嵌入式 Linux 软件指南

# 序言

# 目录

目录

[序言 1](#_Toc1714018863)

[目录 2](#_Toc886127346)

[第一章：Linux 概论 8](#_Toc2082909122)

[1. Linux 操作系统历史 8](#_Toc35045191)

[(1)、创世纪的 UNIX 8](#_Toc293183565)

[(2)、UNIX 与 MINIX 11](#_Toc521003882)

[(3)、LINUX 12](#_Toc2141897145)

[4. linux 操作系统和常用命令 17](#_Toc793132123)

[（1）、认识 Linux 17](#_Toc1883424783)

[（2）、 Linux 中的命令（软件） 34](#_Toc336128420)

[（3）、Shell 脚本 44](#_Toc249275332)

[（3）、Linux 实用 52](#_Toc222060208)

[（4）、Linux 网络配置 55](#_Toc1986291369)

[第二章：Linux 应用开发 56](#_Toc1460659186)

[1. 操作系统内核 56](#_Toc1177714841)

[（1）、 操作系统有四个基本特性 56](#_Toc78894488)

[（2）、 操作系统分类 58](#_Toc121440968)

[（3）、 操作系统的内核结构 60](#_Toc523913827)

[2. 开发环境 62](#_Toc806859234)

[（1）、 GCC 63](#_Toc1935899998)

[（3）、 Makefile 72](#_Toc194533084)

[3. Linux c 87](#_Toc44881802)

[C 语言标准 88](#_Toc644722325)

[C 编译器 89](#_Toc1438021112)

[C 标准库 89](#_Toc1617572205)

[4. Linux 进程 106](#_Toc1713456774)

[Linux 进程模型 107](#_Toc560916277)

[Linux 进程状态 111](#_Toc572879479)

[Linux 进程关系 112](#_Toc667935569)

[Linux 进程调度 113](#_Toc280443154)

[进程的创建与运行 120](#_Toc1686260068)

[5. 守护进程 128](#_Toc234470784)

[守护进程概述 128](#_Toc1166570501)

[进程与终端 128](#_Toc1621685542)

[查看守护进程 128](#_Toc269515975)

[Linux守护进程编写 129](#_Toc1459754066)

[6. 线程 134](#_Toc2142689424)

[7. 线程属性 150](#_Toc263929472)

[8. 线程同步异步 153](#_Toc105402541)

[线程同步 154](#_Toc1878630559)

[线程异步 172](#_Toc600057893)

[9. IPC 178](#_Toc354677873)

[管道（Pipe） 178](#_Toc2100690768)

[消息队列（Message Queues）： 184](#_Toc438865614)

[共享内存（Shared Memory）： 188](#_Toc1815337059)

[信号（Signals）： 191](#_Toc1130921961)

[信号量（Semaphores）： 195](#_Toc517760103)

[互斥锁（Mutexes）： 199](#_Toc1936778028)

[套接字（Sockets）： 201](#_Toc1654835788)

[文件锁（File Locking）： 205](#_Toc1324619337)

[10. 网络编程 210](#_Toc1725194378)

[网络基础概念 210](#_Toc1849368872)

[编程方式 214](#_Toc1369501139)

[应用协议 253](#_Toc222433055)

[第三章：软件调试 259](#_Toc1139906336)

[1. GDB 259](#_Toc839589696)

[GDB 简介 259](#_Toc1935889829)

[GDB 安装 260](#_Toc1700822614)

[GDB 基本使用 265](#_Toc1412469176)

[GDB 命令介绍 277](#_Toc456341750)

[2. Linux 栈回溯 294](#_Toc1981265768)

[ELF 294](#_Toc951245596)

[CPU 代码执行方式 295](#_Toc690812535)

[ELF 文件格式分析 297](#_Toc1000352621)

[binutils 工具集 317](#_Toc425447490)

[栈回溯原理 332](#_Toc960328510)

[第四章：图形显示 354](#_Toc312623039)

[1. Framebuffer 概念 354](#_Toc420653266)

[FrameBuffer 的操作原理 357](#_Toc1224257983)

[FrameBuffer 编程 358](#_Toc418025580)

[2. GUI 386](#_Toc151800177)

[GUI 有关名词简介 387](#_Toc1824315876)

[MiniGUI 390](#_Toc772703454)

[3. QT 412](#_Toc105007297)

[简介 412](#_Toc115697842)

[安装 413](#_Toc440556865)

[开始 413](#_Toc1235929259)

[Qt 主要模块 418](#_Toc633457945)

[Object 428](#_Toc229851245)

[QMetaObject 429](#_Toc743281399)

[Signal-Slot 432](#_Toc1958077282)

[Event System 441](#_Toc1955045623)

[Garbage Collection 442](#_Toc445166624)

[Widget 442](#_Toc1180094774)

[WindowOption 443](#_Toc29995031)

[Attribute 444](#_Toc1585072960)

[Event 444](#_Toc2019684470)

[Layout 448](#_Toc1965884860)

[Style 450](#_Toc1138411926)

[更进一步 450](#_Toc1284669998)

[第五章：大容量数据存储 451](#_Toc274742963)

[1. SQL 451](#_Toc972194047)

[数据库产品 451](#_Toc88431946)

[SQL 储存结构 453](#_Toc965555498)

[SQL 结构化语言 454](#_Toc1972546668)

[SQLite 459](#_Toc513879436)

[第六章：软件框架设计 468](#_Toc1925884008)

[1. C 程序的背后 468](#_Toc137686060)

[1.1 C 函数的汇编结构 468](#_Toc934532702)

[1.2 C 程序库 471](#_Toc1002658343)

[1.3 C 程序编译连接 472](#_Toc555711640)

[1.4 extern C 关键字 477](#_Toc1086332880)

[2. 面向对象编程 481](#_Toc679490571)

[2.1 概念解释 481](#_Toc1328415094)

[2.2 举个例子解释 482](#_Toc1191340177)

[2.3 面向对象编程的特性 486](#_Toc795188414)

[2.4 五大基本原则：SPR, OCP, LSP, DIP, ISP 487](#_Toc1768971960)

[第七章：交叉开发 488](#_Toc279785788)

[1、嵌入式 Linux 环境 488](#_Toc1428646359)

[1.1、Linux 完整操作系统的整体架构 488](#_Toc1998823205)

[2、嵌入式 mcu 环境 489](#_Toc1023067188)

[（1）、 交叉编译环境 489](#_Toc1239239994)

[1. 嵌入式 Linux 系统环境组成 495](#_Toc1806385181)

[2. 网络下载 rftp nfs dnw 495](#_Toc1468233812)

[3. ext4 文件系统 495](#_Toc271851120)

[Ext 495](#_Toc1836380212)

[核心设计 497](#_Toc905823124)

[数据存放区 497](#_Toc144051942)

[查看文件或者文件系统的状态 498](#_Toc1654781424)

[中介数据(metadata) 499](#_Toc2044235051)

[示例说明 501](#_Toc1428721941)

[1. inode的作用 501](#_Toc1929524387)

[示例 501](#_Toc868945450)

[2. block的重要性 502](#_Toc1517153887)

[示例 502](#_Toc747596237)

[3. inode和block与文件大小的关系 502](#_Toc694008470)

[计算单文件最大容量 503](#_Toc2031033324)

[查看磁盘和文档的容量 504](#_Toc525996598)

[1. 查看文件系统的整体磁盘容量 504](#_Toc831694530)

[总结 506](#_Toc818082378)

[第八章：系统内核 506](#_Toc1528654941)

[1. 嵌入式操作系统的构建方法 506](#_Toc1387406171)

[2. uboot 的功能和作用 506](#_Toc1904415258)

[uboot简介 506](#_Toc60661865)

[启动uboot 507](#_Toc568337617)

[uboot命令 507](#_Toc948271788)

[uboot常见命令 511](#_Toc855850279)

[mmc命令 511](#_Toc189825929)

[查看mmc设备 512](#_Toc1228057576)

[查看分区信息 513](#_Toc137012990)

[mmc操作 513](#_Toc41165487)

[文件系统操作命令 513](#_Toc103641116)

[FAT格式文件系统 514](#_Toc1376252984)

[ext4格式文件系统 516](#_Toc1847550668)

[uboot启动内核过程 517](#_Toc1571874928)

[uboot环境参数介绍 520](#_Toc1648104104)

[3. uboot 编译与配置 521](#_Toc1536447232)

[Linux U-Boot 开发指南 521](#_Toc330214405)

[1 前言 521](#_Toc1792156047)

[1.1 编写目的 521](#_Toc1043745008)

[1.2 适用范围 521](#_Toc226965808)

[1.3 相关人员 521](#_Toc1073394340)

[2 LICHEE 类宏关键字解释 521](#_Toc825785748)

[3 编译方法介绍 522](#_Toc1095911258)

[3.1 准备编译工具链 522](#_Toc443064579)

[3.2 快速编译 boot0 及 U-Boot 522](#_Toc1573381985)

[3.3 编译 U-Boot 522](#_Toc1789919728)

[3.4 编译 boot0/fes/sboot 522](#_Toc326614255)

[4 U-Boot 功能及其配置方法/文件介绍 523](#_Toc2099378583)

[4.1 U-Boot 功能介绍 523](#_Toc474130611)

[4.2 U-Boot 功能配置方法介绍 524](#_Toc1144696634)

[4.2.1 通过 defconfig 方式配置 524](#_Toc1480549877)

[4.2.2 通过 menuconfig 方式配置 524](#_Toc1861536782)

[4.3 U-Boot 配置参数文件介绍 525](#_Toc901628244)

[4.3.1 U-Boot-dts 路径 525](#_Toc1541211742)

[4.3.2 U-Boot-dts，defconfig 配置 525](#_Toc282390751)

[4.3.3 U-Boot-dts 注意事项 526](#_Toc1849900032)

[4.3.3.1 编译注意事项 526](#_Toc249578373)

[4.3.3.2 语法注意事项 527](#_Toc472216681)

[4.3.3.3 运行时注意事项 527](#_Toc930473961)

