个新建文件 */
fd = open(argv[1], O_WRONLY | O_CREAT | O_EXCL);
if (0 > fd) {
    fprintf(stderr, "open error: %s: %s\n", argv[1], strerror(errno));
    goto err2;
}

/* I/O 多路复用 poll 初始化 */
if (snd_pcm_poll_init())
    goto err3;

/* 开始录音 */
ret = snd_pcm_start(pcm);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_start error: %s\n", snd_strerror(ret));
    goto err3;
}

for (;;) {

    ret = poll(pfds, count, -1); //调用 poll
    if (0 > ret) {
        perror("poll error");
        goto err3;
    }

    ret = snd_pcm_poll_descriptors_revents(pcm, pfds, count, &revents);
    if (0 > ret)
        goto err3;
    if (revents & POLLERR)
        goto err3;
    if (revents & POLLIN) { //可读数据
        avail = snd_pcm_avail_update(pcm); //检查有多少帧数据可读
        while (avail >= period_size) { //每次读取一个周期

            ret = snd_pcm_readi(pcm, buf, period_size); //读取 PCM 数据 一个周期
            if (0 > ret) {
                fprintf(stderr, "snd_pcm_readi error: %s\n", snd_strerror(ret));
                goto err3;
            }

            ret = write(fd, buf, ret * 4); //将读取到的数据写入文件中
            if (0 >= ret)
```

```

        goto err3;

        avail = snd_pcm_avail_update(pcm); //再次读取、更新 avail
    }
}

err3:
    close(fd); //关闭文件
err2:
    free(buf); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    exit(EXIT_FAILURE);
}

```

28.8 PCM 设备的状态

本小节向大家介绍 PCM 设备的状态有哪些，alsa-lib 提供了函数 `snd_pcm_state()` 用于获取 PCM 设备当前的状态，其函数原型如下所示：

```
snd_pcm_state_t snd_pcm_state(snd_pcm_t *pcm);
```

可以看到它的返回值是一个 `snd_pcm_state_t` 类型的变量，`snd_pcm_state_t` 其实是一个枚举类型，描述了 PCM 设备包含的所有状态，如下所示：

```

enum snd_pcm_state_t {
    SND_PCM_STATE_OPEN = 0,
    SND_PCM_STATE_SETUP,
    SND_PCM_STATE_PREPARED,
    SND_PCM_STATE_RUNNING,
    SND_PCM_STATE_XRUN,
    SND_PCM_STATE_DRAINING,
    SND_PCM_STATE_PAUSED,
    SND_PCM_STATE_SUSPENDED,
    SND_PCM_STATE_DISCONNECTED,
    SND_PCM_STATE_LAST = SND_PCM_STATE_DISCONNECTED,
    SND_PCM_STATE_PRIVATE1 = 1024
}

```

SND_PCM_STATE_OPEN

该状态表示 PCM 设备处于打开状态，譬如当调用 `snd_pcm_open()` 后，PCM 设备就处于该状态。

SND_PCM_STATE_SETUP

alsa-lib 文档中的解释为“Setup installed”！该状态表示设备已经初始化完成了，参数已经配置好了。

SND_PCM_STATE_PREPARED

该状态表示设备已经准备好了，可以开始了“Ready to start”！譬如可以开始播放了、可以开始录音了。

前面提到了这个状态，当应用程序调用 `snd_pcm_hw_params()` 函数之后，设备就处于 `SND_PCM_STATE_PREPARED` 状态了。应用程序中，可以调用 `snd_pcm_prepare()` 函数使设备处于 `SND_PCM_STATE_PREPARED` 状态，该函数原型如下所示：

```
int snd_pcm_prepare(snd_pcm_t *pcm);
```

该行函数调用成功返回 0，失败将返回一个负数错误码。

函数调用成功，PCM 设备将处于 `SND_PCM_STATE_PREPARED` 状态。事实上，应用程序调用时 `snd_pcm_hw_params()` 时，函数内部会自动调用 `snd_pcm_prepare()`，所以为什么调用 `snd_pcm_hw_params()` 之后设备就已经处于 `SND_PCM_STATE_PREPARED` 状态了；调用 `snd_pcm_hw_params()` 函数，其实应该发生了两种状态的转变为：首先由 `SND_PCM_STATE_OPEN` 变为 `SND_PCM_STATE_SETUP` 状态、再由 `SND_PCM_STATE_SETUP` 变为 `SND_PCM_STATE_PREPARED` 状态。

SND_PCM_STATE_RUNNING

该状态表示设备正在运行，譬如正在播放、正在录音。

上小节我们提到，应用程序可以调用 `snd_pcm_start()` 函数以启动 PCM 设备，启动成功之后，设备开始播放或采集，此时设备处于 `SND_PCM_STATE_RUNNING` 状态。

此外，当设备处于 `SND_PCM_STATE_PREPARED` 状态时，应用程序调用 `snd_pcm_readi/snd_pcm_writeti` 进行读写数据时，这些函数内部会自动调用 `snd_pcm_start()` 函数；譬如播放模式下，调用 `snd_pcm_writeti` 写入数据后，会自动开启 PCM 设备进行播放，这里要注意，一定是在数据写入到环形缓冲区之后、才开启 PCM 设备播放音频，因为一旦开启之后，环形缓冲区中必须要有至少一个周期的数据可供音频设备播放，否则将会发生欠载（underrun）、函数调用以错误形式返回；在录音模式下，调用 `snd_pcm_readi()` 函数后，自动开启 PCM 进行音频采集。

所以这就是为什么示例代码 28.5.1、示例代码 28.5.2、示例代码 28.6.1 这几个示例中并没有调用 `snd_pcm_start()` 函数的原因。

当设备处于运行状态时，应用程序可调用 `snd_pcm_drop()` 或 `snd_pcm_drain()` 函数使设备停止运行，譬如停止播放、停止音频采集；它们的函数原型如下所示：

```
int snd_pcm_drain(snd_pcm_t *pcm);
```

```
int snd_pcm_drop(snd_pcm_t *pcm);
```

函数调用成功返回 0；失败返回负值错误码。

这两个函数都可使设备停止运行，它们的区别如下：

- `snd_pcm_drop()` 函数将立即停止 PCM，丢弃挂起的帧；
- `snd_pcm_drain()` 函数并不会立即停止 PCM，而是处理完挂起的帧之后再停止 PCM；对于播放，会等待所有待播放的帧播放完毕（应该就是环形缓冲区中的待播放数据），然后停止 PCM；对于录音，停止 PCM 之前会检索残留帧。

当调用 `snd_pcm_drop()` 或 `snd_pcm_drain()` 停止 PCM 设备后，设备将回到 `SND_PCM_STATE_SETUP` 状态。

SND_PCM_STATE_XRUN

当发生 XRUN 时，设备会处于 `SND_PCM_STATE_XRUN` 状态，XRUN 前面给大家解释过了，这里不再重述！当处于 `SND_PCM_STATE_XRUN` 状态时，应用程序可以调用 `snd_pcm_prepare()` 使设备恢复，使其回到 `SND_PCM_STATE_PREPARED` 状态。

SND_PCM_STATE_DRAINING

这个状态笔者没弄清楚，alsa-lib 文档中的解释为 “Draining: running (playback) or stopped (capture)”。

SND_PCM_STATE_PAUSED

pause 就是暂停的意思, 所以该状态表示设备处于暂停状态。譬如当设备正在运行时 (也就是处于 SND_PCM_STATE_RUNNING 状态), 应用程序调用 `snd_pcm_pause()` 函数可让设备暂停, 其函数原型如下所示:

```
int snd_pcm_pause(snd_pcm_t *pcm, int enable);
```

函数 `snd_pcm_pause()` 既可以使用的设备暂停、同样也可使其恢复 (从暂停恢复运行, 即 SND_PCM_STATE_RUNNING--->SND_PCM_STATE_RUNNING), 通过参数 `enable` 控制; 当 `enable` 等于 1, 表示使设备暂停; `enable` 等于 0 表示使设备恢复运行。

`snd_pcm_pause()` 函数调用成功返回 0; 失败返回一个负值错误码。

这里有个问题需要注意, 并不是所有的音频设备硬件上支持暂停的功能, 可以通过 `snd_pcm_hw_params_can_pause()` 函数来判断设备是否支持暂停, 其函数原型如下所示:

```
int snd_pcm_hw_params_can_pause(const snd_pcm_hw_params_t *params);
```

函数返回 1 表示硬件支持暂停; 返回 0 表示硬件不支持暂停。

SND_PCM_STATE_SUSPENDED

该状态表示硬件已经挂起 `suspended`, 如果硬件发生了挂起, 应用程序可以调用 `snd_pcm_resume()` 函数从挂起中恢复, 并确保不会丢失样本数据 (精细恢复)。`snd_pcm_resume()` 函数原型如下所示:

```
int snd_pcm_resume(snd_pcm_t *pcm);
```

函数调用成功返回 0; 失败返回一个负值错误码。

当然, 并非所有硬件都支持此功能, 可以调用 `snd_pcm_hw_params_can_resume()` 函数判断硬件是否支持从挂起中恢复, 其函数原型如下所示:

```
int snd_pcm_hw_params_can_resume(const snd_pcm_hw_params_t *params);
```

函数调用返回 1 表示支持, 返回 0 表示不支持。

SND_PCM_STATE_DISCONNECTED

该状态表示硬件已经断开连接。

状态之间的转换

通过上面的介绍, 我们已经知道了 PCM 设备的几种不同的状态、以及它们的一个转换关系, 为了能够加深大家的印象, 笔者对其进行了整理, 主要整理了 SND_PCM_STATE_OPEN、SND_PCM_STATE_SETUP、SND_PCM_STATE_PREPARED、SND_PCM_STATE_RUNNING、SND_PCM_STATE_XRUN 以及 SND_PCM_STATE_PAUSED 这 6 种状态之间的转换关系, 如下图所示:

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/7/20 邓涛创建

```

*****/

```

```

#include <stdio.h>

```

```

#include <stdlib.h>

```

```

#include <errno.h>

```

```

#include <string.h>

```

```

#include <termios.h>

```

```

#include <signal.h>

```

```

#include <alsa/asoundlib.h>

```

```

/*****

```

宏定义

```

*****/

```

```

#define PCM_PLAYBACK_DEV    "hw:0,0"

```

```

/*****

```

WAV 音频文件解析相关数据结构申明

```

*****/

```

```

typedef struct WAV_RIFF {

```

```

    char ChunkID[4];           /* "RIFF" */

```

```

    u_int32_t ChunkSize;      /* 从下一个地址开始到文件末尾的总字节数 */

```

```

    char Format[4];           /* "WAVE" */

```

```

} __attribute__((packed)) RIFF_t;

```

```

typedef struct WAV_FMT {

```

```

    char Subchunk1ID[4];      /* "fmt " */

```

```

    u_int32_t Subchunk1Size;   /* 16 for PCM */

```

```

    u_int16_t AudioFormat;     /* PCM = 1*/

```

```

    u_int16_t NumChannels;     /* Mono = 1, Stereo = 2, etc. */

```

```

    u_int32_t SampleRate;      /* 8000, 44100, etc. */

```

```

    u_int32_t ByteRate;        /* = SampleRate * NumChannels * BitsPerSample/8 */

```

```

    u_int16_t BlockAlign;      /* = NumChannels * BitsPerSample/8 */

```

```

    u_int16_t BitsPerSample;   /* 8bits, 16bits, etc. */

```

```

} __attribute__((packed)) FMT_t;

```

```

static FMT_t wav_fmt;

```

```

typedef struct WAV_DATA {

```

```

    char Subchunk2ID[4];      /* "data" */

```

```

    u_int32_t Subchunk2Size;   /* data size */

```

```

} __attribute__((packed)) DATA_t;

```



```

/*****
static 静态全局变量定义
*****/
static snd_pcm_t *pcm = NULL;           //pcm 句柄
static unsigned int buf_bytes;         //应用程序缓冲区的大小（字节为单位）
static void *buf = NULL;              //指向应用程序缓冲区的指针
static int fd = -1;                   //指向 WAV 音频文件的文件描述符
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16;     //周期数（设备驱动层 buffer 的大小）
static struct termios old_cfg;        //用于保存终端当前的配置参数

/*****
static 静态函数
*****/
static void snd_playback_async_callback(snd_async_handler_t *handler)
{
    snd_pcm_t *handle = snd_async_handler_get_pcm(handler); //获取 PCM 句柄
    snd_pcm_sframes_t avail;
    int ret;

    avail = snd_pcm_avail_update(handle); //获取环形缓冲区中有多少帧数据需要填充
    while (avail >= period_size) { //我们一次写入一个周期

        memset(buf, 0x00, buf_bytes); //buf 清零
        ret = read(fd, buf, buf_bytes);
        if (0 >= ret)
            goto out;

        ret = snd_pcm_writei(handle, buf, period_size);
        if (0 > ret) {
            fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
            goto out;
        }
        else if (ret < period_size) { //实际写入的帧数小于指定的帧数
            //此时我们需要调整下音频文件的读位置
            //将读位置向后移动（往回移）(period_size-ret)*frame_bytes 个字节
            //frame_bytes 表示一帧的字节大小
            if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
                perror("lseek error");
                goto out;
            }
        }
    }
}

```

```

        avail = snd_pcm_avail_update(handle); //再次获取、更新 avail
    }

    return;
out:
    snd_pcm_drain(pcm); //停止 PCM
    snd_pcm_close(handle); //关闭 pcm 设备
    tcsetattr(STDIN_FILENO, TCSANOW, &old_cfg); //退出前恢复终端的状态
    free(buf);
    close(fd); //关闭打开的音频文件
    exit(EXIT_FAILURE); //退出程序
}

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    snd_async_handler_t *async_handler = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_PLAYBACK_DEV, SND_PCM_STREAM_PLAYBACK, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_PLAYBACK_DEV, snd_strerror(ret));
        return -1;
    }

    /* 实例化 hwparams 对象 */
    snd_pcm_hw_params_malloc(&hwparams);

    /* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
    ret = snd_pcm_hw_params_any(pcm, hwparams);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /******
    设置参数
    *****/
    /* 设置访问类型: 交错模式 */
    ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);

```

```
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置数据格式: 有符号 16 位、小端模式 */
ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, wav_fmt.SampleRate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, wav_fmt.NumChannels);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期大小: period_size */
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期数 (驱动层环形缓冲区 buffer 的大小): periods */
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 使配置生效 */
ret = snd_pcm_hw_params(pcm, hwparams);
snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
```

```
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
        goto err1;
    }

    buf_bytes = period_size * wav_fmt.BlockAlign; //变量赋值, 一个周期的字节大小

    /* 注册异步处理函数 */
    ret = snd_async_add_pcm_handler(&async_handler, pcm, snd_playback_async_callback, NULL);
    if (0 > ret) {
        fprintf(stderr, "snd_async_add_pcm_handler error: %s\n", snd_strerror(ret));
        goto err1;
    }

    return 0;

err2:
    snd_pcm_hw_params_free(hwparams); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    return -1;
}

static int open_wav_file(const char *file)
{
    RIFF_t wav_riff;
    DATA_t wav_data;
    int ret;

    fd = open(file, O_RDONLY);
    if (0 > fd) {
        fprintf(stderr, "open error: %s: %s\n", file, strerror(errno));
        return -1;
    }

    /* 读取 RIFF chunk */
    ret = read(fd, &wav_riff, sizeof(RIFF_t));
    if (sizeof(RIFF_t) != ret) {
        if (0 > ret)
            perror("read error");
        else
            fprintf(stderr, "check error: %s\n", file);
        close(fd);
    }
}
```

```
        return -1;
    }

    if (strcmp("RIFF", wav_riff.ChunkID, 4) || //校验
        strcmp("WAVE", wav_riff.Format, 4)) {
        fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    /* 读取 sub-chunk-fmt */
    ret = read(fd, &wav_fmt, sizeof(FMT_t));
    if (sizeof(FMT_t) != ret) {
        if (0 > ret)
            perror("read error");
        else
            fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    if (strcmp("fmt ", wav_fmt.Subchunk1ID, 4) || //校验
        strcmp("fmt ", wav_fmt.Subchunk1ID, 4)) {
        fprintf(stderr, "check error: %s\n", file);
        close(fd);
        return -1;
    }

    /* 打印音频文件的信息 */
    printf("<<<<音频文件格式信息>>>>\n\n");
    printf("  file name:      %s\n", file);
    printf("  Subchunk1Size: %u\n", wav_fmt.Subchunk1Size);
    printf("  AudioFormat:    %u\n", wav_fmt.AudioFormat);
    printf("  NumChannels:    %u\n", wav_fmt.NumChannels);
    printf("  SampleRate:     %u\n", wav_fmt.SampleRate);
    printf("  ByteRate:       %u\n", wav_fmt.ByteRate);
    printf("  BlockAlign:     %u\n", wav_fmt.BlockAlign);
    printf("  BitsPerSample: %u\n\n", wav_fmt.BitsPerSample);

    /* sub-chunk-data */
    if (0 > lseek(fd, sizeof(RIFF_t) + 8 + wav_fmt.Subchunk1Size,
                SEEK_SET)) {
        perror("lseek error");
        close(fd);
    }
}
```

```

        return -1;
    }

    while(sizeof(DATA_t) == read(fd, &wav_data, sizeof(DATA_t))) {

        /* 找到 sub-chunk-data */
        if (!strcmp("data", wav_data.Subchunk2ID, 4))//校验
            return 0;

        if (0 > lseek(fd, wav_data.Subchunk2Size, SEEK_CUR)) {
            perror("lseek error");
            close(fd);
            return -1;
        }
    }

    fprintf(stderr, "check error: %s\n", file);
    return -1;
}

/*****
main 主函数
*****/
int main(int argc, char *argv[])
{
    snd_pcm_sframes_t avail;
    struct termios new_cfg;
    sigset_t sset;
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <audio_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 屏蔽 SIGIO 信号 */
    sigemptyset(&sset);
    sigaddset(&sset, SIGIO);
    sigprocmask(SIG_BLOCK, &sset, NULL);

    /* 打开 WAV 音频文件 */
    if (open_wav_file(argv[1]))
        exit(EXIT_FAILURE);
}

```

```
/* 初始化 PCM Playback 设备 */
if (snd_pcm_init())
    goto err1;

/* 申请读缓冲区 */
buf = malloc(buf_bytes);
if (NULL == buf) {
    perror("malloc error");
    goto err2;
}

/* 终端配置 */
tcgetattr(STDIN_FILENO, &old_cfg); //获取终端<标准输入-标准输出构成了一套终端>
memcpy(&new_cfg, &old_cfg, sizeof(struct termios)); //备份
new_cfg.c_lflag &= ~ICANON; //将终端设置为非规范模式
new_cfg.c_lflag &= ~ECHO; //禁用回显
tcsetattr(STDIN_FILENO, TCSANOW, &new_cfg); //使配置生效

/* 播放: 先将环形缓冲区填满数据 */
avail = snd_pcm_avail_update(pcm); //获取环形缓冲区中有多少帧数据需要填充
while (avail >= period_size) { //我们一次写入一个周期

    memset(buf, 0x00, buf_bytes); //buf 清零
    ret = read(fd, buf, buf_bytes);
    if (0 >= ret)
        goto err3;

    ret = snd_pcm_writei(pcm, buf, period_size); //向环形缓冲区中写入数据
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
        goto err3;
    }
    else if (ret < period_size) { //实际写入的帧数小于指定的帧数
        //此时我们需要调整下音频文件的读位置
        //将读位置向后移动 (来回移) (period_size-ret)*frame_bytes 个字节
        //frame_bytes 表示一帧的字节大小
        if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
            perror("lseek error");
            goto err3;
        }
    }
}
}
```

```

    avail = snd_pcm_avail_update(pcm); //再次获取、更新 avail
}

sigprocmask(SIG_UNBLOCK, &sset, NULL); //取消 SIGIO 信号屏蔽

char ch;
for (;;) {

    ch = getchar(); //获取用户输入的控制字符
    switch (ch) {
        case 'q': //Q 键退出程序
            sigprocmask(SIG_BLOCK, &sset, NULL); //屏蔽 SIGIO 信号
            goto err3;
        case ' ': //空格暂停/恢复
            switch (snd_pcm_state(pcm)) {

                case SND_PCM_STATE_PAUSED: //如果是暂停状态则恢复运行
                    ret = snd_pcm_pause(pcm, 0);
                    if (0 > ret)
                        fprintf(stderr, "snd_pcm_pause error: %s\n", snd_strerror(ret));
                    break;
                case SND_PCM_STATE_RUNNING: //如果是运行状态则暂停
                    ret = snd_pcm_pause(pcm, 1);
                    if (0 > ret)
                        fprintf(stderr, "snd_pcm_pause error: %s\n", snd_strerror(ret));
                    break;
            }
            break;
    }
}

err3:
    snd_pcm_drop(pcm); //停止 PCM
    tcsetattr(STDIN_FILENO, TCSANOW, &old_cfg); //退出前恢复终端的状态
    free(buf); //释放内存
err2:
    snd_pcm_close(pcm); //关闭 pcm 设备
err1:
    close(fd); //关闭打开的音频文件
    exit(EXIT_FAILURE);
}

```

上述示例程序是在示例代码 28.6.1 基础上进行修改了, 加入了用户控制单元, 程序设定:

- q: 在终端按下 q 键退出应用程序;

- 终端按下空格键暂停播放, 再次按下恢复播放。

下面给大家简单地介绍下上述示例代码的设计, 在 main() 函数中, 我们首先屏蔽了 SIGIO 信号, 如下:

```
/* 屏蔽 SIGIO 信号 */
sigemptyset(&ssset);
sigaddset(&ssset, SIGIO);
sigprocmask(SIG_BLOCK, &ssset, NULL);
```

这主要是为了程序设计上的安全考虑, 等把环形缓冲区填满数据之后, 再取消 SIGIO 信号屏蔽。当然, 你也可以不这样做。

接着打开用户传入的音频文件、初始化 PCM 播放设备、申请应用程序所需的缓冲区:

```
/* 打开 WAV 音频文件 */
if (open_wav_file(argv[1]))
    exit(EXIT_FAILURE);

/* 初始化 PCM Playback 设备 */
if (snd_pcm_init())
    goto err1;

/* 申请读缓冲区 */
buf = malloc(buf_bytes);
if (NULL == buf) {
    perror("malloc error");
    goto err2;
}
```

接着对终端进行设置, 将终端配置为非规范模式、取消回显, 配置为非规范模式之后, 用户输入的字符会直接被应用程序读取到, 而无需按下回车键; 取消回显, 意味着用户输入的字符, 在终端不会显示出来, 这些内容在串口应用编程章节给大家详细介绍过, 这里就不再啰嗦!

```
/* 终端配置 */
tcgetattr(STDIN_FILENO, &old_cfg); //获取终端<标准输入-标准输出构成了一套终端>
memcpy(&new_cfg, &old_cfg, sizeof(struct termios)); //备份
new_cfg.c_lflag &= ~ICANON; //将终端设置为非规范模式
new_cfg.c_lflag &= ~ECHO; //禁用回显
tcsetattr(STDIN_FILENO, TCSANOW, &new_cfg); //使配置生效
```

接下来将数据写入环形缓冲区, 开始播放。

取消 SIGIO 信号信号屏蔽。

最后进入 for() 循环中, 通过 getchar() 读取用户输入的字符, 用户输入 q 时退出程序, 这里需要注意, 退出程序时需要调用 tcsetattr() 将终端配置参数恢复到之前的状态, 否则你的终端将可能会出现下面这种情况:

```
root@ATK-IMX6U:~# root@ATK-IMX6U:~# root@ATK-IMX6U:~# root@ATK-IMX6U:~# root@ATK-IMX6U:~# root@ATK-IMX6U:~#
```

这个时候你就只能重启了。

用户输入空格暂停或恢复, 调用 snd_pcm_pause() 实现暂停/恢复。

代码比较简单, 笔者也就不再多说了!

编译示例代码

执行命令编译应用程序:

```
 ${CC} -o testApp testApp.c -lasound
```

测试应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统/home/root 目录下, 并准备一个 WAV 音频文件, 接着我们执行测试程序:

```
root@ATK-IMX6U:~# ls
driver  EXO-Overdose.wav  QDesktop-fb  shell  testApp
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./testApp ./EXO-Overdose.wav
<<<<音频文件格式信息>>>>

file name:      ./EXO-Overdose.wav
Subchunk1Size: 16
AudioFormat:    1
NumChannels:    2
SampleRate:     44100
ByteRate:       176400
BlockAlign:     4
BitsPerSample:  16
```

图 28.8.2 运行测试程序

运行之后, 开始播放音乐, 此时我们可以通过空格键来暂停播放、再按空格键恢复播放, 按 q 键退出程序, 大家自己去测试。

Tips: 本测试程序不能放在后台运行, 一旦放入后台, 程序将停止 (不是终止、是暂停运行), 因为这个程序在设计逻辑上就不符合放置在后台, 因为程序中会读取用户从终端 (标准输入) 输入的字符, 如果放入后台, 那用户输入的字符就不可能被该程序所读取到, 这是其一; 其二, 程序中修改了终端的配置。

28.8.2 snd_pcm_readi/snd_pcm_writei 错误处理

当 `snd_pcm_readi/snd_pcm_writei` 调用出错时, 会返回一个小于 0 (负值) 的错误码, 可调用 `snd_strerror()` 函数获取对应的错误描述信息。前面的示例代码中我们并没有对 `snd_pcm_readi/snd_pcm_writei` 的错误返回做过多、细节的处理, 而是简单地在出错之后退出。

事实上, 当调用 `snd_pcm_readi/snd_pcm_writei` 出错时, 可根据不同的情况作进一步的处理, 在 `alsa-lib` 文档中有介绍到, `snd_pcm_readi/snd_pcm_writei` 函数的不同错误返回值, 表示不同的含义, 如下所示:

• `snd_pcm_writei()`

```
snd_pcm_sframes_t snd_pcm_writei (snd_pcm_t* pcm,
                                const void * buffer,
                                snd_pcm_uframes_t size)
)
```

Write interleaved frames to a PCM.

Parameters

pcm PCM handle
buffer frames containing buffer
size frames to be written

Returns

a positive number of frames actually written otherwise a negative error code

Return values

-EBADFD PCM is not in the right state (`SND_PCM_STATE_PREPARED` or `SND_PCM_STATE_RUNNING`)
 -EPIPE an underrun occurred
 -ESTRPIPE a suspend event occurred (stream is suspended and waiting for an application recovery)

图 28.8.3 `snd_pcm_writei` 函数的错误返回值描述

`snd_pcm_readi()`函数与它相同。

当返回值等于-EBADFD, 表示 PCM 设备的状态不对, 因为执行 `snd_pcm_readi/snd_pcm_writei` 读取/写入数据需要 PCM 设备处于 `SND_PCM_STATE_PREPARED` 或 `SND_PCM_STATE_RUNNING` 状态, 前面已经详细地给大家介绍了 PCM 设备的状态间转换问题。

当返回值等于-EPIPE, 表示发生了 XRUN, 此时可以怎么做呢? 这个可以根据自己的实际需要进行处理, 譬如调用 `snd_pcm_drop()`停止 PCM 设备, 或者调用 `snd_pcm_prepare()`使设备恢复进入准备状态。

当返回值等于-ESTRPIPE, 表示硬件发生了挂起, 此时 PCM 设备处于 `SND_PCM_STATE_SUSPENDED` 状态, 譬如你可以调用 `snd_pcm_resume()`函数从挂起中精确恢复, 如果硬件不支持, 还可调用 `snd_pcm_prepare()`函数使设备进入准备状态, 或者执行其它的处理, 根据应用需求的进行相应的处理。

以上给大家介绍了调用 `snd_pcm_readi/snd_pcm_writei` 函数出错时的一些情况以及可以采取的一些措施!

28.9 混音器设置

前面给大家介绍了 `alsa-utils` 提供的两个声卡配置工具: `alsamixer` 和 `amixer`。这两个工具同样是基于 `alsa-lib` 库函数编写的, 本小节我们来学习如何在自己的应用程序中通过调用 `alsa-lib` 库函数对声卡混音器进行配置, 譬如音量调节。

混音器相关的接口在 `alsa-lib` 的 `Mixer Interface` 模块中有介绍, 点击图 28.2.2 中 “Mixer Interface” 可查看混音器相关接口的介绍, 如下所示:

ALSA project - the C library reference

Main Page	Related Pages	Modules	Data Structures	Files	Examples
---------------------------	-------------------------------	-------------------------	---------------------------------	-----------------------	--------------------------

Mixer Interface

Modules

- [Simple Mixer Interface](#)

Macros

```
#define snd_mixer_class_alloc(ptr)
    allocate an invalid snd_mixer_class_t using standard alloca More...
```

Typedefs

```
typedef struct _snd_mixer snd_mixer_t
typedef struct _snd_mixer_class snd_mixer_class_t
typedef struct _snd_mixer_elem snd_mixer_elem_t
typedef int(* snd_mixer_callback_t)(snd_mixer_t *ctl, unsigned int mask, snd_mixer_elem_t *elem)
    Mixer callback function. More...
typedef int(* snd_mixer_elem_callback_t)(snd_mixer_elem_t *elem, unsigned int mask)
    Mixer element callback function. More...
typedef int(* snd_mixer_compare_t)(const snd_mixer_elem_t *e1, const snd_mixer_elem_t *e2)
    Compare function for sorting mixer elements. More...
typedef int(* snd_mixer_event_t)(snd_mixer_class_t *class_, unsigned int mask, snd_mixer_elem_t *elem, snd_mixer_elem_t *melem)
    Event callback for the mixer class. More...
```

Enumerations

```
enum snd_mixer_elem_type { SND_MIXER_ELEM_SIMPLE, SND_MIXER_ELEM_LAST = SND_MIXER_ELEM_SIMPLE }
```

图 28.9.1 Mixer Interface 模块

大家可以简单地浏览下该模块下提供了那些函数，点击函数名可以查看该函数的简单介绍信息。

28.9.1 打开混音器: `snd_mixer_open`

在使用混音器之后，需要打开混音器，调用 `snd_mixer_open()` 函数打开一个空的混音器，其函数原型如下所示：

```
int snd_mixer_open(snd_mixer_t **mixerp, int mode);
```

alsa-lib 使用 `snd_mixer_t` 数据结构描述混音器，调用 `snd_mixer_open()` 函数会实例化一个 `snd_mixer_t` 对象，并将对象的指针（也就是混音器的句柄）通过 `mixerp` 返回出来。参数 `mode` 指定了打开模式，通常设置为 0 使用默认模式即可！

函数调用成功返回 0；失败返回一个小于 0 的错误码。

使用示例：

```
snd_mixer_t *mixer = NULL;
int ret;

ret = snd_mixer_open(&mixer, 0);
if (0 > ret)
    fprintf(stderr, "snd_mixer_open error: %s\n", snd_strerror(ret));
```

28.9.2 Attach 关联设备: `snd_mixer_attach`

调用 `snd_mixer_open()` 函数打开并实例化了一个空的混音器，接下来我们要去关联声卡控制设备，调用 `snd_mixer_attach()` 函数进行关联，其函数原型如下所示：

```
int snd_mixer_attach(snd_mixer_t *mixer, const char *name);
```

参数 `mixer` 对应的是混音器的句柄, 参数 `name` 指定了声卡控制设备的名字, 同样这里使用的也是逻辑设备名, 而非设备节点的名字, 命名方式为 "hw:i", `i` 表示声卡的卡号, 通常一个声卡对应一个控制设备; 譬如 "hw:0" 表示声卡 0 的控制设备, 这其实就对应 /dev/snd/controlC0 设备。与 `snd_pcm_open()` 函数中 PCM 设备的命名一样, `snd_mixer_attach()` 函数中声卡控制设备的命名也有其它方式, 这里暂时先不管这个问题。

调用 `snd_mixer_open()` 函数会将参数 `name` 所指定的控制设备与混音器 `mixer` 进行关联。

函数调用成功返回 0; 失败返回一个小于 0 的错误码。

使用示例:

```
ret = snd_mixer_attach(mixer, "hw:0");
if (0 > ret)
    fprintf(stderr, "snd_mixer_attach error: %s\n", snd_strerror(ret));
```

28.9.3 注册: `snd_mixer_selem_register`

调用 `snd_mixer_selem_register()` 函数注册混音器, 其函数原型如下所示:

```
int snd_mixer_selem_register(
    snd_mixer_t *mixer,
    struct snd_mixer_selem_regopt *options,
    snd_mixer_class_t **classp);
```

参数 `options` 和参数 `classp` 直接设置为 NULL 即可。

函数调用成功返回 0; 失败返回一个小于 0 的错误码。

使用示例:

```
ret = snd_mixer_selem_register(mixer, NULL, NULL);
if (0 > ret)
    fprintf(stderr, "snd_mixer_selem_register error: %s\n", snd_strerror(ret));
```

28.9.4 加载: `snd_mixer_load`

最后需要加载混音器, 调用 `snd_mixer_load()` 函数完成加载, 函数原型如下所示:

```
int snd_mixer_load(snd_mixer_t * mixer);
```

函数调用成功返回 0; 失败返回小于 0 的错误码。

使用示例:

```
ret = snd_mixer_load(mixer);
if (0 > ret)
    fprintf(stderr, "snd_mixer_load error: %s\n", snd_strerror(ret));
```

28.9.5 查找元素

经过上面一系列步骤之后, 接下来就可以使用混音器了, `alsa-lib` 中把混音器的配置项称为元素 (element), 譬如耳机音量调节 `Headphone` 是一个元素、`'Headphone Playback ZC'` 是一个元素、`'Right Output Mixer PCM'` 也是一个元素。

`snd_mixer_first_elem` 和 `snd_mixer_last_elem`

`alsa-lib` 使用数据结构 `snd_mixer_elem_t` 来描述一个元素, 所以一个 `snd_mixer_elem_t` 对象就是一个元素。混音器有很多的元素 (也就是有很多配置项), 通过 `snd_mixer_first_elem()` 函数可以找到混音器的第一个元素, 其函数原型如下所示:

```
snd_mixer_elem_t *snd_mixer_first_elem(snd_mixer_t *mixer);
```

通过 `snd_mixer_last_elem()` 函数可找到混音器的最后一个元素, 如下:

```
snd_mixer_elem_t *snd_mixer_last_elem(snd_mixer_t *mixer);
```

snd_mixer_elem_next 和 snd_mixer_elem_prev

调用 `snd_mixer_elem_next()` 和 `snd_mixer_elem_prev()` 函数可获取指定元素的下一个元素和上一个元素:

```
snd_mixer_elem_t *snd_mixer_elem_next(snd_mixer_elem_t *elem);
```

```
snd_mixer_elem_t *snd_mixer_elem_prev(snd_mixer_elem_t *elem);
```

所以通过 `snd_mixer_first_elem` 和 `snd_mixer_elem_next()` 或者 `snd_mixer_last_elem()` 和 `snd_mixer_elem_prev()` 就可以遍历整个混音器中的所有元素, 如下所示:

```
snd_mixer_elem_t *elem = NULL;
```

```
elem = snd_mixer_first_elem(mixer); //找到第一个元素
```

```
while (elem) {
```

```
.....
```

```
.....
```

```
    snd_mixer_elem_next(elem); //找到下一个元素
```

```
}
```

snd_mixer_selem_get_name

调用 `snd_mixer_selem_get_name()` 函数可获取指定元素的名字, 如下所示:

```
const char *snd_mixer_selem_get_name(snd_mixer_elem_t *elem);
```

获取元素的名字之后, 进行对比, 以确定是否是我们要找的元素:

```
const char *name = snd_mixer_selem_get_name(elem);
```

```
if (!strcmp(name, "Headphone")) {
```

```
    //该配置项是"Headphone"
```

```
}
```

```
else {
```

```
    //该配置项不是"Headphone"
```

```
}
```

28.9.6 获取/更改元素的配置值

前面给大家提到了混音器的配置值有两种类型, 第一种它的配置值是在一个范围内的数值, 譬如音量大小的调节; 另一种则是 `bool` 类型, 用于控制开启或关闭, 譬如 0 表示关闭配置、1 表示使能配置。

snd_mixer_selem_has_playback_volume/snd_mixer_selem_has_capture_volume

我们可以调用 `snd_mixer_selem_has_playback_volume()` (播放) 或 `snd_mixer_selem_has_capture_volume()` (录音) 函数来判断一个指定元素的配置值是否是 `volume` 类型, 也就是上面说的第一种情况。函数原型如下所示:

```
int snd_mixer_selem_has_playback_volume(snd_mixer_elem_t *elem);
```

```
int snd_mixer_selem_has_capture_volume(snd_mixer_elem_t *elem);
```

函数返回 0 表示不是 `volume` 类型; 返回 1 表示是 `volume` 类型。

snd_mixer_selem_has_playback_switch/snd_mixer_selem_has_capture_switch

调用 `snd_mixer_selem_has_playback_switch` (播放) `snd_mixer_selem_has_capture_switch` (录音) 函数判断一个指定元素的配置值是否是 `switch` 类型, 也就是上面说的第二种情况。函数原型如下所示:

```
int snd_mixer_selem_has_playback_switch(snd_mixer_elem_t *elem);
int snd_mixer_selem_has_capture_switch(snd_mixer_elem_t *elem);
```

函数返回 0 表示不是 `switch` 类型; 返回 1 表示是 `switch` 类型。

snd_mixer_selem_has_playback_channel/snd_mixer_selem_has_capture_channel

通过 `snd_mixer_selem_has_playback_channel` (播放) 或 `snd_mixer_selem_has_capture_channel` (录音) 函数可判断指定元素是否包含指定通道, 其函数原型如下所示:

```
int snd_mixer_selem_has_playback_channel(
    snd_mixer_elem_t *elem,
    snd_mixer_selem_channel_id_t channel
);
```

```
int snd_mixer_selem_has_capture_channel(
    snd_mixer_elem_t *elem,
    snd_mixer_selem_channel_id_t channel
);
```

参数 `channel` 用于指定一个通道, `snd_mixer_selem_channel_id_t` 是一个枚举类型, 如下所示:

```
enum snd_mixer_selem_channel_id_t {
    SND_MIXER_SCHN_UNKNOWN = -1,
    SND_MIXER_SCHN_FRONT_LEFT = 0,           //左前
    SND_MIXER_SCHN_FRONT_RIGHT,            //右前
    SND_MIXER_SCHN_REAR_LEFT,              //左后
    SND_MIXER_SCHN_REAR_RIGHT,             //右后
    SND_MIXER_SCHN_FRONT_CENTER,           //前中
    SND_MIXER_SCHN_WOOFER,                 //低音喇叭
    SND_MIXER_SCHN_SIDE_LEFT,              //左侧
    SND_MIXER_SCHN_SIDE_RIGHT,             //右侧
    SND_MIXER_SCHN_REAR_CENTER,            //后中
    SND_MIXER_SCHN_LAST = 31,
    SND_MIXER_SCHN_MONO = SND_MIXER_SCHN_FRONT_LEFT //单声道
};
```

如果元素是双声道元素, 通常只包含左前 (`SND_MIXER_SCHN_FRONT_LEFT`) 和右前 (`SND_MIXER_SCHN_FRONT_RIGHT`) 两个声道。如果是单声道设备, 通常只包含 `SND_MIXER_SCHN_MONO`, 其数值等于 `SND_MIXER_SCHN_FRONT_LEFT`。

可以调用 `snd_mixer_selem_is_playback_mono` (播放) 或 `snd_mixer_selem_is_capture_mono` (录音) 函数判断一个指定的元素是否是单声道元素, 其函数原型如下所示:

```
int snd_mixer_selem_is_playback_mono(snd_mixer_elem_t *elem);
int snd_mixer_selem_is_capture_mono(snd_mixer_elem_t *elem);
```

snd_mixer_selem_get_playback_volume/snd_mixer_selem_get_capture_volume

调用 `snd_mixer_selem_get_playback_volume` (播放) 或 `snd_mixer_selem_get_capture_volume` (录音) 获取指定元素的音量大小, 其函数原型如下所示:

```
int snd_mixer_selem_get_playback_volume(  
    snd_mixer_elem_t *elem,  
    snd_mixer_selem_channel_id_t channel,  
    long *value  
);
```

```
int snd_mixer_selem_get_capture_volume(  
    snd_mixer_elem_t *elem,  
    snd_mixer_selem_channel_id_t channel,  
    long *value  
);
```

参数 `elem` 指定对应的元素，参数 `channel` 指定该元素的某个声道。调用 `snd_mixer_selem_get_playback_volume()` 函数可获取 `elem` 元素的 `channel` 声道对应的音量大小，并将获取到的音量值通过 `value` 返回出来。

函数调用成功返回 0，失败返回一个小于 0 的错误码。

譬如，获取左前声道的音量（播放）：

```
long value;
```

```
snd_mixer_selem_get_playback_volume(elem, SND_MIXER_SCHN_FRONT_LEFT, &value);
```

snd_mixer_selem_set_playback_volume/snd_mixer_selem_set_capture_volume

设置指定元素的音量值，其函数原型如下所示：

```
int snd_mixer_selem_set_playback_volume(  
    snd_mixer_elem_t *elem,  
    snd_mixer_selem_channel_id_t channel,  
    long value  
);
```

```
int snd_mixer_selem_set_capture_volume(  
    snd_mixer_elem_t *elem,  
    snd_mixer_selem_channel_id_t channel,  
    long value  
);
```

调用 `snd_mixer_selem_set_playback_volume`（播放）或 `snd_mixer_selem_set_capture_volume`（录音）设置元素的某个声道的音量，参数 `elem` 指定元素、参数 `channel` 指定该元素的某个声道，参数 `value` 指定音量值。

调用 `snd_mixer_selem_set_playback_volume_all/snd_mixer_selem_set_capture_volume_all` 可一次性设置指定元素所有声道的音量，函数原型如下所示：

```
int snd_mixer_selem_set_playback_volume_all(  
    snd_mixer_elem_t *elem,  
    long value  
);
```

```
int snd_mixer_selem_set_capture_volume_all(  
    snd_mixer_elem_t *elem,  
    long value  
);
```



```

    snd_mixer_elem_t *elem,
    long value
);

```

snd_mixer_selem_get_playback_volume_range/snd_mixer_selem_get_capture_volume_range

获取指定元素的音量范围，其函数原型如下所示：

```

int snd_mixer_selem_get_playback_volume_range(
    snd_mixer_elem_t *elem,
    long *min,
    long *max
);

```

```

int snd_mixer_selem_get_capture_volume_range(
    snd_mixer_elem_t *elem,
    long *min,
    long *max
);

```

28.9.7 示例程序

本小节我们将对示例代码 28.8.1 进行修改，添加音量控制，示例代码如下所示：

本例程源码对应的路径为：[开发板光盘 ->11](#)、[Linux C 应用编程例程源码 ->28_alsa-lib->pcm_playback_mixer.c](#)。

示例代码 28.9.1 PCM 播放示例程序（加入状态控制）

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名：pcm_playback_mixer.c
作者：邓涛
版本：V1.0
描述：一个简单地 PCM 播放示例代码--使用异步方式、加入混音器设置
其他：无
论坛：www.openedv.com
日志：初版 V1.0 2021/7/20 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <termios.h>
#include <signal.h>
#include <alsa/asoundlib.h>

/*****

```

宏定义

```

*****/
#define PCM_PLAYBACK_DEV    "hw:0,0"
#define MIXER_DEV          "hw:0"

/*****
WAV 音频文件解析相关数据结构申明
*****/
typedef struct WAV_RIFF {
    char ChunkID[4];           /* "RIFF" */
    u_int32_t ChunkSize;      /* 从下一个地址开始到文件末尾的总字节数 */
    char Format[4];           /* "WAVE" */
} __attribute__((packed)) RIFF_t;

typedef struct WAV_FMT {
    char Subchunk1ID[4];      /* "fmt " */
    u_int32_t Subchunk1Size;  /* 16 for PCM */
    u_int16_t AudioFormat;    /* PCM = 1*/
    u_int16_t NumChannels;    /* Mono = 1, Stereo = 2, etc. */
    u_int32_t SampleRate;    /* 8000, 44100, etc. */
    u_int32_t ByteRate;      /* = SampleRate * NumChannels * BitsPerSample/8 */
    u_int16_t BlockAlign;    /* = NumChannels * BitsPerSample/8 */
    u_int16_t BitsPerSample; /* 8bits, 16bits, etc. */
} __attribute__((packed)) FMT_t;
static FMT_t wav_fmt;

typedef struct WAV_DATA {
    char Subchunk2ID[4];      /* "data" */
    u_int32_t Subchunk2Size;  /* data size */
} __attribute__((packed)) DATA_t;

/*****
static 静态全局变量定义
*****/
static snd_pcm_t *pcm = NULL;           //pcm 句柄
static snd_mixer_t *mixer = NULL;       //混音器句柄
static snd_mixer_elem_t *playback_vol_elem = NULL; //播放<音量控制>元素
static unsigned int buf_bytes;         //应用程序缓冲区的大小（字节为单位）
static void *buf = NULL;               //指向应用程序缓冲区的指针
static int fd = -1;                    //指向 WAV 音频文件的文件描述符
static snd_pcm_uframes_t period_size = 1024; //周期大小（单位：帧）
static unsigned int periods = 16;      //周期数（设备驱动层 buffer 的大小）
static struct termios old_cfg;         //用于保存终端当前的配置参数

```

```

/*****
static 静态函数
*****/
static void snd_playback_async_callback(snd_async_handler_t *handler)
{
    snd_pcm_t *handle = snd_async_handler_get_pcm(handler); //获取 PCM 句柄
    snd_pcm_sframes_t avail;
    int ret;

    avail = snd_pcm_avail_update(handle); //获取环形缓冲区中有多少帧数据需要填充
    while (avail >= period_size) { //我们一次写入一个周期

        memset(buf, 0x00, buf_bytes); //buf 清零
        ret = read(fd, buf, buf_bytes);
        if (0 >= ret)
            goto out;

        ret = snd_pcm_writei(handle, buf, period_size);
        if (0 > ret) {
            fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
            goto out;
        }
        else if (ret < period_size) { //实际写入的帧数小于指定的帧数
            //此时我们需要调整下音频文件的读位置
            //将读位置向后移动 (往回移) (period_size-ret)*frame_bytes 个字节
            //frame_bytes 表示一帧的字节大小
            if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
                perror("lseek error");
                goto out;
            }
        }
    }

    avail = snd_pcm_avail_update(handle); //再次获取、更新 avail
}

return;
out:
    snd_pcm_drain(pcm); //停止 PCM
    snd_mixer_close(mixer); //关闭混音器
    snd_pcm_close(handle); //关闭 pcm 设备
    tcsetattr(STDIN_FILENO, TCSANOW, &old_cfg); //退出前恢复终端的状态
    free(buf);

```

```
close(fd); //关闭打开的音频文件
exit(EXIT_FAILURE); //退出程序
}

static int snd_pcm_init(void)
{
    snd_pcm_hw_params_t *hwparams = NULL;
    snd_async_handler_t *async_handler = NULL;
    int ret;

    /* 打开 PCM 设备 */
    ret = snd_pcm_open(&pcm, PCM_PLAYBACK_DEV, SND_PCM_STREAM_PLAYBACK, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_open error: %s: %s\n",
                PCM_PLAYBACK_DEV, snd_strerror(ret));
        return -1;
    }

    /* 实例化 hwparams 对象 */
    snd_pcm_hw_params_malloc(&hwparams);

    /* 获取 PCM 设备当前硬件配置,对 hwparams 进行初始化 */
    ret = snd_pcm_hw_params_any(pcm, hwparams);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_any error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /******
    设置参数
    *****/
    /* 设置访问类型: 交错模式 */
    ret = snd_pcm_hw_params_set_access(pcm, hwparams, SND_PCM_ACCESS_RW_INTERLEAVED);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_set_access error: %s\n", snd_strerror(ret));
        goto err2;
    }

    /* 设置数据格式: 有符号 16 位、小端模式 */
    ret = snd_pcm_hw_params_set_format(pcm, hwparams, SND_PCM_FORMAT_S16_LE);
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_hw_params_set_format error: %s\n", snd_strerror(ret));
        goto err2;
    }
}
```

```
}

/* 设置采样率 */
ret = snd_pcm_hw_params_set_rate(pcm, hwparams, wav_fmt.SampleRate, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_rate error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置声道数: 双声道 */
ret = snd_pcm_hw_params_set_channels(pcm, hwparams, wav_fmt.NumChannels);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_channels error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期大小: period_size */
ret = snd_pcm_hw_params_set_period_size(pcm, hwparams, period_size, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_period_size error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 设置周期数 (驱动层环形缓冲区 buffer 的大小): periods */
ret = snd_pcm_hw_params_set_periods(pcm, hwparams, periods, 0);
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params_set_periods error: %s\n", snd_strerror(ret));
    goto err2;
}

/* 使配置生效 */
ret = snd_pcm_hw_params(pcm, hwparams);
snd_pcm_hw_params_free(hwparams); //释放 hwparams 对象占用的内存
if (0 > ret) {
    fprintf(stderr, "snd_pcm_hw_params error: %s\n", snd_strerror(ret));
    goto err1;
}

buf_bytes = period_size * wav_fmt.BlockAlign; //变量赋值, 一个周期的字节大小

/* 注册异步处理函数 */
ret = snd_async_add_pcm_handler(&async_handler, pcm, snd_playback_async_callback, NULL);
if (0 > ret) {
```

```
        fprintf(stderr, "snd_async_add_pcm_handler error: %s\n", snd_strerror(ret));
        goto err1;
    }

    return 0;

err2:
    snd_pcm_hw_params_free(hwparams); //释放内存
err1:
    snd_pcm_close(pcm); //关闭 pcm 设备
    return -1;
}

static int snd_mixer_init(void)
{
    snd_mixer_elem_t *elem = NULL;
    const char *elem_name;
    long minvol, maxvol;
    int ret;

    /* 打开混音器 */
    ret = snd_mixer_open(&mixer, 0);
    if (0 > ret) {
        fprintf(stderr, "snd_mixer_open error: %s\n", snd_strerror(ret));
        return -1;
    }

    /* 关联一个声卡控制设备 */
    ret = snd_mixer_attach(mixer, MIXER_DEV);
    if (0 > ret) {
        fprintf(stderr, "snd_mixer_attach error: %s\n", snd_strerror(ret));
        goto err;
    }

    /* 注册混音器 */
    ret = snd_mixer_selem_register(mixer, NULL, NULL);
    if (0 > ret) {
        fprintf(stderr, "snd_mixer_selem_register error: %s\n", snd_strerror(ret));
        goto err;
    }

    /* 加载混音器 */
    ret = snd_mixer_load(mixer);
```

```

    if (0 > ret) {
        fprintf(stderr, "snd_mixer_load error: %s\n", snd_strerror(ret));
        goto err;
    }

    /* 遍历混音器中的元素 */
    elem = snd_mixer_first_elem(mixer); //找到第一个元素
    while (elem) {

        elem_name = snd_mixer_selem_get_name(elem); //获取元素的名称
        /* 针对开发板出厂系统: WM8960 声卡设备 */
        if (!strcmp("Speaker", elem_name) || //耳机音量<对喇叭外音输出有效>
            !strcmp("Headphone", elem_name) || //喇叭音量<对耳机输出有效>
            !strcmp("Playback", elem_name)) { //播放音量<总的音量控制, 对喇叭和耳机输出都有效>
            if (snd_mixer_selem_has_playback_volume(elem)) { //是否是音量控制元素
                snd_mixer_selem_get_playback_volume_range(elem, &minvol, &maxvol); //获取音量可设置
                //范围
                snd_mixer_selem_set_playback_volume_all(elem, (maxvol-minvol)*0.9 + minvol); //全部设置
                //为90%

                if (!strcmp("Playback", elem_name))
                    playback_vol_elem = elem;
            }
        }

        elem = snd_mixer_elem_next(elem);
    }

    return 0;

err:
    snd_mixer_close(mixer);
    return -1;
}

static int open_wav_file(const char *file)
{
    RIFF_t wav_riff;
    DATA_t wav_data;
    int ret;

    fd = open(file, O_RDONLY);
    if (0 > fd) {

```

```
fprintf(stderr, "open error: %s: %s\n", file, strerror(errno));
return -1;
}

/* 读取 RIFF chunk */
ret = read(fd, &wav_riff, sizeof(RIFF_t));
if (sizeof(RIFF_t) != ret) {
    if (0 > ret)
        perror("read error");
    else
        fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

if (strcmp("RIFF", wav_riff.ChunkID, 4) || //校验
    strcmp("WAVE", wav_riff.Format, 4)) {
    fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

/* 读取 sub-chunk-fmt */
ret = read(fd, &wav_fmt, sizeof(FMT_t));
if (sizeof(FMT_t) != ret) {
    if (0 > ret)
        perror("read error");
    else
        fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

if (strcmp("fmt ", wav_fmt.Subchunk1ID, 4) { //校验
    fprintf(stderr, "check error: %s\n", file);
    close(fd);
    return -1;
}

/* 打印音频文件的信息 */
printf("<<<<音频文件格式信息>>>>\n\n");
printf("  file name:      %s\n", file);
printf("  Subchunk1Size: %u\n", wav_fmt.Subchunk1Size);
```



```

printf("  AudioFormat:   %u\n", wav_fmt.AudioFormat);
printf("  NumChannels:    %u\n", wav_fmt.NumChannels);
printf("  SampleRate:      %u\n", wav_fmt.SampleRate);
printf("  ByteRate:        %u\n", wav_fmt.ByteRate);
printf("  BlockAlign:      %u\n", wav_fmt.BlockAlign);
printf("  BitsPerSample:   %u\n\n", wav_fmt.BitsPerSample);

/* sub-chunk-data */
if (0 > lseek(fd, sizeof(RIFF_t) + 8 + wav_fmt.Subchunk1Size,
             SEEK_SET)) {
    perror("lseek error");
    close(fd);
    return -1;
}

while(sizeof(DATA_t) == read(fd, &wav_data, sizeof(DATA_t))) {

    /* 找到 sub-chunk-data */
    if (!strcmp("data", wav_data.Subchunk2ID, 4))//校验
        return 0;

    if (0 > lseek(fd, wav_data.Subchunk2Size, SEEK_CUR)) {
        perror("lseek error");
        close(fd);
        return -1;
    }
}

fprintf(stderr, "check error: %s\n", file);
return -1;
}

static void show_help(void)
{
    printf("<<<<<<<基于 alsa-lib 音乐播放器>>>>>>>\n\n"
           "操作菜单:\n"
           "  q                退出程序\n"
           "  space<空格>     暂停播放/恢复播放\n"
           "  w                音量增加++\n"
           "  s                音量减小--\n\n");
}

/*****

```

```
main 主函数
*****/
int main(int argc, char *argv[])
{
    snd_pcm_sframes_t avail;
    struct termios new_cfg;
    sigset_t sset;
    int ret;

    if (2 != argc) {
        fprintf(stderr, "Usage: %s <audio_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* 屏蔽 SIGIO 信号 */
    sigemptyset(&sset);
    sigaddset(&sset, SIGIO);
    sigprocmask(SIG_BLOCK, &sset, NULL);

    /* 打开 WAV 音频文件 */
    if (open_wav_file(argv[1]))
        exit(EXIT_FAILURE);

    /* 初始化 PCM Playback 设备 */
    if (snd_pcm_init())
        goto err1;

    /* 初始化混音器 */
    if (snd_mixer_init())
        goto err2;

    /* 申请读缓冲区 */
    buf = malloc(buf_bytes);
    if (NULL == buf) {
        perror("malloc error");
        goto err3;
    }

    /* 终端配置 */
    tcgetattr(STDIN_FILENO, &old_cfg); //获取终端<标准输入-标准输出构成了一套终端>
    memcpy(&new_cfg, &old_cfg, sizeof(struct termios)); //备份
    new_cfg.c_lflag &= ~ICANON; //将终端设置为非规范模式
    new_cfg.c_lflag &= ~ECHO; //禁用回显
```

```
tcsetattr(STDIN_FILENO, TCSANOW, &new_cfg); //使配置生效

/* 播放: 先将环形缓冲区填满数据 */
avail = snd_pcm_avail_update(pcm); //获取环形缓冲区中有多少帧数据需要填充
while (avail >= period_size) { //我们一次写入一个周期

    memset(buf, 0x00, buf_bytes); //buf 清零
    ret = read(fd, buf, buf_bytes);
    if (0 >= ret)
        goto err4;

    ret = snd_pcm_writei(pcm, buf, period_size); //向环形缓冲区中写入数据
    if (0 > ret) {
        fprintf(stderr, "snd_pcm_writei error: %s\n", snd_strerror(ret));
        goto err4;
    }
    else if (ret < period_size) { //实际写入的帧数小于指定的帧数
        //此时我们需要调整下音频文件的读位置
        //将读位置向后移动 (来回移) (period_size-ret)*frame_bytes 个字节
        //frame_bytes 表示一帧的字节大小
        if (0 > lseek(fd, (ret-period_size) * wav_fmt.BlockAlign, SEEK_CUR)) {
            perror("lseek error");
            goto err4;
        }
    }
}

    avail = snd_pcm_avail_update(pcm); //再次获取、更新 avail
}

sigprocmask(SIG_UNBLOCK, &sset, NULL); //取消 SIGIO 信号屏蔽

/* 显示帮助信息 */
show_help();

/* 等待获取用户输入 */
char ch;
long vol;
for (;) {

    ch = getchar(); //获取用户输入的控制字符
    switch (ch) {
        case 'q': //Q 键退出程序
            sigprocmask(SIG_BLOCK, &sset, NULL); //屏蔽 SIGIO 信号
```

```
    goto err4;
    case ' ': //空格暂停/恢复
        switch (snd_pcm_state(pcm)) {

            case SND_PCM_STATE_PAUSED: //如果是暂停状态则恢复运行
                ret = snd_pcm_pause(pcm, 0);
                if (0 > ret)
                    fprintf(stderr, "snd_pcm_pause error: %s\n", snd_strerror(ret));
                break;
            case SND_PCM_STATE_RUNNING: //如果是运行状态则暂停
                ret = snd_pcm_pause(pcm, 1);
                if (0 > ret)
                    fprintf(stderr, "snd_pcm_pause error: %s\n", snd_strerror(ret));
                break;
        }
        break;
    case 'w': //音量增加
        if (playback_vol_elem) {
            //获取音量
            snd_mixer_selem_get_playback_volume(playback_vol_elem,
                SND_MIXER_SCHN_FRONT_LEFT, &vol);

            vol++;
            //设置音量
            snd_mixer_selem_set_playback_volume_all(playback_vol_elem, vol);
        }
        break;
    case 's': //音量降低
        if (playback_vol_elem) {
            //获取音量
            snd_mixer_selem_get_playback_volume(playback_vol_elem,
                SND_MIXER_SCHN_FRONT_LEFT, &vol);

            vol--;
            //设置音量
            snd_mixer_selem_set_playback_volume_all(playback_vol_elem, vol);
        }
        break;
    }
}

err4:
snd_pcm_drop(pcm); //停止 PCM
tcsetattr(STDIN_FILENO, TCSANOW, &old_cfg); //退出前恢复终端的状态
free(buf); //释放内存
```

```
err3:
    snd_mixer_close(mixer); //关闭混音器
err2:
    snd_pcm_close(pcm); //关闭 pcm 设备
err1:
    close(fd);          //关闭打开的音频文件
    exit(EXIT_FAILURE);
}
```

main()函数中调用了自定义函数 `snd_mixer_init()`对声卡混音器进行了初始化, `snd_mixer_init()`函数中做的事情,也就是上面给大家所介绍的流程:首先打开一个空的混音器、attach 关联一个声卡控制设备、注册混音器、加载混音器,整个这一套操作完成之后,就可以去使用混音器了;查找混音器中的元素,对元素进行配置。

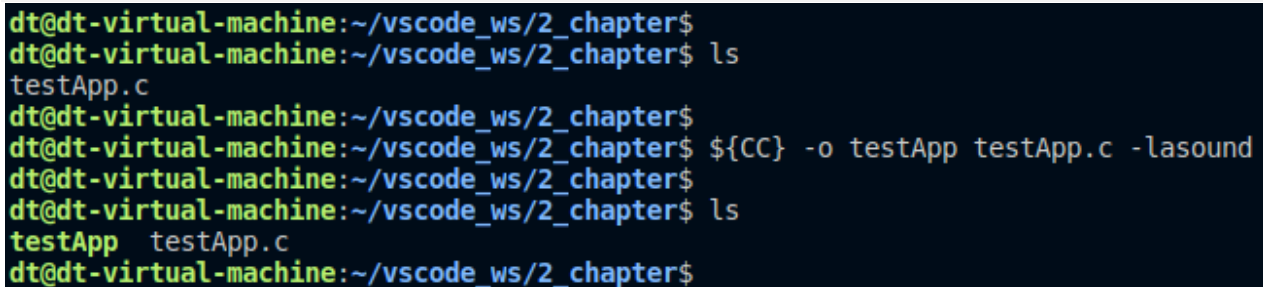
在 `snd_mixer_init()`函数中,我们对 WM8960 声卡的"Speaker"元素(喇叭输出音量)、“Headphone”元素(耳机输出音量)以及"Playback"元素(播放音量)进行了配置,将它们都设置为 90%;之后将"Playback"元素的句柄赋值给全局静态变量 `playback_vol_elem`。

回到 `main()`函数,在 for 循环中,获取用户输入的控制字符,在这里我们添加了 w 和 s,当用户按下 w 键时增加音量、按下 s 键时降低音量,这里控制的音量是 WM8960 的"Playback"音量(播放音量)。

编译应用程序

编译上述示例代码:

```
${CC} -o testApp testApp.c -lasound
```



```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o testApp testApp.c -lasound
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 28.9.2 编译示例代码

测试应用程序

将编译得到的可执行文件拷贝到开发板 Linux 系统的/home/root 目录下,准备一个 WAV 格式的音频文件,执行测试程序:

```
./testApp ./EXO-Overdose.wav
```



```

root@ATK-IMX6U:~#
root@ATK-IMX6U:~# alsaloop --help
Usage: alsaloop [OPTION]...

-h,--help            help
-g,--config          configuration file (one line = one job specified)
-d,--daemonize       daemonize the main process and use syslog for errors
-P,--pdevice        playback device
-C,--cdevice        capture device
-X,--pctl           playback ctl device
-Y,--cctl           capture ctl device
-l,--latency        requested latency in frames
-t,--tlatency       requested latency in usec (1/1000000sec)
-f,--format         sample format
-c,--channels       channels
-r,--rate           rate
-n,--resample       resample in alsa-lib
-A,--samplerate     use converter (0=sincbest,1=sincmedium,2=sincfastest,
                    3=zerohold,4=linear)
-B,--buffer         buffer size in frames
-E,--period         period size in frames
-s,--seconds        duration of loop in seconds
-b,--nblock         non-block mode (very early process wakeup)
-S,--sync           sync mode(0=none,1=simple,2=captshift,3=playshift,4=samplerate,
                    5=auto)
-a,--slave         stream parameters slave mode (0=auto, 1=on, 2=off)
-T,--thread         thread number (-1 = create unique)
-m,--mixer          redirect mixer, argument is:
                    SRC_SLAVE_ID(PLAYBACK)[@DST_SLAVE_ID(CAPTURE)]
-O,--ossmixer      rescan and redirect oss mixer, argument is:
                    ALSA_ID@OSS_ID (for example: "Master@VOLUME")
-e,--effect        apply an effect (bandpass filter sweep)
-v,--verbose        verbose mode (more -v means more verbose)
-w,--workaround    use workaround (serialopen)
-U,--xrun          xrun profiling
-W,--wake          process wake timeout in ms
-z,--syslog        use syslog for errors

Recognized sample formats are: S8 U8 S16_LE S16_BE U16_LE U16_BE S24_LE S24_BE U24_LE U24_BE S32_LE S32_BE
SUBFRAME_BE MU_LAW A_LAW IMA_ADPCM MPEG GSM SPECIAL S24_3LE S24_3BE U24_3LE U24_3BE S20_3LE S20_3BE U20_3
SD_U8 DSD_U16_LE DSD_U32_LE DSD_U16_BE

Tip #1 (usable 500ms latency, good CPU usage, superb xrun prevention):
    alsaloop -t 500000
Tip #2 (superb 1ms latency, but heavy CPU usage):
    alsaloop -t 1000
root@ATK-IMX6U:~#

```

图 28.10.1 alsaloop 工具使用帮助信息

譬如直接运行"alsaloop -t 1000"可以进行测试,大家可以自己亲自测试下。

循环测试原理上很简单,录制音频、然后再播放出来,但是事实上并不如此,还需要考虑到很多的因素,因为对于录音和播放来说,录制一个周期和播放一个周期,硬件上处理这一个周期所花费的时间并不相同,一个是 ADC 过程、而一个是 DAC 过程,所以往往很容易出现 XRUN,所以如何有效合理地设计你的应用程序将变得很重要、以最大限度降低 XRUN 情况的发生。

笔者测试过 alsaloop 工具,虽然也会出现 XRUN,但比较少;如果对此有兴趣的读者,可以参考 alsaloop 程序的源代码,直接下载 alsa-util 源码包,在 alsa-util 源码包中就可以找到 alsaloop 程序的源码,如下所示:

> alsa-utils-1.1.4

名称	修改日期	类型	大小
alsaconf	2017/5/12 16:06	文件夹	
alsactl	2017/5/12 16:06	文件夹	
alsa-info	2017/5/12 16:06	文件夹	
alsaloop	2017/5/12 16:06	文件夹	
alsamixer	2017/5/12 16:06	文件夹	
alsaucm	2017/5/12 16:06	文件夹	
amidi	2017/5/12 16:06	文件夹	
amixer	2017/5/12 16:06	文件夹	
aplay	2017/5/12 16:06	文件夹	
bat	2017/5/12 16:06	文件夹	
iecset	2017/5/12 16:06	文件夹	
include	2017/5/12 16:06	文件夹	
m4	2017/5/12 16:06	文件夹	
po	2017/5/12 16:06	文件夹	

图 28.10.2 alsaloop 源码

除了 alsaloop 的源码之外,还包括前面所介绍的 aplay、alsamixer、amixer、alsactl 等这些工具的源码都在这里,有兴趣的读者可以看看。

28.11 总结

本章我们学习了 Linux 下的音频应用编程,应用程序基于 alsa-lib 库实现播放、录音等功能,本章并没有做过多深入的学习,仅仅只是给大家介绍了 alsa-lib 库函数中一些基本的 API 接口,其中还有绝大部分的接口并没有给大家介绍,如果大家有兴趣,可以自己深入研究、学习!

本小节我们来聊一聊 ALSA 插件。

28.11.1 ALSA 插件 (plugin)

ALSA 提供了一些 PCM 插件,以扩展 PCM 设备的功能和特性,插件负责各种样本转换、通道之间的样本复制等。

调用 `snd_pcm_open()` 函数时,需要填写 PCM 设备名,alsa-lib 库使用逻辑设备名而不是设备节点名。前面编写的示例程序中,我们使用了 "hw:i,j" 这种格式的名字,这其实指定的是一个名为 hw 的插件,而冒号后面的两个数字 i 和 j 表示两个参数,也就是使用该插件时传入的两个参数(第一个参数表示声卡号,第二个参数表示设备号)。

开发板 Linux 系统的 `/usr/share/alsa/` 目录下有一个名为 `alsa.conf` 的文件,如下所示:


```
#
# ALSA library configuration file
#
# pre-load the configuration files

@hooks [
    {
        func load
        files [
            {
                @func concat
                strings [
                    { @func datadir }
                    "/alsa.conf.d/"
                ]
            }
            "/etc/asound.conf"
            "~/.asoundrc"
        ]
        errors false
    }
]

# load card-specific configuration files (on request)

cards.@hooks [
    {
        func load
        files [
```

图 28.11.1 alsa.conf 文件

该文件是 alsa-lib 库的配置文件,调用 `snd_pcm_open()`函数时会加载`/usr/share/alsa/alsa.conf`文件并解析,从上图中可知,`/usr/share/alsa/alsa.conf`文件中会加载并解析`/etc/asound.conf`和`~/.asoundrc`这两个配置文件,在我们的开发板出厂系统中,有`/etc/asound.conf`配置文件、但并没有`~/.asoundrc`文件。

`/usr/share/alsa/alsa.conf`配置文件作为 alsa-lib 库函数的主要入口点,对 alsa-lib 进行了配置并定义了一些基本、通用的 PCM 插件;而`.asoundrc`和`asound.conf`文件的引入提供用户定制化需求,用户可以在这两个文件中根据自己的需求定义插件。

关于插件的定义以及相关的解释说明,大家可以参考以下两份 ALSA 提供的文档:

<https://www.alsa-project.org/main/index.php/Asoundrc>

https://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html

譬如开发板出厂系统`/etc/asound.conf`文件中定义很多的 PCM 插件,如下所示:

```
defaults.pcm.rate_converter "linear"

pcm.dmix_48000{
type dmix
ipc_key 5678293
ipc_key_add_uid yes
slave{
pcm "hw:0,0"
period_time 40000
buffer_time 320000
format S16_LE
rate 48000
}
}

pcm.dmix_44100{
type dmix
ipc_key 5678293
ipc_key_add_uid yes
slave{
pcm "hw:0,0"
period_time 40000
buffer_time 360000
format S16_LE
rate 44100
}
}