[5 U-Boot 常用命令介绍 529](#_Toc386591363)

[5.1 env 命令说明 529](#_Toc513382168)

[5.2 sunxi\_flash read 命令说明 531](#_Toc1034115077)

[5.2.1 使用方法 531](#_Toc1762844348)

[5.2.2 使用示例 531](#_Toc213449188)

[5.3 fastboot 命令说明 531](#_Toc458506358)

[5.3.1 使用前提 531](#_Toc1263464804)

[5.3.2 使用步骤 532](#_Toc1749896420)

[5.3.3 fastboot 基本命令使用示例 532](#_Toc788720763)

[5.4 fat 命令说明 533](#_Toc908137203)

[5.5 md 命令说明 537](#_Toc646157780)

[5.6 FDT 命令说明 537](#_Toc1015686571)

[5.6.1 查询配置 538](#_Toc1981531543)

[5.6.2 修改配置 542](#_Toc1471943528)

[5.6.2.1 修改整数配置 542](#_Toc2111597829)

[5.6.2.2 修改字符串配置 543](#_Toc277112475)

[5.6.3 GPIO 或者 PIN 配置特殊说明 544](#_Toc897841866)

[5.6.3.1 Pin 配置说明 544](#_Toc1754033909)

[5.6.3.2 查看 PIN 配置 544](#_Toc603726730)

[5.6.3.3 修改 PIN 配置 546](#_Toc849736801)

[5.6.3.4 GPIO 配置说明 546](#_Toc80680872)

[6 基本调试方法介绍 548](#_Toc1748423364)

[7 进入烧写的方法 549](#_Toc182803030)

[8 常用接口函数 549](#_Toc1942217654)

[8.1 fdt 相关接口 549](#_Toc502567961)

[8.2 env 相关接口函数 552](#_Toc1724014772)

[8.3 调用 U-Boot 命令行 553](#_Toc77124758)

[8.4 Flash 的读写 554](#_Toc204984345)

[8.5 获取分区信息 555](#_Toc1973593145)

[8.6 GPIO 相关操作 557](#_Toc549341439)

[9 常用资源的初始化阶段 558](#_Toc1135458306)

[4. Linux 内核配置 559](#_Toc212700861)

[5. 添加一个 Linux 源码文件 560](#_Toc1062723607)

[6. busybox 功能和作用 564](#_Toc22089736)

[一、Busybox简介 564](#_Toc1975545209)

[（1-1）开源项目 564](#_Toc1276172795)

[（1-2）程序本体较小 564](#_Toc480596094)

[（1-3）使用简单 565](#_Toc1091526365)

[二、Busybox源码目录结构 565](#_Toc878585567)

[三、Busybox程序主体 566](#_Toc1269316857)

[四、Busybox程序运行剖析和实列 570](#_Toc1999663569)

[第九部分：驱动程序 572](#_Toc1524743347)

[1. Linux 平台设备驱动 572](#_Toc137519780)

[dmesg 582](#_Toc1833711464)

[modprobe pltdev 584](#_Toc849203228)

[2. Linux 内核模块的加载卸载 593](#_Toc101633961)

[3. Linux 字符设备驱动模板 593](#_Toc2110823939)

[4. Linux 驱动中的信号量与自旋锁 601](#_Toc1747045094)

[define LOCK\_PADSIZE (offsetof(struct raw\_spinlock, dep\_map)) 603](#_Toc1855667870)

[5. Linux 驱动中断的常用函数及作用 615](#_Toc567067022)

[6. Linux 驱动 platform 虚拟总线 627](#_Toc449298247)

[7. IPCC 通信框架 644](#_Toc1936348743)

[进入remoteproc 子系统目录 653](#_Toc168006738)

[导入M4 内核固件名称 654](#_Toc632101278)

[启动M4 内核 654](#_Toc1731082749)

[停止M4 内核 654](#_Toc670574699)

[8. OpenAMP 655](#_Toc208632402)

[符录 657](#_Toc1808207507)

[参考文献 657](#_Toc875559045)

[索引 657](#_Toc34741900)

# 第一章：Linux 概论

## 1. Linux 操作系统历史

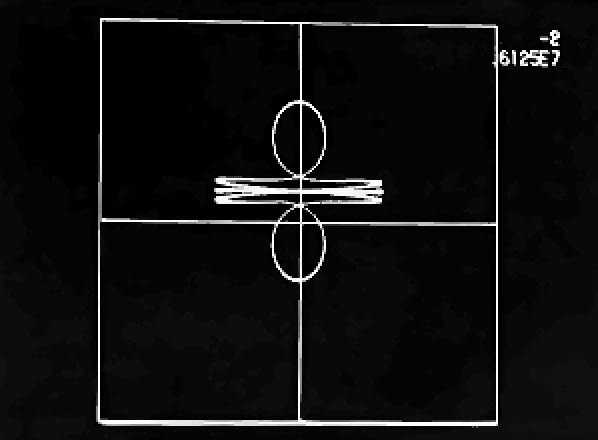
### (1)、创世纪的 UNIX

追溯 Linux 的历史，要从 UNIX 的诞生开始说起。

UNIX 操作系统是由肯•汤普森（Ken Thompson）和丹尼斯•里奇（Dennis Ritchie）合作创造的。关于 UNIX 的部分技术思路则是来源于当时的大型机 Multics 工程计划，该计划是由贝尔实验室、美国麻省理工学院和通用电气公司联合发起，目标是取代当时的批处理操作系统，拥有多道程序的分时处理能力，能够同时为多个机构和公司提供计算支持。这个愿望是非常美好的，美好到以至于目标太过于庞大与空洞，导致 Multics 的开发人员都不知道该做成什么样子才能满足公司对这个工程的要求，所以计划失败了。Multics 虽然失败了，肯•汤普森的故事还在继续。



肯•汤普森出于自己玩 Space Travel 的需求在 Multics 失败后，决定编写一个较小的操作系统来替代 Mutics，小到能够运行在 Digital PDP-7 小型机上。肯参考了 Mutics 中的一些功能设计，编写出了一个能够满足基本功能的分时操作系统。同组人员为了调侃这个只支持两个用户的系统便称这个系统为 UNICS（Uniplexed Information and Computing System）。后来，随着 UNICS 这个名字的传播，谐音的 UNIX 明显更受大家欢迎，于是这个分时系统被正式命名为 UNIX 。



UNIX 最初是由汇编编写的，运行在上面的一些应用则是由 B 语言和汇编语言混合编写而成。由于 B 语言在进行系统编程时总是处处受挫，所以汤普逊和里奇对其进行了改造和扩展，完成后的语言和原来的 B 语言差异较大，所以被命名为 C 语言。C 语言的高效和强大使得编写操作系统变成了可能。汤普逊和里奇在 1973 年中用 C 语言重写了 Unix ，这个首次不是用汇编编写的操作系统成为了当时操作系统届最大胆的创新，也非常具有革命意义。C 语言的简洁紧凑、易移植、易读、易修改的特性赋予了 UNIX 强大的生命力，为 UNIX 的发展奠定了坚实的基础。

1974 年，汤普逊和里奇合作在 《ACM通信》上发表了一篇关于 UNIX 的文章《The UNIX Time-Sharing System》。这篇文章系统的介绍了 UNIX 的目标与实现，UNIX 特性：采用 C 高级语言编写、简单的用户交互 Shell 、高效便捷的树形文件系统、文件，设备，设备的统一处理、内核和核外程序的有机组合、丰富的系统程序。当时计算机大多还处于批处理阶段，这样的系统迅速的在计算机存在的地方运行了起来。

UNIX 的流行和发展并没有给诞生它的 AT&T 公司带来多少利益，但是 AT&T 却在 UNIX 中做了不少的投入。为了取得回报，1982年，AT&T 基于版本 7 开发了UNIX System Ⅲ 的第一个版本，这是一个商业版本仅供出售。同时这个新的 UNIX 商业发布版本不再包含源代码。

UNIX 的哲学 KISS（keep it simple, stupid）成为了程序开发的基础哲学之一。

### (2)、UNIX 与 MINIX

20世纪80年代，由于 AT&T 所有的 UNIX 版权的限制和闭源的操作，荷兰的 Andrew S. Tanenbaum 教授在 1987 年决定写一个不包含任何 AT&T 源代码的 UNIX 系统，名为 MINIX（即小型的 UNIX ），并开放全部源代码给大学教学和研究工作，Minix 于 2000 年重新改为BSD授权，成为自由和开放源码软件，为全球注册商标。



MINIX 属于微内核，对于宏内核的 UNIX 来说，它的内核更小，提供的功能也仅限于线程管理，内存管理和进程间通信。其余的如文件系统，设备驱动等，都是由多进程的组件提供。这种驱动由进程隔离的方式使得系统的各种服务不会相互的影响，同时缺点也是致命致命的，频繁的系统调用会造成频繁的进程切换，而系统切换带来的性能损失是不不可忽略的。

MINIX 的设计目标就是用于大学的教学和研究，所以作为它的开发者 Andrew S. Tanenbaum 教授不愿意在这个系统上添加过多的功能。MINIX 的诞生在计算机历史上并没有掀起多大的波浪，但是它为计算机的教育事业作出的贡献确实巨大的。MINIX 之父 Andrew S. Tanenbaum 的经典之作《操作系统设计与实现》，给 Linux 操作系统的创造者 Linus Torvalds 带来了很大的启发，而 Linux 给世界带来了巨大的改变。

### (3)、LINUX

在同一时刻，操作系统届有着百花切放，百家争鸣的现状。MINIX 虽然也开源，但是其教育系统的特性不愿意添加太多的代码，使其一直停留不前。同为微内核 GNU Hurd 内核由于设计的激进也是迟迟的难产。由 UNIX 衍生出的各种 UNIX 系操作系统大多是闭源的，BSD 虽然为开源系统，但其和 UNIX 的司法官司一直打到了 1994 年 1 月。这桩法律诉讼将 BSD 后续的开发，特别是自由软件，延迟了两年，而没有法律问题的 Linux 内核在这段时间获得了极大的支持。Linux 跟 386BSD 的开发几乎同时起步，Linus 说，当时如果有自由的基于 386 的 Unix-like 操作系统，他就可能不会创造 Linux 。

Linux 内核最初是由林纳斯•托瓦兹（Linus Torvalds）在赫尔辛基大学读书时出于个人爱好而编写的，当时他只是觉得教学用的迷你版 UNIX 操作系统 Minix 能用的东西太少了，也可能是出于实用主义。于是决定自己开发一个操作系统。第 1 版本于 1991 年 9 月在 MINIX 的新闻组论坛上发布，当时仅有 10000 多+行代码。刚开始只有很少的操作系统爱好者在讨论这个系统，并为这个操作系统编写补丁和更多的支持。随着 Linux 在爱好者的支持下，系统越发的完善，论坛上讨论 Linux 的声音也越来越多。渐渐最初的 MINIX 论坛上只剩下讨论 Linux 的声音了。Linux 最初的名字叫 FreaX ,寓意“自由”以及“奇特”,最后带上了一个常用字母X作为结尾,类似与 Unix 之类的系统一样。但是后来以为管理员觉得 FreaX 不好听,而且难念,所以就改成了 Linux ,前面直接取自开发者 Linus ,加上 X 后缀就成了 。

继承于 UNIX 的传统，标准的 POSIX 操作系统接口，开源的特性以及合适的操作系统空窗期让 Linux 在全世界快速的流行起来。随着全世界的工程师的参与，Linux 具备了一个现代操作系统所有的属性：多任务处理（Multitasking）、多用户支持（Multiuser）、虚拟内存（Virtual Memory）、文件系统（File System）、设备驱动程序（Device Drivers）、网络支持（Networking）、安全性（Security）、任务调度（Scheduling）、错误检测和恢复（Error Detection and Recovery）、用户界面命令解释器（Shell）、图形用户界面（Graphical User Interface，GUI）。而在其之上的各种应用软件生态覆盖了各行各业。Linux 运行在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型机和超级计算机。

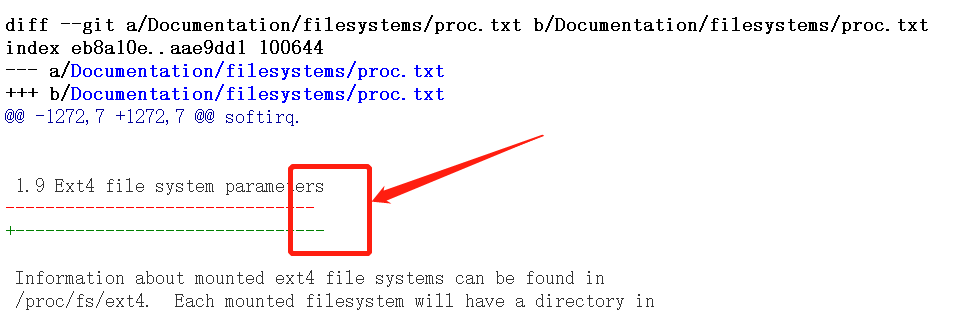
严格来讲，Linux 这个词本身只表示 Linux 内核，但实际上人们已经习惯了用 Linux 来形容整个基于 Linux 内核，并且使用 GNU 工程各种工具和数据库的操作系统。

Linux 是一个开源系统，开源也是 Linux 最大的优势。林纳斯•托瓦兹没有保留 Linux 源代码的版权，在公开源码的同时欢迎任何人给 Linux 提供修改代码。同时 Linux 也是免费的，意味着你可以不花一分钱就可以使用到 Linux 强大的操作系统和完善的软件生态。为了更好的管理和协作，Linux 加入了自由软件基金会，有自由的 GPL 许可证。

1996年，Torvalds 为 Linux 选定了企鹅作为它的吉祥物。Larry Ewing 提供了吉祥物的初稿。为什么是一只企鹅，据说是因为 Linus 小时候被企鹅咬过，也有解释是企鹅是南极洲的标志性动物，根据国际公约，南极洲为全人类共同所有，不属于世界上的任何国家，任何国家都无权将南极洲纳入其版图。Linux 选择企鹅图案作为 Logo，其含义是：开放源代码的 Linux 为全人类共同所有，可公司无权将其私有。



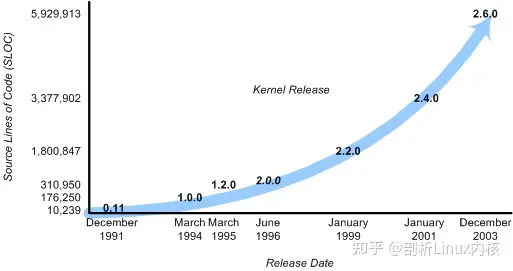
Linux 拥有庞大的社区和社区管理人员，Linus 拥有 Linux 代码的裁定权，能够决定什么样的代码能够合并到 Linux 内核中，而 Linux 各个系统的管理员会审核其他开发者给提交到 Linux 子系统的代码，经过测试后提交给 Linux 等待合并。如同 Linux 的树形文化一样，Linux 社区也呈现出完全树型的结构。层层的管理级别并没有拖乱 Linux 的发展进度，Linux 内核可以让所有人参与进来，非常严谨，只要有错误，任何人都可以修改它。在 Linux git 仓库中的 commit 号为 690b0543a813b0ecfc51b0374c0ce6c8275435f0 的补丁就是一位小女孩提出的。

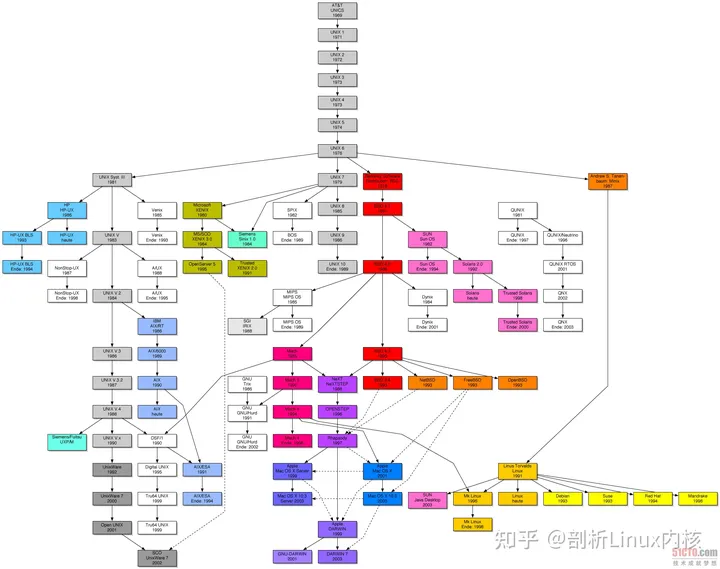


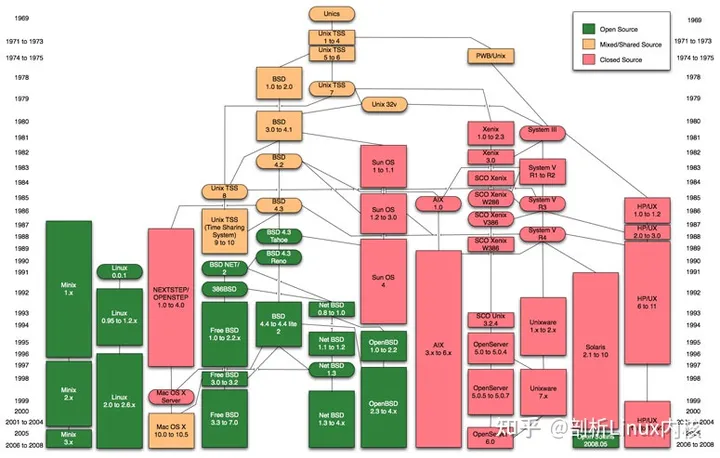
虽然只是添加了一个字符，但是更改的理由是补丁作者的女儿发现那个 「 s 」字母很孤单，用她的话说是不开心，加上「 - 」后就显得开心了。最终 Linus 同意了这个补丁的合并请求，因为 Linux 是严谨的，人人可参与的。

Linux 的社区非常的活跃，全世界的开发者都在为 Linux 的驱动做着贡献。Linux 的可移植性使得 Linux 往往会被很快的移植到新的 CPU 设备中。就这样硬件成就着软件，软件也在成就着硬件。

* 1983年：Richard Stallman创建了以创建一个自由的操作系统为目标的GNU计划。
* 1989年：Richard Stallman撰写了第一版的GNU GPL。
* 1991年：第一个版本0.0.2发布，代码只有约一万行
* 1992年：Linux0.96拥有4万行代码，大约1000人使用且大部分是黑客
* 1993年：Linux0.99发布，代码量10万行，用户约2万，采用GUN的GPL协议
* 1994: 3月, Torvalds认为内核的所有组件已经完全成熟，他放出了Linux的1.0版本。XFree86项目组提供了一个图形化用户界面（GUI）.同年Red Hat公司 和SUSE 发行了他们各自的Linux 1.0分发版本。
* 1995: Linux被移植到DEC Alpha和Sun公司的SPARC平台上，而在接下来的几年里它又被广泛地移植到更多的平台上。
* 1996: Linux内核2.0版本发布。此时内核已经支持多处理器，因而成为各大公司的绝佳选择。
* 1998：很多大公司，诸如IBM, Compaq ，Oracle表示支持Linux系统。另外，一部分程序员开始了图形化用户界面KDE的开发。
* 1999：一些程序员开始致力于开发图形化环境GNOME，它可以替代依靠Qt工具包才能工作的KDE。在这一年里IBM宣布了一项支持Linux的浩大的工程。
* 2004: XFree86小组分裂，同现有的X Windows标准组织 共同成立[http://X.Org](https://link.zhihu.com/?target=http%3A//X.Org)基金会，促使了X Window ServerLinux版本极其快速而迅猛的发展。