pcm.dmix_32000{
type dmix
ipc_key 5678293
ipc_key_add_uid yes
slave{
pcm "hw:0,0"
period_time 40000
buffer_time 360000
format S16_LE
rate 32000
}
}
```

图 28.11.2 /etc/asound.conf 文件中定义的插件

上图中的每一个 `pcm.name { }` 就定义了一个插件, `name` 表示插件的名字, 譬如 `dmix_48000`、`dmix_44100`、`dmix_32000` 等; 而点号前面的 `pcm` 表示 `name` 是一个 PCM 插件, 用于 PCM 设备; 中括号 `{ }` 里边的内容则是对插件的属性定义。

中括号 `{ }` 中, `type` 字段指定了插件的类型, `alsa-lib` 支持多种不同的插件类型, 譬如 `hw`、`plughw`、`mmap_emul`、`shm`、`linear`、`plug`、`multi`、`share`、`dmix`、`dsnoop`、`softvol` 等等, 不同的类型的插件支持不同的功能、特性, 下面给大家简单地进行介绍。

hw 插件

该插件直接与 ALSA 内核驱动程序通信, 这是一种没有任何转换的原始通信。应用程序调用 `alsa-lib` 库函数直接操作了底层音频硬件设置, 譬如对 PCM 设备的配置、直接作用于硬件。

plughw 插件

该插件能够提供诸如采样率转换这样的软件特性, 硬件本身并不支持这样的特性。譬如, 应用程序播放的音频文件是 48000 采样率, 但是底层音频硬件本身并不支持这种采样率, 所以调用 `snd_pcm_hw_params_set_rate()` 函数将 PCM 设备的采样率设置为 48000 时会导致错误!

这时可以使用 `plughw` 插件, 它支持采样率转换这样的软件特性。

dmix 插件

该支持混音, 将多个应用程序的音频数据进行混合。

softvol 插件

支持软件音量。

关于这些插件更加详细地介绍说明, 请查看 ALSA 提供的文档。

第二十九章 网络基础知识

Linux 系统是依靠互联网平台迅速发展起来的, 所以它具有强大的网络功能支持, 也是 Linux 系统的一大特点。互联网对人类社会产生了巨大影响, 它几乎改变了人们生活的方方面面, 可见互联网对人类社会的重要性!

本章我们便来学习一些网络基础知识, 为下一章学习网络编程打下一个基础; 本章会向大家介绍网络基础知识, 譬如网络通信概述、OSI 七层模型、IP 地址、TCP/IP 协议族、TCP 和 UDP 协议等等, 其中并不会深入、详细地介绍这些内容, 旨在以引导大家入门、了解为主; 如果感兴趣的读者可以自行查阅相关书籍进行深入学习。

本章将会讨论如下主题内容。

- 网络通信概述
- 网络相关基础知识: OSI 七层模型与 TCP/IP 四层模型、TCP 与 UDP 协议、IP 地址与端口号的概念。

29.1 网络通信概述

网络通信本质上是一种进程间通信，是位于网络中不同主机上的进程之间的通信，属于 IPC 的一种，通常称为 socket IPC，在第十章中给大家简单地提到过，如图 10.2.1 中所示。所以网络通信是为了解决在网络环境中，不同主机上的应用程序之间的通信问题。

大概可以分为三个层次，如下所示：

- (1)、硬件层：网卡设备，收发网络数据
- (2)、驱动层：网卡驱动（Linux 内核网卡驱动代码）
- (3)、应用层：上层应用程序（调用 socket 接口或更高级别接口实现网络相关应用程序）

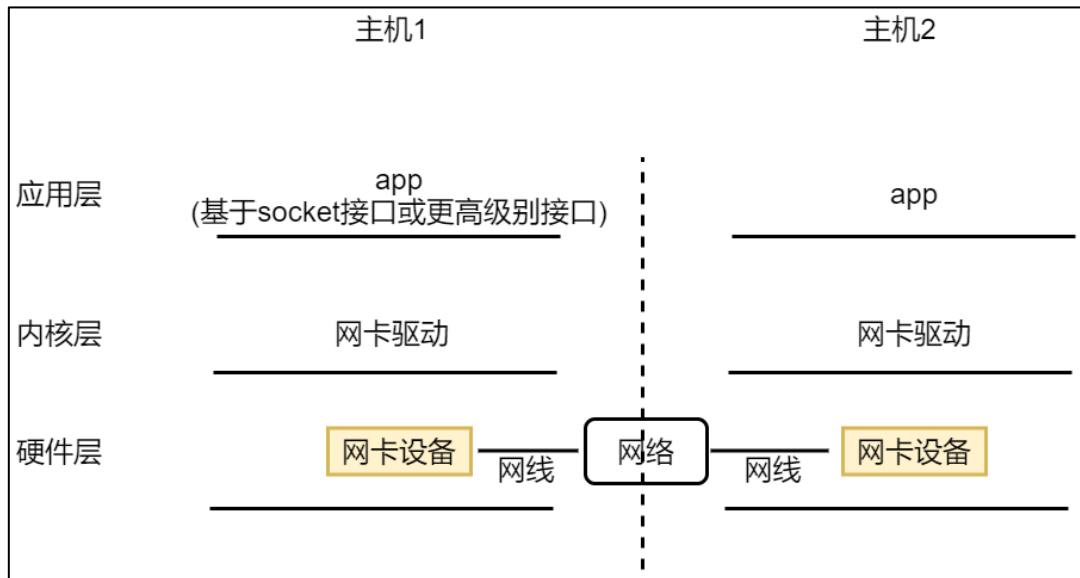


图 29.1.1 网络连接

在硬件上，两台主机都提供了网卡设备，也就满足了进行网络通信最基本的要求，网卡设备是实现网络数据收发的硬件基础。并且通信的两台主机之间需要建立网络连接，这样两台主机之间才可以进行数据传输，譬如通过网线进行数据传输。网络数据的传输媒介有很多种，大体上分为有线传输（譬如双绞线网线、光纤等）和无线传输（譬如 WIFI、蓝牙、ZigBee、4G/5G/GPRS 等），PC 机通常使用有线网络，而手机等移动设备通常使用无线网络。

在内核层，提供了网卡驱动程序，可以驱动底层网卡硬件设备，同时向应用层提供 socket 接口。

在应用层，应用程序基于内核提供的 socket 接口进行应用编程，实现自己的网络应用程序。需要注意的是，socket 接口是内核向应用层提供的一套网络编程接口，所以我们学习网络编程其实就是学习 socket 编程，如何基于 socket 接口编写应用程序。

除了 socket 接口之外，在应用层通常还会使用一些更为高级的编程接口，譬如 http、网络控件等，那么这些接口实际上是对 socket 接口的一种更高级别的封装。在正式学习 socket 编程之前，我们需要先了解一些网络基础知识，为后面的学习打下一个理论基础。

网络通信知识庞大，其中涉及到一大堆的网络协议（TCP/IP 协议族），笔者不可能把这些内容给大家介绍清楚，本章仅仅只是进行简单介绍，以了解为目的。

29.2 网络互连模型：OSI 七层模型

七层模型，亦称 OSI（Open System Interconnection）。OSI 七层参考模型是国际标准化组织（ISO）制定的一个用于计算机或通信系统间网络互联的标准体系，一般称为 OSI 参考模型或七层模型。OSI 七层模型是一个网络互连模型，从上到下依次是：

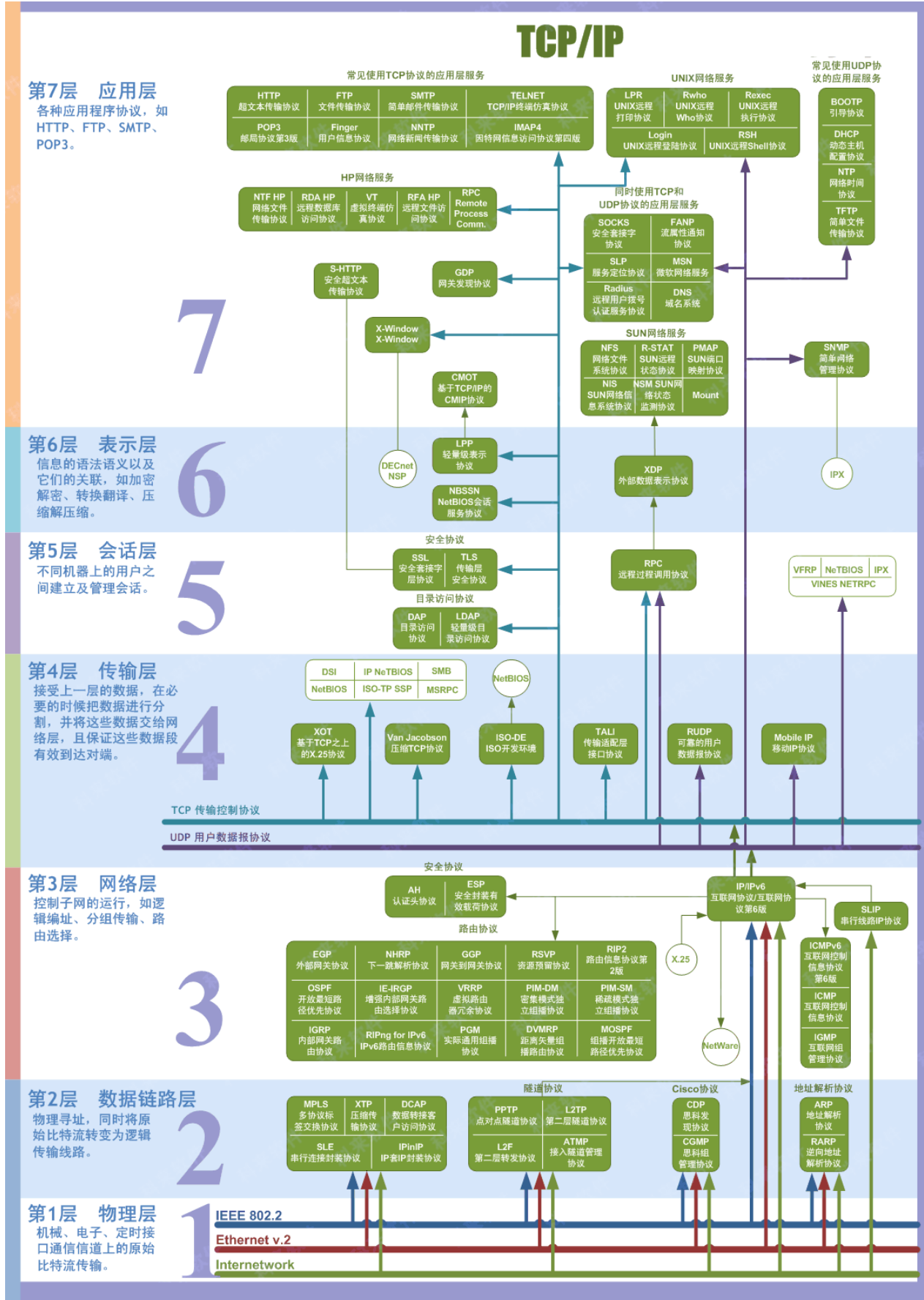


图 29.2.1 OSI 七层模型 (此图取自于网络)

从上可知,网络通信的模型分了很多层,为什么需要分为这么多层次?原因就在于网络是一种非常复杂的通信,需要进行分层,每一层需要去实现不同的功能。下面我们来详细看下 OSI 参考模型中每一层的作用。

应用层

应用层 (Application Layer) 是 OSI 参考模型中的最高层,是最靠近用户的一层,为上层用户提供应用接口,也为用户直接提供各种网络服务。我们常见应用层的网络服务协议有: HTTP、FTP、TFTP、SMTP、SNMP、DNS、TELNET、HTTPS、POP3、DHCP。

表示层

表示层 (Presentation Layer) 提供各种用于应用层数据的编码和转换功能,确保一个系统的应用层发送的数据能被另一个系统的应用层识别。如果必要,该层可提供一种标准表示形式,用于将计算机内部的多种数据格式转换成通信中采用的标准表示形式。数据压缩/解压缩和加密/解密 (提供网络的安全性) 也是表示层可提供的功能之一。

会话层

会话层 (Session Layer) 对应主机进程,指本地主机与远程主机正在进行的会话。会话层就是负责建立、管理和终止表示层实体之间的通信会话。该层的通信由不同设备中的应用程序之间的服务请求和响应组成。将不同实体之间表示层的连接称为会话。因此会话层的任务就是组织和协调两个会话进程之间的通信,并对数据交换进行管理。

传输层

传输层 (Transport Layer) 定义传输数据的协议端口号,以及端到端的流控和差错校验。该层建立了主机端到端的连接,传输层的作用是为上层协议提供端到端的可靠和透明的数据传输服务,包括差错校验处理和流控等问题。我们通常说的, TCP、UDP 协议就工作在这一层,端口号既是这里的“端”。

网络层

进行逻辑地址寻址,实现不同网络之间的路径选择。本层通过 IP 寻址来建立两个节点之间的连接,为源端发送的数据包选择合适的路由和交换节点,正确无误地按照地址传送给目的端的运输层。网络层 (Network Layer) 也就是通常说的 IP 层。该层包含的协议有: IP (Ipv4、Ipv6)、ICMP、IGMP 等。

数据链路层

数据链路层 (Data Link Layer) 是 OSI 参考模型中的第二层,负责建立和管理节点间逻辑连接、进行硬件地址寻址、差错检测等功能。将比特组合成字节进而组合成帧,用 MAC 地址访问介质,错误发现但不能纠正。

数据链路层又分为 2 个子层: 逻辑链路控制子层 (LLC) 和媒体访问控制子层 (MAC)。MAC 子层的主要任务是解决共享型网络中多用户对信道竞争的问题,完成网络介质的访问控制; LLC 子层的主要任务是建立和维护网络连接,执行差错校验、流量控制和链路控制。

数据链路层的具体工作是接收来自物理层的位流形式的数据,并封装成帧,传送到上一层;同样,也将来自上层的数据帧,拆装为位流形式的数据转发到物理层;并且,还负责处理接收端发回的确认帧的信息,以便提供可靠的数据传输。

物理层

物理层 (Physical Layer) 是 OSI 参考模型的最低层,物理层的主要功能是: 利用传输介质为数据链路层提供物理连接,实现比特流的透明传输,物理层的作用是实现相邻计算机节点之间比特流的透明传送,尽可能屏蔽掉具体传输介质和物理设备的差异。使数据链路层不必考虑网络的具体传输介质是什么。“透明传

送比特流”表示经实际电路传送后的比特流没有发生变化，对传送的比特流来说，这个电路好像是看不见的。

实际上，网络数据信号的传输是通过物理层实现的，通过物理介质传输比特流。物理层规定了物理设备标准、电平、传输速率等。常用设备有（各种物理设备）集线器、中继器、调制解调器、网线、双绞线、同轴电缆等，这些都是物理层的传输介质。

以上便是对 OSI 参考模型中的各个层进行的简单介绍，网上也有很多文章对 OSI 参考模型做过详细介绍。除了 OSI 七层模型之外，大家可能还听过 TCP/IP 四层模型、TCP/IP 五层模型，那么这些又是什么呢？接下来将向大家介绍。

29.2.1 TCP/IP 四层/五层模型

事实上，TCP/IP 模型是 OSI 模型的简化版本，我们来看看 OSI 七层模型和 TCP/IP 五层模型之间的对应的关系：

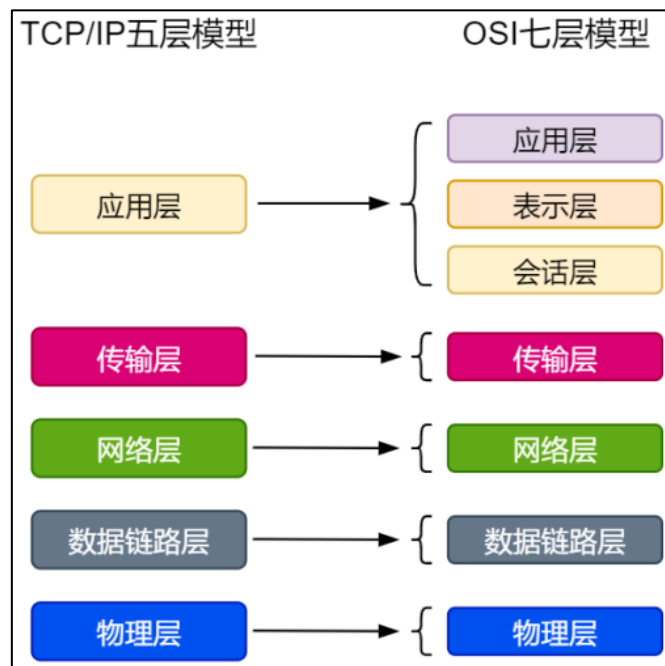


图 29.2.2 OSI 七层模型与 TCP/IP 五层模型

所以由上图可知，TCP/IP 五层模型中，将 OSI 七层模型的最上三层（应用层、表示层和会话层）合并为一个层，即应用层，所以 TCP/IP 五层模型包括：应用层、传输层、网络层、数据链路层以及物理层。除了 TCP/IP 五层模型外，还有 TCP/IP 四层模型，与五层模型唯一不同的就是将数据链路层和物理层合并为网络接口层，如下图所示：

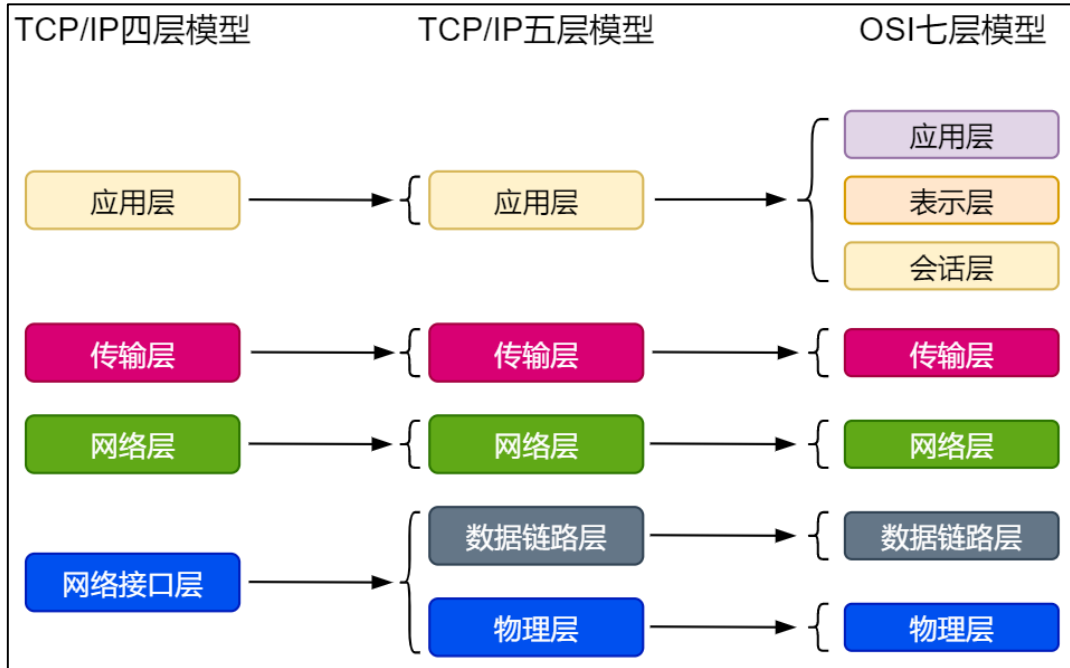


图 29.2.3 OSI 模型与 TCP/IP 模型

由上图可知，四层模型包括：应用层、传输层、网络层以及网络接口层。而在实际的应用中还是使用 TCP/IP 四层模型，五层模型是专门为介绍网络原理而设计的。

29.2.2 数据的封装与拆封

网络通信中，数据从上层到下层交付时，要进行封装；同理，当目标主机接收到数据时，数据由下层传递给上层时需要进行拆封。这就是数据的封装与拆封。

数据的封装过程如下图所示：

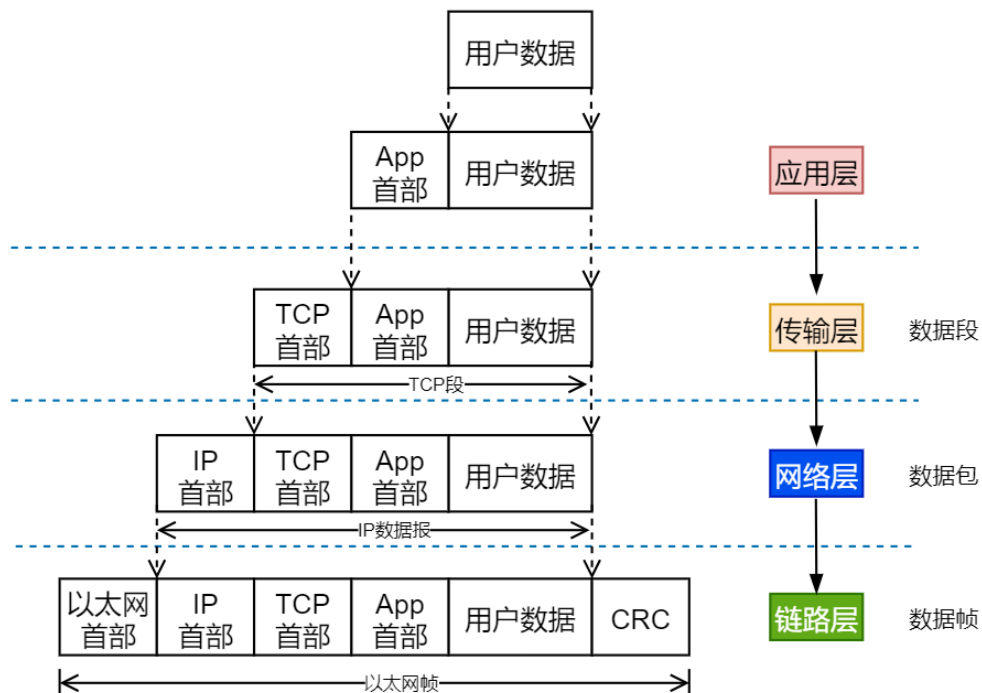


图 29.2.4 数据的封装

当用户发送数据时, 将数据向下交给传输层, 但是在交给传输层之前, 应用层相关协议会对用户数据进行封装, 譬如 MQTT、HTTP 等协议, 其实就是在用户数据前添加一个应用程序头部, 这是处于应用层的操作, 最后应用层通过调用传输层接口来将封装好的数据交给传输层。

传输层会在数据前面加上传输层首部 (此处以 TCP 协议为例, 图中的传输层首部为 TCP 首部, 也可以是 UDP 首部), 然后向下交给网络层。

同样地, 网络层会在数据前面加上网络层首部 (IP 首部), 然后将数据向下交给链路层, 链路层会对数据进行最后一次封装, 即在数据前面加上链路层首部 (此处使用以太网接口为例, 对应以太网首部), 然后将数据交给网卡。

最后, 由网卡硬件设备将数据转换成物理链路上的电平信号, 数据就这样被发送到了网络中。这就是网络数据的发送过程, 从图中可以看到, 各层协议均会对数据进行相应的封装, 可以概括为 TCP/IP 模型中的各层协议对数据进行封装的过程。

以上便是网络数据的封装过程, 当数据被目标主机接收到之后, 会进行相反的拆封过程, 将每一层的首部进行拆解最终得到用户数据。所以, 数据的接收过程与发送过程正好相反, 可以概括为 TCP/IP 模型中的各层协议对数据进行解析的过程。

29.3 IP 地址

Internet 依靠 TCP/IP 协议, 在全球范围内实现不同硬件结构、不同操作系统、不同网络系统的主机之间的互联。在 Internet 上, 每一个节点都依靠唯一的 IP 地址相互区分和相互联系, IP 地址用于标识互联网中的每台主机的身份, 设计人员为每个接入网络中的主机都分配一个 IP 地址 (Internet Protocol Address), 只有合法的 IP 地址才能接入互联网中并且与其他主机进行网络通信, IP 地址是软件地址, 不是硬件地址, 硬件 MAC 地址是存储在网卡中的, 应用于局域网中寻找目标主机。

29.3.1 IP 地址的编址方式

互联网中的每一台主机都需要一个唯一的 IP 地址以标识自己的身份, 那么 IP 地址究竟是什么, 如何去定义一个 IP 呢? 我们需要对 IP 地址的编址方式进行了解。

传统的 IP 地址是一个 32 位二进制数的地址, 也叫 IPv4 地址, 由 4 个 8 位字段组成。除了 IPv4 之外, 还有 IPv6, IPv6 采用 128 位地址长度, 8 个 16 位字段组成, 本小节我们暂时不去理会 IPv6 地址。

在网络通信数据包中, IP 地址以 32 位二进制的形式表示; 而在人机交互中, 通常使用点分十进制方式表示, 譬如 192.168.1.1, 这就是点分十进制的表示方式。

IP 地址中的 32 位实际上包含 2 部分, 分别为网络地址和主机地址, 可通过子网掩码来确定网络地址和主机地址分别占用多少位。

29.3.2 IP 地址的分类

根据 IP 地址中网络地址和主机地址两部分分别占多少位的不同, 将 IP 地址划分为 5 类, 分别为 A、B、C、D、E 五类, 如下所示:

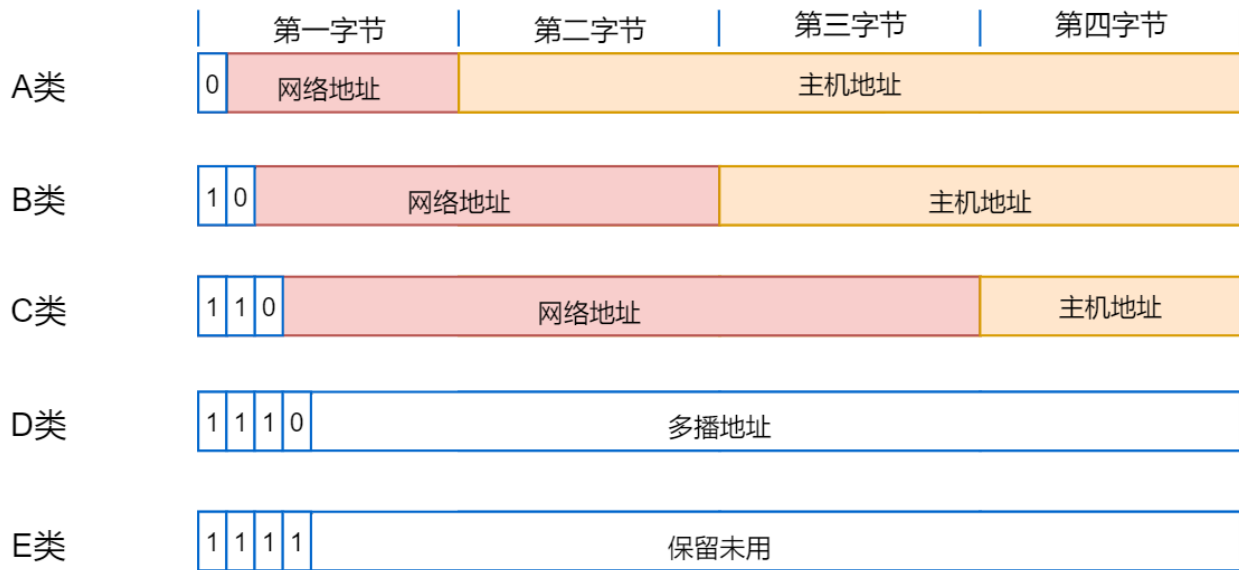


图 29.3.1 5 类 IP 地址的划分

1、A 类 IP 地址

从上图中可以看到，一个 A 类 IP 地址由 1 个字节网络地址和 3 个字节主机地址组成，而网络地址的最高位必须为 0，因此可知，网络地址取值范围为 0~127，一共 128 个网络地址。当然，这 128 个网络地址中，其中 3 个网络地址用作特殊用途，因此可用的网络地址有 125 个。

(1)、A 类地址的第一字节为网络地址，其它 3 个字节为主机地址；

(2)、A 类地址范围为：1.0.0.1 ~ 127.255.255.254；

(3)、A 类地址中设有私有地址和保留地址：

①、10.X.X.X 是私有地址，所谓私有地址就是在互联网中不能使用，而被用在局域网中使用的地址。

②、127.X.X.X 是保留地址，用作循环测试使用。

2、B 类 IP 地址

一个 B 类 IP 地址由 2 个字节的网络地址和 2 个字节的主机地址组成，网络地址的最高位必须是“10”，因此，网络地址第一个字节的取值范围为 128~191，IP 地址范围从 128.0.0.0 到 191.255.255.255。对于 B 类地址来说，一共拥有 16384 个网络地址，其中可用的网络地址有 16382 个，每个网络地址能容纳约 6 万($2^{16}-2=65534$) 多个主机。

(1)、B 类地址中第 1 字节和第 2 字节为网络地址，其它 2 个字节为主机地址。

(2)、B 类地址范围：128.0.0.1 ~ 191.255.255.254。

(3)、B 类地址中设有私有地址和保留地址：

①、172.16.0.0 ~ 172.31.255.255 是私有地址

②、169.254.X.X 是保留地址。如果你的 IP 地址是自动获取 IP 地址，而你在网络上又没有找到可用的 DHCP 服务器。就会得到其中一个 IP。

3、C 类 IP 地址

一个 C 类 IP 地址由 3 字节的网络地址和 1 字节的主机地址组成，网络地址的最高位必须是“110”，因此 C 类 IP 地址的第一个字节的取值范围为 192~223。范围从 192.0.0.0 到 223.255.255.255，网络地址可达 209 万余个，每个网络地址能容纳 254 个主机。

(1)、C 类地址第 1 字节、第 2 字节和第 3 个字节为网络地址，第 4 个字节为主机地址。另外第 1 个字节的高三位固定为 110。

(2)、C 类地址范围为：192.0.0.1 ~ 223.255.255.254。

(3)、C 类地址中的私有地址: 192.168.X.X 是私有地址。

4、D 类 IP 地址

D 类 IP 地址第一个字节以“1110”开始, 它是一个专门保留的地址, 它并不指向特定的网络, 目前这一类地址被用在多点广播(多播, Multicast), 多点广播地址用来一次寻址一组计算机, 它标识共享同一协议的一组计算机。

(1)、D 类地址不分网络地址和主机地址, 它的第 1 个字节的高四位固定为 1110。

(2)、D 类地址范围: 224.0.0.1 ~ 239.255.255.254。

5、E 类 IP 地址

E 类 IP 地址以“11110”开始, 为将来使用保留。全零(“0.0.0.0”)地址对应于当前主机。全“1”的 IP 地址(“255.255.255.255”)是当前子网的广播地址。

(1)、E 类地址也不分网络地址和主机地址, 它的第 1 个字节的前五位固定为 11110。

(2)、E 类地址范围: 240.0.0.1 ~ 255.255.255.254。

总结

以上就给大家介绍了这 5 类 IP 地址, 其中在 A、B、C 三类地址中, 各保留了一个区域作为私有地址:

A 类地址: 10.0.0.0~10.255.255.255

B 类地址: 172.16.0.0~172.31.255.255

C 类地址: 192.168.0.0~192.168.255.255

A 类地址的第一组数字为 1~126。

B 类地址的第一组数字为 128~191。

C 类地址的第一组数字为 192~223。

A 类地址的表示范围为: 0.0.0.0~126.255.255.255, 默认网络掩码为: 255.0.0.0; A 类地址分配给规模特别大的网络使用。A 类地址用第一组数字表示网络地址, 后面三组数字作为连接于网络上的主机对应的地址。分配给具有大量主机而局域网络个数较少的大型网络, 譬如 IBM 公司的网络。

B 类地址的表示范围为: 128.0.0.0~191.255.255.255, 默认网络掩码为: 255.255.0.0; B 类地址分配给一般的中型网络。B 类地址用第一、二组数字表示网络地址, 后面两组数字代表网络上的主机地址。

C 类地址的表示范围为: 192.0.0.0~223.255.255.255, 默认网络掩码为: 255.255.255.0; C 类地址分配给小型网络, 如一般的局域网和校园网, 它可连接的主机数量是最少的, 采用把所属的用户分为若干的网段进行管理。C 类地址用前三组数字表示网络地址, 最后一组数字作为网络上的主机地址。

29.3.3 特殊的 IP 地址

下面给大家介绍一些特殊的 IP 地址, 这些 IP 地址不能分配给任何一个网络的主机使用。

直接广播地址

直接广播(Direct Broadcast Address): 向某个网络上所有的主机发送报文。TCP/IP 规定, 主机号各位全部为“1”的 IP 地址用于广播, 叫作广播地址。譬如一个 IP 地址是 192.168.0.181, 这是一个 C 类地址, 所以它的主机号只有一个字节, 那么对主机号全取 1 得到一个广播地址 192.168.0.255, 向这个地址发送数据就能让同一网络下的所有主机接收到。

A、B、C 三类地址的广播地址结构如下:

- A 类地址的广播地址为: XXX.255.255.255 (XXX 为 A 类地址中网络地址对应的取值范围, 譬如: 120.255.255.255)。
- B 类地址的广播地址为: XXX.XXX.255.255 (XXX 为 B 类地址中网络地址的取值范围, 譬如 139.22.255.255)。

- C 类地址的广播地址为: XXX.XXX.XXX.255 (XXX 为 C 类地址中网络地址的取值范围, 譬如 203.120.16.255)。

受限广播地址

直接广播要求发送方必须广播网络对应的网络号。但有些主机在启动时, 往往并不知道本网络的网络号, 这时候如果想要向本网络广播, 只能采用受限广播地址 (Limited Broadcast Address)。

受限广播地址是在本网络内部进行广播的一种广播地址, TCP/IP 规定, 32 比特全为“1”的 IP 地址用于本网络内的广播, 也就是 255.255.255.255。

多播地址

多播地址用在一对多的通信中, 即一个发送者, 多个接收者, 不论接受者数量的多少, 发送者只发送一次数据包。多播地址属于 D 类地址, D 类地址只能用作目的地址, 而不能作为主机中的源地址。

环回地址

环回地址 (Loopback Address) 是用于网络软件测试以及本机进程之间通信的特殊地址。把 A 类地址中的 127.XXX.XXX.XXX 的所有地址都称为环回地址, 主要用来测试网络协议是否工作正常的作用。比如在电脑中使用 ping 命令去 ping 127.1.1.1 就可以测试本地 TCP/IP 协议是否正常。

不能将环回地址作为任何一台主机的 IP 地址使用。

0.0.0.0 地址

IP 地址 32bit 全为 0 的地址 (也就是 0.0.0.0) 表示本网络上的本主机, 只能用作源地址。

0.0.0.0 是不能被 ping 通的, 在服务器中, 0.0.0.0 并不是一个真实的 IP 地址, 它表示本机中所有的 IPv4 地址。监听 0.0.0.0 的端口, 就是监听本机中所有 IP 的端口。

29.3.4 如何判断 2 个 IP 地址是否在同一个网段内

如何判断两个 IP 地址是否处于同一个子网, 可通过网络标识来进行判断, 网络标识定义如下:

网络标识 = IP 地址 & 子网掩码

2 个 IP 地址的网络标识相同, 那么它们就处于同一网络。譬如 192.168.1.50 和 192.168.1.100, 这 2 个都是 C 类地址, 对应的子网掩码为 255.255.255.0, 很明显, 这两个 IP 地址与子网掩码进行按位与操作时得到的结果 (网络标识) 是一样的, 所以它们处于同一网络。

29.4 TCP/IP 协议

首先给大家说明的是, TCP/IP 协议它其实是一个协议族, 包含了众多的协议, 譬如应用层协议 HTTP、FTP、MQTT...以及传输层协议 TCP、UDP 等这些都属于 TCP/IP 协议, 可参考图 29.2.1 所示。

所以, 我们一般说 TCP/IP 协议, 它不是指某一个具体的网络协议, 而是一个协议族。网络通信当中涉及到的网络协议实在太多了, 对于应用开发来说, 可能使用更多的是应用层协议, 譬如 HTTP、FTP、SMTP 等。

HTTP 协议

HTTP 超文本传输协议 (英文: HyperText Transfer Protocol, 缩写: HTTP) 是一种用于分布式、协作式和超媒体信息系统的的应用层协议。HTTP 是万维网数据通信的基础。HTTP 的应用最为广泛, 譬如大家经常会打开网页浏览器查询资料, 通过浏览器便可开启 HTTP 通信。

HTTP 协议工作于客户端 (用户)、服务器端 (网站) 模式下, 浏览器作为 HTTP 客户端通过 URL 向 HTTP 服务端即 WEB 服务器发送请求。Web 服务器根据接收到的请求后, 向客户端发送响应信息。借助这种浏览器和服务器之间的 HTTP 通信, 我们能够足不出户地获取网络中的各种信息。

FTP 协议

FTP 协议的英文全称为 File Transfer Protocol, 简称为 FTP, 它是一种文件传输协议, 从一个主机向一个主机传输文件的协议。FTP 协议同样也是基于客户端-服务器模式, 在客户端和服务端之间进行文件传输, 譬如我们通常会使用 FTP 协议在两台主机之间进行文件传输, 譬如一台 Ubuntu 系统主机和一台 Windows 系统主机, 将一台主机作为 FTP 服务器、另一台主机作为 FTP 客户端, 建立 FTP 连接之后, 客户端可以从服务器下载文件, 同样也可以将文件上传至服务器。

FTP 除了基本的文件上传/下载功能外, 还有目录操作、权限设置、身份验证等机制, 许多网盘的文件传输功能都是基于 FTP 实现的。

其它的 TCP/IP 协议就不给大家介绍了, 有兴趣的读者可以自行百度了解。

下小节我们重点给大家介绍下工作在传输层的 TCP、UDP 协议, 这两种协议相比各位读者听得比较多。

29.5 TCP 协议

TCP (Transmission Control Protocol, 传输控制协议) 是一种面向连接的、可靠的、基于 IP 的传输协议。由图 29.2.1 可知, TCP 协议工作在传输层, 对上服务 socket 接口, 对下调用 IP 层 (网络层)。

关于 TCP 协议我们需要理解的重点如下:

- ①、TCP 协议工作在传输层, 对上服务 socket 接口, 对下调用 IP 层;
- ②、TCP 是一种面向连接的传输协议, 通信之前必须通过三次握手与客户端建立连接关系后才可通信;
- ③、TCP 协议提供可靠传输, 不怕丢包、乱序。

TCP 协议如何保证可靠传输?

①、TCP 协议采用发送应答机制, 即发送端发送的每个 TCP 报文段都必须得到接收方的应答, 才能认为这个 TCP 报文段传输成功。

②、TCP 协议采用超时重传机制, 发送端在发送出一个 TCP 报文段之后启动定时器, 如果在定时时间内未收到应答, 它将重新发送该报文段。

③、由于 TCP 报文段最终是以 IP 数据报发送的, 而 IP 数据报到达接收端可能乱序、重复、所以 TCP 协议还会将接收到的 TCP 报文段重排、整理、再交付给应用层。

29.5.1 TCP 协议的特性

TCP 协议的特点如下所示:

● 面向连接的

TCP 是一个面向连接的协议, 无论哪一方向另一方发送数据之前, 都必须先在双方之间建立一个 TCP 连接, 否则将无法发送数据, 通过三次握手建立连接, 后面在介绍。

● 确认与重传

当数据从主机 A 发送到主机 B 时, 主机 B 会返回给主机 A 一个确认应答; TCP 通过确认应答 ACK 实现可靠的数据传输。当发送端将数据发送出去之后会等待对端的确认应答。如果有确认应答, 说明数据已经成功到达对端。反之, 数据丢失的可能性比较大。

在一定的时间内如果没有收到确认应答, 发送端就可以认为数据已经丢失, 并进行重发。由此, 即使产生了丢失, 仍然可以保证数据能够到达对端, 实现可靠传输。

● 全双工通信

TCP 连接一旦建立, 就可以在连接上进行双向的通信。任何一个主机都可以向另一个主机发送数据, 数据是双向流通的, 所以 TCP 协议是一个全双工的协议。

- 基于字节流而非报文

将数据按字节大小进行编号,接收端通过 ACK 来确认收到的数据编号,通过这种机制能够保证 TCP 协议的有序性和完整性,因此 TCP 能够提供可靠性传输。

- 流量控制(滑动窗口协议)

TCP 流量控制主要是针对接收端的处理速度不如发送端发送速度快的问题,消除发送方使接收方缓存溢出的可能性。TCP 流量控制主要使用滑动窗口协议,滑动窗口是接受数据端使用的窗口大小,用来告诉发送端接收端的缓存大小,以此可以控制发送端发送数据的大小,从而达到流量控制的目的。这个窗口大小就是我们一次传输几个数据。对所有数据帧按顺序赋予编号,发送方在发送过程中始终保持着一个发送窗口,只有落在发送窗口内的帧才允许被发送;同时接收方也维持着一个接收窗口,只有落在接收窗口内的帧才允许接收。这样通过调整发送方窗口和接收方窗口的大小可以实现流量控制。

- 差错控制

TCP 协议除了确认应答与重传机制外,TCP 协议也会采用校验和的方式来检验数据的有效性,主机在接收数据的时候,会将重复的报文丢弃,将乱序的报文重组,发现某段报文丢失了会请求发送方进行重发,因此在 TCP 往上层协议递交的数据是顺序的、无差错的完整数据。

- 拥塞控制

如果网络上的负载(发送到网络上的分组数)大于网络上的容量(网络同时能处理的分组数),就可能引起拥塞,判断网络拥塞的两个因素:延时和吞吐量。拥塞控制机制是:开环(预防)和闭环(消除)。

流量控制是通过接收方来控制流量的一种方式;而拥塞控制则是通过发送方来控制流量的一种方式。TCP 发送方可能因为 IP 网络的拥塞而被遏制,TCP 拥塞控制就是为了解决这个问题(注意和 TCP 流量控制的区别)。

TCP 拥塞控制的几种方法:慢启动,拥塞避免,快重传和快恢复。

29.5.2 TCP 报文格式

从图 29.2.4 可知,当数据由上层发送到传输层时,数据会被封装为 TCP 数据段,我们将其称为 TCP 报文(或 TCP 报文段),TCP 报文由 TCP 首部+数据区域组成,一般 TCP 首部通常为 20 个字节大小,具体格式如下图所示:

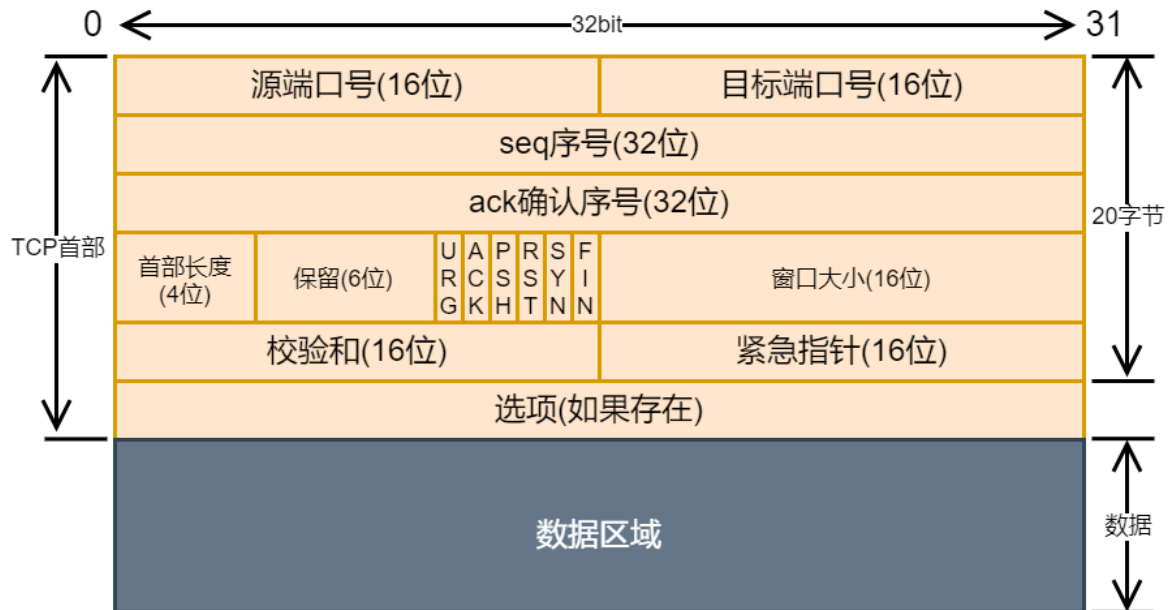


图 29.5.1 TCP 报文格式

下面分别对其中的字段进行介绍:

源端口号和目标端口号

源端口号和目标端口号各占 2 个字节, 一个 4 个字节, 关于端口号的概念会在 29.5.3 小节进行介绍。每个 TCP 报文都包含源主机和目标主机的端口号, 用于寻找发送端和接收端应用进程, 这两个值加上 IP 首部中的源 IP 地址和目标 IP 地址就能确定唯一一个 TCP 连接。有时一个 IP 地址和一个端口号也称为 socket (插口)。

序号

占 4 个字节, 用来标识从 TCP 发送端向 TCP 接收端发送的数据字节流, 它的值表示在这个报文段中的第一个数据字节所处位置码, 根据接收到的数据区域长度, 就能计算出报文最后一个数据所处的序号, 因为 TCP 协议会对发送或者接收的数据进行编号 (按字节的形式), 那么使用序号对每个字节进行计数, 就能很轻易管理这些数据。

在 TCP 传送的数据流中, 每一个字节都有一个序号。例如, 一报文段的序号为 300, 而且数据共 100 字节, 则下一个报文段的序号就是 400; 序号是 32bit 的无符号数, 序号到达 $2^{32}-1$ 后从 0 开始。

确认序号

确认序号占 4 字节, 是期望收到对方下次发送的数据的第一个字节的序号, 也就是期望收到的下一个报文段的首部中的序号; 确认序号应该是上次已成功收到数据字节序号+1。只有 ACK 标志为 1 时, 确认序号才有效。TCP 为应用层提供全双工服务, 这意味数据能在两个方向上独立地进行传输, 因此确认序号通常会与反向数据 (即接收端传输给发送端的数据) 封装在同一个报文中 (即捎带), 所以连接的每一端都必须保持每个方向上的传输数据序号准确性。

首部长度

首部长度字段占 4 个 bit 位, 它指出了 TCP 报文段首部长度, 以字节为单位, 最大能记录 $15 \times 4 = 60$ 字节的首部长度, 因此, TCP 报文段首部最大长度为 60 字节。在字段后接下来有 6bit 空间是保留未用的, 供以后应用, 现在置为 0。

6 个标志位: URG/ACK/PSH/RST/SYN/FIN

保留位之后有 6 个标志位, 分别如下:

- ①、**URG**: 首部中的紧急指针字段标志, 如果是 1 表示紧急指针字段有效。
- ②、**ACK**: 只有当 ACK=1 时, 确认序号字段才有效。
- ③、**PSH**: 当 PSH=1 时, 接收方应该尽快将本报文段立即传送给其应用层。
- ④、**RST**: 当 RST=1 时, 表示出现连接错误, 必须释放连接, 然后再重建传输连接。复位比特还用来拒绝一个不合法的报文段或拒绝打开一个连接。
- ⑤、**SYN**: SYN=1, ACK=0 时表示请求建立一个连接, 携带 SYN 标志的 TCP 报文段为同步报文段。
- ⑥、**FIN**: 为 1 表示发送方没有数据要传输了, 要求释放连接。

窗口大小

占用 2 个字节大小, 表示从确认号开始, 本报文的发送方可以接收的字节数, 即接收窗口大小, 用于流量控制。

校验和

对整个的 TCP 报文段, 包括 TCP 首部和 TCP 数据, 以 16 位字进行计算所得。这是一个强制性的字段。

紧急指针

本报文段中的紧急数据的最后一个字节的序号。

选项

选项字段的大小是不确定的, 最多 40 字节。

29.5.3 建立 TCP 连接: 三次握手

前面我们提到过, TCP 协议是一个面向连接的协议, 双方在进行网络通信之间, 都必须先在双方之间建立一条连接, 俗称“握手”, 可能在学习网络编程之前, 大家或多或少都听过“三次握手”、“四次挥手”这些词语, 那么“三次握手”、“四次挥手”究竟是什么意思, 本小节将详细讨论一个 TCP 连接是如何建立的, 需要经过哪些过程。

“三次握手”其实是指建立 TCP 连接的一个过程, 通信双方建立一个 TCP 连接需要经过“三次握手”这样一个过程。

首先建立连接的过程是由客户端发起, 而服务器会时刻监听、等待着客户端的连接, 其示意图如下所示:

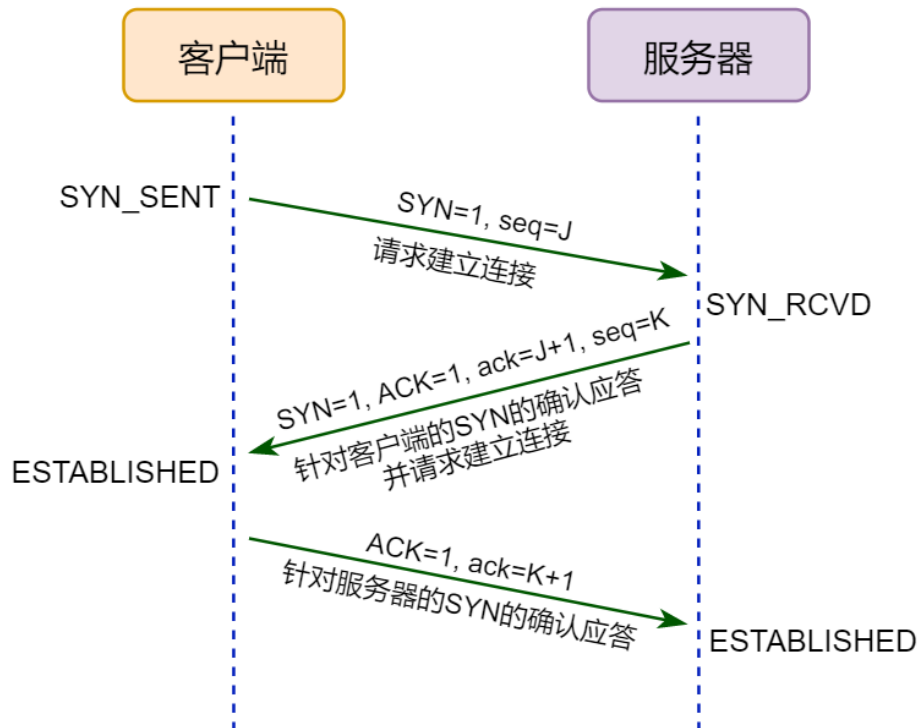


图 29.5.2 三次握手示例图

TCP 连接一般来说会经历以下过程:

● 第一次握手

客户端将 TCP 报文标志位 SYN 置为 1, 随机产生一个序号值 $seq=J$, 保存在 TCP 首部的序列号(Sequence Number)字段里, 指明客户端打算连接的服务器的端口, 并将该数据包发送给服务器端, 发送完毕后, 客户端进入 SYN_SENT 状态, 等待服务器端确认。

● 第二次握手

服务器端收到数据包后由标志位 SYN=1 知道客户端请求建立连接, 服务器端将 TCP 报文标志位 SYN 和 ACK 都置为 1, $ack=J+1$, 随机产生一个序号值 $seq=K$, 并将该数据包发送给客户端以确认连接请求, 服务器端进入 SYN_RCVD 状态。

● 第三次握手

客户端收到确认后, 检查 ack 是否为 $J+1$, ACK 是否为 1, 如果正确则将标志位 ACK 置为 1, $ack=K+1$, 并将该数据包发送给服务器端, 服务器端检查 ack 是否为 $K+1$, ACK 是否为 1, 如果正确则连接建立成功, 客户端和服务器端进入 ESTABLISHED 状态, 完成三次握手, 随后客户端与服务器端之间可以开始传输数据了。

注意:上面写的 ack 和 ACK, 不是同一个概念:

小写的 ack 代表的是头部的确认号 Acknowledge number, ack 。

大写的 ACK, 则是 TCP 首部的标志位, 用于标志的 TCP 包是否对上一个包进行了确认操作, 如果确认了, 则把 ACK 标志位设置成 1。

在完成握手后, 客户端与服务器就成功建立了连接, 同时双方都得到了彼此的窗口大小, 序列号等信息, 在传输 TCP 报文段的时候, 每个 TCP 报文段首部的 SYN 标志都会被置 0, 因为它只用于发起连接, 同步序号。

为什么需要三次握手?

其实 TCP 三次握手过程跟现实生活中的人与人之间的电话交流是很类似的, 譬如 A 打电话给 B:

A: “喂, 你能听到我的声音吗?”

B: “我听得到呀, 你能听到我的声音吗?”

A: “我能听到你, ………”

……

经过三次的互相确认, 大家就会认为对方对听的到自己说话, 才开始接下来的沟通交流, 否则, 如果不进行确认, 那么你在说话的时候, 对方不一定能听到你的声音。所以, TCP 的三次握手是为了保证传输的安全、可靠。

29.5.4 关闭 TCP 连接: 四次挥手

除了“三次握手”, 还有“四次挥手”, “四次挥手”(有一些书也会称为四次握手)其实是指关闭 TCP 连接的一个过程, 当通信双方需要关闭 TCP 连接时需要经过“四次挥手”这样一个过程。

四次挥手即终止 TCP 连接, 就是指断开一个 TCP 连接时, 需要客户端和服务端总共发送 4 个包以确认连接的断开。在 socket 编程中, 这一过程由客户端或服务端任一方执行 close 来触发。

由于 TCP 连接是全双工的, 因此, 每个方向都必须单独进行关闭, 这一原则是当一方完成数据发送任务后, 发送一个 FIN 来终止这一方向的连接, 收到一个 FIN 只是意味着这一方向上没有数据流动了, 即不会再收到数据了, 但是在这个 TCP 连接上仍然能够发送数据, 直到这一方向也发送了 FIN。首先进行关闭的一方将执行主动关闭, 而另一方则执行被动关闭。

四次挥手过程的示意图如下:

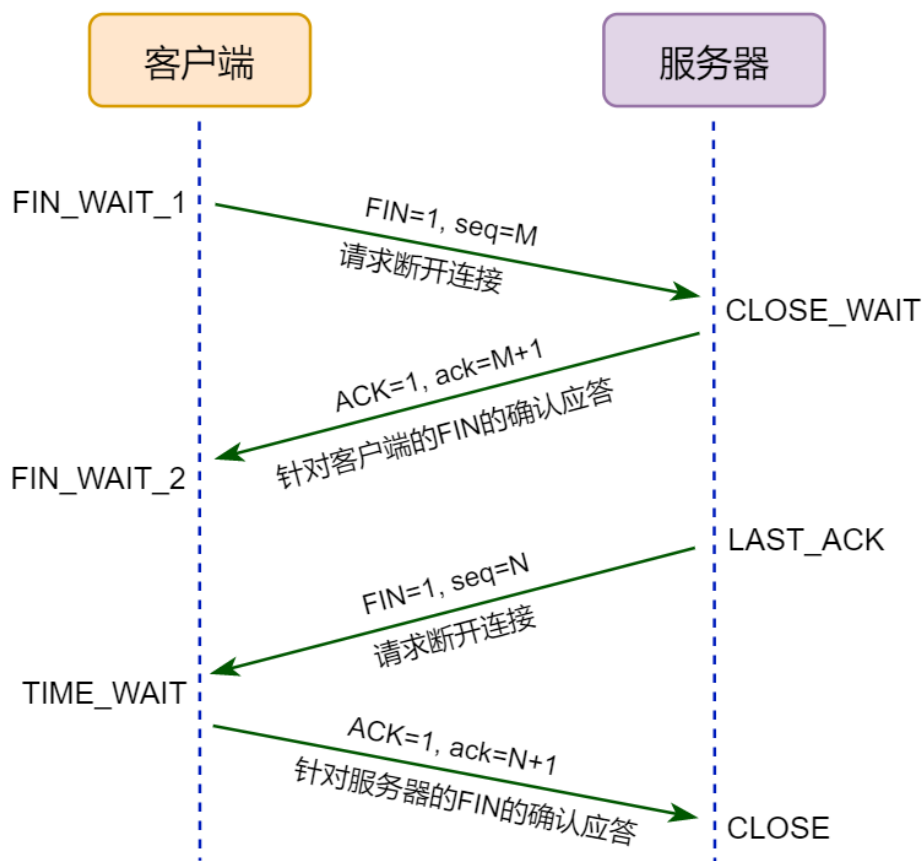


图 29.5.3 四次挥手示意图

挥手请求可以是 Client 端, 也可以是 Server 端发起的, 我们假设是 Client 端发起:

● 第一次挥手

Client 端发起挥手请求, 向 Server 端发出一个 FIN 报文段主动进行关闭连接, 此时报文段的 FIN 标志位被设置为 1。此时, Client 端进入 FIN_WAIT_1 状态, 这表示 Client 端没有数据要发送给 Server 端了。

● 第二次挥手

Server 端收到了 Client 端发送的 FIN 报文段，向 Client 端返回一个 ACK 报文段，此时报文段的 ACK 标志位被设置为 1。ack 设为 seq 加 1，Client 端进入 FIN_WAIT_2 状态，Server 端告诉 Client 端，我确认并同意你的关闭请求。

● 第三次挥手

Server 端向 Client 端发送一个 FIN 报文段请求关闭连接，此时报文段的 FIN 标志位被设置为 1，同时 Client 端进入 LAST_ACK 状态。

● 第四次挥手

Client 端收到 Server 端发送的 FIN 报文段后，向 Server 端发送 ACK 报文段（此时报文段的 ACK 标志位被设置为 1），然后 Client 端进入 TIME_WAIT 状态。Server 端收到 Client 端的 ACK 报文段以后，就关闭连接。此时，Client 端等待 2MSL 的时间后依然没有收到回复，则证明 Server 端已正常关闭，那好，Client 端也可以关闭连接了。

这就是关闭 TCP 连接的四次挥手过程。所以 TCP 协议传输数据的整个过程就如同下图所示：

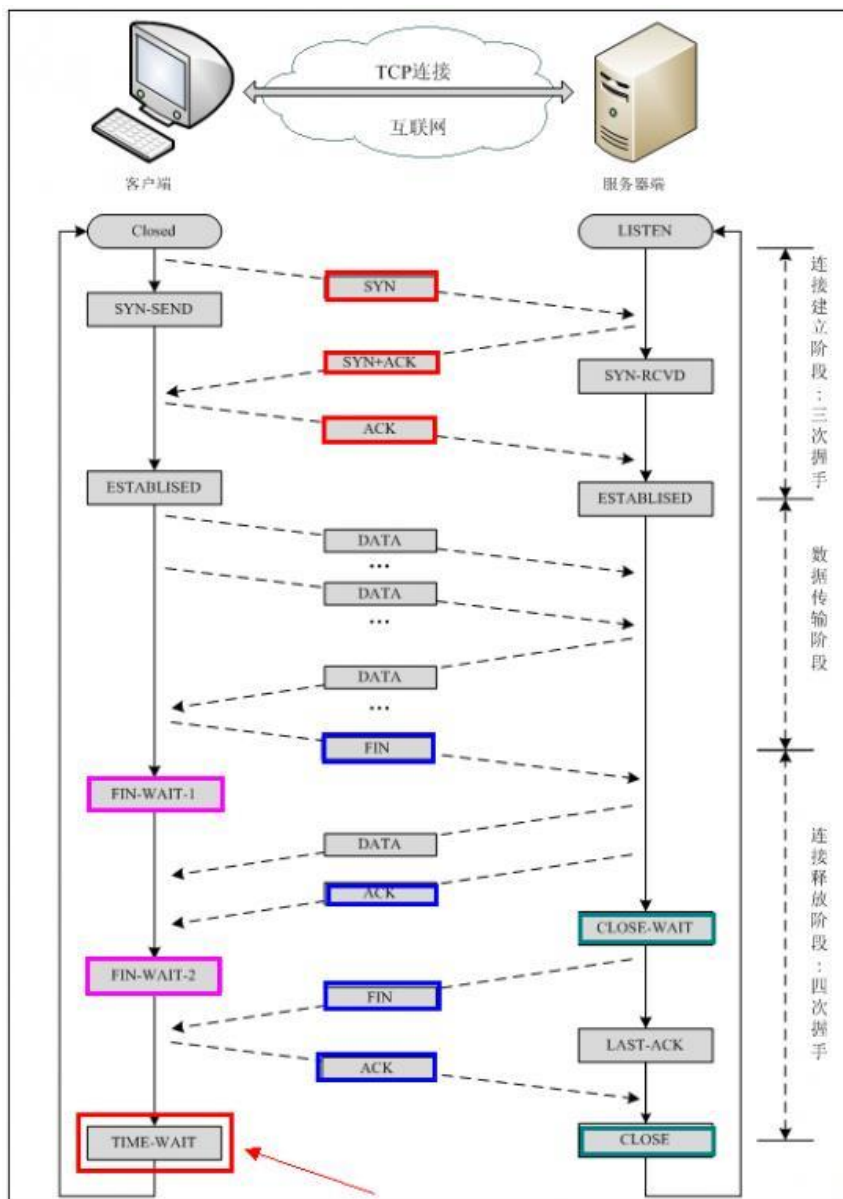


图 29.5.4 TCP 数据传输整个过程（此图取自网络）

在正式进行数据传输之前,需要先建立连接,当成功建立 TCP 连接之后,双方就可以进行数据传输了。当不再需要传输数据时,关闭连接即可!

29.5.5 TCP 状态说明

TCP 协议在建立连接、断开连接以及数据传输过程中都会呈现出现不同的状态,不同的状态采取的动作也是不同的,需要处理各个状态之间的关系。图 29.5.2、图 29.5.3 以及图 29.5.4 中就出现了一些状态标志,除了这些状态标志之外,还有其它一些 TCP 状态,对这些 TCP 状态的说明如下所示:

- **CLOSED 状态:** 表示一个初始状态。
- **LISTENING 状态:** 这是一个非常容易理解的状态,表示服务器端的某个 SOCKET 处于监听状态,监听客户端的连接请求,可以接受连接了。譬如服务器能够提供某种服务,它会监听客户端 TCP 端口的连接请求,处于 LISTENING 状态,端口是开放的,等待被客户端连接。
- **SYN_SENT 状态(客户端状态):** 当客户端调用 connect()函数连接时,它首先会发送 SYN 报文给服务器请求建立连接,因此也随即它会进入到 SYN_SENT 状态,并等待服务器的发送三次握手手中的第 2 个报文。SYN_SENT 状态表示客户端已发送 SYN 报文。
- **SYN_RECV 状态(服务端状态):** 这个状态表示服务器接收到了 SYN 报文,在正常情况下,这个状态是服务器端的 SOCKET 在建立 TCP 连接时的三次握手过程中的一个中间状态,很短暂,基本上用 netstat 你是很难看到这种状态的,除非你特意写了一个客户端测试程序,故意将三次 TCP 握手过程中最后一个 ACK 报文不予发送。因此这种状态时,当收到客户端的 ACK 报文后,它会进入到 ESTABLISHED 状态。
- **ESTABLISHED 状态:** 这个容易理解了,表示连接已经建立了。
- **FIN_WAIT_1 和 FIN_WAIT_2 状态:** 其实 FIN_WAIT_1 和 FIN_WAIT_2 状态的真正含义都是表示等待对方的 FIN 报文。而这两种状态的区别是:FIN_WAIT_1 状态实际上是当 SOCKET 在 ESTABLISHED 状态时,它想主动关闭连接,向对方发送了 FIN 报文,此时该 SOCKET 即进入到 FIN_WAIT_1 状态。而当对方回应 ACK 报文后,则进入到 FIN_WAIT_2 状态,当然在实际的正常情况下,无论对方何种情况下,都应该马上回应 ACK 报文,所以 FIN_WAIT_1 状态一般是比较难见到的,而 FIN_WAIT_2 状态还有时常常常可以用 netstat 看到。
- **TIME_WAIT 状态:** 表示收到了对方的 FIN 报文,并发送出了 ACK 报文,就等 2MSL 后即可回到 CLOSED 可用状态了。如果 FIN_WAIT_1 状态下,收到了对方同时带 FIN 标志和 ACK 标志的报文时,可以直接进入到 TIME_WAIT 状态,而无须经过 FIN_WAIT_2 状态。
- **CLOSE_WAIT 状态:** 这种状态的含义其实是表示在等待关闭。怎么理解呢?当对方 close 一个 SOCKET 后发送 FIN 报文给自己,你系统毫无疑问地会回应一个 ACK 报文给对方,此时则进入到 CLOSE_WAIT 状态。接下来呢,实际上你真正需要考虑的事情是察看你是否还有数据发送给对方,如果没有的话,那么你也就可以 close 这个 SOCKET,发送 FIN 报文给对方,也即关闭连接。所以你在 CLOSE_WAIT 状态下,需要完成的事情是等待你去关闭连接。
- **LAST_ACK 状态:** 它是被动关闭一方在发送 FIN 报文后,最后等待对方的 ACK 报文。当收到 ACK 报文后,也即可以进入到 CLOSED 状态了。

以上便是关于 TCP 状态的一些描述说明,状态之间的转换关系就如上图中所示。

29.5.6 UDP 协议

除了 TCP 协议外,还有 UDP 协议,想必大家都听说过,UDP 是 User Datagram Protocol 的简称,中文名是用户数据报协议,是一种无连接、不可靠的协议,同样它也是工作在传顺层。它只是简单地实现从一端主机到另一端主机的数据传输功能,这些数据通过 IP 层发送,在网络中传输,到达目标主机的顺序是无法预知的,因此需要应用程序对这些数据进行排序处理,这就带来了很大的不方便,此外,UDP 协议更没有

流量控制、拥塞控制等功能,在发送的一端,UDP 只是把上层应用的数据封装到 UDP 报文中,在差错检测方面,仅仅是对数据进行了简单的校验,然后将其封装到 IP 数据报中发送出去。而在接收端,无论是否收到数据,它都不会产生一个应答发送给源主机,并且如果接收到数据发送校验错误,那么接收端就会丢弃该 UDP 报文,也不会告诉源主机,这样子传输的数据是无法保障其准确性的,如果想要其准确性,那么就需

要应用程序来保障了。

UDP 协议的特点:

- ①、无连接、不可靠;
- ②、尽可能提供交付数据服务,出现差错直接丢弃,无反馈;
- ③、面向报文,发送方的 UDP 拿到上层数据直接添加个 UDP 首部,然后进行校验后就递交给 IP 层,而接收的一方在接收到 UDP 报文后简单进行校验,然后直接去除数据递交给上层应用;
- ④、速度快,因为 UDP 协议没有 TCP 协议的握手、确认、窗口、重传、拥塞控制等机制,UDP 是一个无状态的传输协议,所以它在传递数据时非常快,即使在网络拥塞的时候 UDP 也不会降低发送的数据。

UDP 虽然有很多缺点,但也有自己的优点,所以它也有很的应用场合,因为在如今的网络环境下,UDP 协议传输出现错误的概率是很小的,并且它的实时性是非常好,常用于实时视频的传输,比如直播、网络电话等,因为即使是出现了数据丢失的情况,导致视频卡顿,这也不是什么大不了的事情,所以,UDP 协议还是会被应用与对传输速度有要求,并且可以容忍出现差错的数据传输中。

29.6 端口号的概念

前面给大家介绍了 IP 地址,互联网中的每一台主机都需要一个唯一的 IP 地址以标识自己的身份,网络中传输的数据包通过 IP 地址找到对应的目标主机;一台主机通常只有一个 IP 地址,但主机上运行的网络进程却通常不止一个,譬如 Windows 电脑上运行着 QQ、微信、钉钉、网页浏览器等,这些进程都需要进行网络连接,它们都可通过网络发送/接收数据,那么这里就有一个问题?主机接收到网络数据之后,如何确定该数据是哪个进程对应的接收数据呢?其实就是通常端口号来确定的。

端口号本质上就是一个数字编号,用来在一台主机中唯一标识一个能上网(能够进行网络通信)的进程,端口号的取值范围为 0~65535。一台主机通常只有一个 IP 地址,但是可能有多个端口号,每个端口号表示一个能上网的进程。一台拥有 IP 地址的主机可以提供许多服务,比如 Web 服务、FTP 服务、SMTP 服务等,这些服务都是能够进行网络通信的进程,IP 地址只能区分网络中不同的主机,并不能区分主机中的这些进程,显然不能只靠 IP 地址,因此才有了端口号。通过“IP 地址+端口号”来区分主机不同的进程。

很多常见的服务器它都有特定的端口号,具体详情如下表所示:

服务	端口号	说明
HTTP 服务	80	超文本传输协议
FTP 服务	21	文件传输协议,使得主机间可以共享文件
SMTP 服务	25	简单邮件传输协议,它帮助每台计算机在发送或中转信件时找到下一个目的地。
TFTP 服务	69	简单文件传输协议,主机之间进行简单文件传输
SSH 服务	22	安全外壳协议,专为远程登录会话和其他网络服务提供安全性的协议
Telnet 服务	23	终端远程登录协议,它为用户提供了在本地计算机上完成远程主机工作的能力。
POP3 服务	110	邮局协议版本 3,本协议主要用于支持使用客户端远程管理在服务器上的电子邮件

表 29.6.1 常见服务的默认端口号

第三十章 socket 编程基础

Linux 系统是依靠互联网平台迅速发展起来的, 所以它具有强大的网络功能支持, 也是 Linux 系统的一大特点。互联网对人类社会产生了巨大影响, 它几乎改变了人们生活的方方面面, 可见互联网对人类社会的重要性!

本章我们便来学习 Linux 下的网络编程, 我们一般称为 socket 编程, 在上一章中给大家介绍过, socket 是内核向应用层提供的一套网络编程接口, 用户基于 socket 接口可开发自己的网络相关应用程序。

本章作为网络编程基础内容, 其中并不会深入、详细地介绍 socket 编程, 旨在以引导大家入门为主; 其一在于网络编程本就是一门非常难、非常深奥的技能, 市面上有很多关于 Linux/UNIX 网络编程类书籍, 这些书籍专门介绍了网络编程相关知识内容, 而且书本非常厚, 可将其内容之多、难点之多; 其二在于笔者对网络编程了解知之甚少, 掌握的知识、技能太少, 无法向大家传授更加深入的知识、内容, 如果大家以后有机会从事网络编程开发相关工作, 可以购买此类书籍深入学习、研究。

本章将会讨论如下主题内容。

- socket 简介
- socket 编程 API 介绍

30.1 socket 简介

套接字 (socket) 是 Linux 下的一种进程间通信机制 (socket IPC), 在前面的内容中已经给大家提到过, 使用 socket IPC 可以使得在不同主机上的应用程序之间进行通信 (网络通信), 当然也可以是同一台主机上的不同应用程序。socket IPC 通常使用客户端<--->服务器这种模式完成通信, 多个客户端可以同时连接到服务器中, 与服务器之间完成数据交互。

内核向应用层提供了 socket 接口, 对于应用程序开发人员来说, 我们只需要调用 socket 接口开发自己的应用程序即可! socket 是应用层与 TCP/IP 协议通信的中间软件抽象层, 它是一组接口。在设计模式中, socket 其实就是一个门面模式, 它把复杂的 TCP/IP 协议隐藏在 socket 接口后面, 对用户来说, 一组简单的接口就是全部, 让 socket 去组织数据, 以符合指定的协议。所以, 我们无需深入的去理解 tcp/udp 等各种复杂的 TCP/IP 协议, socket 已经为我们封装好了, 我们只需要遵循 socket 的规定去编程, 写出的程序自然遵循 tcp/udp 标准的。

当前网络中的主流程序设计都是使用 socket 进行编程的, 因为它简单易用, 它还是一个标准 (BSD socket), 能在不同平台很方便移植, 比如你的一个应用程序是基于 socket 接口编写的, 那么它可以移植到任何实现 BSD socket 标准的平台, 譬如 LwIP, 它兼容 BSD Socket; 又譬如 Windows, 它也实现了一套基于 socket 的套接字接口, 更甚至在国产操作系统中, 如 RT-Thread, 它也实现了 BSD socket 标准的 socket 接口。

30.2 socket 编程接口介绍

本小节我们向大家介绍, socket 编程中使用到的一些接口函数。使用 socket 接口需要在我们的应用程序代码中包含两个头文件:

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
```

30.2.1 socket()函数

socket()函数原型如下所示:

```
#include <sys/types.h> /* See NOTES */
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

socket()函数类似于 open()函数, 它用于创建一个网络通信端点 (打开一个网络通信), 如果成功则返回一个网络文件描述符, 通常把这个文件描述符称为 socket 描述符 (socket descriptor), 这个 socket 描述符跟文件描述符一样, 后续的操作都有用到它, 把它作为参数, 通过它来进行一些读写操作。

该函数包括 3 个参数, 如下所示:

domain

参数 domain 用于指定一个通信域; 这将选择将用于通信的协议族。可选的协议族如下表所示:

协议族名字	说明	帮助信息
AF_UNIX, AF_LOCAL	Local communication	unix(7)
AF_INET	IPv4 Internet protocols	ip(7)
AF_INET6	IPv6 Internet protocols	ipv6(7)
AF_IPX	IPX - Novell protocols	
AF_NETLINK	Kernel user interface device	netlink(7)

AF_X25	ITU-T X.25/ISO-8208 protocol	x25(7)
AF_AX25	Amateur radio AX.25 protocol	
AF_ATMPVC	Access to raw ATM PVCs	
AF_APPLETALK	AppleTalk	ddp(7)
AF_PACKET	Low level packet interface	packet(7)
AF_ALG	Interface to kernel crypto API	

表 30.2.1 协议族描述

对于 TCP/IP 协议来说, 通常选择 AF_INET 就可以了, 当然如果你的 IP 协议的版本支持 IPv6, 那么可以选择 AF_INET6。

type

参数 type 指定套接字的类型, 当前支持的类型有:

type	说明
SOCK_STREAM	提供有序的、可靠的、双向的、基于连接的字节流, 能保证数据正确传送到对方, 用于 TCP 协议; 可以支持带外数据传输机制。
SOCK_DGRAM	固定长度的、无连接的、不可靠的报文传递, 用于 UDP 协议
SOCK_SEQPACKET	固定长度的、有序的、可靠的、面向连接的报文传递
SOCK_RAW	表示原始套接字, 它允许应用程序访问网络层的原始数据包, 这个套接字用得比较少, 暂时不用理会它。
SOCK_RDM	提供不保证排序的可靠数据报层。
SOCK_PACKET	已过时, 不应在应用程序中使用

表 30.2.2 套接字类型描述

protocol

参数 protocol 通常设置为 0, 表示为给定的通信域和套接字类型选择默认协议。当对同一域和套接字类型支持多个协议时, 可以使用 protocol 参数选择一个特定协议。在 AF_INET 通信域中, 套接字类型为 SOCK_STREAM 的默认协议是传输控制协议 (Transmission Control Protocol, TCP 协议)。在 AF_INET 通信域中, 套接字类型为 SOCK_DGRAM 的默认协议是 UDP。

调用 socket() 与调用 open() 函数很类似, 调用成功情况下, 均会返回用于文件 I/O 的文件描述符, 只不过对于 socket() 来说, 其返回的文件描述符一般称为 socket 描述符。当不再需要该文件描述符时, 可调用 close() 函数来关闭套接字, 释放相应的资源。

如果 socket() 函数调用失败, 则会返回 -1, 并且会设置 errno 变量以指示错误类型。

使用示例

```
int socket_fd = socket(AF_INET, SOCK_STREAM, 0); //打开套接字
if (0 > socket_fd) {
    perror("socket error");
    exit(-1);
}

.....

.....

close(socket_fd); //关闭套接字
```

30.2.2 bind()函数

bind()函数原型如下所示:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

bind()函数用于将一个 IP 地址或端口号与一个套接字进行绑定(将套接字与地址进行关联)。将一个客户端的套接字关联上一个地址没有多少新意,可以让系统选一个默认的地址。一般来讲,会将一个服务器的套接字绑定到一个众所周知的地址---即一个固定的与服务器进行通信的客户端应用程序提前就知道的地址(注意这里说的地址包括 IP 地址和端口号)。因为对于客户端来说,它与服务器进行通信,首先需要知道服务器的 IP 地址以及对应的端口号,所以通常服务器的 IP 地址以及端口号都是众所周知的。

调用 bind()函数将参数 sockfd 指定的套接字与一个地址 addr 进行绑定,成功返回 0,失败情况下返回-1,并设置 errno 以提示错误原因。

参数 addr 是一个指针,指向一个 struct sockaddr 类型变量,如下所示:

示例代码 30.2.1 struct sockaddr 结构体

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
}
```

第二个成员 sa_data 是一个 char 类型数组,一共 14 个字节,在这 14 个字节中就包括了 IP 地址、端口号等信息,这个结构对用户并不友好,它把这些信息都封装在了 sa_data 数组中,这样使得用户是无法对 sa_data 数组进行赋值。事实上,这是一个通用的 socket 地址结构体。

一般我们在使用的时候都会使用 struct sockaddr_in 结构体,sockaddr_in 和 sockaddr 是并列的结构(占用的空间是一样的),指向 sockaddr_in 的结构体的指针也可以指向 sockaddr 的结构体,并代替它,而且 sockaddr_in 结构对用户将更加友好,在使用的时候进行类型转换就可以了。该结构体内容如下所示:

示例代码 30.2.2 struct sockaddr_in 结构体

```
struct sockaddr_in {
    sa_family_t sin_family; /* 协议族 */
    in_port_t sin_port; /* 端口号 */
    struct in_addr sin_addr; /* IP 地址 */
    unsigned char sin_zero[8];
};
```

这个结构体的第一个字段是与 sockaddr 结构体是一致的,而剩下的字段就是 sa_data 数组连续的 14 字节信息里面的内容,只不过从新定义了成员变量而已,sin_port 字段是我们需要填写的端口号信息,sin_addr 字段是我们需要填写的 IP 地址信息,剩下 sin_zero 区域的 8 字节保留未用。

最后一个参数 addrlen 指定了 addr 所指向的结构体对应的字节长度。

使用示例

```
struct sockaddr_in socket_addr;
memset(&socket_addr, 0x0, sizeof(socket_addr)); //清零

//填充变量
socket_addr.sin_family = AF_INET;
socket_addr.sin_addr.s_addr = htonl(INADDR_ANY);
socket_addr.sin_port = htons(5555);
```

```
//将地址与套接字进行关联、绑定
```

```
bind(socket_fd, (struct sockaddr *)&socket_addr, sizeof(socket_addr));
```

注意, 代码中的 `htons` 和 `htonl` 并不是函数, 只是一个宏定义, 主要的作用在于为了避免大小端的问题, 需要这些宏需要在我们的应用程序代码中包含头文件 `<netinet/in.h>`。

Tips: `bind()` 函数并不是总是需要调用的, 只有用户进程想与一个具体的 IP 地址或端口号相关联的时候才需要调用这个函数。如果用户进程没有这个必要, 那么程序可以依赖内核的自动的选址机制来完成自动地址选择, 通常在客户端应用程序中会这样做。

30.2.3 listen() 函数

`listen()` 函数只能在服务器进程中使用, 让服务器进程进入监听状态, 等待客户端的连接请求, `listen()` 函数一般在 `bind()` 函数之后调用, 在 `accept()` 函数之前调用, 它的函数原型是:

```
int listen(int sockfd, int backlog);
```

无法在一个已经连接的套接字 (即已经成功执行 `connect()` 的套接字或由 `accept()` 调用返回的套接字) 上执行 `listen()`。

参数 `backlog` 用来描述 `sockfd` 的等待连接队列能够达到的最大值。在服务器进程正处理客户端连接请求的时候, 可能还存在其它的客户端请求建立连接, 因为 TCP 连接是一个过程, 由于同时尝试连接的用户过多, 使得服务器进程无法快速地完成所有的连接请求, 那怎么办呢? 直接丢掉其他客户端的连接肯定不是一个很好的解决方法。因此内核会在自己的进程空间里维护一个队列, 这些连接请求就会被放入一个队列中, 服务器进程会按照先来后到的顺序去处理这些连接请求, 这样的队列内核不可能让其任意大, 所以必须有一个大小的上限, 这个 `backlog` 参数告诉内核使用这个数值作为队列的上限。而当一个客户端的连接请求到达并且该队列为满时, 客户端可能会收到一个表示连接失败的错误, 本次请求会被丢弃不作处理。

30.2.4 accept() 函数

服务器调用 `listen()` 函数之后, 就会进入到监听状态, 等待客户端的连接请求, 使用 `accept()` 函数获取客户端的连接请求并建立连接。函数原型如下所示:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

为了能够正常让客户端能正常连接到服务器, 服务器必须遵循以下处理流程:

- ①、调用 `socket()` 函数打开套接字;
- ②、调用 `bind()` 函数将套接字与一个端口号以及 IP 地址进行绑定;
- ③、调用 `listen()` 函数让服务器进程进入监听状态, 监听客户端的连接请求;
- ④、调用 `accept()` 函数处理到来的连接请求。

`accept()` 函数通常只用于服务器应用程序中, 如果调用 `accept()` 函数时, 并没有客户端请求连接 (等待连接队列中也没有等待连接的请求), 此时 `accept()` 会进入阻塞状态, 直到有客户端连接请求到达为止。当有客户端连接请求到达时, `accept()` 函数与远程客户端之间建立连接, `accept()` 函数返回一个新的套接字。这个套接字与 `socket()` 函数返回的套接字并不同, `socket()` 函数返回的是服务器的套接字 (以服务器为例), 而 `accept()` 函数返回的套接字连接到调用 `connect()` 的客户端, 服务器通过该套接字与客户端进行数据交互, 譬如向客户端发送数据、或从客户端接收数据。

所以, 理解 `accept()` 函数的关键点在于它会创建一个新的套接字, 其实这个新的套接字就是与执行 `connect()` (客户端调用 `connect()` 向服务器发起连接请求) 的客户端之间建立了连接, 这个套接字代表了服务器与客户端的一个连接。如果 `accept()` 函数执行出错, 将会返回 -1, 并会设置 `errno` 以指示错误原因。

参数 `addr` 是一个传出参数, 参数 `addr` 用来返回已连接的客户端的 IP 地址与端口号等信息。参数 `addrlen` 应设置为 `addr` 所指向的对象的字节长度, 如果我们对客户端的 IP 地址与端口号这些信息不感兴趣, 可以把 `addr` 和 `addrlen` 均置为空指针 `NULL`。

30.2.5 connect()函数

connect()函数原型如下所示:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

该函数用于客户端应用程序中, 客户端调用 connect()函数将套接字 sockfd 与远程服务器进行连接, 参数 addr 指定了待连接的服务器的 IP 地址以及端口号等信息, 参数 addrlen 指定了 addr 指向的 struct sockaddr 对象的字节大小。

客户端通过 connect()函数请求与服务器建立连接, 对于 TCP 连接来说, 调用该函数将发生 TCP 连接的握手过程, 并最终建立一个 TCP 连接, 而对于 UDP 协议来说, 调用这个函数只是在 sockfd 中记录服务器 IP 地址与端口号, 而不发送任何数据。

函数调用成功则返回 0, 失败返回-1, 并设置 errno 以指示错误原因。

30.2.6 发送和接收函数

一旦客户端与服务器建立好连接之后, 我们就可以通过套接字描述符来收发数据了 (对于客户端使用 socket()返回的套接字描述符, 而对于服务器来说, 需要使用 accept()返回的套接字描述符), 这与我们读写普通文件是差不多的操作, 譬如可以调用 read()或 recv()函数读取网络数据, 调用 write()或 send()函数发送数据。

read()函数

read()函数大家都很熟悉了, 通过 read()函数从一个文件描述符中读取指定字节大小的数据并放入到指定的缓冲区中, read()调用成功将返回读取到的字节数, 此返回值受文件剩余字节数限制, 当返回值小于指定的字节数时并不意味着错误; 这可能是因为当前可读取的字节数小于指定的字节数 (比如已经接近文件结尾, 或者正在从管道或者终端读取数据, 或者 read()函数被信号中断等), 出错返回-1 并设置 errno, 如果在调 read 之前已到达文件末尾, 则这次 read 返回 0。

套接字描述符也是文件描述符, 所以使用 read()函数读取网络数据时, read()函数的参数 fd 就是对应的套接字描述符。

recv()函数

recv()函数原型如下所示:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

不论是客户端还是服务器都可以通过 recv()函数读取网络数据, 它与 read()函数的功能是相似的。参数 sockfd 指定套接字描述符, 参数 buf 指向了一个数据接收缓冲区, 参数 len 指定了读取数据的字节大小, 参数 flags 可以指定一些标志用于控制如何接收数据。

函数 recv()与 read()很相似, 但是 recv()可以通过指定 flags 标志来控制如何接收数据, 这些标志如下所示:

标志	描述
MSG_CMSG_CLOEXEC	为 UNIX 域套接字上接收的文件描述符设置执行时关闭标志
MSG_DONTWAIT	启动非阻塞操作 (相当于 O_NONBLOCK)
MSG_ERRQUEUE	接收错误信息作为辅助数据
MSG_OOB	如果协议支持, 获取带外数据
MSG_PEEK	返回数据包内容而不真正取走数据包
MSG_TRUNC	即使数据包被截断, 也返回数据包的长度
MSG_WAITALL	等待知道所有的数据可用 (仅 SOCK_STREAM)

表 30.2.3 recv 函数标志描述

通常一般我们将 `flags` 参数设置为 0, 当然, 你可以根据自己的需求设置该参数。

当指定 `MSG_PEEK` 标志时, 可以查看下一个要读取的数据但不真正取走它, 当再次调用 `read` 或 `recv` 函数时, 会返回刚才查看的数据。

对于 `SOCK_STREAM` 类型套接字, 接收的数据可以比指定的字节大小少。`MSG_WAITALL` 标志会阻止这种行为, 知道所请求的数据全部返回, `recv` 函数才会返回。对于 `SOCK_DGRAM` 和 `SOCK_SEQPACKET` 套接字, `MSG_WAITALL` 标志并不会改变什么行为, 因为这些基于报文的套接字类型一次读取就返回整个报文。

如果发送者已经调用 `shutdown` 来结束传输, 或者网络协议支持按默认的顺序关闭并且发送端已经关闭, 那么当所有的数据接收完毕后, `recv` 会返回 0。

`recv` 在调用成功情况下返回实际读取到的字节数。

write()函数

通过 `write()` 函数可以向套接字描述符中写入数据, 函数调用成功返回写入的字节数, 失败返回 -1, 并设置 `errno` 变量。

send()函数

函数原型如下所示:

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

`send` 和 `write` 很相似, 但是 `send` 可以通过参数 `flags` 指定一些标志, 来改变处理传输数据的方式。这些标志如下所示:

标志	描述
<code>MSG_CONFIRM</code>	提供链路层反馈以保持地址映射有效
<code>MSG_DONTROUTE</code>	勿将数据包路由出本地网络
<code>MSG_DONTWAIT</code>	允许非阻塞操作 (等价于使用 <code>O_NONBLOCK</code>)
<code>MSG_EOR</code>	如果协议支持, 标志记录结束
<code>MSG_MORE</code>	延迟发送数据包允许写更多数据
<code>MSG_NOSIGNAL</code>	在写无连接的套接字时不产生 <code>SIGPIPE</code> 信号
<code>MSG_OOB</code>	如果协议支持, 发送带外数据

表 30.2.4 send 函数标志描述

即使 `send()` 成功返回, 也并不表示连接的另一端的进程就一定接收了数据, 我们所能保证的只是当 `send` 成功返回时, 数据已经被无错误的发送到网络驱动程序上。

30.2.7 close()关闭套接字

当不再需要套接字描述符时, 可调用 `close()` 函数来关闭套接字, 释放相应的资源。

30.3 IP 地址格式转换函数

对于人来说, 我们更容易阅读的是点分十进制的 IP 地址, 譬如 192.168.1.110、192.168.1.50, 这其实是一种字符串的形式, 但是计算机所需要理解的是二进制形式的 IP 地址, 所以我们就需要在点分十进制字符串和二进制地址之间进行转换。

点分十进制字符串和二进制地址之间的转换函数主要有: `inet_aton`、`inet_addr`、`inet_ntoa`、`inet_ntop`、`inet_pton` 这五个, 在我们的应用程序中使用它们需要包含头文件 `<sys/socket.h>`、`<arpa/inet.h>` 以及 `<netinet/in.h>`。

30.3.1 inet_aton、inet_addr、inet_ntoa 函数

这些函数可将一个 IP 地址在点分十进制表示形式和二进制表示形式之间进行转换, 这些函数已经废弃了, 基本不用这些函数了, 但是在一些旧的代码中可能还会看到这些函数。完成此类转换工作我们应该使用下面介绍的这些函数。

30.3.2 inet_ntop、inet_pton 函数

inet_ntop()、inet_pton()与 inet_ntoa()、inet_aton()类似, 但它们还支持 IPv6 地址。它们将二进制 IPv4 或 IPv6 地址转换成以点分十进制表示的字符串形式, 或将点分十进制表示的字符串形式转换成二进制 IPv4 或 IPv6 地址。使用这两个函数只需包含<arpa/inet.h>头文件即可!

inet_pton()函数

inet_pton()函数原型如下所示:

```
int inet_pton(int af, const char *src, void *dst);
```

inet_pton()函数将点分十进制表示的字符串形式转换成二进制 IPv4 或 IPv6 地址。

将字符串 src 转换为二进制地址, 参数 af 必须是 AF_INET 或 AF_INET6, AF_INET 表示待转换的 IPv4 地址, AF_INET6 表示待转换的是 IPv6 地址; 并将转换后得到的地址存放在参数 dst 所指向的对象中, 如果参数 af 被指定为 AF_INET, 则参数 dst 所指对象应该是一个 struct in_addr 结构体的对象; 如果参数 af 被指定为 AF_INET6, 则参数 dst 所指对象应该是一个 struct in6_addr 结构体的对象。

inet_pton()转换成功返回 1 (已成功转换)。如果 src 不包含表示指定地址族中有效网络地址的字符串, 则返回 0。如果 af 不包含有效的地址族, 则返回-1 并将 errno 设置为 EAFNOSUPPORT。

使用示例

示例代码 30.3.1 inet_pton()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define IPV4_ADDR "192.168.1.222"

int main(void)
{
    struct in_addr addr;

    inet_pton(AF_INET, IPV4_ADDR, &addr);
    printf("ip addr: 0x%x\n", addr.s_addr);
    exit(0);
}
```

测试结果:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
a.out testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./a.out
ip addr: 0xde01a8c0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 30.3.1 测试结果

inet_ntop()函数

inet_ntop()函数执行与 inet_pton()相反的操作, 函数原型如下所示:

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

参数 af 与 inet_pton()函数的 af 参数意义相同。

参数 src 应指向一个 struct in_addr 结构体对象或 struct in6_addr 结构体对象, 依据参数 af 而定。函数 inet_ntop()会将参数 src 指向的二进制 IP 地址转换为点分十进制形式的字符串, 并将字符串存放在参数 dst 所指的缓冲区中, 参数 size 指定了该缓冲区的大小。

inet_ntop()在成功时会返回 dst 指针。如果 size 的值太小了, 那么将会返回 NULL 并将 errno 设置为 ENOSPC。

使用示例

示例代码 30.3.2 inet_ntop()函数使用示例

```

#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>

int main(void)
{
    struct in_addr addr;
    char buf[20] = {0};

    addr.s_addr = 0xde01a8c0;
    inet_ntop(AF_INET, &addr, buf, sizeof(buf));
    printf("ip addr: %s\n", buf);
    exit(0);
}

```

测试结果:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./a.out
ip addr: 192.168.1.222
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 30.3.2 测试结果

30.4 socket 编程实战

经过上面的介绍,本小节我们将进行编程实战,实现一个简单地服务器和一个简单地客户端应用程序。

30.4.1 编写服务器程序

编写服务器应用程序的流程如下:

- ①、调用 `socket()` 函数打开套接字,得到套接字描述符;
- ②、调用 `bind()` 函数将套接字与 IP 地址、端口号进行绑定;
- ③、调用 `listen()` 函数让服务器进程进入监听状态;
- ④、调用 `accept()` 函数获取客户端的连接请求并建立连接;
- ⑤、调用 `read/recv`、`write/send` 与客户端进行通信;
- ⑥、调用 `close()` 关闭套接字。

下面,我们就根据上面列举的步骤来编写一个简单地服务器应用程序,代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->30_socket->socket_server.c](#)。

示例代码 30.4.1 简单地服务器应用程序示例代码

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : socket_server.c
作者 : 邓涛
版本 : V1.0
描述 : 一个简单地 TCP 服务器应用程序示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/20 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

```



```
#define SERVER_PORT    8888    //端口号不能发生冲突,不常用的端口号通常大于 5000
```

```
int main(void)
{
    struct sockaddr_in server_addr = {0};
    struct sockaddr_in client_addr = {0};
    char ip_str[20] = {0};
    int sockfd, connfd;
    int addrlen = sizeof(client_addr);
    char recvbuf[512];
    int ret;

    /* 打开套接字, 得到套接字描述符 */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (0 > sockfd) {
        perror("socket error");
        exit(EXIT_FAILURE);
    }

    /* 将套接字与指定端口号进行绑定 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(SERVER_PORT);

    ret = bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    if (0 > ret) {
        perror("bind error");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    /* 使服务器进入监听状态 */
    ret = listen(sockfd, 50);
    if (0 > ret) {
        perror("listen error");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    /* 阻塞等待客户端连接 */
    connfd = accept(sockfd, (struct sockaddr *)&client_addr, &addrlen);
    if (0 > connfd) {
```

```
    perror("accept error");
    close(sockfd);
    exit(EXIT_FAILURE);
}

printf("有客户端接入...\n");
inet_ntop(AF_INET, &client_addr.sin_addr.s_addr, ip_str, sizeof(ip_str));
printf("客户端主机的 IP 地址: %s\n", ip_str);
printf("客户端进程的端口号: %d\n", client_addr.sin_port);

/* 接收客户端发送过来的数据 */
for (;;) {

    // 接收缓冲区清零
    memset(recvbuf, 0x0, sizeof(recvbuf));

    // 读数据
    ret = recv(connfd, recvbuf, sizeof(recvbuf), 0);
    if(0 >= ret) {
        perror("recv error");
        close(connfd);
        break;
    }

    // 将读取到的数据以字符串形式打印出来
    printf("from client: %s\n", recvbuf);

    // 如果读取到"exit"则关闭套接字退出程序
    if (0 == strncmp("exit", recvbuf, 4)) {
        printf("server exit...\n");
        close(connfd);
        break;
    }
}

/* 关闭套接字 */
close(sockfd);
exit(EXIT_SUCCESS);
}
```

以上我们实现了一个非常简单地服务器应用程序, 根据上面列举的步骤完成了这个示例代码, 最终的功能是, 当客户端连接到服务器之后, 客户端会向服务器 (也就是本程序) 发送数据, 在我们服务器应用程序中会读取客户端发送的数据并将其打印出来, 就是这么简单的一个功能。

SERVER_PORT 宏指定了本服务器绑定的端口号, 这里我们将端口号设置为 8888, 端口不能与其它服务器的端口号发生冲突, 不常用的端口号通常大于 5000。

代码就不再解释了, 都非常简单!

30.4.2 编写客户端程序

接下来我们再编写一个简单地客户端应用程序, 客户端的功能是连接上小节所实现的服务器, 连接成功之后向服务器发送数据, 发送的数据由用户输入。示例代码如下所示:

本例程源码对应的路径为: [开发板光盘->11、Linux C 应用编程例程源码->30_socket->socket_client.c](#)。

示例代码 30.4.2 简单地客户端应用程序示例代码

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : socket_client.c
作者 : 邓涛
版本 : V1.0
描述 : 一个简单地 TCP 客户端应用程序示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/20 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define SERVER_PORT    8888           //服务器的端口号
#define SERVER_IP      "192.168.1.150" //服务器的 IP 地址

int main(void)
{
    struct sockaddr_in server_addr = {0};
    char buf[512];
    int sockfd;
    int ret;

    /* 打开套接字, 得到套接字描述符 */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (0 > sockfd) {
        perror("socket error");
    }
}

```

```
        exit(EXIT_FAILURE);
    }

    /* 调用 connect 连接远端服务器 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT); //端口号
    inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr); //IP 地址

    ret = connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    if (0 > ret) {
        perror("connect error");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("服务器连接成功...\n\n");

    /* 向服务器发送数据 */
    for (;) {
        // 清理缓冲区
        memset(buf, 0x0, sizeof(buf));

        // 接收用户输入的字符串数据
        printf("Please enter a string: ");
        fgets(buf, sizeof(buf), stdin);

        // 将用户输入的数据发送给服务器
        ret = send(sockfd, buf, strlen(buf), 0);
        if (0 > ret) {
            perror("send error");
            break;
        }

        //输入了"exit", 退出循环
        if (0 == strncmp(buf, "exit", 4))
            break;
    }

    close(sockfd);
    exit(EXIT_SUCCESS);
}
```

代码不再说明! 需要注意的是 SERVER_IP 和 SERVER_PORT 指的是服务器的 IP 地址和端口号, 服务器的 IP 地址根据实际情况进行设置, 服务器应用程序示例代码 30.4.1 中我们绑定的端口号为 8888, 所以在客户端应用程序中我们也需要指定 SERVER_PORT 为 8888。

30.4.3 编译测试

这里笔者将服务器程序运行在开发板上, 而将客户端应用程序运行在 Ubuntu 系统, 当然你也可以将客户端和服务器程序都运行在开发板或 Ubuntu 系统, 这都是没问题的。

首先编译服务器应用程序和客户端应用程序:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
socket_client.c socket_server.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o client socket_client.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o server socket_server.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls  
client server socket_client.c socket_server.c  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 30.4.1 编译客户端应用程序和服务器应用程序

编译得到 client 和 server 可执行文件, server 可执行文件在开发板上运行, client 可执行文件在 PC 端 Ubuntu 系统下运行。将 server 可执行文件拷贝到开发板/home/root 目录下, 如下所示:

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ls  
driver server shell  
root@ATK-IMX6U:~#
```

图 30.4.2 将服务器执行文件拷贝到开发板

在开发板执行 server:

```
root@ATK-IMX6U:~#  
root@ATK-IMX6U:~# ./server  
█
```

图 30.4.3 先执行服务器应用程序

接着在 Ubuntu 系统执行客户端程序:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$  
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./client  
服务器连接成功...  
  