Linux 的历史也不是一片平坦，在其刚发布的时候就引来了“Linux 已经过时的言论”。

1992 年，MINIX 的作者 Andrew S. Tanenbaum，在新闻组 comp.os.minix 上发表了一篇 [《LINUX is obsolete》](https://www.oreilly.com/openbook/opensources/book/appa.html)的文章，批判了在 1991 年写一个宏内核系统真是一个糟糕的主意。Linus 也有力的回击了 Andrew 的评论，如果微内核当前已经成熟可用的话，他也不用大费周章的编写 Linux 操作系统了。但在21世纪的现在来看，Linux 系统经受住了历史的考验，我们的 Linux 系统依然稳健的运行在各类硬件上。

自由与商业的竞争。苹果曾聘请过 Torvalds ，因为理念不和而分道扬镳。微软和 Linux 阵营在 1997 年到 2001 年间还是有着很多敌对的情况。这种情况在 1998 年 Eric S. Raymond 发表《万圣节文件》的时候变得明显起来。这里由一位微软工程师写的关于寻求解决自由软件对微软的威胁的策略的文章。

SCO-Linux 争议。2003 年 3 月，SCO Group 指责 IBM 把 UNIX 的代码移植到 Linux 侵犯了他们的版权。SCO 声称它们拥有代码的版权并 IBM 提起了诉讼。Red Hat 又提起了反诉讼，因此 SCO 又提起了其他相关的诉讼。在这些诉讼进行的同时，SCO 开始把 Linux 的许可权卖给那些不愿意冒受 SCO 投诉的险的用户。因为 Novell 也声称拥有 UNIX 的版权，所以它又对 SCO 提起了诉讼。接着 SCO 便声明破产了。

Linux 诞生的时候并没有多么注意商标，虽然一直使用 Linux 的名称。直到从 1994 年开始，越来越多不同国家的人想要将 Linux 注册成自己的商标然后收取金。Linux 的开发人员非常反感这种做法，后来，Torvalds 在 Linux 国际的帮助下得到了 Linux 这个商标，随后他把这个商标转让给了 Linux 国际。目前这个商标的保护由一个专门的基金会——非营利性的Linux标识协会——来管理。

## 4. linux 操作系统和常用命令

### （1）、认识 Linux

Linux 是一个现代操作系统内核，它提供的主要功能是管理好计算机中的硬件，尽可能的屏蔽好各类硬件的差异，为计算机中的软件提供一个标准的统一调用接口。为什么要拥有一个这样的内核呢，这涉及到软件和硬件的协调了。软件是运行在硬件之上的，更具体的说是 CPU 中央处理器提供了基本的动作，这些动作被称为指令集，我们编排好一条条指令，这些指令按照规则组合起来后，就可以称之为软件，软便是松软可变动的意思，意味这软件是随时可变的。但是硬件不一样，硬件在制造好之后，就不会再改变了。在今天，我们研发出了各种个样的硬件，就比如组成计算机的几大件：CPU、内存、硬盘、主板、鼠标、键盘等，不同的厂商造出来的东西都不一样。同样的，不同的硬件就会用不同的动作，不同的动作对应着不同的指令，软件如何想让这些硬件正常的工作起来，那就要添加各种动作，编写软件的人还要思考如何协调好这么多的硬件。面对着越来越庞大的计算机体系，个人的能力是非常有限的，要多个人考虑合作。而考虑合作，就需要有序的沟通，所以对于计算机来说，操作系统内核就出来了，有人负责整个操作系统架构，有人负责具体的各个硬件驱动，也有人负责写其他的程序，需要的时候调用这个操作系统提供的接口就可以了。这下不同的事情就可以交给不同的人来做，分工好了，面对再复杂的事情我们都能化整为零完美解决了。

#### 内核

计算机软件份大致分为两类：系统软件和应用软件。系统软件的任务是管理自身和计算机硬件，应用软件是用来执行用户需要的功能。内核是最基本的系统软件，它是计算机中所有应用软件运行的基石。

#### 用户

Linux 相当于一座大房子，Linux 的用户则相当于房子的主人或者房子的住户，Linux基于用户身份对资源访问进行控制。

Linux 用户账号分为三类：

* 超级用户：root用户是Linux操作系统中默认的超级用户账号，对本主机拥有最高的权限，系统中超级用户是唯一的。
* 普通用户：由root用户或其他管理员用户创建，拥有的权限会受到限制，一般只在用户自己的宿主目录中拥有完整权限。
* 程序用户：在安装Linux操作系统及部分应用程序时，会添加一些特定的低权限用户账号，这些用户一般不允许登录到系统，仅用于维持系统或某个程序的正常运行，如bin、daemon、ftp、mail等。

#### 用户账户管理



与 Linux 用户相关的文件为 /etc/passwd。

nihao@nihao-z690:~$ cat /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
bin:x:2:2:bin:/bin:/usr/sbin/nologin  
sys:x:3:3:sys:/dev:/usr/sbin/nologin  
sync:x:4:65534:sync:/bin:/bin/sync  
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin  
kernoops:x:113:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin  
avahi:x:114:121:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin  
nihao:x:1000:1000:nihao,,,:/home/nihao:/bin/bash  
sshd:x:129:65534::/run/sshd:/usr/sbin/nologin

/etc/passwd是一个系统文件，而不是目录。它是一个文本文件，用于存储系统中所有用户账户的基本信息。

每一行代表一个用户账户，字段之间使用冒号进行分隔，通常包含以下信息：

[用户名]:x:[UID]:[GID]:[用户描述信息]:[主目录]:[登陆后默认使用的 Shell]

* 用户名：标识用户的唯一名称。
* 密码占位符：在过去，密码哈希值存储在此处，现在则通常为 "x" 表示密码已被移动到 /etc/shadow 文件中。
* 用户标识号（UID）：每个用户都有一个唯一的数字标识。
* 组标识号（GID）：指定用户所属的主要组的标识号。
* 用户描述信息：包含有关用户的备注信息或全名等。
* 主目录：用户登录后的起始目录。
* Shell：定义用户登录后默认使用的 shell。

**useradd 添加用户**

基本语法：useradd 用户名（功能描述：添加新用户）

案例：[root@~]#useradd hadoop

**userdel 删除用户**

基本语法：

userdel 用户名（功能描述：删除用户但保存用户主目录）

userdel -r 用户名（功能描述：用户和用户主目录，都删除）

案例：

删除用户但保存用户主目录: [root@~]#userdel hadoop  
删除用户和用户主目录:[root@~]#userdel -r hadoop

**passwd 设置用户密码**

基本语法：passwd 用户名（功能描述：设置用户密码）

案例：  


**id 判断用户是否存在**

基本语法：id 用户名

**su 切换用户**

基本语法：

su 用户名称（功能描述：切换用户，只能获得用户的执行行权限，不能获得环境变量）su - 用户名称（功能描述：切换到用户并获得该用户的环境变量及执行权限）

案例:

[root@~]#su hadoop  
[root@~]#su - hadoop

**who查看登录用户信息**

基本语法:

whoami（功能描述：显示自身用户名称）  
who am i（功能描述：显示登录用户的用户名）  
who（功能描述：看当前有哪些用户登录到了了本台机器器上）

#### 用户组管理

Linux作为多用户、多任务的系统，为了提供更好的用户管理和权限控制机制，使系统管理员能够更高效地管理用户、控制权限，并对文件和目录进行适当的保护。其引入了用户组概念，通过用户组可以简化管理过程，提高系统的安全性和可管理性。

* 基本组（私有组） ：基本组账号只有一个，一般为创建用户时指定的组。
* 附加组（公共组） ：用户除了基本组以外，额外添加指定的组。附加组可以有多个，可以在基本组，也可以同时在附加组  
  

"/etc/group" 是 Linux 系统中的一个文件，用于存储系统中所有用户组的信息。每一行代表一个用户组，它包含了用户组的名称、组密码（通常为空）、组标识号（GID）以及属于该用户组的用户列表。

**groupadd 新增组**

基本语法: groupadd 组名

案例：

增加一个 hadoop 组：groupadd hadoop  
**groupdel 删除组**

基本语法：groupdel 组名

案例：

删除 hadoop 组：groupdel hadoop  
**groupmod 修改组**

基本语法：groupmod -n 新组名 老组名

案例：

修改 hadoop 组名称为 hadoop1 : groupmod -n hadoop1 hadoop  
**查看创建了哪些组**

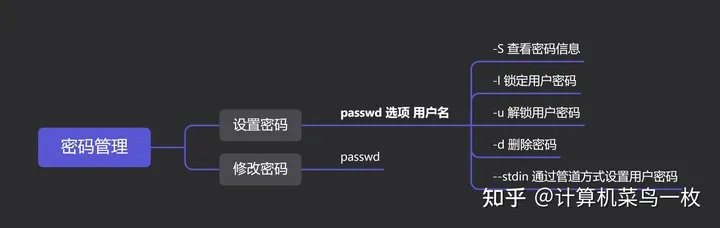
基本语法：cat /etc/group  
**usermod 修改用户所属组**

基本语法：usermod -g 用户组 用户名

案例：

将用户 hadoop 加入 mygroup 用户组：sermod -g mygroup hadoop

#### 用户的密码管理



注意：只有root管理员才可以修该其他用户的密码，普通账户只能修改自己的密码，并且普通用户没有修改其他用户密码的权限

在linux密码管理中，我们还应该熟悉密码策略的配置，如：

密码策略：可以通过配置文件或者命令来设置密码策略，以限制密码的复杂度和有效期限。密码策略通常包括以下内容：

* 密码长度：设置密码的最小长度要求。
* 复杂度要求：可以要求密码包含字母、数字、特殊字符等不同类型的字符。
* 历史密码限制：禁止用户在一段时间内重复使用先前使用过的密码。
* 密码有效期：设置密码的有效期限，过了有效期后需要用户更改密码。
* 警告通知：在密码即将过期之前发送警告通知给用户。

chage 命令：chage 命令用于修改用户的密码策略，比如修改密码过期时间、密码失效时间等。可以使用 chage -l username 查看用户当前的密码策略信息，使用 chage -m 修改密码最小长度，使用 chage -M 修改密码有效期等。

/etc/pam.d/ 目录中的配置文件：PAM 是 Linux 中的身份验证管理机制，可以通过修改 /etc/pam.d/ 目录下的文件来配置密码策略。常见的文件有 common-password 和 system-auth 等，可以在文件中配置密码策略规则。

密码加密算法：Linux 使用密码哈希算法（如 MD5、SHA-256、SHA-512）将用户密码进行加密存储，以确保密码的安全性。

/etc/shadow是一个重要的系统文件，它存储了用户的加密密码和相关的账户信息。该文件只能由系统管理员或具有特权的用户访问。其中包含了每个用户账户的密码哈希值、上次修改密码的时间、密码过期时间、账号是否被锁定等信息。这里的密码哈希值是用户密码经过加密处理后的结果，实际密码并不存储在该文件中。通过将用户输入的密码与保存在/etc/shadow中的密码哈希值进行比对，系统可以验证用户的身份。

#### 用户的权限管理

Linux系统一般将文件可存/取访问的**身份**分为 3 个类别：owner、group、others，且 3 种身份各有read、write、execute等**权限**。

在多用户（可以不同时）计算机系统的管理中，权限是指某个特定的用户具有特定的系统资源使用权力，像是文件夹、特定系统指令的使用。

**常见权限**

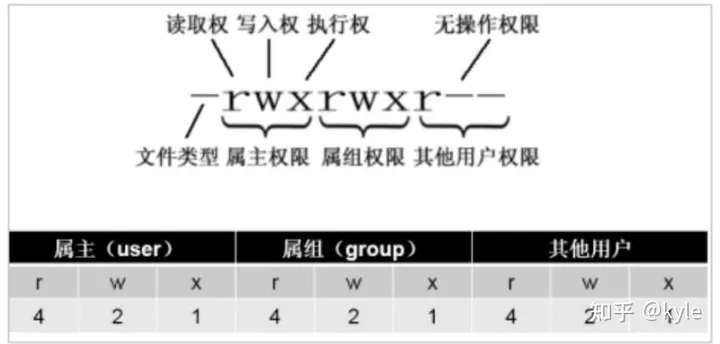
在 Linux 中，文件和目录的权限是由三个基本权限位来控制的，分别是读取权限（r）、写入权限（w）和执行权限（x）。这些权限位可以分别应用于文件的所有者、所属组和其他用户。以下是更详细的说明：

* 读取权限（r）：读取权限允许用户查看文件的内容或目录的列表。对文件而言，具有读取权限的用户可以读取文件的内容；对目录而言，具有读取权限的用户可以列出目录中的文件和子目录。
* 写入权限（w）：写入权限允许用户修改文件的内容或目录的结构。对文件而言，具有写入权限的用户可以编辑或修改文件的内容；对目录而言，具有写入权限的用户可以创建、删除或重命名文件和子目录。
* 执行权限（x）：执行权限允许用户执行可执行文件或进入目录。对文件而言，具有执行权限的用户可以运行该文件作为程序；对目录而言，具有执行权限的用户可以进入该目录并访问其中的文件和子目录。
* 设置权限：可以使用 chmod 命令来设置文件和目录的权限。例如，使用以下命令将文件设置为所有者可读写，所属组和其他用户只可读取的权限：
* 特殊权限：除了基本权限位，还存在一些特殊权限位，如 SUID、SGID 和 sticky 位。SUID 权限允许在执行文件时临时获得文件所有者的权限，SGID 权限允许在执行文件时临时获得文件所属组的权限，sticky 位用于目录，防止其他用户删除其他人的文件。

可以使用 ls -l 命令来查看文件和目录的详细权限信息。列表中的第一列显示了权限位的设置情况，如 rwxr-xr--。



按照字母位置解释：



权限分段，每一段代表不同的用户：

* 属主权限：当前文件所属用户的权限，在Linux中每一个文件都有所属的用户
* 属组权限：当前文件的用户所在组的其它成员的权限其它
* 用户权限：跟文件所属用户不在同组的其它用户的权限



#### sudo 权限

sudo 是一个独立的程序，需要安装和配置才能使用。通常情况下，sudo 配置文件位于 /etc/sudoers 或 /etc/sudoers.d 目录下，可使用 visudo 命令编辑该文件。

sudo 可以根据需要为不同的用户或组分配不同的授权和限制。通常情况下，授权的方式是通过添加条目到 sudoers 文件中实现的。可以使用 ALL 表示所有命令，也可以使用特定的命令或命令列表。可以限制用户仅能在特定的终端或特定的时间段内执行 sudo 命令。

一旦配置了 sudo，普通用户可以使用 sudo 命令以管理员身份执行需要的命令。需要在命令前加上 sudo 关键字，按提示输入账户密码即可。如果用户未被授权执行该命令，则会收到错误提示。

由于 sudo 具有很高的权限，因此必须谨慎使用。可以对关键命令进行审计，查看用户是否滥用了权限。可以使用特定的日志工具监控 sudo 命令的执行记录。此外，要定期更新系统和软件以确保系统安全。

超级管理员是一个特殊的用户，具有系统的最高权限。在 Ubuntu 中，默认情况下没有启用 root 用户或超级管理员账户，而是使用 sudo 权限管理来代替直接使用 root 权限。因此，Ubuntu 用户通常需要使用 sudo 命令来执行管理员任务。

#### sudo 与 su

su是最简单的身份切换，用su我们能够进行各种用户的切换，一般都是su - username，然后输入password (注意：这里的密码不会在屏幕中显示出来)，root 用 su 切换到其它身份的时候无需输入 password 。

在 Linux 中，su 和 sudo 是用于切换用户身份的命令，但它们在功能和使用方式上有一些区别：

**su（Switch User）命令：**

功能：su 命令用于切换到其他用户账号，并且可以切换为超级用户（root）。

使用方式：默认情况下，使用 su 命令需要输入目标用户的密码。例如，使用 su username 可以切换到名为 "username" 的用户账号，或使用 su 切换为超级用户。

特点：切换为目标用户后，会进入该用户的环境，包括工作目录、环境变量等。

**sudo（Superuser Do）命令：**

功能：sudo 命令允许普通用户以其他用户（通常是超级用户）的权限来执行命令或操作。

使用方式：使用 sudo 命令时，需要输入当前用户的密码，而不是目标用户的密码。例如，使用 sudo command 可以以超级用户权限执行指定的命令。

特点：通过 sudo 执行的命令会在日志中留下记录，以便审计和追踪。

**区别总结如下：**

su 是切换用户的命令，需要知道目标用户的密码，且切换后会进入该用户的环境。

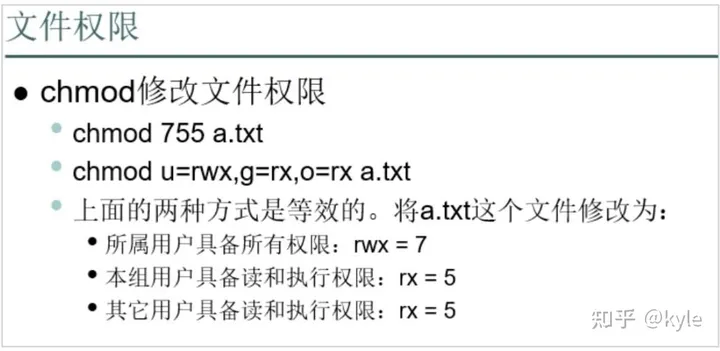
sudo 是以其他用户权限执行命令的命令，需要输入当前用户的密码，具有审计和追踪功能。

一般来说，sudo 命令更为常用和推荐，因为它提供了更细粒度的权限控制，可以指定哪些用户可以执行特定的命令，从而提高系统的安全性。在使用 sudo 命令时，管理员可以将需要临时获取超级用户权限的命令添加到 /etc/sudoers 配置文件中，并配置相应的访问权限。

补充：  
ACL（Access Control List）是一种在Linux系统中用于细粒度控制文件和目录权限的机制。传统的Unix文件权限模型使用9位权限标志（rwxrwxrwx）来控制所有者、所属组和其他用户对文件的访问权限。  
而ACL提供了更加灵活和精细的权限控制，它允许你为特定用户或用户组附加额外的权限。ACL可以应用于文件和目录，并且能够设置不同类型的权限，例如读取、写入、执行以及其他自定义的权限。  
使用ACL，你可以做到以下几点：  
给一个文件或目录授予额外的用户或用户组访问权限，而不仅限于所有者、所属组和其他用户。  
通过ACL为特定用户或用户组添加个别权限，而不是应用通用的文件权限。  
控制默认权限，以便新创建的文件或目录继承父级目录的ACL，而不仅仅是文件权限。  
要在Linux系统中使用ACL，你需要确保文件系统支持ACL并已启用。多数Linux发行版默认开启了ACL功能。可以使用acl命令检查文件系统是否支持ACL，使用getfacl和setfacl命令来查看和修改ACL规则。  
总而言之，ACL提供了更加灵活和细粒度的权限控制机制，使管理员能够更好地管理用户对文件和目录的访问权限。

#### 权限管理

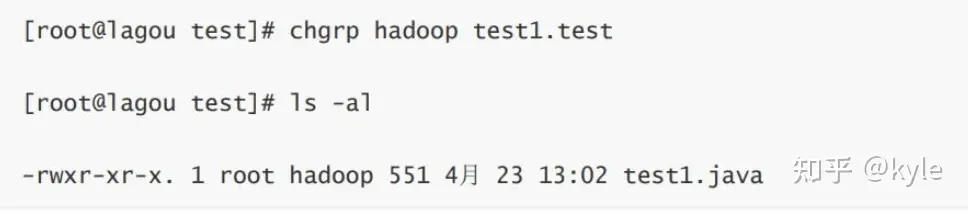
**修改文件权限**:



**chown 改变所有者**

基本语法：chown 最终用户（功能描述：改变文件或者目录的所有者）

案例：[root@lagoutest]#chown hadoop test1.txt

**chgrp改变所属组**  
基本语法：chgrp 最终用户组（功能描述：改变文件或者目录的所属组）  
案例：  


#### 用户登录管理

用户登录选项的配置通常在 /etc/login.defs 或 /etc/default/useradd 文件中。可以设置默认的 shell、UID 和 GID 的范围、密码有效期等。可以根据需要进行调整。