Please enter a string: █
```

图 30.4.4 执行客户端应用程序

客户端运行之后将会去连接远端服务器, 连接成功便会打印出信息“服务器连接成功...”, 此时服务器也会监测到客户端连接, 会打印相应的信息, 如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./server
有客户端接入...
客户端主机的IP地址: 192.168.1.187
客户端进程的端口号: 4794
```

图 30.4.5 服务器监测到客户端连接

接下来我们便可以在客户端处输入字符串, 客户端程序会将我们输入的字符串信息发送给服务器, 服务器接收到之后将其打印出来, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./client
服务器连接成功...

Please enter a string: aaaaaaaaaaaaaaaaaa
Please enter a string: bbbbbbbbbbbbbbb
Please enter a string: dadasdada
Please enter a string: Hello World!!!!
Please enter a string: daddasdadasd
Please enter a string: exit
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 30.4.6 输入字符串信息

```
root@ATK-IMX6U:~# ./server
有客户端接入...
客户端主机的IP地址: 192.168.1.187
客户端进程的端口号: 4794
from client: aaaaaaaaaaaaaaaaaa

from client: bbbbbbbbbbbbbbb

from client: dadasdada

from client: Hello World!!!!

from client: daddasdadasd

from client: exit

server exit...
root@ATK-IMX6U:~#
```

图 30.4.7 服务器接收到客户端发送的信息

总结

到此, 本章的内容就结束了, 内容讲得非常浅, 目的其实并不是让大家学会网络编程, 这个是不可能的, 旨在以引导大家入门为主, 让大家对 socket 网络编程有一个基本的了解和认识。因为网络编程本就是应用程序中一门比较专业的方向, 如果大家将来想从事这方面的工作、或者以后从事了这方面工作, 再去找资料好好学习, 如果没有这个打算, 那就不要去深入研究这个, 有了基本的了解、认识就行了。

第三十一章 CAN 应用编程基础

本章我们学习 CAN 应用编程, CAN 是目前应用非常广泛的现场总线之一, 主要应用于汽车电子和工业领域, 尤其是汽车领域, 汽车上大量的传感器与模块都是通过 CAN 总线连接起来的。CAN 总线目前是自动化领域发展的热点技术之一, 由于其高可靠性, CAN 总线目前广泛的应用于工业自动化、船舶、汽车、医疗和工业设备等方面。

在我们的 I.MX6U ALPHA/Mini 开发板上提供了一个 CAN 接口, 使用该接口可以进行 CAN 总线通信, 本章我们就来学习一下如何编写一个简单地应用程序来测试开发板上的 CAN 接口。需要说明的是, 本章自然不会深入给大家讲解 CAN 应用编程, 同样, CAN 总线技术也是一个非常专业的技术方向, 笔者并不从事这方面的工作, 对 CAN 的使用、了解少之又少, 所以本章同样旨在以引导大家入门为主!

本章将会讨论如下主题内容。

- CAN 总线介绍
- CAN 应用编程介绍

31.1 CAN 基础知识

在学习 CAN 应用编程之前, 本小节简单地给大家介绍下 CAN 相关的基础知识。

本小节内容参考了由瑞萨电子编写的《CAN 入门教程》, 该手册在开发板资料包已经给大家提供了, 路径为: 4、参考资料->CAN 入门教程.pdf。

31.1.1 什么是 CAN?

CAN 是 Controller Area Network 的缩写(以下称为 CAN), 即控制器局域网, 是 ISO 国际标准化的串行通信协议。

CAN 总线最初是由德国电气商博世公司开发, 其最初动机就是为了解决现代汽车中庞大的电子控制系统之间的通讯, 减少不断增加的信号线。于是, 他们设计了一个单一的网络总线, 所有的外围器件可以被挂接在该总线上。

在当前的汽车工业中, 出于对安全性、舒适性、方便性、低公害、低成本的要求, 各种各样的电子控制系统被开发了出来, 车上的电子控制系统越来越多, 譬如发动机管理控制、变速箱控制、汽车仪表、空调、车门控制、灯光控制、气囊控制、转向控制、胎压监测、制动系统、雷达、自适应巡航、电子防盗系统等。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同, 由多条总线构成的情况很多, 线束的数量也随之增加。为适应“减少线束的数量”、“通过多个 LAN, 进行大量数据的高速通信”的需要, 1986 年德国电气商博世公司开发出面向汽车的 CAN 通信协议。此后, CAN 通过 ISO11898 及 ISO11519 进行了标准化, 现在在欧洲已是汽车网络的标准协议, 目前已是当前应用最广泛的现场总线之一。

CAN 是一种多主方式的串行通讯总线, 基本设计规范要求有高的位速率, 高抗电磁干扰性, 而且能够检测出产生的任何错误。经过几十年的发展, 现在, CAN 的高性能、高可靠性以及高实时性已被认同, 并被广泛地应用于工业自动化、船舶、医疗设备、工业设备等方面。

以汽车电子为例, 汽车上有空调、车门、发动机、大量传感器等, 这些部件、模块都是通过 CAN 总线连在一起形成一个网络, 车载网络构想图如下所示:

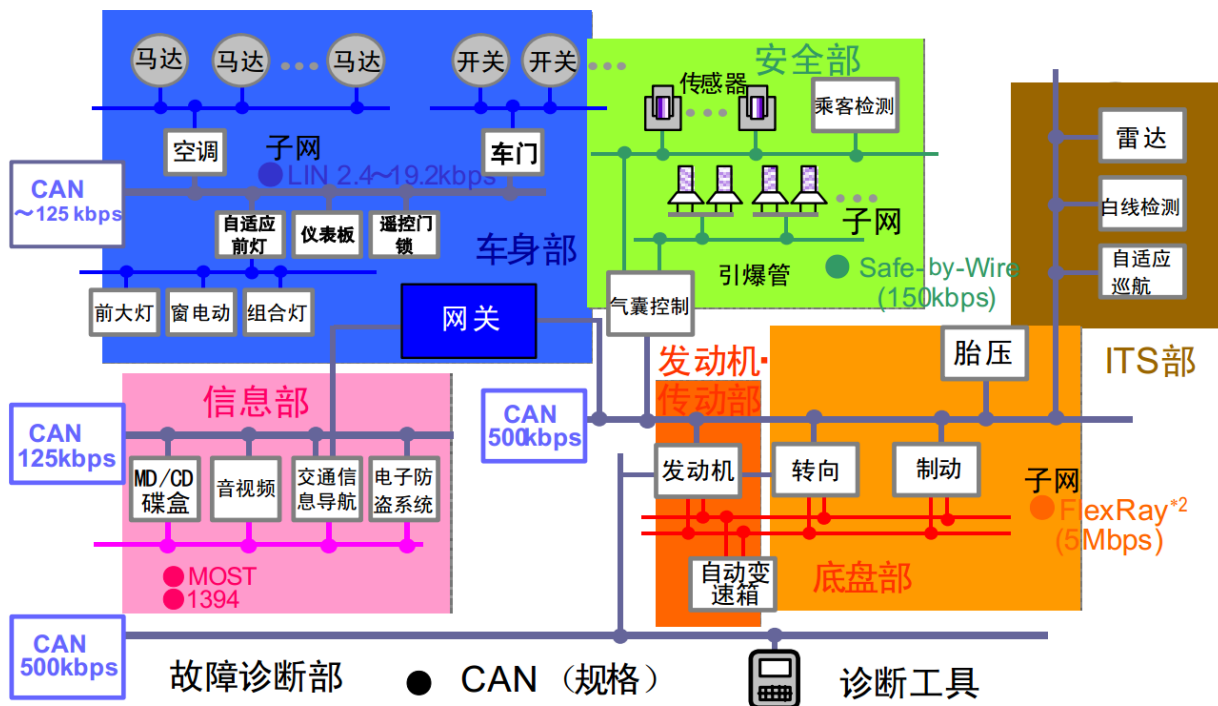


图 31.1.1 车载网络构想图

31.1.2 CAN 的特点

CAN 通信协议具有以下特点:

(1)、多主控制

在总线空闲时,所有的单元都可开始发送消息(多主控制)。

最先访问总线的单元可获得发送权(CSMA/CA 方式*1)。

多个单元同时开始发送时,发送高优先级 ID 消息的单元可获得发送权。

(2)、消息的发送

在 CAN 协议中,所有的消息都以固定的格式发送。总线空闲时,所有与总线相连的单元都可以开始发送新消息。两个以上的单元同时开始发送消息时,根据标识符(Identifier 以下称为 ID)决定优先级。ID 并不是表示发送的目的地址,而是表示访问总线的消息的优先级。两个以上的单元同时开始发送消息时,对各消息 ID 的每个位进行逐个仲裁比较。仲裁获胜(被判定为优先级最高)的单元可继续发送消息,仲裁失利的单元则立刻停止发送而进行接收工作。

(3)、系统的柔软性

与总线相连的单元没有类似于“地址”的信息。因此在总线上增加单元时,连接在总线上的其它单元的软硬件及应用层都不需要改变。

(4)、通信速度

根据整个网络的规模,可设定适合的通信速度。

在同一网络中,所有单元必须设定成统一的通信速度。即使有一个单元的通信速度与其它的不一样,此单元也会输出错误信号,妨碍整个网络的通信。不同网络间则可以有不同的通信速度。

(5)、远程数据请求

可通过发送“遥控帧”请求其他单元发送数据。

(6)、具有错误检测功能·错误通知功能·错误恢复功能

所有的单元都可以检测错误(错误检测功能)。

检测出错误的单元会立即同时通知其他所有单元(错误通知功能)。

正在发送消息的单元一旦检测出错误,会强制结束当前的发送。强制结束发送的单元会不断反复地重新发送此消息直到成功发送为止(错误恢复功能)。

(7)、故障封闭

CAN 可以判断出错误的类型是总线上暂时的数据错误(如外部噪声等)还是持续的数据错误(如单元内部故障、驱动器故障、断线等)。由此功能,当总线上发生持续数据错误时,可将引起此故障的单元从总线上隔离出去。

(8)、连接

CAN 总线是可同时连接多个单元的总线。可连接的单元总数理论上是没有限制的。但实际上可连接的单元数受总线上的时间延迟及电气负载的限制。降低通信速度,可连接的单元数增加;提高通信速度,则可连接的单元数减少。

31.1.3 CAN 的电气属性

CAN 总线使用两根线来连接各个单元: CAN_H 和 CAN_L, CAN 控制器通过判断这两根线上的电位差来得到总线电平, CAN 总线电平分为显性电平和隐性电平两种。显性电平表示逻辑“0”,此时 CAN_H 电

平比 CAN_L 高, 分别为 3.5V 和 1.5V, 电位差为 2V。隐形电平表示逻辑“1”, 此时 CAN_H 和 CAN_L 电压都为 2.5V 左右, 电位差为 0V。CAN 总线就通过显性和隐形电平的变化来将具体的数据发送出去, 如下图所示:

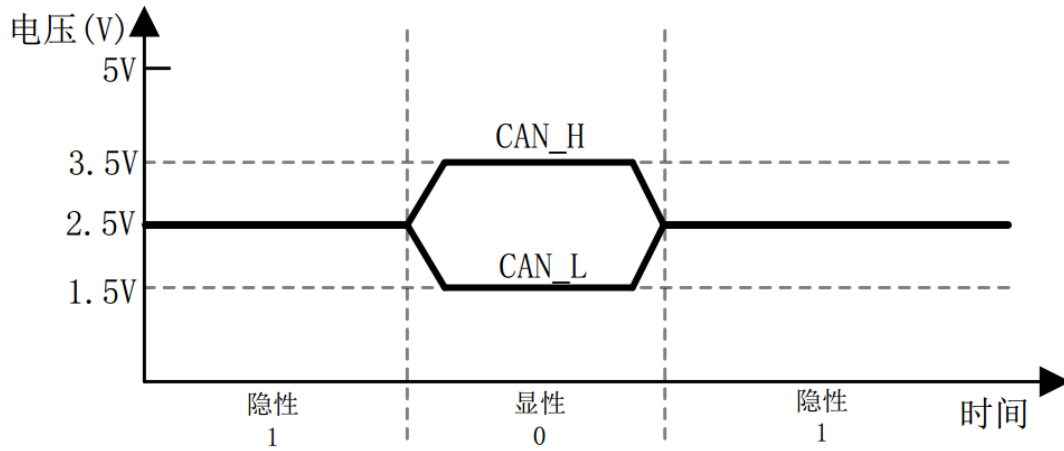


图 31.1.2 CAN 电气属性

CAN 总线上没有节点传输数据的时候一直处于隐性状态, 也就是说总线空闲状态的时候一直处于隐性。

31.1.4 CAN 网络拓扑

CAN 是一种分布式的控制总线, CAN 总线作为一种控制器局域网, 和普通的以太网一样, 它的网络由很多的 CAN 节点构成, 其网络拓扑结构如下图所示:

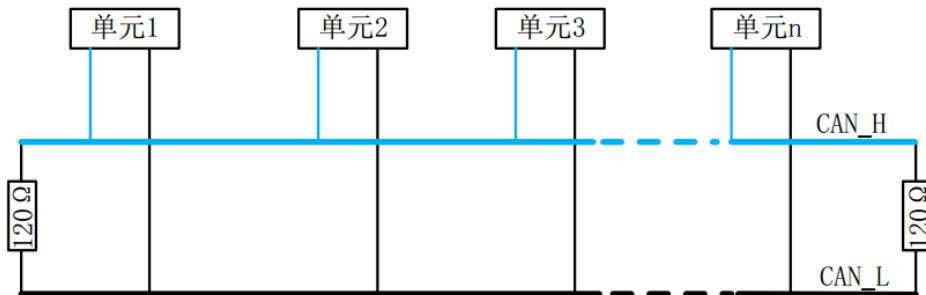


图 31.1.3 CAN 网络拓扑图

CAN 网络的每个节点非常简单, 均由一个 MCU (微控制器)、一个 CAN 控制器和一个 CAN 收发器构成, 然后通过 CAN_H 和 CAN_L 这两根线连接在一起形成一个 CAN 局域网。CAN 能够使用多种物理介质, 例如双绞线、光纤等。最常用的就是双绞线。信号使用差分电压传送, 两条信号线被称为“CAN_H”和“CAN_L”, 在我们的开发板上, CAN 接口使用了这两条信号线, CAN 接口也只有这两条信号线。

由此可知, CAN 控制器局域网和普通的以太网一样, 每一个 CAN 节点就相当于局域网中的一台主机。

途中所有的 CAN 节点单元都采用 CAN_H 和 CAN_L 这两根线连接在一起, CAN_H 接 CAN_H、CAN_L 接 CAN_L, CAN 总线两端要各接一个 120Ω 的端接电阻, 用于匹配总线阻抗, 吸收信号反射及回拨, 提高数据通信的抗干扰能力以及可靠性。

CAN 总线传输速度可达 1Mbps/S, 最新的 CAN-FD 最高速度可达 5Mbps/S, 甚至更高, CAN-FD 不在本章讨论范围, 感兴趣的可以自行查阅相关资料。CAN 传输速度和总线距离有关, 总线距离越短, 传输速度越快。

31.1.5 CAN 总线通信模型

CAN 总线传输协议参考了 OSI 开放系统互连模型，也就是前面所介绍的 OSI 七层模型（具体详情参考 29.2 小节）。虽然 CAN 传输协议参考了 OSI 七层模型，但是实际上 CAN 协议只定义了“传输层”、“数据链路层”以及“物理层”这三层，而应用层协议可以由 CAN 用户定义成适合特别工业领域的任何方案。已在工业控制和制造业领域得到广泛应用的标准是 DeviceNet，这是为 PLC 和智能传感器设计的。在汽车工业，许多制造商都有他们自己的应用层协议标准。

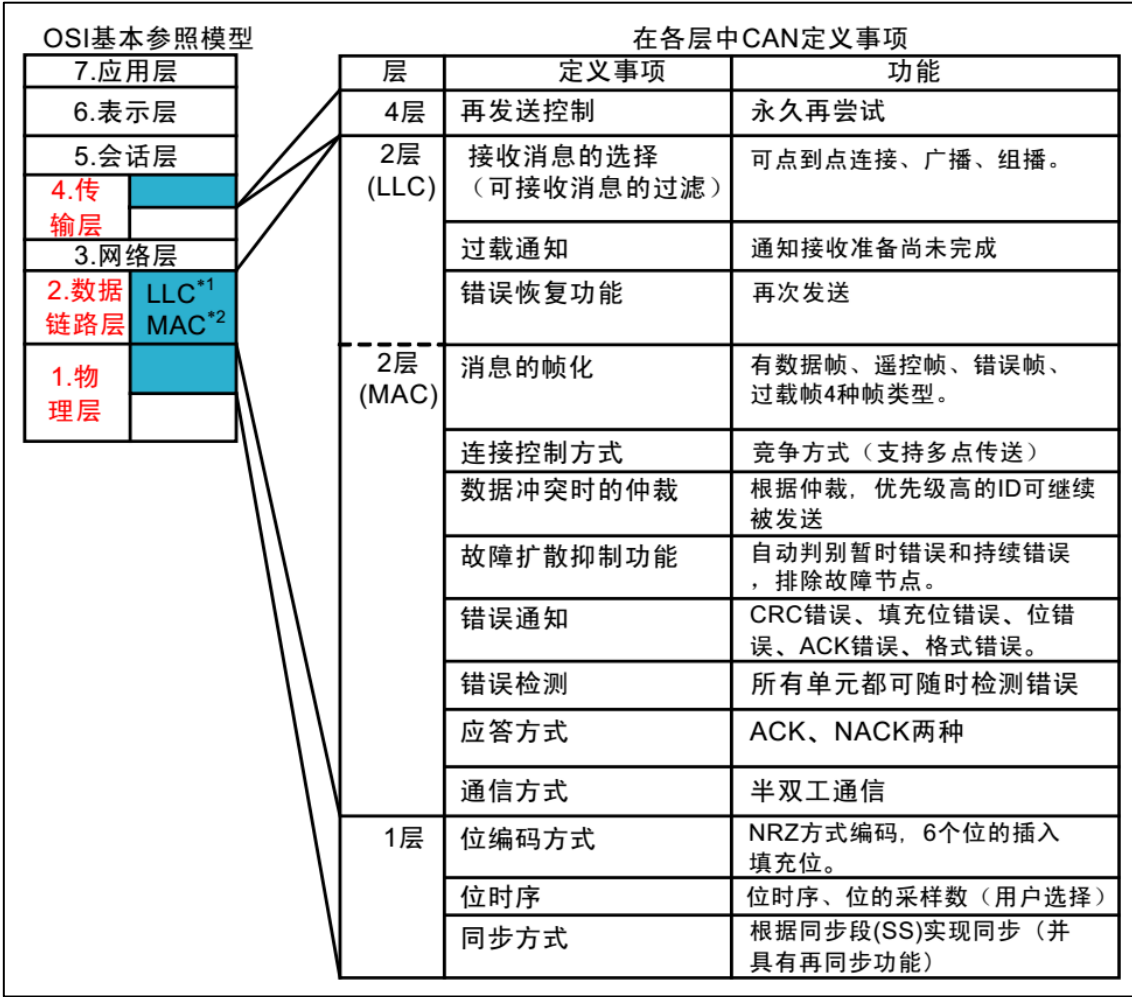


图 31.1.4 OSI 七层模型和 CAN 协议

CAN 协议只参考了 OSI 模型中的“传输层”、“数据链路层”以及“物理层”，因此出现了各种不同的应用层协议，比如用在自动化技术的现场总线标准 DeviceNet，用于工业控制的 CanOpen,用于乘用车的诊断协议 OBD、UDS(统一诊断服务, ISO14229)，用于商用车的 CAN 总线协议 SAEJ1939。

数据链路层分为 MAC 子层和 LLC 子层，MAC 子层是 CAN 协议的核心部分。数据链路层的功能是将物理层收到的信号组织成有意义的消息，并提供传送错误控制等传输控制的流程。具体地说，就是消息的帧化、仲裁、应答、错误的检测或报告。数据链路层的功能通常在 CAN 控制器的硬件中执行。

在物理层定义了信号实际的发送方式、位时序、位的编码方式及同步的步骤。但具体地说，信号电平、通信速度、采样点、驱动器和总线的电气特性、连接器的形态等均未定义。这些必须由用户根据系统需求自行确定。

31.1.6 CAN 帧的种类

CAN 通信协议定义了 5 种类型的报文帧，分别是：数据帧、遥控帧、错误帧、过载帧、间隔帧。通信是通过这 5 种类型的帧进行的。其中数据帧和遥控帧有标准格式和扩展格式两种，标准格式有 11 位标识符 (ID)，扩展格式有 29 个标识符 (ID)。这 5 种类型的帧如下表所示：

帧	用途
数据帧	用于发送单元向接收单元传送数据的帧。
遥控帧	用于接收单元向具有相同 ID 的发送单元请求数据的帧。
错误帧	用于当检测出错误时向其它单元通知错误的帧。
过载帧	用于接收单元通知其尚未做好接收准备的帧。
间隔帧	用于将数据帧及遥控帧与前面的帧分离开来的帧。

表 31.1.1 帧的种类及用途

其中数据帧是使用最多的帧类型，这里重点介绍一下数据帧，数据帧结构如下图所示：

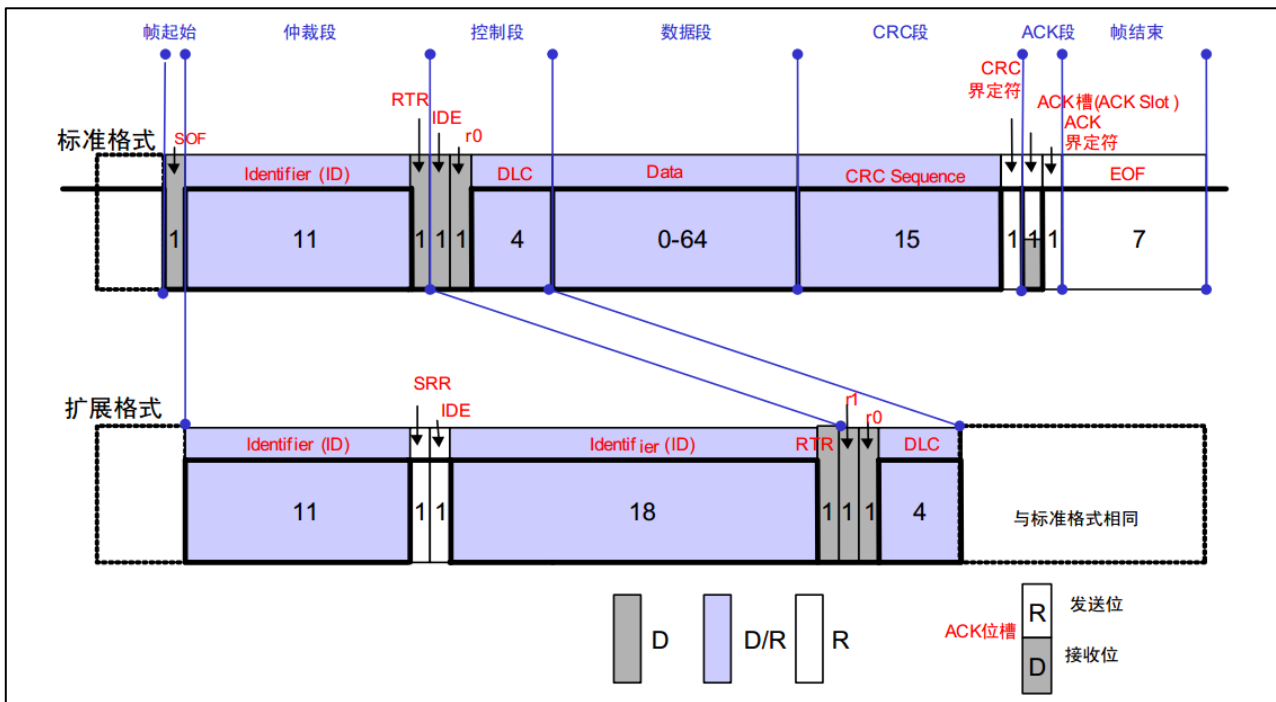


图 31.1.5 数据帧的构成

图 31.1.5 给出了数据帧标准格式和扩展格式两种帧结构，图中 D 表示显性电平 0，R 表示隐性电平 1，D/R 表示显性或隐性，也就是 0 或 1，我们来简单分析一下数据帧的这 7 个段。

数据帧由 7 个段构成：

(1)、帧起始

表示数据帧开始的段。

(2)、仲裁段

表示该帧优先级的段。

(3)、控制段

表示数据的字节数及保留位的段。

(4)、数据段

数据的内容, 可发送 0~8 个字节的数。

(5)、CRC 段

检查帧的传输错误的段。

(6)、ACK 段

表示确认正常接收的段。

(7)、帧结束

表示数据帧结束的段。

关于更加详细的内容, 请大家参考由瑞萨电子编写的《CAN 入门教程》, 本小节内容到此结束! 接下来向大家介绍 CAN 的应用编程。

31.2 SocketCan 应用编程

由于 Linux 系统将 CAN 设备作为网络设备进行管理, 因此在 CAN 总线应用开发方面, Linux 提供了 SocketCAN 应用编程接口, 使得 CAN 总线通信近似于和以太网的通信, 应用程序开发接口更加通用, 也更加灵活。

SocketCAN 中大部分的数据结构和函数在头文件 `linux/can.h` 中进行了定义, 所以, 在我们的应用程序中一定要包含 `<linux/can.h>` 头文件。

31.2.1 创建 socket 套接字

CAN 总线套接字的创建采用标准的网络套接字操作来完成, 网络套接字在头文件 `<sys/socket.h>` 中定义。创建 CAN 套接字的方法如下:

```
int sockfd = -1;

/* 创建套接字 */
sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
if(0 > sockfd) {
    perror("socket error");
    exit(EXIT_FAILURE);
}
```

`socket` 函数在 30.2.1 小节中给大家详细介绍过, 第一个参数用于指定通信域, 在 SocketCan 中, 通常将其设置为 `PF_CAN`, 指定为 CAN 通信协议; 第二个参数用于指定套接字的类型, 通常将其设置为 `SOCK_RAW`; 第三个参数通常设置为 `CAN_RAW`。

31.2.2 将套接字与 CAN 设备进行绑定

譬如, 将创建的套接字与 `can0` 进行绑定, 示例代码如下所示:

```
.....
struct ifreq ifr = {0};
struct sockaddr_can can_addr = {0};
int ret;

.....
```

```
strcpy(ifr.ifr_name, "can0"); //指定名字
ioctl(sockfd, SIOCGIFINDEX, &ifr);

can_addr.can_family = AF_CAN; //填充数据
can_addr.can_ifindex = ifr.ifr_ifindex;

/* 将套接字与 can0 进行绑定 */
ret = bind(sockfd, (struct sockaddr *)&can_addr, sizeof(can_addr));
if (0 > ret) {
    perror("bind error");
    close(sockfd);
    exit(EXIT_FAILURE);
}
```

bind()函数在 30.2.2 小节中给大家详细介绍过，这里不再重述！

这里出现了两个结构体：struct ifreq 和 struct sockaddr_can，其中 struct ifreq 定义在<net/if.h>头文件中，而 struct sockaddr_can 定义在<linux/can.h>头文件中，这些结构体就不给大家解释了，有兴趣的自己去查，笔者也不太懂！

31.2.3 设置过滤规则

在我们的应用程序中，如果没有设置过滤规则，应用程序默认会接收所有 ID 的报文；如果我们的应用程序只需要接收某些特定 ID 的报文（亦或者不接受所有报文，只发送报文），则可以通过 setsockopt 函数设置过滤规则，譬如某应用程序只接收 ID 为 0x60A 和 0x60B 的报文帧，则可将其它不符合规则的帧全部给过滤掉，示例代码如下所示：

```
struct can_filter rfilter[2]; //定义一个 can_filter 结构体对象

// 填充过滤规则，只接收 ID 为(can_id & can_mask)的报文
rfilter[0].can_id = 0x60A;
rfilter[0].can_mask = 0x7FF;
rfilter[1].can_id = 0x60B;
rfilter[1].can_mask = 0x7FF;

// 调用 setsockopt 设置过滤规则
setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
```

struct can_filter 结构体中只有两个成员，can_id 和 can_mask。

如果应用程序不接收所有报文，在这种仅仅发送数据的应用中，可以在内核中省略接收队列，以此减少 CPU 资源的消耗。此时可将 setsockopt()函数的第 4 个参数设置为 NULL，将第 5 个参数设置为 0，如下所示：

```
setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
```

31.2.4 数据发送/接收

在数据收发的内容方面，CAN 总线与标准套接字通信稍有不同，每一次通信都采用 struct can_frame 结构体将数据封装成帧。结构体定义如下：

示例代码 31.2.1 struct can_frame 结构体

```
struct can_frame {
    canid_t can_id;    /* CAN 标识符 */
    __u8 can_dlc;     /* 数据长度 (最长为 8 个字节) */
    __u8 __pad;      /* padding */
    __u8 __res0;     /* reserved / padding */
    __u8 __res1;     /* reserved / padding */
    __u8 data[8];    /* 数据 */
};
```

can_id 为帧的标识符, 如果是标准帧, 就使用 can_id 的低 11 位; 如果为扩展帧, 就使用 0~28 位。can_id 的第 29、30、31 位是帧的标志位, 用来定义帧的类型, 定义如下:

```
/* special address description flags for the CAN_ID */
#define CAN_EFF_FLAG 0x80000000U /* 扩展帧的标识 */
#define CAN_RTR_FLAG 0x40000000U /* 远程帧的标识 */
#define CAN_ERR_FLAG 0x20000000U /* 错误帧的标识, 用于错误检查 */

/* mask */
#define CAN_SFF_MASK 0x000007FFU /* <can_id & CAN_SFF_MASK>获取标准帧 ID */
#define CAN_EFF_MASK 0x1FFFFFFFU /* <can_id & CAN_EFF_MASK>获取标准帧 ID */
#define CAN_ERR_MASK 0x1FFFFFFFU /* omit EFF, RTR, ERR flags */
```

(1)、数据发送

对于数据发送, 使用 write()函数来实现, 譬如要发送的数据帧包含了三个字节数据 0xA0、0xB0 以及 0xC0, 帧 ID 为 123, 可采用如下方法进行发送:

```
struct can_frame frame; //定义一个 can_frame 变量
int ret;

frame.can_id = 123; //如果为扩展帧, 那么 frame.can_id = CAN_EFF_FLAG | 123;
frame.can_dlc = 3; //数据长度为 3
frame.data[0] = 0xA0; //数据内容为 0xA0
frame.data[1] = 0xB0; //数据内容为 0xB0
frame.data[2] = 0xC0; //数据内容为 0xC0

ret = write(sockfd, &frame, sizeof(frame)); //发送数据
if(sizeof(frame) != ret) //如果 ret 不等于帧长度, 就说明发送失败
    perror("write error");
```

如果要发送远程帧(帧 ID 为 123), 可采用如下方法进行发送:

```
struct can_frame frame;

frame.can_id = CAN_RTR_FLAG | 123;

write(sockfd, &frame, sizeof(frame));
```

(2)、数据接收

数据接收使用 read()函数来实现, 如下所示:

```
struct can_frame frame;
```

```
int ret = read(sockfd, &frame, sizeof(frame));
```

(3)、错误处理

当应用程序接收到一帧数据之后, 可以通过判断 `can_id` 中的 `CAN_ERR_FLAG` 位来判断接收的帧是否为错误帧。如果为错误帧, 可以通过 `can_id` 的其他符号位来判断错误的具体原因。错误帧的符号位在头文件 `<linux/can/error.h>` 中定义。

```
/* error class (mask) in can_id */
#define CAN_ERR_TX_TIMEOUT 0x00000001U /* TX timeout (by netdevice driver) */
#define CAN_ERR_LOSTARB 0x00000002U /* lost arbitration / data[0] */
#define CAN_ERR_CRTL 0x00000004U /* controller problems / data[1] */
#define CAN_ERR_PROT 0x00000008U /* protocol violations / data[2..3] */
#define CAN_ERR_TRX 0x00000010U /* transceiver status / data[4] */
#define CAN_ERR_ACK 0x00000020U /* received no ACK on transmission */
#define CAN_ERR_BUSOFF 0x00000040U /* bus off */
#define CAN_ERR_BUSERROR 0x00000080U /* bus error (may flood) */
#define CAN_ERR_RESTARTED 0x00000100U /* controller restarted */
.....
.....
```

31.2.5 回环功能设置

在默认情况下, CAN 的本地回环功能是开启的, 可以使用下面的方法关闭或开启本地回环功能:

```
int loopback = 0; //0 表示关闭, 1 表示开启(默认)
```

```
setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

在本地回环功能开启的情况下, 所有的发送帧都会被回环到与 CAN 总线接口对应的套接字上。

31.3 CAN 应用编程实战

本小节我们来编写简单地 CAN 应用程序。在 Linux 系统中, CAN 总线设备作为网络设备被系统进行统一管理。在控制台下, CAN 总线的配置和以太网的配置使用相同的命令。

使用 `ifconfig` 命令查看 CAN 设备, 如下所示:


```
root@ATK-IMX6U:~# ifconfig -a
can0    Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        NOARP MTU:16  Metric:1
        RX packets:0  errors:0  dropped:0  overruns:0  frame:0
        TX packets:0  errors:0  dropped:0  overruns:0  carrier:0
        collisions:0 txqueuelen:10
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
        Interrupt:26
        CAN设备

eth0    Link encap:Ethernet  HWaddr 88:f9:a5:4e:b9:35
        inet addr:192.168.1.193  Bcast:192.168.1.255  Mask:255.255.255.0
        inet6 addr: fe80::8af9:a5ff:fe4e:b935/64  Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:96601  errors:0  dropped:726  overruns:0  frame:0
        TX packets:300  errors:0  dropped:0  overruns:0  carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:10960253 (10.4 MiB)  TX bytes:45909 (44.8 KiB)

eth1    Link encap:Ethernet  HWaddr 88:43:98:84:13:bf
        UP BROADCAST MULTICAST DYNAMIC  MTU:1500  Metric:1
        RX packets:0  errors:0  dropped:0  overruns:0  frame:0
        TX packets:0  errors:0  dropped:0  overruns:0  carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

图 31.3.1 查看 can 设备

我们的开发板上只有一个 CAN 接口，即 can0，如下图如所示：

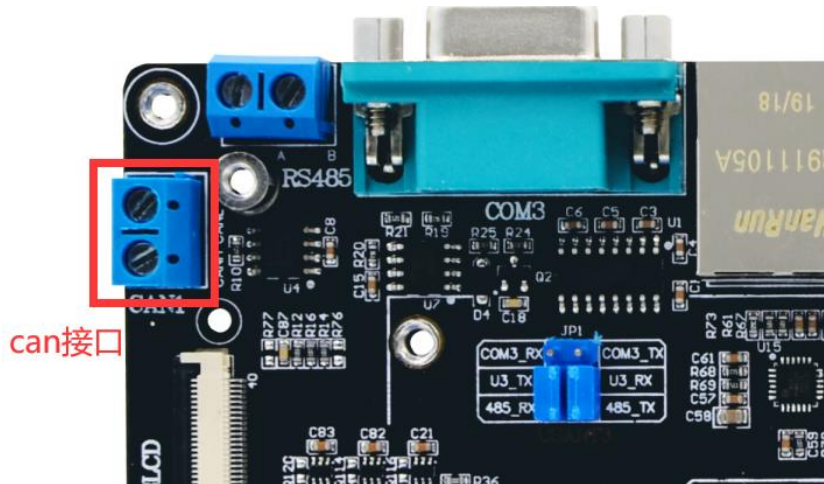


图 31.3.2 ALPHA 板上的 can 接口

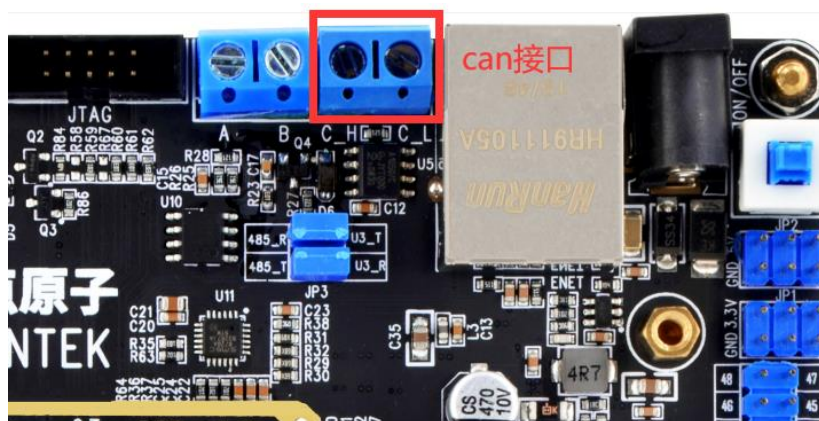


图 31.3.3 Mini 板上的 can 接口

在开发板出厂系统中，提供了一些工具用于测试 can，我们可以使用这些工具进行测试。由于我们板子上只有一个 can 接口，如果想要进行测试，可以使用两块开发板，将它们的 CAN 接口进行对接，进行收发

测试, 如果大家手中只有一块开发板, 那这种方式自然是行不通的; 除此之外, 我们还可以使用一些测试 CAN 的设备进行测试, 譬如 CAN 分析仪, 这里笔者使用 CAN 分析仪来进行测试。

如果大家有购买 CAN 分析仪等这类 CAN 测试设备, 就直接使用它进行测试即可! 关于 CAN 分析仪以及配套上位机的使用方法, 请大家参考它们提供的使用说明书, 如果不会使用, 请咨询厂家的技术支持!

在测试之前, 使用 CAN 分析仪或者其它测试 CAN 的设备连接到 ALPHA|Mini 开发板底板上的 CAN 接口处, CANH 连接到仪器的 CANH、CANL 连接到 CAN 仪器的 CANL。并且打开 CAN 分析仪配套的上位机软件, 笔者使用的是创芯科技的推出的一款 CAN 分析仪, 其配套的上位机软件界面如下所示:

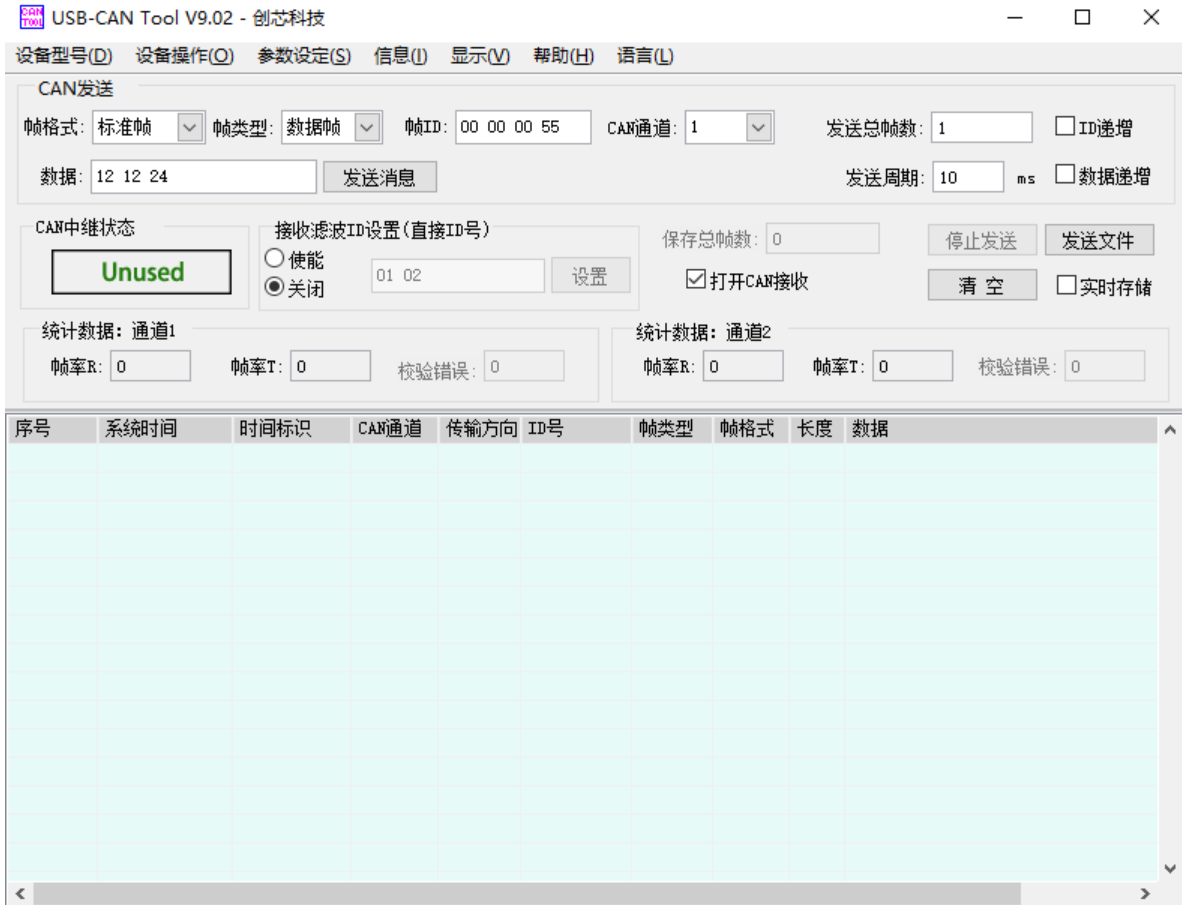


图 31.3.4 CAN 上位机

设备连接好之后, 通过上位机软件启动 CAN 分析仪, 接下来我们进行测试。

在进行测试之前需要对开发板上的 can 设备进行配置, 执行以下命令:

```
ifconfig can0 down          #先关闭 can0 设备
ip link set can0 up type can bitrate 1000000 triple-sampling on    #设置波特率为 1000000
```

注意, CAN 分析仪设置的波特率要和开发板 CAN 设备的波特率一致!

配置完成之后, 接着可以使用 cansend 命令发送数据,

```
cansend can0 123#01.02.03.04.05.06.07.08
```

```
root@ATK-IMX6U: ~#
root@ATK-IMX6U: ~# cansend can0 123#01.02.03.04.05.06.07.08
root@ATK-IMX6U: ~# cansend can0 123#01.02.03.04.05.06.07.08
root@ATK-IMX6U: ~# cansend can0 123#01.02.03.04.05.06.07.08
root@ATK-IMX6U: ~# cansend can0 123#01.02.03.04.05.06.07.08
root@ATK-IMX6U: ~#
```

图 31.3.5 使用 cansend 命令发送数据

“#”号前面的 123 表示帧 ID, 后面的数字表示要发送的数据, 此时上位机便会接收到开发板发送过来的数据, 如下所示:

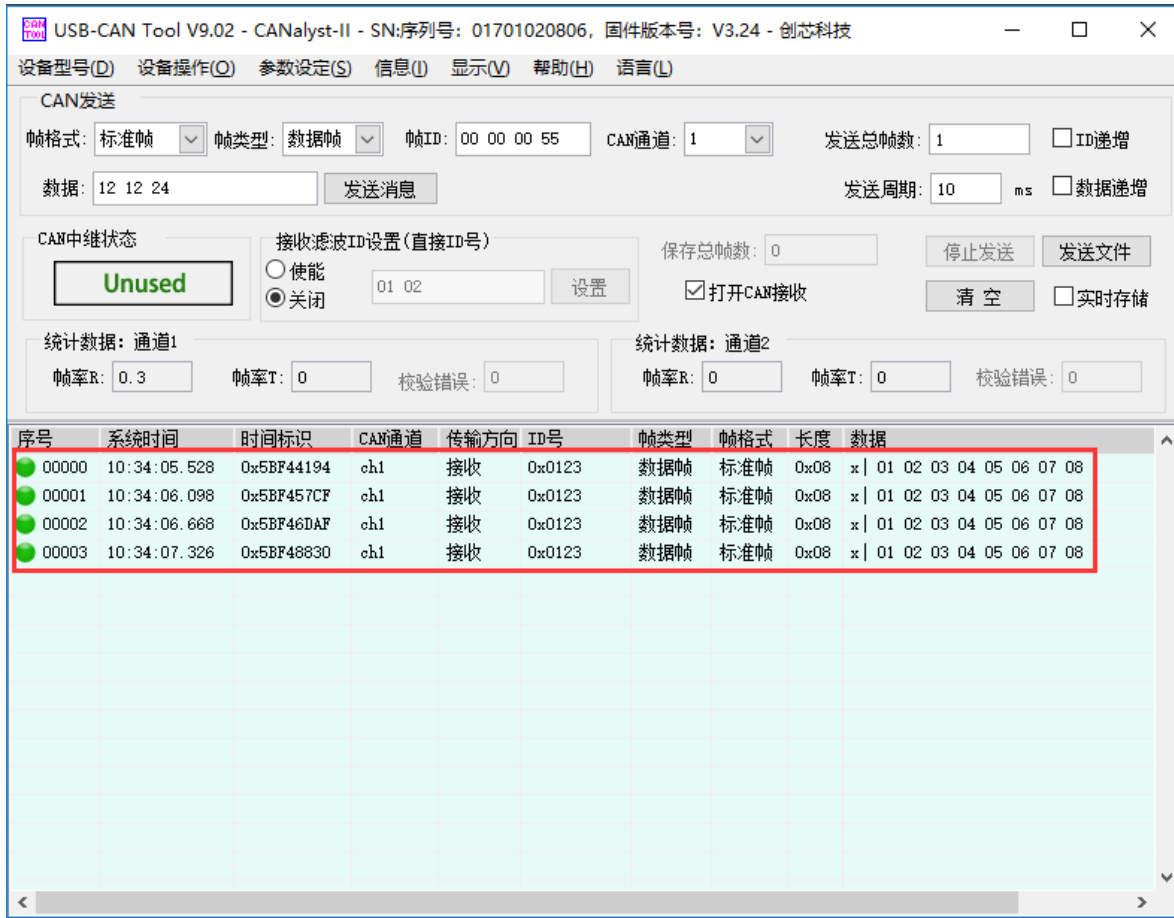


图 31.3.6 上位机软件收到数据

接着测试开发板接收 CAN 数据, 首先在开发板上执行 `candump` 命令:

```
candump -ta can0
```

接着通过 CAN 分析仪上位机软件, 向开发板发送数据, 如下所示:

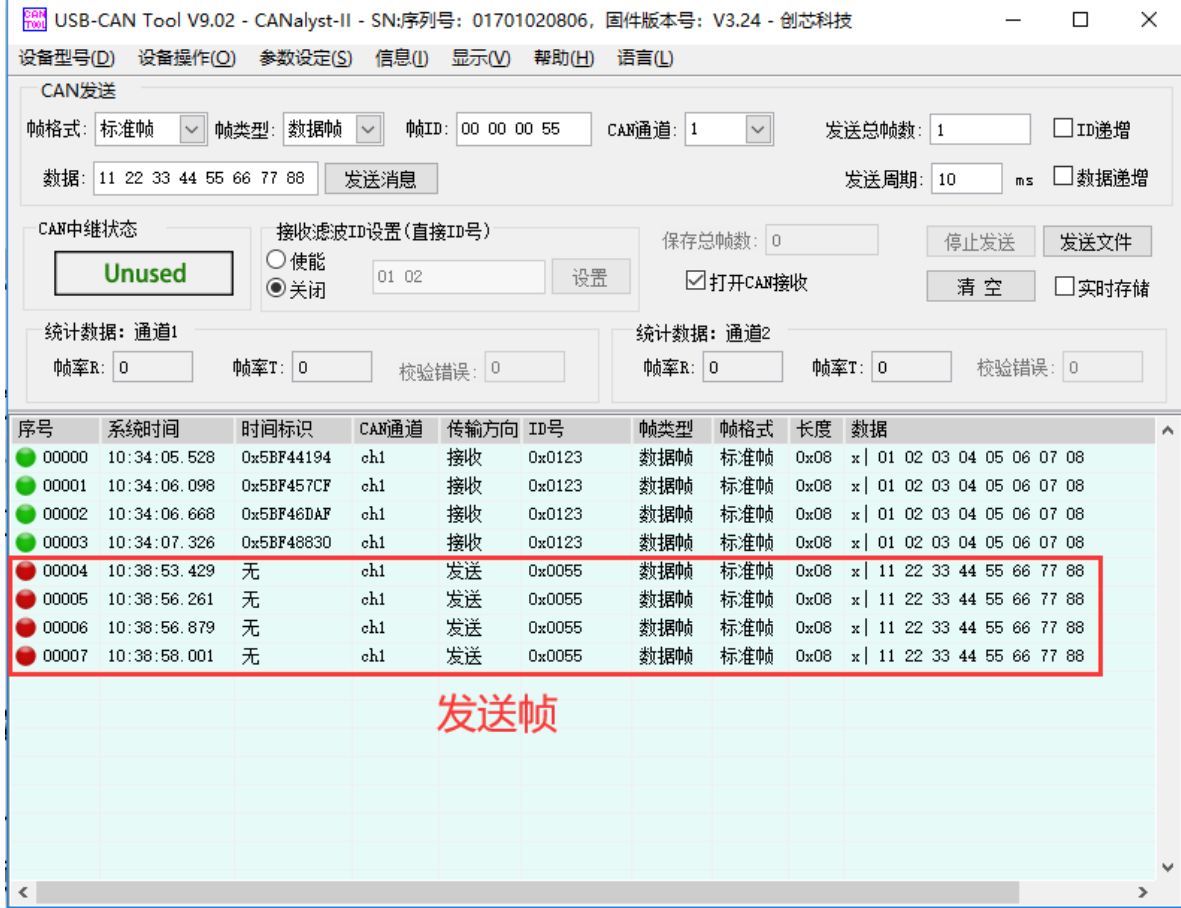


图 31.3.7 CAN 上位机发送数据

此时开发板便能接收到上位机发送过来的数据，如下所示：

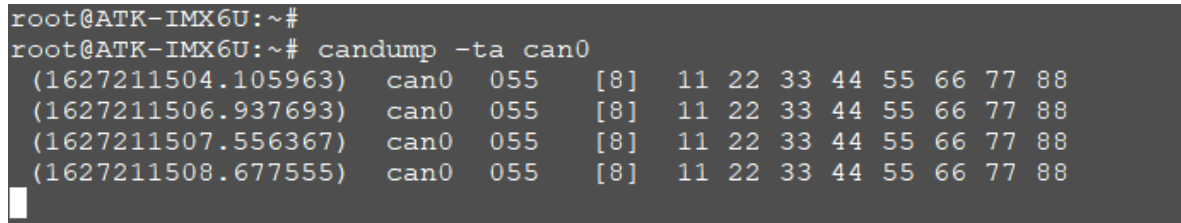


图 31.3.8 开发板接收到数据

测试完成之后，按 Ctrl + C 退出 candump 程序，关于 CAN 的测试就到这里了。

31.3.1 CAN 数据发送实例

通过前面的学习，本小节我们来编写一个非常简单地 CAN 数据发送的示例代码，代码笔者已经写好了，如下所示。

本例程序源码对应的路径为：**开发板光盘->11、Linux C 应用编程例程源码->31_can->can_write.c**。

示例代码 31.3.1 CAN 数据发送示例代码

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : can_write.c
作者 : 邓涛
版本 : V1.0

```

描述 : 一个简单地 CAN 数据发送示例代码

其他 : 无

论坛 : www.openedv.com

日志 : 初版 V1.0 2021/7/20 邓涛创建

*****/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <net/if.h>

int main(void)
{
    struct ifreq ifr = {0};
    struct sockaddr_can can_addr = {0};
    struct can_frame frame = {0};
    int sockfd = -1;
    int ret;

    /* 打开套接字 */
    sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    if(0 > sockfd) {
        perror("socket error");
        exit(EXIT_FAILURE);
    }

    /* 指定 can0 设备 */
    strcpy(ifr.ifr_name, "can0");
    ioctl(sockfd, SIOCGIFINDEX, &ifr);
    can_addr.can_family = AF_CAN;
    can_addr.can_ifindex = ifr.ifr_ifindex;

    /* 将 can0 与套接字进行绑定 */
    ret = bind(sockfd, (struct sockaddr *)&can_addr, sizeof(can_addr));
    if (0 > ret) {
        perror("bind error");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
}
```

```

}

/* 设置过滤规则: 不接受任何报文、仅发送数据 */
setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);

/* 发送数据 */
frame.data[0] = 0xA0;
frame.data[1] = 0xB0;
frame.data[2] = 0xC0;
frame.data[3] = 0xD0;
frame.data[4] = 0xE0;
frame.data[5] = 0xF0;
frame.can_dlc = 6; //一次发送 6 个字节数据
frame.can_id = 0x123; //帧 ID 为 0x123,标准帧

for (;;) {

    ret = write(sockfd, &frame, sizeof(frame)); //发送数据
    if(sizeof(frame) != ret) { //如果 ret 不等于帧长度,就说明发送失败
        perror("write error");
        goto out;
    }

    sleep(1); //一秒钟发送一次
}

out:
/* 关闭套接字 */
close(sockfd);
exit(EXIT_SUCCESS);
}

```

代码比较简单, 这里就不再给做过多的解释说明了, 直接编译:

```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
can_write.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o can_write can_write.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
can_write can_write.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 31.3.9 编译示例代码

将编译得到的可执行文件 `can_write` 拷贝到开发板 Linux 系统 `/home/root` 目录下, 然后运行该程序, 程序运行后, 会每隔 1 秒中通过 `can0` 发送一帧数据, 一次发送 6 个字节数据, 帧 ID 为 `0x123`, 此时 CAN 上位机便会接收到开发板发送过来的数据, 如下所示:

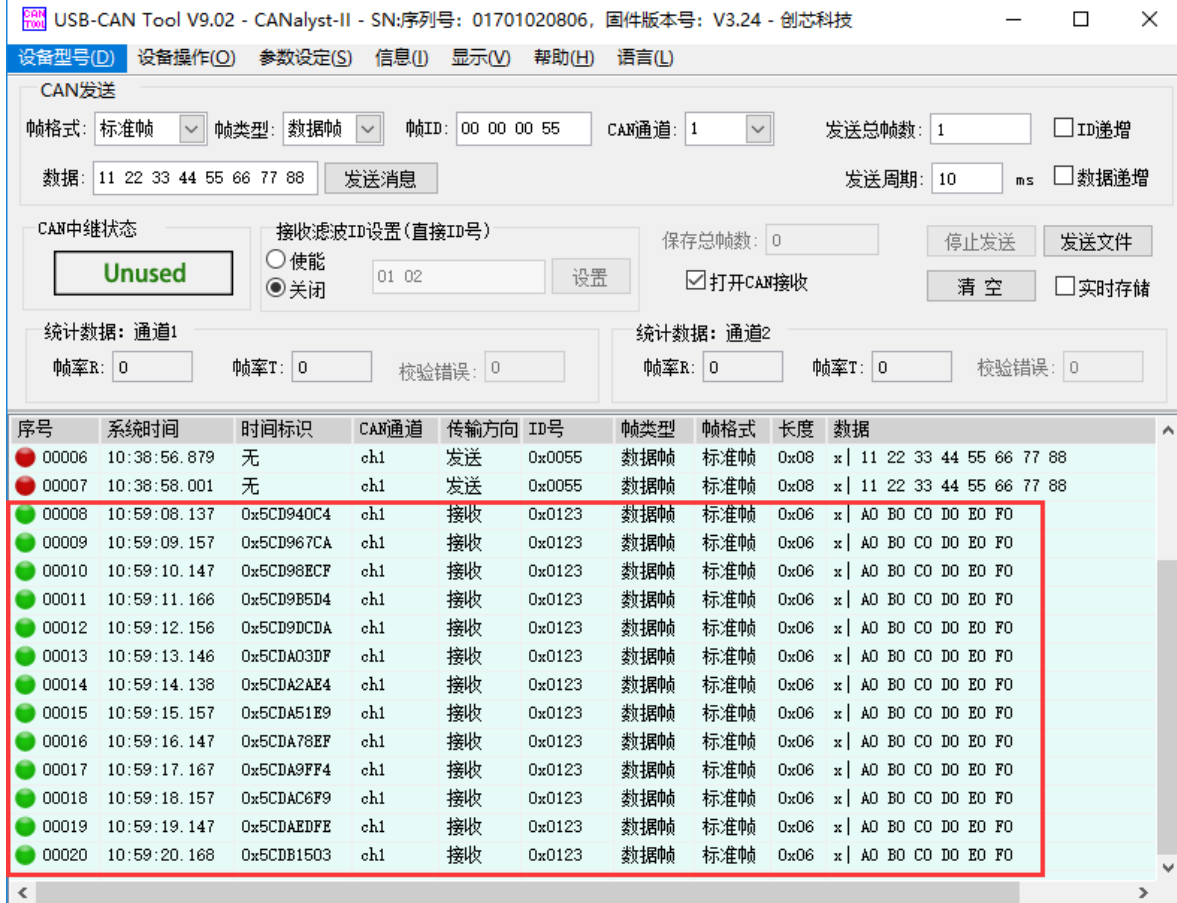


图 31.3.10 CAN 上位机接收到数据

31.3.2 CAN 数据接收实例

本小节我们来编写一个非常简单地 CAN 数据接收的示例代码，代码笔者已经写好了，如下所示。本例程源码对应的路径为：[开发板光盘->11、Linux C 应用编程例程源码->31_can->can_read.c](#)。

示例代码 31.3.2 CAN 数据接收示例代码

```

/*****
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
文件名 : can_read.c
作者 : 邓涛
版本 : V1.0
描述 : 一个简单地 CAN 数据读取示例代码
其他 : 无
论坛 : www.openedv.com
日志 : 初版 V1.0 2021/7/20 邓涛创建
*****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
    
```

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include <net/if.h>

int main(void)
{
    struct ifreq ifr = {0};
    struct sockaddr_can can_addr = {0};
    struct can_frame frame = {0};
    int sockfd = -1;
    int i;
    int ret;

    /* 打开套接字 */
    sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    if(0 > sockfd) {
        perror("socket error");
        exit(EXIT_FAILURE);
    }

    /* 指定 can0 设备 */
    strcpy(ifr.ifr_name, "can0");
    ioctl(sockfd, SIOCGIFINDEX, &ifr);
    can_addr.can_family = AF_CAN;
    can_addr.can_ifindex = ifr.ifr_ifindex;

    /* 将 can0 与套接字进行绑定 */
    ret = bind(sockfd, (struct sockaddr *)&can_addr, sizeof(can_addr));
    if(0 > ret) {
        perror("bind error");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    /* 设置过滤规则 */
    //setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);

    /* 接收数据 */
    for (;;) {
        if(0 > read(sockfd, &frame, sizeof(struct can_frame))) {
            perror("read error");
        }
    }
}
```



```

        break;
    }

    /* 校验是否接收到错误帧 */
    if (frame.can_id & CAN_ERR_FLAG) {
        printf("Error frame!\n");
        break;
    }

    /* 校验帧格式 */
    if (frame.can_id & CAN_EFF_FLAG)    //扩展帧
        printf("扩展帧 <0x%08x> ", frame.can_id & CAN_EFF_MASK);
    else    //标准帧
        printf("标准帧 <0x%03x> ", frame.can_id & CAN_SFF_MASK);

    /* 校验帧类型: 数据帧还是远程帧 */
    if (frame.can_id & CAN_RTR_FLAG) {
        printf("remote request\n");
        continue;
    }

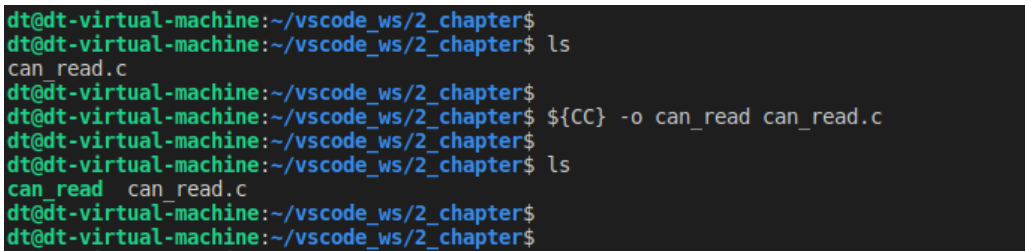
    /* 打印数据长度 */
    printf("[%d] ", frame.can_dlc);

    /* 打印数据 */
    for (i = 0; i < frame.can_dlc; i++)
        printf("%02x ", frame.data[i]);
    printf("\n");
}

/* 关闭套接字 */
close(sockfd);
exit(EXIT_SUCCESS);
}

```

编译示例代码:



```

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
can_read.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ${CC} -o can_read can_read.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
can_read  can_read.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

```

图 31.3.11 编译示例代码

将编译得到的可执行文件 `can_read` 拷贝到开发板 Linux 系统 `/home/root` 目录下, 然后执行程序, 接着通过 CAN 上位机向开发板发送数据, 此时开发板便会接收到上位机发送过来的数据, 如下所示:

```
root@ATK-IMX6U:~# ./can_read
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> remote request
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
扩展帧 <0x00000055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
标准帧 <0x055> [8] 11 22 33 44 55 66 77 88
```

图 31.3.12 开发板接收到上位机的数据

第三篇 进阶篇

本章开始将进入到进阶篇内容的学习,本篇将会规划一些实战小项目来跟大家一起完成,譬如 MQTT 实战小项目,具体的内容目前还在持续规划中!学习第三篇的内容可以帮助提升大家的编程能力、开发能力、以及遇到问题时的解决能力,当然前提条件是需要掌握好第一篇以及第二篇的内容。

第三十二章 CMake 入门与进阶

在前面两篇内容中,我们编写了很多示例程序,但这些示例程序都只有一个.c源文件,非常简单。所以,编译这些示例代码其实都非常简单,直接使用GCC编译器编译即可,连Makefile都不需要。但是,在实际的项目中,并非如此简单,一个工程中可能包含几十、成百甚至上千个源文件,这些源文件按照其类型、功能、模块分别放置在不同的目录中;面对这样的一个工程,通常会使用make工具进行管理、编译,make工具依赖于Makefile文件,通过Makefile文件来定义整个工程的编译规则,使用make工具来解析Makefile所定义的编译规则。

Makefile带来的好处就是——“自动化编译”,一旦写好,只需要一个make命令,整个工程完全按照Makefile文件定义的编译规则进行自动编译,极大的提高了软件开发的效率。大都数的IDE都有这个工具,譬如Visual C++的nmake、linux下的GNU make、Qt的qmake等等,这些make工具遵循着不同的规范和标准,对应的Makefile文件其语法、格式也不相同,这样就带来了一个严峻的问题:如果软件想跨平台,必须要保证能够在不同平台下编译,而如果使用上面的make工具,就得为每一种标准写一次Makefile,这将是一件让人抓狂的工作。

而cmake就是针对这个问题所诞生,允许开发者编写一种与平台无关的CMakeLists.txt文件来制定整个工程的编译流程,再根据具体的编译平台,生成本地化的Makefile和工程文件,最后执行make编译。

因此,对于大多数项目,我们应当考虑使用更自动化一些的cmake或者autotools来生成Makefile,而不是直接动手编写Makefile。

本章我们便来学习cmake,本章将会讨论如下主题内容。

- cmake是什么?
- cmake和Makefile之间的关系
- 如何使用cmake

32.1 cmake 简介

cmake 是一个跨平台的自动构建工具, 前面导语部分也已经给大家介绍了, cmake 的诞生主要是为了解决直接使用 make+Makefile 这种方式无法实现跨平台的问题, 所以 cmake 是可以实现跨平台的编译工具, 这是它最大的特点, 当然除了这个之外, cmake 还包含以下优点:

- 开放源代码。我们可以直接从 cmake 官网 <https://cmake.org/> 下载到它的源代码;
- 跨平台。cmake 并不直接编译、构建出最终的可执行文件或库文件, 它允许开发者编写一种与平台无关的 CMakeLists.txt 文件来制定整个工程的编译流程, cmake 工具会解析 CMakeLists.txt 文件语法规则, 再根据当前的编译平台, 生成本地化的 Makefile 和工程文件, 最后通过 make 工具来编译整个工程; 所以由此可知, cmake 仅仅只是根据不同平台生成对应的 Makefile, 最终还是通过 make 工具来编译工程源码, 但是 cmake 却是跨平台的。
- 语法规则简单。Makefile 语法规则比较复杂, 对于一个初学者来说, 通常并不那么友好, 并且 Makefile 语法规则在不同平台下往往是不一样的; 而 cmake 依赖的是 CMakeLists.txt 文件, 该文件的语法规则与平台无关, 并且语法规则简单、容易理解! cmake 工具通过解析 CMakeLists.txt 自动帮我们生成 Makefile, 这样就不需要我们自己手动编写 Makefile 了。

32.2 cmake 和 Makefile

直观上理解, cmake 就是用来产生 Makefile 的工具, 解析 CMakeLists.txt 自动生成 Makefile:

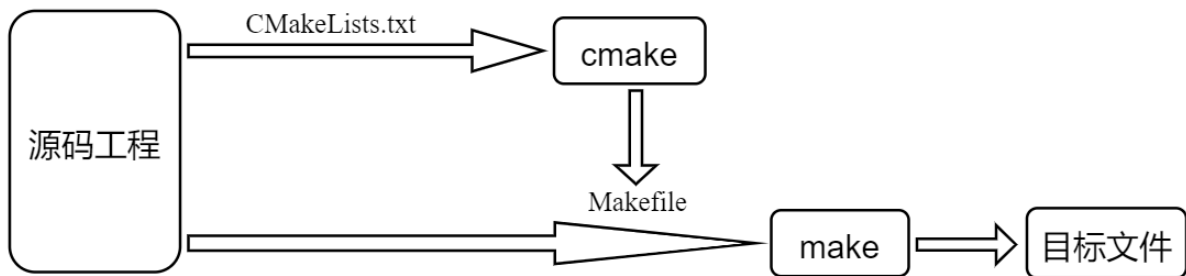


图 32.2.1 cmake 与 Makefile

除了 cmake 之外, 还有一些其它的自动构建工具, 常用的譬如 automake、autoconf 等, 有兴趣的朋友可以自己了解下。

32.3 cmake 的使用方法

cmake 就是一个工具命令, 在 Ubuntu 系统下通过 apt-get 命令可以在线安装, 如下所示:

```
sudo apt-get install cmake
```

```
dt@dt-virtual-machine:~$ sudo apt-get install cmake
[sudo] dt 的密码:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
cmake 已经是最新版 (3.5.1-1ubuntu3)。
升级了 0 个软件包, 新安装了 0 个软件包, 要卸载 0 个软件包, 有 154 个软件包未被升级。
dt@dt-virtual-machine:~$
```

图 32.3.1 安装 cmake 工具

笔者的 Ubuntu 系统上已经安装了 cmake 工具, 安装完成之后可以通过 cmake --version 命令查看 cmake 的版本号, 如下所示:

```
dt@dt-virtual-machine:~$ cmake --version
cmake version 3.5.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
dt@dt-virtual-machine:~$
```

图 32.3.2 查看 cmake 版本号

由上图可知, 当前系统安装的 `cmake` 对应的版本号为 3.5.1, `cmake` 工具版本更新也是比较快的, 从官网 <https://cmake.org/> 可知, `cmake` 最新版本为 3.22.0, 不过这都没关系, 其实使用哪个版本都是可以的, 差别并不会太大, 所以这个大家不用担心。

安装完 `cmake` 工具之后, 接着我们就来学习如何去使用 `cmake`。`cmake` 官方也给大家提供相应教程, 链接地址如下所示:

<https://cmake.org/documentation/> //文档总链接地址

<https://cmake.org/cmake/help/latest/guide/tutorial/index.html> //培训教程

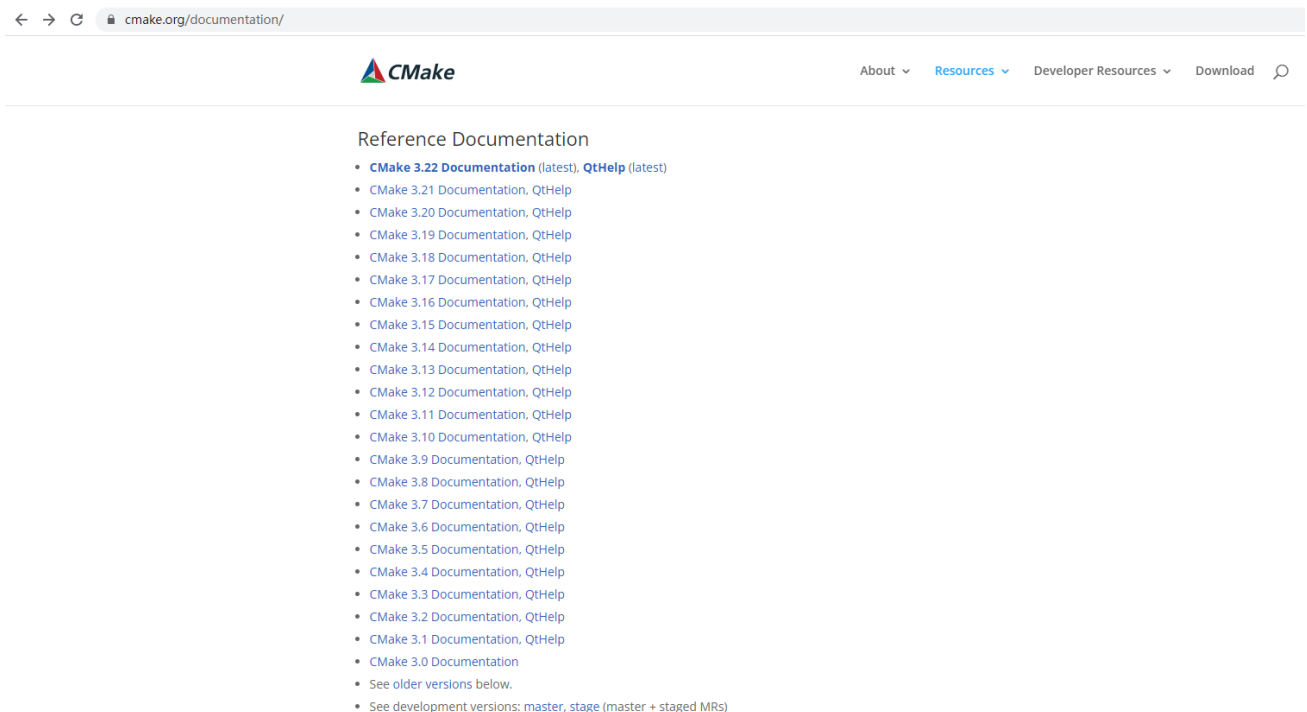


图 32.3.3 cmake 官方文档

如果大家自学能力强, 完全可以参考官方提供的培训教程学习 `cmake`; 对于 `cmake` 的学习, 笔者给大家两个建议:

- 从简单开始、再到复杂!
- 重点是自己动手多练习。

本小节我们将从一个非常简单的示例开始向大家介绍如何使用 `cmake`, 再从这个示例进一步扩展、提出更多需求, 来看看 `cmake` 如何去满足这些需求。

32.3.1 示例一: 单个源文件

单个源文件的程序通常是最简单的, 一个经典的 C 程序“Hello World”, 如何用 `cmake` 来进行构建呢?

```
//main.c
#include <stdio.h>
```

```
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

现在我们需要新建一个 CMakeLists.txt 文件, CMakeLists.txt 文件会被 cmake 工具解析, 就好比是 Makefile 文件会被 make 工具解析一样; CMakeLists.txt 创建完成之后, 在文件中写入如下内容:

```
project(HELLO)
add_executable(hello ./main.c)
```

写入完成之后, 保存退出, 当前工程目录结构如下所示:

```
├── CMakeLists.txt
└── main.c
```

在我们的工程目录下有两个文件, 源文件 main.c 和 CMakeLists.txt, 接着我们在工程目录下直接执行 cmake 命令, 如下所示:

```
cmake ./
```

cmake 后面携带的路径指定了 CMakeLists.txt 文件的所在路径, 执行结果如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
CMakeLists.txt main.c
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cmake ./
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake CMakeLists.txt main.c Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.3.4 执行 cmake

执行完 cmake 之后, 除了源文件 main.c 和 CMakeLists.txt 之外, 可以看到当前目录下生成了很多其它的文件或文件夹, 包括: CMakeCache.txt、CmakeFiles、cmake_install.cmake、Makefile, 重点是生成了这个 Makefile 文件, 有了 Makefile 之后, 接着我们使用 make 工具编译我们的工程, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake CMakeLists.txt main.c Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ make
Scanning dependencies of target hello
[ 50%] Building C object CMakeFiles/hello.dir/main.c.o
[100%] Linking C executable hello
[100%] Built target hello
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake CMakeLists.txt hello main.c Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.3.5 make 编译工程

通过 make 编译之后得到了一个可执行文件 hello, 这个名字是在 CMakeLists.txt 文件中指定的, 稍后向大家介绍。通过 file 命令可以查看到 hello 是一个 x86-64 架构下的可执行文件, 所以只能在我们的 Ubuntu PC 上运行:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
8885e4e29c746d, not stripped
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.3.6 file 查看可执行文件

为了验证 hello 可执行文件运行结果是否与源代码相同, 我们直接在 Ubuntu 下运行即可, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ./hello
Hello World!
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.3.7 执行 hello

CMakeLists.txt 文件

上面我们通过了一个非常简单例子向大家演示了如何使用 cmake, 重点在于去编写一个 CMakeLists.txt 文件, 现在来看看 CMakeLists.txt 文件中写的都是什么意思。

● 第一行 project(HELLO)

project 是一个命令, 命令的使用方式有点类似于 C 语言中的函数, 因为命令后面需要提供一对括号, 并且通常需要我们提供参数, 多个参数使用空格分隔而不是逗号“,”。

project 命令用于设置工程的名称, 括号中的参数 HELLO 便是我们要设置的工程名称; 设置工程名称并不是强制性的, 但是最好加上。

● 第二行 add_executable(hello ./main.c)

add_executable 同样也是一个命令, 用于生成一个可执行文件, 在本例中传入了两个参数, 第一个参数表示生成的可执行文件对应的文件名, 第二个参数表示对应的源文件; 所以 add_executable(hello ./main.c) 表示需要生成一个名为 hello 的可执行文件, 所需源文件为当前目录下的 main.c。

使用 out-of-source 方式构建

在上面的例子中, cmake 生成的文件以及最终的可执行文件 hello 与工程的源码文件 main.c 混在了一起, 这使得工程看起来非常乱, 当我们需要清理 cmake 产生的文件时将变得非常麻烦, 这不是我们想看到的; 我们需要将构建过程生成的文件与源文件分离开来, 不让它们混杂在一起, 也就是使用 out-of-source 方式构建。

将 cmake 编译生成的文件清理下, 然后在工程目录下创建一个 build 目录, 如下所示:

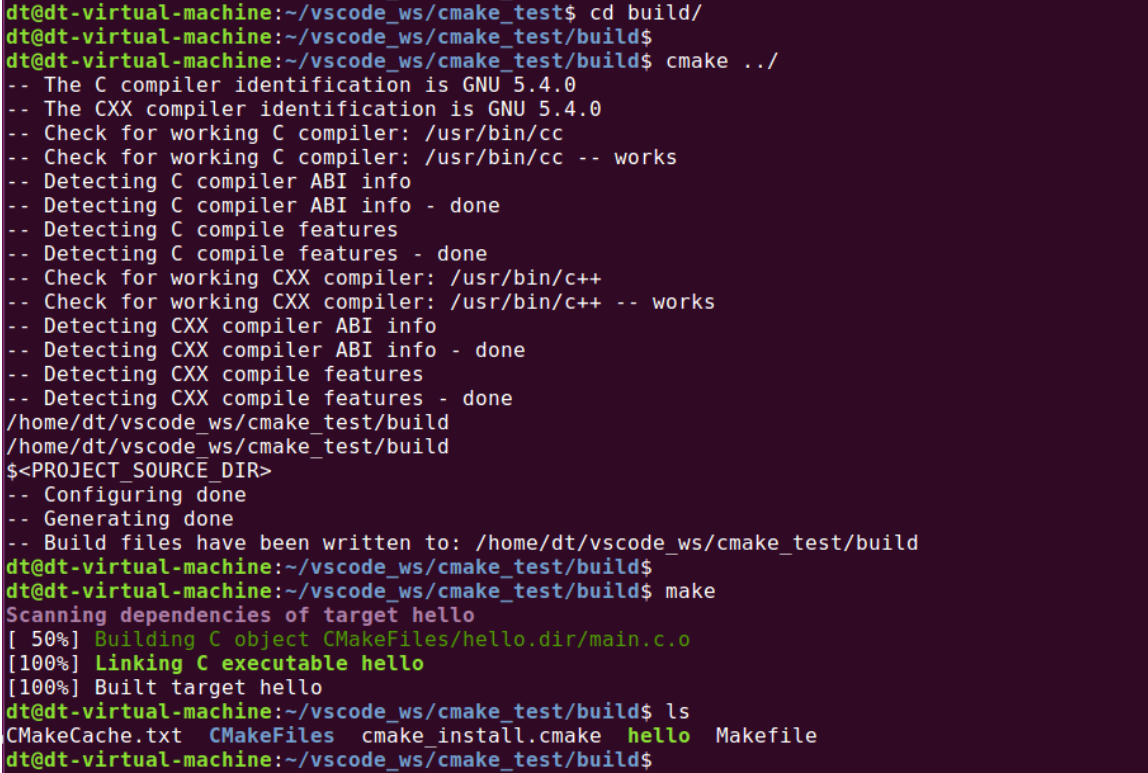
```
|— build
```



```
├── CMakeLists.txt
└── main.c
```

然后进入到 build 目录下执行 cmake:

```
cd build/
cmake ../
make
```



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ../
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
/home/dt/vscode_ws/cmake_test/build
/home/dt/vscode_ws/cmake_test/build
$<PROJECT_SOURCE_DIR>
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ make
Scanning dependencies of target hello
[ 50%] Building C object CMakeFiles/hello.dir/main.c.o
[100%] Linking C executable hello
[100%] Built target hello
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake hello Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.3.8 在 build 目录下编译

这样 cmake 生成的中间文件以及 make 编译生成的可执行文件就全部在 build 目录下了, 如果要清理工程, 直接删除 build 目录即可, 这样就方便多了。

32.3.2 示例二: 多个源文件

一个源文件的例子似乎没什么意思, 我们再加入一个 hello.h 头文件和 hello.c 源文件。在 hello.c 文件中定义了一个函数 hello, 然后在 main.c 源文件中将会调用该函数:

- hello.h 文件内容

```
#ifndef __TEST_HELLO_
#define __TEST_HELLO_

void hello(const char *name);

#endif // __TEST_HELLO_
```

- hello.c 文件内容

```
#include <stdio.h>
#include "hello.h"
```

```
void hello(const char *name)
{
    printf("Hello %s!\n", name);
}
```

- main.c 文件内容

```
#include "hello.h"
```

```
int main(void)
{
    hello("World");
    return 0;
}
```

- 然后准备好 CMakeLists.txt 文件

```
project(HELLO)
set(SRC_LIST main.c hello.c)
add_executable(hello ${SRC_LIST})
```

工程目录结构如下所示:

```
├── build    //文件夹
├── CMakeLists.txt
├── hello.c
├── hello.h
└── main.c
```

同样, 进入到 build 目录下, 执行 cmake、再执行 make 编译工程, 最终就会得到可执行文件 hello。

在本例子中, CMakeLists.txt 文件中使用到了 set 命令, set 命令用于设置变量, 如果变量不存在则创建该变量并设置它; 在本例中, 我们定义了一个 SRC_LIST 变量, SRC_LIST 变量是一个源文件列表, 记录生成可执行文件 hello 所需的源文件 main.c 和 hello.c, 而在 add_executable 命令引用了该变量; 当然我们也可以不去定义 SRC_LIST 变量, 直接将源文件列表写在 add_executable 命令中, 如下:

```
add_executable(hello main.c hello.c)
```

32.3.3 示例三: 生成库文件

在本例中, 除了生成可执行文件 hello 之外, 我们还需要将 hello.c 编译为静态库文件或者动态库文件, 在示例二的基础上对 CMakeLists.txt 文件进行修改, 如下所示:

```
project(HELLO)
add_library(libhello hello.c)
add_executable(hello main.c)
target_link_libraries(hello libhello)
```

进入到 build 目录下, 执行 cmake、再执行 make 编译工程, 编译完成之后, 在 build 目录下就会生成可执行文件 hello 和库文件, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  hello  liblibhello.a  Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.3.9 生成可执行文件和库文件

目录结构如下所示:

```

├── build
│   ├── hello
│   └── liblibhello.a
├── CMakeLists.txt
├── hello.c
├── hello.h
└── main.c

```

CMakeLists.txt 文件解释

本例中我们使用到了 `add_library` 命令和 `target_link_libraries` 命令。

`add_library` 命令用于生成库文件，在本例中我们传入了两个参数，第一个参数表示库文件的名称，需要注意的是，这个名字是不包含前缀和后缀的名称；在 Linux 系统中，库文件的前缀是 `lib`，动态库文件的后缀是 `.so`，而静态库文件的后缀是 `.a`；所以，意味着最终生成的库文件对应的名称会自动添加上前缀和后缀。

第二个参数表示库文件对应的源文件。

本例中，`add_library` 命令生成了一个静态库文件 `liblibhello.a`，如果要生成动态库文件，可以这样做：

```
add_library(libhello SHARED hello.c) #生成动态库文件
```

```
add_library(libhello STATIC hello.c) #生成静态库文件
```

`target_link_libraries` 命令为目标指定依赖库，在本例中，`hello.c` 被编译为库文件，并将其链接进 `hello` 程序。

修改生成的库文件名字

本例中有一点非常不爽，生成的库为 `liblibhello.a`，名字非常不好看；如果想生成 `libhello.a` 该怎么办？直接修改 `add_library` 命令的参数，像下面这样可以吗？

```
add_library(hello hello.c)
```

答案是不行的，因为 `hello` 这个目标已经存在了（`add_executable(hello main.c)`），目标名对于整个工程来说是唯一的，不可出现相同名称的目标，所以这种方法肯定是不行的，实际上我们只需要在 `CMakeLists.txt` 文件中添加下面这条命令即可：

```
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")
```

`set_target_properties` 用于设置目标的属性，这里通过 `set_target_properties` 命令对 `libhello` 目标的 `OUTPUT_NAME` 属性进行了设置，将其设置为 `hello`。

我们进行实验，此时 `CMakeLists.txt` 文件中的内容如下所示：

```
cmake_minimum_required(VERSION 3.5)
```

```
project(HELLO)
```

```
add_library(libhello SHARED hello.c)
```

```
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")
```

```
add_executable(hello main.c)
```

```
target_link_libraries(hello libhello)
```

除了添加 `set_target_properties` 命令之外，我们还加入了 `cmake_minimum_required` 命令，该命令用于设置当前工程的 `cmake` 最低版本号要求，当然这个并不是强制性的，但是最好还是加上。进入到 `build` 目录下，使用 `cmake+make` 编译整个工程，编译完成之后会发现，生成的库文件为 `libhello.a`，而不是 `liblibhello.a`。

```

├── build
│   ├── hello
│   └── libhello.so
├── CMakeLists.txt

```

```

├── hello.c
├── hello.h
└── main.c

```

32.3.4 示例四：将源文件组织到不同的目录

上面的示例中，我们已经加入了多个源文件，但是这些源文件都是放在同一个目录下，这样还是不太正规，我们应该将这些源文件按照类型、功能、模块给它们放置到不同的目录下，于是笔者将工程源码进行了整理，当前目录结构如下所示：

```

├── build          #build 目录
├── CMakeLists.txt
├── libhello
│   ├── CMakeLists.txt
│   ├── hello.c
│   └── hello.h
└── src
    ├── CMakeLists.txt
    └── main.c

```

在工程目录下，我们创建了 `src` 和 `libhello` 目录，并将 `hello.c` 和 `hello.h` 文件移动到 `libhello` 目录下，将 `main.c` 文件移动到 `src` 目录下，并且在顶层目录、`libhello` 目录以及 `src` 目录下都有一个 `CMakeLists.txt` 文件。`CMakeLists.txt` 文件的数量从 1 个一下变成了 3 个，顿时感觉到有点触不及防！还好每一个都不复杂！我们来看看每一个 `CMakeLists.txt` 文件的内容。

- 顶层 `CMakeLists.txt`

```

cmake_minimum_required(VERSION 3.5)
project(HELLO)
add_subdirectory(libhello)
add_subdirectory(src)

```

- `src` 目录下的 `CMakeLists.txt`

```

include_directories(${PROJECT_SOURCE_DIR}/libhello)
add_executable(hello main.c)
target_link_libraries(hello libhello)

```

- `libhello` 目录下的 `CMakeLists.txt`

```

add_library(libhello hello.c)
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")

```

顶层 `CMakeLists.txt` 中使用了 `add_subdirectory` 命令，该命令告诉 `cmake` 去子目录中寻找新的 `CMakeLists.txt` 文件并解析它；而在 `src` 的 `CMakeList.txt` 文件中，新增加了 `include_directories` 命令用来指明头文件所在的路径，并且使用到了 `PROJECT_SOURCE_DIR` 变量，该变量指向了一个路径，从命名上可知，该变量表示工程源码的目录。

和前面一样，进入到 `build` 目录下进行构建、编译，最终会得到可执行文件 `hello` (`build/src/hello`) 和库文件 `libhello.a` (`build/libhello/libhello.a`)：

```

├── build
│   ├── libhello
│   │   └── libhello.a
│   └── src

```

```

|       └── hello
├── CMakeLists.txt
├── libhello
|   ├── CMakeLists.txt
|   ├── hello.c
|   └── hello.h
└── src
    ├── CMakeLists.txt
    └── main.c

```

32.3.5 示例五：将生成的可执行文件和库文件放置到单独的目录下

前面还有一点不爽，在默认情况下，make 编译生成的可执行文件和库文件会与 cmake 命令产生的中间文件（CMakeCache.txt、CmakeFiles、cmake_install.cmake 以及 Makefile 等）混在一起，也就是它们在同一个目录下；如果我想让可执行文件单独放置在 bin 目录下，而库文件单独放置在 lib 目录下，就像下面这样：

```

├── build
|   ├── lib
|   |   └── libhello.a
|   └── bin
|       └── hello

```

将库文件存放在 build 目录下的 lib 目录中，而将可执行文件存放在 build 目录下的 bin 目录中，这个时候又该怎么做呢？这个时候我们可以通过两个变量来实现，将 src 目录下的 CMakeList.txt 文件进行修改，如下所示：

```

include_directories(${PROJECT_SOURCE_DIR}/libhello)
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
add_executable(hello main.c)
target_link_libraries(hello libhello)

```

然后再对 libhello 目录下的 CMakeList.txt 文件进行修改，如下所示：

```

set(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
add_library(libhello hello.c)
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")

```

修改完成之后，再次按照步骤对工程进行构建、编译，此时便会按照我们的要求将生成的可执行文件 hello 放置在 build/bin 目录下、库文件 libhello.a 放置在 build/lib 目录下。最终的目录结构就如下所示：

```

├── build
|   ├── bin
|   |   └── hello
|   └── lib
|       └── libhello.a
├── CMakeLists.txt
├── libhello
|   ├── CMakeLists.txt
|   ├── hello.c
|   └── hello.h
└── src

```

```
├── CMakeLists.txt
└── main.c
```

其实实现这个需求非常简单, 通过对 `LIBRARY_OUTPUT_PATH` 和 `EXECUTABLE_OUTPUT_PATH` 变量进行设置即可完成; `EXECUTABLE_OUTPUT_PATH` 变量控制可执行文件的输出路径, 而 `LIBRARY_OUTPUT_PATH` 变量控制库文件的输出路径。

32.4 CMakeLists.txt 语法规则

在上一小节中, 笔者通过几个简单地示例向大家演示了 `cmake` 的使用方法, 由此可知, `cmake` 的使用方法其实还是非常简单的, 重点在于编写 `CMakeLists.txt`, `CMakeLists.txt` 的语法规则也简单, 并没有 `Makefile` 的语法规则那么复杂难以理解! 本小节我们来学习 `CMakeLists.txt` 的语法规则。

32.4.1 简单地语法介绍

- 注释

在 `CMakeLists.txt` 文件中, 使用 “#” 号进行单行注释, 譬如:

```
#
# 这是注释信息
#
cmake_minimum_required(VERSION 3.5)
project(HELLO)
```

大多数脚本语言都是使用 “#” 号进行注释。

- 命令 (command)

通常在 `CMakeLists.txt` 文件中, 使用最多的是命令, 譬如上例中的 `cmake_minimum_required`、`project` 都是命令; 命令的使用方式有点类似于 C 语言中的函数, 因为命令后面需要提供一对括号, 并且通常需要我们提供参数, 多个参数使用空格分隔而不是逗号 “,”, 这是与函数不同的地方。命令的语法格式如下所示:

```
command(参数 1 参数 2 参数 3 ...)
```

不同的命令所需的参数不同, 需要注意的是, 参数可以分为必要参数和可选参数 (通常称为选项), 很多命令都提供了这两类参数, 必要参数使用 `<参数>` 表示, 而可选参数使用 `[参数]` 表示, 譬如 `set` 命令:

```
set(<variable> <value>... [PARENT_SCOPE])
```

`set` 命令用于设置变量, 第一个参数 `<variable>` 和第二个参数 `<value>` 是必要参数, 在参数列表 (... 表示参数个数没有限制) 的最后可以添加一个可选参数 `PARENT_SCOPE` (`PARENT_SCOPE` 选项), 既然是可选的, 那就不是必须的, 根据实际使用情况确定是否需要添加。

在 `CMakeLists.txt` 中, 命令名不区分大小写, 可以使用大写字母或小写字母书写命令名, 譬如:

```
project(HELLO) #小写
PROJECT(HELLO) #大写
```

这俩的效果是相同的, 指定的是同一个命令, 并没区别; 这个主要看个人喜好, 个人喜欢用小写字母, 主要是为了和变量区分开来, 因为 `cmake` 的内置变量其名称都是使用大写字母组成的。

- 变量 (variable)

在 `CMakeLists.txt` 文件中可以使用变量, 使用 `set` 命令可以对变量进行设置, 譬如:

```
# 设置变量 MY_VAL
set(MY_VAL "Hello World!")
```

上例中, 通过 `set` 命令对变量 `MY_VAL` 进行设置, 将其内容设置为 "Hello World!"; 那如何引用这个变量呢? 这与 `Makefile` 是相同的, 通过 `${MY_VAL}` 方式来引用变量, 如下所示:

```
#设置变量 MY_VAL
set(MY_VAL "Hello World!")