使用 usermod 命令可以禁止或允许用户登录。可以通过修改用户账户的 shell 为 /sbin/nologin 来禁止用户登录。例如，使用以下命令禁止用户登录：

sudo usermod -s /sbin/nologin 用户名

可以使用文件权限和访问控制列表 (ACL) 来限制用户对特定文件和目录的访问权限。可以设置文件的所有者和所属组，并设置合适的访问权限。

使用 id、who、w 等命令可以查看用户账户的信息，如用户名、用户 ID、所属用户组、登录状态等。

#### 文件

Linux 内核是要对外提供服务的，而传递服务的对象就是文件。Linux 中有一切皆文件的说法，文件是 Linux 内核向我们展示信息的基本单元，组织文件的系统为文件系统树。使用 Linux 的过程就是我们在 Linux 文件系统树中找到我们需要的文件，然后或者操作，或者读取它，从而达到我们的目的或者获取我们需要的信息。

文件是信息的载体，在 Linux 中，文件存着三种型态：文件本体，软链接和硬链接。

# 创建一个 test.txt 空文件  
root@M5Core135:~# touch test.txt  
# 创建这个文件的硬链接  
root@M5Core135:~# ln test.txt ln\_test.txt  
# 创建这个文件的软链接  
root@M5Core135:~# ln -s test.txt lns\_test.txt  
# 显示创建的文件  
root@M5Core135:~# ls -la  
total 8  
drwxr-xr-x 2 root root 4096 Jul 20 14:22 .  
drwx------ 13 root root 4096 Jul 20 14:22 ..  
lrwxrwxrwx 1 root root 8 Jul 20 14:22 lns\_test.txt -> test.txt  
-rw-r--r-- 2 root root 0 Jul 20 14:22 ln\_test.txt  
-rw-r--r-- 2 root root 0 Jul 20 14:22 test.txt  
# 显示创建的文件描述  
root@M5Core135:~# file test.txt   
test.txt: empty  
root@M5Core135:~# file ln\_test.txt   
ln\_test.txt: empty  
root@M5Core135:~# file lns\_test.txt   
lns\_test.txt: symbolic link to test.txt

文件本体就是标准的文件，有内容，有权限，有用户属性等等，对于软链接和硬链接，相当于文件不同形式的分身。  
硬链接是指多个文件实际上指向同一个索引节点（inode）。每个文件都有一个 inode，inode 包含文件的元数据和数据块的地址。所有硬链接文件对应的 inode 的链接计数都会加一。硬链接文件之间没有物理上的区别，它们只是同一 inode 的多个别名。当一个硬链接文件被删除时，只是链接计数减一，只有在链接计数降为零时，文件的内容才会被释放。硬链接不能跨越文件系统边界，因为 inode 是特定于文件系统的。硬链接相当于文件本体的一个别名，同时也是文件的动态备份，无论操作文件本体或者硬链接都相当于操作该文件。只有全部文件本体和硬链接全部删除时才会真正的删除文件，如果只删除文件本体，那硬链接就会变成文件本体。  
软链接是一个特殊的文件，其中包含指向另一个文件的路径。软链接是通过文件路径而不是 inode 直接引用目标文件的。软链接可以链接到不存在的文件，也可以跨越文件系统。软链接文件有自己的 inode，而其内容是指向目标文件的路径。如果原始文件被删除，软链接将变为坏链接（dangling link）。软链接也可能引起“循环链接”（circular link）问题，即链接形成了一个环，导致无限循环。软链接只是相当于文件的一个分身，没有备份功能。

#### 文件系统树

组织文件的系统被称为文件系统树，基本的 Linux 文件系统树由根节点和多个根目录组成，不同的根目录存放着不同的文件，他们决定着 Linux 不同的功能。



**/root**

这是系统管理员(root user)的目录。对于系统来说，系统管理员就好比是上帝，它能对系统做任何事情，甚至包括删除你的文件。因此，请小心使用root帐号。  
**/bin**

这里存放了标准的 Linux 的工具，比如像 " ls "、" vi " 还有 " more " 等等。通常来说，这个目录已经包含在你的 " PATH " 系统变量里面了。什么意思呢？就是当你在终端里输入 ls ，系统就会去 PATH 环境变量的目录下面查找是不是有 ls 这个程序。系统所需要的那些命令位于此目录  
**/etc**

这里主要存放了系统配置方面的文件。举个例子：你安装了 samba 这个套件，当你想要修改 samba 配置文件的时候，你会发现配置文件就在 /etc/samba 目录下。  
**/dev**

这里主要存放与设备有关的文件( Unix 和 Linux 系统均把设备当成文件)。想连线打印机吗? 系统就是从这个目录开始工作的。另外还有一些包括磁盘设备文件、USB 设备文件等都放在这个目录。  
**/home**

这里主要存放普通用户的个人数据。具体每个用户的设置文件，用户的桌面文件夹，还有用户的数据都放在这里。每个用户都有自己的用户目录，位置为：/home/用户名。当然，root用户除外。  
**/tmp**

这是临时目录，只存在于内存中，关机便会消失。对于某些程序来说，有些文件被用了一次两次之后，就不会再被用到，像这样的文件就放在这里。有些 Linux 系统会定期自动对这个目录进行清理，因此，千万不要把重要的数据放在这里。  
**/usr**

在这个目录下，你可以找到那些不适合放在 /bin 或 /etc 目录下的额外的工具。比如像游戏，一些打印工具拉等等。/usr 目录包含了许多子目录 /usr/bin 目录用于存放程序，/usr/share 用于存放一些共享的数据，比如音乐文件或者图标等等，/usr/lib 目录用于存放那些不能直接运行的，但却是许多程序运行所必需的一些函数库文件。你的软件包管理器会自动帮你管理好 /usr 目录的。

当我们安装一个 Linux 发行版官方提供的软件包时，大多安装在这里。 如果有涉及服务器配置文件的，会把配置文件安装在 /etc 目录中。/usr 目录下包括涉及字体目录 /usr/share/fonts ，帮助目录 /usr/share/man 或 /usr/share/doc，普通用户可执行文件目录 /usr/bin 或 /usr/local/bin 或 /usr/X11R6/bin ，超级权限用户 root 的可执行命令存放目录，比如 /usr/sbin 或/usr/X11R6/sbin 或/usr/local/sbin 等，还有程序的头文件存放目录 /usr/include。  
**/usr/local**

这里主要存放那些手动安装的软件，即不是通过 apt-get 安装的软件。它和 /usr 目录具有相类似的目录结构。让软件包管理器来管理 /usr 目录，而把自定义的脚本 scripts 放到 /usr/local 目录下面

/usr/local 这个目录一般是用来存放用户自编译安装软件的存放目录；一般是通过源码包安装的软件，如果没有特别指定安装目录的话，一般是安装在这个目录中。  
**/opt**

这里主要存放那些可选的程序。你想尝试最新的 firefox 测试版吗?那就装到 /opt 目录下吧，这样，当你尝试完，想删掉 firefox 的时候，你就可以直接删除它，而不影响系统其他任何设置。安装到 /opt 目录下的程序，它所有的数据、库文件等等都是放在同个目录下面。  
**/media**

有些 Linux 的发行版使用这个目录来挂载那些可移动的磁盘。  
**/boot**

Linux 的内核及引导系统程序所需要的文件目录，比如 vmlinuz initrd.img 文件都位于这个目录中。在一般情况下，GRUB 或 LILO 系统引导管理器也位于这个目录。  
**/lost+found**

在ext2或ext3文件系统中，当系统意外崩溃或机器意外关机，而产生一些文件碎片放在这里。当系统启动的过程中 fsck 工具会检查这里，并修复已经损坏的文件系统。 有时系统发生问题，有很多的文件被移到这个目录中，可能会用手工的方式来修复，或移到文件到原来的位置上。  
**/mnt**

这个目录一般是用于存放挂载储存设备的挂载目录的，比如有 cdrom 等目录。可以参看 /etc/fstab 的定义。有时我们可以把让系统开机自动挂载文件系统，把挂载点放在这里也是可以的。主要看 /etc/fstab 中怎 么定义了。比如光驱可以挂载到 /mnt/cdrom 。  
**/proc**

操作系统运行时，进程信息及内核信息（比如cpu、硬盘分区、内存信息等）存放在这里。/proc 目录伪装的文件系统 proc 的挂载目录，proc 并不是真正的文件系统，它的定义可以参见 /etc/fstab 。  
**/sbin**  
大多是涉及系统管理的命令的存放，是超级权限用户 root 的可执行命令存放地，普通用户无权限执行这个目录下的命令，这个目录和 /usr/sbin; /usr/X11R6/sbin或 /usr/local/sbin 目录是相似的；我们记住就行了，凡是目录 /sbin 中包含的都是需要 root 权限才能执行的。  
**/var**  
这个目录的内容是经常变动的，看名字就知道，我们可以理解为 vary 的缩写，/var 下有 /var/log 这是用来存放系统日志的目录。/var/www 目录是定义 Apache 服务器站点存放目录；/var/lib 用来存放一些库文件，比如 MySQL 的，以及 MySQL 数据库的的存放地；  
/var/log 系统日志存放，分析日志要看这个目录的东西；  
/var/spool 打印机、邮件、代理服务器等假脱机目录；

Linux 或者说 UNIX 类的现代操作系统自身只有用户、权限、文件这三个内容，在使用 Linux 的过程中，我们基本都是围绕着这三个内容展开的。同时，这三个内容的组织形式，用户组，权限组，文件树也是非常重要的。

### （2）、 Linux 中的命令（软件）

Linux 中的命令其实就是一个个的可执行程序，脚本也算可执行程序的一种。可执行程序又叫软件，软件的组织形式是软件包，有专门的程序去管理软件包。

Linus Torvalds 开始编写和编译 Linux 内核时，他需要一套非常基本的系统实用程序来开始执行一些稍微有用的工作。内核并不提供命令或任何类型的命令 shell，比如 Bash，它本身是没有任何用处的，因此，Linus 使用了免费提供的 GNU 核心实用程序，并为 Linux 重新编译了它们。这让他拥有了一个完整的、即便是相当基本的操作系统。