#引用变量 MY_VAL
message(${MY_VAL})
```

变量可以分为 `cmake` 内置变量以及自定义变量, 譬如上例中所定义的 `MY_VAL` 就是一个自定义变量; 譬如在 32.3.5 小节中所使用的 `LIBRARY_OUTPUT_PATH` 和 `EXECUTABLE_OUTPUT_PATH` 变量则是 `cmake` 的内置变量, 每一个内置变量都有自己的含义, 像这样的内置变量还有很多, 稍后向大家介绍。

32.4.2 部分常用命令

`cmake` 提供了很多命令, 每一个命令都有它自己的功能、作用, 通过这个链接地址 <https://cmake.org/cmake/help/v3.5/manual/cmake-commands.7.html> 可以查询到所有的命令及其相应的介绍、使用方法等等, 如下所示:

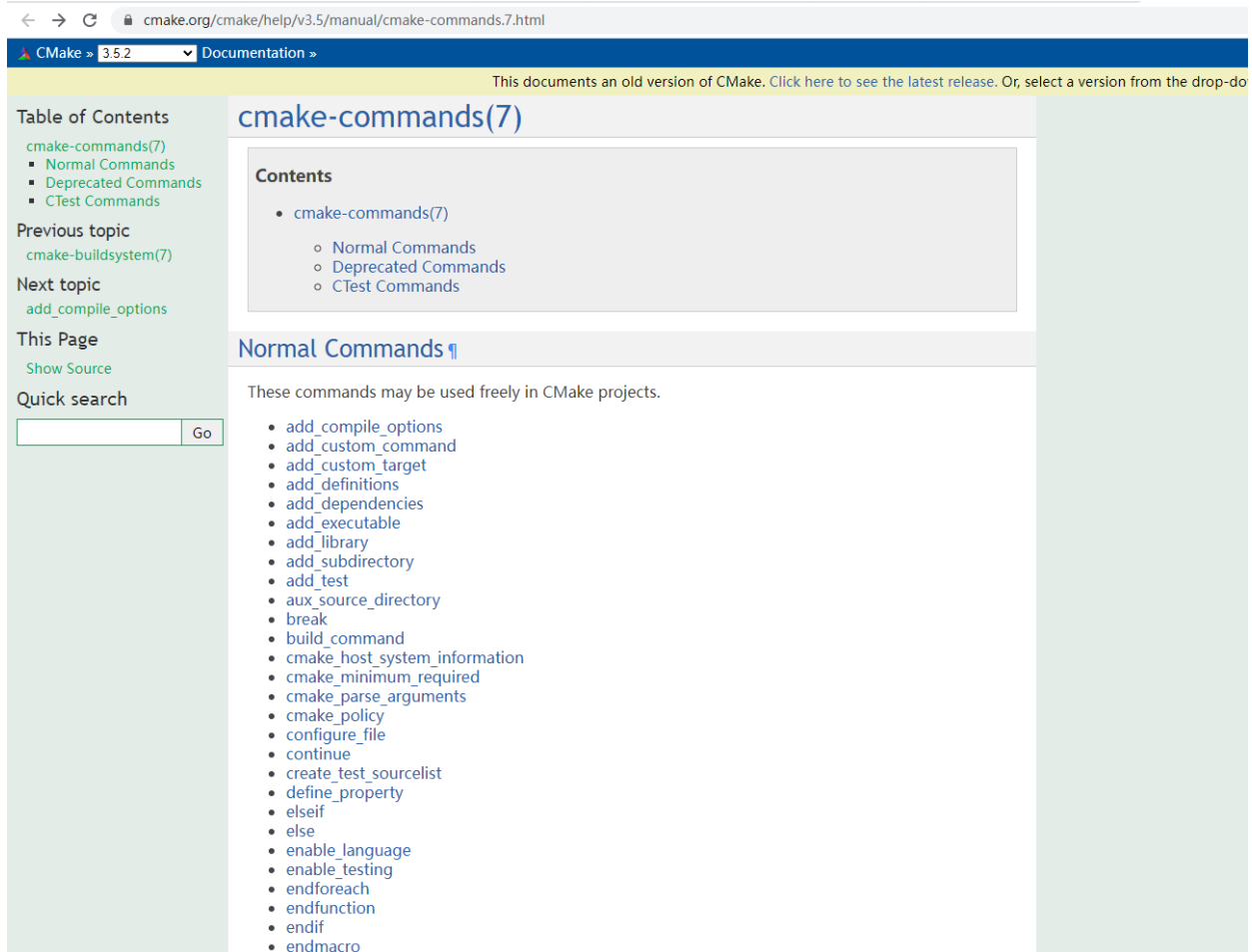


图 32.4.1 cmake 命令查询

大家可以把这个链接地址保存起来, 可以把它当成字典的形式在有需要进行查询, 由于命令非常多, 笔者不可能将所有命令都给大家介绍一遍, 这里给大家介绍一些基本的命令, 如下表所示:

command	说明
<code>add_executable</code>	可执行程序目标

add_library	库文件目标
add_subdirectory	去指定目录中寻找新的 CMakeLists.txt 文件
aux_source_directory	收集目录中的文件名并赋值给变量
cmake_minimum_required	设置 cmake 的最低版本号要求
get_target_property	获取目标的属性
include_directories	设置所有目标头文件的搜索路径, 相当于 gcc 的-I 选项
link_directories	设置所有目标库文件的搜索路径, 相当于 gcc 的-L 选项
link_libraries	设置所有目标需要链接的库
list	列表相关的操作
message	用于打印、输出信息
project	设置工程名字
set	设置变量
set_target_properties	设置目标属性
target_include_directories	设置指定目标头文件的搜索路径
target_link_libraries	设置指定目标库文件的搜索路径
target_sources	设置指定目标所需的源文件

表 32.4.1 常用基本的命令

接下来详细地给大家介绍每一个命令。

● add_executable

add_executable 命令用于添加一个可执行程序目标, 并设置目标所需的源文件, 该命令定义如下所示:

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] source1 [source2 ...])
```

该命令提供了一些可选参数, 这些可选参数的含义笔者就不多说了, 通常不需要加入, 具体的含义大家可以自己查看 cmake 官方文档 (https://cmake.org/cmake/help/v3.5/command/add_executable.html); 只需传入目标名和对应的源文件即可, 譬如:

```
#生成可执行文件 hello
```

```
add_executable(hello 1.c 2.c 3.c)
```

定义了一个可执行程序目标 hello, 生成该目标文件所需的源文件为 1.c、2.c 和 3.c。需要注意的是, 源文件路径既可以使用相对路径、也可以使用绝对路径, 相对路径被解释为相对于当前源码路径(注意, 这里源码指的是 CMakeLists.txt 文件, 因为 CMakeLists.txt 被称为 cmake 的源码, 若无特别说明, 后续将沿用这个概念!)。

● add_library

add_library 命令用于添加一个库文件目标, 并设置目标所需的源文件, 该命令定义如下所示:

```
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            source1 [source2 ...])
```

第一个参数 name 指定目标的名字, 参数 source1...source2 对应源文件列表; add_library 命令默认生成的库文件是静态库文件, 通过 SHARED 选项可使其生成动态库文件, 具体的使用方法如下:

```
#生成静态库文件 libmylib.a
```

```
add_library(mylib STATIC 1.c 2.c 3.c)
```

```
#生成动态库文件 libmylib.so
```

```
add_library(mylib SHARED 1.c 2.c 3.c)
```


与 `add_executable` 命令相同, `add_library` 命令中源文件既可以使用相对路径指定、也可以使用绝对路径指定, 相对路径被解释为相对于当前源码路径。

不管是 `add_executable`、还是 `add_library`, 它们所定义的目标名在整个工程中必须是唯一的, 不可出现两个目标名相同的目标。

● `add_subdirectory`

`add_subdirectory` 命令告诉 `cmake` 去指定的目录中寻找源码并执行它, 有点像 `Makefile` 的 `include`, 其定义如下所示:

```
add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
```

参数 `source_dir` 指定一个目录, 告诉 `cmake` 去该目录下寻找 `CMakeLists.txt` 文件并执行它; 参数 `binary_dir` 指定了一个路径, 该路径作为子源码 (调用 `add_subdirectory` 命令的源码称为当前源码或父源码, 被执行的源码称为子源码) 的输出文件 (`cmake` 命令所产生的中间文件) 目录, `binary_dir` 参数是一个可选参数, 如果没有显式指定, 则会使用一个默认的输出文件目录; 为了后续便于表述, 我们将输出文件目录称为 `BINARY_DIR`。

譬如工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
└── src
    ├── CMakeLists.txt
    └── main.c
```

顶层 `CMakeLists.txt` 文件内容如下所示:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

# 告诉 cmake 去 src 目录下寻找 CMakeLists.txt
add_subdirectory(src)
```

`src` 目录下的 `CMakeLists.txt` 文件:

```
# src 下的 CMakeLists.txt
add_executable(hello main.c)
```

进入到 `build` 目录下, 执行 `cmake`、`make` 进行构建编译; 在本例中, 顶层源码对应的输出文件会存放在 `build` 目录, 也就是执行 `cmake` 命令所在目录; 子源码 (`src` 目录下的 `CMakeLists.txt`) 对应的输出文件会存放在 `build/src` 目录, 包括生成的可执行文件默认会与这些中间文件放置在同一个目录, 如下所示:

```
├── build
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   ├── cmake_install.cmake
│   ├── Makefile
│   └── src
│       ├── CMakeFiles
│       ├── cmake_install.cmake
│       ├── hello
│       └── Makefile
└── CMakeLists.txt
```

```
└── src
    ├── CMakeLists.txt
    └── main.c
```

所以由此可知,当前源码调用 `add_subdirectory` 命令执行子源码时,若没有为子源码指定 `BINARY_DIR`,默认情况下,会在当前源码的 `BINARY_DIR` 中创建与子目录(子源码所在目录)同名的文件夹,将其作为子源码的 `BINARY_DIR`。

接下来我们修改顶层 `CMakeCache.txt` 文件:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

# 告诉 cmake 去 src 目录下寻找 CMakeLists.txt
add_subdirectory(src output)
```

指定子源码的 `BINARY_DIR` 为 `output`,这里使用的是相对路径方式, `add_subdirectory` 命令对于相对路径的解释为:相对于当前源码的 `BINARY_DIR`;修改完成之后,再次进入到 `build` 目录下执行 `cmake`、`make` 命令进行构建、编译,此时会在 `build` 目录下生成一个 `output` 目录,这就是子源码的 `BINARY_DIR`。

设置 `BINARY_DIR` 可以使用相对路径、也可以是绝对路径,相对路径则是相对于当前源码的 `BINARY_DIR`,并不是当前源码路径,这个要理解。

通过 `add_subdirectory` 命令加载、执行一个外部文件夹中的源码,既可以是当前源码路径的子目录、也可以是与当前源码路径平级的目录亦或者是当前源码路径上级目录等等;对于当前源码路径的子目录,不强制调用者显式指定子源码的 `BINARY_DIR`;如果不是当前源码路径的子目录,则需要调用者显式指定 `BINARY_DIR`,否则执行源码时会报错。接下来进行测试,譬如工程目录结构如下所示:

```
└── build
└── CMakeLists.txt
└── lib
    └── CMakeLists.txt
└── src
    ├── CMakeLists.txt
    └── main.c
```

这里一共有 3 个 `CMakeLists.txt` 文件, `lib` 目录和 `src` 目录是平级关系,顶层 `CMakeLists.txt` 内容如下:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

# 加载 src 目录下的源码
add_subdirectory(src)
    src 目录下的 CMakeLists.txt:

# src 目录 CMakeLists.txt
add_executable(hello main.c)

# 加载平级目录 lib 中的源码
add_subdirectory(..lib)
```

此时调用 `add_subdirectory` 加载 `lib` 目录的源码时并为显式指定 `BINARY_DIR`, 我们看看会怎么样, 进入到 `build` 目录下, 执行 `cmake` 命令, 如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Error at src/CMakeLists.txt:2 (add_subdirectory):
add_subdirectory not given a binary directory but the given source
directory "/home/dt/vscode_ws/cmake_test/lib" is not a subdirectory of
"/home/dt/vscode_ws/cmake_test/src".  When specifying an out-of-tree source
a binary directory must be explicitly specified.

-- Configuring incomplete, errors occurred!
See also "/home/dt/vscode_ws/cmake_test/build/CMakeFiles/CMakeOutput.log".
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.2 cmake 打印信息

果不其然确实发生了报错, 而且提示我们 `add_subdirectory` 命令必须要指定 `BINARY_DIR`, 那我们将 `src` 目录下的 `CMakeLists.txt` 进行修改, 显式指定 `BINARY_DIR`, 如下所示:

```
# src 目录 CMakeLists.txt
add_executable(hello main.c)

# 加载同级目录 lib 中的源码
add_subdirectory(..lib output)
```

接着再次执行 `cmake` (每次执行 `cmake` 前进行清理, 将 `build` 目录下生成的所有文件全部删除):

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
hehehehe
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.3 cmake 打印信息

可以看到这次执行 `cmake` 没有报错打印了。

- `aux_source_directory`

`aux_source_directory` 命令会查找目录中的所有源文件, 其命令定义如下:

```
aux_source_directory(<dir> <variable>)
```

从指定的目录中查找所有源文件，并将扫描到的源文件路径信息存放到<variable>变量中，譬如目录结构如下：

```

├── build
├── CMakeLists.txt
└── src
    ├── 1.c
    ├── 2.c
    ├── 2.cpp
    └── main.c

```

CMakeCache.txt 内容如下所示：

```

# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

# 查找 src 目录下的所有源文件
aux_source_directory(src SRC_LIST)
message("${SRC_LIST}") # 打印 SRC_LIST 变量

```

进入到 build 目录下，执行 cmake ..命令，打印信息如下所示：

```

dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
src/1.c;src/2.c;src/main.c;src/2.cpp → message的打印信息
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$

```

图 32.4.4 cmake 命令打印信息

由此可见，aux_source_directory 会将扫描到的每一个源文件添加到 SRC_LIST 变量中，组成一个字符串列表，使用分号“;”分隔。

同理，aux_source_directory 既可以使用相对路径，也可以使用绝对路径，相对路径是相对于当前源码路径。

- **get_target_property 和 set_target_properties**

分别用于获取/设置目标的属性，这个后面再给大家进行专题介绍。

- **include_directories**

include_directories 命令用于设置头文件的搜索路径，相当于 gcc 的-I 选项，其定义如下所示：

```
include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])
```

默认情况下会将指定目录添加到头文件搜索列表(可以认为每一个 CMakeLists.txt 源码都有自己的头文件搜索列表)的最后面, 可以通过设置 `CMAKE_INCLUDE_DIRECTORIES_BEFORE` 变量为 `ON` 来改变它默认行为, 将目录添加到列表前面。也可以在每次调用 `include_directories` 命令时使用 `AFTER` 或 `BEFORE` 选项来指定是添加到列表的前面或者后面。如果使用 `SYSTEM` 选项, 会把指定目录当成系统的搜索目录。既可以使用绝对路径来指定头文件搜索目录、也可以使用相对路径来指定, 相对路径被解释为当前源码路径的相对路径。

譬如工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
├── include
│   └── hello.h
└── main.c
```

源文件 `main.c` 中使用了 `include` 目录下的头文件 `hello.h`, `CMakeLists.txt` 内容如下:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

include_directories(include)
add_executable(hello main.c)
```

使用 `include_directories` 命令将当前目录下的 `include` 文件夹添加到头文件搜索列表中, 进入 `build` 目录下, 执行 `cmake`、`make` 进行构建、编译, 编译过程是没有问题的, 不会报错提示头文件找不到; 但如果去掉 `include_directories(include)` 这条命令, 编译肯定会报错, 大家可以动手试试!

默认情况下, `include` 目录被添加到头文件搜索列表的最后面, 通过 `AFTER` 或 `BEFORE` 选项可显式指定添加到列表后面或前面:

```
# 添加到列表后面
include_directories(AFTER include)

# 添加到列表前面
include_directories(BEFORE include)
```

当调用 `add_subdirectory` 命令加载子源码时, 会将 `include_directories` 命令包含的目录列表向下传递给子源码(子源码从父源码中继承过来), 我们测试下, 譬如工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
├── include
│   └── hello.h
└── src
    ├── CMakeLists.txt
    └── main.c
```

`src` 目录下 `main.c` 源文件中使用了 `hello.h` 头文件, 顶层 `CMakeLists.txt` 内容如下所示:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")
```

```
include_directories(include)
```

```
add_subdirectory(src)
```

顶层 CMakeLists.txt 源码中调用了 include_directories 将 include 目录添加到当前源码的头文件搜索列表中, 接着调用 add_subdirectory 命令加载、执行子源码; src 目录下 CMakeLists.txt 内容如下所示:

```
# src 目录 CMakeLists.txt
```

```
add_executable(hello main.c)
```

进入到 build 目录, 进行构建、编译, 整个编译过程是没有问题的。

● link_directories 和 link_libraries

link_directories 命令用于设置库文件的搜索路径, 相当于 gcc 编译器的 -L 选项; link_libraries 命令用于设置需要链接的库文件, 相当于 gcc 编译器的 -l 选项; 命令定义如下所示:

```
link_directories(directory1 directory2 ...)
```

```
link_libraries([item1 [item2 [...]]])
```

```
[[debug|optimized|general] <item>] ...)
```

link_directories 会将指定目录添加到**库文件搜索列表** (可以认为每一个 CMakeLists.txt 源码都有自己的库文件搜索列表) 中; 同理, link_libraries 命令会将指定库文件添加到链接库列表。link_directories 命令可以使用绝对路径或相对路径指定目录, 相对路径被解释为当前源码路径的相对路径。

譬如工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
├── include
│   └── hello.h
├── lib
│   └── libhello.so
└── main.c
```

在 lib 目录下有一个动态库文件 libhello.so, 编译链接 main.c 源文件时需要链接 libhello.so; CMakeLists.txt 文件内容如下所示:

```
# 顶层 CMakeLists.txt
```

```
cmake_minimum_required("VERSION" "3.5")
```

```
project("HELLO")
```

```
include_directories(include)
```

```
link_directories(lib)
```

```
link_libraries(hello)
```

```
add_executable(main main.c)
```

库文件名既可以使用简写, 也可以使用库文件名的全称, 譬如:

```
# 简写
```

```
link_libraries(hello)
```

```
# 全称
```

```
link_libraries(libhello.so)
```

link_libraries 命令也可以指定库文件的全路径 (绝对路径 / 开头), 如果不是 / 开头, link_libraries 会认为调用者传入的是库文件名, 而非库文件全路径, 譬如上述 CMakeLists.txt 可以修改为下面这种方式:

```
# 顶层 CMakeLists.txt
```

```
cmake_minimum_required("VERSION" "3.5")
project("HELLO")
```

```
include_directories(include)
link_libraries(${PROJECT_SOURCE_DIR}/lib/libhello.so)
add_executable(main main.c)
```

与 `include_directories` 命令相同, 当调用 `add_subdirectory` 命令加载子源码时, 会将 `link_directories` 命令包含的目录列表以及 `link_libraries` 命令包含的链接库列表向下传递给子源码(子源码从父源码中继承过来)。这里不再演示了, 大家可以自己测试下。

● list

`list` 命令是一个关于列表操作的命令, 譬如获取列表的长度、从列表中返回由索引值指定的元素、将元素追加到列表中等等。命令定义如下:

```
list(LENGTH <list> <output variable>)
list(GET <list> <element index> [<element index> ...]
    <output variable>)
list(APPEND <list> [<element> ...])
list(FIND <list> <value> <output variable>)
list(INSERT <list> <element_index> <element> [<element> ...])
list(REMOVE_ITEM <list> <value> [<value> ...])
list(REMOVE_AT <list> <index> [<index> ...])
list(REMOVE_DUPLICATES <list>)
list(REVERSE <list>)
list(SORT <list>)
```

列表这个概念还没给大家介绍, 列表其实就是字符串数组(或者叫字符串列表、字符串数组), 稍后再向大家说明。

`LENGTH` 选项用于返回列表长度;

`GET` 选项从列表中返回由索引值指定的元素;

`APPEND` 选项将元素追加到列表后面;

`FIND` 选项将返回列表中指定元素的索引值, 如果未找到, 则返回-1。

`INSERT` 选项将向列表中的指定位置插入元素。

`REMOVE_AT` 和 `REMOVE_ITEM` 选项将从列表中删除元素, 不同之处在于 `REMOVE_ITEM` 将删除给定的元素, 而 `REMOVE_AT` 将删除给定索引值的元素。

`REMOVE_DUPLICATES` 选项将删除列表中的重复元素。

`REVERSE` 选项就地反转列表的内容。

`SORT` 选项按字母顺序对列表进行排序。

● message

`message` 命令用于打印、输出信息, 类似于 Linux 的 `echo` 命令, 命令定义如下所示:

```
message([<mode>] "message to display" ...)
```

可选的 `mode` 关键字用于确定消息的类型, 如下:

mode	说明
none (无)	重要信息、普通信息
STATUS	附带信息

WARNING	CMake 警告, 继续处理
AUTHOR_WARNING	CMake 警告(开发), 继续处理
SEND_ERROR	CMake 错误, 继续处理, 但跳过生成
FATAL_ERROR	CMake 错误, 停止处理和生成
DEPRECATION	如果变量 CMAKE_ERROR_DEPRECATED 或 CMAKE_WARN_DEPRECATED 分别启用, 则 CMake 弃用错误或警告, 否则没有消息。

表 32.4.2 message 命令 mode 关键字说明

所以可以使用这个命令作为 CMakeLists.txt 源码中的输出打印语句, 譬如:

```
# 打印"Hello World"
message("Hello World!")
```

- **project**

project 命令用于设置工程名称:

```
# 设置工程名称为 HELLO
project(HELLO)
```

执行这个之后会引入两个变量: HELLO_SOURCE_DIR 和 HELLO_BINARY_DIR, 注意这两个变量名的前缀就是工程名称, HELLO_SOURCE_DIR 变量指的是 HELLO 工程源码目录、HELLO_BINARY_DIR 变量指的是 HELLO 工程源码的输出文件目录; 我们可以使用 message 命令打印变量, 譬如 CMakeLists.txt 内容如下所示:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

message(${HELLO_SOURCE_DIR})
message(${HELLO_BINARY_DIR})
```

进入 build 目录下, 执行 cmake:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
/home/dt/vscode_ws/cmake_test HELLO_BINARY_DIR
/home/dt/vscode_ws/cmake_test/build HELLO_BINARY_DIR
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.5 cmake 命令打印信息

但如果不加入 `project(HELLO)` 命令, 这两个变量是不存在的; 工程源码目录指的是顶层源码所在目录, `cmake` 定义了两个等价的变量 `PROJECT_SOURCE_DIR` 和 `PROJECT_BINARY_DIR`, 通常在 `CMakeLists.txt` 源码中都会使用这两个等价的变量。

通常只需要在顶层 `CMakeLists.txt` 源码中调用 `project` 即可!

- **set**

`set` 命令用于设置变量, 命令定义如下所示:

```
set(<variable> <value>... [PARENT_SCOPE])
```

设置变量的值, 可选参数 `PARENT_SCOPE` 影响变量的作用域, 这个我们稍后再说。

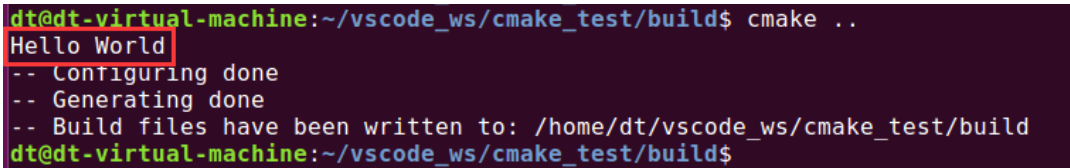
譬如 `CMakeLists.txt` 源码内容如下所示:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

# set 命令
set(VAR1 Hello)      #设置变量 VAR1=Hello
set(VAR2 World)     #设置变量 VAR2=World

# 打印变量
message("${VAR1}" " ${VAR2})
```

对应的打印信息:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello World
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.6 cmake 打印信息

字符串列表

通过 `set` 命令实现字符串列表, 如下所示:

```
# 字符串列表
set(SRC_LIST 1.c 2.c 3.c 4.c 5.c)
```

此时 `SRC_LIST` 就是一个列表, 它包含了 5 个元素 (1.c、2.c、3.c、4.c、5.c), 列表的各个元素使用分号 “;” 分隔, 如下:

```
SRC_LIST = 1.c;2.c;3.c;4.c;5.c      #列表
```

我们来测试一下, 譬如 `CMakeLists.txt` 源码内容如下所示:

```
# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project("HELLO")

# set 命令
set(SRC_LIST 1.c 2.c 3.c 4.c 5.c)

# 打印变量
message("${SRC_LIST}")
```

执行 cmake 命令打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
1.c2.c3.c4.c5.c
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.7 cmake 打印信息

乍一看这个打印信息你是不是觉得 SRC_LIST 就是一个普通的变量 (SRC_LIST=1.c2.c3.c4.c5.c), 并不是列表呢? 事实并非如此, 我们可以修改 message 命令, 将 \${SRC_LIST} 放置在双引号中, 如下:

打印变量

```
message("${SRC_LIST}")
```

再次执行 cmake, 打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
1.c;2.c;3.c;4.c;5.c
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.8 cmake 打印信息

可以看到此时打印出来的确实是一个列表, 为何加了双引号就会这样呢? 关于双引号的作用请看 32.4.4 小节。既然是列表, 那自然可以使用 list 命令对列表进行相关的操作:

顶层 CMakeLists.txt

```
cmake_minimum_required("VERSION" "3.5")
project("HELLO")
```

列表

```
set(SRC_LIST main.c world.c hello.c)
message("SRC_LIST: ${SRC_LIST}")
```

#列表操作

```
list(LENGTH SRC_LIST L_LEN)
message("列表长度: ${L_LEN}")
```

```
list(GET SRC_LIST 1 VAR1)
message("获取列表中 index=1 的元素: ${VAR1}")
```

```
list(APPEND SRC_LIST hello_world.c) #追加元素
message("SRC_LIST: ${SRC_LIST}")
```

```
list(SORT SRC_LIST) #排序
message("SRC_LIST: ${SRC_LIST}")
```

cmake 打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapters$ cmake .
SRC_LIST: main.c;world.c;hello.c
列表长度: 3
获取列表中index=1的元素: world.c
SRC_LIST: main.c;world.c;hello.c;hello_world.c
SRC_LIST: hello.c;hello_world.c;main.c;world.c
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapters$
```

图 32.4.9 cmake 打印信息

除此之外, 在 cmake 中可以使用循环语句依次读取列表中的各个元素, 后续再向大家介绍。

● target_include_directories 和 target_link_libraries

target_include_directories 命令为指定目标设置头文件搜索路径, 而 target_link_libraries 命令为指定目标设置链接库文件, 这听起来跟 include_directories 和 link_libraries 命令有着相同的作用, 确实如此, 它们的功能的确相同, 但是在一些细节方面却有不同, 关于它们之间的区别稍后再给大家进行解释!

target_include_directories 和 target_link_libraries 命令定义如下所示:

```
target_include_directories(<target> [SYSTEM] [BEFORE]
    <INTERFACE|PUBLIC|PRIVATE> [items1...]
    [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])

target_link_libraries(<target>
    <PRIVATE|PUBLIC|INTERFACE> <item>...
    [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

这两命令都有一个相同的参数<target>目标, 这个目标指的就是譬如 add_executable、add_library 命令所创建的目标。首先对于 target_include_directories 命令来说, SYSTEM、BEFORE 这两个选项与 include_directories 命令中 SYSTEM、BEFORE 选项的意义相同, 这里不再多说!

我们重点关注的是 INTERFACE|PUBLIC|PRIVATE 这三个选项有何不同? 通过一个示例向大家说明, 譬如工程目录结构如下所示:

```
├── build                //build 目录
├── CMakeLists.txt
├── hello_world         //生成 libhello_world.so,调用 libhello.so 和 libworld.so
│   ├── CMakeLists.txt
│   ├── hello          //生成 libhello.so
│   │   ├── CMakeLists.txt
│   │   ├── hello.c
│   │   └── hello.h    //libhello.so 对外头文件
│   ├── hello_world.c
│   ├── hello_world.h //libhello_world.so 对外头文件
│   └── world          //生成 libworld.so
│       ├── CMakeLists.txt
│       ├── world.c
│       └── world.h    //libworld.so 对外头文件
└── main.c
```

调用关系:



根据以上工程，我们对 INTERFACE、PUBLIC、PRIVATE 三个关键字进行说明：

PRIVATE: 私有的。main.c 程序调用了 libhello_world.so，生成 libhello_world.so 时，只在 hello_world.c 中包含了 hello.h，libhello_world.so 对外的头文件——hello_world.h 中不包含 hello.h。而且 main.c 不会调用 hello.c 中的函数，或者说 main.c 不知道 hello.c 的存在，它只知道 libhello_world.so 的存在；那么在 hello_world/CMakeLists.txt 中应该写入：

```
target_link_libraries(hello_world PRIVATE hello)
target_include_directories(hello_world PRIVATE hello)
```

INTERFACE: 接口。生成 libhello_world.so 时，只在 libhello_world.so 对外的头文件——hello_world.h 中包含了 hello.h，hello_world.c 中不包含 hello.h，即 libhello_world.so 不使用 libhello.so 提供的功能，但是 main.c 需要使用 libhello.so 中的功能。那么在 hello_world/CMakeLists.txt 中应该写入：

```
target_link_libraries(hello-world INTERFACE hello)
target_include_directories(hello-world INTERFACE hello)
```

PUBLIC: 公开的。PUBLIC = PRIVATE + INTERFACE。生成 libhello_world.so 时，在 hello_world.c 和 hello_world.h 中都包含了 hello.h。并且 main.c 中也需要使用 libhello.so 提供的功能。那么在 hello_world/CMakeLists.txt 中应该写入：

```
target_link_libraries(hello-world PUBLIC hello)
target_include_directories(hello-world PUBLIC hello)
```

不知道大家看懂了没有，其实理解起来很简单，对于 target_include_directories 来说，这些关键字用于指示何时需要传递给目标的包含目录列表，指定了包含目录列表的使用范围（scope）：

- 当使用 PRIVATE 关键字修饰时，意味着包含目录列表仅用于当前目标；
- 当使用 INTERFACE 关键字修饰时，意味着包含目录列表不用于当前目标、只能用于依赖该目标的其它目标，也就是说 cmake 会将包含目录列表传递给当前目标的依赖目标；
- 当使用 PUBLIC 关键字修饰时，这就是以上两个的集合，包含目录列表既用于当前目标、也会传递给当前目标的依赖目标。

对于 target_link_libraries 亦是如此，只不过包含目录列表换成了链接库列表。譬如：

target_link_libraries(hello_world INTERFACE hello)：表示目标 hello_world 不需要链接 hello 库，但是对于 hello_world 目标的依赖目标（依赖于 hello_world 的目标）它们需要链接 hello 库。

以上便是笔者对 INTERFACE、PUBLIC、PRIVATE 这三个关键字的概括性理解，所以整出这几个关键字主要还是为了控制包含目录列表或链接库列表的使用范围，这就是 target_include_directories、target_link_libraries 命令与 include_directories、link_libraries 命令的不同之处。target_include_directories()、target_link_libraries() 的功能完全可以使用 include_directories()、link_libraries() 来实现。但是笔者强烈建议大家使用 target_include_directories() 和 target_link_libraries()。为什么？保持清晰！

include_directories()、link_libraries() 是针对当前源码中的所有目标，并且还会向下传递（譬如通过 add_subdirectory 加载子源码时，也会将其传递给子源码）。在一个大的工程当中，这通常不规范、有时还会编译出现错误、混乱，所以我们应尽量使用 target_include_directories() 和 target_link_libraries()，保持整个工程的目录清晰。

总结

本节内容到此结束了，给大家介绍了一些基本、常用的命令，并进行了详细的解释说明，除此之外，还有很多的命令并未提及，我们会在后面进行专题的介绍，大家要自己多动手、多多练习，这样才能越来越熟练！

32.4.3 部分常用变量

变量也是 `cmake` 中的一个重头戏, `cmake` 提供了很多内置变量, 每一个变量都有它自己的含义, 通过这个链接地址 <https://cmake.org/cmake/help/v3.5/manual/cmake-variables.7.html> 可以查询到所有的内置变量及其相应的介绍, 如下所示:

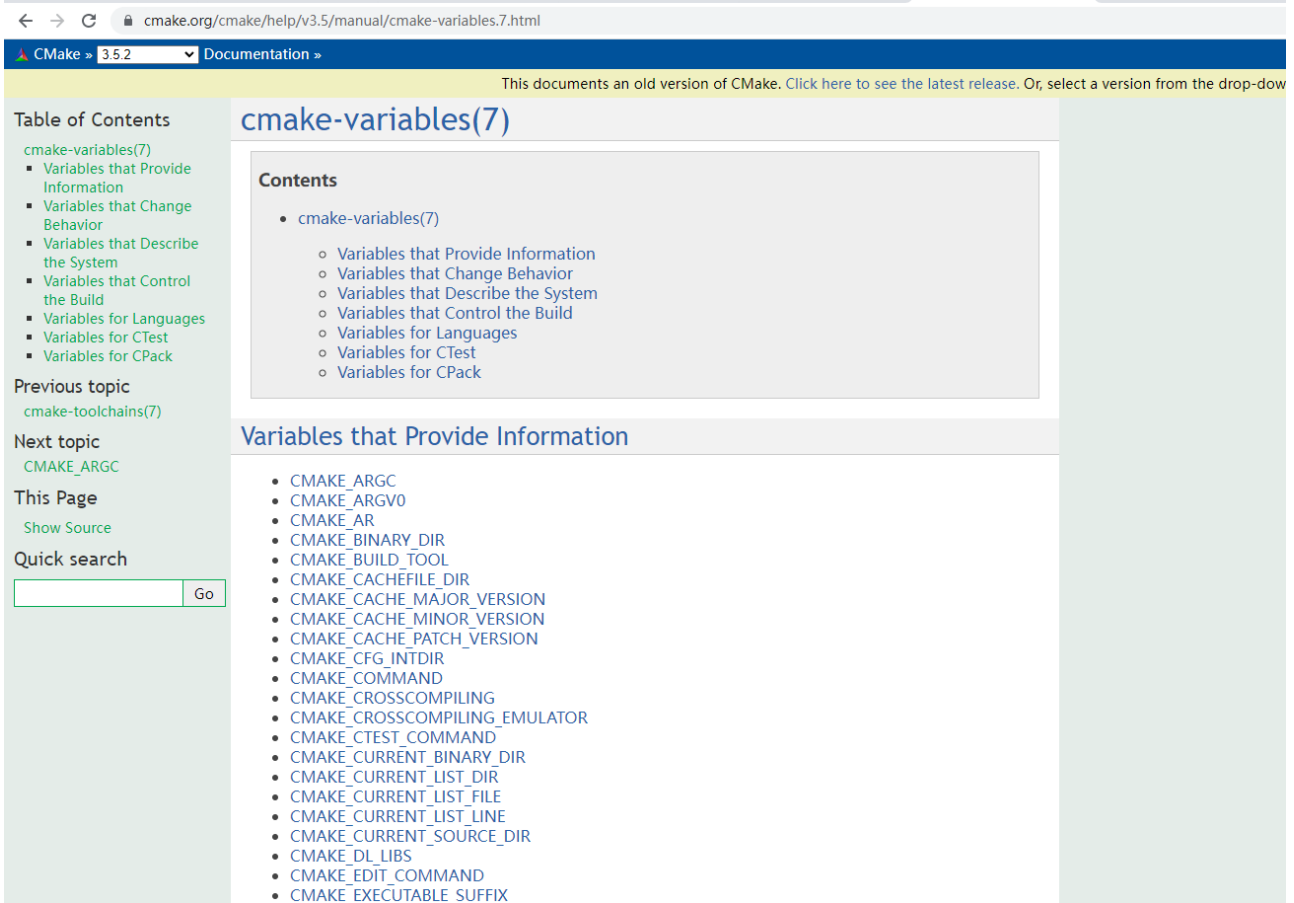


图 32.4.10 cmake 内置变量

在这一份文档中, 对变量进行分类, 分为: 提供信息的变量、改变行为的变量、描述系统的变量、控制编译的变量等等, 笔者也按照这个分类给大家介绍一些基本、常用的变量。

- 提供信息的变量

顾名思义, 这种变量可以提供某种信息, 既然如此, 那么我们通常只需要读取变量即可, 而不需要对变量进行修改:

变量	说明
<code>PROJECT_SOURCE_DIR</code>	工程顶层目录, 也就是顶层 <code>CMakeLists.txt</code> 源码所在目录
<code>PROJECT_BINARY_DIR</code>	工程 <code>BINARY_DIR</code> , 也就是顶层 <code>CMakeLists.txt</code> 源码的 <code>BINARY_DIR</code>
<code>CMAKE_SOURCE_DIR</code>	与 <code>PROJECT_SOURCE_DIR</code> 等价
<code>CMAKE_BINARY_DIR</code>	与 <code>PROJECT_BINARY_DIR</code> 等价
<code>CMAKE_CURRENT_SOURCE_DIR</code>	当前源码所在路径
<code>CMAKE_CURRENT_BINARY_DIR</code>	当前源码的 <code>BINARY_DIR</code>
<code>CMAKE_MAJOR_VERSION</code>	<code>cmake</code> 的主版本号

CMAKE_MINOR_VERSION	cmake 的次版本号
CMAKE_VERSION	cmake 的版本号 (主+次+修订)
PROJECT_VERSION_MAJOR	工程的主版本号
PROJECT_VERSION_MINOR	工程的次版本号
PROJECT_VERSION	工程的版本号
CMAKE_PROJECT_NAME	工程的名字
PROJECT_NAME	工程名, 与 CMAKE_PROJECT_NAME 等价

图 32.4.11 提供信息的变量

PROJECT_SOURCE_DIR 和 PROJECT_BINARY_DIR

PROJECT_SOURCE_DIR 变量表示工程的顶级目录, 也就是顶层 CMakeLists.txt 文件所在目录; PROJECT_BINARY_DIR 变量表示工程的 BINARY_DIR, 也就是顶层 CMakeLists.txt 源码对应的 BINARY_DIR (输出文件目录)。

譬如工程目录结构如下所示:

```

├── build
├── CMakeLists.txt
└── main.c

```

CMakeLists.txt 文件内容如下:

```

# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO)

message(${PROJECT_SOURCE_DIR})
message(${PROJECT_BINARY_DIR})

```

CMakeLists.txt 中我们打印了 PROJECT_SOURCE_DIR 和 PROJECT_BINARY_DIR 变量, 进入到 build 目录下, 执行 cmake:

```

dt@dt-virtual-machine:~/vscode_ws/cmake_test$ pwd
/home/dt/vscode_ws/cmake_test
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt main.c
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
/home/dt/vscode_ws/cmake_test
/home/dt/vscode_ws/cmake_test/build
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$

```

图 32.4.12 cmake 打印信息

从打印信息可知, `PROJECT_SOURCE_DIR` 指的就是工程的顶层 `CMakeLists.txt` 源码所在路径, 而 `PROJECT_BINARY_DIR` 指的是我们执行 `cmake` 命令的所在目录, 也是顶层 `CMakeLists.txt` 源码的 `BINARY_DIR`。

➤ `CMAKE_SOURCE_DIR` 和 `CMAKE_BINARY_DIR`

与上面两个等价, 大家自己打印出来看看便知!

➤ `CMAKE_CURRENT_SOURCE_DIR` 和 `CMAKE_CURRENT_BINARY_DIR`

指的是当前源码的路径以及当前源码的 `BINARY_DIR`, 通过示例来看看, 譬如工程目录结构如下所示:

```

├── build
├── CMakeLists.txt
├── main.c
└── src
    └── CMakeLists.txt

```

顶层 `CMakeLists.txt` 文件通过 `add_subdirectory` 加载子目录 `src` 下的 `CMakeLists.txt`, `src` 目录下 `CMakeLists.txt` 文件内容如下所示:

```

# src 下的 CMakeLists.txt
message(${PROJECT_SOURCE_DIR})
message(${PROJECT_BINARY_DIR})
message(${CMAKE_CURRENT_SOURCE_DIR})
message(${CMAKE_CURRENT_BINARY_DIR})

```

通过 `message` 将这些变量打印出来, 对比看看, 进入到 `build` 目录下, 执行 `cmake`:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
/home/dt/vscode_ws/cmake_test
/home/dt/vscode_ws/cmake_test/build
/home/dt/vscode_ws/cmake_test/src
/home/dt/vscode_ws/cmake_test/build/src
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.13 cmake 打印信息

➤ **CMAKE_VERSION**、**CMAKE_MAJOR_VERSION** 和 **CMAKE_MINOR_VERSION**

记录 cmake 的版本号, 如下:

```
# CMakeLists.txt
message(${CMAKE_VERSION})
message(${CMAKE_MAJOR_VERSION})
message(${CMAKE_MINOR_VERSION})
```

打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
3.5.1
3
5
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.14 cmake 打印信息

➤ **PROJECT_VERSION**、**PROJECT_VERSION_MAJOR** 和 **PROJECT_VERSION_MINOR**

记录工程的版本号, 其实可以给工程设置一个版本号, 通过 `project()` 命令进行设置, 如下:

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0) #设置工程版本号为 1.1.0

# 打印
message(${PROJECT_VERSION})
message(${PROJECT_VERSION_MAJOR})
message(${PROJECT_VERSION_MINOR})
```

打印信息如下:


```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
1.1.0
1
1
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.15 cmake 打印信息

➤ **CMAKE_PROJECT_NAME** 和 **PROJECT_NAME**

这两者是等价的，记录了工程的名字：

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0) #设置工程版本号为 1.1.0

# 打印工程名字
message(${CMAKE_PROJECT_NAME})
message(${PROJECT_NAME})
```

打印信息如下：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
HELLO
HELLO
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.16 cmake 打印信息

● **改变行为的变量**

顾名思义，意味着这些变量可以改变某些行为，所以我们可以通过对这些变量进行设置以改变行为。

变量	说明
BUILD_SHARED_LIBS	控制 cmake 是否生成动态库
CMAKE_BUILD_TYPE	指定工程的构建类型，release 或 debug
CMAKE_SYSROOT	对应编译器的在--sysroot 选项
CMAKE_IGNORE_PATH	设置被 find_xxx 命令忽略的目录列表
CMAKE_INCLUDE_PATH	为 find_file()和 find_path()命令指定搜索路径的目录列表
CMAKE_INCLUDE_DIRECTORIES_BEFORE	用于控制 include_directories()命令的行为
CMAKE_LIBRARY_PATH	指定 find_library()命令的搜索路径的目录列表
CMAKE_MODULE_PATH	指定要由 include()或 find_package()命令加载的 CMake 模块的搜索路径的目录列表
CMAKE_PROGRAM_PATH	指定 find_program()命令的搜索路径的目录列表

图 32.4.17 改变行为的变量

➤ **BUILD_SHARED_LIB**

对于 `add_library()` 命令, 当没有显式指定生成动态库时 (`SHARED` 选项), 默认生成的是静态库; 其实我们可以通过 `BUILD_SHARED_LIBS` 变量来控制 `add_library()` 命令的行为, 当将变量设置为 `on` 时表示使能动态库, 则 `add_library()` 默认生成的便是动态库文件; 当变量设置为 `off` 或未设置时, `add_library()` 默认生成的便是静态库文件。测试如下:

譬如工程目录结构如下所示:

```

├── build
├── CMakeLists.txt
├── hello
│   └── hello.c
└── world
    └── world.c

```

顶层 `CMakeLists.txt` 文件如下所示:

```

# 顶层 CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0)

set(BUILD_SHARED_LIBS on)
add_library(hello hello/hello.c)
add_library(world world/world.c)

```

进入到 `build` 目录下, 执行 `cmake`、`make` 进行构建、编译, 将会生成动态库文件 `libhello.so`、`libworld.so`。

➤ CMAKE_BUILD_TYPE

设置编译类型 `Debug` 或者 `Release`。`debug` 版会生成相关调试信息, 可以使用 `GDB` 进行调试; `release` 不会生成调试信息:

```

# Debug 版本
set(CMAKE_BUILD_TYPE Debug)

# Release 版本
set(CMAKE_BUILD_TYPE Release)

```

关于这个 `Debug` 或者 `Release` 版本的问题, 后续有机会再给大家进行专题介绍。

➤ CMAKE_SYSROOT

`cmake` 会将该变量传递给编译器 `--sysroot` 选项, 通常我们在设置交叉编译时会使用到, 后面再说!

➤ CMAKE_INCLUDE_PATH

为 `find_file()` 和 `find_path()` 命令指定搜索路径的目录列表。这两个命令前面没给大家介绍, 它们分别用于查找文件、路径, 我们需要传入一个文件名, `find_file()` 命令会将该文件的全路径返回给我们; 而 `find_path()` 命令则会将文件的所在目录返回给我们。

这两个命令去哪找文件呢? 也就是通过 `CMAKE_INCLUDE_PATH` 变量来进行指定, `CMAKE_INCLUDE_PATH` 指定了一个目录列表, `find_file()`、`find_path()` 会去这个目录列表中查找文件。接下来我们进行测试。

譬如工程目录结构如下所示:

```

├── build
├── CMakeLists.txt
└── src

```

```
└── hello.c
```

顶层 CMakeLists.txt 文件内容如下:

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0)           #设置工程版本号为 1.1.0

find_file(P_VAR hello.c)
message(${P_VAR})
```

通过 find_file 命令查找 hello.c 文件, 将路径信息记录在 P_VAR 变量中; 现在我们没有设置 CMAKE_INCLUDE_PATH 变量, 看看能不能找到 hello.c 文件, cmake 打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
P_VAR-NOTFOUND
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.18 cmake 打印信息

很明显提示没有找到, 现在我们对 CMAKE_INCLUDE_PATH 变量进行设置, 如下所示:

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0)           #设置工程版本号为 1.1.0

# 设置 CMAKE_INCLUDE_PATH 变量
set(CMAKE_INCLUDE_PATH ${PROJECT_SOURCE_DIR}/src)

# 查找文件
find_file(P_VAR hello.c)
message(${P_VAR})
```

此时打印信息为:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
/home/dt/vscode_ws/cmake_test/src/hello.c
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.19 cmake 打印信息

这次就成功找到了 hello.c 文件, 并将文件的全路径返回给我们。

➤ CMAKE_LIBRARY_PATH

指定 find_library() 命令的搜索路径的目录列表。find_library() 命令用于搜索库文件, find_library() 将会从 CMAKE_LIBRARY_PATH 变量设置的目录列表中进行搜索。

➤ CMAKE_MODULE_PATH

指定要由 include() 或 find_package() 命令加载的 CMake 模块的搜索路径的目录列表。

➤ CMAKE_INCLUDE_DIRECTORIES_BEFORE

这个变量在前面给大家提到过,它可以改变 `include_directories()`命令的行为。`include_directories()`命令默认情况下会将目录添加到列表的后面,如果将 `CMAKE_INCLUDE_DIRECTORIES_BEFORE` 设置为 `on`,则 `include_directories()`命令会将目录添加到列表前面;同理若将 `CMAKE_INCLUDE_DIRECTORIES_BEFORE` 设置为 `off` 或未设置该变量, `include_directories()`会将目录添加到列表后面。

➤ CMAKE_IGNORE_PATH

要被 `find_program()`、`find_library()`、`find_file()`和 `find_path()`命令忽略的目录列表。表示这些命令不会去 `CMAKE_IGNORE_PATH` 变量指定的目录列表中搜索。

● 描述系统的变量

顾名思义,这些变量描述了系统相关的一些信息:

变量	说明
<code>CMAKE_HOST_SYSTEM_NAME</code>	运行 <code>cmake</code> 的操作系统名称 (其实就是 <code>uname -s</code>)
<code>CMAKE_HOST_SYSTEM_PROCESSOR</code>	运行 <code>cmake</code> 的操作系统处理器名称 (<code>uname -p</code>)
<code>CMAKE_HOST_SYSTEM</code>	运行 <code>cmake</code> 的操作系统 (复合信息)
<code>CMAKE_HOST_SYSTEM_VERSION</code>	运行 <code>cmake</code> 的操作系统版本号 (<code>uname -r</code>)
<code>CMAKE_HOST_UNIX</code>	如果运行 <code>cmake</code> 的操作系统是 UNIX 和类 UNIX,则该变量为 <code>true</code> , 否则是空值
<code>CMAKE_HOST_WIN32</code>	如果运行 <code>cmake</code> 的操作系统是 Windows, 则该变量为 <code>true</code> , 否则是空值
<code>CMAKE_SYSTEM_NAME</code>	目标主机操作系统的名称
<code>CMAKE_SYSTEM_PROCESSOR</code>	目标主机的处理器名称
<code>CMAKE_SYSTEM</code>	目标主机的操作系统 (复合信息)
<code>CMAKE_SYSTEM_VERSION</code>	目标主机操作系统的版本号
<code>ENV</code>	用于访问环境变量
<code>UNIX</code>	与 <code>CMAKE_HOST_UNIX</code> 等价
<code>WIN32</code>	与 <code>CMAKE_HOST_WIN32</code> 等价

表 32.4.3 描述系统的变量

➤ CMAKE_HOST_SYSTEM_NAME 、 CMAKE_HOST_SYSTEM_PROCESSOR 、

CMAKE_HOST_SYSTEM 和 CMAKE_HOST_SYSTEM_VERSION

这四个变量描述的是运行 `cmake` 的主机相关的信息,我们直接打印出来看看即可:

打印信息

```
message(${CMAKE_HOST_SYSTEM_NAME})
message(${CMAKE_HOST_SYSTEM_PROCESSOR})
message(${CMAKE_HOST_SYSTEM})
message(${CMAKE_HOST_SYSTEM_VERSION})
```

对应的打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Linux
x86_64
Linux-4.15.0-142-generic
4.15.0-142-generic
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.20 cmake 打印信息

大家自己对照一看就知道了, 笔者就不再多说了。

- **CMAKE_SYSTEM_NAME** 、 **CMAKE_SYSTEM_PROCESSOR** 、 **CMAKE_SYSTEM** 和 **CMAKE_SYSTEM_VERSION**

这 4 个变量则是用于描述目标主机相关的信息, 目标主机指的是可执行文件运行的主机, 譬如我们的 ARM 开发板。

打印信息

```
message(${CMAKE_SYSTEM_NAME})
message(${CMAKE_SYSTEM_PROCESSOR})
message(${CMAKE_SYSTEM})
message(${CMAKE_SYSTEM_VERSION})
```

cmake 打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Linux
x86_64
Linux-4.15.0-142-generic
4.15.0-142-generic
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.21 cmake 打印信息

因为我们并没有对 cmake 配置交叉编译, 默认会使用 Ubuntu 系统 (运行 cmake 的主机) 本身的编译工具, 所以生成的目标文件 (可执行文件或库文件) 只能运行在 Ubuntu 系统中, 所以这 4 个变量记录的依然是 Ubuntu 主机的信息。

- **ENV**

这个变量可用于访问环境变量, 用法很简单 `$ENV{VAR}`

访问环境变量

```
message($ENV{XXX})
```

通过 `$ENV{XXX}` 访问 XXX 环境变量, 我们来测试一下, 首先在 Ubuntu 系统下使用 `export` 命令导出 XXX 环境变量:

```
export XXX="Hello World!"
cd build/
cmake ..
```

打印信息如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello World!
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.22 cmake 打印信息

从打印信息可知, ENV 变量确实可以访问到 Linux 系统的环境变量。

● 控制编译的变量

这些变量可以控制编译过程, 具体如下所示:

变量	说明
EXECUTABLE_OUTPUT_PATH	可执行程序的输出路径
LIBRARY_OUTPUT_PATH	库文件的输出路径

表 32.4.4 控制编译的变量

这两个变量前面我们已经用到过了, 分别用来设置可执行文件的输出目录以及库文件的输出目录, 接下来我们进行简单地测试。

譬如工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
├── hello
│   ├── hello.c
│   └── hello.h
└── main.c
```

hello.c 会被编译成动态库文件 libhello.so, 而 main.c 会被编译成可执行程序, main.c 源码中调用了 hello.c 提供的函数; 顶层 CMakeLists.txt 文件内容如下所示:

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0)           #设置工程版本号为 1.1.0

# 设置可执行文件和库文件输出路径
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)

# 头文件包含
include_directories(hello)

# 动态库目标
add_library(hello SHARED hello/hello.c)

# 可执行程序目标
add_executable(main main.c)
target_link_libraries(main PRIVATE hello) #链接库
```

进入到 build 目录下, 执行 cmake、make 进行构建、编译, 最终会生成可执行文件 main 和库文件 libhello.so, 目录结构如下所示:

```

├── build
│   ├── bin
│   │   └── main
│   └── lib
│       └── libhello.so
├── CMakeLists.txt
├── hello
│   ├── hello.c
│   └── hello.h
└── main.c

```

这是因为我们通过设置 EXECUTABLE_OUTPUT_PATH 和 LIBRARY_OUTPUT_PATH 才会使得生成的可执行程序在 build/bin 目录下、生成的库文件在 build/lib 目录下, 如果把这两行给注释掉, 那么生成的文件在 build 目录中, 因为默认情况下, 最终的目标文件的输出目录就是源码的 BINARY_DIR。

32.4.4 双引号的作用

CMake 中, 双引号的作用我们可以从两个方面进行介绍, 命令参数和引用变量。

命令参数

调用命令时, 参数可以使用双引号, 譬如:

```
project("HELLO")
```

也可以不使用双引号, 譬如:

```
project(HELLO)
```

那它们有什么区别呢? 在本例中是没有区别的, 命令中多个参数之间使用空格进行分隔, 而 cmake 会将双引号引起来的内容作为一个整体, 当它当成一个参数, 假如你的参数中有空格(空格是参数的一部分), 那么就可以使用双引号, 如下所示:

```
message>Hello World)
```

```
message("Hello World")
```

在这个例子中, 第一个 message 命令传入了两个参数, 而第二个 message 命令只传入一个参数; 在第一个 message 命令中, 打印信息时, 会将两个独立的字符串 Hello 和 World 都打印出来, 而且 World 会紧跟在 Hello 之后, 如下:

```
HelloWorld
```

而第二个 message 命令只有一个参数, 所以打印信息如下:

```
Hello World
```

这就是双引号在参数中的一个作用。

引用变量

我们先来看个例子, 如下所示:

```
# CMakeLists.txt
```

```
set(MY_LIST Hello World China)
```

```
message(${MY_LIST})
```

这个例子的打印信息如下:

```
HelloWorldChina
```

在这个例子中, MY_LIST 是一个列表, 该列表包含了 3 个元素, 分别是 Hello、World、China。但这个 message 命令打印时却将这三个元素全部打印出来, 并且各个元素之间没有任何分隔。此时我们可以在引用变量 (`${MY_LIST}`) 时加上双引号, 如下所示:

```
# CMakeLists.txt
```

```
set(MY_LIST Hello World China)
```

```
message("${MY_LIST}")
```

此时 message 打印信息如下:

```
Hello;World;China
```

因为此时 `${MY_LIST}` 是一个列表, 我们用 `"${MY_LIST}"` 这种形式的时候, 表示要让 CMake 把这个数组的所有元素当成一个整体, 而不是分散的个体。于是, 为了保持数组的含义, 又提供一个整体的表达方式, CMake 就会用分号 “;” 把这数组的多个元素连接起来。

而如果不加双引号时, CMake 不会数组当成一个整体看待, 而是会将数组中的各个元素提取出进行打印输出。

32.4.5 条件判断

在 cmake 中可以使用条件判断, 条件判断形式如下:

```
if(expression)
```

```
  # then section.
```

```
  command1(args ...)
```

```
  command2(args ...)
```

```
  ...
```

```
elseif(expression2)
```

```
  # elseif section.
```

```
  command1(args ...)
```

```
  command2(args ...)
```

```
  ...
```

```
else(expression)
```

```
  # else section.
```

```
  command1(args ...)
```

```
  command2(args ...)
```

```
  ...
```

```
endif(expression)
```

else 和 endif 括号中的 `<expression>` 可写可不写, 如果写了, 就必须和 if 中的 `<expression>` 一致。

expression 就是一个进行判断的表达式, 表达式对照表如下:

表达式	true	false	说明
<code><constant></code>	如果 constant 为 1、ON、YES、TRUE、Y 或非零数, 则为真	如果 constant 为 0、OFF、NO、FALSE、N、IGNORE、NOTFOUND、空字符串或以后缀 NOTFOUND 结尾, 则为 False。	布尔值大小写不敏感; 如果与这些常量都不匹配, 则将其视为变量或字符串

<variable string>	已经定义并且不是 false 的变量	未定义或者是 false 的变量	变量就是字符串
NOT <expression>	expression 为 false	expression 为 true	
<expr1> AND <expr2>	expr1 和 expr2 同时为 true	expr1 和 expr2 至少有一个为 false	
<expr1> OR <expr2>	expr1 和 expr2 至少有一个为 true	expr1 和 expr2 都是 false	
COMMAND name	name 是一个已经定义的命令、宏或者函数	name 未定义	
TARGET name	name 是 add_executable()、add_library() 或 add_custom_target() 定义的目标	name 未定义	
TEST name	name 是由 add_test() 命令创建的现有测试名称	name 未创建	
EXISTS path	path 指定的文件或目录存在	path 指定的文件或目录不存在	仅适用于完整路径
IS_DIRECTORY path	path 指定的路径为目录	path 指定的路径不为目录	仅适用于完整路径
IS_SYMLINK path	path 为符号链接	path 不是符号链接	仅适用于完整路径
IS_ABSOLUTE path	path 为绝对路径	path 不是绝对路径	
<variable string> MATCHES regex	variable 与正则表达式 regex 匹配成功	variable 与正则表达式 regex 匹配失败	
<variable string> IN_LIST <variable>	右边列表中包含左边的元素	右边列表中不含左边的元素	
DEFINED <variable>	如果给定的变量已定义, 则为真。	如果给定的变量未定义	只要变量已经被设置, 它是真还是假并不重要。(注意宏不是变量。)
<variable string> LESS <variable string>	如果给定的字符串或变量的值是有效数字且小于右侧的数字, 则为真。	左侧的数字大于或等于右侧的数字	
<variable string> GREATER <variable string>	如果给定的字符串或变量的值是有效数字	左侧的数字小于或等于右侧的数字	

	且大于右侧的数字, 则为真。		
<variable string> EQUAL <variable string>	如果给定的字符串或变量的值是有效数字并且等于右侧的值, 则为真	左侧的数字不等于右侧的数字	

表 32.4.5 if 条件判断<表达式>对照表

上表中只是列出其中一部分表达式, 还有其它一些表达式这里并未列出, 大家可以通过 <https://cmake.org/cmake/help/v3.5/command/if.html> 这个链接地址进行查看, 现在我们对上表中的表达式进行详解。

- **<constant>**

在 if(constant)条件判断中, 如果 constant 是 1、ON、YES、TRUE、Y 或非零数字, 那么这个 if 条件就是 true; 如果 constant 是 0、OFF、NO、FALSE、N、IGNORE、NOTFOUND、空字符串或以后缀-NOTFOUND 结尾, 那么这个条件判断的结果就是 false。

在 cmake 中, 可以把 1、ON、YES、TRUE、Y 或非零数字以及 0、OFF、NO、FALSE、N、IGNORE、NOTFOUND、空字符串或以后缀-NOTFOUND 结尾这些理解为常量, 类似于布尔值, 而且它们不区分大小写; 如果参数不是这些特定常量之一, 则将其视为变量或字符串, 并使用除<constant>之外的表达式。

```
if(ON)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(YES)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(true)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(100)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(0)
    message(true)
```

```
else()
    message(false)
endif()
```

输出为: false

```
if(N)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(NO)
    message(true)
else()
    message(false)
endif()
```

输出为: false

- **<variable/string>**

在 if(<variable/string>)条件判断中, 如果变量已经定义, 并且它的值是一个非假常量, 则条件为真; 否则为假, 注意宏参数不是变量 (在 cmake 中也可以使用宏, 这个后面再给大家介绍)。

```
set(GG Hello)
if(GG)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
set(GG NO)
if(GG)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(GG)
    message(true)
else()
    message(false)
endif()
```

输出为: false

- **NOT <expression>**

NOT 其实就类似于 C 语言中的取反, 在 if(NOT <expression>)条件判断中, 如果表达式 expression 为真, 则条件判断为假; 如果表达式 expression 为假, 则条件判断为真。

```
if(NOT GG)
    message(true)
else()
    message(false)
endif()
```

输出为: true

因为 GG 变量没有定义, 所以 GG 表达式为假, 但因为前面有 NOT 关键字, 进行取反操作, 整个 if 条件判断为真。

```
if(NOT YES)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(NOT 0)
    message(true)
else()
    message(false)
endif()
```

输出为: true

- **<expr1> AND <expr2>**

这个就类似于 C 语言中的逻辑与 (&&), 只有 expr1 和 expr2 同时为真时, 条件判断才为真; 否则条件判断为假。

```
if(yes AND on)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(yes AND no)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(false AND no)
    message(true)
else()
    message(false)
endif()
```

输出为: false

- **<expr1> OR <expr2>**

类似于 C 语言中的逻辑或 (||)，当 expr1 或 expr2 至少有一个为真时，条件判断为真；否则为假。

```
if(false OR no)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(yes OR no)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(ON OR yes)
    message(true)
else()
    message(false)
endif()
```

输出为: true

- **COMMAND command-name**

如果 command-name 是一个已经定义的命令、宏或函数时，条件判断为真；否则为假。

除了宏之外，在 cmake 中还可以定义函数，这个我们也会在后面向大家介绍。

```
if(COMMAND yyds)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(COMMAND project)
    message(true)
else()
    message(false)
endif()
```

输出为: true

- **TARGET target-name**

如果 target-name 是 add_executable()、add_library()或 add_custom_target()定义的目标（这些目标在整个工程中必须是唯一的，不可出现两个名字相同的目标），则条件判断为真；否则为假。

```
if(TARGET hello)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
add_library(hello hello.c)
```

```
if(TARGET hello)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

● EXISTS path

如果 path 指定的文件或目录存在, 则条件判断为真; 否则为假。需要注意的是, path 必须是文件或目录的全路径, 也就是绝对路径。

譬如工程目录结构如下所示:

```
|— build
|— CMakeLists.txt
|— hello
|   |— hello.c
|   |— hello.h
|— main.c
```

在顶层 CMakeLists.txt 文件中使用 if(EXISTS path)进行判断:

```
if(EXISTS ${PROJECT_BINARY_DIR})
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

```
if(EXISTS ${PROJECT_BINARY_DIR}/hello)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

```
if(EXISTS ${PROJECT_BINARY_DIR}/world)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: false

```
if(EXISTS ${PROJECT_BINARY_DIR}/hello/hello.c)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

- **IS_DIRECTORY path**

如果 path 指定的路径是一个目录, 则条件判断为真; 否则为假, 同样, path 也必须是一个绝对路径。还是以上例中的工程目录结构为例:

```
if(IS_DIRECTORY ${PROJECT_BINARY_DIR}/hello)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(IS_DIRECTORY ${PROJECT_BINARY_DIR}/hello/hello.c)
    message(true)
else()
    message(false)
endif()
```

输出为: true

- **IS_ABSOLUTE path**

如果给定的路径 path 是一个绝对路径, 则条件判断为真; 否则为假。

```
if(IS_ABSOLUTE ${PROJECT_BINARY_DIR})
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
if(IS_ABSOLUTE ./hello)
    message(true)
else()
    message(false)
endif()
```

输出为: false

- **<variable|string> MATCHES regex**

这个表达式用的比较多, 可以用来匹配字符串, 可以使用正则表达式进行匹配。

如果给定的字符串或变量的值与给定的正则表达式匹配, 则为真, 否则为假。

```
set(MY_STR "Hello World")

if(MY_STR MATCHES "Hello World")
    message(true)
else()
    message(false)
endif()
```

输出为: true

其实也可以引用变量:

```
set(MY_STR "Hello World")
```

```
if(${MY_STR} MATCHES "Hello World")
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
set(MY_STR "Hello World")
```

```
if("Hello World" MATCHES "Hello World")
    message(true)
else()
    message(false)
endif()
```

输出为: true

- **<variable|string> IN_LIST <variable>**

如果左边给定的变量或字符串是右边列表中的某个元素相同, 则条件判断为真; 否则为假。

```
set(MY_LIST Hello World China)
```

```
if(Hello IN_LIST MY_LIST)
    message(true)
else()
    message(false)
endif()
```

输出为: true

```
set(MY_LIST Hello World China)
```

```
set(Hello China)
```

```
if(Hello IN_LIST MY_LIST)
    message(true)
else()
    message(false)
endif()
```

输出为: true

- **DEFINED <variable>**

如果给定的变量已经定义, 则条件判断为真, 否则为假; 只要变量已经被设置(定义), if 条件判断就是真, 至于变量的值是真还是假并不重要。

```
if(DEFINED yyds)
    message(true)
else()
    message(false)
```



```
endif()
```

输出为: false

```
set(yyds "YYDS")
```

```
if(DEFINED yyds)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

- **<variable|string> LESS <variable|string>**

如果左边给定的字符串或变量的值是有效数字并且小于右侧的值, 则为真。否则为假。

测试如下:

```
if(100 LESS 20)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: false

```
if(20 LESS 100)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

- **<variable|string> GREATER <variable|string>**

如果左边给定的字符串或变量的值是有效数字并且大于右侧的值, 则为真。否则为假。

测试如下:

```
if(20 GREATER 100)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: false

```
if(100 GREATER 20)
```

```
    message(true)
```

```
else()
```

```
    message(false)
```

```
endif()
```

输出为: true

- **<variable|string> EQUAL <variable|string>**

如果左边给定的字符串或变量的值是有效数字并且等于右侧的值, 则为真。否则为假。

测试如下:

```
if(100 EQUAL 20)
    message(true)
else()
    message(false)
endif()
```

输出为: false

```
if(100 EQUAL 100)
    message(true)
else()
    message(false)
endif()
```

输出为: true

● elseif 分支

可以使用 elseif 组成多个不同的分支:

```
set(MY_LIST Hello World China)

if>Hello IN_LIST MY_LIST)
    message>Hello)
elseif(World IN_LIST MY_LIST)
    message(World)
elseif(China IN_LIST MY_LIST)
    message(China)
else()
    message>false)
endif()
```

32.4.6 循环语句

cmake 中除了 if 条件判断之外,还支持循环语句,包括 foreach()循环、while()循环。

一、foreach 循环

①、foreach 基本用法

foreach 循环的基本用法如下所示:

```
foreach(loop_var arg1 arg2 ...)
    command1(args ...)
    command2(args ...)
    ...
endforeach(loop_var)
```

endforeach 括号中的<loop_var>可写可不写,如果写了,就必须和 foreach 中的<loop_var>一致。

参数 loop_var 是一个循环变量,循环过程中会将参数列表中的变量依次赋值给他,类似于 C 语言 for 循环中经常使用的变量 i。

```
# foreach 循环测试
```

```
foreach(loop_var A B C D)
    message("${loop_var}")
endforeach()
```

打印信息为:

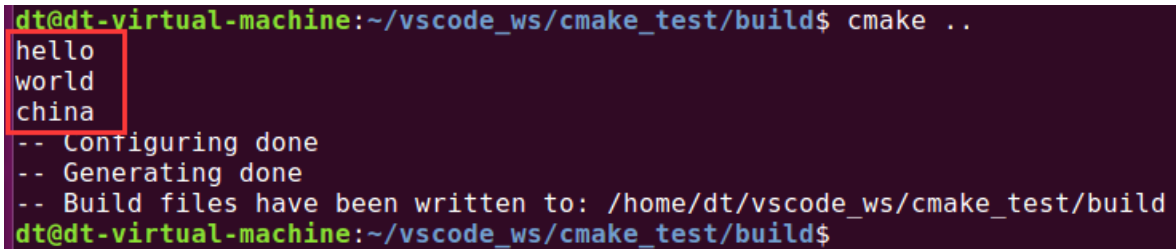
```
A
B
C
D
```

使用 foreach 可以编译一个列表中的所有元素, 如下所示:

```
# foreach 循环测试
set(my_list hello world china)

foreach(loop_var ${my_list})
    message("${loop_var}")
endforeach()
```

打印信息如下:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
hello
world
china
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.23 cmake 打印信息

②、foreach 循环之 RANGE 关键字

用法如下所示:

```
foreach(loop_var RANGE stop)
foreach(loop_var RANGE start stop [step])
```

对于第一种方式, 循环会从 0 到指定的数字 stop, 包含 stop, stop 不能为负数。

而对于第二种, 循环从指定的数字 start 开始到 stop 结束, 步长为 step, 不过 step 参数是一个可选参数, 如果不指定, 默认 step=1; 三个参数都不能为负数, 而且 stop 不能比 start 小。

接下来我们进行测试, 测试一:

```
# foreach 循环测试
foreach(loop_var RANGE 4)
    message("${loop_var}")
endforeach()
```

打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
0
1
2
3
4
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.24 cmake 打印信息

测试二:

```
# foreach 循环测试
foreach(loop_var RANGE 1 4 1)
    message("${loop_var}")
endforeach()
```

打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
1
2
3
4
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.25 cmake 打印信息

③、foreach 循环之 IN 关键字

用法如下:

```
foreach(loop_var IN [LISTS [list1 [...]]]
            [ITEMS [item1 [...]])
```

循环列表中的每一个元素, 或者直接指定元素。

接下来进行测试, 测试一:

```
# foreach 循环测试
set(my_list A B C D)

foreach(loop_var IN LISTS my_list)
    message("${loop_var}")
endforeach()
```

打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
A
B
C
D
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.26 cmake 打印信息

测试二:

```
# foreach 循环测试
foreach(loop_var IN ITEMS A B C D)
    message("${loop_var}")
endforeach()
```

打印信息同上。

二、while 循环

while 循环用法如下:

```
while(condition)
    command1(args ...)
    command1(args ...)
    ...
endwhile(condition)
```

endwhile 括号中的 condition 可写可不写, 如果写了, 就必须和 while 中的 condition 一致。

cmake 中 while 循环的含义与 C 语言中 while 循环的含义相同, 但条件 condition 为真时, 执行循环体中的命令, 而条件 condition 的语法形式与 if 条件判断中的语法形式相同。

```
# while 循环测试
set(loop_var 4)

while(loop_var GREATER 0)
    message("${loop_var}")
    math(EXPR loop_var "${loop_var} - 1")
endwhile()
```

输出结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
4
3
2
1
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.27 cmake 打印信息

上例中, while 循环的条件是(loop_var GREATER 0), 等价于(loop_var > 0), 当 loop_var 变量的有效数值大于 0 时, 执行 while 循环体; 在 while 循环体中使用到了 cmake 中的数学运算命令 math(), 关于数学运算下小节会向大家介绍。

在 while 循环体中, 打印 loop_var, 之后将 loop_var 减一。

三、break、continue

cmake 中, 也可以在循环体中使用类似于 C 语言中的 break 和 continue 语句。

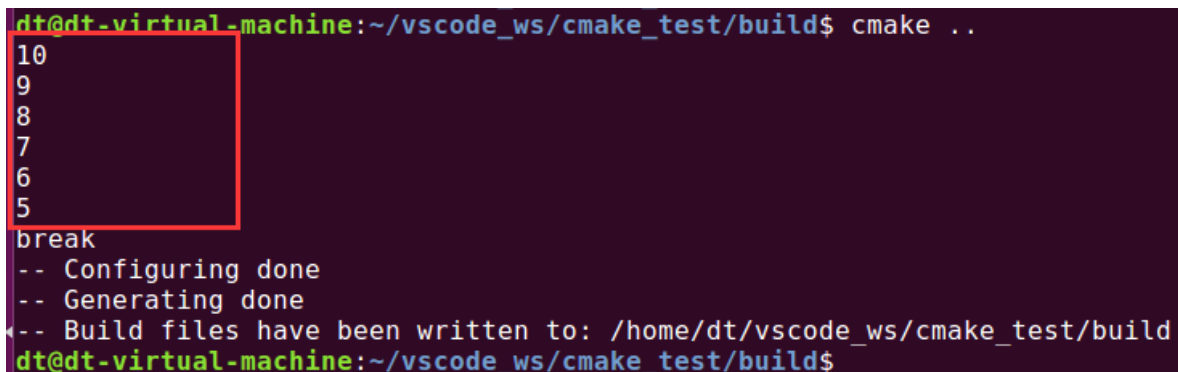
①、break

break()命令用于跳出循环, 和在 C 语言中的作用是一样的, 测试如下:

```
# while...break 测试
set(loop_var 10)

while(loop_var GREATER 0)  #loop_var>0 时 执行循环体
  message("${loop_var}")
  if(loop_var LESS 6) #当 loop_var 小于 6 时
    message("break")
    break()  #跳出循环
  endif()
  math(EXPR loop_var "${loop_var} - 1")#loop_var--
endwhile()
```

打印信息如下:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
10
9
8
7
6
5
break
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.28 cmake 打印信息

整个代码笔者就不再解释了, 注释已经写得很清楚了!

②、continue

continue()命令用于结束本次循环, 执行下一次循环, 测试如下:

```
# while...continue 测试
# 打印所有偶数
set(loop_var 10)

while(loop_var GREATER 0)  #loop_var>0 时 执行循环体
  math(EXPR var "${loop_var} % 2")  #求余

  if(var EQUAL 0) #如果 var=0,表示它是偶数
```

```

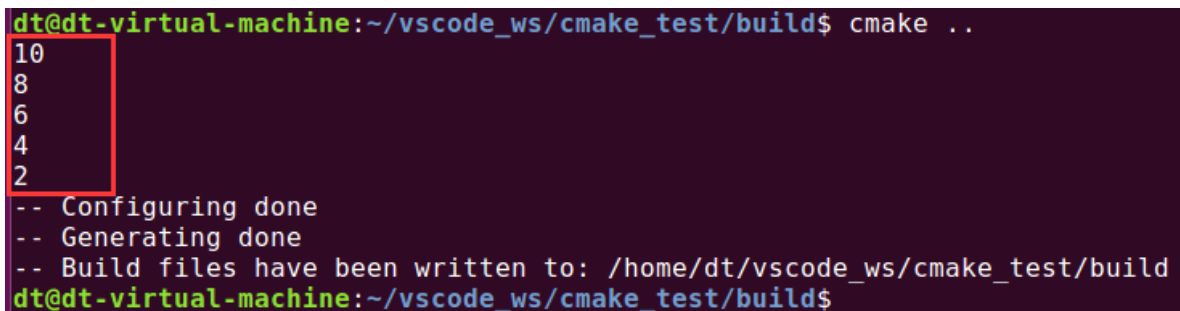
message("${loop_var}") #打印这个偶数
math(EXPR loop_var "${loop_var} - 1")#loop_var--
continue() # 执行下一次循环
endif()

math(EXPR loop_var "${loop_var} - 1")#loop_var--
endwhile()

```

这段 cmake 代码是求 0 到 10 之间的偶数（左闭右开），并将偶数打印出来，使用到了 `continue()` 命令，代码不再解释，注释已经写得很清楚了。

打印结果如下：



```

dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
10
8
6
4
2
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$

```

图 32.4.29 cmake 打印信息

关于 `break()` 和 `continue()` 命令的使用就介绍到这里了。

32.4.7 数学运算 math

在 cmake 中如何使用数学运算呢？其实，cmake 提供了一个命令用于实现数学运算功能，这个命令就是 `math()`，如下所示：

```
math(EXPR <output variable> <math expression>)
```

`math` 命令中，第一个参数是一个固定的关键字 `EXPR`，第二个参数是一个返回参数，将数学运算结果存放在这个变量中；而第三个参数则是一个数学运算表达式，支持的运算符包括：`+`（加）、`-`（减）、`*`（乘）、`/`（除）、`%`（求余）、`|`（按位或）、`&`（按位与）、`^`（按位异或）、`~`（按位取反）、`<<`（左移）、`>>`（右移）以及这些运算符的组合运算，它们的含义与 C 语言中相同。

譬如：

```

math(EXPR out_var "1+1")    #计算 1+1
math(EXPR out_var "100 * 2") ##计算 100x2
math(EXPR out_var "10 & 20") #计算 10 & 20

```

我们进行测试：

```

# math()命令测试
math(EXPR out_var "100 + 100")
message("${out_var}")

math(EXPR out_var "100 - 50")
message("${out_var}")

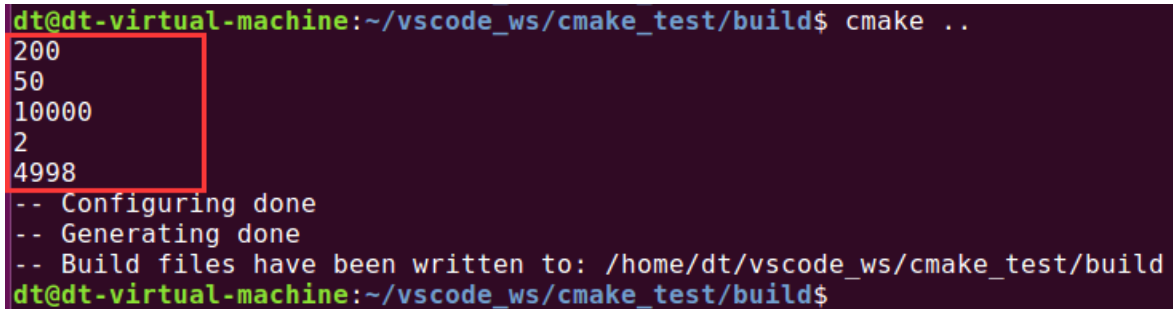
math(EXPR out_var "100 * 100")
message("${out_var}")

```

```
math(EXPR out_var "100 / 50")
message("${out_var}")

math(EXPR out_var "(100 & 100) * 50 - 2")
message("${out_var}")
```

测试结果如下:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
200
50
10000
2
4998
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.4.30 cmake 打印信息

32.5 cmake 进阶

上小节, 已经将 cmake 中常用的命令 `command`、变量 `variable` 都给大家进行了详细介绍, 通过上小节的学习, 相信大家已经掌握了 cmake 工具的基本使用方法; 本小节我们在进一步学习 cmake, 看看 cmake 还有哪些东西。

32.5.1 定义函数

在 cmake 中我们也可以定义函数, cmake 提供了 `function()` 命令用于定义一个函数, 使用方法如下所示:

```
function(<name> [arg1 [arg2 [arg3 ...]])
    command1(args ...)
    command2(args ...)
    ...
endfunction(<name>)
```

`endfunction` 括号中的 `<name>` 可写可不写, 如果写了, 就必须和 `function` 括号中的 `<name>` 一致。

①、基本使用方法

第一个参数 `name` 表示函数的名字, `arg1`、`arg2`... 表示传递给函数的参数。调用函数的方法其实就跟使用命令一样, 一个简单地示例如下所示:

```
# function 函数测试
# 函数名: xyz
function(xyz arg1 arg2)
    message("${arg1} ${arg2}")
endfunction()
```

```
# 调用函数
xyz>Hello World)
```

打印信息如下:


```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello World
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.1 cmake 打印信息

②、使用 return()命令

在 function()函数中也可以使用 C 语言中的 return 语句退出函数，如下所示：

```
# function 函数测试
# 函数名: xyz
function(xyz)
    message>Hello)
    return() # 退出函数
    message(World)
endfunction()

# 调用函数
xyz()
```

执行结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.2 cmake 打印信息

只打印了 Hello，并没有打印 World，说明 return()命令是生效的，执行 return()命令之后就已经退出当前函数了，所以并不会打印 World。但是需要注意的是，return 并不可以用于返回参数，那函数中如何返回参数给调用者呢？关于这个问题，后续再给大家讲解，因为这里涉及到其它一些问题，本小节暂时先不去理会这个问题。

③、可变参函数

在 cmake 中，调用函数时实际传入的参数个数不需要等于函数定义的参数个数（甚至函数定义时，参数个数为 0），但是实际传入的参数个数必须大于或等于函数定义的参数个数，如下所示：

```
# function 函数测试
# 函数名: xyz
function(xyz arg1)
    message>${arg1})
endfunction()

# 调用函数
xyz>Hello World China)
```

函数 xyz 定义时只有一个参数，但是实际调用时我们传入了 3 个参数，注意这并不会报错，是符合 function()语法规则的，会正常执行，打印信息如下：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.3 cmake 打印信息

从打印信息可知, `message()`命令打印出了调用者传入的第一个参数, 也就是 `Hello`。

这种设计有什么用途呢? 正如我们的标题所言, 这种设计可用于实现可变参函数(与 C 语言中的可变参数函数概念相同); 但是有个问题, 就如上例中所示, 用户传入了 3 个参数, 但是函数定义时并没有定义这些形参, 函数中如何引用到第二个参数 `World` 以及第三个参数 `China` 呢? 其实 `cmake` 早就为大家考虑到了, 并给出了相应的解决方案, 就是接下来向大家介绍的内部变量。

④、函数的内部变量

`function()`函数中可以使用内部变量, 所谓函数的内部变量, 指的就是在函数内部使用的内置变量, 这些内部变量如下所示:

函数中的内部变量	说明
ARGVX	X 是一个数字, 譬如 <code>ARGV0</code> 、 <code>ARGV1</code> 、 <code>ARGV2</code> 、 <code>ARGV3...</code> , 这些变量表示函数的参数, <code>ARGV0</code> 为第一个参数、 <code>ARGV1</code> 为第二个参数, 依次类推!
ARGV	实际调用时传入的参数会存放在 <code>ARGV</code> 变量中(如果是多个参数, 那它就是一个参数列表)
ARGN	假如定义函数时参数为 2 个, 实际调用时传入了 4 个, 则 <code>ARGN</code> 存放了剩下的 2 个参数(如果是多个参数, 那它也是一个参数列表)
ARGC	调用函数时, 实际传入的参数个数

我们可以进行测试:

```
# function 函数测试
# 函数名: xyz
function(xyz arg1 arg2)
    message("ARGC: ${ARGC}")
    message("ARGV: ${ARGV}")
    message("ARGN: ${ARGN}")
    message("ARGV0: ${ARGV0}")
    message("ARGV1: ${ARGV1}")

    # 循环打印出各个参数
    set(i 0)
    foreach(loop ${ARGV})
        message("arg${i}: " ${loop})
        math(EXPR i "${i} + 1")
    endforeach()
endfunction()

# 调用函数
xyz(A B C D E F G)
```

源码执行结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
ARGC: 7
ARGV: A;B;C;D;E;F;G
ARGN: C;D;E;F;G
ARGV0: A
ARGV1: B
arg0: A
arg1: B
arg2: C
arg3: D
arg4: E
arg5: F
arg6: G
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.4 cmake 打印信息

这个大家自己去对照一下就知道了。

⑤、函数的作用域

在 cmake 中, 通过 `function()` 命令定义的函数类似于一个自定义命令 (实际上并不是), 当然, 事实上, cmake 提供了自定义命令的方式, 譬如通过 `add_custom_command()` 来实现, 如果大家有兴趣, 可以自己去学习下, 笔者便不再进行介绍了。

使用 `function()` 定义的函数, 我们需要对它的使用范围进行一个简单地了解, 譬如如有如下工程目录结构:

```
├── build
├── CMakeLists.txt
├── hello
│   └── CMakeLists.txt
```

我们在顶层目录下定义了一个函数 `xyz`, 顶层 `CMakeLists.txt` 源码内容如下:

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0)

# 函数名: xyz
function(xyz)
    message("Hello World!")
endfunction()

# 加载子源码
add_subdirectory(hello)
```

接着我们在子源码中调用 `xyz()` 函数, `hello` 目录下的 `CMakeLists.txt` 如下所示:

```
# hello 目录下的 CMakeLists.txt
message("这是子源码")
xyz()      # 调用 xyz() 函数
```

大家觉得这样子可以调用成功吗? 事实上, 这是没问题的, 父源码中定义的函数、在子源码中是可以调用的, 打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
这是子源码
Hello World!
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.5 cmake 打印信息

那反过来,子源码中定义的函数,在父源码中可以使用吗?我们来进行测试,顶层 CMakeLists.txt 源码内容如下:

```
# CMakeLists.txt
cmake_minimum_required("VERSION" "3.5")
project(HELLO VERSION 1.1.0)           #设置工程版本号为 1.1.0

# 加载子源码
add_subdirectory(hello)
message("这是父源码")
xyz()
```

在父源码中调用 xyz()函数,在子源码中定义 xyz()函数,如下所示:

```
message("这是子源码")

# 函数名: xyz
function(xyz)
    message("Hello World!")
endfunction()
```

进入到 build 目录执行 cmake,如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
这是子源码
这是父源码
Hello World!
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.6 cmake 打印信息

事实证明,这样也是可以的,说明通过 function()定义的函数它的使用范围是全局的,并不局限于当前源码、可以在其子源码或者父源码中被使用。

32.5.2 宏定义

cmake 提供了定义宏的方法,cmake 中函数 function 和宏定义 macro 在某种程度上来说是一样的,都是创建一段有名字的代码可以在后面被调用,还可以传参数。通过 macro()命令定义宏,如下所示:

```
macro(<name> [arg1 [arg2 [arg3 ...]])
    COMMAND1(ARGS ...)
    COMMAND2(ARGS ...)
    ...
endmacro(<name>)
```

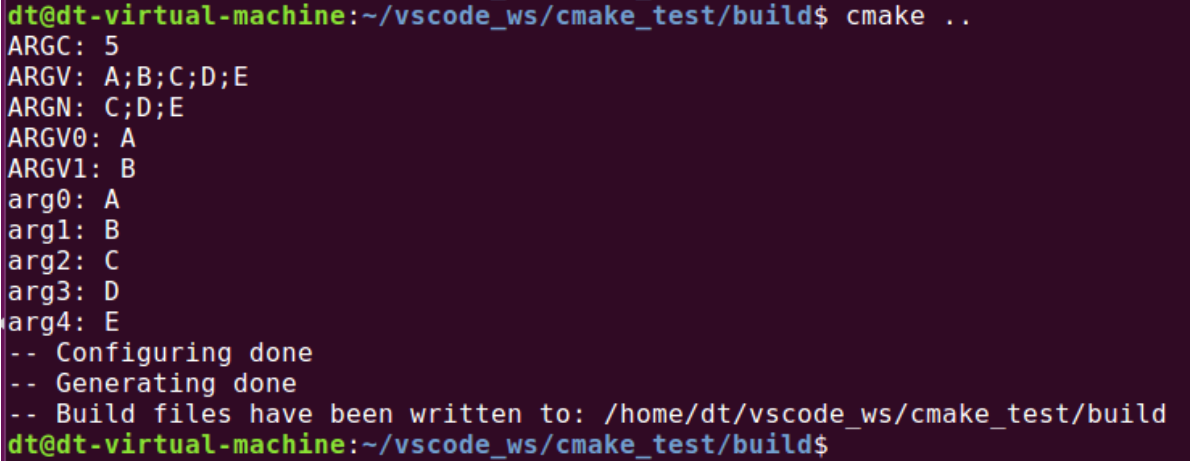
endmacro 括号中的<name>可写可不写, 如果写了, 就必须和 macro 括号中的<name>一致。参数 name 表示宏定义的名字, 在宏定义中也可以使用前面给大家介绍的 ARGVX (X 是一个数字)、ARGC、ARGV、ARGN 这些变量, 所以这些也是宏定义的内部变量, 如下所示:

```
# macro 宏定义测试
macro(XYZ arg1 arg2)
    message("ARGC: ${ARGC}")
    message("ARGV: ${ARGV}")
    message("ARGN: ${ARGN}")
    message("ARGV0: ${ARGV0}")
    message("ARGV1: ${ARGV1}")

    # 循环打印出各个参数
    set(i 0)
    foreach(loop ${ARGV})
        message("arg${i}: " ${loop})
        math(EXPR i "${i} + 1")
    endforeach()
endmacro()

# 使用宏
XYZ(A B C D E)
```

源码打印信息如下:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
ARGC: 5
ARGV: A;B;C;D;E
ARGN: C;D;E
ARGV0: A
ARGV1: B
arg0: A
arg1: B
arg2: C
arg3: D
arg4: E
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.7 cmake 打印信息

从定义上看他们貌似一模一样, 宏和函数确实差不多, 但还是有区别的, 譬如, 宏的参数和诸如 ARGV、ARGC、ARGN 之类的值不是通常 CMake 意义上的变量, 它们是字符串替换, 就像 C 语言预处理器对宏所做的一样, 因此, 您将无法使用以下命令:

```
if(ARGV1) # ARGV1 is not a variable
if(DEFINED ARGV2) # ARGV2 is not a variable
if(ARGC GREATER 2) # ARGC is not a variable
foreach(loop_var IN LISTS ARGN) # ARGN is not a variable
```

因为在宏定义中,宏的参数和诸如 ARGV、ARGV、ARGN 等这些值并不是变量,它们是字符串替换,也就是说,当 cmake 执行宏定义时,会先将宏的参数和 ARGV、ARGV、ARGN 等这些值进行字符串替换,然后再去执行这段宏,其实就像是 C 语言中的预处理步骤,这是与函数不同的地方。

我们来进行测试:

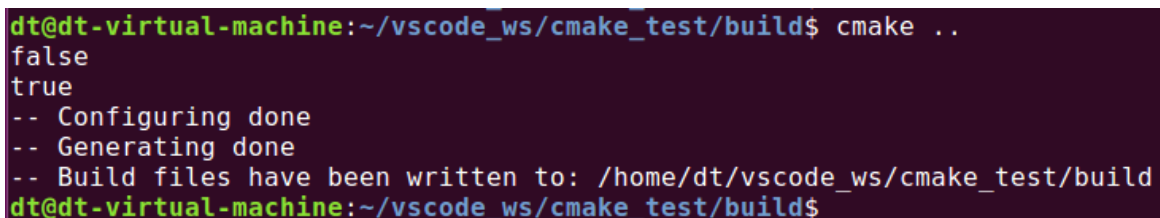
```
# macro 宏
macro(abc arg1 arg2)
    if(DEFINED ARGV)
        message(true)
    else()
        message(false)
    endif()
endmacro()

# function 函数
function(xyz arg1 arg2)
    if(DEFINED ARGV)
        message(true)
    else()
        message(false)
    endif()
endfunction()

# 调用宏
abc(A B C D)

# 调用函数
xyz(A B C D)
```

上面的代码中,我们定义了一个宏 abc 和一个函数 xyz,它们俩的代码是一样的,都是在内部使用 if() 判断 ARGV 是不是一个变量,如果是打印 true,如果不是打印 false;下面会分别调用宏 abc 和函数 xyz,打印信息如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
false
true
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.8 cmake 打印信息

所以从打印信息可知,在宏定义中,ARGV 确实不是变量,其实在执行宏之前,会将 ARGV 进行替换,如下所示:

```
if(DEFINED 4)
    message(true)
else()
    message(false)
```

```
endif()
```

把 ARGV 替换为 4 (因为我们实际传入了 4 个参数)。

当然,除此之外,cmake 中函数和宏定义还有其它的区别,譬如函数有自己的作用域、而宏定义是没有作用域的概念。

32.5.3 文件操作

cmake 提供了 file()命令可对文件进行一系列操作,譬如读写文件、删除文件、文件重命名、拷贝文件、创建目录等等,本小节我们一起来学习这个功能强大的 file()命令。

①、写文件:写、追加内容

使用 file()命令写文件,使用方式如下所示:

```
file(WRITE <filename> <content>...)
```

```
file(APPEND <filename> <content>...)
```

将<content>写入名为<filename>的文件中。如果文件不存在,它将被创建;如果文件已经存在,WRITE 模式将覆盖它,APPEND 模式将内容追加到文件末尾。

测试代码如下:

```
# file()写文件测试
```

```
file(WRITE wtest.txt "Hello World!") #给定内容生成 wtest.txt 文件
```

```
file(APPEND wtest.txt " China") #给定内容追加到 wtest.txt 文件末尾
```

注意文件可以使用绝对路径或相对路径指定,相对路径被解释为相对于当前源码路径。

执行 CMakeLists.txt 代码之后,会在当前源码目录下生成一个名为 wtest.txt 的文件,如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake .. 执行cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cd ..
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt wtest.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.5.9 执行 cmake 前后对比

接着查看 wtest.txt 文件中内容,如下所示:

```
Hello World! China
~
~
~
~
~
```

图 32.5.10 wtest.txt 文件的内容

②、写文件:由内容生成文件

由内容生成文件的命令为:

```
file(GENERATE OUTPUT output-file
```

```
<INPUT input-file|CONTENT content>
```

[CONDITION expression])

output-file: 指定输出文件名, 可以带路径(绝对路径或相对路径);

INPUT input-file: 指定输入文件, 通过输入文件的内容来生成输出文件;

CONTENT content: 指定内容, 直接指定内容来生成输出文件;

CONDITION expression: 如果表达式 expression 条件判断为真, 则生成文件、否则不生成文件。

同样, 指定文件既可以使用相对路径、也可使用绝对路径, 不过在这里, 相对路径被解释为相对于当前源码的 `BINARY_DIR` 路径, 而不是当前源码路径。

测试代码如下:

由前面生成的 `wtest.txt` 中的内容去生成 `out1.txt` 文件

```
file(GENERATE OUTPUT out1.txt INPUT "${PROJECT_SOURCE_DIR}/wtest.txt")
```

由指定的内容生成 `out2.txt`

```
file(GENERATE OUTPUT out2.txt CONTENT "This is the out2.txt file")
```

由指定的内容生成 `out3.txt`, 加上条件控制, 用户可根据实际情况

用表达式判断是否需要生成文件, 这里只是演示, 直接是 1

```
file(GENERATE OUTPUT out3.txt CONTENT "This is the out3.txt file" CONDITION 1)
```

进入到 `build` 目录下执行 `cmake`:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt wtest.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake Makefile out1.txt out2.txt out3.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.11 执行 `cmake` 前后对比

执行完 `cmake` 之后会在 `build` 目录(也就是顶层源码的 `BINARY_DIR`)下生成了 `out1.txt`、`out2.txt` 和 `out3.txt` 三个文件, 内容如下:


```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ more out1.txt
Hello World! China
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ more out2.txt
This is the out2.txt file
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ more out3.txt
This is the out3.txt file
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.12 outX.txt 文件的内容

③、读文件：字节读取

file()读文件命令格式如下:

```
file(READ <filename> <variable>
      [OFFSET <offset>] [LIMIT <max-in>] [HEX])
```

从名为<filename>的文件中读取内容并将其存储在<variable>中。

可选择从给定的<offset>开始,最多读取<max-in>字节。HEX 选项使数据转换为十六进制表示(对二进制数据有用)。

同样,指定文件既可以使用相对路径、也可使用绝对路径,相对路径被解释为相对于当前源码路径。

测试代码如下:

```
# file()读文件测试
file(READ "${PROJECT_SOURCE_DIR}/wtest.txt" out_var)    #读取前面生成的 wtest.txt
message(${out_var})    # 打印输出

# 读取 wtest.txt 文件: 限定起始字节和大小
file(READ "${PROJECT_SOURCE_DIR}/wtest.txt" out_var OFFSET 0 LIMIT 10)
message(${out_var})

# 读取 wtest.txt 文件: 以二进制形式读取, 限定起始字节和大小,
file(READ "${PROJECT_SOURCE_DIR}/wtest.txt" out_var OFFSET 0 LIMIT 5 HEX)
message(${out_var})
```

打印信息如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello World! China
Hello Wor
48656c6c6f
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.13 cmake 打印信息

④、以字符串形式读取

命令格式如下所示:

```
file(STRINGS <filename> <variable> [<options>...])
```

从<filename>文件中解析 ASCII 字符串列表并将其存储在<variable>中。这个命令专用于读取字符串,会将文件中的二进制数据将被忽略,回车符(\r, CR)字符被忽略。

filename: 指定需要读取的文件, 可使用绝对路径、也可使用相对路径, 相对路径被解释为相对于当前源码路径。

variable: 存放字符串的变量。

options: 可选的参数, 可选择 0 个、1 个或多个选项, 这些选项包括:

- **LENGTH_MAXIMUM <max-len>**: 读取的字符串的最大长度;
- **LENGTH_MINIMUM <min-len>**: 读取的字符串的最小长度;
- **LIMIT_COUNT <max-num>**: 读取的行数;
- **LIMIT_INPUT <max-in>**: 读取的字节数;
- **LIMIT_OUTPUT <max-out>**: 存储到变量的限制字节数;
- **NEWLINE_CONSUME**: 把换行符也考虑进去;
- **NO_HEX_CONVERSION**: 除非提供此选项, 否则 Intel Hex 和 Motorola S-record 文件在读取时会自动转换为二进制文件。
- **REGEX <regex>**: 只读取符合正则表达式的行;
- **ENCODING <encoding-type>**: 指定输入文件的编码格式, 目前支持的编码有: UTF-8、UTF-16LE、UTF-16BE、UTF-32LE、UTF-32BE。如果未提供 ENCODING 选项并且文件具有字节顺序标记, 则 ENCODING 选项将默认为尊重字节顺序标记。

测试代码如下:

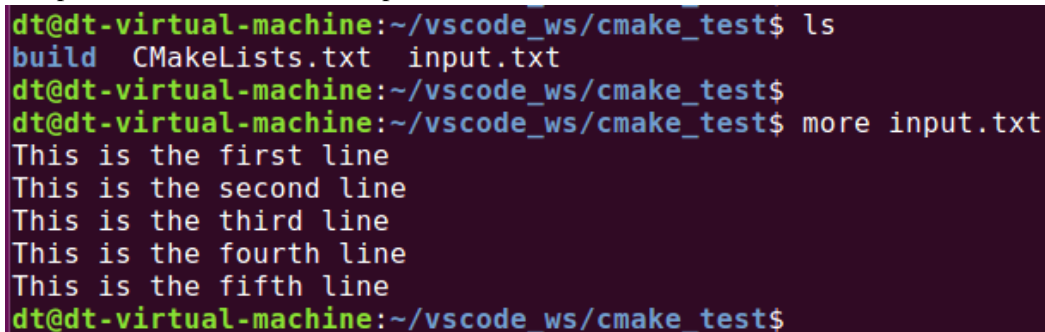
```
# 从 input.txt 文件读取字符串
file(STRINGS "${PROJECT_SOURCE_DIR}/input.txt" out_var)
message("${out_var}")

# 限定读取字符串的最大长度
file(STRINGS "${PROJECT_SOURCE_DIR}/input.txt" out_var LENGTH_MAXIMUM 5)
message("${out_var}")

# 限定读取字符串的最小长度
file(STRINGS "${PROJECT_SOURCE_DIR}/input.txt" out_var LENGTH_MINIMUM 4)
message("${out_var}")

# 限定读取行数
file(STRINGS "${PROJECT_SOURCE_DIR}/input.txt" out_var LIMIT_COUNT 3)
message("${out_var}")
```

从 input.txt 文件读取字符串, input.txt 文件的内容如下所示:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt input.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ more input.txt
This is the first line
This is the second line
This is the third line
This is the fourth line
This is the fifth line
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.5.14 input.txt 文件内容

上述代码执行的结果如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
This is the first line;This is the second line;This is the third line;This is the fourth line;This is the fifth line
This ;is th;e fir;st li;ne;This ;is th;e sec;ond l;ine;This ;is th;e thi;rd li;ne;This ;is th;e fou;rt h;line;This ;is th;e fif;th li;ne
This is the first line;This is the second line;This is the third line;This is the fourth line;This is the fifth line
This is the first line;This is the second line;This is the third line
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.15 cmake 打印信息

大家自己去对比就知道这些选项具体是什么意思了，这里便不再多说！

⑤、计算文件的 hash 值

file()命令可以计算指定文件内容的加密散列（hash 值）并将其存储在变量中。命令格式如下所示：

```
file(<MD5|SHA1|SHA224|SHA256|SHA384|SHA512> <filename> <variable>)
```

MD5|SHA1|SHA224|SHA256|SHA384|SHA512 表示不同的计算 hash 的算法，必须要指定其中之一，filename 指定文件（可使用绝对路径、也可使用相对路径，相对路径被解释为相对于当前源码的 BINARY_DIR），将计算结果存储在 variable 变量中。

测试代码如下：

```
# 计算文件的 hash 值
file(SHA256 "${PROJECT_SOURCE_DIR}/input.txt" out_var)
message("${out_var}")
```

这里我们还是用上面创建的 input.txt 文件，使用 SHA256 算法进行计算，结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt input.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
b688d7c58eb760303e430b26c971368be567cd80436c0e6160c0106786f12f28
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.16 cmake 打印信息

⑥、文件重命名

使用 file()命令可以对文件进行重命名操作，命令格式如下：

```
file(RENAME <oldname> <newname>)
```

oldname 指的是原文件，newname 指的是重命名后的新文件，文件既可以使用绝对路径指定，也可以使用相对路径指定，相对路径被解释为相对于当前源码路径。

测试代码：

```
# 文件重命名
file(RENAME "${PROJECT_SOURCE_DIR}/input.txt" "${PROJECT_SOURCE_DIR}/output.txt")
```

测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt input.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cd ..
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt output.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.5.17 file 重命名操作

⑦、删除文件

使用 `file()` 命令可以删除文件，命令格式如下：

```
file(REMOVE [<files>...])
```

```
file(REMOVE_RECURSE [<files>...])
```

`REMOVE` 选项将删除给定的文件，但不可以删除目录；而 `REMOVE_RECURSE` 选项将删除给定的文件或目录、以及非空目录。指定文件或目录既可以使用绝对路径、也可以使用相对路径，相对路径被解释为相对于当前源码路径。

测试代码：

```
# file 删除文件或目录测试
```

```
file(REMOVE "${PROJECT_SOURCE_DIR}/out1.txt")
```

```
file(REMOVE_RECURSE "${PROJECT_SOURCE_DIR}/out2.txt" "${PROJECT_SOURCE_DIR}/empty-dir"
"${PROJECT_SOURCE_DIR}/Non_empty-dir")
```

`out1.txt` 和 `out2.txt` 是普通文件，`empty-dir` 是一个空目录，而 `Non_empty-dir` 是一个非空目录，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ tree
.
├── build
├── CMakeLists.txt
├── empty-dir
├── Non_empty-dir
│   └── test.txt
├── out1.txt
└── out2.txt

3 directories, 4 files
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.5.18 目录结构图

进入到 `build` 目录下，执行 `cmake`：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt empty-dir Non_empty-dir out1.txt out2.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cd ..
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
```

图 32.5.19 执行 cmake

执行完 cmake 命令之后，这些文件以及文件夹都被删除了。

总结

关于 file()命令就给大家介绍这么多了，其实 file()命令的功能很强大，除了以上给大家介绍的基本功能外，还支持文件下载、文件锁等功能，大家有兴趣可以自己去了解。

32.5.4 设置交叉编译

前面笔者一直没给大家提过如何去设置交叉编译，因为如果不设置交叉编译，默认情况下，cmake 会使用主机系统（运行 cmake 命令的操作系统）的编译器来编译我们的工程，那么得到的可执行文件或库文件只能在 Ubuntu 系统运行，如果我们需要使得编译得到的可执行文件或库文件能够在我们的开发板（ARM 平台）上运行，则需要配置交叉编译，本小节将进行介绍。

前面我们已经安装了阿尔法 I.MX6U 硬件平台对应的交叉编译工具，如果大家还没安装交叉编译工具，可以参考“[开发板光盘资料 A-基础资料/【正点原子】I.MX6U 用户快速体验 V1.7.3.pdf](#)”文档中的第四章内容，根据文档的指示安装好交叉编译工具，当然如果你已经在 Ubuntu 系统下安装过了，就不用再次安装了。

我们使用的交叉编译器如下：

```
arm-poky-linux-gnueabi-gcc      #C 编译器
arm-poky-linux-gnueabi-g++     #C++编译器
```

其实配置交叉编译非常简单，只需要设置几个变量即可，如下所示：

```
# 配置 ARM 交叉编译
set(CMAKE_SYSTEM_NAME Linux)    #设置目标系统名字
set(CMAKE_SYSTEM_PROCESSOR arm) #设置目标处理器架构

# 指定编译器的 sysroot 路径
set(TOOLCHAIN_DIR /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots)
set(CMAKE_SYSROOT ${TOOLCHAIN_DIR}/cortexa7hf-neon-poky-linux-gnueabi)

# 指定交叉编译器 arm-gcc 和 arm-g++
set(CMAKE_C_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++)
```

```
# 为编译器添加编译选项
```

```
set(CMAKE_C_FLAGS "-march=armv7ve -mcpu=cortex-a7")
```

```
set(CMAKE_CXX_FLAGS "-march=armv7ve -mcpu=cortex-a7")
```

```
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
```

```
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
```

```
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

CMAKE_SYSTEM_NAME 变量在前面给大家介绍过,表示目标主机(譬如 ARM 开发板)的操作系统名称,这里将其设置为 Linux,表示目标操作系统是 Linux 系统。

CMAKE_SYSTEM_PROCESSOR 变量表示目标架构名称。

CMAKE_SYSROOT 变量前面也给大家介绍过,该变量的值会传递给 gcc 编译器的 --sysroot 选项,也就是 --sysroot=\${CMAKE_SYSROOT}, --sysroot 选项指定了编译器的 sysroot 目录,也就是编译器的系统根目录,编译过程中需要链接的库、头文件等,就会去该目录下寻找,譬如标准 C 库、标准 C 头文件这些。

CMAKE_C_COMPILER 变量指定了 C 语言编译器 gcc,由于是交叉编译,所以应该指定为 arm-gcc。

CMAKE_CXX_COMPILER 变量指定了 C++ 语言编译器 g++,由于是交叉编译,所以应该指定为 arm-g++。

CMAKE_C_FLAGS 变量为 gcc 编译器添加编译选项,CMAKE_CXX_FLAGS 变量为 g++ 编译器添加编译选项。

CMAKE_FIND_ROOT_PATH_MODE_LIBRARY 和 CMAKE_FIND_ROOT_PATH_MODE_INCLUDE 被设置为 ONLY; CMAKE_FIND_ROOT_PATH_MODE_INCLUDE 变量控制 CMAKE_SYSROOT 中的路径是否被 find_file() 和 find_path() 使用。如果设置为 ONLY,则只会搜索 CMAKE_SYSROOT 中的路径,如果设置为 NEVER,则 CMAKE_SYSROOT 中的路径将被忽略并且仅使用主机系统路径。如果设置为 BOTH,则将搜索主机系统路径和 CMAKE_SYSROOT 中的路径。

同理,CMAKE_FIND_ROOT_PATH_MODE_LIBRARY 变量控制 CMAKE_SYSROOT 中的路径是否被 find_library() 使用,如果设置为 ONLY,则只会搜索 CMAKE_SYSROOT 中的路径,如果设置为 NEVER,则 CMAKE_SYSROOT 中的路径将被忽略并且仅使用主机系统路径。如果设置为 BOTH,则将搜索主机系统路径和 CMAKE_SYSROOT 中的路径。

CMAKE_SYSROOT、CMAKE_C_COMPILER、CMAKE_CXX_COMPILER 这些变量涉及到交叉编译工具的安装路径,需要根据自己的实际安装路径来确定。

接着我们进行测试,譬如工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
└── main.c
```

main.c 源文件中调用了 printf() 函数打印了 “Hello World!” 字符串,CMakeLists.txt 文件内容如下:

```
# CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.5)
```

```
#####
```

```
# 配置 ARM 交叉编译
```

```
#####
```

```
set(CMAKE_SYSTEM_NAME Linux) #设置目标系统名字
```

```
set(CMAKE_SYSTEM_PROCESSOR arm) #设置目标处理器架构
```

```

# 指定编译器的 sysroot 路径
set(TOOLCHAIN_DIR /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots)
set(CMAKE_SYSROOT ${TOOLCHAIN_DIR}/cortexa7hf-neon-poky-linux-gnueabi)

# 指定交叉编译器 arm-linux-gcc 和 arm-linux-g++
set(CMAKE_C_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-
poky-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-
gnueabi/arm-poky-linux-gnueabi-g++)

# 为编译器添加编译选项
set(CMAKE_C_FLAGS "-march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7")
set(CMAKE_CXX_FLAGS "-march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
#####
# end
#####

project(HELLO)           #设置工程名称
add_executable(main main.c)

```

这里要注意，配置 ARM 交叉编译的这些代码需要放置在 `project()` 命令之前，否则不会生效！接着进入到 `build` 目录下，然后执行 `cmake`，此时笔者发现，执行 `cmake` 会报错！如下所示：

```

dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt main.c
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
-- The C compiler identification is GNU 5.3.0
-- The CXX compiler identification is GNU 5.3.0
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc -- broken
CMake Error at /usr/share/cmake-3.5/Modules/CMakeTestCCompiler.cmake:61 (message):
  The C compiler
  "/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc"
  is not able to compile a simple test program.

It fails with the following output:

Change Dir: /home/dt/vscode_ws/cmake_test/build/CMakeFiles/CMakeTmp

Run Build Command:"/usr/bin/make" "cmTC_fe43a/fast"
/usr/bin/make -f CMakeFiles/cmTC_fe43a.dir/build.make
CMakeFiles/cmTC_fe43a.dir/build

make[1]: Entering directory
'/home/dt/vscode_ws/cmake_test/build/CMakeFiles/CMakeTmp'
Building C object CMakeFiles/cmTC_fe43a.dir/testCCompiler.c.o

/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
--sysroot=/opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi
-o CMakeFiles/cmTC_fe43a.dir/testCCompiler.c.o -c
/home/dt/vscode_ws/cmake_test/build/CMakeFiles/CMakeTmp/testCCompiler.c

Linking C executable cmTC_fe43a
/usr/bin/cmake -E cmake_link_script CMakeFiles/cmTC_fe43a.dir/link.txt
--verbose=1

```

图 32.5.20 cmake 报错信息

一开始笔者也是死活想不明白，`CMakeLists.txt` 代码没问题呀，咋会报错？后来笔者实在没办法，尝试换一个高版本的 `cmake` 来运行，果然就没问题了；因为笔者用的是 Ubuntu 系统自带的 `cmake` 工具，前面也

给大家看了, 它的版本是 3.5.1, 当前最新 cmake 版本已经更新到了 3.22 了, 所以 3.5.1 这个版本可能确实是太旧了, 导致这里出错, 所以笔者建议大家去下载一个高版本的 cmake, 然后使用这个高版本的 cmake 工具, 不然会报错。

那怎么去下载高版本的 cmake, 其实非常简单, 我们首先进入到 cmake 的 GitHub 链接地址 <https://github.com/Kitware/CMake/releases>, 笔者并没有使用最新的 cmake, 而是使用了 3.16.0, 为了保持一致, 也建议大家下载这个版本, 往后翻页找到这个版本, 如下所示:

11 Dec 2019

robertmaynard

v3.16.1

4771c4e

Compare

v3.16.1

PGP sig by 2D2CEF1034921684

Assets 14

26 Nov 2019

robertmaynard

v3.16.0

1b4482f

Compare

v3.16.0

PGP sig by 2D2CEF1034921684

Assets 14

cmake-3.16.0-Darwin-x86_64.dmg	34.9 MB
cmake-3.16.0-Darwin-x86_64.tar.gz	34.1 MB
cmake-3.16.0-Linux-x86_64.sh	37.7 MB
cmake-3.16.0-Linux-x86_64.tar.gz	37.7 MB
cmake-3.16.0-SHA-256.txt	932 Bytes
cmake-3.16.0-SHA-256.txt.asc	833 Bytes
cmake-3.16.0-win32-x86.msi	21 MB
cmake-3.16.0-win32-x86.zip	28.3 MB
cmake-3.16.0-win64-x64.msi	22.9 MB
cmake-3.16.0-win64-x64.zip	31 MB
cmake-3.16.0.tar.gz	8.69 MB

图 32.5.21 cmake 3.16.0 版本

这里我们下载 cmake-3.16.0-Linux-x86_64.tar.gz 压缩包文件, 这个不是 cmake 的源码工程, 而是可以在 x86-64 的 Linux 系统下运行的可执行程序, 其中就包括了 cmake 工具, 所以我们下载这个即可, 非常方便, 都不用自己编译!

下载成功之后将其拷贝到 Ubuntu 系统的用户家目录下, 并将其解压到某个目录, 解压之后生成 cmake-3.16.0-Linux-x86_64 文件夹, 这里笔者选择将其解压到家目录下的 tools 目录中, 如下所示:

```
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ pwd
/home/dt/tools
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 cmake-3.5.1 freetype jpeg libsocketcan-0.0.12 png tslib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$
```

图 32.5.22 解压

cmake 工具就在 cmake-3.16.0-Linux-x86_64/bin 目录下。

现在重新进入到我们的工程目录下, 进入到 build 目录执行 cmake, 如下所示:


```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt main.c
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake ..
-- The C compiler identification is GNU 5.3.0
-- The CXX compiler identification is GNU 5.3.0
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++
-- Check for working CXX compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.23 使用高版本 cmake 执行

接着执行 make 命令编译:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ make
Scanning dependencies of target main
[ 50%] Building C object CMakeFiles/main.dir/main.c.o
[100%] Linking C executable main
[100%] Built target main
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake main Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ file main
main: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter
b92f99289c426, not stripped
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.24 编译

编译生成的 main 可执行文件, 通过 file 命令查看可知, 它是一个 ARM 架构的可执行程序, 可以把它拷贝到开发板上去运行, 肯定是没有问题的, 这里就不再演示了。

上例中的这种交叉编译配置方式自然是没有问题的, 但是不规范, 通常的做法是, 将这些配置项(也就是变量的设置)单独拿出来写在一个单独的配置文件中, 而不直接写入到 CMakeLists.txt 源码中, 然后在执行 cmake 命令时, 指定配置文件给 cmake, 让它去配置交叉编译环境。

如何指定配置文件呢? 通过如下方式:

```
cmake -DCMAKE_TOOLCHAIN_FILE=cfg_file_path ..
```

通过 -DCMAKE_TOOLCHAIN_FILE 选项指定配置文件, -D 是 cmake 命令提供的一个选项, 通过该选项可以创建一个缓存变量(缓存变量就是全局变量, 在整个工程中都是生效的, 会覆盖 CMakeLists.txt 源码中定义的同名变量), 所以 -DCMAKE_TOOLCHAIN_FILE 其实就是设置了缓存变量 CMAKE_TOOLCHAIN_FILE, 它的值就是“=”号后面的内容, cmake 会执行 CMAKE_TOOLCHAIN_FILE 变量所指定的源文件, 对交叉编译进行设置; 现在我们进行测试, 在工程源码目录下创建一个配置文件 arm-linux-setup.cmake, 内容如下:

```
#####
# 配置 ARM 交叉编译
#####
set(CMAKE_SYSTEM_NAME Linux)    #设置目标系统名字
set(CMAKE_SYSTEM_PROCESSOR arm) #设置目标处理器架构

# 指定编译器的 sysroot 路径
set(TOOLCHAIN_DIR /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots)
set(CMAKE_SYSROOT ${TOOLCHAIN_DIR}/cortexa7hf-neon-poky-linux-gnueabi)
```

```
# 指定交叉编译器 arm-linux-gcc 和 arm-linux-g++
set(CMAKE_C_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-
poky-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER      ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-
gnueabi/arm-poky-linux-gnueabi-g++)

# 为编译器添加编译选项
set(CMAKE_C_FLAGS "-march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7")
set(CMAKE_CXX_FLAGS "-march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
#####
# end
#####
```

此时 CMakeLists.txt 文件内容如下（剔除了交叉编译的配置项）：

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.5)
project(HELLO)
add_executable(main main.c)
```

此时工程目录结构如下所示：

```
├── arm-linux-setup.cmake
├── build
├── CMakeLists.txt
└── main.c
```

进入到 build 目录下，执行 cmake：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake -DCMAKE_TOOLCHAIN_FILE=./arm-linux-setup.cmake ..
-- The C compiler identification is GNU 5.3.0
-- The CXX compiler identification is GNU 5.3.0
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.25 执行 cmake

接着执行 make 编译：

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ make
Scanning dependencies of target main
[ 50%] Building C object CMakeFiles/main.dir/main.c.o
[100%] Linking C executable main
[100%] Built target main
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake main Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ file main
main: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
b92f99289c426, not stripped
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.26 make 编译

所以这种方式也是没有问题的,推荐使用这种方式配置交叉编译,而不是直接写入到 CMakeLists.txt 源码中。

32.5.5 变量的作用域

如同 C 语言一样,在 cmake 中,变量也有作用域的概念,本小节我们就来聊一聊关于 cmake 中变量作用域的问题。

本小节从三个方面进行介绍:函数作用域、目录作用域以及全局作用域。

一、函数作用域 (function scope)

我把这个作用域叫做函数作用域,当在函数内通过 set 将变量 var 与当前函数作用域绑定时,变量 var 仅在函数作用域内有效,出了这个作用域,如果这个作用域外也有同名的变量 var,那么使用的将是域外同名变量 var; func1() 内部调用 func2(), 嵌套调用的函数 func2() 内部如果也引用变量 var, 那么该变量 var 应该是 func1() 内部定义的变量,如果有的话;如果 func1() 内部没有绑定变量 var, 那么就会使用 func1() 作用域外定义的变量 var, 依次向外搜索。

以上这段话大家可能不好理解,我们通过几个示例来看看函数作用域。

①、函数内部引用函数外部定义的变量

示例代码如下所示:

```
# 函数 xyz
function(xyz)
    message(${ABC}) #引用变量 ABC
endfunction()

set(ABC "Hello World") #定义变量 ABC

xyz() # 调用函数
```

ABC 是函数外部定义的一个变量,在函数 xyz 中引用了该变量,打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello World
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.27 cmake 打印信息

所以可知, 函数内可以引用函数外部定义的变量。

②、函数内定义的变量是否可以被外部引用

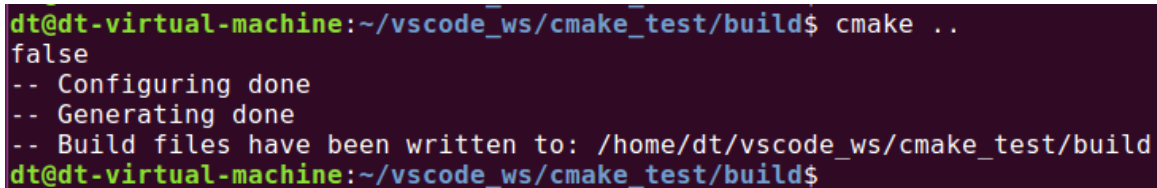
示例代码如下所示:

```
# 函数 xyz
function(xyz)
    set(ABC "Hello World")#定义变量 ABC
endfunction()

xyz()      # 调用函数

if(DEFINED ABC)
    message("true")
    message("${ABC}")    #引用函数内定义的变量 ABC
else()
    message("false")
endif()
```

函数内定义了变量 ABC, 外部调用函数之后, 通过 if(DEFINED ABC)来判断变量 ABC 是否有定义, 如果定义了该变量打印 true 并将变量打印出来, 如果没有定义该变量则打印 false。测试结果如下:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
false
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.28 cmake 打印信息

所以可知, 函数内部定义的变量仅在函数内部可使用, 出了函数之后便无效了, 这其实跟 C 语言中差不多, 函数中定义的变量可以认为是局部变量, 外部自然是无法去引用的。

③、函数内定义与外部同名的变量

测试代码如下所示:

```
# 函数 xyz
function(xyz)
    message("函数内部")
    message("${ABC}")
    set(ABC "Hello China!")#设置变量 ABC
    message("${ABC}")
endfunction()

set(ABC "Hello World!")#定义变量 ABC
xyz()      # 调用函数
message("函数外部")
message("${ABC}")
```

在这段代码中, 我们在函数外定义了变量 ABC="Hello World!", 在函数内去设置变量 ABC="Hello China!", 函数执行完之后, 在外部调用 message()打印变量 ABC。如果按照 C 语言中的理解, 那么函数外部

打印 ABC 变量的值应该等于"Hello China!" (大家不要去关注变量的定义是否需要放在函数定义之前, 这种解释性脚本语言是没有类似于 C 语言中申明这种概念的, 函数虽然定义了, 但是调用函数是在定义变量之后的), 但事实是不是这样呢, 我们来看看打印信息:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
函数内部
Hello World!
Hello China!
函数外部
Hello World!
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.29 cmake 打印信息

从打印信息可知, 事实并非我们上面所假设那样, 函数内调用 set 去设置变量 ABC, 并不是设置了外部变量 ABC 的值, 而是在函数新建了一个变量 ABC, 这个与 C 语言是不一样的, 跟 Python 很像, 如果大家学过 Python 的话应该就知道。

所以函数内部的代码中, 调用 set 之前, 引用了变量 ABC, 此时它会搜索函数内是否定义了该变量, 如果没有, 它会向外搜索, 结果就找到了外部定义的变量 ABC, 所以函数内部的第一条打印信息是"Hello World!"; 调用 set 之后, 函数内也创建了一个变量 ABC, 此时再次引用 ABC 将使用函数内定义的变量, 而非是外部定义的变量, 所以第二条打印信息是"Hello China!"。

④、函数内如何设置外部定义的变量

那如果需要在函数内修改外部定义的变量, 该如何做呢? 譬如下面这段代码:

```
# 函数 xyz
function(xyz)
    set(ABC "Hello China!")
endfunction()

set(ABC "Hello World!")
xyz()      # 调用函数
message("${ABC}")
```

通过前面的介绍可知, xyz()函数内通过 set 只是创建了一个在函数内部使用的变量 ABC, 而并非是要去修改外部定义的变量 ABC, 那如何能使得函数内可以去修改外部定义的变量呢? 其实也非常简单, set 命令提供了一个可选项 PARENT_SCOPE, 只需在调用 set 命令时在参数列表末尾加上 PARENT_SCOPE 关键字即可, 如下所示:

```
# 函数 xyz
function(xyz)
    set(ABC "Hello China!" PARENT_SCOPE)  #加上 PARENT_SCOPE
endfunction()

set(ABC "Hello World!")
xyz()      # 调用函数
message("${ABC}")
```

再来看看打印信息:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
Hello China!
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.30 cmake 打印信息

打印信息证明, 加上 PARENT_SCOPE 之后确实可以, 那 PARENT_SCOPE 选项究竟是什么?

官方给出的解释是这样的: 如果添加了 PARENT_SCOPE 选项, 则变量将设置在当前作用域范围之上的作用域范围内, 每个目录(在这里“目录”指的是包含了 CMakeLists.txt 的目录)或函数都会创建一个新作用域, 此命令会将变量的值设置到父目录或上层调用函数中(函数嵌套的情况下)。

这是什么意思呢? 其实就是说, 如果 set 命令添加了 PARENT_SCOPE 选项, 那就意味着并不是在当前作用域(set 命令所在作用域)内设置这个变量, 而是在当前作用域的上一层作用域(父作用域)中设置该变量; 当前作用域的上一层作用域该怎么理解呢? 这个根据具体的情况而定, 下面举几个例子进行说明。

示例代码 1:

示例代码 32.5.1 示例代码 1

```
# 函数 xyz
function(xyz)
    set(ABC "Hello China!" PARENT_SCOPE)    #加上 PARENT_SCOPE
endfunction()

set(ABC "Hello World!")
xyz()    # 调用函数
message("${ABC}")
```

在这个例子中, 函数 xyz 中调用 set 时添加了 PARENT_SCOPE 选项, 意味着会在函数 xyz 的上一层作用域中设置 ABC 变量, 函数的上一层作用域也就是调用 xyz() 函数时所在的作用域, 也就是当前源码对应的作用域(当前目录作用域)。

示例代码 2:

```
# 函数 func2
function(func2)
    set(ABC "Hello People!" PARENT_SCOPE)
endfunction()

# 函数 func1
function(func1)
    set(ABC "Hello China!")
    func2()
endfunction()

set(ABC "Hello World!")
func1()
message("${ABC}")
```

在这个示例中, 函数 func1 中调用了 func2, 那么函数 func2 的上一层作用域就是 func1 函数对应的作用域。

示例代码 3:

有如下工程目录结构:

```

├── build
├── CMakeLists.txt
└── src
    └── CMakeLists.txt

```

顶层 CMakeLists.txt 文件内容如下:

```

# CMakeLists.txt
cmake_minimum_required(VERSION 3.5)
project(TEST)

add_subdirectory(src)
xyz()
message("${ABC}")

```

顶层源码调用 src 目录下的子源码, 子源码下定义了一个函数 xyz, 如下所示:

```

# src 下的 CMakeLists.txt
function(xyz)
    set(ABC "Hello World!" PARENT_SCOPE)
endfunction()

```

在这种情况下, 函数 xyz 的上一层作用域便是顶层目录作用域 (顶层源码作用域), 关键是看“谁”调用该函数。

同理下面这种情况也是如此:

顶层 CMakeLists.txt 文件:

```

# CMakeLists.txt
cmake_minimum_required(VERSION 3.5)
project(TEST)

add_subdirectory(src)
message("${ABC}")

```

src 目录下的 CMakeLists.txt 文件:

```

# src 下的 CMakeLists.txt
set(ABC "Hello World!" PARENT_SCOPE)

```

变量 ABC 会在顶层源码中被设置, 而不是 set 命令所在的作用域中。

⑤、函数的返回值如何实现?

前面给大家介绍函数的时候提到过, cmake 中函数也可以有返回值, 但是不能通过 return() 命令来实现, 由于当时没介绍 PARENT_SCOPE, 所以没法给大家讲解如何去实返回值, 现在我们已经知道了 PARENT_SCOPE 选项的作用, 其实就是通过这个选项来实现函数的返回值功能。

先来看个示例:

```

# 顶层 CMakeLists.txt
cmake_minimum_required(VERSION 3.5)
project(TEST)

# 定义一个函数 xyz

```

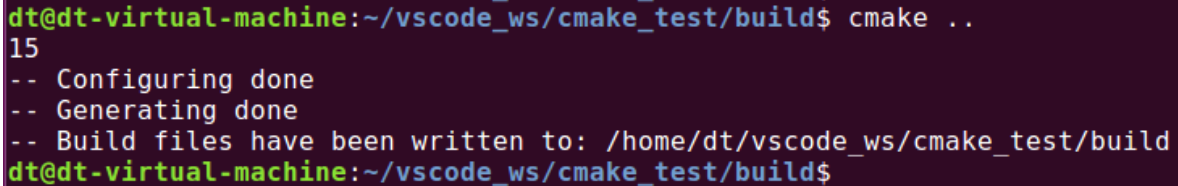
实现两个数相加, 并将结果通过 out 参数返回给调用者

```
function(xyz out var1 var2)
    math(EXPR temp "${var1} + ${var2}")
    set(${out} ${temp} PARENT_SCOPE)
endfunction()
```

```
xyz(out_var 5 10)
```

```
message("${out_var}")
```

打印结果如下:



```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
15
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.31 cmake 打印信息

看到这里不知道大家明白了没, 其实很简单, 调用 xyz() 函数时, 传入的 out_var 是作为一个参数传入进去的, 而不是变量名, 但现在需要将其变成一个变量名, 怎么做呢? 那就是在函数中获取参数 out 的值, 将参数 out 的值作为变量名, 然后用 set 创建该变量, 并添加了 PARENT_SCOPE 选项。所以通过 message 便可以打印出该变量, 因为这个变量在源码中定义了。

二、目录作用域 (Directory Scope)

我把这个作用域叫做目录作用域。子目录会将父目录的所有变量拷贝到当前 CMakeLists.txt 源码中, 当前 CMakeLists.txt 中的变量的作用域仅在当前目录有效。

目录作用域有两个特点: 向下有效 (上层作用域中定义的变量在下层作用域中是有效的), 值拷贝。举个栗子来进一步阐述!

譬如目录结构如下所示:

```
├── CMakeLists.txt
└── sub_dir
    ├── CMakeLists.txt
```

父目录 CMakeLists.txt 文件内容如下:

```
# 父源码
cmake_minimum_required(VERSION 3.5)
project(TEST)

set(parent_var "Hello parent")
message("parent-<parent_var>: ${parent_var}")
add_subdirectory(sub_dir)
message("parent-<parent_var>: ${parent_var}")
```

在父源码中, 我们定义了一个变量 parent_var, 并将其设置为 "Hello parent"。

子源码 CMakeLists.txt 内容:

```
message("subdir-<parent_var>: ${parent_var}")
set(parent_var "Hello child")
message("变量修改之后")
```



```
message("subdir-<parent_var>: ${parent_var}")
```

在子源码中,第 1 行打印了 `parent_var` 变量,这个变量是由父源码所创建的,由于变量向下有效,所以在子源码中也可以使用;第 2 行,我们去修改 `parent_var` 变量,将其设置为"Hello child",但这是子源码新建的一个变量,并没改变父源码中的 `parent_var` 变量,也就是说这里的 `set` 并不影响父源码中的 `parent_var` 变量,仅仅只是改变了子源码中的 `parent_var` 变量,这就是值拷贝的含义(子源码从父源码中拷贝了一份变量,副本)。

执行结果如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
parent-<parent_var>: Hello parent
subdir-<parent_var>: Hello parent
变量修改之后
subdir-<parent_var>: Hello child
parent-<parent_var>: Hello parent
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.32 cmake 打印信息

三、全局作用域 (Persistent Cache 持久缓存、缓存变量)

缓存变量在整个 `cmake` 工程的编译生命周期内都有效,所以这些变量的作用域是全局范围的,工程内的其他任意目录都可以访问缓存变量,注意 `cmake` 是从上到下来解析 `CMakeLists.txt` 文件的。

缓存变量可以通过 `set` 命令来定义,使用 `set` 命令时添加 `CACHE` 选项来实现;除此之外,还有其它多种方式可以定义缓存变量,譬如前面给大家介绍的 `cmake -D` 选项是经常用来定义缓存变量的方法,`cmake -DXXX`,就表示创建了一个名为 `XXX` 的全局变量;关于缓存变量笔者就不过多的介绍了,有兴趣的读者可以自己去研究下。

32.5.6 属性

本小节简单地向大家介绍一下 `cmake` 中的属性相关的概念。

属性大概可以分为多种:全局属性、目录属性(源码属性)、目标属性以及其它一些分类。在 <https://cmake.org/cmake/help/v3.5/manual/cmake-properties.7.html> 中有详细介绍。如下:

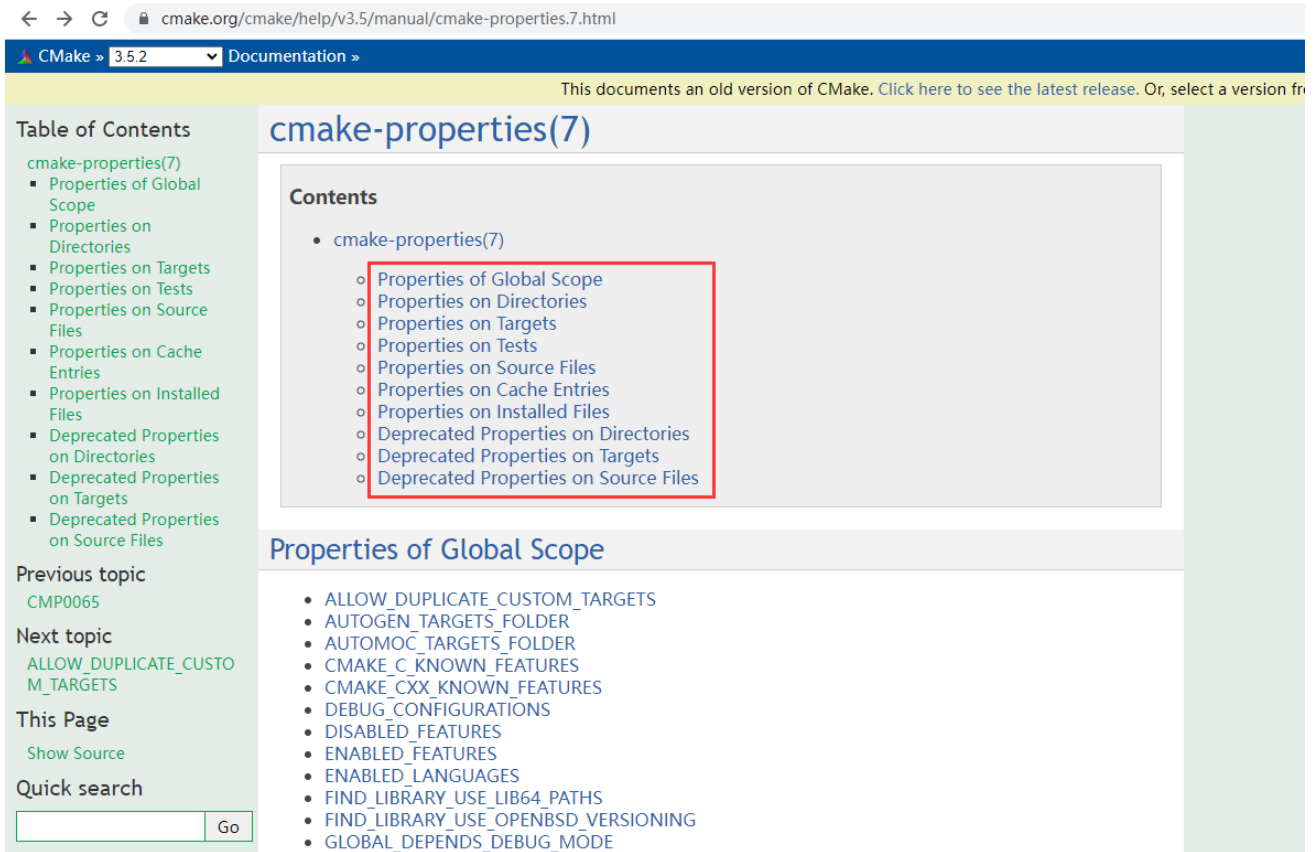


图 32.5.33 属性介绍

属性会影响到一些行为，这里重点给大家介绍下目录属性和目标属性，其它的大家自己去看。

一、目录属性

目录属性其实就是 CMakeLists.txt 源码的属性，来看看有哪些：

Properties on Directories

- ADDITIONAL_MAKE_CLEAN_FILES
- CACHE_VARIABLES
- CLEAN_NO_CUSTOM
- CMAKE_CONFIGURE_DEPENDS
- COMPILE_DEFINITIONS
- COMPILE_OPTIONS
- DEFINITIONS
- EXCLUDE_FROM_ALL
- IMPLICIT_DEPENDS_INCLUDE_TRANSFORM
- INCLUDE_DIRECTORIES
- INCLUDE_REGULAR_EXPRESSION
- INTERPROCEDURAL_OPTIMIZATION_<CONFIG>
- INTERPROCEDURAL_OPTIMIZATION
- LINK_DIRECTORIES
- LISTFILE_STACK
- MACROS
- PARENT_DIRECTORY
- RULE_LAUNCH_COMPILE
- RULE_LAUNCH_CUSTOM
- RULE_LAUNCH_LINK
- TEST_INCLUDE_FILE
- VARIABLES
- VS_GLOBAL_SECTION_POST_<section>
- VS_GLOBAL_SECTION_PRE_<section>

图 32.5.34 目录属性

这里我们随便挑几个来讲解:

CACHE_VARIABLES

当前目录中可用的缓存变量列表。

CLEAN_NO_CUSTOM

如果设置为 true 以告诉 Makefile Generators 在 make clean 操作期间不要删除此目录的自定义命令的输出文件。如何获取或设置属性稍后再给大家介绍。

INCLUDE_DIRECTORIES

此属性是目录的头文件搜索路径列表，其实就是 include_directories() 命令所添加的目录，include_directories() 命令会将指定的目录添加到 INCLUDE_DIRECTORIES 属性中，所以 INCLUDE_DIRECTORIES 属性其实就是一个头文件搜索路径列表。

测试代码如下：

```
# 父源码
cmake_minimum_required(VERSION 3.5)
project(TEST)

#获取目录的 INCLUDE_DIRECTORIES 属性
get_directory_property(out_var INCLUDE_DIRECTORIES)
message("${out_var}")

#调用 include_directories 添加头文件搜索目录
```

```
include_directories(include)
```

```
#再次获取 INCLUDE_DIRECTORIES 属性
```

```
get_directory_property(out_var INCLUDE_DIRECTORIES)
```

```
message("${out_var}")
```

```
#再次调用 include_directories, 将目录放在列表前面
```

```
include_directories(BEFORE hello)
```

```
#再次获取 INCLUDE_DIRECTORIES 属性
```

```
get_directory_property(out_var INCLUDE_DIRECTORIES)
```

```
message("${out_var}")
```

本例中, 使用了 `get_directory_property()` 命令, 该命令用于获取目录的属性, 使用方法如下:

```
get_directory_property(<variable> [DIRECTORY <dir>] <prop-name>)
```

将属性的值存储在 `variable` 变量中; 第二个参数是一个可选参数, 可指定一个目录, 如果不指定, 则默认是当前源码所在目录; 第三个参数 `prop-name` 表示对应的属性名称。

上述代码的打印信息如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
/home/dt/vscode_ws/cmake_test/include
/home/dt/vscode_ws/cmake_test/hello;/home/dt/vscode_ws/cmake_test/include
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.35 cmake 打印信息

第一个 `message` 打印的是空信息, 说明此时 `INCLUDE_DIRECTORIES` 是空的, 没有添加任何目录。`include_directories()` 命令默认将目录添加到 `INCLUDE_DIRECTORIES` 列表的末尾, 可显式指定 `BEFORE` 或 `AFTER` 将目录添加到列表的前面或后面, 在前面给大家介绍过。

既然如此, 那是不是可以直接去设置 `INCLUDE_DIRECTORIES` 属性来添加头文件搜索目录, 而不使用 `include_directories()` 命令来添加? 这样当然是可以的, 可以使用 `set_directory_properties()` 命令设置目录属性, 如下所示:

```
set_directory_properties(PROPERTIES prop1 value1 prop2 value2)
```

接下来进行测试, 假设工程目录结构如下所示:

```
├── build
├── CMakeLists.txt
├── include
│   └── hello.h
└── main.c
```

源文件 `main.c` 中包含了 `hello.h` 头文件, `hello.h` 头文件在 `include` 目录下, `CMakeLists.txt` 如下:

```
# 父源码
```

```
cmake_minimum_required(VERSION 3.5)
```

```
project(TEST)
```

```
set_directory_properties(PROPERTIES INCLUDE_DIRECTORIES /home/dt/vscode_ws/cmake_test/include)
```

```
get_directory_property(out_var INCLUDE_DIRECTORIES)
```

```
message("${out_var}")
```

```
add_executable(main main.c)
```

进入到 build 目录下, 执行 cmake、make 构建、编译:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ ls
build CMakeLists.txt include main.c
dt@dt-virtual-machine:~/vscode_ws/cmake_test$
dt@dt-virtual-machine:~/vscode_ws/cmake_test$ cd build/
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
/home/dt/vscode_ws/cmake_test/include
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ make
Scanning dependencies of target main
[ 50%] Building C object CMakeFiles/main.dir/main.c.o
[100%] Linking C executable main
[100%] Built target main
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake main Makefile
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.36 cmake 打印信息

编译成功, 说明这种做法是没有问题的, 需要注意的是, 调用 `set_directory_properties()` 命令设置属性时, 需要使用绝对路径。

父目录的 `INCLUDE_DIRECTORIES` 属性可以初始化、填充子目录的 `INCLUDE_DIRECTORIES` 属性, 测试如下:

譬如工程目录结构如下:

```
├── CMakeLists.txt
└── subdir
    └── CMakeLists.txt
```

父源码内容如下所示:

```
# 父源码
cmake_minimum_required(VERSION 3.5)
project(TEST)

#调用 include_directories 添加 2 个目录
include_directories(include hello)
get_directory_property(p_list INCLUDE_DIRECTORIES)
message("${p_list}")

#调用子源码
add_subdirectory(subdir)
```

子源码内容:

```
#子源码
```

```
get_directory_property(c_list INCLUDE_DIRECTORIES)
```

```
message("${c_list}")
```

打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
/home/dt/vscode_ws/cmake_test/include;/home/dt/vscode_ws/cmake_test/hello
/home/dt/vscode_ws/cmake_test/include;/home/dt/vscode_ws/cmake_test/hello
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.37 cmake 打印信息

LINK_DIRECTORIES

此属性是目录的库文件搜索路径列表, 其实就是 `link_directories()` 命令所添加的目录, `link_directories` 命令会将指定的目录添加到 `LINK_DIRECTORIES` 属性中, 所以 `LINK_DIRECTORIES` 属性其实就是一个库文件搜索路径列表。

测试如下:

```
# 父源码
```

```
cmake_minimum_required(VERSION 3.5)
```

```
project(TEST)
```

```
#获取目录的 LINK_DIRECTORIES 属性
```

```
get_directory_property(out_var LINK_DIRECTORIES)
```

```
message("${out_var}")
```

```
#添加库文件搜索目录
```

```
link_directories(include hello)
```

```
get_directory_property(out_var LINK_DIRECTORIES)
```

```
message("${out_var}")
```

打印信息如下:

```
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$ cmake ..
/home/dt/vscode_ws/cmake_test/include;/home/dt/vscode_ws/cmake_test/hello
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/cmake_test/build
dt@dt-virtual-machine:~/vscode_ws/cmake_test/build$
```

图 32.5.38 cmake 打印信息

同样, 父目录的 `LINK_DIRECTORIES` 属性可以初始化、填充子目录的 `LINK_DIRECTORIES` 属性。也直接去设置 `LINK_DIRECTORIES` 属性来添加库文件搜索目录, 而不使用 `link_directories()` 命令来添加, 大家可以去测试一下, 这里不再演示了!

MACROS

当期目录中可用的宏命令列表。

PARENT_DIRECTORY

加载当前子目录的源目录, 其实就是说, 当前源码的父源码所在路径, 对于顶级目录, 该值为空字符串。此属性只读、不可修改。

VARIABLES

当前目录中定义的变量列表。

只读属性、不可修改!

二、目标属性

目标属性, 顾名思义就是目标对应的属性, 如下:



图 32.5.39 目标属性

目标属性可通过 `get_target_property`、`set_target_property` 命令获取或设置。

目标属性比较多, 这里挑几个给大家介绍下:

BINARY_DIR

只读属性, 定义目标的目录中 `CMAKE_CURRENT_BINARY_DIR` 变量的值。

SOURCE_DIR

只读属性, 定义目标的目录中 CMAKE_CURRENT_SOURCE_DIR 变量的值。

INCLUDE_DIRECTORIES

目标的头文件搜索路径列表, target_include_directories()命令会将目录添加到 INCLUDE_DIRECTORIES 列表中, INCLUDE_DIRECTORIES 会拷贝目录属性中的 INCLUDE_DIRECTORIES 属性作为初始值。

INTERFACE_INCLUDE_DIRECTORIES

target_include_directories()命令使用 PUBLIC 和 INTERFACE 关键字的值填充此属性。

INTERFACE_LINK_LIBRARIES

target_link_libraries()命令使用 PUBLIC 和 INTERFACE 关键字的值填充此属性。

LIBRARY_OUTPUT_DIRECTORY

LIBRARY_OUTPUT_NAME

LINK_LIBRARIES

目标的链接依赖库列表。

OUTPUT_NAME

目标文件的输出名称。

TYPE

目标的类型, 它将是 STATIC_LIBRARY、MODULE_LIBRARY、SHARED_LIBRARY、INTERFACE_LIBRARY、EXECUTABLE 之一或内部目标类型之一。

32.6 总结

关于 cmake, 笔者就给大家介绍这么多, 已经基本够用了; 其实还有很多的命令以及变量没有介绍到, 但完全可以自己去看, 然后去实战测试!

第三十三章 实战小项目之 MQTT 物联网

物联网曾被认为是继计算机、互联网之后,信息技术行业的第三次浪潮。随着基础通讯设施的不断完善,尤其是 5G 的出现,进一步降低了万物互联的门槛和成本。物联网本身也是 AI 和区块链应用很好的落地场景之一,各大云服务商也在纷纷上架物联网平台和服务。

物联网通讯是物联网的一个核心内容,目前物联网的通讯协议并没有一个统一的标准,比较常见的有 MQTT、CoAP、DDS、XMPP 等,在这其中, MQTT (消息队列遥测传输协议)应该是应用最广泛的标准之一。目前, MQTT 已逐渐成为 IoT 领域最热门的协议,也是国内外各大物联网平台最主流的传输协议,阿里云 IoT 物联网平台很多设备都是通过 MQTT 接入。

本章我们将从最基础的知识开始,向您讲解 MQTT 协议的应用,通过本章的学习,我们将一起在开发板上实现一个简单地物联网小项目,实现远程控制开发板上的外设,譬如 LED、蜂鸣器;亦或者远程获取开发板运行的状态信息。

本章将会讨论如下主题内容。

- MQTT 是什么?
- MQTT 协议介绍
- MQTT 客户端库移植到开发板
- 如何开发 MQTT 客户端应用程序

33.1 MQTT 简介

《MQTT 协议规范中文版》一书中对 MQTT (Message Queuing Telemetry Transport, 消息队列遥测传输) 进行了描述:

MQTT 是一种基于客户端服务端架构的发布/订阅模式的消息传输协议。它的设计思想是轻巧、开放、简单、规范,易于实现。这些特点使得它对很多场景来说都是很好的选择,特别是对于受限的环境如机器与机器的通信 (M2M) 以及物联网环境 (IoT)。---MQTT 协议中文版

以上这段话很好的描述了 MQTT 的全部含义,它是一种轻巧、开放、简单、规范的网络通信协议。与 HTTP 协议一样, MQTT 协议也是应用层协议,工作在 TCP/IP 四层模型中的最上层(应用层),构建于 TCP/IP 协议上。MQTT 最大优点在于,可以以极少的代码和有限的带宽,为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议,使其在物联网、小型设备、移动应用等方面有较广泛的应用。

如今, MQTT 成为了最受欢迎的物联网协议,已广泛应用于车联网、智能家居、即时聊天应用和工业互联网等领域。目前通过 MQTT 协议连接的设备已经过亿,这些都得益于 MQTT 协议为设备提供了稳定、可靠、易用的通信基础。

33.1.1 MQTT 的主要特性

MQTT 协议是为工作在低带宽、不可靠网络的远程传感器和控制设备之间的通讯而设计的协议,它具有以下主要的几项特性:

- ①、使用发布/订阅消息模式,提供一对多的消息发布,解除应用程序耦合。
- ②、基于 TCP/IP 提供网络连接。

主流的 MQTT 是基于 TCP 连接进行数据推送的,但是同样也有基于 UDP 的版本,叫做 MQTT-SN。这两种版本由于基于不同的连接方式,优缺点自然也就各有不同了。

- ③、支持 QoS 服务质量等级。

根据消息的重要性不同设置不同的服务质量等级。

- ④、小型传输,开销很小,协议交换最小化,以降低网络流量。

这就是为什么在介绍里说它非常适合"在物联网领域,传感器与服务器的通信,信息的收集",要知道嵌入式设备的运算能力和带宽都相对薄弱,使用这种协议来传递消息再适合不过了,在手机移动应用方面, MQTT 是一种不错的 Android 消息推送方案。

- ⑤、使用 will 遗嘱机制来通知客户端异常断线。
- ⑥、基于主题发布/订阅消息,对负载内容屏蔽的消息传输。
- ⑦、支持心跳机制。

以上便是 MQTT 的主要特性了。

33.1.2 MQTT 历史

MQTT 协议最初版本是在 1999 年建立的,该协议的发明人是的 Andy Stanford-Clark 和 Arlen Nipper。



图 33.1.1 MQTT 协议发明人之一 Andy-Stanford-Clark



图 33.1.2 MQTT 协议发明人之一 Arlen Nipper

MQTT 最初是用于石油管道的传感器与卫星之间数据传输。他们当时正在开发一个利用卫星通讯监控输油管道的项目，为了实现这个项目要求，他们需要开发一种用于嵌入式设备的通讯协议，这种通讯协议必须满足以下条件：

- 易于实现，服务器必须要实现成千上万个客户端的接入
- 数据传输的服务质量可控，根据数据的重要性和特性，设置不同等级的服务质量
- 占用带宽小，单次数据量小，但不能出错
- 必须能够适应高延迟、掉线、断网等网络通信不可靠的风险
- 设备连接状态可知，云端与设备端保持长连接

通过以上几个条件可知：

- MQTT 服务器可以连接大量的远程传感器和控制设备, 与远程客户端保持长连接, 具有一定的实时性。
- 云端向设备端发送消息, 设备端可以在最短的时间内接收到并作出回应。
- MQTT 更适合需要实时控制的场合, 尤其适合执行器。
- 云端与客户端需要保持长连接, 要能够获取到设备的连接状态, 就需要时不时地发送心跳包, 这就不会省电, 所以, MQTT 并不适合低功耗场合。

从以上几点不难看出, MQTT 从诞生之初就是专为低带宽、高延迟或不可靠的网络而设计的。虽然历经几十年的更新和变化, 以上这些特点仍然是 MQTT 协议的核心特点。但是与最初不同的是, MQTT 协议已经从嵌入式系统应用拓展到开放的物联网 (IoT) 领域。

33.1.3 MQTT 版本

目前 MQTT 主流版本有两个, 分别是 MQTT3.1.1 和 MQTT5。MQTT3.1.1 是在 2014 年 10 月发布的, 而 MQTT5 是在 2019 年 3 月发布的。虽然 MQTT3.1.1 与 MQTT5 在时间相差了将近五年, 但是 MQTT3.1.1 作为一个经典的版本, 目前仍然是主流版本, 能够满足大部分实际需求。

MQTT5 是在 MQTT3.1.1 的基础上进行了升级, 因此 MQTT5 是完全兼容 MQTT3.1.1 的。而 MQTT5 是在 MQTT3.1.1 的基础上添加了更多的功能、补充完善 MQTT 协议。具体增加了哪些功能, 这里不作说明, 如果大家有兴趣可以自行百度了解。

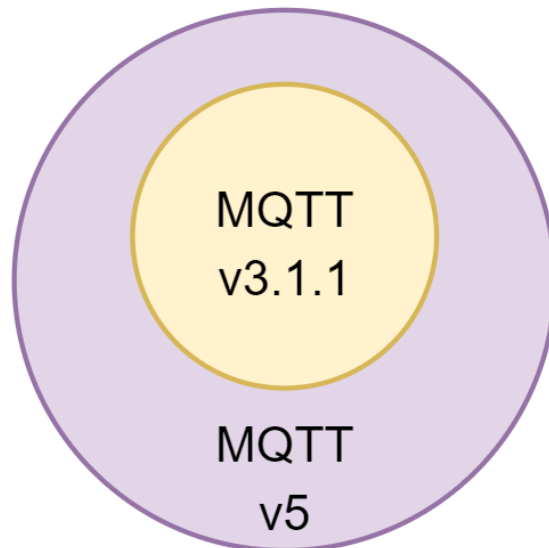


图 33.1.3 MQTT 3.1.1 与 MQTT 5

33.2 MQTT 协议 (上)

在上小节中, 我们了解了 MQTT 的背景知识和基本特点, 本小节开始我们将一起了解 MQTT 通信基本原理。

33.2.1 MQTT 通信基本原理

MQTT 是一种基于客户端-服务端架构的消息传输协议, 所以在 MQTT 协议通信中, 有两个最为重要的角色, 它们便是服务端和客户端。

服务端

MQTT 服务端通常是一台服务器 (broker)，它是 MQTT 信息传输的枢纽，负责将 MQTT 客户端发送来的信息传递给 MQTT 客户端；MQTT 服务端还负责管理 MQTT 客户端，以确保客户端之间的通讯顺畅，保证 MQTT 信息得以正确接收和准确投递。

客户端

MQTT 客户端可以向服务端发布信息，也可以从服务端收取信息；我们把客户端发送信息的行为称为“发布”信息。而客户端要想从服务端收取信息，则首先要向服务端“订阅”信息。“订阅”信息这一操作很像我们在使用微信时“关注”了某个公众号，当公众号的作者发布新的文章时，微信官方会向关注了该公众号的所有用户发送信息，告诉他们有新文章更新了，以便用户查看。

MQTT 主题

上面我们讲到了，客户端想要从服务器获取信息，首先需要订阅信息，那客户端如何订阅信息呢？这里我们要引入“主题 (Topic)”的概念，“主题”在 MQTT 通信中是一个非常重要的概念，客户端发布信息以及订阅信息都是围绕“主题”来进行的，并且 MQTT 服务端在管理 MQTT 信息时，也是使用“主题”来控制的。

客户端发布消息时需要为消息指定一个“主题”，表示将消息发布到该主题；而对于订阅消息的客户端来说，可通过订阅“主题”来订阅消息，这样当其它客户端或自己（当前客户端）向该主题发布消息时，MQTT 服务端就会将该主题的信息发送给该主题的订阅者（客户端）。

为了便于您更好地理解服务端是如何通过“主题”来控制客户端之间的信息通讯，我们来看看下图实例：

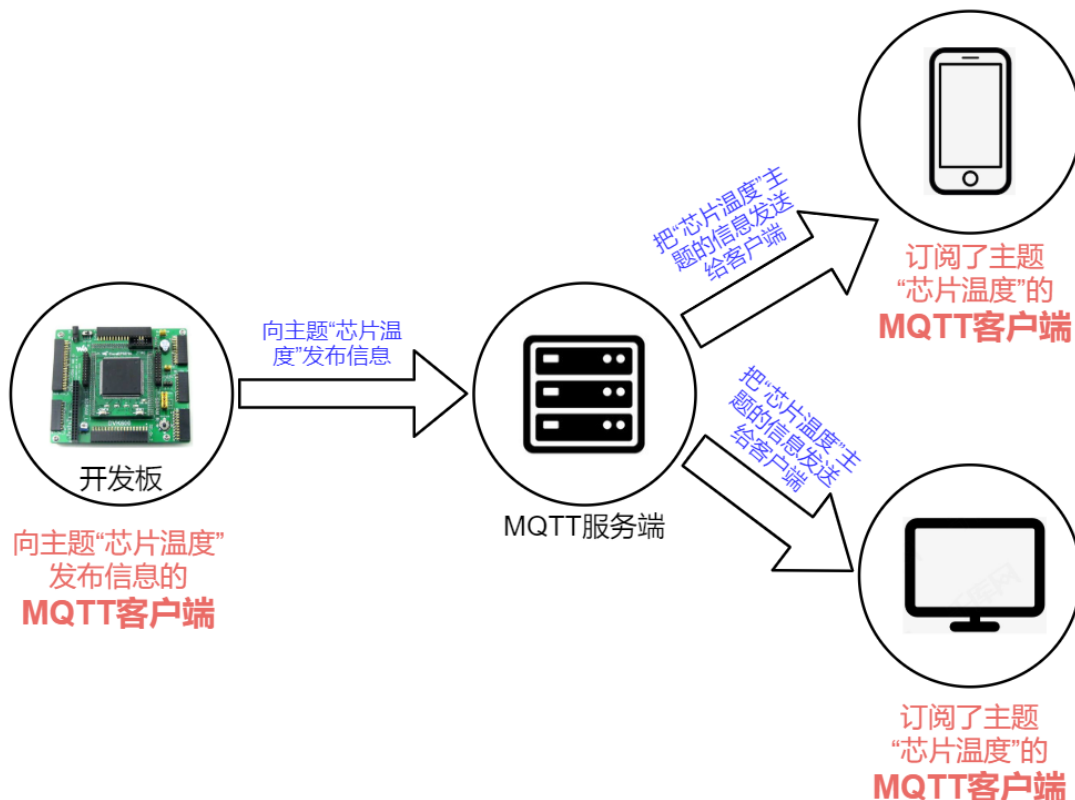


图 33.2.1 MQTT 通信示例 1

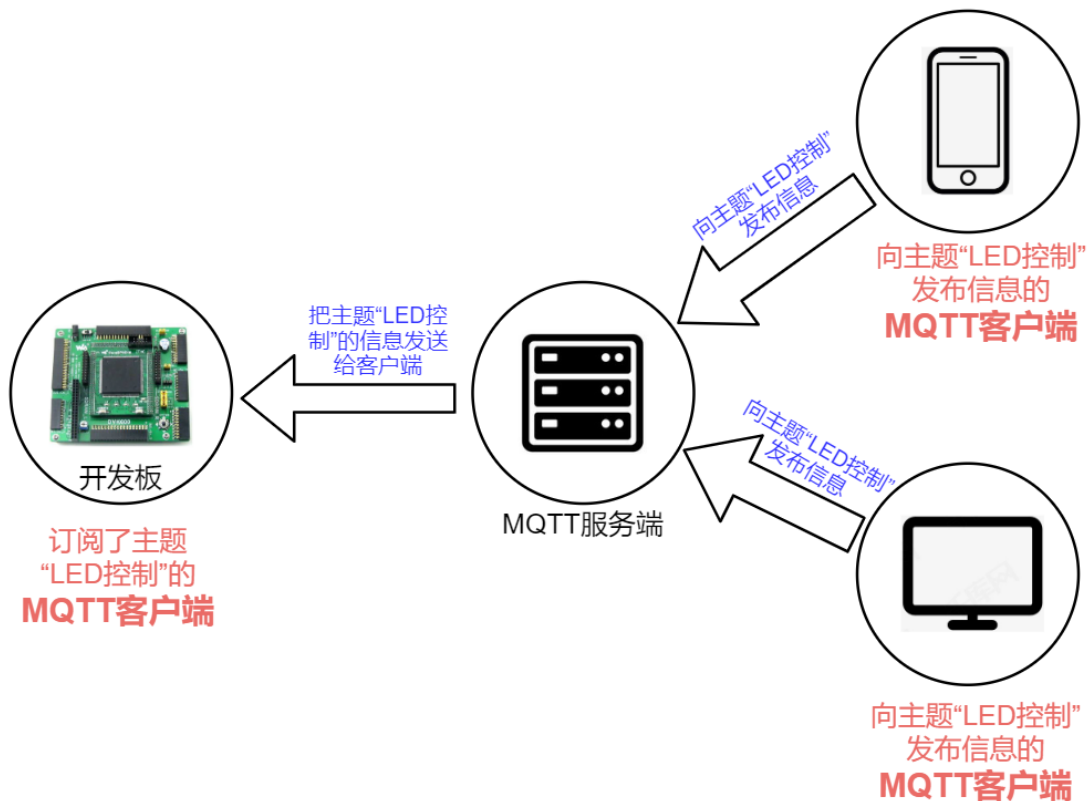
在以上图示中一共有三个 MQTT 客户端，它们分别是开发板、手机和电脑。MQTT 服务端在管理 MQTT 通信时使用了“主题”来对信息进行管理。比如上图所示，假设我们需要利用手机和电脑获取开发板在运行过程中 SoC 芯片的温度，那么首先手机和电脑这两个客户端需要向 MQTT 服务器订阅主题“芯片温度”；接下来，当开发板客户端向服务端的“芯片温度”主题发布信息（假设信息的内容就是当前的温度值）后，

服务端就会首先检查都有哪些客户端订阅了“芯片温度”这一主题的信息，而当它发现订阅了该主题的客户端有一个手机和一个电脑，于是服务端就会将刚刚收到的“芯片温度”信息转发给订阅了该主题的手机和电脑客户端。

通过以上的这种实例，手机和电脑便可以获取到开发板运行时 SoC 芯片的温度值。

以上实例中，开发板是“芯片温度”主题的发布者，而手机和电脑则是该主题的订阅者。

值得注意的是，MQTT 客户端在通信时，角色往往不是单一的，一个客户端既可以作为信息发布者也可以同时作为信息订阅者。如下图所示：



上图中的所有客户端都是围绕“LED 控制”这一主题进行通信。此时，对于“LED 控制”这一主题来说，手机和电脑客户端成为了 MQTT 信息的发布者而开发板则成为了 MQTT 信息的订阅者（接收者）。

所以由此可知，针对不同的主题，MQTT 客户端可以切换自己的角色，它们可能对主题 A 来说是信息发布者，但是对于主题 B 就成了信息订阅者，所以一个 MQTT 客户端它的角色并不是固定的，所以大家一定要理解“主题”这个概念。

MQTT 发布/订阅特性

从以上实例我们可以看到，MQTT 通信的核心枢纽是 MQTT 服务端，它负责将 MQTT 客户端发送来的信息传递给 MQTT 客户端，还负责管理 MQTT 客户端，以确保客户端之间的通讯顺畅，保证 MQTT 信息得以正确接收和准确投递。

正是因为有了服务端对 MQTT 信息的接收、储存、处理和发送，客户端在发布和订阅信息时，可以相互独立、且在空间上可以分离、时间上可以异步，这就是 MQTT 发布/订阅的特性：客户端相互独立、空间上可分离、时间上可异步，具体介绍如下：

- **客户端相互独立：**MQTT 客户端是一个个独立的个体，它们无需了解彼此的存在，依然可以实现信息交流。譬如在上面的实例中，开发板客户端在发布“芯片温度”信息时，开发板客户端本身完全不知道有多少个 MQTT 客户端订阅了“芯片温度”这一主题；而订阅了“芯片温度”主题的手

机和电脑客户端也完全不知道彼此的存在,大家只要订阅了“芯片温度”这一主题, MQTT 服务端就会在每次收到新信息时,将信息发送给订阅了“芯片温度”主题的客户端。

- **空间上分离:** 空间上分离相对容易理解, MQTT 客户端以及 MQTT 服务端它们在通信时是处于同一个通信网络中的,这个网络可以是互联网或者局域网;只要客户端联网,无论他们远在天边还是近在眼前,都可以实现彼此间的通讯交流;其实网络通信本就是如此,所以并不是 MQTT 通信所特有的。
- **时间上可异步:** MQTT 客户端在发送和接收信息时无需同步。这一特点对物联网设备尤为重要,前面我们也介绍了, MQTT 从诞生之初就是专为低带宽、高延迟或不可靠的网络而设计的,高延迟和不可靠网络必然就会导致时间上的异步;物联网设备在运行过程中发生意外掉线是非常正常的情况,我们使用上面的实例二的场景来作说明,当开发板在运行过程中,可能会由于突然断电(假设开发板是通过电源适配器供电的)导致掉线,这时开发板会断开与 MQTT 服务端的连接。假设此时我们的手机客户端向开发板客户端所订阅的“LED 控制”主题发布了信息,而开发板恰恰不在线,这时, MQTT 服务端可以将“LED 控制”主题的新信息保存,待开发板客户端再次上线后,服务端再将“LED 控制”信息推送给开发板。所以这就必然导致了,手机发送信息与开发板接收信息在时间上是异步的。

总结

本小节向大家介绍了 MQTT 通信的基本原理,在 MQTT 通信中,1 个服务端、多个客户端之间围绕“主题”进行了通信,所以本节重要在于大家需要理解各个客户端的相互关系以及服务端在其中所起的作用,并且理解“主题”这个概念以及 MQTT 发布/订阅模式的特性,后面向大家介绍具体的通信过程时,要迅速的反应过来。

讲到这里请您注意:对于 MQTT 发布/订阅模式的特性,我们总结的几个特点中都有一个“可”字。这个“可”字意味着客户端彼此之间可以独立,空间可以分离,时间可以异步。在我们实际应用中,客户端之间的关系既可以独立也可以相互依存。在空间上,既可以相距甚远,也可以彼此相邻。在时间上,既可以异步也可以同步。这个“可”字所体现的是 MQTT 通讯的灵活性。

33.2.2 连接 MQTT 服务端

MQTT 客户端之间想要实现通信,必须要通过 MQTT 服务端。所以,客户端无论是发布信息还是订阅信息都必须先连接到服务端。下面我们来看一下,客户端连接服务端的详细过程。

MQTT 客户端连接服务端总共包含了两个步骤:

- ①、首先客户端需要向服务端发送连接请求,这个连接请求实际上就是向服务端发送一个 CONNECT 报文,也就是发送了一个 CONNECT 数据包。

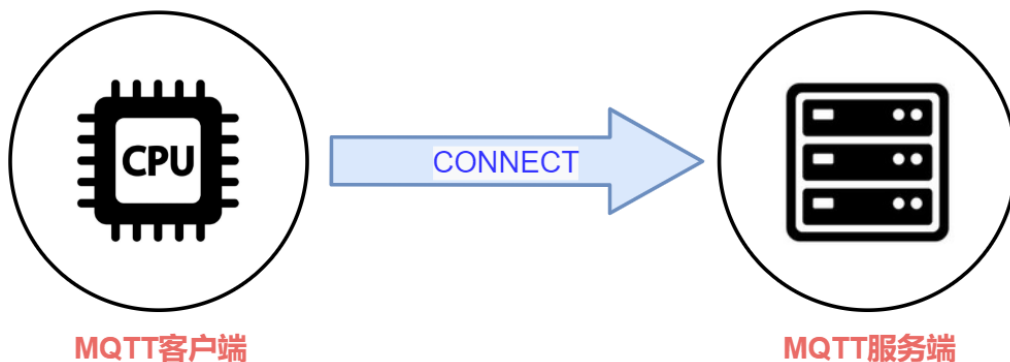


图 33.2.3 客户端向服务端发送连接请求---CONNECT 报文

②、MQTT 服务端收到连接请求后，会向客户端发送连接确认。连接确认实际上是向客户端发送一个 CONNACK 报文，也就是 CONNACK 数据包。

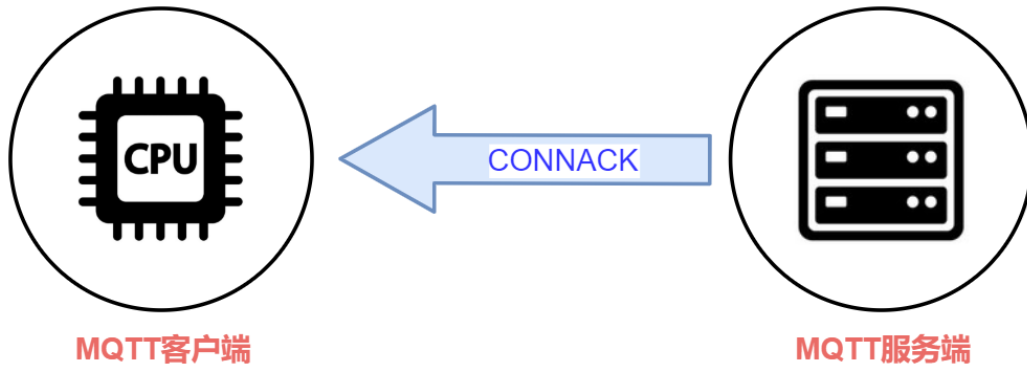


图 33.2.4 服务端向客户端发送连接确认---CONNACK 报文

以上就是 MQTT 客户端连接服务端的详细步骤，总结一句话就是：客户端先向服务端发送 CONNECT 报文，服务端收到连接请求后，再向待连接的客户端发送 CONNACK 报文。接下来，我们来看看 CONNECT 报文和 CONNACK 报文数据包中包含了哪些信息。

CONNECT 报文

在上面的描述中我们看到，MQTT 客户端要想连接服务端，首先要向服务端发送 CONNECT 报文。如果此 CONNECT 报文的格式或内容不符合 MQTT 规范，则服务器会拒绝客户端的连接请求。

CONNECT 报文包含的信息如下图所示：

CONNECT 报文	
名称	内容
clientID	"client-id"
keepAlive	60
cleanSession	true
willTopic(可选)	"clientWill"
willMessage(可选)	"client offline"
willRetain(可选)	false
willQos(可选)	0
username(可选)	"myname"
password(可选)	"dt29144"
.....