#### SHELL



对于计算机来说，Linux 内核只是一个管理计算机各种硬件的程序，它不具有和人们交互的能力。为此我们需要一个能够理解人类指令并让内核按照人类意愿管理和操作计算机的程序，这个程序就是 Shell。Shell 的变种有多个名称，下面是关于 Shell 程序信息的一个表格：

|  |  |  |  |
| --- | --- | --- | --- |
| shell | Complete path-name | Prompt for root user | Prompt for non root user |
| Bourne shell (sh) | /bin/sh and /sbin/sh | # | $ |
| GNU Bourne-Again shell (bash) | /bin/bash | bash-VersionNumber# | bash-VersionNumber$ |
| C shell (csh) | /bin/csh | # | % |
| Korn shell (ksh) | /bin/ksh | # | $ |
| Z Shell (zsh) | /bin/zsh | # | % |

**Bourne Shell (sh)**  
Bourne shell 由 Steve Bourne 在 AT&T 贝尔实验室开发，被认为是第一个UNIX shell。它被表示为 sh。由于紧凑和操作速度快，Bourne shell 广受欢迎。

以上特性使得它成为 Solaris 操作系统的默认外壳。它也被用作所有 Solaris 系统管理脚本的默认外壳。然而，Bourne shell有一些明显的缺点：

它没有内置的处理逻辑和算术运算的功能。

另外，与Linux中大多数的 shell 不同，Bourne shell不能回忆以前使用过的命令。

它功能不全面， 无法提供适当的交互式使用。

Bourne shell的完整路径名称是/bin/sh和/sbin/sh。默认情况下，它对 root 用户使用 *#* 提示符，对非 root 用户使用 *$* 提示符。

**GNU Bourne-Again Shell (bash)**

GNU Bourne-Again shell 更多被称为 Bash shell，它被设计成与Bourne shell兼容。Bash shell 融合了 Linux 中不同类型 shell 的有用功能，如 Korn shell 和 C shell。

与Bourne shell不同的是，它能够自动回忆以前使用过的命令，并可以使用方向键编辑它们。

GNU Bourne-Again shell的完整路径名称是/bin/bash。默认情况下，它对 root 用户使用的提示符是 *bash-版本号#*，对非 root 用户的提示符是 *bash-版本号$*。

**C Shell (csh)**  
发 C shell 是把有用的编程功能容纳进去，如对算术运算的内置支持和类似于 C 语言的语法。

此外，它还包含了 Linux 中不同类型的 shell（如Bourne shell）中所缺少的历史命令。C shell 的另一个显著功能是“别名”。

C shell的完整路径名称是/bin/csh。默认情况下，它对 root 用户的提示符是 *主机名#*，对非 root 用户的提示符是 *主机名$*。

**Korn Shell (ksh)**  
Korn shell 是由 David Korn 在 AT&T 贝尔实验室开发的，用于改进 Bourne shell。它被表示为 ksh。Korn shell 本质上是 Bourne shell 的一个超集。

除了支持 Bourne shell 所支持的一切外，Korn shell 还为用户提供了新的功能。它有着内置的对算术运算的支持，同时提供类似于 C shell 的交互功能。

Korn shell 运行为 Bourne shell 编写的脚本，同时提供类似于 C 语言的字符串、数组和函数运算。它还支持为 C shell 编写的脚本。此外，它比 Linux 中大多数不同类型的 shell（包括 C shell）要快。

Korn shell 的完整路径名称是/bin/ksh。默认情况下，它对 root 用户使用 *#* 提示符，对非 root 用户使用 *$* 提示符。

**Z Shell (zsh)**  
Z Shell 或 zsh 是 sh shell 的扩展，在自定义方面做了大量改进。如果你想要一个具有更多功能的现代 shell，zsh shell就是你要找的。

Z shell 有以下亮点：

* 根据给定的条件生成文件名
* 支持插件和主题
* 索引内置函数
* 自动补全命令
* 以及更多……

Shell 程序更多的是提供一个和用户交互的接口，它能够以用户手动输入的方式一条一条命令执行，同时也可以解析已经编写好的程序脚本，自动进行命令执行。Shell 程序本身不提供或者只提供基本的内置环境命令，而我们平时执行的命令主要是由软件包中的软件来提供的，所以学习命令的使用其实就是学习软件包中的软件使用方法。

#### Coreutils

Coreutils 软件包是 GNU 项目的一部分，它是一组基本的命令行工具集，用于操作和管理Linux和其他类Unix系统中的文件和数据流。这些工具被广泛用于终端和脚本中，提供了许多常用的功能和实用程序，基本的 Linux 命令就是由这个软件包进行提供了。Coreutils 的官网地址是：<https://www.gnu.org/software/coreutils> 。

**GNU coreutils核心程序**  


Linux 的基本系统操作命令有十大类：

* 目录处理命令：ls，mkdir，cd，pwd，rmdir，cp，rm
* 文件处理命令：touch，cat，more，less，head，tail
* 链接命令：ln
* 权限管理命令：chmod，chown，chgrp，umask
* 文件搜索命令：find，locate，which，whereis，grep
* 帮助命令；man，whatis，help
* 用户管理命令：useradd，passwd，who，w，su
* 压缩解压缩命令：gzip，gunzip，tar，zip，unizp
* 网络命令：write，wall，ping，ifconfig，mail，last，netstat
* 关机重启命令：shutdown，logout

#### util-linux

Util-linux 软件包包含 Coreutils 没有的工具。其中比较重要的是加载、卸载、格式化、分区和管理硬盘驱动器，打开 tty 端口和得到内核消息。

Util-linux 包含下面程序：

agetty, arch, blockdev, cal, cfdisk, chkdupexe, col, colcrt, colrm, column, ctrlaltdel, cytune, ddate, dmesg, elvtune, fdformat, fdisk, fsck.cramfs, fsck.minix, getopt, hexdump, hwclock, ipcrm, ipcs, isosize, line, logger, look, losetup, mcookie, mkfs, mkfs.bfs, mkfs.cramfs, mkfs.minix, mkswap, more, mount, namei, parse.bash, parse.tcsh, pg, pivot\_root, ramsize (link to rdev), raw, rdev, readprofile, rename, renice, rev, rootflags (link to rdev), script, setfdprm, setsid, setterm, sfdisk, swapoff (link to swapon), swapon, test.bash, test.tcsh, tunelp, ul, umount, vidmode (link to rdev), whereis 和 write

简短说明：

* agetty：打开一个TTY端口，显示登录提示，执行 /bin/login 命令。
* arch：显示机器的体系结构信息。
* blockdev：在命令行中调用块设备的ioctl。
* cal：显示一个简单的日历。
* cfdisk：处理指定设备的分区表。
* chkdupexe：找出多余的可执行文件。
* col：过滤来自输入的反向行馈送内容。
* colcrt：预览 nroff 的输出。
* colrm：在某个文件中删除列。
* column：把输出格式化为几列。
* ctrlaltdel：设置CTRL+ALT+DEL组合键的功能，为硬重启或软重启。
* cytune：查询和修改cyclade驱动器的中断入口。
* ddate：把阳历日期转换为Discordian日期。
* dmesg：显示内核的启动信息。
* elvtune：调整I/O端口平衡。
* fdformat：低级格式化一张软盘。
* fdisk：磁盘分区管理程序。
* fsck.cramfs：对Cramfs文件系统的一致性进行检查。
* fsck.minix：对MINIX文件系统的一致性进行检查。
* getopt：解析命令参数。
* hexdump：用用户指定的方式(包括ASCII, 十进制, 十六进制, 八进制)显示一个文件或者标准输入的数据。
* hwclock：查询和设置硬件时钟。(也被称为RTC或BIOS时钟)
* ipcrm：删除一个指定的资源。
* ipcs：提供IPC机制的信息。
* isosize：输出iso9660文件系统的长度。
* line：单行拷贝。
* logger：设置系统日志的入口。
* look：显示以某个给定字符串为开头的行。
* losetup：启动和控制回环(loop)设备。
* mcookie：为xauth产生magic cookie，128位随机数。
* mkfs：在一个设备(通常是一个硬盘分区)设备上建立文件系统。
* mkfs.bfs：创建 SCO bfs 文件系统。
* mkfs.cramfs：创建cramfs文件系统。
* mkfs.minix：创建 Minix 文件系统。
* mkswap：初始化指定设备或文件，以用做交换分区。
* more：分屏显示文件，但less更好用。
* mount：把一个文件系统从一个设备挂载到一个目录。
* namei：显示指定路径的符号链接。
* pg：显示文本文件内容，一次显示一屏。
* pivot\_root：使某个文件系统成为当前进程的根文件系统。
* ramsize：显示或者改变内存虚拟盘的大小。
* rdev：查询和设置内核的根设备和其他信息。
* readprofile：显示/proc/profile文件的信息。
* rename：对文件重命名。
* renice：修改正在运行进程的优先级。
* rev：颠倒一个文件每行字符的顺序。
* rootflags：在挂载根设备时，查询和设置额外的信息。
* script：为终端会话过程建立一个typescipt文件，记录会话过程中终端的输出。
* setfdprm：设置软盘参数。
* setsid：在一个新的会话过程中运行程序。
* setterm：设置终端属性。
* sfdisk：磁盘分区表管理工具。
* swapdev：能在启动内核中设置交换分区设备。
* swapoff：取消对指定交换设备和交换文件的使用。
* swapon：使指定的交换设备和交换文件生效。
* tunelp：设置打印设备的参数。
* ul：加下划线。
* umount：卸载一个被挂载的文件系统。
* vidmode：查询和设置视频模式。
* whereis：确定某命令二进制文件,源文件和手册文档的位置。
* write：发一个消息给另一个用户,如果他开启了writting的话。

Util-linux 依赖于: Bash, Binutils, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Ncurses, Sed, Zlib 软件包。