图 33.2.5 CONNECT 报文的内容

所谓报文就是一个数据包，MQTT 报文组成分为三个部分：固定头（Fixed header）、可变头（Variable header）以及有效载荷（Payload，消息体）。这里我们简单地介绍一下：

- **固定头 (Fixed header)**: 存在于所有 MQTT 报文中, 固定头中有报文类型标识, 可用于识别是哪一种 MQTT 报文, 譬如该报文是 CONNECT 报文还是 CONNACK 报文, 亦或是其它类型报文。
- **可变头 (Variable header)**: 存在于部分类型的 MQTT 报文中, 报文的类型决定了可变头是否存在及其具体的内容。
- **消息体 (Payload)**: 存在于部分类型的 MQTT 报文中, payload 就是消息载体的意思。

关于 MQTT 报文格式更加详细的内容, 如果大家有兴趣可以自己去了解, 笔者在这里就不再啰嗦了!

回到 CONNECT 报文中, 从图 33.2.5 中可知, CONNECT 报文包含了很多的信息, 左边的是信息的名称 (变量名), 右边则是信息的具体内容 (变量的值), 右边这些具体内容只是笔者给出的一个示例, 不是所有的 CONNECT 报文中的 clientId 信息内容都是 “client-id”, 这里只是举个例子而已!

另外也请注意, 上图中有些信息名称旁边标注了 “可选” 字样, 而有些则没有。那些没有标注 “可选” 字样的信息是必须包含在 CONNECT 报文中的。而对于标注了 “可选” 字样的信息, CONNECT 报文既可以包含它们也可以没有它们, 具体情况而定!

接下来笔者将向大家介绍 CONNECT 报文中这些信息表示什么意思, 有什么作用? 不过, 考虑到我们刚刚接触 MQTT 协议, 目前应先从最基础的内容开始学起, 所以本小节我们只介绍那些未标注 “可选” 字样的信息。

clientId--客户端 id

clientId 是 MQTT 客户端的标识, 也就是 MQTT 客户端的名字, MQTT 服务端可通过 clientId 来区分不同的客户端, MQTT 服务端用该标识来识别客户端。因此 clientId 必须是独立的, 如果两个 MQTT 客户端使用相同 clientId 标识, 服务端会把它们当成同一个客户端来处理。通常 clientId 是由一串字符所构成的, 譬如, 在上面的示例中, clientId 是 “client-id”。

keepAlive--心跳时间间隔

对于 MQTT 服务器来说, 它要判断一台 MQTT 客户端是否依然与它保持着连接状态, 可以检查这台客户端是不是经常发送消息给服务端, 如果服务端经常收到客户端的消息, 那么没问题, 这个客户端肯定在线。

但是有些客户端并不经常发送消息给服务端, 对于这种客户端, MQTT 协议使用了类似心跳检测的方法来判断客户端是否在线。前面在介绍 MQTT 时, 曾提到过 MQTT 支持心跳机制, 心跳机制其实就是用来判断客户端是否与服务端保持着连接的一种方法, 也就是说通过心跳机制来检测客户端是否在线。客户端在没有向服务端发送信息时 (空闲时), 可以定时向服务端发送一个心跳数据包, 这个心跳包也被称作心跳请求, 心跳请求的作用正是用于告知服务端, 当前客户端依然在线, 服务端在收到客户端的心跳请求后, 会回复一条消息, 这条回复消息被称作心跳响应。

关于 MQTT 的心跳机制后面还会向大家进行讲解, 这里便不再多说了! CONNECT 报文中的 keepAlive 其实是指定了心跳时间间隔, 也就是客户端向服务端发送心跳包的时间间隔。譬如 keepAlive=60, 表示告诉服务端, 客户端将会每隔 60 秒左右向服务端发送心跳包。

cleanSession--清除会话

所谓 “清除会话” 这一翻译源自 MQTT 官方文档中文版。这是一个布尔值, cleanSession 标志可用于控制客户端与服务端在连接和断开连接时的行为, 我们举个例子来进行说明, QQ、微信这些聊天软件大家都用过, 假设当前你的 QQ 账号没有登录或者说当前处于离线状态, 与服务器断开了连接; 而在离线期间, 你的 QQ 好友给你发了几条信息; 由于当前你的 QQ 处于离线状态, 自然是接收不到好友发送过来的信息, 但是, 当你的 QQ 恢复连接状态时, 立马会接收到好友在离线期间所发给你的信息。

而 cleanSession 就与这个有关系, 它是一个布尔值, 如果连接服务端时 cleanSession=0, 当 MQTT 客户端由离线 (与服务端断开连接) 再次上线时, 离线期间发给客户端的所有 QoS>0 的消息仍然可以接收到; 如果连接服务端时 cleanSession=1, 当 MQTT 客户端由离线 (与服务端断开连接) 再次上线时, 离线期间发

给客户端的所有消息一律接收不到。注意，这里我们提到了 QoS，关于 QoS 的概念后续再向大家介绍，这里暂时先不去理会，先记住有这么个东西。

说白了，想接收离线消息，客户端连接服务端时必须使用 `cleanSession=0`；除了这个作用之外，如果 `cleanSession=0`，则 MQTT 服务端会在客户端断开连接之后“记住”MQTT 客户端在线期间所订阅的所有“主题”；也就是说，服务端会保存、存储客户端所订阅的主题。

如果 `cleanSession=1`，客户端既无法接收到离线消息、服务端也不会记住该客户端所订阅的主题，服务端不会保存客户端的会话状态，每次连接都是一次新的会话；既然是新的会话，那就不会保存以前的会话状态信息，一切从“新”开始、从而丢弃以前的会话状态信息（包括：离线期间发送给客户端的消息以及客户端订阅的主题等），这就是“清除会话”的含义。

所以这就是 `cleanSession` 的作用，总的来说：`cleanSession` 设置为 1，表示此次连接将创建一个新的临时会话，在客户端断开后，这个会话会自动销毁。而 `cleanSession` 设置为 0，表示创建一个持久性会话，在客户端断开连接时，会话仍然保持并保存离线消息，直到会话超时注销。

关于 `cleanSession` 就给大家介绍这么多，已经解释得很清楚了，相信大家已经明白了。

以上就是 CONNECT 报文的主要内容了，关于 CONNECT 报文中的其它内容，我们会在后续内容中给大家讲解。下面再看看 MQTT 服务端接收到客户端发来的连接请求后所回复的 CONNACK 报文详细内容。

CONNACK 报文

CONNACK 报文包含的信息如下图所示：

CONNACK 报文	
名称	内容
sessionPresent	true
returnCode	0
.....

图 33.2.6 CONNACK 报文的内容

CONNACK 报文包括两个信息，一个是 `returnCode`(连接返回码)，另一个是 `sessionPresent`。以下是这两个信息的说明：

returnCode--连接返回码

当服务端收到了客户端的连接请求后，会向客户端发送 `returnCode`(连接返回码)，用来说明连接情况。如果客户端与服务端成功连接，则返回数字“0”。如果未能成功连接，返回码将会是一个非零的数字，具体这个数字的含义，请见下表：

返回码	说明
0	连接成功
1	连接被服务端拒绝，原因是不支持客户端的 MQTT 协议版本
2	连接被服务端拒绝，原因是不支持客户端标识符的编码。可能造成此原因是客户端标识符编码是 UTF-8，但是服务端不允许使用此编码。

3	连接被服务端拒绝, 原因是服务端不可用。即, 网络连接已经建立, 但 MQTT 服务不可用。
4	连接被服务端拒绝, 原因是用户名或密码无效。
5	连接被服务端拒绝, 原因是客户端未被授权连接到此服务端。
6-255	保留备用

表 33.2.1 返回码说明

sessionPresent

sessionPresent 与 CONNECT 报文中的 cleanSession 有关系, 前面说了, 客户端连接服务端时, 如果 cleanSession=0, 服务端会保存与客户端的会话状态, 记录客户端订阅的主题, 即使客户端断开了与服务端的连接, 会话任然保持并保存离线消息, 直到客户端重新上线, 这些离线消息就会发给客户端。

在 cleanSession=0 的情况下, 当客户端连接到服务器之后, 可通过 CONNACK 报文中返回的 sessionPresent 来查询服务端是否为客户端保存了会话状态 (客户端上一次连接时的会话状态信息), 如果服务端已为客户端保存了上一次连接时的会话状态, 则 sessionPresent=1, 如果没有保存会话状态, 则 sessionPresent=0。

如果 cleanSession=1, 在这种情况下, 客户端是不需要服务端保存会话状态的, 那么服务端发送的确认连接 CONNACK 报文中, sessionPresent 肯定是 false (sessionPresent=0), 也就是说, 服务端没有保存客户端的会话状态信息。

简言之, CONNACK 报文的 sessionPresent 与 CONNECT 报文的 cleanSession 相互配合。其作用是客户端发送连接请求时, 服务端告知客户端有没有保存会话状态。这个被服务端保存的会话状态是来自于上一次客户端连接时, 譬如离线消息以及上一次连接时客户端所订阅的主题。

Tips: 关于 MQTT 协议的参考资料, 链接地址如下:

http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718027

33.2.3 断开连接

当 MQTT 客户端连接到服务端之后, 在后续的通信过程中, 如果客户端想要断开与服务端的连接, 此时客户端可以主动向服务端发送一个 DISCONNECT 报文来断开与服务端的连接, 如下图所示:

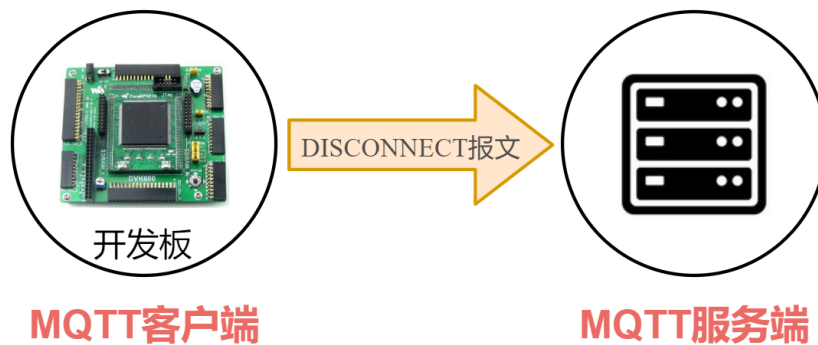


图 33.2.7 客户端断开与服务端的连接

33.2.4 发布消息、订阅主题与取消订阅主题

上小节我们学习了 MQTT 客户端连接 MQTT 服务端的详细过程, 当客户端连接到服务端之后, 便可以发布消息或订阅主题了, 本小节我们便来学习客户端如何实现发布消息、订阅主题以及取消订阅主题。

PUBLISH - 发布消息

当客户端连接到服务端之后,就可以向服务端发布消息了,每条发布的消息必须指定一个“主题”,表示向某主题发布消息;MQTT 服务端可以通过主题来确定将消息转发给哪些客户端(订阅了该主题的客户端)。例如图 33.2.1 所示,开发板客户端向主题“芯片温度”发布了消息,这个消息就是当前芯片的温度值,而手机和电脑这两个客户端订阅了主题“芯片温度”;所以,当服务端接收到开发板发布的消息之后,就会将这个消息转发给订阅了主题“芯片温度”的手机和电脑这两个客户端,这样,手机和电脑就会收到开发板发布的消息。

MQTT 客户端向服务端发布消息其实就是向服务端发送一个 PUBLISH 报文,服务端收到客户端发送过来的 PUBLISH 报文之后,会向发送方回复一个报文。根据 QoS 的不同,回复的报文类型也是不同的,并且整个发布消息的过程也将会有所区别;譬如对于 QoS=1 时,客户端向服务端发送 PUBLISH 报文,服务端收到 PUBLISH 报文之后会向发送方回复 PUBACK 报文;而对于 QoS=2 的情况,将会更加复杂,这个后续再给大家介绍。

下图是 PUBLISH 报文包含的信息:

PUBLISH 报文	
名称	内容
packetId	111
topicName	"myTopic"
payload	"Hello World"
qos	0
retain	false
dup	dup
.....

图 33.2.8 PUBLISH 报文的内容

同样,左侧是信息的名称,右侧是信息的具体内容。下面我们将一一进行介绍:

packetId--报文标识符

报文标识符可用于对 MQTT 报文进行标识(识别不同的报文)。不同的 MQTT 报文所拥有的标识符不同。MQTT 设备可以通过该标识符对 MQTT 报文进行甄别和管理, MQTT 协议内部使用的标识符。请注意:报文标识符的内容与 QoS 级别有密不可分的关系。只有 QoS 级别大于 0 时,报文标识符才是非零数值。如果 QoS 等于 0,报文标识符为 0,为什么是这样的呢?后面向大家介绍 QoS 概念时会做一个简单地说明!

topicName--主题名字

这个就是发布消息时对应的主题的名字,这是一个字符串,譬如上图中 topicName="myTopic",表示会将消息发布到“myTopic”这个主题。

payload--有效载荷

有效载荷是我们希望通过 MQTT 所发送的实际内容。我们可以使用 MQTT 协议发送字符串文本,图像等格式的内容。这些内容都是通过有效载荷所发送的。

qos--服务质量等级

QoS (Quality of Service) 表示 MQTT 消息的服务质量等级。QoS 有三个级别: 0、1 和 2, QoS 决定 MQTT 通信有什么样的服务保障。有关 QoS 的详细信息我们会在后续内容中详细讲解。

retain--保留标志

在默认情况下, 当客户端订阅了某一主题后, 并不会马上接收到该主题的信息。因为客户端订阅该主题之后, 并没有其它客户端向该主题发布消息; 只有在客户端订阅该主题后, 服务端接收到该主题的新消息时, 服务端才会将最新接收到的该主题消息推送给客户端。

但是在有些情况下, 我们需要客户端在订阅了某一主题后马上接收到一条该主题的信息。这时候就需要用到保留标志这一信息。关于保留标志的具体使用方法, 我们将在本教程的后续部分进行详细讲解。

dup--重发标志

dup 标志指示此消息是否重复。

当 MQTT 报文的接收方没有及时向报文发送发回复**确认收到报文**时, 发送方会以为对方没有收到信息, 会再次重复发送 MQTT 报文 (譬如客户端向服务端发送 PUBLISH 报文, 服务端收到 PUBLISH 报文之后需要向客户端回复一个 PUBACK 报文, 如果客户端没收到 PUBACK 报文, 则会认为服务端可能没接收到自己发送的报文, 将会再次发送 PUBLISH 报文)。在重复发送 MQTT 报文时, 发送方会将此“dup--重发标志”设置为 true。请注意, 重发标志只在 QoS 级别大于 0 时使用。有关 QoS 的详细信息, 我们将会在后续内容中为您做详细介绍。

SUBSCRIBE--订阅主题

当客户端连接到服务端后, 除了可以发布消息, 也可以接收消息。我们在之前的课程讲过, 所有 MQTT 消息都有主题。客户端要想接收消息, 首先要订阅该主题的主题。这样, 当有客户端向该主题发布消息后, 订阅了该主题的客户端就能接收到消息了。

客户端要想订阅主题, 首先要向服务端发送主题订阅请求。客户端是通过向服务端发送 SUBSCRIBE 报文来实现这一请求的。该报文包含有一系列“订阅主题名”。请留意, 一个 SUBSCRIBE 报文可以包含有单个或者多个订阅主题名。也就是说, 一个 SUBSCRIBE 报文可以用于订阅一个或者多个主题。

在以上 PUBLISH 报文讲解中, 我们曾经提到过 QoS (服务质量等级) 这一概念。同样的, 客户端在订阅主题时也可以明确 QoS。服务端会根据 SUBSCRIBE 中的 QoS 来提供相应的服务保障。

另外每一个 SUBSCRIBE 报文还包含有“报文标识符”。报文标识符可用于对 MQTT 报文进行标识。不同的 MQTT 报文所拥有的标识符不同。MQTT 设备可以通过该标识符对 MQTT 报文进行甄别和管理。

当客户端向服务端发送 SUBSCRIBE 报文, 服务端接收到 SUBSCRIBE 报文之后会向客户端回复一个 SUBACK 报文 (订阅确认报文), 如下图所示:

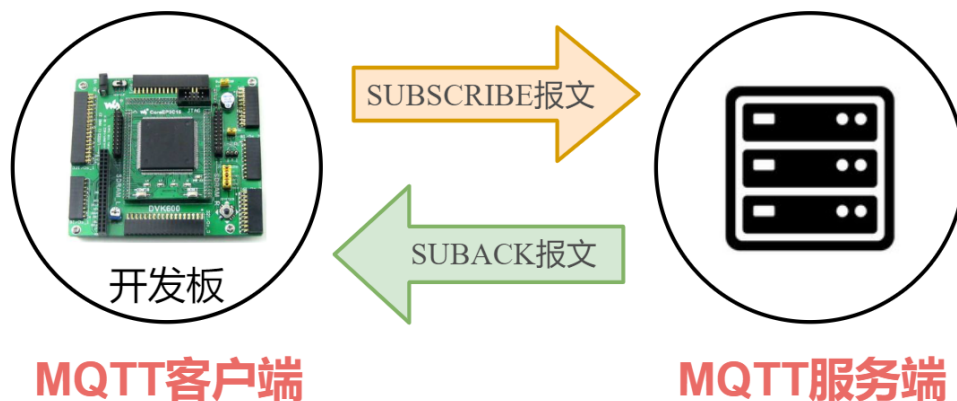


图 33.2.9 客户端订阅主题

服务端接收到客户端的订阅报文后, 会向客户端发送 SUBACK 报文确认订阅。

SUBACK 报文包含有“订阅返回码”和“报文标识符”这两个信息。

returnCode--订阅返回码

客户端向服务端发送订阅请求后，服务端会给客户端返回一个订阅返回码。

在之前的讲解中我们说过，客户端可通过一个 SUBSCRIBE 报文发送多个主题的订阅请求。服务端会针对 SUBSCRIBE 报文中的所有订阅主题来逐一回复给客户端一个返回码。

这个返回码的作用是告知客户端是否成功订阅了主题。以下是返回码的详细说明。

返回码	说明
0x00	订阅成功--QoS0
0x01	订阅成功--QoS1
0x02	订阅成功--QoS2
0x80	订阅失败

表 33.2.2 returnCode 返回码

由上表可知，当 returnCode=0、1 或 2 这三种情况时，都表示订阅成功；具体返回的数字是多少，根据订阅主题时 QoS 的不同，服务端的返回码也会有所不同！

UNSUBSCRIBE--取消订阅主题

客户端订阅了某一主题之后，可以随时取消订阅，MQTT 协议提供了这样的操作。

客户端通过向服务端发送一个 UNSUBSCRIBE 报文来取消订阅主题，当服务端接收到 UNSUBSCRIBE 报文后，会向发送发回复一个 UNSUBACK 报文（取消订阅确认报文），如下图所示：

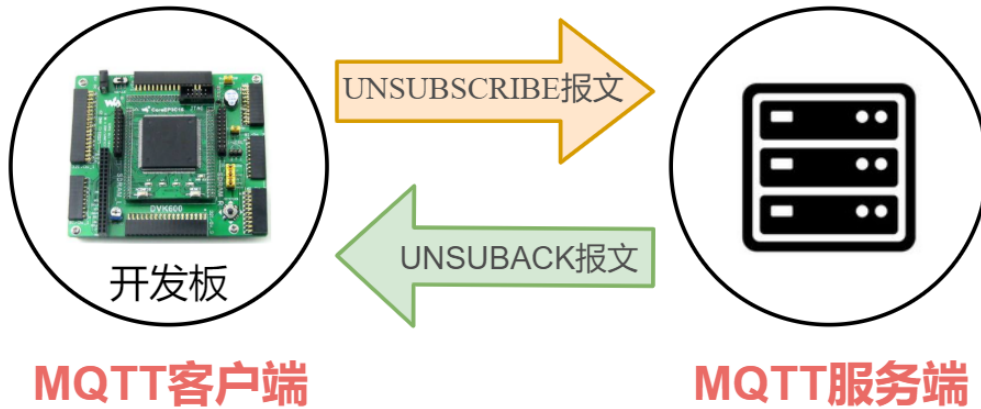


图 33.2.10 客户端取消订阅主题

UNSUBSCRIBE 报文包含两个重要信息，第一个是取消订阅的主题名称，同一个 UNSUBSCRIBE 报文可以同时包含多个取消订阅的主题名称。另外，UNSUBSCRIBE 报文也包含“报文标识符”，MQTT 设备可以通过该标识符对报文进行管理。

当服务端接收到 UNSUBSCRIBE 报文后，会向客户端发送取消订阅确认报文 - UNSUBACK 报文。该报文含有客户端所发送的“取消订阅报文标识符”。

客户端接收到 UNSUBACK 报文后就可以确认取消主题订阅已经成功完成了。

33.2.5 主题的进阶知识

通过前面的学习，我们可以知道在 MQTT 通信当中，主题是一个很重要的概念之一。在本小节中，我们来进一步了解 MQTT 主题的概念以及关于主题的进阶知识点。

1、主题的基本形式

主题的基本形式就是一个字符串, 譬如: "myTopic"、"currentTemp"、"LEDControl"等, 虽然看起来简单, 但是有几个点需要大家注意一下:

- **主题是区分大小写的。**所以"LEDControl"和"ledControl"是两个不同的主题。
- **主题可以使用空格。**譬如"LED Control", 虽然主题允许使用空格, 但是笔者建议大家尽量不要使用空格。
- **不要使用中文主题。**虽然有些 MQTT 服务器支持中文主题, 但是绝大部分 MQTT 服务器是不支持中文主题的, 所以大家不要使用中文主题, 而是使用 ASCII 字符来作为 MQTT 主题。

2、主题分级

MQTT 主题可以是一个简单的字符串, 譬如: "myTopic"、"currentTemp"、"LEDControl", 事实上, MQTT 协议为了更好的对主题进行管理和分类, 支持主题分级, 对主题进行分级处理, 各个级别之间使用"/"符号进行分隔。如下所示:

```
"home/sensor/led/brightness"
```

在以上示例中一共有四级主题, 分别是第 1 级 home、第 2 级 sensor、第三级 led、第 4 级 brightness。主题的每一级至少需要一个字符; 而只有一个简单字符串的主题, 譬如"myTopic"、"currentTemp"、"LEDControl", 这些都是单一级别的主题。

我们再来看几个分级主题的示例:

```
"home/sensor/kitchen/temperature"
```

```
"home/sensor/kitchen/brightness"
```

```
"home/sensor/bedroom/temperature"
```

```
"home/sensor/bedroom/brightness"
```

需要注意的是, 主题名称不要使用"/"开头, 譬如:

```
"/home/sensor/led/brightness"
```

这样是不行的。

3、主题通配符

当客户端订阅主题时, 可以使用通配符同时订阅多个主题。通配符只能在订阅主题时使用, 下面我们将介绍两种通配符: 单级通配符和多级通配符。

单级通配符: +

单级通配符可以匹配任意一个主题级别, 注意是一个主题级别, 譬如示例如下:

```
"home/sensor/+/status"
```

当客户端订阅了上述主题之后, 将会收到以下主题的信息内容:

```
"home/sensor/led/status"
```

```
"home/sensor/key/status"
```

```
"home/sensor/beeper/status"
```

.....

相反, 而以下这些主题的信息是无法接收到的:

```
"dt/sensor/led/status"
```

```
"home/kash/key/status"
```

```
"home/sensor/led/brightness"
```

.....

以上这些注意将无法接收到, 原因在于这些主题无法与"home/sensor/+status"相匹配。

这就是单级通配符的概念。

多级通配符:

多级通配符自然是可以匹配任意数量个主题级别, 而不再是单一主题级别, 多级通配符使用“#”号来表示, 譬如:

```
"home/sensor/#"
```

当客户端订阅了上面这个主题之后, 便可以收到如下注意的信息:

```
"home/sensor/led"
"home/sensor/key"
"home/sensor/beeper"
"home/sensor/led/status"
"home/sensor/led/brightness"
"home/sensor/key/status"
"home/sensor/beeper/status"
.....
```

相反, 如下主题的信息是无法接收到的:

```
"home/kash/led"
"dt/sensor/led"
"dt/kash/led"
.....
```

这就是多级通配符的概念。

4、主题应用注意事项

以\$开头的主题

以\$号开头的主题是 MQTT 服务端系统保留的特殊主题, 客户端不可随意订阅或向其发布信息, 譬如:

```
"$SYS/monitor/Clients"
"$SYS/monitor/+"
"$SYS/#"
```

以上这些主题示例我们不可随便订阅或向其发布信息, 类似的主题还有很多, 但都是以\$符号开头, 这里便不再一一列举了。

不要使用“/”作为主题开头

前面就给大家提到过了, 我们尽量不要使用“/”作为主题的开头, 这样做没有什么意义, 而且额外产生一个没有用处的主题级别。

主题中不要使用空格

虽然, MQTT 支持在主题中使用空格, 但是我们应该尽量避免使用空格。因为空格在编程当中是一个比较特殊的字符, 除了空格之外的其它特殊字符也应该不要使用。

保持主题简洁明了

MQTT 是一种轻量级的通讯协议, 它常用于网络带宽受限的环境, 因此我们应尽量让主题简洁明了, 从而让设备间交互的内容更加简洁, 以更好的适应网络带宽受限的环境。

主题中尽量使用 ASCII 字符

虽然有些 MQTT 设备支持 UTF-8 字符作为 MQTT 主题, 但是笔者建议您在主题中尽量使用 ASCII 字符。

33.3 MQTT 初体验

到目前为止, 我们已经学习了 MQTT 通信的基本原理、客户端连接服务端的原理、客户端发布消息、订阅主题等等理论知识。但是光有理论知识是不够的, 还需要动手实践来验证我们的理论知识, 那本小节我们将一起动手实践来体验 MQTT 通信的魅力!

既然是 MQTT 通信, 那就需要有两个重要的角色: MQTT 客户端和 MQTT 服务端。小节我们将使用电脑作为客户端, 而服务端我们将使用公用 MQTT 服务器。

33.3.1 下载、安装 MQTT.fx 客户端软件

想要将电脑作为 MQTT 客户端, 我们需要在电脑上安装一个 MQTT 客户端软件, MQTT 客户端软件很多, 这里笔者推荐 MQTT.fx 这款软件, 这款软件是目前主流的 MQTT 桌面客户端, 它支持 Windows、Mac、Linux 等多种操作系统, 它的官网是 <http://mqttfx.jensd.de/>。

进入到官网下载 MQTT.fx 软件, 如下所示:

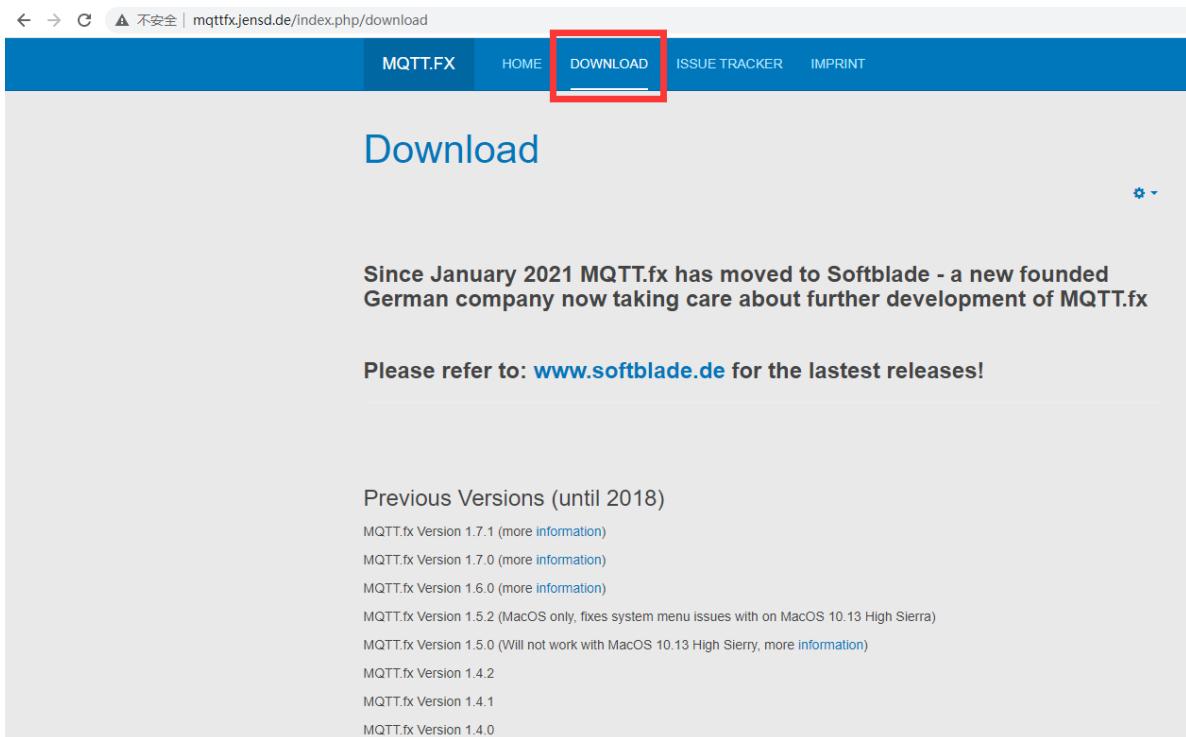


图 33.3.1 MQTT.fx 下载地址

以 1.7.1 版本为例, 进入到 <http://www.jensd.de/apps/mqttfx/1.7.1/> 链接地址进行下载, 如下:

← → ↻ ⚠ 不安全 | jensd.de/apps/mqttfx/1.7.1/

Index of /apps/mqttfx/1.7.1









Name	Last modified	Size	Description
 Parent Directory		-	
 mqttfx-1.7.1-1.i386.rpm	2018-10-02 09:44	99M	
 mqttfx-1.7.1-1.x86_64.>	2018-10-02 09:35	96M	
 mqttfx-1.7.1-32bit.deb	2018-10-02 09:43	75M	
 mqttfx-1.7.1-64bit.deb	2018-10-02 09:34	72M	
 mqttfx-1.7.1-macos.dmg	2018-09-28 14:27	55M	
 mqttfx-1.7.1-windows.>	2018-09-28 14:27	51M	← Windows 32位版本
 mqttfx-1.7.1-windows.>	2018-09-28 14:27	47M	← Windows 64位版本

图 33.3.2 1.7.1 版本下载地址

根据自己的需求下载即可！笔者使用的是 Windows 64 位系统，所以笔者下载的是最后一个，下载完成之后得到一个 exe 安装包文件：

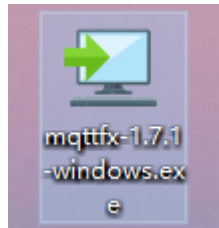


图 33.3.3 mqttfx-1.7.1-windows.exe

直接双击即可安装，傻瓜式安装，非常简单！

安装完成，打开软件之后界面如下所示：

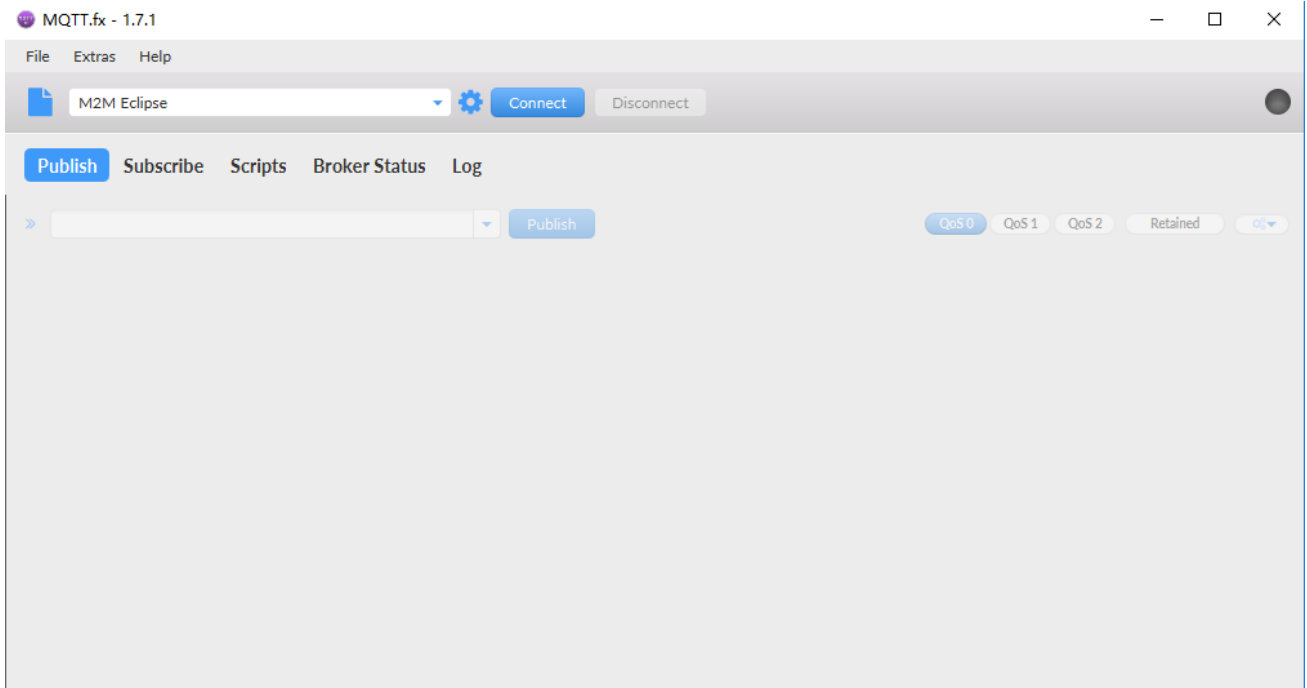


图 33.3.4 MQTT.fx 软件界面

至于怎么使用，稍后再向大家介绍。

33.3.2 MQTT 服务端

MQTT 客户端搞定之后，接下来就是 MQTT 服务端的问题了。我们可以自己搭建一个 MQTT 服务器，也可以使用现成的 MQTT 服务器。

自己搭建 MQTT 服务器

本章我们不使用自己搭建的服务器，因为如果你没有公网 IP 的主机，搭建的服务器也只能在局域网内使用，外部网络无法接入。

作为学习，笔者还是给大家简单地提一下关于自己如何去搭建 MQTT 的服务器。MQTT 服务器非常多，如 apache 的 ActiveMQ、emttqd、HiveMQ、Emitter、Mosquitto、Moquette 等等，既有开源的服务器也有商业服务器；作为学习，我们推荐 Mosquitto，Mosquitto 是一个高质量轻量级的开源 MQTT Broker，支持 MQTTv3.1 和 MQTTv3.1.1 协议，也是目前主流的开源 MQTT Broker，它的官方地址是 <https://mosquitto.org/>。

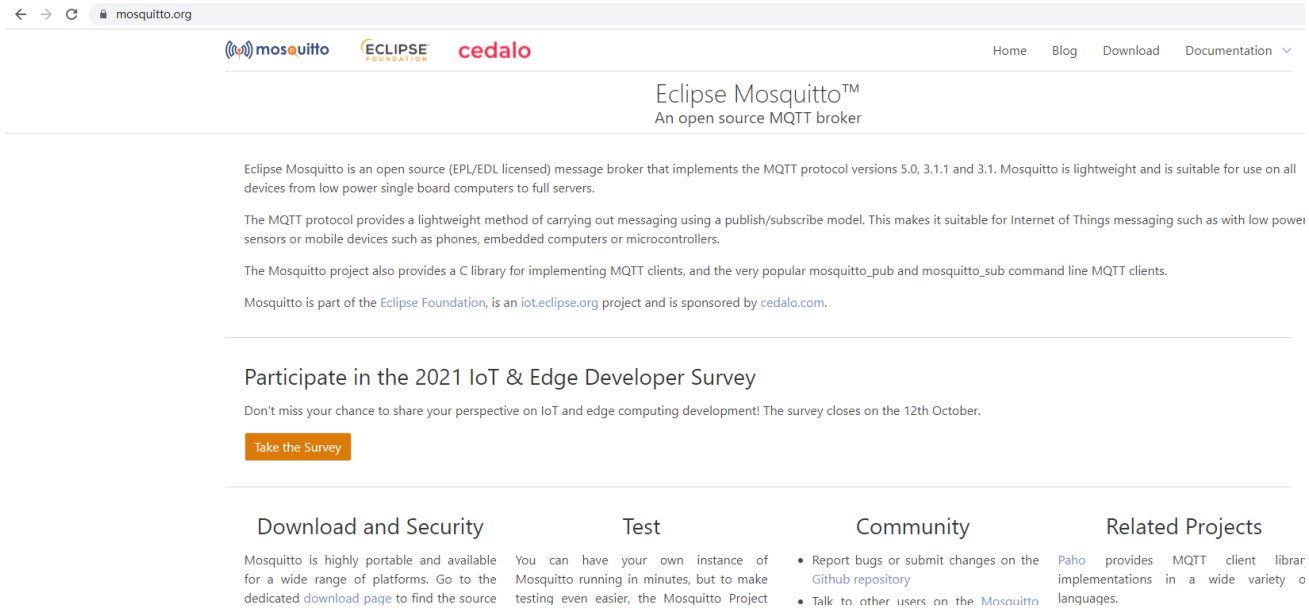


图 33.3.5 Mosquitto 官网

使用 Mosquitto 开源项目来搭建一个属于自己的 MQTT 服务器其实非常简单，具体的搭建方法笔者就不再介绍了，网上有很多帖子、博客对此进行了详细的讲解，如果有兴趣的读者可以百度搜索“搭建 Mosquitto 服务器”便可找到一大堆相关的文章。

公用 MQTT 服务器

除了自己搭建服务器之外，我们还可以使用现有的 MQTT 服务器，譬如阿里云、百度云、华为云等提供的 MQTT 服务，不过这些大平台貌似都是收费的；对于我们学习测试来说非常不友好，那既然如此我们还有别的选择吗？当然有，我们可以使用公用 MQTT 服务器，这些服务器都是免费供大家学习测试使用的，需要注意的是这些公用 MQTT 服务器**仅用于学习测试，不可拿来商用！**

以下给大家列举了一些公用 MQTT 服务器：

test.mosquitto.org (国外)

MQTT 服务器地址：test.mosquitto.org
 TCP 端口：1883
 TCP/TLS 端口：8883
 WebSockets 端口：8080
 Websocket/TLS 端口：8081

broker.hivemq.com (国外)

MQTT 服务器地址：broker.hivemq.com
 TCP 端口：1883
 WebSockets 端口：8000

iot.eclipse.org (国外)

MQTT 服务器地址：broker.hivemq.com
 TCP 端口：1883
 WebSockets 端口：8000

以上几个公用 MQTT 服务器都是国外的，大家可能因为网络的问题连接不上或者连接很慢，延迟会比较大；以下几个则是国内的公用 MQTT 服务器：

然也物联(国内)

官网地址: <http://www.ranye-iot.net>
MQTT 服务器地址: test.ranye-iot.net
TCP 端口: 1883
TCP/TLS 端口: 8883

通信猫 (国内)

MQTT 服务器地址: mq.tongxinmao.com
TCP 端口: 1883

推荐大家使用国内的这些 MQTT 服务器, 连接快、延迟低!

后续笔者将使用然也物联的 MQTT 服务器进行测试, 当然也推荐大家使用这个, 与本教程保持一致!

33.3.3 动手测试

现在我们要动手进行 MQTT 通信测试了, 首先打开之前安装的 MQTT 客户端软件 MQTT.fx:

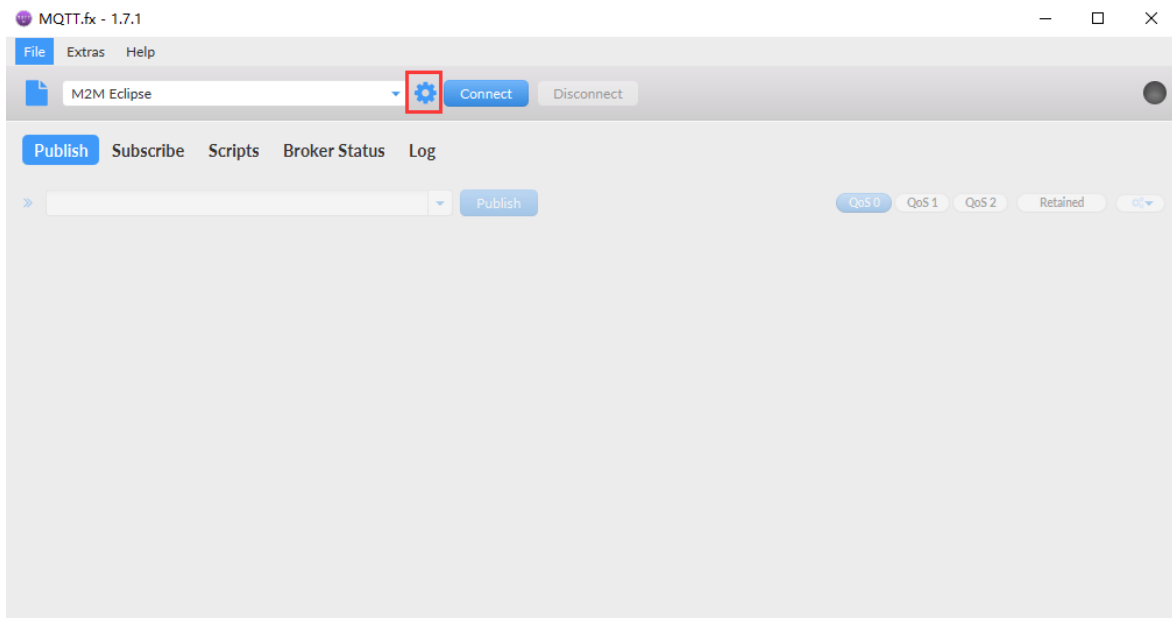


图 33.3.6 MQTT.fx 软件

打开之后如上图所示, 点击上图中红框标注的**齿轮按钮**打开配置界面:

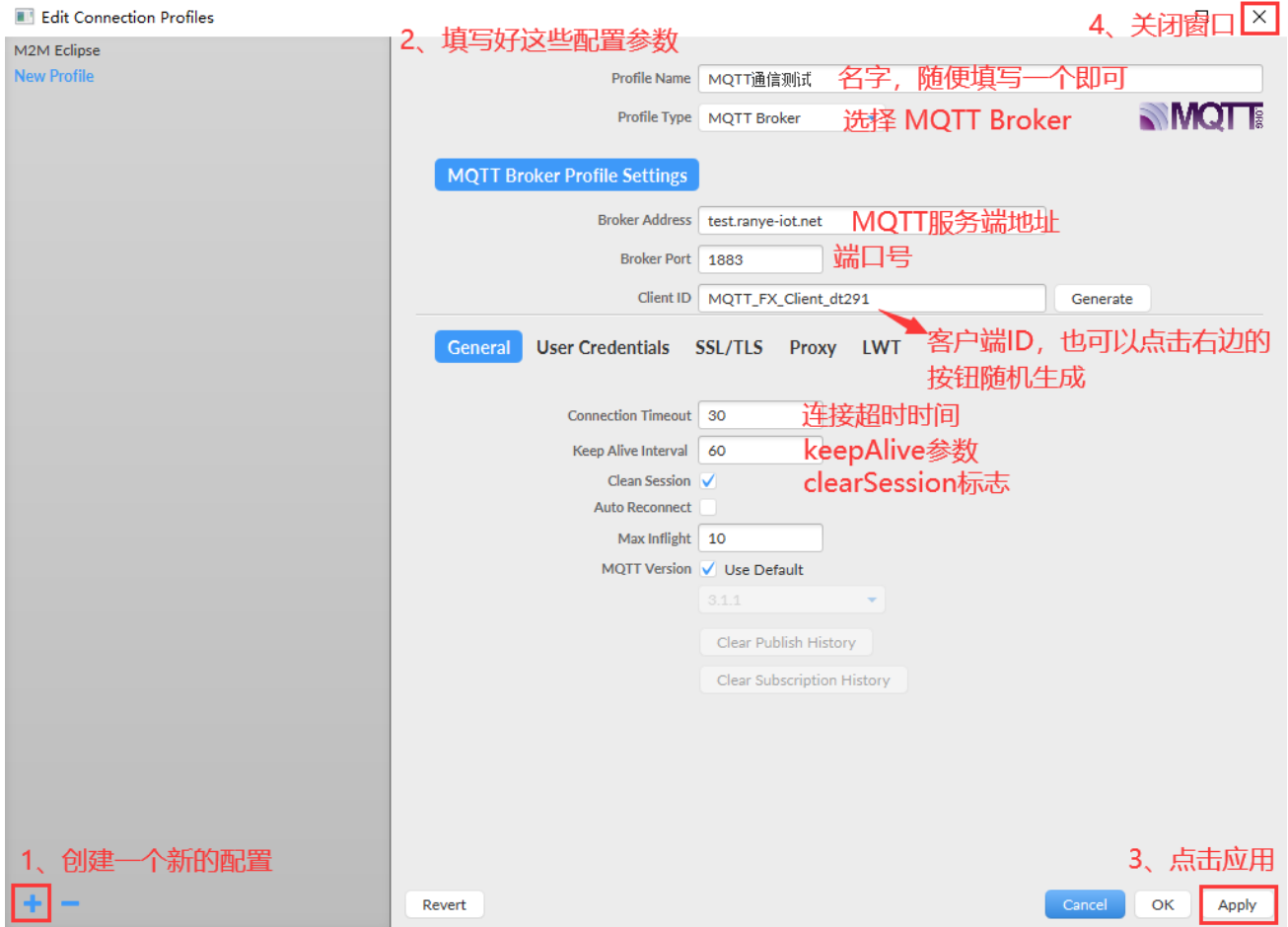


图 33.3.7 创建一个新的配置

首先点击 1 所示的“+”号创建一个新的配置，然后填写好各个配置参数，笔者使用的然也物联提供的公用 MQTT 服务器，服务器地址 test.ranye-iot.net，对应的端口号为 1883；clientId、keepAlive、cleanSession（勾选 Clean Session 表示 cleanSession=1）这些参数前面都给大家介绍过，这里便不再啰嗦！

配置完成之后，点击 3 所示的 Apply 按钮应用配置，最后点击右上角“X”关闭窗口：

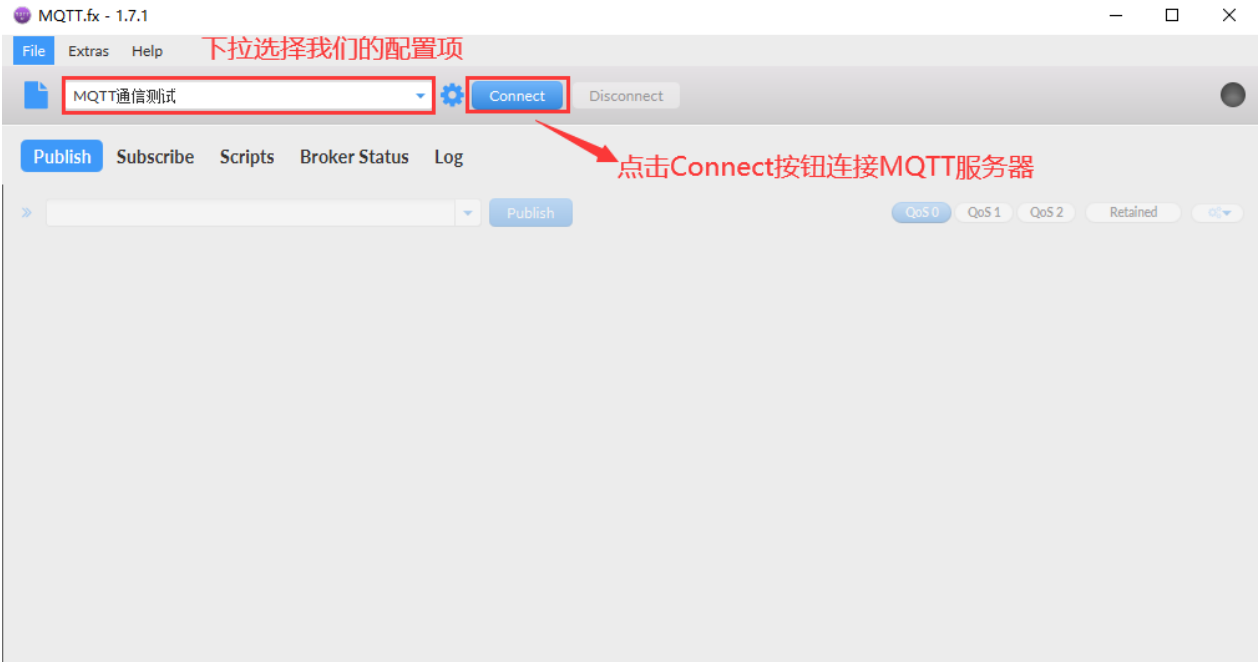


图 33.3.8 连接服务器

连接成功之后如下所示:

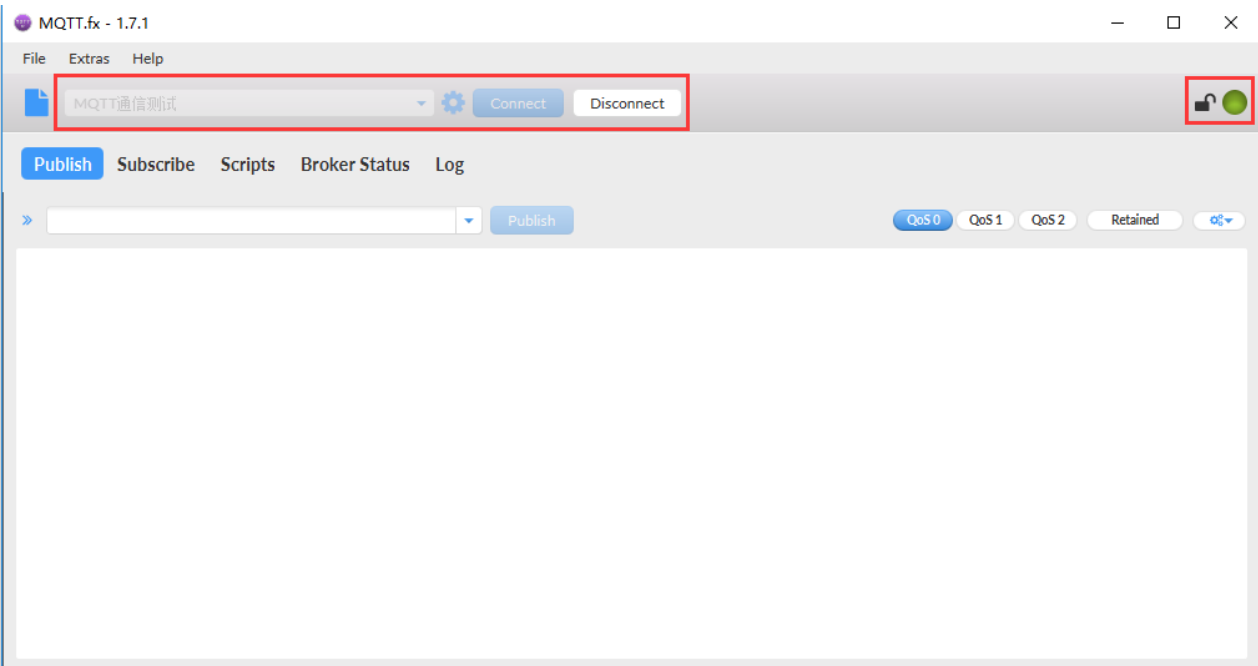


图 33.3.9 连接成功

这样我们的电脑作为 MQTT 客户端就已经成功连接到 MQTT 服务器了, 接下来我们便可以发布消息、订阅主题以及取消订阅主题了。

订阅主题

我们先订阅一个主题, 如下图所示:

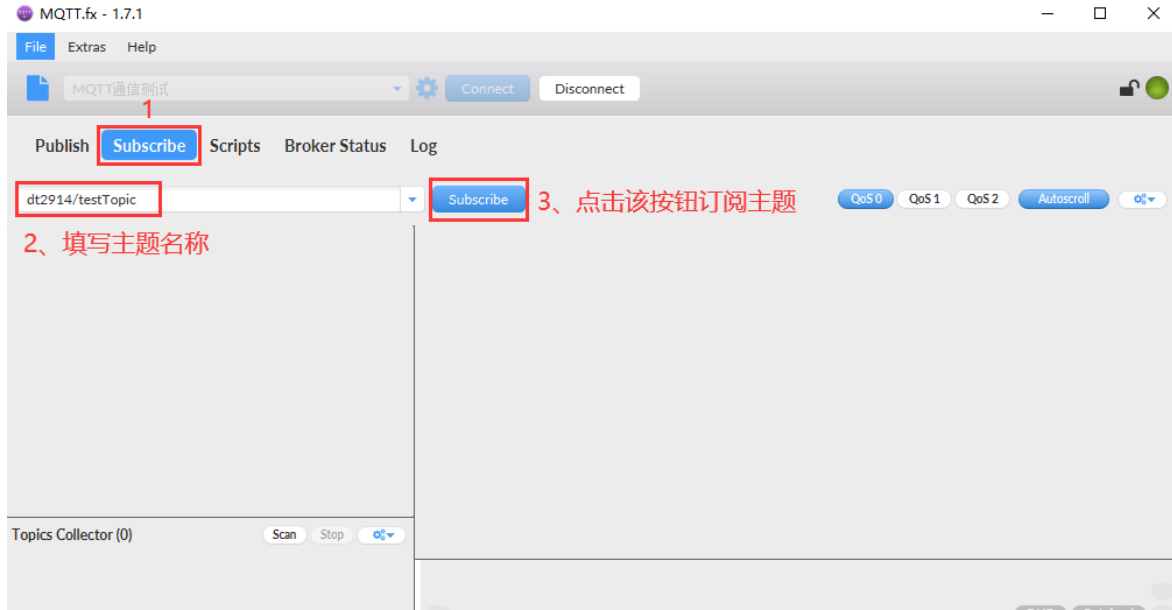


图 33.3.10 订阅主题

首先点击 1 所示的“Subscribe”切换到主题订阅界面，接着填写需要订阅的主题名称，示例中我们使用了分级主题“dt2914/testTopic”，最后点击 3 所示的“Subscribe”按钮向服务器发出订阅请求。订阅成功之后如下图所示：

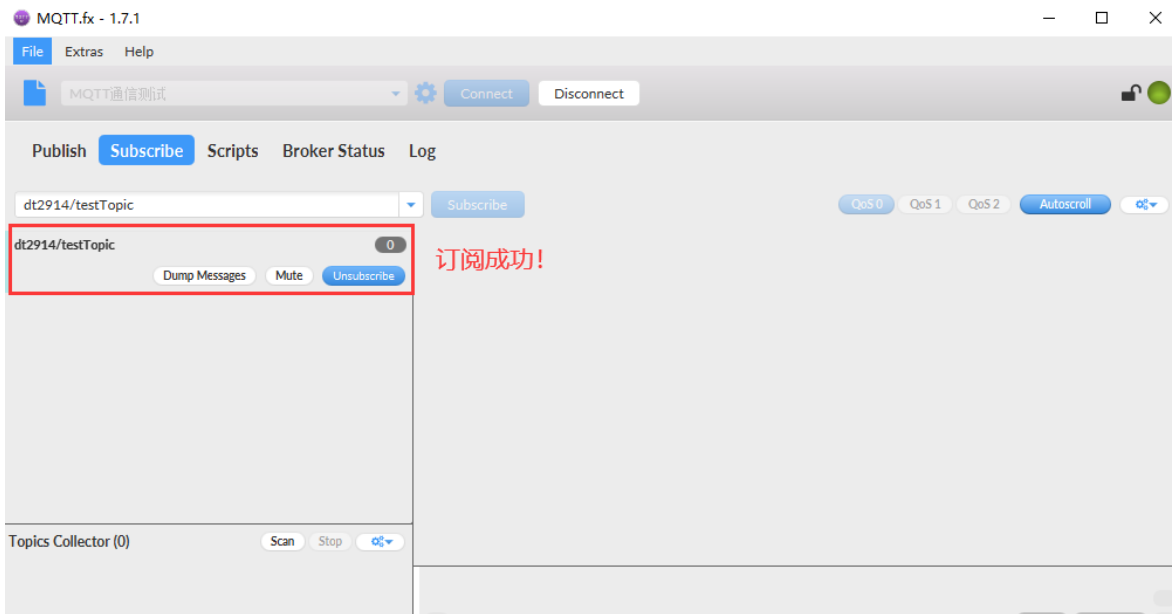


图 33.3.11 订阅成功

左边会列举出当前客户端所订阅的主题，右边显示消息列表。

发布消息

上例中我们的电脑客户端已经成功订阅了主题“dt2914/testTopic”，接下来我们尝试向该主题发布消息。如下图所示：



图 33.3.12 发布消息

首先点击 1 所示的“Publish”字样切换到发布消息界面，在 2 所示处填写主题名称，这里笔者需要向“dt2914/testTopic”主题发布消息；在 3 所示空白处填写需要发布的内容，然后点击 4 所示的“Publish”按钮发布消息。

此时我们的电脑客户端便会接收到自己发布的消息，如下所示：

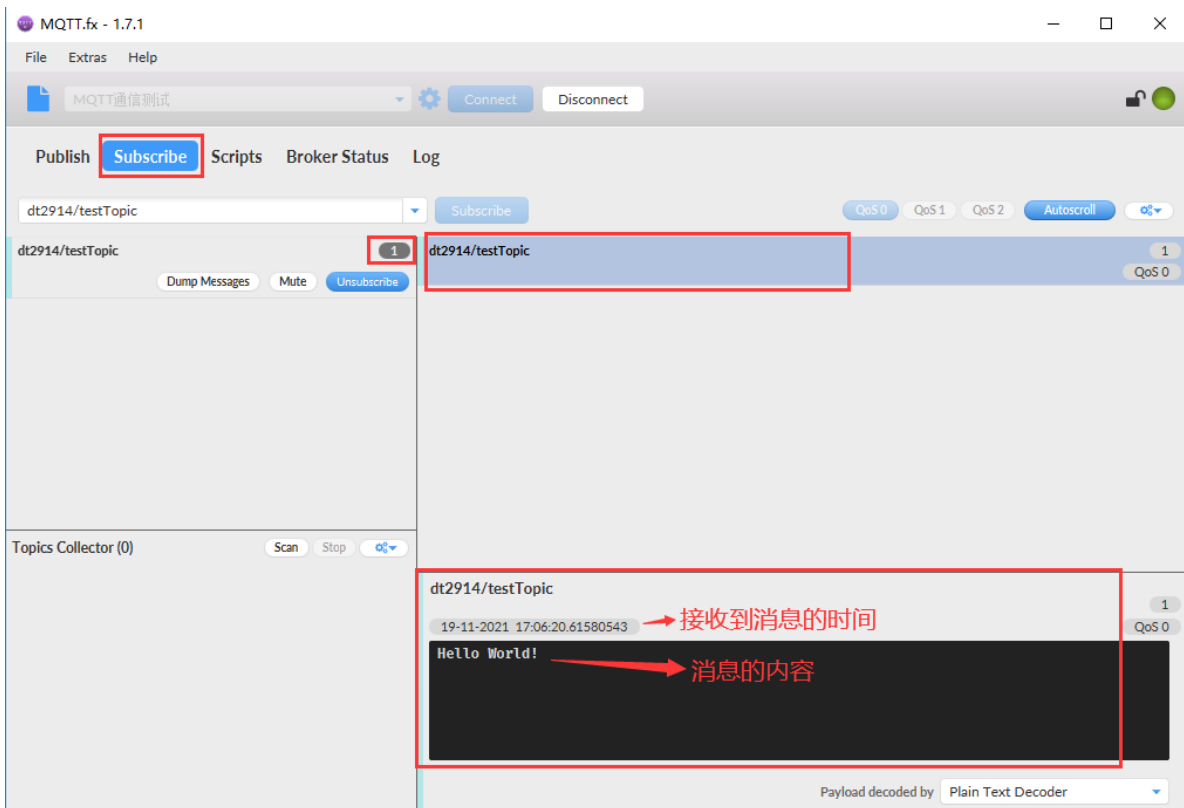


图 33.3.13 接收到消息

这样可能体现不出效果, 因为上例中是自己订阅了“dt2914/testTopic”主题, 接着又是自己向该主题发布消息, 虽然是自发自收, 但是这个消息肯定是经过了 MQTT 服务端的。

既然如此, 大家可以找来另一台电脑, 在另一台电脑上也安装 MQTT.fx 客户端软件, 一台电脑作为主题订阅者、而另一台电脑作为消息发布者进行测试。除此之外, 还有一个更简单的办法, 直接在电脑上运行两个 MQTT.fx 进程, 连接服务器时使用不同的 clientId, 这样就是两个不同的 MQTT 客户端了。

大家可以自己去测试, 笔者就不再啰嗦了!

取消订阅主题

取消订阅主题非常简单, 如下图所示:

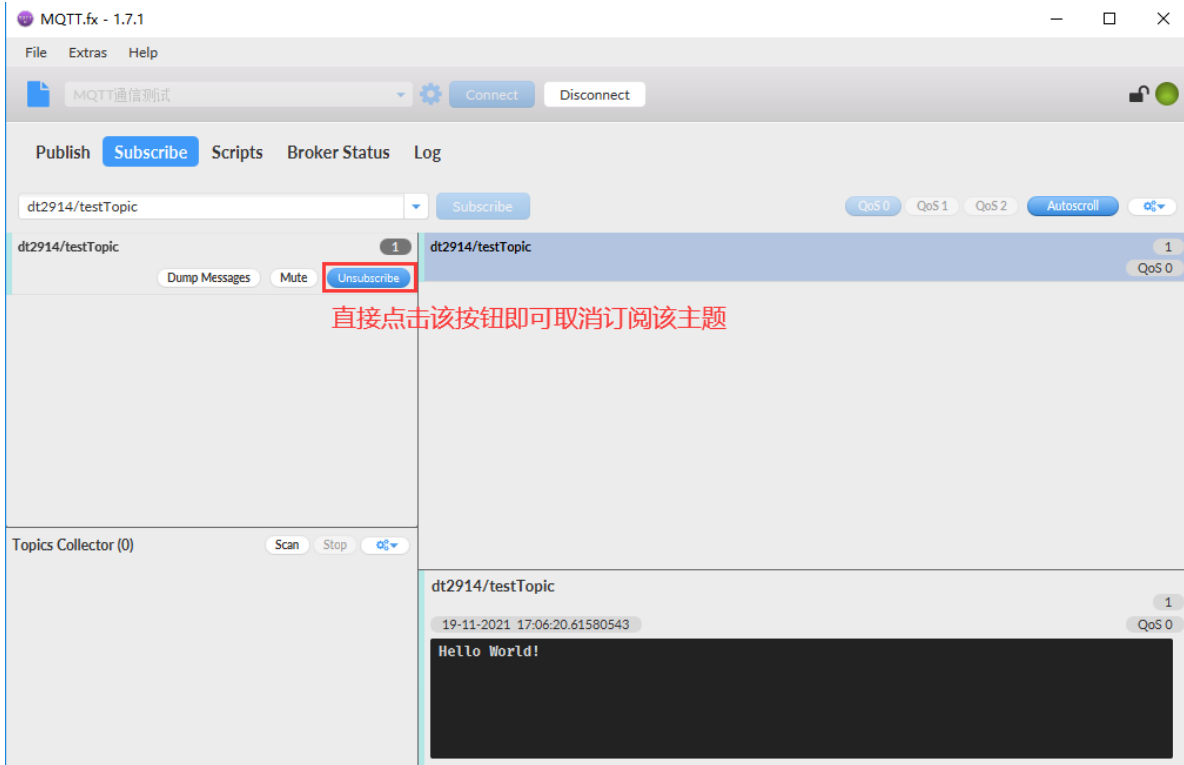


图 33.3.14 取消订阅主题

33.3.4 使用手机作为客户端

前面我们使用电脑作为客户端进行了测试, 其实我们的手机也可以作为 MQTT 客户端, 同样也需要安装一个客户端软件, 这里笔者使用的是 MQTTTool 工具, 在手机的应用商城可以找到该软件, 下载安装即可! 安装完成之后如下所示:



图 33.3.15 MQTTTool 软件

打开该软件然后进行配置、连接 MQTT 服务器，如下所示：



图 33.3.16 连接 MQTT 服务器

连接成功之后如下所示:

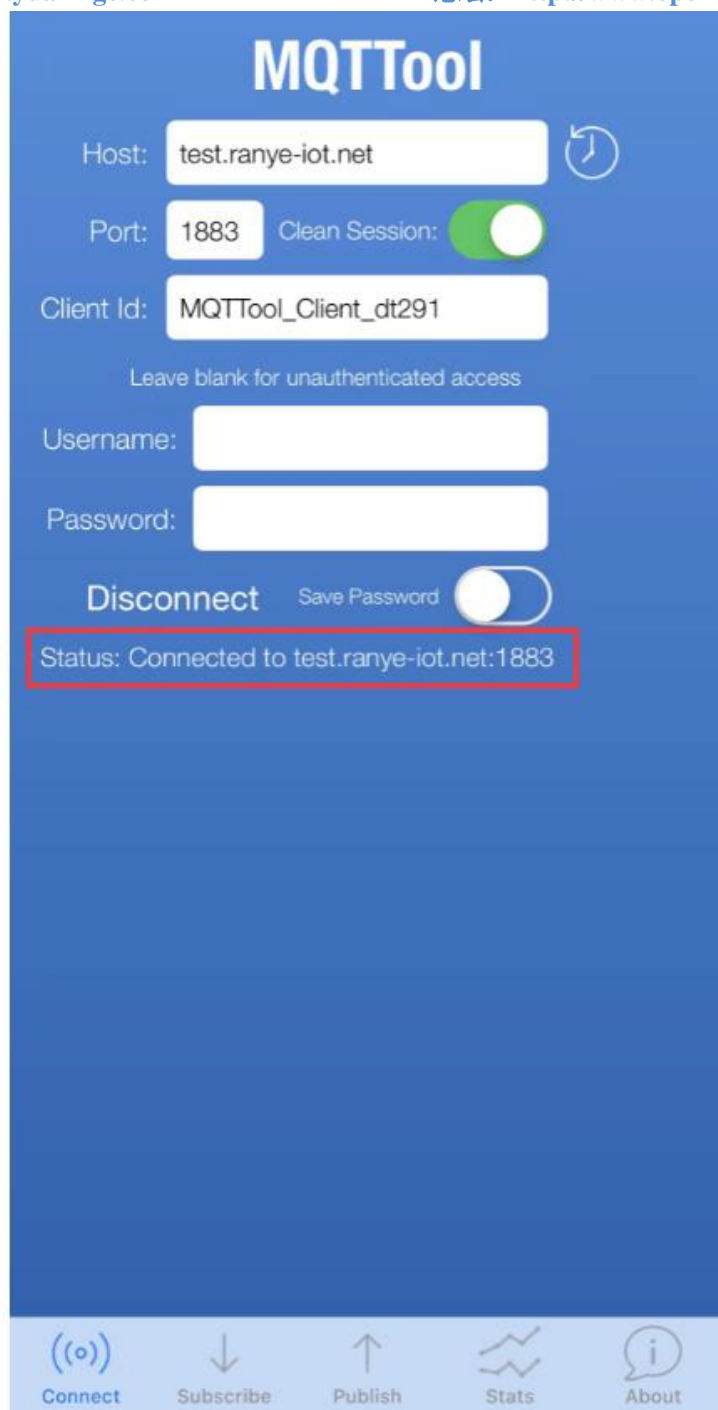


图 33.3.17 服务器连接成功

然后点击下边的“Subscribe”按钮可订阅主题、点击“Publish”按钮可发布消息。现在我们进行一个测试，在前面的示例中，我们的电脑客户端订阅了“dt2914/testTopic”主题，现在我们要使用手机客户端向“dt2914/testTopic”主题发布消息，如下所示：



图 33.3.18 发布消息

点击“Publish”按钮发布消息，接着电脑客户端便会接收到手机发布消息，如下所示：

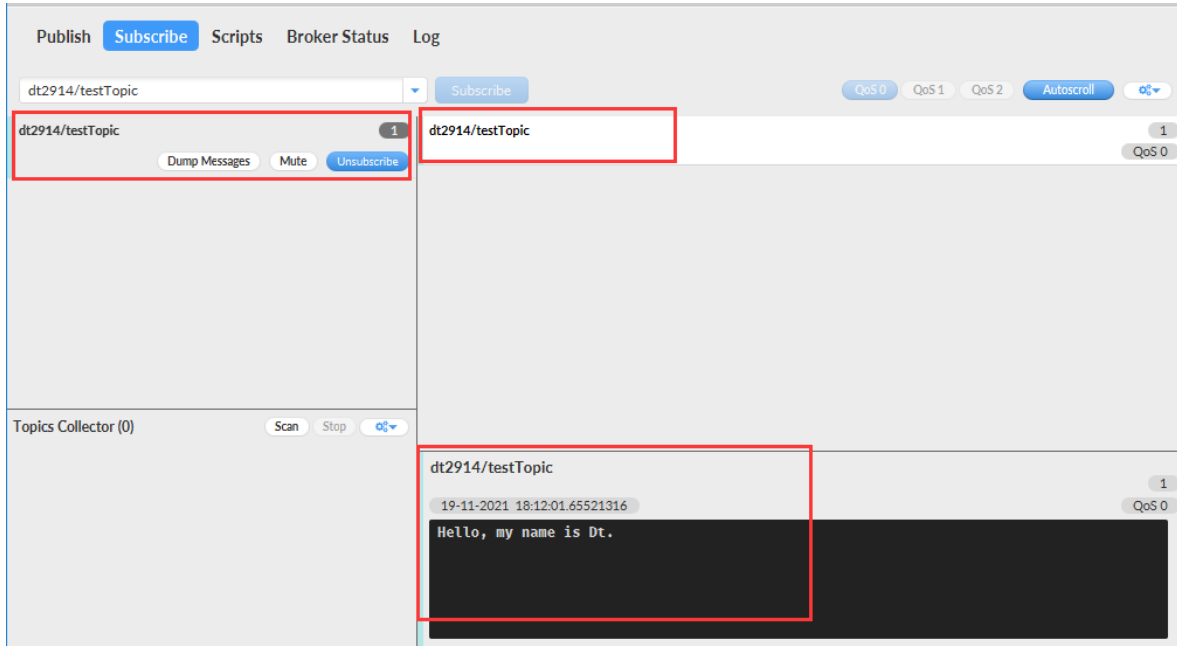


图 33.3.19 接收到消息

大家可以自己去测试，如果大家跟我一样使用的是然也物联提供的公用 MQTT 服务器，在测试过程有几个问题需要注意下：

请注意：

此公共版MQTT服务为开放式免费服务。因此本服务主要用于个人学习测试使用。服务器会进行定期进行维护。在维护过程中，本服务将出现短暂不可用情况，且服务器会断开所有设备连接。

为了保证本服务稳定运行，请您在使用时遵守以下要求。

- MQTT信息体 (Payload) 大小请不要超过100字节，否则服务器会拒绝接收。
- 请控制您的客户端信息发布频率不要过高。向同一主题发布信息的频率请不要高于每秒1次。

为了确保MQTT服务器正常运行，MQTT服务器管理员会封禁违反以上规则的客户端IP。

图 33.3.20 然也物联公用 MQTT 服务器使用注意事项

以上便是本小节的全部内容了，下小节我们将进一步来学习 MQTT 协议相关的理论知识！

33.4 MQTT 协议（下）

在 33.2 小节中给大家介绍了 MQTT 通信的基本原理，本小节我们将进一步来学习 MQTT 协议相关的理论知识，包括 QoS 的概念、保留消息、MQTT 心跳机制、遗嘱的概念、以及用户密码认证等内容，废话不多说，直接开始吧！

33.4.1 QoS 是什么？

前面已经多次提及到 QoS 这个概念，但当时并未向大家解释 QoS 的概念。本小节我们就来解释下 QoS 究竟是什么！

QoS 是什么?

QoS 是 Quality of Service 的缩写, 所以中文名便是服务质量。一个物联网通信中有些信息非常重要, 我们需要确保这类重要信息可以准确无误的发送和接收, 而有些信息则相对不那么重要, 这类信息如果在传输中丢失不会影响系统的运行; QoS 便用于告诉客户端或服务器哪些信息是重要信息, 需要准确无误的传输、不可丢失; 哪些信息不是那么重要, 即使在传输过程中丢失也无妨!

MQTT 设计了一套保证消息稳定传输的机制, 包括消息应答、存储和重传。在这套机制下, 提供了三种不同级别的 QoS (Quality of Service), 也就是 MQTT 协议有三种服务质量等级:

- QoS = 0: 最多发一次;
- QoS = 1: 最少发一次;
- QoS = 2: 保证收一次。

以上三种不同的服务质量级别意味着不同的 MQTT 传输流程。对于较为重要的 MQTT 消息, 我们通常会选择 QoS>0 的服务级别 (即 QoS 为 1 或 2)。

另外这里提到的“发”与“收”有两种可能。一种是客户端发布消息时, 将消息发送给服务端。一种是客户端订阅了某一主题消息后, 服务端将消息发送给客户端。因此发布消息和接收消息的可能是服务端也可能是客户端。为了避免为您造成混淆, 我们在本节教程后面的描述中将使用“发送端”来描述发送 MQTT 消息的设备, 而使用“接收端”来描述接收 MQTT 消息的设备。

接下来我们仔细看一下这三种服务质量级别的具体含义。

QoS = 0: 最多发一次

0 是服务质量 QoS 的最低级别。当 QoS 为 0 级时, MQTT 协议并不保证所有信息都能得以传输。也就是说, QoS=0 的情况下, MQTT 服务端和客户端不会对消息传输是否成功进行确认和检查。消息能否成功传输全看网络环境是否稳定。

也就是说发送一次之后就不管了, 最多一次, 不管发送是否失败! 发送端一旦发送完消息后, 就完成任务了, 发送端不会检查发出的消息能否被正确接收到。

在网络环境稳定的情况下, 信息传输一般是不会出现问题的。这完全依赖于 TCP 重传机制, 如果网络不好, TCP 的重传也不是 100% 可靠, 加上 MQTT 是发送方发出去的消息是依赖代理服务器完成转发的, 所以消息最多一次。

QoS = 1: 最少发一次

当 QoS 级别为 1 时, 发送端在消息发送完成后, 会检查接收端是否已经成功接收到了消息, 如下图所示:

QoS = 1

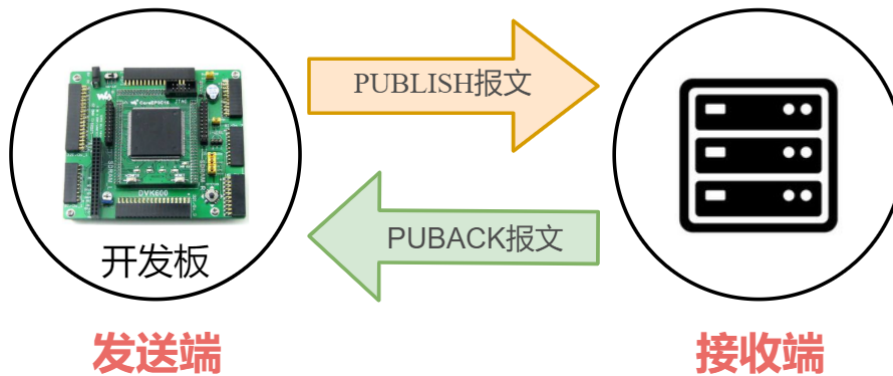


图 33.4.1 QoS1 消息传输情况

发送端向接收端发送 PUBLISH 报文,当接收端收到 PUBLISH 报文后会向发送端回复一个 PUBACK 报文,如果发送端收到 PUBACK 报文,那么它就知道消息已经被接收端成功接收!

假如过了一段时间后,发送端没有收到 PUBACK 报文,那么发送端会再次发送消息(发送 PUBLISH 报文),然后再次等待接收端的 PUBACK 确认报文。因此,当 QoS=1 时,发送端在一定时间内没有收到接收端的 PUBACK 确认报文,会重复发送同一条消息。

所以 QoS=1 时,每一条消息就至少会传输一次,但也可能会重复传输多次。当发送端重复发送一条消息时,会将 PUBLISH 报文中的 dup 标志设置为 true,如图 33.2.8 中所示。这就是为了告诉接收端,此消息为重复发送的消息,那么我们的 MQTT 客户端在接收到消息之后,可以去判断 dup 标志以确定此消息是否为重复消息,应用程序应该对此作出相应的处理。

注意: QoS=1 时, MQTT 服务器是不会进行去重的,只要发布者或者服务器没有收到 PUBACK 报文,就认为主题消息没有发送成功进入重发;服务器或者订阅者,不会根据 dup 标志的值进行去重(也就是说协议本身不会去重),需要我们的客户端应用程序去进行判断、处理。

QoS = 2: 保证收一次

MQTT 服务质量最高级是 2 级,即 QoS=2。当 MQTT 服务质量为 2 级时, MQTT 协议可以确保接收端只接收一次消息(注意是只接收到一次,在 QoS=1 的情况下,接收端接收到消息的次数可能不止一次: ≥ 1)。

为了确保接收端只接收到一次消息, PUBLISH 报文的收发过程相对更加复杂。发送端需要接收端进行两次消息确认,因此, 2 级 MQTT 服务质量是最安全的服务级别,也是最慢的服务级别。我们来看看整体的过程:

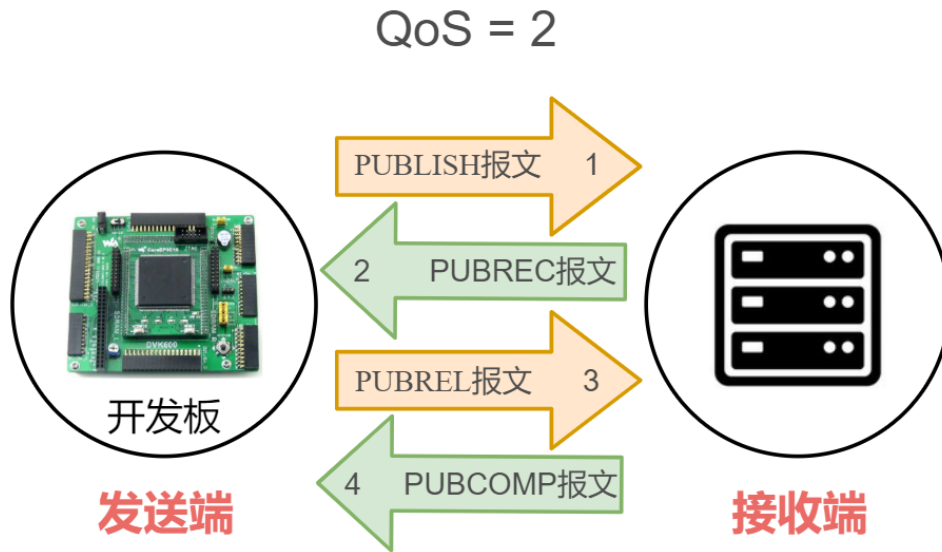


图 33.4.2 QoS2 消息传输情况

从上到下, 按照 1、2、3、4 的顺序进行:

- ①、首先发送端向接收端发送 PUBLISH 报文;
- ②、接收端接收到 PUBLISH 报文后, 向发送端回复一个 PUBREC 报文(官方称其为--发布收到);
- ③、发送端接收到 PUBREC 报文后, 会再次向接收端发送 PUBREL 报文(官方称其为--发布释放);
- ④、接收端接收到 PUBREL 报文后, 会再次向发送端回复一个 PUBCOMP 报文(官方称其为--发布完成), 如果发送端接收到 PUBCOMP 报文表示消息传输成功, 它确认接收端已经成功接收到消息, 整个过程结束!

以上只是列出了 QoS2 消息传输的基本流程, 那 MQTT 协议是如何保证接收端只能接收到一次消息的呢? 关于这个问题我们不再向大家进行介绍, 因为这些都是由 MQTT 协议来控制完成的, 对于应用编程来说, 我们并不需要知道其内部实现的机制和原理。当然, 如果大家有兴趣, 可以自己百度了解!

所以您只需要牢记一点, 那就是 QoS=2 可以保证接收端只收一次消息。

如何设置 QoS?

了解了 QoS 之后, 我们该如何在 MQTT 通信中设置 QoS 等级呢? 其实非常简单, 发布消息和订阅主题时都可以设置 QoS, 前面都已经给大家讲过了; PUBLISH 报文中包含了 qos 参数, 用于设置客户端发布消息时使用的 QoS 级别; 同理, SUBSCRIBE 报文中也包含了 qos 参数, 用于设置客户端订阅主题时使用的 QoS 级别。

换句话说, 无论是发布(PUBLISH)还是订阅(SUBSCRIBE), 都可以使用数据包中的 qos 参数设置服务质量级别。在前面我们使用的 MQTT.fx 客户端软件中也可以体现出来:

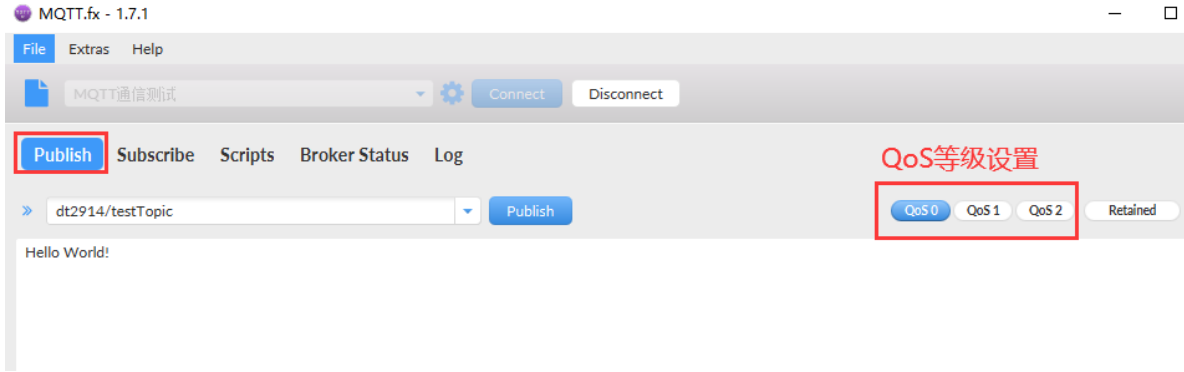


图 33.4.3 MQTT.fx 发布消息时设置 QoS

点击选中 QoS0 时表示发布消息时将 QoS 等级设置为最低级 0。
点击选中 QoS1 时表示发布消息时将 QoS 等级设置为 1。
点击选中 QoS2 时表示发布消息时将 QoS 等级设置为最高级 2。

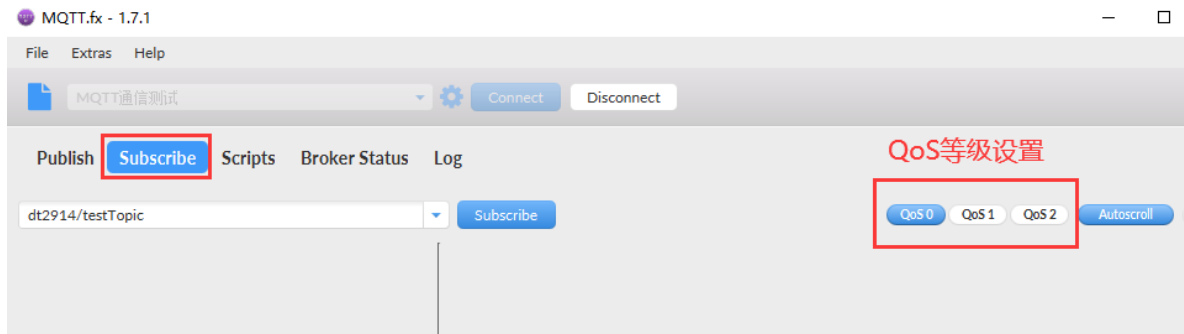


图 33.4.4 MQTT.fx 订阅主题时设置 QoS

点击选中 QoS0 时表示订阅主题时将 QoS 等级设置为最低级 0。
点击选中 QoS1 时表示订阅主题时将 QoS 等级设置为 1。
点击选中 QoS2 时表示订阅主题时将 QoS 等级设置为最高级 2。

另外, 要想实现 $QoS > 0$ 的 MQTT 通信, 客户端在连接服务端时务必要将 `cleanSession` 设置为 `false`。如果这一步没有实现, 那么客户端是无法实现 $QoS > 0$ 的 MQTT 通信, 因为如果 `cleanSession` 设置为 `true`, 则意味着客户端不会接收到任何离线消息, 包括 `QoS1` 和 `QoS2` 的情况。所以这一点非常关键, 请您务必要留意。

服务质量降级

讲到这里, 不知道有没有朋友会感到好奇。假如客户端在发布消息和订阅主题时使用不同级别的 QoS, 将会发生什么情况呢。如下图所示, 假如客户端 A 发布到主题 1 的消息是采用 $QoS=2$, 然而客户端 B 订阅主题 1 采用 $QoS=1$ 。那么服务端该如何来应对这一情况呢?



图 33.4.5 MQTT 发布/订阅示例 1

在这种情况下，服务端会使用较低级别 QoS 来提供服务。如上图所示，虽然 A 发送到主题 1 的消息采用 QoS 为 2，但是服务端发送主题 1 的消息给 B 时，采用的 QoS 为 1。这是因为 B 在订阅主题 1 时采用的 QoS 为 1。

总之，对于发布和订阅消息的客户端，服务端会主动采用较低级别的 QoS 来实现消息传输。

33.4.2 保留消息

前面我们提到，PUBLISH 报文中有一个 `retain` 标志，也就是保留标志，是一个布尔值，当 `retain` 设置为 `true` 时表示保留消息，如果设置为 `false` 表示不保留消息。那保留消息到底是什么意思、有什么特殊用途呢？这些答案将在本小节揭晓！

保留消息的作用

要说明“保留消息”这一概念，我们先看一个场景。假设我们正在利用 MQTT 协议开发一套智能家居物联网系统，该系统中有一台专门用于检测和发布室温信息的 MQTT 客户端，它每到整点时（也就是每隔一个小时）就会测量当前室温并且向 MQTT 服务端发布室温测量结果。

假设在该智能家居物联网系统中，还有一台信息显示客户端。这台客户端的作用就是把当前的室温显示在屏幕上以便我们实时了解室内温度。换句话说，这台信息显示客户端一启动就会订阅室温主题，这样室温检测客户端一发布消息，显示客户端就能获取到最新的温度消息并显示在屏幕上。

假设某天上午 7:00，我们的室温检测客户端将最新的室温消息发布到了服务端，那么订阅了室温消息的显示客户端也就马上获取到室温消息并且显示在屏幕上。然而在 7:05 分的时候，由于一些原因导致显示客户端发生了重启，显示客户端重启之后、再次启动程序后立刻订阅室温主题。

但这时候问题出现了，室温测量客户端每到整点才发布一次温度信息。上一次发布时间是 7:00，下一次发布时间是 8:00。所以，尽管显示客户端又重新订阅了室温主题，它还要等到 8:00 钟才能收到最新室温消息。在 8:00 前的几十分钟里，显示客户端无法获知当前室温信息，也就无法将室温信息显示在屏幕上供我们查阅。

为了避免以上情况出现，我们可以让室温测量客户端在每次向室温主题发布消息时将 `retain` 标志设置为 `true`，以告诉服务端接收到此消息之后需要保留这个消息，这样服务端就会将该消息进行存储、保留，无论显示客户端在任何时间订阅室温主题，订阅之后都会马上收到该主题中的“保留消息”，也就是温度测量客户端发布的最新室温消息。

这就是保留消息的作用，其实非常简单，就是让服务端对客户端发布的消息进行保留，如果有其它客户端订阅了该消息对应的主题时，服务端会立即将保留消息推送给订阅者，而不必等到发送者向主题发布新消息时订阅者才会收到消息。

更新保留消息

但是需要注意的是，每一个主题只能有一个“保留消息”，如果客户端想要更新“保留消息”，就需要向该主题发送一条新的“保留消息”，这样服务端会将新的“保留消息”覆盖旧的“保留消息”。当有客户端订阅该主题时，服务端就会将最新的“保留消息”发送给订阅者。

删除保留消息

如果我们要删除保留消息又该怎么做呢？其实非常简单，只需要向该主题发布一条空的“保留消息”，即可。

使用 MQTT.fx 客户端进行测试

以上给大家详细讲解了 MQTT 协议中“保留消息”的作用，以及如何更新或删除保留消息，那接下来我们将使用 MQTT.fx 客户端进行测试、验证。

首先打开 MQTT.fx 客户端软件，连接好服务器，接下来进行测试，我们先向某个主题发布消息，譬如“dt2914/testTopic”主题：

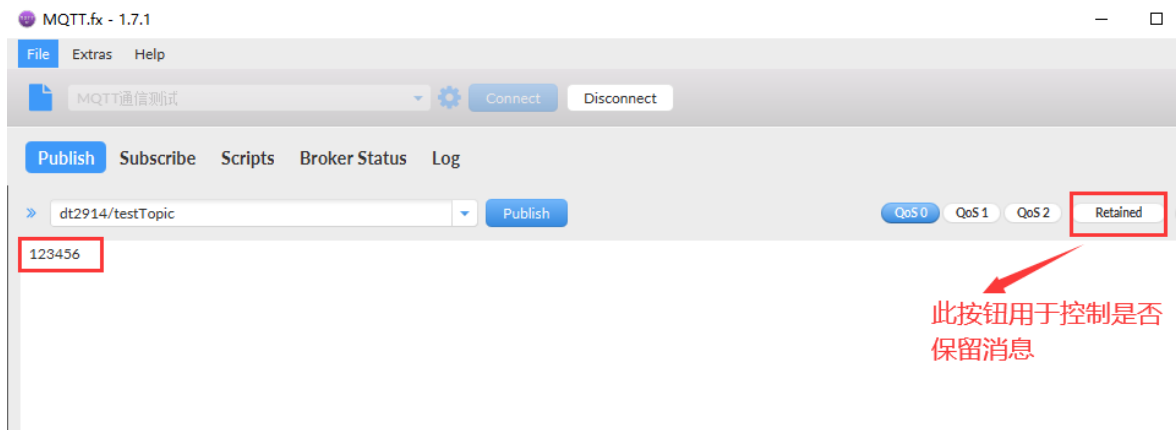


图 33.4.6 发布消息（不保留）

上图中右边的“Retained”按钮也就是保留标志，当点击选中时表示使能“保留消息”这一功能，如果未选中，则表示禁用“保留消息”这一功能，也就是消息不保留。我们先不保留消息，填写好需要发送的内容之后，点击“Publish”按钮发布消息。

现在我们订阅“dt2914/testTopic”主题：

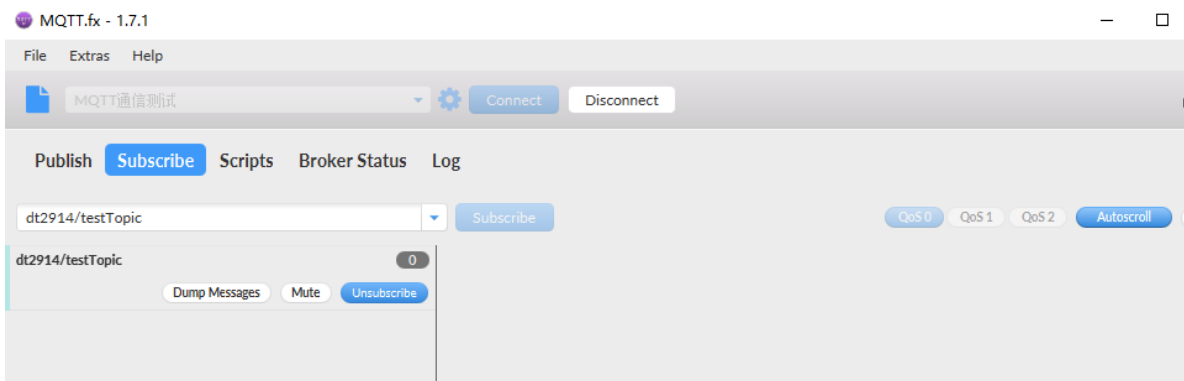


图 33.4.7 订阅主题

订阅之后我们的客户端并没有接收到前面发布的消息，因为发布在前、订阅在后，必须要等到发布者下一次向该主题发布新消息时才会收到。

现在取消订阅主题“dt2914/testTopic”，然后再向主题“dt2914/testTopic”发布消息，此时我们使能“保留消息”这一功能，如下：

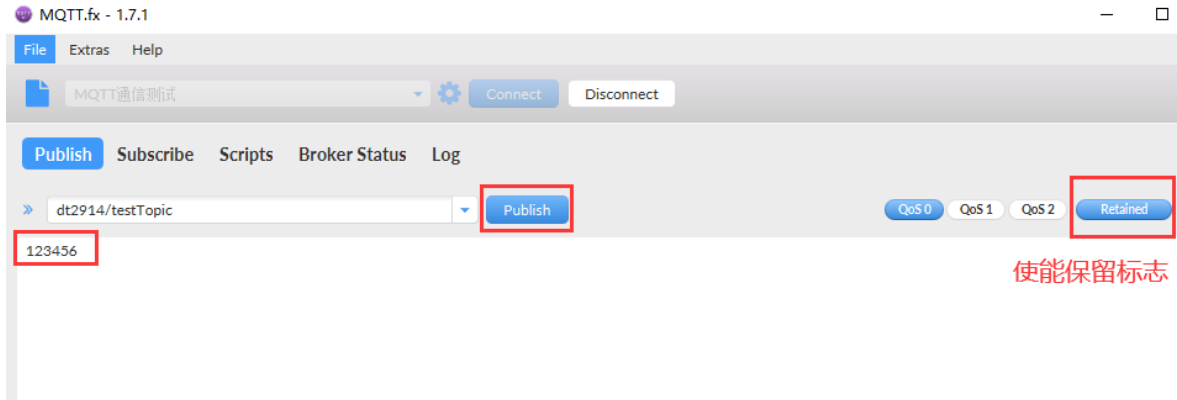


图 33.4.8 发布消息（保留消息）

点击 Publish 按钮发布消息，之后我们再重新订阅主题“dt2914/testTopic”，如下：

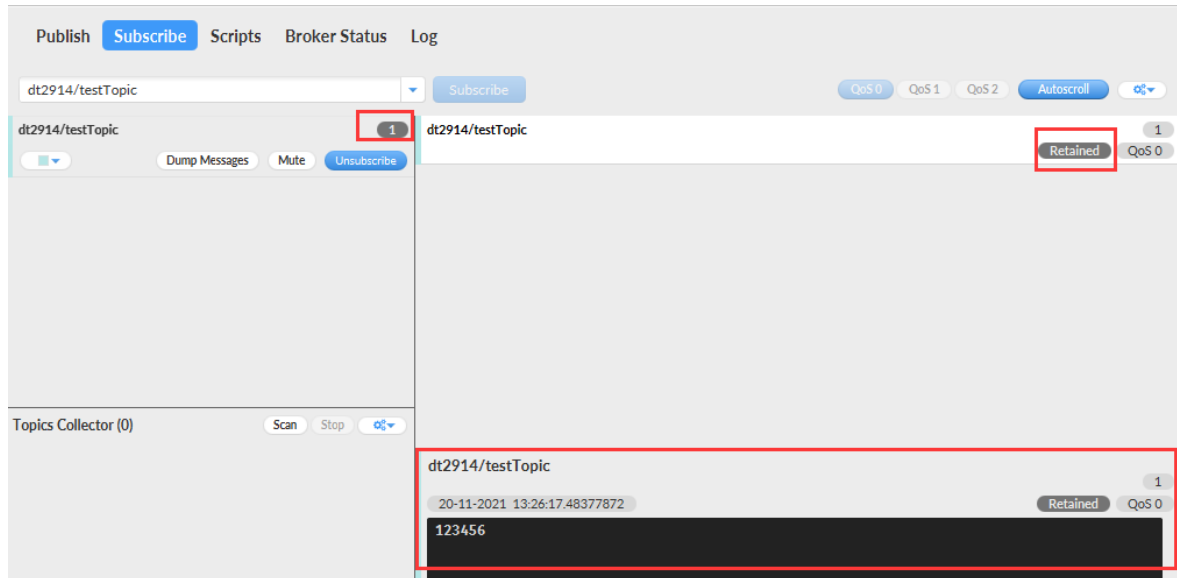


图 33.4.9 订阅主题

当订阅之后我们的客户端立马就收到了一条消息，并且还标识了这条消息是“保留消息”。这就是“保留消息”的作用。除此之外，大家还可以自己测试更新/删除保留消息，这里就不再演示了。

33.4.3 MQTT 的心跳机制

在医院里，医生利用心跳来判断患者是否还有生命体征。对于 MQTT 服务器来说，它要判断 MQTT 客户端是否依然与它保持着连接，就是检查该客户端是不是经常给它发送数据包，如果经常收到客户端的消息，那么证明该客户端肯定在线。

但是有些客户端并不经常发送消息给服务端，譬如有些客户端每隔一天或没隔一个小时才会向服务端发送消息，除此之外，还有些客户端甚至不会向服务端发送消息，它只订阅主题、负责接收服务端发送给它的消息；那么对于这种客户端，MQTT 协议使用类似心跳检测的方法，来判断客户端是否在线，这就是我们说的心跳机制。

心跳机制的原理就在于：让客户端在没有向服务端发送消息的这个空闲时间里，定时向服务端发送一个心跳包，这个心跳包被称为心跳请求，其实质就是向服务端发送一个 PINGREQ 报文；当服务端收到

PINGREQ 报文后就知道该客户端依然在线, 然后向客户端回复一个 PINGRESP 报文, 称为心跳响应! 如下图所示:

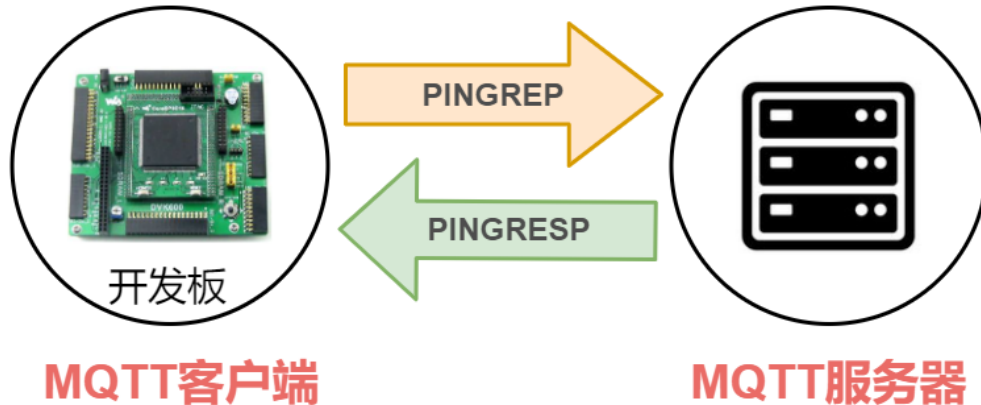


图 33.4.10 心跳请求

由于心跳请求是定时发送的(通过 keepAlive 设置时间间隔, 也是告诉服务端, 客户端将会多少多少秒向它发送心跳请求, 这样服务端就会知道了); 一旦服务器未收到客户端的心跳包, 那么服务器就会知道, 这台客户端可能已经掉线了。

这个心跳机制不仅可以用于服务端判断客户端是否在线, 客户端也可使用心跳机制来判断自己与服务端是否保持连接。如果客户端在发送心跳请求(PINGREQ)后, 没有收到服务端的心跳响应(PINGRESP), 那么客户端就会认为自己与服务端已经断开连接了。

33.4.4 MQTT 的遗嘱机制

“遗嘱”, 大家听到这个词是不是感觉跟死亡有关系, 事实确实如此, 不过在我们的 MQTT 协议中, 死亡指的是“客户端掉线”、“与服务端断开了连接”这种意思。

客户端断开与服务端的连接通常是有两种方式的:

- 客户端主动向服务端发送 DISCONNECT 报文, 请求断开连接, 自然服务端也就知道了客户端要离线了;
- 客户端**意外掉线**。被动与服务端断开了连接。

客户端意外掉线这种情况对于物联网设备来说是比较常见的, 也是必须要考虑在内的情况。前面我们在介绍 MQTT 协议的时候就提到过, MQTT 从诞生之初就是专为低带宽、高延迟或不可靠网络等环境而设计的; 所以针对这种意外掉线的情况, MQTT 协议使用了遗嘱机制来服务客户端、管理客户端。

MQTT 协议允许客户端在“活着”的时候就写好遗嘱, 这样一旦客户端**意外断线**, 服务端就可以将客户端的遗嘱公之于众。请注意, 在这段话中, 我将**意外断线**这几个字特意做了加粗处理, 这是因为, 客户端的遗嘱只在意外断线时才会发布, 如果客户端正常的断开了与服务端的连接(主动断开), 这个遗嘱机制是不会启动的, 服务端也不会将客户端的遗嘱公布。

那什么是意外断线? 其实除了客户端主动向服务端发送 DISCONNECT 报文请求断开连接这种情况之外, 其它断线的情况都属于意外断开连接, 譬如网络不稳定、客户端设备没电关机了等。

客户端如何设置自己的“遗嘱”信息

客户端连接服务端时发送的 CONNECT 报文中有这样几个参数, 如下图红框中所示:

CONNECT 报文	
名称	内容
clientID	"client-id"
keepAlive	60
cleanSession	true
willTopic(可选)	"clientWill"
willMessage(可选)	"client offline"
willRetain(可选)	false
willQos(可选)	0
username(可选)	"myname"
password(可选)	"dt29144"
.....

图 33.4.11 CONNECT 报文信息

这几个参数都是以 will 开头的，will 其实就是“遗嘱”的英文单词，所以由此可知，遗嘱需要在客户端连接服务端时就需要设置好，下面分别介绍一下：

willTopic -- 遗嘱主题

遗嘱消息和普通 MQTT 消息很相似，也有主题和正文内容。willTopic 的作用正是告知服务端，本客户端的遗嘱主题是什么。只有那些订阅了这一遗嘱主题的客户端才会收到本客户端的遗嘱消息。

以上图为例，此遗嘱主题为“clientWill”，也就是说，只有订阅了主题“clientWill”的客户端，才会收到这台客户端的遗嘱消息。当然，客户端也可以主动向遗嘱主题发布消息，这样通常会有一些妙用，譬如当客户端 A 上线时可以向自己的遗嘱主题发布一条消息，那么那些订阅了该遗嘱主题的客户端可以收到这条消息，这些订阅者也就知道了客户端 A 已经上线了。

所以这个可以用来实现一个上线通知的小功能。

willMessage -- 遗嘱消息

遗嘱消息定义了遗嘱的内容。在本示例中，那些订阅了主题“clientWill”的客户端会在客户端意外断线时，收到服务端发布的“client offline”这样的信息。

willRetain -- 遗嘱消息的保留标志

遗嘱消息也可以设置为保留标志，用于告诉服务端是否需要保留遗嘱消息。

willQoS -- 遗嘱消息的 QoS

对于遗嘱消息来说，同样可以使用服务质量来控制遗嘱消息的传递和接收。这里的服务质量与普通 MQTT 消息的服务质量是一样的概念。也可以设置为 0、1、2。对于不同的服务质量级别，服务端会使用不同的服务质量来发布遗嘱消息。

MQTT.fx 如何设置客户端的“遗嘱”

MQTT.fx 软件如何为电脑客户端设置遗嘱呢？首先进入到配置页面中，如下所示：

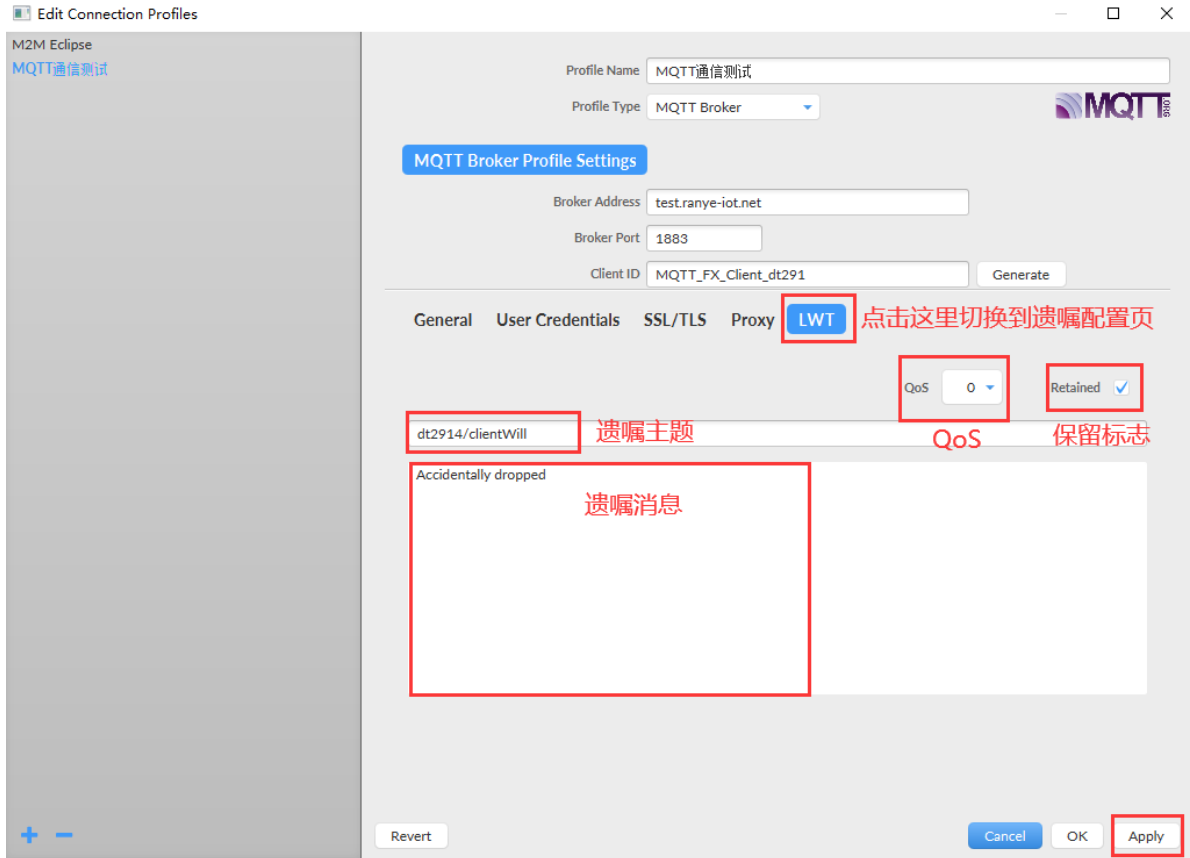


图 33.4.12 MQTT.fx 软件遗嘱设置

33.4.5 MQTT 用户密码认证

到目前为止，CONNECT 报文中还有两个参数没给大家介绍，如下所示：

CONNECT报文	
名称	内容
clientID	"client-id"
keepAlive	60
cleanSession	true
willTopic(可选)	"clientWill"
willMessage(可选)	"client offline"
willRetain(可选)	false
willQos(可选)	0
username(可选)	"myname"
password(可选)	"dt29144"
.....

图 33.4.13 CONNECT 报文

也就是上图中红框所示的两个参数: `username` (用户名) 和 `password` (密码), 这里的用户名和密码是客户端连接服务端时进行认证所需要的。

有些 MQTT 服务端需要客户端在连接时提供用户名和密码, 只有客户端正确提供了用户名和密码后, 才能连接服务端, 否则服务端将会拒绝客户端连接, 那么客户端也就无法发布和订阅消息了。但有些 MQTT 服务端并不需要客户端提供用户名、密码进行认证, 譬如我们前面使用的公用 MQTT 服务器, 无需提供用户名和密码也可连接。所以这个 `username` 和 `password` 是可选的参数, 而非必须的, 重点在于 MQTT 服务端是否需要用户名、密码认证。

有些 MQTT 服务端开启了用户名、密码认证, 这种服务端需要客户端在连接时正确提供用户名、密码认证信息才能连接成功; 当然, 那些没有开启用户名、密码认证的服务端无需客户端提供用户名和密码认证信息。

接下来我们使用 MQTT.fx 来测试一下, 笔者使用的是然也物联提供的公共版 MQTT 服务器, 这个服务器提供了一个测试用的用户名和密码, 用户名是 `test-user`、密码是 `ranye-iot`, 那我们使用这个用户名和密码连接服务端试试:

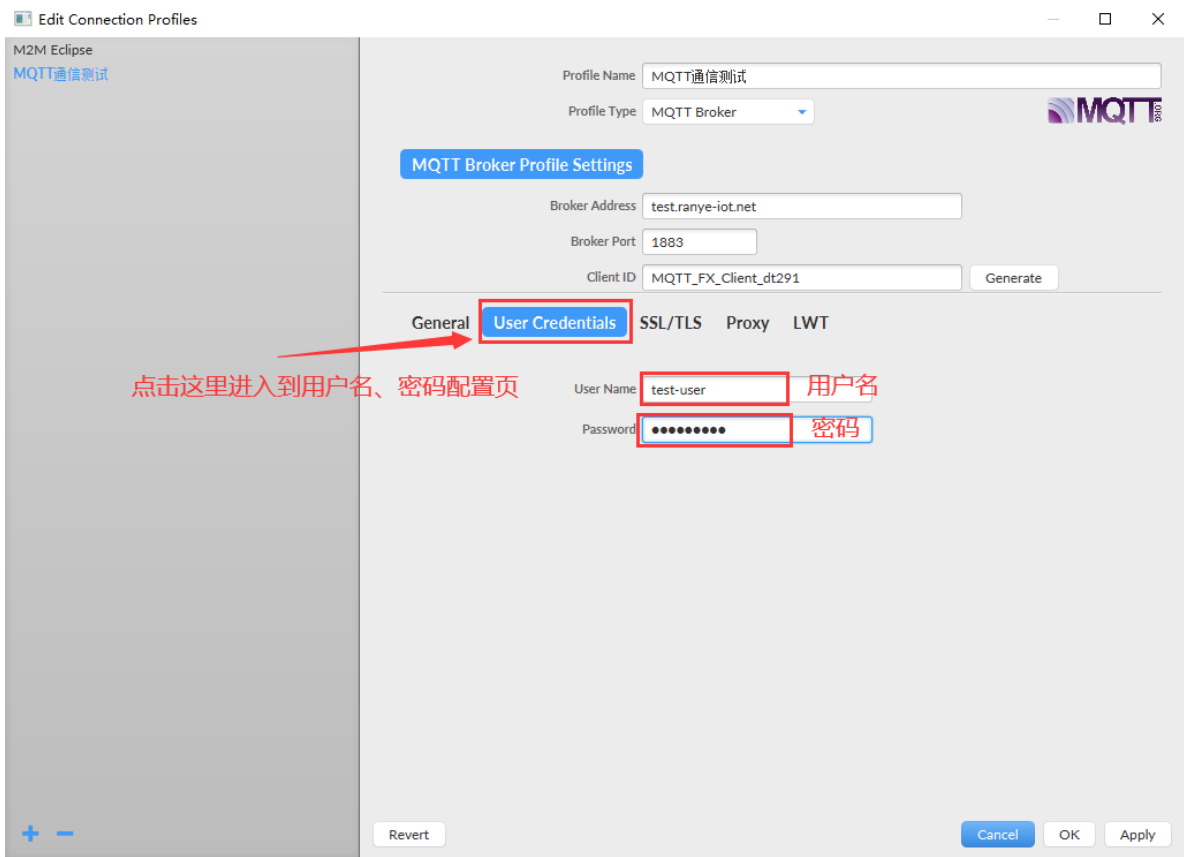


图 33.4.14 用户名密码配置

填写完用户名、密码之后点击右下角“Apply”按钮应用, 然后关闭配置窗口。然后点击 `Connect` 按钮连接服务端:

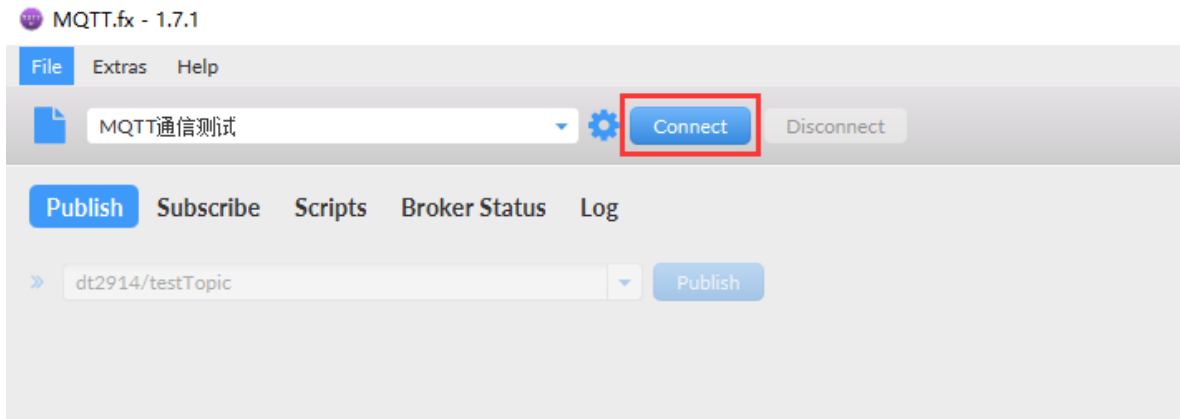


图 33.4.15 点击 Connect 按钮连接服务端

点击 Connect 按钮连接服务器，就会成功连接上。在手机上使用 MQTTTool 工具也可以，大家自己去试试。

用户名和密码除了用于在连接服务端时进行认证、校验这一功能外，有些 MQTT 服务端也利用此信息来识别客户端属于哪一个用户，从而对客户端进行管理。譬如用户可以拥有私人主题，这些主题只有该用户可以发布和订阅；对于私人主题，服务端就可以利用客户端连接时的用户名和密码来判断该客户端是否有发布订阅该用户私人主题的权限。

申请社区版 MQTT 服务

在前面的示例中，我们都是使用的公共版 MQTT 服务器进行了测试，既然是公共版，那就意味着大家都可以使用，只要输入了正确的服务器地址就可以连接，大家都可以对相同的主题发布消息、订阅该主题，导致我们发布的信息谁都能看到（只要它订阅了这个主题），这对物联网项目的安全性以及实用性方面来说都是不合适的，所以这些公用 MQTT 服务也仅供用于测试、学习！

事实上，**然也物联平台**也提供了免费的社区版 MQTT 服务，社区版 MQTT 服务是面向**个人用户**的免费 MQTT 服务。与公共版 MQTT 服务不同的是，社区版 MQTT 服务中，用户个人主题和信息传输受到用户名和密码保护。即，A 用户的个人主题只有 A 用户可以发布和订阅，其他用户不能对该主题进行订阅和发布，这样会使得安全性得到提升。

当客户端连接社区版 MQTT 服务端时，需要提供正确的用户名和密码，服务端会对此进行验证，如果没有提供正确的用户名、密码信息，则服务器将拒绝为用户提供 MQTT 服务，也就是拒绝客户端连接。所以，我们个人用户可使用**然也物联**的社区版 MQTT 服务来搭建自己的私人物联网项目，**注意仅限于个人用户使用，不可商用！**

那接下来，笔者将向大家介绍如何通过**然也物联平台**申请社区版 MQTT 服务。

首先进入到**然也物联**的官方网站：<http://www.ranye-iot.net/>



图 33.4.16 然也物联官方网址

点击上边的“注册用户”注册一个用户：

注册用户

登录名 ?

姓

名

电子邮件

密码

确认密码

图 33.4.17 用户注册

大家根据指示填写信息，完成用户注册。

注册完成之后登陆然也物联平台，登陆成功之后如下所示：



图 33.4.18 登陆平台

左上角会出现一个“平台申请”链接，点击“平台申请”链接即可申请然也物联的社区版 MQTT 服务，如下所示：

平台申请

本平台功能测试运行中!

以下申请表格为然也物联社区版MQTT服务申请表。如需了解然也物联社区版的详细信息, [请点击此处](#)。

申请前请留意:

在审核您的申请资料时, 我们需要与您进行微信沟通, 因此在申请然也物联社区版平台时, 需要提供您的微信手机号码, 我们将通过然也物联官方微信号加您为微信好友, 完成申请流程。

每个账户只能提交一次社区版服务申请。如果您已经提交了申请, 请耐心等待我们的审核, 不要重复申请。感谢您对然也物联的关心与信任!

微信手机号码 *

在审核您的申请资料时, 我们需要与您进行微信沟通。请正确提供您的微信手机号码。一个手机号码只能注册一次。请勿反复提交申请。如您已经提交申请, 请耐心等待审核结果。

连接平台密码 *

使用物联网硬件设备连接然也物联平台时所需的认证密码。仅支持英文字母和数字。请不要超过8个字符。不要输入中文或特殊字符。 0 / 8

正确回答以下问题后, 本申请表格的“提交申请”按钮才会显示。

然也物联平台使用何种物联网协议实现物联网设备通讯? (回答时注意大小写不要写错) *

您必须正确回答问题方可提交表格!

MQTT主题单级通配符的符号是? *

您必须正确回答问题方可提交表格!

QoS最高级别是? (请使用阿拉伯数字作答) *

您必须正确回答问题方可提交表格!

用户协议和隐私政策 *

您已仔细阅读并且同意[用户协议](#)和[隐私政策](#)?

图 33.4.19 社区版 MQTT 服务申请

同样, 大家根据指示说明填写好信息。在填写信息之前, 仔细阅读相关说明以及相应的要求, 最后有三个题目需要大家填写正确, 只有正确回答所有问题最终才会显示“提交申请”这个按钮。这样做为了防止申请服务的用户是真正需要使用 MQTT 服务的用户、而不是随随便便的一个用户。

当我们提交申请之后, 页面会出现一个提示信息, 这个大家要认真看一下, 提示中说到: 然也物联官方工作人员会在未来几天之内添加你申请时留下的微信号, 以人工的形式进一步完成申请审核。所以大家要留意下自己的微信, 未来几天内会不会有然也物联官方工作人员添加你为好友, 到时你要同意一下。

当我们的申请通过之后,官方工作人员会通过微信通知我们,告诉我们申请已经成功了!接着工作人员会将相关的使用注意事项、使用方法通过微信发送给您,大家要认真阅读尤其是注意事项;如果违反了它的规定,将会停止您使用社区版服务。

除了注意事项之外,还包括对于我们使用社区版 MQTT 服务非常重要的信息,譬如社区版 MQTT 服务器的地址 (iot.ranye-iot.net)、端口 (1883) 以及然也物联给我们提供的客户端连接认证信息。

然也物联给每一个申请的用户提供了 8 组客户端连接认证信息,也就是 8 组用户名、密码、clientId,也就是说允许我们同时使用 8 台客户端设备连接服务端;每一台客户端设备连接服务端时使用其中一组用户名、密码、clientId 信息,只有用户名、密码、clientId 匹配、服务端的认证才会通过、才可成功连接到服务端。

社区版 MQTT 服务器为用户提供了个人专属主题级别,只有用户自己的客户端设备才可以使用自己的个人专属主题级别,譬如向个人专属主题发布消息、订阅个人专属主题;而其它用户是无法向您的个人专属主题发布消息、也不能订阅您的个人专属主题,因为社区版 MQTT 服务中个人专属主题级别受到了用户名、密码保护,同样您也不能向其他用户的个人专属主题发布消息以及订阅其他用户的个人专属主题;这样使得我们的 MQTT 物联网通信安全性大大提升!

在后续我们会自己编写一个 MQTT 客户端程序,在我们的开发板上运行,将开发板作为 MQTT 客户端去连接然也物联社区版 MQTT 服务器。

33.5 移植 MQTT 客户端库

前面的示例中,我们使用 MQTT.fx 客户端软件在自己的电脑上进行了测试(或在手机上使用 MQTTool 工具进行测试),如果需要在开发板上进行测试,将开发板作为 MQTT 客户端,我们需要自己去编写客户端程序。

首先在编写客户端程序之前,需要移植 MQTT 客户端库到我们的开发板上,基于 MQTT 客户端库来编写一个 MQTT 客户端应用程序。那么接下来笔者将向大家介绍如何移植 MQTT 客户端库。

下载 MQTT 客户端库源码

如何下载 MQTT 客户端库源码包?首先我们进入到 MQTT 的官网地址: <https://mqtt.org/>

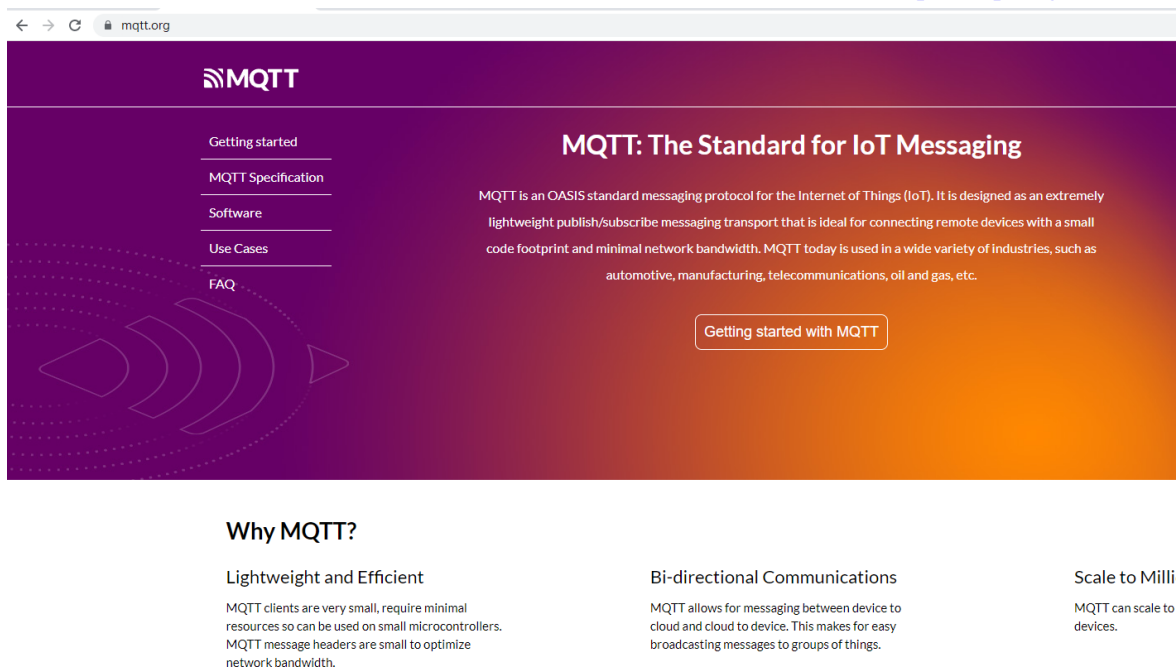


图 33.5.1 MQTT 官网

点击“Software”链接地址,找到“Client libraries”项,如下所示:

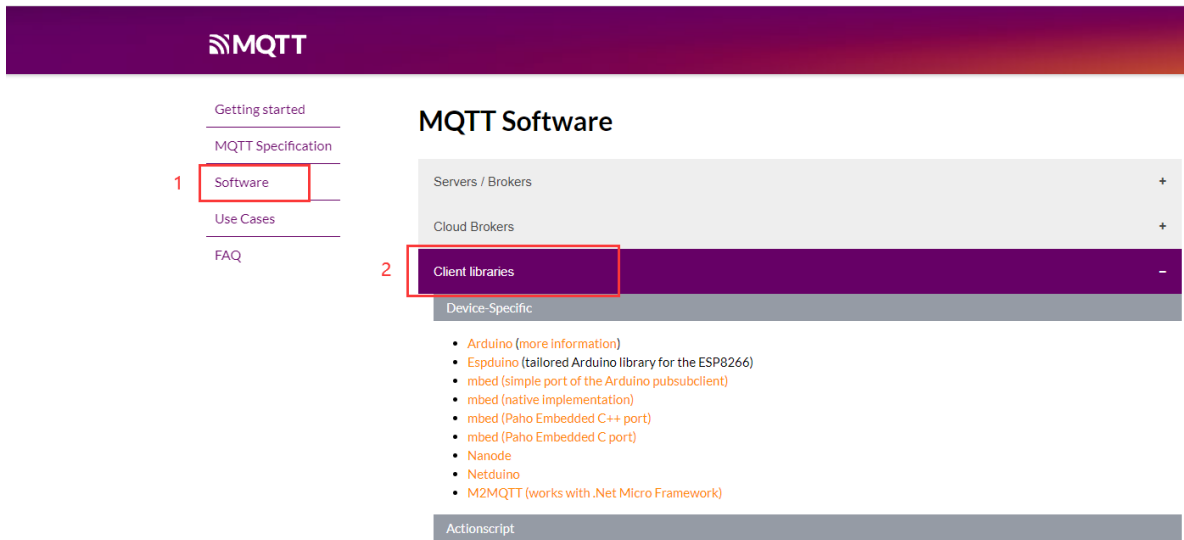


图 33.5.2 Client libraries

MQTT 客户端库支持多种不同的编程语言, 譬如 C、C++、Go、Java、Lua、Objective-C、Python 等, 对于我们来说, 我们使用的是 C 语言开发, 所以要选择 MQTT C 客户端库, 如下所示:

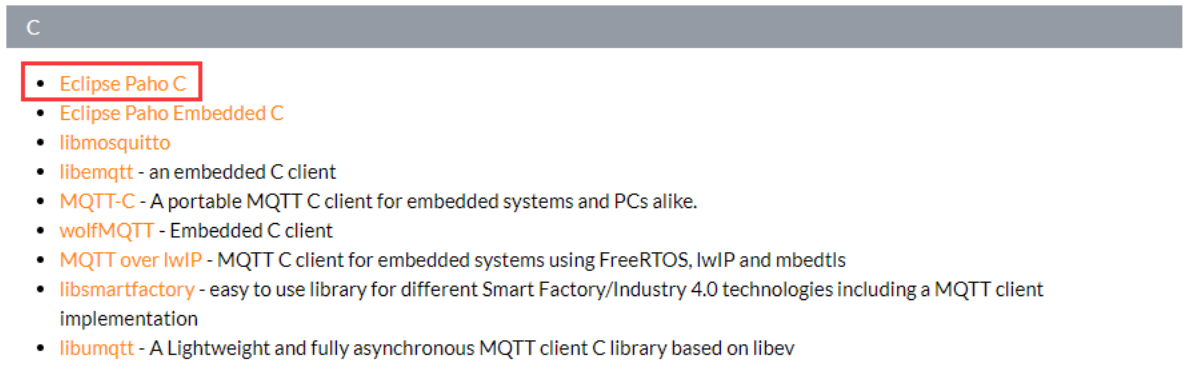


图 33.5.3 MQTT C 客户端库

这里有多种不同的 MQTT C 客户端库, 笔者推荐大家使用第一个 Eclipse Paho C, 这是一个“MQTT C Client for Posix and Windows”, Paho MQTT C 客户端库是用 ANSI 标准 C 编写的功能齐全的 MQTT 客户端库, 可运行在 Linux 系统下, 支持 MQTT3.1、MQTT3.1.1、MQTT5.0。

点击“Eclipse Paho C”链接地址, 如下:

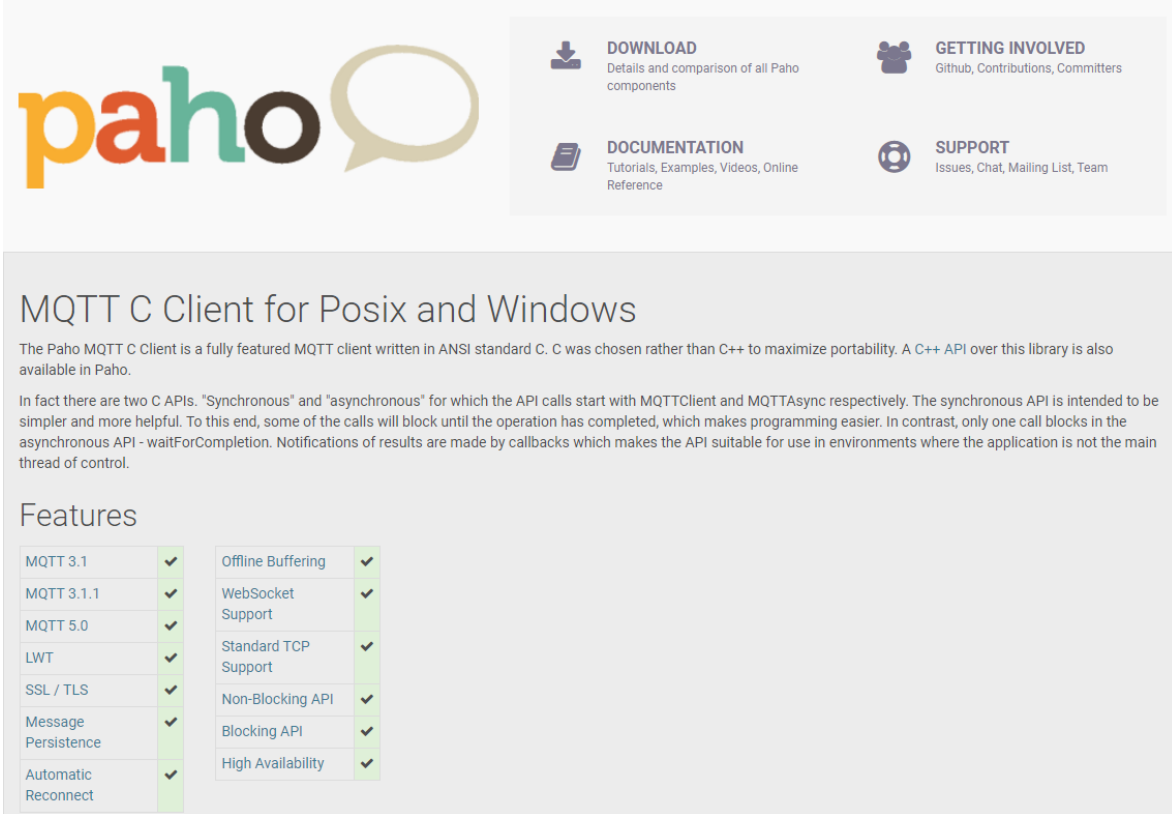


图 33.5.4 Paho MQTT C

在这个页面中会有一些简单地介绍信息，大家可以自己看一看。我们往下看，找到它的下载地址：



图 33.5.5 找到源码链接地址

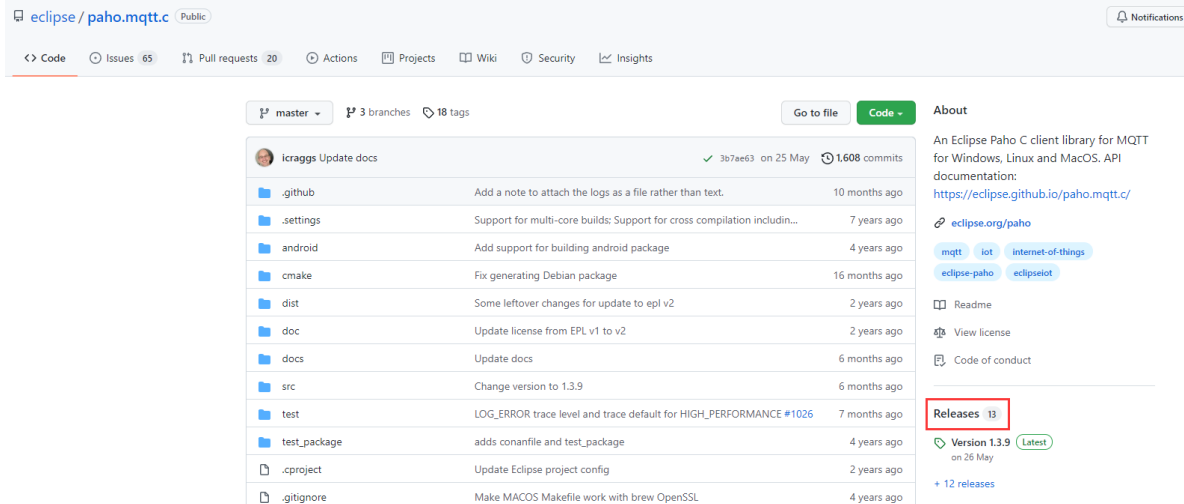


图 33.5.6 git 仓库地址

点击右边的“Release”找到它的发布版本，如下所示：

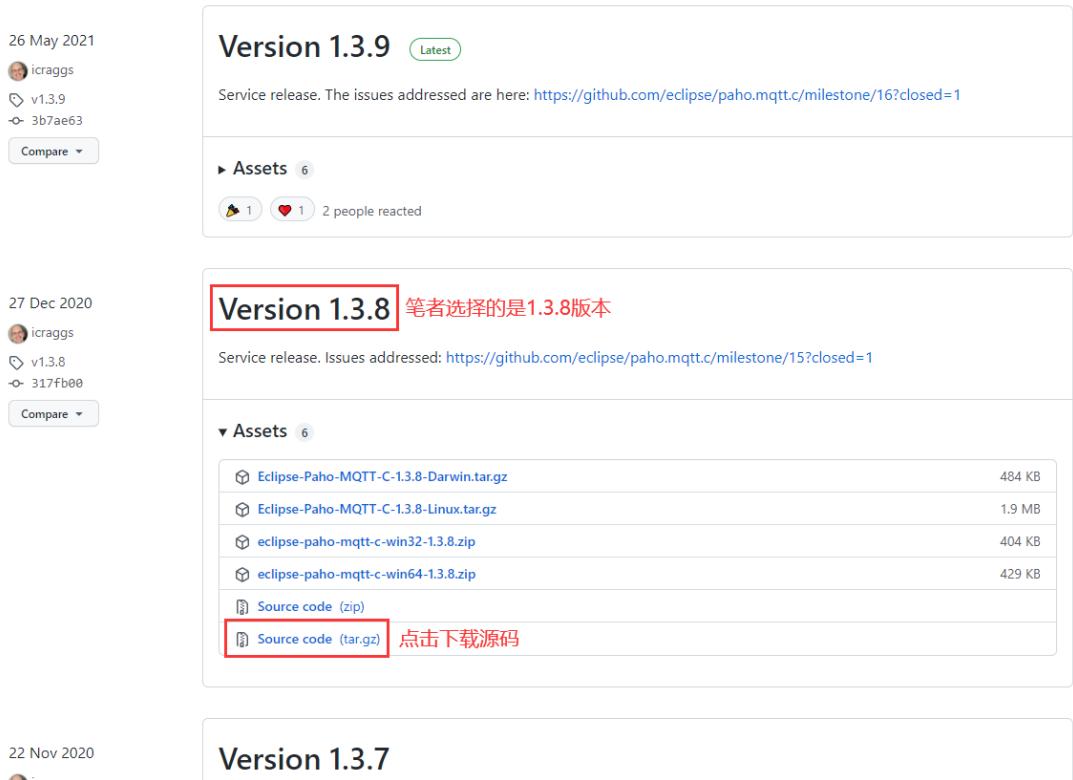


图 33.5.7 Paho MQTT C 客户端库源码下载

目前最新的版本是 1.3.9，我们不使用最新版本，建议大家使用 1.3.8 版本，如上图所示，点击“Source code (tar.gz)”链接地址下载客户端库源码。

下载成功之后会得到如下压缩文件：

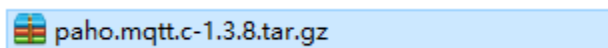


图 33.5.8 paho.mqtt.c 客户端库源码

交叉编译 MQTT C 客户端库源码

将 paho.mqtt.c-1.3.8.tar.gz 压缩文件拷贝到 Ubuntu 系统某个目录下, 如下所示:

```
dt@dt-virtual-machine:~/tools$ pwd
/home/dt/tools
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 paho.mqtt.c-1.3.8.tar.gz png tslib zlib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$
```

图 33.5.9 将 paho.mqtt.c-1.3.8.tar.gz 拷贝到 Ubuntu

接着将其解压到当前目录, 如下所示:

```
dt@dt-virtual-machine:~/tools$ tar -xzf paho.mqtt.c-1.3.8.tar.gz
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 paho.mqtt.c-1.3.8 paho.mqtt.c-1.3.8.tar.gz png tslib zlib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ cd paho.mqtt.c-1.3.8/
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$ ls
about.html build.xml CMakeLists.txt deploy_rsa.enc docs LICENSE PULL_REQUEST_TEMPLATE.md test
android cbuil.build.bat CODE OF CONDUCT.md dist edl-v10 Makefile README.md test_package
appveyor.yml cmake CONTRIBUTING.md doc epl-v20 notice.html src travis-build.sh
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
```

解压成功之后会得到 paho.mqtt.c-1.3.8 文件夹, 这就是 paho MQTT C 客户端库源码工程, 进入到该目录下, 可以看到工程顶级目录下有一个 CMakeLists.txt 文件, 所以可知这是一个由 cmake 构建的工程。

首先我们要新建一个交叉编译配置文件 arm-linux-setup.cmake, 进入到 cmake 目录下, 新建 arm-linux-setup.cmake 文件, 并输入以下内容:

```
#####
# 配置 ARM 交叉编译
#####
set(CMAKE_SYSTEM_NAME Linux) #设置目标系统名字
set(CMAKE_SYSTEM_PROCESSOR arm) #设置目标处理器架构

# 指定编译器的 sysroot 路径
set(TOOLCHAIN_DIR /opt/fsl-ix86-x11/4.1.15-2.1.0/sysroots)
set(CMAKE_SYSROOT ${TOOLCHAIN_DIR}/cortexa7hf-neon-poky-linux-gnueabi)

# 指定交叉编译器 arm-linux-gcc
set(CMAKE_C_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc)

# 为编译器添加编译选项
set(CMAKE_C_FLAGS "-march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
#####
# end
#####
```

这是配置交叉编译, 需要根据自己实际情况修改。

编写完成之后保存退出。

回到工程的顶层目录, 新建一个名为 build 的目录, 如下所示:

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$ pwd
/home/dt/tools/paho.mqtt.c-1.3.8
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$ mkdir build
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$ cd build/
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$
```

图 33.5.10 创建 build 目录

进入到 build 目录下, 执行 cmake 进行构建:

```
~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake -DCMAKE_BUILD_TYPE=Release -
DCMAKE_INSTALL_PREFIX=/home/dt/tools/paho.mqtt.c-1.3.8/install -
DCMAKE_TOOLCHAIN_FILE=./cmake/arm-linux-setup.cmake -DPAHO_WITH_SSL=TRUE -
DPAHO_BUILD_SAMPLES=TRUE ..
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$ ~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/home/dt/tools/paho.mqtt.c-1.3.8/i
ninstall -DCMAKE_TOOLCHAIN_FILE=./cmake/arm-linux-setup.cmake -DPAHO_WITH_SSL=TRUE -DPAHO_BUILD_SAMPLES=TRUE ..
-- The C compiler identification is GNU 5.3.0
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- CMake version: 3.16.0
-- CMake system name: Linux
-- Timestamp is 2021-11-20T12:23:20Z
-- Found OpenSSL: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/cortexa7hf-neon-poky-linux-gnueabi/usr/lib/libcrypto.so (found version "1.0.2h")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/tools/paho.mqtt.c-1.3.8/build
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$
```

图 33.5.11 执行 cmake 构建

~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake 这是第三十二章时笔者下载的 3.16.0 版本的 cmake 工具, 您得根据自己的实际路径来指定; CMAKE_BUILD_TYPE、CMAKE_INSTALL_PREFIX、CMAKE_TOOLCHAIN_FILE 都是 cmake 变量, CMAKE_INSTALL_PREFIX 这个指定了安装路径, 笔者将安装路径设置为顶层目录下的 install 目录。

除此之外, 还定义了两个缓存变量 PAHO_WITH_SSL 和 PAHO_BUILD_SAMPLES, 具体是什么意思大家可以自己查看工程顶级目录下的 README.md 文件, 在 README.md 文件中对工程的编译进行了简单介绍。

cmake 执行完毕之后, 接着执行 make 编译:

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$ make -j10
Scanning dependencies of target common_ssl_obj
Scanning dependencies of target common_obj
Scanning dependencies of target thread
[ 1%] Building C object test/CMakeFiles/thread.dir/thread.c.o
[ 2%] Building C object test/CMakeFiles/thread.dir/_/src/Thread.c.o
[ 3%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTTime.c.o
[ 3%] Building C object src/CMakeFiles/common_ssl_obj.dir/Clients.c.o
[ 4%] Building C object src/CMakeFiles/common_obj.dir/Clients.c.o
[ 5%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTProtocolClient.c.o
[ 6%] Building C object src/CMakeFiles/common_obj.dir/utf-8.c.o
[ 7%] Building C object src/CMakeFiles/common_obj.dir/MQTTTime.c.o
[ 7%] Building C object src/CMakeFiles/common_obj.dir/MQTTProtocolClient.c.o
[ 8%] Building C object src/CMakeFiles/common_ssl_obj.dir/utf-8.c.o
[ 9%] Building C object src/CMakeFiles/common_obj.dir/MQTTPacket.c.o
[ 10%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTPacketOut.c.o
[ 11%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTPacket.c.o
[ 11%] Building C object src/CMakeFiles/common_ssl_obj.dir/Messages.c.o
[ 12%] Linking C executable thread
[ 13%] Building C object src/CMakeFiles/common_ssl_obj.dir/Tree.c.o
[ 14%] Building C object src/CMakeFiles/common_obj.dir/MQTTPacketOut.c.o
[ 15%] Building C object src/CMakeFiles/common_ssl_obj.dir/Socket.c.o
[ 17%] Building C object src/CMakeFiles/common_ssl_obj.dir/Log.c.o
[ 17%] Built target thread
[ 18%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTPersistence.c.o
[ 18%] Building C object src/CMakeFiles/common_ssl_obj.dir/Thread.c.o
[ 19%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTProtocolOut.c.o
[ 19%] Building C object src/CMakeFiles/common_obj.dir/Messages.c.o
[ 20%] Building C object src/CMakeFiles/common_obj.dir/Tree.c.o
[ 21%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTPersistenceDefault.c.o
[ 22%] Building C object src/CMakeFiles/common_ssl_obj.dir/SocketBuffer.c.o
[ 23%] Building C object src/CMakeFiles/common_obj.dir/Socket.c.o
[ 24%] Building C object src/CMakeFiles/common_obj.dir/Log.c.o
[ 25%] Building C object src/CMakeFiles/common_ssl_obj.dir/LinkedList.c.o
[ 25%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTProperties.c.o
[ 26%] Building C object src/CMakeFiles/common_ssl_obj.dir/MQTTReasonCodes.c.o
[ 26%] Building C object src/CMakeFiles/common_obj.dir/MQTTPersistence.c.o
[ 27%] Building C object src/CMakeFiles/common_ssl_obj.dir/Base64.c.o
[ 28%] Building C object src/CMakeFiles/common_ssl_obj.dir/SHA1.c.o
[ 28%] Building C object src/CMakeFiles/common_ssl_obj.dir/WebSocket.c.o
[ 29%] Building C object src/CMakeFiles/common_ssl_obj.dir/StackTrace.c.o
```

图 33.5.12 make 编译

这里需要给大家简单地说明一下，事实上，MQTT 客户端库依赖于 openssl 库，所以通常在移植 MQTT 客户端库的时候，需要先移植 openssl、交叉编译 openssl 得到库文件以及头文件，然后再来编译 MQTT 客户端库；但我们这里没有去移植 openssl，原因是，我们的开发板出厂系统中已经移植好了 openssl 库，并且我们所使用的交叉编译器在编译工程源码的时候会链接 openssl 库（sysroot 路径指定的）。

编译成功之后，执行 `make install` 进行安装：

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$ make install
[ 19%] Built target common_ssl_obj
[ 38%] Built target common_obj
[ 39%] Built target paho-mqtt3c
[ 41%] Built target paho-mqtt3cs
[ 43%] Built target paho-mqtt3a
[ 47%] Built target paho-mqtt3as
[ 50%] Built target MQTTVersion
[ 52%] Built target paho_cs_sub
[ 55%] Built target paho_cs_pub
[ 57%] Built target paho_c_sub
[ 58%] Built target MQTTAsync_subscribe
[ 59%] Built target MQTTClient_subscribe
[ 62%] Built target paho_c_pub
[ 64%] Built target MQTTClient_publish
[ 65%] Built target MQTTAsync_publish
[ 68%] Built target MQTTAsync_publish_time
[ 70%] Built target MQTTClient_publish_async
[ 72%] Built target test45
[ 73%] Built target test8
[ 75%] Built target test4
[ 76%] Built target test3
[ 77%] Built target test5
[ 78%] Built target test10
[ 79%] Built target test15
[ 81%] Built target test1
[ 84%] Built target test6
[ 86%] Built target test2
[ 88%] Built target test_issue373
[ 90%] Built target test9
[ 91%] Built target test95
```

图 33.5.13 make install 安装

对安装目录下的文件夹进行简单介绍

进入到安装目录下:

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/build$ cd ..
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$ ls
about.html      build          cmake          CONTRIBUTING.md  doc            epl-v20       Makefile
android         build.xml     CMakeLists.txt  deploy_rsa.enc  docs           install       notice.html
appveyor.yml   cbuild.bat   CODE_OF_CONDUCT.md  dist            edl-v10       LICENSE      PULL_REQUEST_TEMPLATE.md
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8$ cd install/
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$ ls
bin  include  lib  share
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$
```

图 33.5.14 安装目录

在安装目录下有 bin、include、lib 以及 share 这 4 个文件夹，bin 目录下包含了一些简单的测试 demo，lib 目录下包含了我们编译出来的库文件，如下所示:

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$ ls
cmake          libpaho-mqtt3as.so.1      libpaho-mqtt3cs.so
libpaho-mqtt3a.so  libpaho-mqtt3as.so.1.3.8  libpaho-mqtt3cs.so.1
libpaho-mqtt3a.so.1  libpaho-mqtt3c.so        libpaho-mqtt3cs.so.1.3.8
libpaho-mqtt3a.so.1.3.8  libpaho-mqtt3c.so.1
libpaho-mqtt3as.so  libpaho-mqtt3c.so.1.3.8
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$
```

图 33.5.15 MQTT 客户端库文件

一共有 4 种类型的库，这里我们简单地介绍一下:

- **libpaho-mqtt3a.so:** 异步模式 MQTT 客户端库 (不支持 SSL)。
- **libpaho-mqtt3as.so:** 异步模式 MQTT 客户端库 (支持 SSL)。
- **libpaho-mqtt3c.so:** 同步模式 MQTT 客户端库 (不支持 SSL)。
- **libpaho-mqtt3cs.so:** 支持 SSL 的同步模式客户端库 (支持 SSL)。

Paho MQTT C 客户端库支持同步操作模式和异步操作模式两种, 关于它们之间的区别笔者不做介绍, 顶级目录下 docs/MQTTClient/html/async.html 文档(直接双击打开)中对此有相应的解释, 有兴趣的可以看一看; docs 目录下提供了很多供用户参考的文档, 包括 API 使用说明、示例代码等等, 在后续的学习过程中, 可以查看这些文档获取帮助。

MQTT 中使用 SSL/TLS 来提供安全性(由 openssl 提供), 使用 SSL 来做一些加密验证, 使得数据传输更加安全可靠。

以上便给大家简单地介绍了下这 4 种库文件之间的区别, 那后续我们将使用 libpaho-mqtt3c.so。

介绍完库文件之后, 再来看看头文件, 进入到 include 目录下:

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/include$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/include$ ls
MQTTAsync.h  MQTTClientPersistence.h  MQTTProperties.h  MQTTSubscribeOpts.h
MQTTClient.h  MQTTExportDeclarations.h  MQTTReasonCodes.h
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/include$
```

图 33.5.16 头文件

在我们的 MQTT 客户端应用程序中只需要包含 MQTTAsync.h 或 MQTTClient.h 头文件即可, 其它那些头文件会被这两个头文件所包含; MQTTAsync.h 是异步模式客户端库对外的头文件, 而 MQTTClient.h 则是同步模式客户端库对外的头文件。因为后续我们将使用同步模式, 所以到时在我们的应用程序中需要包含 MQTTClient.h 头文件。

拷贝库文件到开发板

将编译得到的库文件拷贝到开发板 Linux 系统/usr/lib 目录下, 注意不要破坏原有的链接关系, 建议在操作之前, 先将库文件进行打包, 如下所示:

```
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$ pwd
/home/dt/tools/paho.mqtt.c-1.3.8/install
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install$ cd lib/
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$ ls
cmake                libpaho-mqtt3as.so.1  libpaho-mqtt3cs.so
libpaho-mqtt3a.so    libpaho-mqtt3as.so.1.3.8  libpaho-mqtt3cs.so.1
libpaho-mqtt3a.so.1  libpaho-mqtt3c.so        libpaho-mqtt3cs.so.1.3.8
libpaho-mqtt3a.so.1.3.8  libpaho-mqtt3c.so.1
libpaho-mqtt3as.so    libpaho-mqtt3c.so.1.3.8
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$ rm -rfd cmake
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$ tar -czf libmqtt.tar.gz ./*
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$ ls
libmqtt.tar.gz ← libpaho-mqtt3as.so.1  libpaho-mqtt3cs.so
libpaho-mqtt3a.so    libpaho-mqtt3as.so.1.3.8  libpaho-mqtt3cs.so.1
libpaho-mqtt3a.so.1  libpaho-mqtt3c.so        libpaho-mqtt3cs.so.1.3.8
libpaho-mqtt3a.so.1.3.8  libpaho-mqtt3c.so.1
libpaho-mqtt3as.so    libpaho-mqtt3c.so.1.3.8
dt@dt-virtual-machine:~/tools/paho.mqtt.c-1.3.8/install/lib$
```

图 33.5.17 打包库文件

将压缩包文件 libmqtt.tar.gz 拷贝到开发板 Linux 系统/home/root 目录下, 然后将其解压到/usr/lib 目录:

```
tar -xzf libmqtt.tar.gz -C /usr/lib
```

```
root@ATK-IMX6U:~# ls
driver  libmqtt.tar.gz  shell
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# tar -xzf libmqtt.tar.gz -C /usr/lib
```

图 33.5.18 解压

33.6 MQTT 客户端库 API 介绍

上小节我们已经移植了 MQTT 客户端库到我们的开发板,接下来我们便可以开始编写 MQTT 客户端应用程序了,在编写应用程序之前,笔者需要向大家简单地介绍一下 MQTT 客户端库提供的 API。

本小节我们所介绍的这些库函数都是定义在 `MQTTClient.h` 头文件中,也就是同步模式客户端库 API,所以在我们的应用程序中需要包含头文件 `MQTTClient.h`。

Tips: MQTT 客户端源码顶级目录 `docs/MQTTClient/html/index.html` 文档向用户介绍了 API 的使用方法,并且提供了相应示例代码供用户参考。

Paho MQTT C Client Library

Main Page	Related Pages	Data Structures ▾	Files ▾
-----------	---------------	-------------------	---------

MQTT Client library for C

An MQTT client library in C. © Copyright IBM Corp. 2009, 2020 and others

These pages describe the original more synchronous API which might be considered easier to use. Some of the calls will block. For the new totally asynchronous API where no calls block, which is especially [Documentation](#). The MQTTClient API is not thread safe, whereas the MQTTAsync API is.

An MQTT client application connects to MQTT-capable servers. A typical client is responsible for collecting information from a telemetry device and publishing the information to the server. It can also subscribe to topics.

MQTT clients implement the published MQTT v3 protocol. You can write your own API to the MQTT protocol using the programming language and platform of your choice. This can be time-consuming and error-prone.

To simplify writing MQTT client applications, this library encapsulates the MQTT v3 protocol for you. Using this library enables a fully functional MQTT client application to be written in a few lines of code.

Using the client

Applications that use the client library typically use a similar structure:

- Create a client object
- Set the options to connect to an MQTT server
- Set up callback functions if multi-threaded (asynchronous mode) operation is being used (see [Asynchronous vs synchronous client applications](#)).
- Subscribe to any topics the client needs to receive
- Repeat until finished:
 - Publish any messages the client needs to
 - Handle any incoming messages
- Disconnect the client
- Free any memory being used by the client

Some simple examples are shown here:

- [Synchronous publication example](#)
- [Asynchronous publication example](#)
- [Asynchronous subscription example](#)

Additional information about important concepts is provided here:

- [Asynchronous vs synchronous client applications](#)
- [Callbacks](#)
- [Subscription wildcards](#)
- [Quality of service](#)
- [Tracing](#)

33.6.1 MQTTClient_message 结构体

这里向大家介绍一个结构体 `MQTTClient_message`,该结构体很重要, MQTT 客户端应用程序发布消息和接收消息都是围绕这个结构体。`MQTTClient_message` 数据结构描述了 MQTT 消息的负载和属性等相关

信息, 譬如消息的负载、负载的长度、qos、消息的保留标志、dup 标志等, 但是消息主题不是这个结构体的一部分。该结构体内容如下:

示例代码 33.6.1 MQTTClient_message 结构体

```
typedef struct
{
    int payloadlen;    //负载长度
    void* payload;    //负载
    int qos;          //消息的 qos 等级
    int retained;     //消息的保留标志
    int dup;          //dup 标志 (重复标志)
    int msgid;        //消息标识符, 也就是前面说的 packetId
    .....
} MQTTClient_message;
```

当客户端发布消息时就需要实例化一个 MQTTClient_message 对象, 同理, 当客户端接收到消息时, 其实也就是接收到了 MQTTClient_message 对象。通常在实例化 MQTTClient_message 对象时会使用 MQTTClient_message_initializer 宏对其进行初始化。

33.6.2 创建一个客户端对象

在连接服务端之前, 需要创建一个客户端对象, 使用 MQTTClient_create 函数创建:

```
int MQTTClient_create(MQTTClient *handle,
    const char *serverURL,
    const char *clientId,
    int persistence_type,
    void *persistence_context
);
```

handle: MQTT 客户端句柄;

serverURL: MQTT 服务器地址;

clientId: 客户端 ID;

persistence_type: 客户端使用的持久化类型:

- MQTTCLIENT_PERSISTENCE_NONE: 使用内存持久性。如果运行客户端的设备或系统出现故障或关闭, 则任何传输中消息的当前状态都会丢失, 并且即使在 QoS1 和 QoS2 下也可能无法传递某些消息。
- MQTTCLIENT_PERSISTENCE_DEFAULT: 使用默认的 (基于文件系统) 持久性机制。传输中消息的状态保存在文件系统中, 并在意外故障的情况下提供一些防止消息丢失的保护。
- MQTTCLIENT_PERSISTENCE_USER: 使用特定于应用程序的持久性实现。使用这种类型的持久性可以控制应用程序的持久性机制。应用程序必须实现 MQTTClient_persistence 接口。

persistence_context: 如果使用 MQTTCLIENT_PERSISTENCE_NONE 持久化类型, 则该参数应设置为 NULL。如果选择的是 MQTTCLIENT_PERSISTENCE_DEFAULT 持久化类型, 则该参数应设置为持久化目录的位置, 如果设置为 NULL, 则持久化目录就是客户端应用程序的工作目录。

返回值: 客户端对象创建成功返回 MQTTCLIENT_SUCCESS, 失败将返回一个错误码。

使用示例

```
MQTTClient client;
int rc;
```

```

/* 创建 mqtt 客户端对象 */
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_create(&client, "tcp://iot.ranye-iot.net:1883",
        "dt_mqtt_2_id",
        MQTTCLIENT_PERSISTENCE_NONE, NULL))) {
    printf("Failed to create client, return code %d\n", rc);
    return EXIT_FAILURE;
}

```

注意, "tcp://iot.ranye-iot.net:1883"地址中, 第一个冒号前面的 tcp 表示我们使用的是 TCP 连接; 后面的 1883 表示 MQTT 服务器对应的端口号。

33.6.3 连接服务端

客户端创建之后, 便可以连接服务器了, 调用 MQTTClient_connect 函数连接:

```

int MQTTClient_connect(MQTTClient handle,
    MQTTClient_connectOptions *options
);

```

handle: 客户端句柄;

options: 一个指针。指向一个 MQTTClient_connectOptions 结构体对象。MQTTClient_connectOptions 结构体中包含了 keepAlive、cleanSession 以及一个指向 MQTTClient_willOptions 结构体对象的指针 will_opts; MQTTClient_willOptions 结构体包含了客户端遗嘱相关的信息, 遗嘱主题、遗嘱内容、遗嘱消息的 QoS 等级、遗嘱消息的保留标志等。

返回值: 连接成功返回 MQTTCLIENT_SUCCESS, 是否返回错误码:

- 1: 连接被拒绝。不可接受的协议版本, 不支持客户端的 MQTT 协议版本
- 2: 连接被拒绝: 标识符被拒绝
- 3: 连接被拒绝: 服务器不可用
- 4: 连接被拒绝: 用户名或密码错误
- 5: 连接被拒绝: 未授权
- 6-255: 保留以备将来使用

示例代码 33.6.2 MQTTClient_connectOptions 和 MQTTClient_willOptions

```

typedef struct
{
    int keepAliveInterval;           //keepAlive
    int cleansession;               //cleanSession
    MQTTClient_willOptions *will;   //遗嘱相关
    const char *username;           //用户名
    const char *password;           //密码
    int reliable;                   //控制同步发布消息还是异步发布消息
    .....
    .....
} MQTTClient_connectOptions;

```

```
typedef struct
{
    const char *topicName;           //遗嘱主题
    const char *message;            //遗嘱内容
    int retained;                   //遗嘱消息的保留标志
    int qos;                         //遗嘱消息的 QoS 等级
    .....
    .....
} MQTTClient_willOptions;
```

使用示例:

```
MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
MQTTClient_willOptions will_opts = MQTTClient_willOptions_initializer;

.....

/* 连接服务器 */
will_opts.topicName = "dt2914/willTopic"; //遗嘱主题
will_opts.message = "Abnormally dropped"; //遗嘱内容
will_opts.retained = 1; //遗嘱保留消息
will_opts.qos = 0; //遗嘱 QoS 等级

conn_opts.will = &will_opts;
conn_opts.keepAliveInterval = 30; //客户端 keepAlive 间隔时间
conn_opts.cleansession = 0; //客户端 cleanSession 标志
conn_opts.username = "dt_mqtt_2"; //用户名
conn_opts.password = "dt291444"; //密码
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_connect(client, &conn_opts))) {
    printf("Failed to connect, return code %d\n", rc);
    return EXIT_FAILURE;
}
```

通常在定义 MQTTClient_connectOptions 对象时会使用 MQTTClient_connectOptions_initializer 宏对其进行初始化操作; 而在定义 MQTTClient_willOptions 对象时使用 MQTTClient_willOptions_initializer 宏对其进行初始化。

33.6.4 设置回调函数

调用 MQTTClient_setCallbacks 函数为应用程序设置回调函数, MQTTClient_setCallbacks 可设置多个回调函数, 包括: **断开连接时的回调函数 cl** (当客户端检测到自己掉线时会执行该函数, 如果将其设置为 NULL 表示应用程序不处理断线的情况)、**接收消息的回调函数 ma** (当客户端接收到服务端发送过来的消息时执行该函数, 必须设置此函数否则客户端无法接收消息)、**发布消息的回调函数 dc** (当客户端发布的消息已

经确认发送时执行该回调函数, 如果你的应用程序采用同步方式发布消息或者您不想检查是否成功发送时, 您可以将此设置为 NULL)。

```
int MQTTClient_setCallbacks(MQTTClient handle,
    void *context,
    MQTTClient_connectionLost *cl,
    MQTTClient_messageArrived *ma,
    MQTTClient_deliveryComplete *dc
);
```

handle: 客户端句柄;

context: 执行回调函数的时候, 会将 context 参数传递给回调函数, 因为每一个回调函数都设置了一个参数用来接收 context 参数。

cl: 一个 MQTTClient_connectionLost 类型的函数指针, 如下:

```
typedef void MQTTClient_connectionLost(void *context, char *cause);
```

参数 cause 表示断线的原因, 是一个字符串。

ma: 一个 MQTTClient_messageArrived 类型的函数指针, 如下:

```
typedef int MQTTClient_messageArrived(void *context, char *topicName,
    int topicLen, MQTTClient_message *message);
```

参数 topicName 表示消息的主题名, topicLen 表示主题名的长度; 参数 message 指向一个 MQTTClient_message 对象, 也就是客户端所接收到的消息。

dc: 一个 MQTTClient_deliveryComplete 类型的函数指针, 如下:

```
typedef void MQTTClient_deliveryComplete(void* context, MQTTClient_deliveryToken dt);
```

参数 dt 表示 MQTT 消息的值, 将其称为传递令牌。发布消息时 (应用程序通过 MQTTClient_publishMessage 函数发布消息), MQTT 协议会返回给客户端应用程序一个传递令牌; 应用程序可以通过将调用 MQTTClient_publishMessage() 返回的传递令牌与传递给此回调的令牌进行匹配来检查消息是否已成功发布。

前面提到了“同步发布消息”这个概念, 既然有同步发布, 那必然有异步发布, 确实如何! 那如何控制是同步发布还是异步发布呢? 就是通过 MQTTClient_connectOptions 对象中的 reliable 成员控制的, 这是一个布尔值, 当 reliable=1 时使用同步方式发布消息, 意味着必须完成当前正在发布的消息 (收到确认) 之后才能发布另一个消息; 如果 reliable=0 则使用异步方式发布消息。

当使用 MQTTClient_connectOptions_initializer 宏对 MQTTClient_connectOptions 对象进行初始化时, reliable 标志被初始化为 1, 所以默认是使用了同步方式。

返回值: 成功返回 MQTTCLIENT_SUCCESS, 失败返回 MQTTCLIENT_FAILURE。

注意: 调用 MQTTClient_setCallbacks 函数设置回调必须在连接服务器之前完成!

使用示例:

```
static void delivered(void *context, MQTTClient_deliveryToken dt)
{
    printf("Message with token value %d delivery confirmed\n", dt);
}

static int msgarrvd(void *context, char *topicName, int topicLen,
    MQTTClient_message *message)
{
    printf("Message arrived\n");
```

```
printf("topic: %s\n", topicName);
printf("message: <%d>%s\n", message->payloadlen, (char *)message->payload);
MQTTClient_freeMessage(&message); //释放内存
MQTTClient_free(topicName); //释放内存
return 1;
}

static void connlost(void *context, char *cause)
{
    printf("\nConnection lost\n");
    printf("    cause: %s\n", cause);
}

int main(void)
{
    .....

    /* 设置回调 */
    if (MQTTCLIENT_SUCCESS !=
        (rc = MQTTClient_setCallbacks(client, NULL, connlost,
            msgarrvd, delivered))) {
        printf("Failed to set callbacks, return code %d\n", rc);
        return EXIT_FAILURE;
    }

    .....
}
```

对于 msgarrvd 函数有两个点需要注意:

- 退出函数之前需要释放消息的内存空间, 必须调用 MQTTClient_freeMessage 函数; 同时也要释放主题名称占用的内存空间, 必须调用 MQTTClient_free。
- 函数的返回值。此函数的返回值必须是 0 或 1, 返回 1 表示消息已经成功处理; 返回 0 则表示消息处理存在问题, 在这种情况下, 客户端库将重新调用 MQTTClient_messageArrived() 以尝试再次将消息传递给客户端应用程序, 所以返回 0 时不要释放消息和主题所占用的内存空间, 否则重新投递失败。

33.6.5 发布消息

当客户端成功连接到服务端之后, 便可以发布消息或订阅主题了, 应用程序通过 MQTTClient_publishMessage 库函数来发布一个消息:

```
int MQTTClient_publishMessage(MQTTClient handle,
    const char *topicName,
    MQTTClient_message *msg,
    MQTTClient_deliveryToken *dt
);
```

handle: 客户端句柄;

topicName: 主题名称。向该主题发布消息。

msg: 指向一个 MQTTClient_message 对象的指针。

dt: 返回给应用程序的传递令牌。

返回值: 成功返回 MQTTCLIENT_SUCCESS, 失败返回错误码。

使用示例

```
MQTTClient_message pubmsg = MQTTClient_message_initializer;
MQTTClient_deliveryToken token;

.....

/* 发布消息 */
pubmsg.payload = "online";      //消息内容
pubmsg.payloadlen = 6;         //消息的长度
pubmsg.qos = 0;                //QoS 等级
pubmsg.retained = 1;          //消息的保留标志

if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_publishMessage(client, "dt2914/testTopic", &pubmsg, &token))) {
    printf("Failed to publish message, return code %d\n", rc);
    return EXIT_FAILURE;
}
```

33.6.6 订阅主题和取消订阅主题

客户端应用程序调用 MQTTClient_subscribe 函数来订阅主题:

```
int MQTTClient_subscribe(MQTTClient handle,
    const char *topic,
    int qos
);
```

handle: 客户端句柄;

topic: 主题名称。客户端订阅的主题。

qos: QoS 等级。

返回值: 成功返回 MQTTCLIENT_SUCCESS, 失败返回错误码。

使用示例

```
.....

if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_subscribe(client, "dt2914/testTopic", 0))) {
    printf("Failed to subscribe, return code %d\n", rc);
    return EXIT_FAILURE;
}
```

.....

当客户端想取消之前订阅的主题时, 可调用 MQTTClient_unsubscribe 函数, 如下所示:

```
int MQTTClient_unsubscribe(MQTTClient handle,  
    const char *topic  
);
```

handle: 客户端句柄;

topic: 主题名称。取消订阅该主题。

返回值: 成功返回 MQTTCLIENT_SUCCESS, 失败返回错误码。

33.6.7 断开服务端连接

当客户端需要主动断开与客户端连接时, 可调用 MQTTClient_disconnect 函数:

```
int MQTTClient_disconnect(MQTTClient handle,  
    int timeout  
);
```

handle: 客户端句柄;

timeout: 超时时间。客户端将断开连接延迟最多 timeout 时间 (以毫秒为单位), 以便完成正在进行中的消息传输。

返回值: 如果客户端成功从服务器断开连接, 则返回 MQTTCLIENT_SUCCESS; 如果客户端无法与服务器断开连接, 则返回错误代码。

33.7 编写客户端程序

上小节我们给大家介绍一些基本的 MQTT 客户端库函数, 除了这些基本 API 之外, MQTT 客户端库还提供了其它很多的 API, 本章就不给大家一一介绍了, 大家自己去看。那本小节我们将使用上小节中给大家介绍的几个 API 来编写一个自己的 MQTT 客户端应用程序, 然后使其在我们的开发板上运行, 实现自己的私人物联网项目。

我们这个小项目的功能设计如下:

- 基于然也物联平台提供的社区版 MQTT 服务器实现个人物联网小项目;
- 用户可通过手机或电脑远程控制开发板上的一颗 LED 灯;
- 开发板客户端每隔 30 秒向服务端发送 SoC 当前的温度值, 用户通过手机或电脑可查看到该温度值。

示例程序笔者已经给大家写好了, 由于功能比较简单, 所以代码比较短, 只有一个源文件; 虽然只有一个源文件, 为了学以致用, 我们将使用 cmake 来构建这个小项目。工程目录结构如下所示:

```
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$ ls  
build  cmake  CMakeLists.txt  mqttClient.c  
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$  
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$  
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$ tree  
.  
├── build  
├── cmake  
│   └── arm-linux-setup.cmake  
├── CMakeLists.txt  
└── mqttClient.c  
  
2 directories, 3 files  
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$
```

图 33.7.1 mqtt 小项目工程目录结构

mqtt_prj 工程对应的路径为: 开发板光盘->11、Linux C 应用编程例程源码->33_mqtt->mqtt_prj。

arm-linux-setup.cmake 文件

arm-linux-setup.cmake 源文件用于配置 cmake 交叉编译, 其内容如下所示:

```
#####
# 配置 ARM 交叉编译
#####
set(CMAKE_SYSTEM_NAME Linux)    #设置目标系统名字
set(CMAKE_SYSTEM_PROCESSOR arm) #设置目标处理器架构

# 指定编译器的 sysroot 路径
set(TOOLCHAIN_DIR /opt/fsl-ix11/4.1.15-2.1.0/sysroots)
set(CMAKE_SYSROOT ${TOOLCHAIN_DIR}/cortexa7hf-neon-poky-linux-gnueabi)

# 指定交叉编译器 arm-linux-gcc
set(CMAKE_C_COMPILER ${TOOLCHAIN_DIR}/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-
poky-linux-gnueabi-gcc)

# 为编译器添加编译选项
set(CMAKE_C_FLAGS "-march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7")

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
#####
# end
#####
```

这个就不多说了, 大家需要根据自己的交叉编译器的实际安装路径进行修改。

CMakeLists.txt 文件

文件内容如下所示:

```
#####
# Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
#
# 顶层 CMakeLists.txt
# All rights reserved. This program and the accompanying materials
# are made available under the terms of the Eclipse Public License v2.0
# and Eclipse Distribution License v1.0 which accompany this distribution.
#####/
cmake_minimum_required(VERSION 2.8.12)
project(MQTTClient C)
message(STATUS "CMake version: " ${CMAKE_VERSION})
message(STATUS "CMake system name: " ${CMAKE_SYSTEM_NAME})
```



```
message(STATUS "CMake system processor: " ${CMAKE_SYSTEM_PROCESSOR})
```

```
# 设置可执行文件输出路径
```

```
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
```

```
# 定义可执行文件目标
```

```
add_executable(mqttClient mqttClient.c)
```

```
# 指定 MQTT 客户端库头文件路径、库路径以及链接库
```

```
# ***大家需要根据 MQTT 的实际安装路径设置***
```

```
target_include_directories(mqttClient PRIVATE /home/dt/tools/paho.mqtt.c-1.3.8/install/include)#MQTT 头文件搜索路径
```

```
target_link_directories(mqttClient PRIVATE /home/dt/tools/paho.mqtt.c-1.3.8/install/lib) #MQTT 库文件搜索路径
```

```
target_link_libraries(mqttClient PRIVATE paho-mqtt3c) #MQTT 链接库 libpaho-mqtt3c.so
```

客户端应用程序源文件 mqttClient.c

接下来我们看看客户端源码 mqttClient.c 的内容，如下所示：

示例代码 33.7.1 mqttClient.c 源文件内容

```
/******
```

```
Copyright © ALIENTEK Co., Ltd. 1998-2021. All rights reserved.
```

```
文件名 : mqttClient.c
```

```
作者 : 邓涛
```

```
版本 : V1.0
```

```
描述 : 开发板上的 MQTT 客户端应用程序示例代码
```

```
其他 : 无
```

```
论坛 : www.openedv.com
```

```
日志 : 初版 V1.0 2021/7/20 邓涛创建
```

```
*****/
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
#include "MQTTClient.h" //包含 MQTT 客户端库头文件
```

```
/* #####宏定义##### */
```

```
#define BROKER_ADDRESS "tcp://iot.ranye-iot.net:1883" //然也物联平台社区版 MQTT 服务器地址
```

```
/* 客户端 id、用户名、密码 */
```

- * 当您成功申请到然也物联平台的社区版 MQTT 服务后
- * 然也物联工作人员会给你发送 8 组用于连接社区版 MQTT 服务器
- * 的客户端连接认证信息: 也就是客户端 id、用户名和密码
- * 注意一共有 8 组, 您选择其中一组覆盖下面的示例值
- * 后续我们使用 MQTT.fx 或 MQTTTool 的时候 也需要使用一组连接认证信息
- * 去连接社区版 MQTT 服务器!
- * 由于这是属于个人隐私 笔者不可能将自己的信息写到下面 */

```
#define CLIENTID      "您的客户端 ID"      //客户端 id
#define USERNAME     "您的用户名"         //用户名
#define PASSWORD     "您的密码"           //密码

/* 然也物联社区版 MQTT 服务为每个申请成功的用户
 * 提供了个人专属主题级别, 在官方发给您的微信信息中
 * 提到了
 * 以下 dt_mqtt/ 便是笔者的个人主题级别
 * dt_mqtt 其实就是笔者申请社区版 MQTT 服务时注册的用户名
 * 大家也是一样, 所以你们需要替换下面的 dt_mqtt 前缀
 * 换成你们的个人专属主题级别 (也就是您申请时的用户名)
 */
#define WILL_TOPIC   "dt_mqtt/will"       //遗嘱主题
#define LED_TOPIC    "dt_mqtt/led"        //LED 主题
#define TEMP_TOPIC   "dt_mqtt/temperature" //温度主题
/* ##### */

static int msgarrvd(void *context, char *topicName, int topicLen,
                   MQTTClient_message *message)
{
    if (!strcmp(topicName, LED_TOPIC)) { //校验消息的主题
        if (!strcmp("2", message->payload)) //如果接收到的消息是"2"则设置 LED 为呼吸灯模式
            system("echo heartbeat > /sys/class/leds/sys-led/trigger");
        if (!strcmp("1", message->payload)) { //如果是"1"则 LED 常量
            system("echo none > /sys/class/leds/sys-led/trigger");
            system("echo 1 > /sys/class/leds/sys-led/brightness");
        }
        else if (!strcmp("0", message->payload)) { //如果是"0"则 LED 熄灭
            system("echo none > /sys/class/leds/sys-led/trigger");
            system("echo 0 > /sys/class/leds/sys-led/brightness");
        }

        // 接收到其它数据 不做处理
    }

    /* 释放占用的内存空间 */
}
```

```
MQTTClient_freeMessage(&message);
MQTTClient_free(topicName);

/* 退出 */
return 1;
}

static void connlost(void *context, char *cause)
{
    printf("\nConnection lost\n");
    printf("    cause: %s\n", cause);
}

int main(int argc, char *argv[])
{
    MQTTClient client;
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    MQTTClient_willOptions will_opts = MQTTClient_willOptions_initializer;
    MQTTClient_message pubmsg = MQTTClient_message_initializer;
    int rc;

    /* 创建 mqtt 客户端对象 */
    if (MQTTCLIENT_SUCCESS !=
        (rc = MQTTClient_create(&client, BROKER_ADDRESS, CLIENTID,
            MQTTCLIENT_PERSISTENCE_NONE, NULL))) {
        printf("Failed to create client, return code %d\n", rc);
        rc = EXIT_FAILURE;
        goto exit;
    }

    /* 设置回调 */
    if (MQTTCLIENT_SUCCESS !=
        (rc = MQTTClient_setCallbacks(client, NULL, connlost,
            msgarrvd, NULL))) {
        printf("Failed to set callbacks, return code %d\n", rc);
        rc = EXIT_FAILURE;
        goto destroy_exit;
    }

    /* 连接 MQTT 服务器 */
    will_opts.topicName = WILL_TOPIC;           //遗嘱主题
    will_opts.message = "Unexpected disconnection"; //遗嘱消息
    will_opts.retained = 1;                     //保留消息
}
```

```

will_opts.qos = 0; //QoS

conn_opts.will = &will_opts;
conn_opts.keepAliveInterval = 30; //心跳包间隔时间
conn_opts.cleansession = 0; //cleanSession 标志
conn_opts.username = USERNAME; //用户名
conn_opts.password = PASSWORD; //密码
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_connect(client, &conn_opts))) {
    printf("Failed to connect, return code %d\n", rc);
    rc = EXIT_FAILURE;
    goto destroy_exit;
}

printf("MQTT 服务器连接成功!\n");

/* 发布上线消息 */
pubmsg.payload = "Online"; //消息的内容
pubmsg.payloadlen = 6; //内容的长度
pubmsg.qos = 0; //QoS 等级
pubmsg.retained = 1; //保留消息
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_publishMessage(client, WILL_TOPIC, &pubmsg, NULL))) {
    printf("Failed to publish message, return code %d\n", rc);
    rc = EXIT_FAILURE;
    goto disconnect_exit;
}

/* 订阅主题 dt_mqtt/led */
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_subscribe(client, LED_TOPIC, 0))) {
    printf("Failed to subscribe, return code %d\n", rc);
    rc = EXIT_FAILURE;
    goto disconnect_exit;
}

/* 向服务端发布芯片温度信息 */
for (;) {

    MQTTClient_message tempmsg = MQTTClient_message_initializer;
    char temp_str[10] = {0};
    int fd;

```

```

/* 读取温度值 */
fd = open("/sys/class/thermal/thermal_zone0/temp", O_RDONLY);
read(fd, temp_str, sizeof(temp_str)); //读取 temp 属性文件即可获取温度
close(fd);

/* 发布温度信息 */
tempmsg.payload = temp_str;           //消息的内容
tempmsg.payloadlen = strlen(temp_str); //内容的长度
tempmsg.qos = 0;                      //QoS 等级
tempmsg.retained = 1;                 //保留消息
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_publishMessage(client, TEMP_TOPIC, &tempmsg, NULL))) {
    printf("Failed to publish message, return code %d\n", rc);
    rc = EXIT_FAILURE;
    goto unsubscribe_exit;
}

sleep(30); //每隔 30 秒 更新一次数据
}

unsubscribe_exit:
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_unsubscribe(client, LED_TOPIC))) {
    printf("Failed to unsubscribe, return code %d\n", rc);
    rc = EXIT_FAILURE;
}

disconnect_exit:
if (MQTTCLIENT_SUCCESS !=
    (rc = MQTTClient_disconnect(client, 10000))) {
    printf("Failed to disconnect, return code %d\n", rc);
    rc = EXIT_FAILURE;
}

destroy_exit:
MQTTClient_destroy(&client);
exit:
return rc;
}

```

代码笔者就不再进行了介绍，因为该讲的东西前面都已经讲得很清楚了，没有必要再去啰嗦了！

代码中的三个主题需要向大家简单地说明一下：

WILL_TOPIC: 这是客户端的遗嘱主题。

LED_TOPIC: LED 主题，我们的开发板客户端订阅了该主题，而我们会通过其它客户端，譬如手机或电脑去向这个主题发布信息，那么接收到信息之后根据信息的内容，来对 LED 做出相应的控制，譬如点亮 LED、熄灭 LED。

TEMP_TOPIC: 温度主题, 我们的开发板客户端会向这个主题发布消息, 这个消息的内容就是开发板这个芯片温度值, 开发板的温度值怎么获取? 对于我们 MX6U 开发板来说, 就是读取 `/sys/class/thermal/thermal_zone0/temp` 属性文件。同样, 其它客户端(譬如手机或电脑)会订阅这个温度主题, 所以, 手机或电脑就会收到开发板的温度信息。程序中是设置没个 30 秒发一次。

所以, 大家一想, 其实这东西真的非常简单, 人家 MQTT 协议都给你做好了, 你就只管用就是了!

构建、编译

我们直接进行编译, 进入到工程目录下的 build 目录中, 执行 cmake 构建:

```
~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/arm-linux-setup.cmake -DCMAKE_BUILD_TYPE=Release ..
```

```
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$ ls
build CMakeLists.txt mqttClient.c
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj$ cd build/
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$ ~/tools/cmake-3.16.0-Linux-x86_64/bin/cmake -DCMAKE_TOOLCHAIN_FILE=../cmake/arm-linux-setup.cmake -DCMAKE_BUILD_TYPE=Release ..
-- The C compiler identification is GNU 5.3.0
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc
-- Check for working C compiler: /opt/fsl-imx-x11/4.1.15-2.1.0/sysroots/x86_64-pokysdk-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- CMake version: 3.16.0
-- CMake system name: Linux
-- CMake system processor: arm
-- Configuring done
-- Generating done
-- Build files have been written to: /home/dt/vscode_ws/mqtt_prj/build
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$ ls
bin CMakeCache.txt CMakeFiles cmake_install.cmake Makefile
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$
```

图 33.7.2 执行 cmake 构建

执行 make 编译:

```
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$ ls
bin CMakeCache.txt CMakeFiles cmake_install.cmake Makefile
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$ make
Scanning dependencies of target mqttClient
[ 50%] Building C object CMakeFiles/mqttClient.dir/mqttClient.c.o
[100%] Linking C executable bin/mqttClient
[100%] Built target mqttClient
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$ ls
bin CMakeCache.txt CMakeFiles cmake_install.cmake Makefile
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build$ cd bin/
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build/bin$
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build/bin$ ls
mqttClient
dt@dt-virtual-machine:~/vscode_ws/mqtt_prj/build/bin$
```

图 33.7.3

编译成功之后, 在 build/bin 目录下生成了可执行文件 mqttClient。

将可执行文件 mqttClient 拷贝到开发板 Linux 系统/home/root 目录下。

33.8 演示

在进行测试之前, 确保开发板是能够上网的, 也就是能够连接外网, 注意不是局域网, 知道吧! 执行 mqttClient 客户端应用程序, 如下所示:

```
root@ATK-IMX6U:~# ls
driver mqttClient shell
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./mqttClient
MQTT服务器连接成功!
```

图 33.8.1 在开发板上执行 mqttClient 客户端程序

这样, 我们的开发板作为 MQTT 客户端就成功连接上了 MQTT 服务器, 并且每隔 30 秒向服务器发布温度信息, 同样也会接收 LED 主题的信息。

现在我们使用 MQTT.fx 在电脑上进行测试, 打开 MQTT.fx 客户端软件, 使用然也物联工作人员发给你的 8 组客户端连接认证信息中的其中一组 (不要使用开发板客户端已经使用的那组) 去连接然也物联社区版 MQTT 服务器:

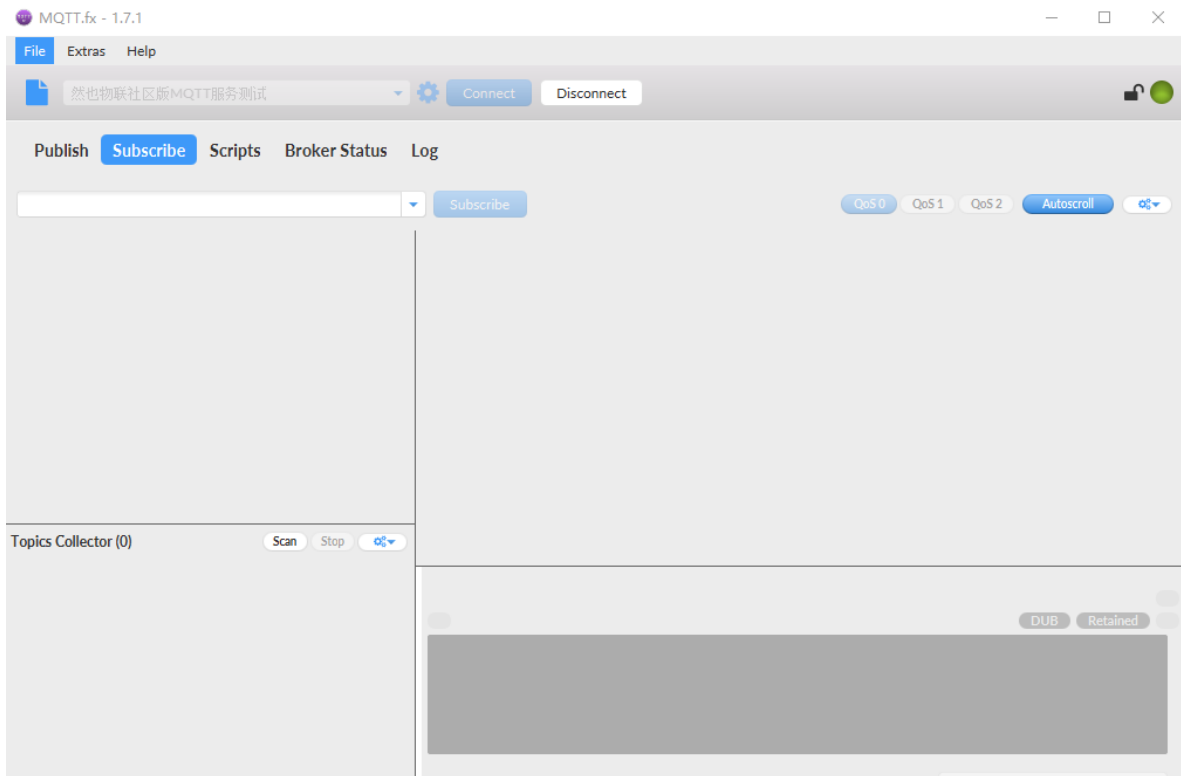


图 33.8.2 MQTT.fx 客户端连接然也物联社区版 MQTT 服务器

接下来我们订阅温度主题"dt_mqtt/temperature", 你需要修改成你的个人专属主题级别, 也就是将 dt_mqtt 换成你的个人专属主题级别。订阅之后立马就会收到温度信息, 并且之后每隔 30 秒会收到开发板客户端发布的温度信息, 如下图所示:

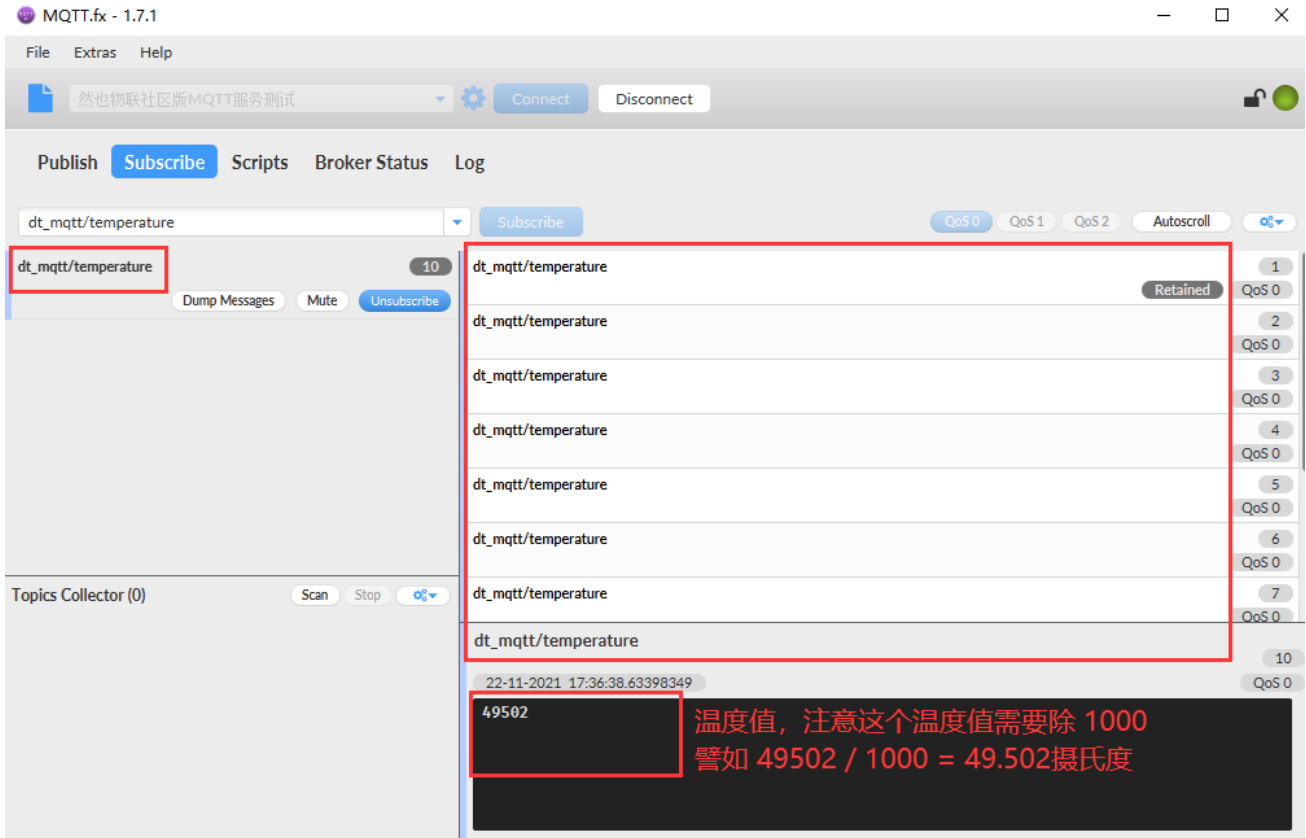


图 33.8.3 订阅温度主题、获取到开发板温度信息

接着我们向 LED 主题“dt_mqtt/led”（同样你也需要修改成你的个人专属主题级别，也就是将 dt_mqtt 换成你的个人专属主题级别）发布消息去控制开发板上的 LED:

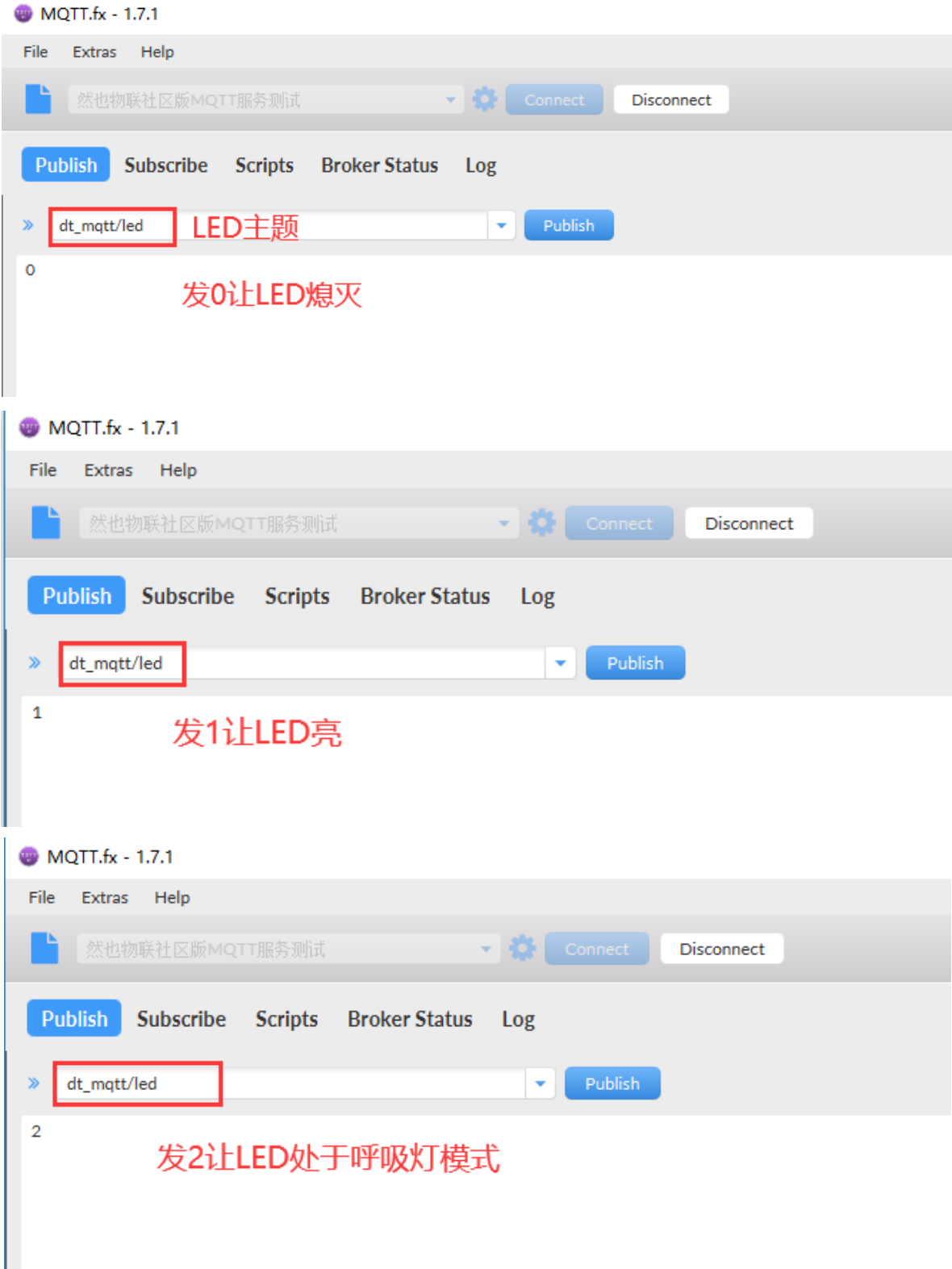


图 33.8.4 向 LED 主题发布消息控制开发板上的 LED

我这里没法给你演示, 你自己看效果, 有没有成功控制板上的 LED 灯, 反正笔者这里是没有问题的。

除了使用电脑之外, 我们还可以使用手机控制或查看开发板芯片温度信息, 在手机上使用 MQTTool 软件连接然也物联社区版 MQTT 服务器 (同样也是使用 8 组连接认证信息中的其中一组, 不要使用开发板和

MQTT.fx 正在使用的这组信息), 连接成功之后订阅温度主题查看开发板温度信息、向 LED 主题发布信息控制开发板上的 LED 灯, 如下所示:

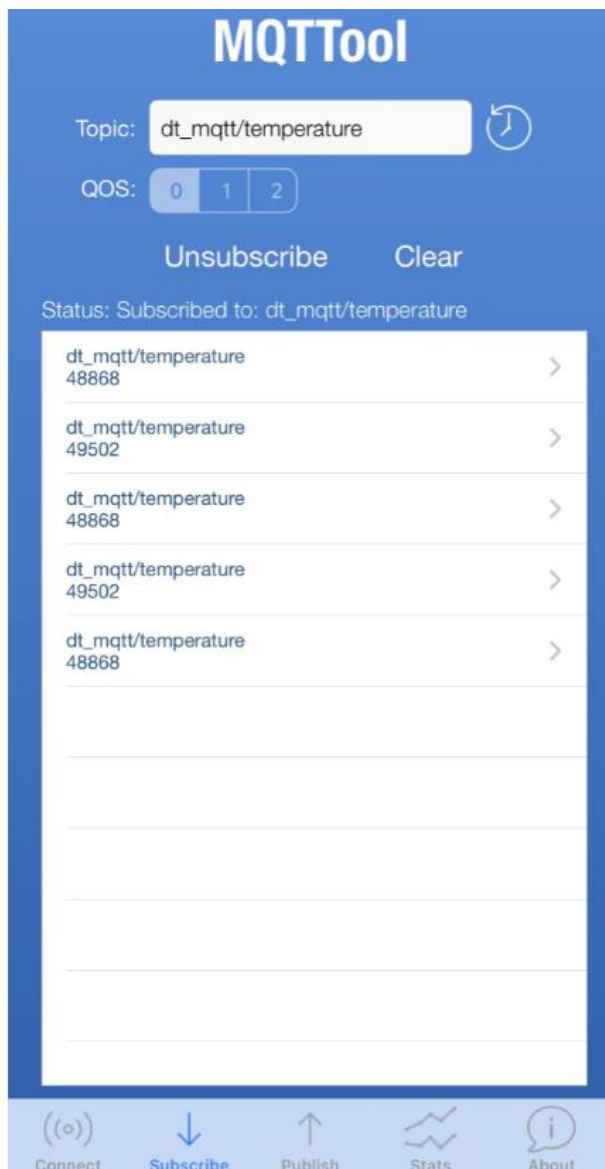


图 33.8.5 订阅温度主题



图 33.8.6 控制 LED 灯

本章我们的内容就结束了!

第三十四章 实战小项目之视频监控

目前常见的视频监控和视频直播都是使用了 RTMP 和 RTSP 流媒体传输协议等。

RTSP (Real-Time Stream Protocol) 由 Real Networks 和 Netscape 共同提出的, 基于文本的多媒体播放控制协议。RTSP 定义流格式, 流数据经由 RTP 传输; RTSP 实时效果非常好, 适合视频聊天, 视频监控等方向。

RTMP (Real Time Message Protocol) 由 Adobe 公司提出, 用来解决多媒体数据传输流的多路复用 (Multiplexing) 和分包 (packetizing) 的问题, 优势在于低延迟, 稳定性高, 支持所有摄像头格式, 浏览器加载 flash 插件就可以直接播放。

RTSP 和 RTMP 的区别:

RTSP 虽然实时性最好, 但是实现复杂, 适合视频聊天和视频监控; RTMP 强在浏览器支持好, 加载 flash 插件后就能直接播放, 所以非常火, 相反在浏览器里播放 rtsp 就很困难了。

本章我们将向大家介绍如何通过 FFmpeg+Nginx、使用 RTMP 推流实现视频监控或直播。

本章将会讨论如下主题内容。

- 视频监控简介
- Nginx 简介
- 如何移植 Nginx
- FFmpeg+Nginx 实现 RTMP 推流
- 拉流

34.1 视频监控简介

本章我们将使用 RTMP 流媒体服务来实现视频监控, RTMP 流媒体服务框架图如下所示:

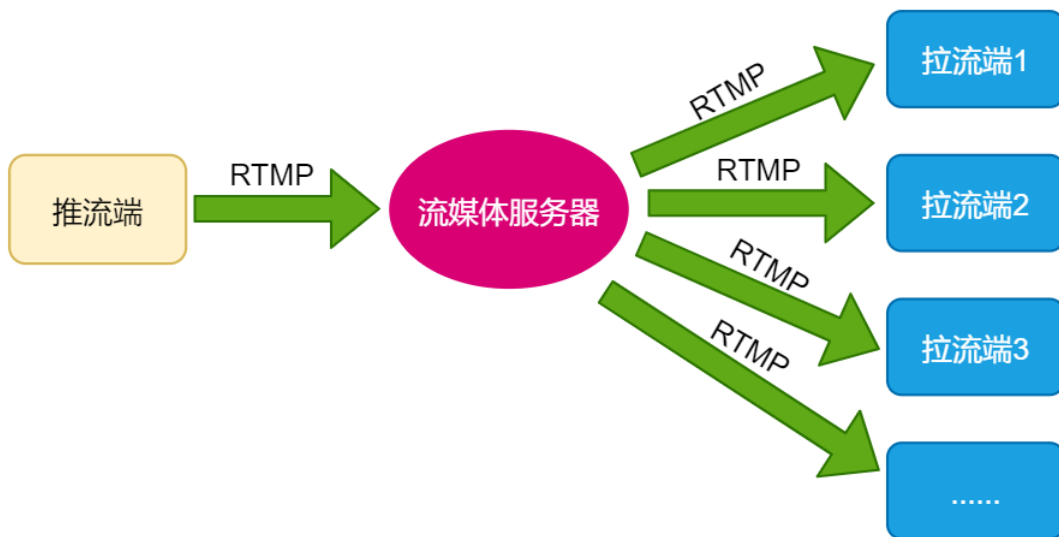


图 34.1.1 流媒体服务

推流端负责将视频数据通过 RTMP 流媒体协议传输给 RTMP 流媒体服务器, 拉流端可以从流媒体服务器中通过 RTMP 协议获取到视频数据; 而流媒体服务器负责接收推流端的视频数据、当有客户端(拉流端)想要获取视频数据时再将其发给相应的客户端。

所以从上图可知, 要想实现 RTMP 视频监控, 必须要有这三部分: 推流客户端、拉流客户端以及流媒体服务器。那这些需要我们去实现吗? 当然不需要, 譬如推流我们可以使用 FFmpeg 来做, 拉流则可以实现 VLC 播放器来做, 而流媒体客户端则使用 Nginx 来搭建即可!

34.2 Nginx 移植

前面也给大家提到了, 我们可以使用 Nginx 来搭建 RTMP 流媒体服务器, 譬如你可以在一台公网 IP 主机上搭建流媒体服务器, 当然, 笔者并没有这个条件; 这里我们选择在开发板上搭建流媒体服务器, 并且推流端也是开发板, 所以在本章的方案中, 开发板既是流媒体服务器、也是推断端。

既然要在开发板上搭建流媒体服务器, 首先我们需要将 Nginx 移植到开发板上, 事实上, 我们的板子出厂系统中就已经移植好了 Nginx, 并且板子在启动进入系统时会自动启动 Nginx, 也就是启动流媒体服务, 所以板子启动之后本身就已经是一台流媒体服务器了。当然, 这里我们不管出厂系统中已经搭建好的流媒体服务, 这里只是给大家说明一下, 本章我们要自己动手亲自移植 Nginx、然后在板子上搭建流媒体服务。

34.2.1 下载 Nginx 源码

进入到 Ubuntu 系统的某个目录下, 执行下面这条命令下载 Nginx 源码:

```
wget http://nginx.org/download/nginx-1.20.0.tar.gz
```

```
dt@dt-virtual-machine:~/tools$ pwd
/home/dt/tools
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 paho.mqtt.c-1.3.8 png tslib zlib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ wget http://nginx.org/download/nginx-1.20.0.tar.gz
--2021-12-10 15:08:24-- http://nginx.org/download/nginx-1.20.0.tar.gz
正在解析主机 nginx.org (nginx.org)... 52.58.199.22, 3.125.197.172, 2a05:d014:edb:5702::6, ...
正在连接 nginx.org (nginx.org)[52.58.199.22]:80... 已连接。
已发出 HTTP 请求, 正在等待回应... 200 OK
长度: 1061070 (1.0M) [application/octet-stream]
正在保存至: "nginx-1.20.0.tar.gz"

nginx-1.20.0.tar.gz          100%[=====]
2021-12-10 15:08:25 (741 KB/s) - 已保存 "nginx-1.20.0.tar.gz" [1061070/1061070]
dt@dt-virtual-machine:~/tools$
```

图 34.2.1 下载 Nginx 源码

这里我们下载的是 1.20 版本, 这是比较新的版本了。下载完成之后将得到一个名为 `nginx-1.20.0.tar.gz` 的压缩包文件。

```
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 nginx-1.20.0.tar.gz paho.mqtt.c-1.3.8 png tslib zlib
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls -lh nginx-1.20.0.tar.gz
-rw-rw-r-- 1 dt dt 1.1M 4月 20 2021 nginx-1.20.0.tar.gz
dt@dt-virtual-machine:~/tools$
```

图 34.2.2 nginx-1.20.0.tar.gz 压缩包文件

这就是 Nginx 的源码包。

34.2.2 下载 nginx-rtmp-module 模块

事实上, 原生的 Nginx 并不支持 RTMP, 我们需要安装第三方模块 `nginx-rtmp-module` 插件使其支持 RTMP。通过下面这条命令下载 `nginx-rtmp-module`。

```
git clone https://github.com/arut/nginx-rtmp-module.git
```

```
dt@dt-virtual-machine:~/tools$ git clone https://github.com/arut/nginx-rtmp-module.git
正克隆到 'nginx-rtmp-module'...
remote: Enumerating objects: 4324, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 4324 (delta 1), reused 3 (delta 0), pack-reused 4317
接收对象中: 100% (4324/4324), 3.11 MiB | 958.00 KiB/s, 完成。
处理 delta 中: 100% (2689/2689), 完成。
检查连接... 完成。
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 nginx-1.20.0.tar.gz nginx-rtmp-module
dt@dt-virtual-machine:~/tools$
```

图 34.2.3 下载 nginx-rtmp-module

下载成功之后将得到 `nginx-rtmp-module` 文件夹。

34.2.3 交叉编译 Nginx

将下载得到的 `nginx-1.20.0.tar.gz` 文件进行解压:

```
tar -xzf nginx-1.20.0.tar.gz
```

```
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 nginx-1.20.0.tar.gz nginx-rtmp-module paho.mqtt.c-1
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ tar -xzf nginx-1.20.0.tar.gz
dt@dt-virtual-machine:~/tools$
dt@dt-virtual-machine:~/tools$ ls
cmake-3.16.0-Linux-x86_64 freetype jpeg libsocketcan-0.0.12 nginx-1.20.0 nginx-1.20.0.tar.gz nginx-rtmp-module
dt@dt-virtual-machine:~/tools$
```

图 34.2.4 解压

解压之后生成 nginx-1.20 文件夹, 进入到该目录下。在进行交叉编译之前, 先对交叉编译工具进行初始化操作:

```
source /opt/fsl-imx-x11/4.1.15-2.1.0/environment-setup-cortexa7hf-neon-poky-linux-gnueabi
```

大家根据自己的实际安装位置填写路径。

首先配置源码、接下执行 make 编译源码, 最后执行 make install 安装即可! 总共的步骤就这三个, 但事实上在编译的过程中会出现一些问题, 我们稍后再看!

配置源码

首先第一步是配置源码, 在配置之前, 需要进行一个简单的修改, 否则配置通不过; 首先打开 nginx 源码目录下的 auto/cc/name 文件, 将 21 行处的 "exit 1" 给注释掉! 如下所示:

```
2 # Copyright (C) Igor Sysoev
3 # Copyright (C) Nginx, Inc.
4
5
6 if [ "$NGX_PLATFORM" != win32 ]; then
7
8     ngx_feature="C compiler"
9     ngx_feature_name=
10    ngx_feature_run=yes
11    ngx_feature_incs=
12    ngx_feature_path=
13    ngx_feature_libs=
14    ngx_feature_test=
15    . auto/feature
16
17    if [ $ngx_found = no ]; then
18        echo
19        echo $0: error: C compiler $CC is not found
20        echo
21        # exit 1 前面加#号注释该行
22    fi
23
24 fi
```

图 34.2.5 修改 name 文件

修改完成之后保存退出。接着打开 auto/types/sizeof 文件, 将 15 行处的 "ngx_size=" 修改为 "ngx_size=4", 并且将 36 行处的 "\$CC" 修改为 "gcc", 如下所示:

```
14 ngx_size=4 修改为 ngx_size=4
15
16
17 cat << END > $NGX_AUTOTEST.c
18
19 #include <sys/types.h>
20 #include <sys/time.h>
21 $NGX_INCLUDE_UNISTD_H
22 #include <signal.h>
23 #include <stdio.h>
24 #include <sys/resource.h>
25 $NGX_INCLUDE_INTTYPES_H
26 $NGX_INCLUDE_AUTO_CONFIG_H
27
28 int main(void) {
29     printf("%d", (int) sizeof($ngx_type));
30     return 0;
31 }
32
33 END
34
35
36 ngx_test="gcc $CC TEST_FLAGS $CC_AUX_FLAGS \
37     -o $NGX_AUTOTEST $NGX_AUTOTEST.c $NGX_LD_OPT $ngx_feature_libs"
38
39 eval "$ngx_test >> $NGX_AUTOCONF_ERR 2>&1"
40
41
42 if [ -x $NGX_AUTOTEST ]; then
43     # ngx_size=`$NGX_AUTOTEST`
44     ngx_size=4 将 ngx_size=`$NGX_AUTOTEST` 修改为 ngx_size=4
45     echo " $ngx_size bytes"
46 fi
47
```

图 34.2.6 修改 sizeof 文件

同样, 修改完成之后保存退出即可! 接着执行下面这条命令进行配置:

```
./configure --prefix=/home/dt/tools/nginx-1.20.0/install \
--with-http_ssl_module \
--with-http_mp4_module \
--with-http_v2_module \
--without-http_upstream_zone_module \
--add-module=/home/dt/tools/nginx-rtmp-module
```

上述命令中, `--prefix` 指定了 nginx 的安装路径, 笔者为了方便直接将其安装到 nginx 源码目录下的 `install` 目录中; `--add-module` 用于添加第三方模块, 譬如我们前面下载的 `nginx-rtmp-module`, 所以 `--add-module` 需要指向 `nginx-rtmp-module` 源码路径, 大家根据自己的实际路径填写。

如下所示:


```
dt@dt-virtual-machine:~/tools/nginx-1.20.0$ ./configure --prefix=/home/dt/tools/nginx-1.20.0/install \  
> --with-http_ssl_module \  
> --with-http_mp4_module \  
> --with-http_v2_module \  
> --without-http_upstream_zone_module \  
> --add-module=/home/dt/tools/nginx-rtmp-module  
checking for OS  
+ Linux 4.15.0-142-generic x86_64  
checking for C compiler ... found but is not working  
  
./configure: error: C compiler arm-poky-linux-gnueabi-gcc -march=armv7ve -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7  
oky-linux-gnueabi is not found  
  
+ using GNU C compiler  
+ gcc version: 5.3.0 (GCC)  
checking for gcc -pipe switch ... found  
checking for -Wl,-E switch ... found  
checking for gcc builtin atomic operations ... found but is not working  
checking for C99 variadic macros ... found but is not working  
checking for gcc variadic macros ... found but is not working  
checking for gcc builtin 64 bit byteswap ... found  
checking for unistd.h ... found  
checking for inttypes.h ... found  
checking for limits.h ... found  
checking for sys/filio.h ... not found  
checking for sys/param.h ... found  
checking for sys/mount.h ... found  
checking for sys/statvfs.h ... found  
checking for crypt.h ... found  
checking for Linux specific features
```

图 34.2.7 nginx 配置

配置成功打印信息如下所示:

```
adding module in /home/dt/tools/nginx-rtmp-module  
+ ngx_rtmp_module was configured  
checking for PCRE library ... found  
checking for PCRE JIT support ... found  
checking for OpenSSL library ... found  
checking for zlib library ... found  
creating objs/Makefile  
  
Configuration summary  
+ using system PCRE library  
+ using system OpenSSL library  
+ using system zlib library  
  
nginx path prefix: "/home/dt/tools/nginx-1.20.0/install"  
nginx binary file: "/home/dt/tools/nginx-1.20.0/install/sbin/nginx"  
nginx modules path: "/home/dt/tools/nginx-1.20.0/install/modules"  
nginx configuration prefix: "/home/dt/tools/nginx-1.20.0/install/conf"  
nginx configuration file: "/home/dt/tools/nginx-1.20.0/install/conf/nginx.conf"  
nginx pid file: "/home/dt/tools/nginx-1.20.0/install/logs/nginx.pid"  
nginx error log file: "/home/dt/tools/nginx-1.20.0/install/logs/error.log"  
nginx http access log file: "/home/dt/tools/nginx-1.20.0/install/logs/access.log"  
nginx http client request body temporary files: "client_body_temp"  
nginx http proxy temporary files: "proxy_temp"  
nginx http fastcgi temporary files: "fastcgi_temp"  
nginx http uwsgi temporary files: "uwsgi_temp"  
nginx http scgi temporary files: "scgi_temp"  
  
dt@dt-virtual-machine:~/tools/nginx-1.20.0$
```

图 34.2.8 配置完成

编译源码

配置完成之后,接着我们执行 make 编译:

```
dt@dt-virtual-machine:~/tools/nginx-1.20.0$ make -j12
make -f objs/Makefile
make[1]: Entering directory '/home/dt/tools/nginx-1.20.0'
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx.o \
src/core/nginx.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_log.o \
src/core/nginx_log.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_palloc.o \
src/core/nginx_palloc.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_array.o \
src/core/nginx_array.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_list.o \
src/core/nginx_list.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_hash.o \
src/core/nginx_hash.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_buf.o \
src/core/nginx_buf.c
arm-poky-linux-gnueabi-gcc -march=armv7ve -mcpu=cortex-a7 --sysroot=/opt/fsl-imx-0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Werror -g -I src/core -I src/event -I src/event/modules -I s
-o objs/src/core/nginx_queue.o \
```

图 34.2.9 执行 make 编译

本次编译并不会成功，将会出现如下错误打印信息：

```
objs/addon/dash/nginx_rtmp_mp4.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_stat_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_control_module.o \
objs/nginx_modules.o \
-ldl -lcrypt -lpcrc -lssl -lcrypto -ldl -lz \
-Wl,-E
objs/src/core/nginx_cycle.o: In function `ngx_init_cycle':
/home/dt/tools/nginx-1.20.0/src/core/nginx_cycle.c:485: undefined reference to `ngx_shm_alloc'
/home/dt/tools/nginx-1.20.0/src/core/nginx_cycle.c:695: undefined reference to `ngx_shm_free'
/home/dt/tools/nginx-1.20.0/src/core/nginx_cycle.c:912: undefined reference to `ngx_shm_free'
objs/src/event/nginx_event.o: In function `ngx_event_module_init':
/home/dt/tools/nginx-1.20.0/src/event/nginx_event.c:549: undefined reference to `ngx_shm_alloc'
collect2: error: ld returned 1 exit status
objs/Makefile:284: recipe for target 'objs/nginx' failed
make[1]: *** [objs/nginx] Error 1
make[1]: Leaving directory '/home/dt/tools/nginx-1.20.0'
Makefile:10: recipe for target 'build' failed
make: *** [build] Error 2
dt@dt-virtual-machine:~/tools/nginx-1.20.0$
```

图 34.2.10 编译报错

这个时候我们需要修改 nginx 源码目录下的 objs/nginx_auto_config.h 文件，将如下内容添加到该头文件中：

```
#ifndef NGX_HAVE_SYSVSHM
#define NGX_HAVE_SYSVSHM 1
#endif
```

```

1 #define NGX_CONFIGURE " --prefix=/home/dt/tools/nginx-1.20.0/install --with-http_ssl_module
  module=/home/dt/tools/nginx-rtmp-module"
2
3 #ifndef NGX_HAVE_SYSVSHM
4 #define NGX_HAVE_SYSVSHM 1 添加宏定义
5 #endif
6
7 #ifndef NGX_COMPILER
8 #define NGX_COMPILER "gcc 5.3.0 (GCC) "
9 #endif
10
11
12 #ifndef NGX_HAVE_GCC_BSWAP64
13 #define NGX_HAVE_GCC_BSWAP64 1
14 #endif
15
16
17 #ifndef NGX_HAVE_O_PATH
18 #define NGX_HAVE_O_PATH 1
19 #endif
20
21

```

图 34.2.11 添加内容

添加完成之后保存退出，再次执行 make 编译，这样就会编译成功了。

```

objs/addon/nginx-rtmp-module/nginx_rtmp_relay_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_bandwidth.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_exec_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_auto_push_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_notify_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_log_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_limit_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_bitop.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_proxy_protocol.o \
objs/addon/hls/nginx_rtmp_hls_module.o \
objs/addon/dash/nginx_rtmp_dash_module.o \
objs/addon/hls/nginx_rtmp_mpegts.o \
objs/addon/dash/nginx_rtmp_mp4.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_stat_module.o \
objs/addon/nginx-rtmp-module/nginx_rtmp_control_module.o \
objs/nginx_modules.o \
-ldl -lcrypt -lpcrc -lssl -lcrypto -ldl -lz \
-Wl,-E
make[1]: Leaving directory '/home/dt/tools/nginx-1.20.0'
dt@dt-virtual-machine:~/tools/nginx-1.20.0$

```

图 34.2.12 编译成功

安装

编译成功之后，接着我们进行安装，执行 make install

```

dt@dt-virtual-machine:~/tools/nginx-1.20.0$ make install
make -f objs/Makefile install
make[1]: Entering directory '/home/dt/tools/nginx-1.20.0'
test -d '/home/dt/tools/nginx-1.20.0/install' || mkdir -p '/home/dt/tools/nginx-1.20.0/install'
test -d '/home/dt/tools/nginx-1.20.0/install/sbin' \
  || mkdir -p '/home/dt/tools/nginx-1.20.0/install/sbin'
test ! -f '/home/dt/tools/nginx-1.20.0/install/sbin/nginx' \
  || mv '/home/dt/tools/nginx-1.20.0/install/sbin/nginx' \
  '/home/dt/tools/nginx-1.20.0/install/sbin/nginx.old'
cp objs/nginx '/home/dt/tools/nginx-1.20.0/install/sbin/nginx'
test -d '/home/dt/tools/nginx-1.20.0/install/conf' \
  || mkdir -p '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/koi-win '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/koi-utf '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/win-utf '/home/dt/tools/nginx-1.20.0/install/conf'
test -f '/home/dt/tools/nginx-1.20.0/install/conf/mime.types' \
  || cp conf/mime.types '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/mime.types '/home/dt/tools/nginx-1.20.0/install/conf/mime.types.default'
test -f '/home/dt/tools/nginx-1.20.0/install/conf/fastcgi_params' \
  || cp conf/fastcgi_params '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/fastcgi_params \
  '/home/dt/tools/nginx-1.20.0/install/conf/fastcgi_params.default'
test -f '/home/dt/tools/nginx-1.20.0/install/conf/fastcgi.conf' \
  || cp conf/fastcgi.conf '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/fastcgi.conf '/home/dt/tools/nginx-1.20.0/install/conf/fastcgi.conf.default'
test -f '/home/dt/tools/nginx-1.20.0/install/conf/uwsgi_params' \
  || cp conf/uwsgi_params '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/uwsgi_params \
  '/home/dt/tools/nginx-1.20.0/install/conf/uwsgi_params.default'
test -f '/home/dt/tools/nginx-1.20.0/install/conf/scgi_params' \
  || cp conf/scgi_params '/home/dt/tools/nginx-1.20.0/install/conf'
cp conf/scgi_params \
  '/home/dt/tools/nginx-1.20.0/install/conf/scgi_params.default'
test -f '/home/dt/tools/nginx-1.20.0/install/conf/nginx.conf' \
  || cp conf/nginx.conf '/home/dt/tools/nginx-1.20.0/install/conf/nginx.conf'
cp conf/nginx.conf '/home/dt/tools/nginx-1.20.0/install/conf/nginx.conf.default'
test -d '/home/dt/tools/nginx-1.20.0/install/logs' \
  || mkdir -p '/home/dt/tools/nginx-1.20.0/install/logs'
test -d '/home/dt/tools/nginx-1.20.0/install/logs' \
  || mkdir -p '/home/dt/tools/nginx-1.20.0/install/logs'
test -d '/home/dt/tools/nginx-1.20.0/install/html' \
  || cp -R html '/home/dt/tools/nginx-1.20.0/install'
test -d '/home/dt/tools/nginx-1.20.0/install/logs' \
  || mkdir -p '/home/dt/tools/nginx-1.20.0/install/logs'
make[1]: Leaving directory '/home/dt/tools/nginx-1.20.0'
dt@dt-virtual-machine:~/tools/nginx-1.20.0$

```

图 34.2.13 安装

笔者将 nginx 安装到了 nginx-1.20.0 目录下的 install 目录中, 进入到 install 目录下:

```

dt@dt-virtual-machine:~/tools/nginx-1.20.0$ cd install/
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install$
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install$ ls
conf  html  logs  sbin
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install$
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install$

```

图 34.2.14 安装目录下的文件夹

conf 目录下有很多的配置文件, 如下所示:

```

dt@dt-virtual-machine:~/tools/nginx-1.20.0/install$ ls conf/*
conf/fastcgi.conf          conf/fastcgi_params.default  conf/mime.types          conf/nginx.conf.default  conf/uwsgi_params
conf/fastcgi.conf.default  conf/koi-utf                 conf/mime.types.default  conf/scgi_params         conf/uwsgi_params.default
conf/fastcgi_params       conf/koi-win                 conf/nginx.conf          conf/scgi_params.default  conf/win-utf
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install$

```

图 34.2.15 conf 目录下的配置文件

其中 nginx.conf 配置文件很重要, 后续我们会对该配置文件进行相应的配置。

sbin 目录下有一个可执行程序 nginx:

```
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$ ls
nginx
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$ file nginx
nginx: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter
966901ebbf81f, not stripped
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$ ls -lh nginx
-rwxrwxr-x 1 dt dt 7.5M 12月 10 16:17 nginx
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$
```

图 34.2.16 nginx 可执行程序

这个可执行文件是一个“not stripped”文件,说明该文件包含了很多的调试信息,所以导致该文件特别大 7.5MB,可以执行下面这条命令去掉调试信息、减少文件的大小:

```
arm-poky-linux-gnueabi-strip --strip-debug nginx
```

```
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$ ls -lh nginx
-rwxrwxr-x 1 dt dt 7.5M 12月 10 16:17 nginx
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$ arm-poky-linux-gnueabi-strip --strip-debug nginx
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$ ls -lh nginx
-rwxrwxr-x 1 dt dt 1.1M 12月 10 16:32 nginx
dt@dt-virtual-machine:~/tools/nginx-1.20.0/install/sbin$
```

图 34.2.17 去除调试信息

nginx 可执行程序用于启动流媒体服务。

现在我们需要将安装目录下的这些文件拷贝到开发板 Linux 系统上,再进行拷贝之前,需要先将开发板出厂系统中已经移植好的 nginx 给移除,进入到开发板 Linux 系统中,执行下面这些命令移除出厂系统自带的 nginx 程序和相应的配置文件:

```
rm -rf /usr/sbin/nginx
rm -rf /etc/nginx/*
```

```
root@ATK-IMX6U:~# ls
driver shell
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# rm -rf /usr/sbin/nginx
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# rm -rf /etc/nginx/*
root@ATK-IMX6U:~#
```

图 34.2.18 删除出厂系统原有的 nginx

接下来我们将 nginx 安装目录下 sbin 中的 nginx 拷贝到开发板 Linux 系统/home/root 目录下,如下所示:

```
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ls
driver nginx shell
root@ATK-IMX6U:~#
root@ATK-IMX6U:~#
```

图 34.2.19 将 nginx 拷贝到开发板

接着再将安装目录下的 conf、logs、html 文件夹拷贝到开发板 Linux 系统的/etc/nginx 目录下,如下所示:

```

root@ATK-IMX6U:~# cd /etc/nginx/
root@ATK-IMX6U:/etc/nginx#
root@ATK-IMX6U:/etc/nginx# pwd
/etc/nginx
root@ATK-IMX6U:/etc/nginx#
root@ATK-IMX6U:/etc/nginx# ls
conf  html  logs
root@ATK-IMX6U:/etc/nginx#
root@ATK-IMX6U:/etc/nginx#

```

图 34.2.20 将 conf、logs、html 文件夹拷贝到开发板

34.3 测试 nginx

上小节我们已经将 nginx 移植到了开发板上, 本小节进行测试、验证, 看看 nginx 是否能够正常工作。先重启开发板, 重启进入系统后, 进入到/home/root 目录下, 执行 nginx 程序。

```
./nginx -V          # 查看版本信息
```

```

root@ATK-IMX6U:~# ls
driver  nginx  shell
root@ATK-IMX6U:~#
root@ATK-IMX6U:~# ./nginx -V
nginx version: nginx/1.20.0
built by gcc 5.3.0 (GCC)
built with OpenSSL 1.0.2h  3 May 2016
TLS SNI support enabled
configure arguments: --prefix=/home/dt/tools/nginx-1.20.0/install --with-http_ssl_module
e/dt/tools/nginx-rtmp-module
root@ATK-IMX6U:~#

```

图 34.3.1 查看 nginx 版本信息

执行./nginx -h 查看帮助信息:

```

root@ATK-IMX6U:~# ./nginx -h
nginx version: nginx/1.20.0
Usage: nginx [-?hvVtTq] [-s signal] [-p prefix]
           [-e filename] [-c filename] [-g directives]

Options:
  -?, -h      : this help
  -v          : show version and exit
  -V          : show version and configure options then exit
  -t          : test configuration and exit
  -T          : test configuration, dump it and exit
  -q          : suppress non-error messages during configuration testing
  -s signal   : send signal to a master process: stop, quit, reopen, reload
  -p prefix   : set prefix path (default: /home/dt/tools/nginx-1.20.0/install/)
  -e filename : set error log file (default: logs/error.log)
  -c filename : set configuration file (default: conf/nginx.conf)
  -g directives : set global directives out of configuration file

root@ATK-IMX6U:~#

```

图 34.3.2 查看帮助信息

接下来我们要启动 nginx, 执行如下命令:

```
./nginx -p /etc/nginx
```

此时 nginx 服务便在后台运行了, 通过 ps 命令可查看到:

```
ps -aux
```

```

root      704  0.0  1.2  5444  3092  ttyxc0  Ss   14:03  0:00  -sh
root      705  0.0  0.1   2724   348  tty1    Ss+  14:03  0:00  /sbin/getty 38400 tty1
root      755  0.0  2.5 172656  6268  ?       S<l  14:03  0:00  /usr/bin/pulseaudio --start --log-target=syslog
root      764  0.0  1.2 12288  3188  ?       S1   14:03  0:00  /usr/sbin/console-kit-daemon --no-daemon
root      841  0.0  0.0  0  0  ?       S    14:03  0:00  [kworker/u2:5]
root      883  0.0  0.0  0  0  ?       S    14:18  0:00  [kworker/0:2]
root      889  0.0  0.0  0  0  ?       S    14:23  0:00  [kworker/0:0]
root      895  0.0  0.6 13220  1544  ?       Ss   14:26  0:00  nginx: master process ./nginx -p /etc/nginx
nobody    896  0.0  0.8 13352  2212  ?       S    14:26  0:00  nginx: worker process
root      898  0.0  0.7  4732  1792  ttyxc0  R+   14:26  0:00  ps -aux
root@ATK-IMX6U:~#

```

图 34.3.3 查看 nginx 服务

此时我们可以打开电脑浏览器，输入开发板的 IP 地址，如下所示：

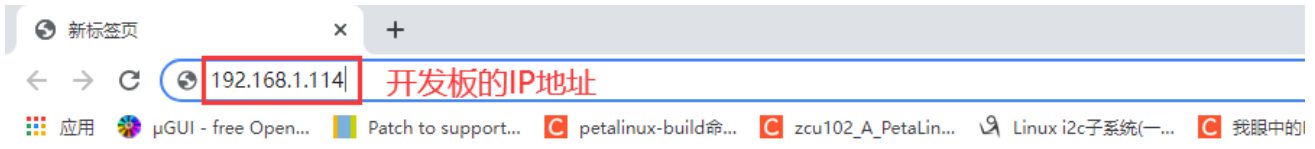


图 34.3.4 输入开发板 IP 地址

按下回车，如下所示：

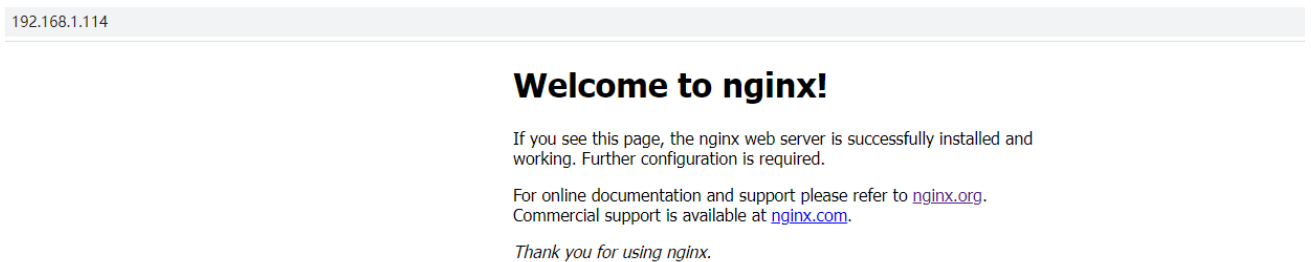


图 34.3.5 welcome to nginx

如果显示出上面这个页面，表示我们的 nginx 已经正常工作了。

34.4 配置 nginx

后续我们会使用 FFmpeg 进行推流，将视频流通过 RTMP 推给 nginx 流媒体服务器，在此之前，我们需要对 nginx 进行配置，打开 nginx 的配置文件/etc/nginx/conf/nginx.conf，添加如下内容：

```

rtmp {
    server {
        listen 1935;          # 监听 1935 端口
        chunk_size 4096;

        application live{

```

```

    allow publish 127.0.0.1;
    allow play all;
    live on;      # 打开直播
    record off;   # 关闭 record
    meta copy;
}

application hls {
    live on;
    hls on;
    hls_path /tmp/hls;
    hls_fragment 8s;
}
}
}

```

如下所示:

```

events {
    worker_connections 1024;
}

rtmp {
    server {
        listen 1935;
        chunk_size 4096;

        application live {
            allow publish 127.0.0.1;
            allow play all;
            live on;
            record off;
            meta copy;
        }

        application hls {
            live on;
            hls on;
            hls_path /tmp/hls;
            hls_fragment 8s;
        }
    }
}

http {

```

添加这一段

图 34.4.1 添加内容

添加完成之后保存退出即可!

接着执行如下命令重启 nginx:

```
./nginx -p /etc/nginx -s reload
```

34.5 使用 FFmpeg 推流

nginx 重启之后,接着我们便可使用 FFmpeg 进行推流,将视频流数据通过 RTMP 推给 nginx 流媒体服务器,执行如下命令进行推流:

```
ffmpeg -re -i /run/media/mmcbk0p1/testVideo.mp4 -c:av copy -f flv rtmp://127.0.0.1/live/mytest
```


简单地介绍一下这些参数, 首先 `-i` 表示输入视频数据, 这里我们使用了一个 `mp4` 视频文件; `rtmp://127.0.0.1/live/mytest` 表示将视频流通过 RTMP 推给流媒体服务器, 这里因为我们的服务器和推流端都是开发板, 所以这个 IP 地址 `127.0.0.1` 指的就是本机的流媒体服务器。

```
root@ATK-IMX6U:~# ffmpeg -re -i /run/media/mmcblk0p1/testVideo.mp4 -c:av copy -f flv rtmp://127.0.0.1/live/mytest
ffmpeg version 3.0 Copyright (c) 2000-2016 the FFmpeg developers
  built with gcc 5.3.0 (GCC)
  configuration: --disable-stripping --enable-pic --enable-shared --enable-pthreads --cross-prefix=arm-poky-linux-
at-abi-hard -mcpu=cortex-a7 --sysroot=/home/linuxdzm/alientek_workspace/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sy
le --extra-cflags=' -O2 -pipe -g -feliminate-unused-debug-types -fdebug-prefix-map=/home/linuxdzm/alientek_workspa
x-gnueabi/ffmpeg/3.0-r0=/usr/src/debug/ffmpeg/3.0-r0 -fdebug-prefix-map=/home/linuxdzm/alientek_workspace/ycoto/fs
=/home/linuxdzm/alientek_workspace/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6ull14x14evk= -march=armv7v
ientek_workspace/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6ull14x14evk' --extra-ldflags='-Wl,-O1 -Wl,-l
ce/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6ull14x14evk --enable-hardcoded-tables --libdir=/usr/lib --
vdevice --enable-avfilter --enable-avformat --disable-avresample --enable-bzlib --disable-libfaac --enable-gpl --d
sable-libmp3lame --disable-openssl --enable-postproc --disable-libschrödinger --disable-libspeex --enable-swresam
--enable-x11grab --enable-libx264 --enable-outdev=xv
  libavutil      55. 17.103 / 55. 17.103
  libavcodec     57. 24.102 / 57. 24.102
  libavformat   57. 25.100 / 57. 25.100
  libavdevice   57.  0.101 / 57.  0.101
  libavfilter    6. 31.100 /  6. 31.100
  libswscale    4.  0.100 /  4.  0.100
  libswresample 2.  0.101 /  2.  0.101
  libpostproc   54.  0.100 / 54.  0.100
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from '/run/media/mmcblk0p1/testVideo.mp4':
Metadata:
  major_brand      : isom
  minor_version   : 512
  compatible_brands: isomiso2mp41
  date            : Thu Apr  6 14:49:08 2017
  encoder         : Lavf57.41.100
Duration: 00:03:33.18, start: 0.000000, bitrate: 650 kb/s
Stream #0:0(und): Video: mpeg4 (Simple Profile) (mp4v / 0x7634706D), yuv420p, 640x360 [SAR 1:1 DAR 16:9], 515
Metadata:
  handler name    : VideoHandler
```

图 34.5.1 FFmpeg 推流

现在我们可以进行拉流了, 可以将我们的 Windows 主机作为拉流端, 使用 VLC 软件进行拉流, VLC 软件大家自己下载、安装好。安装好之后打开 VLC, 如下所示:

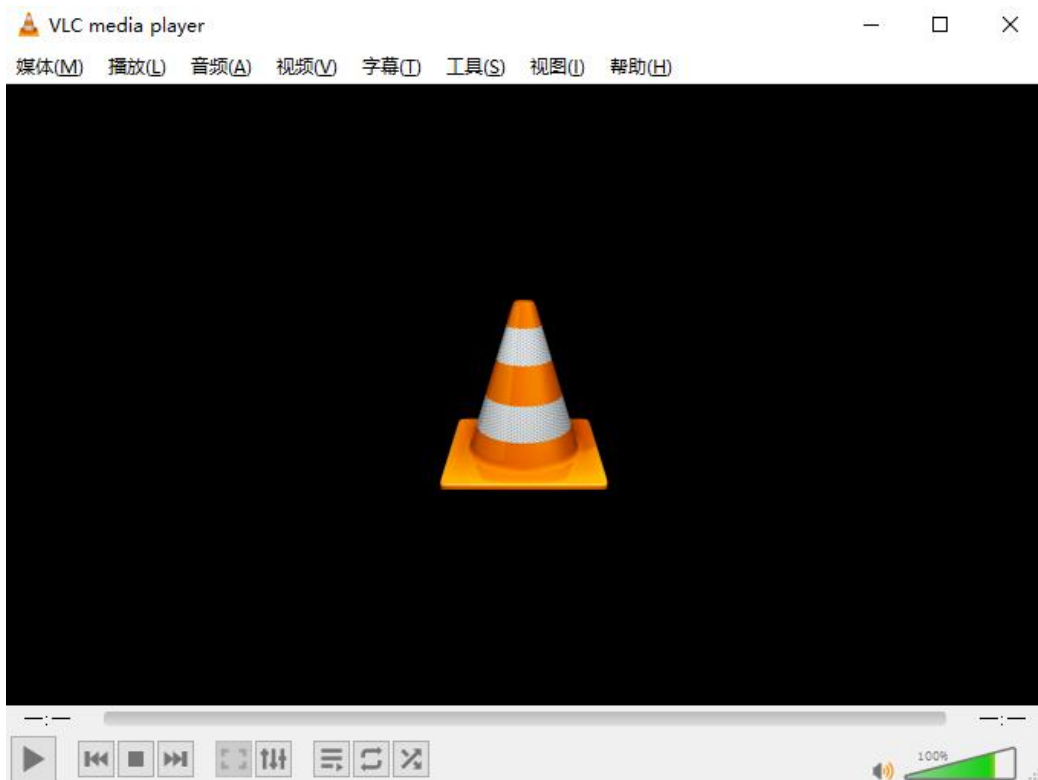


图 34.5.2 VLC 软件

点击左上角“媒体”--->“打开网络串流”:



图 34.5.3 网络串流

输入流媒体服务器的 IP 地址以及路径，笔者使用的开发板对应的 IP 地址为 192.168.1.114，点击“播放”即可从 RTMP 流媒体服务器拉取视频数据进行播放，如下所示：



图 34.5.4 VLC 拉流播放视频

既有画面也有声音！

接下来我们再使用摄像头进行测试，使用 FFmpeg 采集摄像头视频数据将其发送给 nginx 流媒体服务器，执行如下命令：

```
ffmpeg -f v4l2 -video_size 320x240 -framerate 15 -i /dev/video2 -q 10 -f flv rtmp://127.0.0.1/live/mytest
```

```
root@ATK-IMX6U:~# ffmpeg -f v4l2 -video_size 320x240 -framerate 15 -i /dev/video2 -q 10 -f flv rtmp://127.0.0.1/live/mytest
ffmpeg version 3.0 Copyright (c) 2000-2016 the FFmpeg developers
  built with gcc 5.3.0 (GCC)
  configuration: --disable-stripping --enable-pic --enable-shared --enable-threads --cross-prefix=arm-poky-linux-gnueabi- --
at-abi=hard -mcpu=cortex-a7 --sysroot=/home/linuxdzm/alientek_workspace/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6
le --extra-cflags='-O2 -pipe -g -feliminate-unused-debug-types -fdebug-prefix-map=/home/linuxdzm/alientek_workspace/ycoto/fs
x-gnueabi/ffmpeg/3.0-r0=/usr/src/debug/ffmpeg/3.0-r0 -fdebug-prefix-map=/home/linuxdzm/alientek_workspace/ycoto/fsl-release-
=/home/linuxdzm/alientek_workspace/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6ull14x14evk' --arch=armv7ve -mfpu=ne
ientek_workspace/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6ull14x14evk' --extra-ldflags='-Wl,-O1 -Wl,--hash-style=
ce/ycoto/fsl-release-bsp/buildQt5.12.9/tmp/sysroots/imx6ull14x14evk --enable-hardcoded-tables --libdir=/usr/lib --shlibdir=/u
vdevice --enable-avfilter --enable-avformat --disable-avresample --enable-bzlib --disable-libfaac --enable-gpl --disable-libg
sable-libmp3lame --disable-openssl --enable-postproc --disable-libschrödinger --disable-libspeex --enable-swresample --enab
--enable-x11grab --enable-libx264 --enable-outdev=xv
  libavutil      55. 17.103 / 55. 17.103
  libavcodec     57. 24.102 / 57. 24.102
  libavformat    57. 25.100 / 57. 25.100
  libavdevice    57.  0.101 / 57.  0.101
  libavfilter     6. 31.100 /  6. 31.100
  libswscale     4.  0.100 /  4.  0.100
  libswresample  2.  0.101 /  2.  0.101
  libpostproc   54.  0.100 / 54.  0.100
[video4linux2,v4l2 @ 0xdaf240] The driver changed the time per frame from 1/15 to 1/30
Input #0, video4linux2,v4l2, from '/dev/video2':
  Duration: N/A, start: 2138.456763, bitrate: 36864 kb/s
  Stream #0:0: Video: rawvideo (YUY2 / 0x32595559), yuv422, 320x240, 36864 kb/s, 30 fps, 30 tbr, 1000k tbn, 1000k tbc
Output #0, flv, to 'rtmp://127.0.0.1/live/mytest':
  Metadata:
    encoder      : Lavf57.25.100
  Stream #0:0: Video: flv1 (flv) ([2][0][0][0] / 0x0002), yuv420p, 320x240, q=2-31, 200 kb/s, 30 fps, 1k tbn, 30 tbc
  Metadata:
    encoder      : Lavc57.24.102 flv
  Side data:
    unknown side data type 10 (24 bytes)
Stream mapping:
  Stream #0:0 -> #0:0 (rawvideo to flv1)
```

图 34.5.5 摄像头监控

这里笔者使用了一个 USB 摄像头进行测试, 命令执行之后, 接着在 Windows 下使用 VLC 拉流播放摄像头采集到的画面, 实现视频摄像头监控。



图 34.5.6 VLC 拉流播放摄像头采集到的画面

测试发现, 延迟太高了, 导致开发板当前采集到的画面与 VLC 播放到的画面并不同步, 笔者实测大概有 5、6 秒的延迟, 为什么会这样呢? 笔者认为有两方面的原因:

- 我们的 I.MX6U 开发板性能太弱了, 虽然我们执行的是一条命令, 但是 FFmpeg 内部却进行很多的处理, 譬如对视频、音频数据的处理, 由于 I.MX6U 本就没有硬件视频解码, 完全依赖于软件处理, 导致会耗费相当大的时间。
- 在我们本次实验中, 由于服务器和推流端都是开发板, 本来性能就弱, 还做了这么多的工作, 必然就会导致延迟大, 你可以使用一款高性能的开发板进行测试对比, 你就会发现确实是咱们使用开发板性能太弱了!

好了, 本章的内容就到此结束了。