#### 软件包管理

Linux 只是提供一个高度可用的内核，并没有规定上层的软件包组织形式，理论上说只要是符合要求的可执行文件都能得到执行。Linux 中的软件包非常自由有不同的组织形式，这些软件包又组成了不同的发行版。而不同的发行版又使用不同的软件包管理器。总的来说，软件包的管理是一个非常杂乱的时事情。

程序有动态执行库 SO 的依赖，把 SO 和 可执行文件组织起来的软件包也同样有不同的依赖。管理这些依赖是一件非常烦杂的事情，此时各种依赖管理工具诞生了，他们是各类发行版的基石，正是有了软件包依赖管理器，我们安装软件才能从容不迫。

**dpkg**

**Ubuntu、Debian**

dpkg 命令是 Debian Linux 系统用来安装、创建和管理软件包的实用工具。

* **命令行使用**

root@M5Core135:~# dpkg(选项)(参数)  
root@M5Core135:~# dpkg --help  
Usage: dpkg [<option> ...] <command>  
  
# 选项  
-i：安装软件包  
-r：删除软件包  
-P：删除软件包的同时删除其配置文件  
-L：显示于软件包关联的文件  
-l：显示已安装软件包列表  
--unpack：解开软件包  
-c：显示软件包内文件列表  
--confiugre：配置软件包  
  
# 参数  
Deb软件包：指定要操作的.deb软件包

* **示例演示说明**

# 安装包  
root@M5Core135:~# dpkg -i package.deb  
  
# 删除包  
root@M5Core135:~# dpkg -r package  
  
# 删除包（包括配置文件）  
root@M5Core135:~# dpkg -P package  
  
# 列出与该包关联的文件  
root@M5Core135:~# dpkg -L package  
  
# 显示该包的版本  
root@M5Core135:~# dpkg -l package  
  
# 解开deb包的内容  
root@M5Core135:~# dpkg --unpack package.deb  
  
# 搜索所属的包内容  
root@M5Core135:~# dpkg -S keyword  
  
# 列出当前已安装的包  
root@M5Core135:~# dpkg -l  
  
# 列出deb包的内容  
root@M5Core135:~# dpkg -c package.deb  
  
# 配置包  
root@M5Core135:~# dpkg --configure package  
  
# 列出已安装软件包  
root@M5Core135:~# sudo dpkg-query -l  
root@M5Core135:~# sudo dpkg-query -l | less  
root@M5Core135:~# sudo dpkg-query -l | grep tmux

**apt**

**Ubuntu、Debian**

apt-get 命令是 Debian Linux 发行版中的 APT 软件包管理工具，它是 dpkg 的前端工具。所有基于 Debian 的发行都默认使用这个包管理系统。deb 包可以把一个应用的文件包在一起，大体就如同 Windows 上的安装文件。

* **命令行使用**

# apt-get(选项)(参数)  
root@M5Core135:~# apt --help  
Usage: apt [options] command  
  
# 选项  
-c：指定配置文件  
  
# 参数  
管理指令：对APT软件包的管理操作  
软件包：指定要操纵的软件包

* **示例演示说明**

# 更新所有已安装的软件包  
root@M5Core135:~# apt-get upgrade  
  
# 将系统升级到新版本  
root@M5Core135:~# apt-get dist-upgrade  
  
# 更新  
root@M5Core135:~# apt-get update  
  
# 安装一个新软件包  
root@M5Core135:~# apt-get install packagename  
  
# 卸载一个已安装的软件包（保留配置文件）  
root@M5Core135:~# apt-get remove packagename  
  
# 卸载一个已安装的软件包（删除配置文件）  
root@M5Core135:~# apt-get –purge remove packagename  
  
# 来删除你已经删掉的软件  
root@M5Core135:~# apt-get autoclean apt  
  
# 会把安装的软件的备份也删除  
root@M5Core135:~# apt-get clean  
  
# 列出已安装软件包  
root@M5Core135:~# apt list --installed  
root@M5Core135:~# apt list --installed | less  
root@M5Core135:~# apt list --installed | grep tmux

### （3）、Shell 脚本

Shell 脚本就是将需要运行的程序命令用一定的流程控制组织起来，整合到一个文件中，这个文件被称为 Shell 脚本。

因为 Shell 脚本本身包含的就是 Shell 能够执行的命令语句，所以 Shell 脚本不用编译就能由 Shell 直接解释执行。当然解释执行的属性也带来了运行速度慢的缺点，但这个缺点在脚本语言中一般不会去考虑。

#### 简单的 Shell 脚本

我们以 bash 脚本为例子。

Shell 脚本默认是以 .sh 结尾，所以我们首先创建一个 test.sh 的文件。

root@M5Core135:~# touch test.sh  
root@M5Core135:~# ls  
test.sh

然后在该文件中添加下面的内容：

root@M5Core135:~/tmp# cat test.sh   
#!/bin/bash  
echo "hello world!"

其中的 #!/bin/bash 意思是执行这个脚本的程序是 /bin/bash。

echo "hello world!" 是执行一个输出 "hello world!" 的命令。

给该脚本赋予可执行的权限：

root@M5Core135:~# chmod +x test.sh   
root@M5Core135:~# ls test.sh -l  
-rwxr-xr-x 1 root root 32 7月20日 15:08 test.sh

执行脚本：

root@M5Core135:~/tmp# ./test.sh   
hello world!

执行脚本后，输出了 "hello world!" 。

#### Shell 脚本中的变量

变量是可变的参数量，用于表示在脚本程序运行过程中需要代指变化的参数。

Shell 中存在三种类型的变量：

* 局部变量：在当前脚本中起作用的变量。
* 环境变量：Shell 环境中的变量，在 Shell 运行后所有程序都能访问到的变量。
* Shell 变量：这是 Shell 为程序运行时传递的特殊变量，主要是用户输入参数等等。

**操作变量：**

* 创建普通变量：test="hello world!"。
* 创建局部变量：local local\_test="hello world!"。
* 创建局部变量：test="hello world!" ;readonly test
* 创建环境变量：export test\_all="hello world!"
* 使用变量：echo $test 或者 echo ${test}
* 删除变量：unset test

**字符串操作：**

* 字符串拼接：直接将两个字符串写在一起即可。str="hello"" ""world!"、str0="hello";str1=" ";str2="world!";str=$str0$str1$str2
* 字符串截取：通过字符串命令进行截取，比如cut、sed、awk等。也可以通过 Shell 的字符串处理, ${varible:n1:n2}截取字符串变量 varible 中从 n1 位置开始，长度为 n2 的子字符串。
* 获取字符串长度：通过字符串命令，echo $str | awk '{print length($0)}'，echo $str | wc -L。通过 Shell 方式echo ${#str}。

**数组操作：**

* 定义一个数组：

#!/bin/bash  
test=("hello" " " "world" "!")

* 操作数组:

#!/bin/bash  
test=("hello" " " "world" "!")  
echo ${test[0]} # 访问单个数组  
echo ${test[\*]} # 打印整个数组  
test[0]="test" # 对数组 0 号元素重新赋值  
echo ${test[\*]}  
unset test[2] # 删除数组 2 号元素  
echo ${test[\*]}

* 遍历数组：

#!/bin/bash  
a=(1 2 3 4 5 6)  
for((i=0; i<10; i++))  
do  
 echo "a[$i]=${a[$i]}"  
done

* 获取数组长度:

#!/bin/bash  
a=(1 2 3 4 5 6)  
echo ${a[\*]}  
echo "a len: ${#a[\*]}"

* 合并数组：

#!/bin/bash  
a=("hello" " " "world" "!")  
b=("hello" "zhaixue.cc")  
c=(${a[\*]} ${b[\*]})  
echo ${c[\*]}

**特殊变量：**

* $0 ：代表执行的文件名
* $1 ：代表传入的第1个参数
* $n ：代表传入的第n个参数
* $# ：参数个数
* $\* ：以一个单字符串显示所有向脚本传递的参数。
* \*相同，但是使用时加引号，并在引号中返回每个参数
* $$ ：脚本运行的当前进程号
* $! ：后台运行的最后一个进程的ID
* $? ：显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误。

**算术运算：**

Shell 使用算术运算需要特殊的方式：

#！/bin/bash  
num1=21  
num2=5  
add\_val=$((num1 + num2))   
sub\_val=$((num1 - num2))   
mul\_val=$((num1 \* num2))   
div\_val=$((num1 / num2))   
res\_val=$((num1 % num2))   
echo "num1 + num2 : $add\_val"  
echo "num1 - num2 : $sub\_val"  
echo "num1 \* num2 : $mul\_val"  
echo "num1 / num2 : $div\_val"  
echo "num1 % num2 : $res\_val"

**逻辑运算：**

#！/bin/bash  
if [ 1 -eq 1 ] ; then  
 echo "1 等于 1"  
fi  
if [ 1 -ne 2 ] ; then  
 echo "1 不等于 2"  
fi  
if [ 2 -gt 1 ] ; then  
 echo "2 大于 1"  
fi  
if [ 1 -lt 2 ] ; then  
 echo "1 小于 2"  
fi  
if [ 2 -ge 1 ] ; then  
 echo "2 大于等于 1"  
fi  
if [ 1 -le 1 ] ; then  
 echo "1 小于等于 1"  
fi  
if [[ 1 -eq 1 && 1 -gt 0 ]] ; then  
 echo "1 等于 1 同时 1 大于 0"  
fi  
if [[ 1 -eq 2 || 2 -gt 1 ]] ; then  
 echo "1 等于 2 或者 2 大于 1"  
fi

**文件测试运算符:**

#！/bin/bash  
if [ -b "/dev/mmcblk0" ] ; then  
 echo "文件是块设备文件"  
fi  
if [ -c "/dev/tty1" ] ; then  
 echo "文件是字符设备文件"  
fi  
if [ -d "/home" ] ; then  
 echo "检测文件是目录"  
fi  
if [ -f "/root/123.wav" ] ; then  
 echo "文件是普通文件或者文件存在"  
fi  
if [ -g "/root/123.wav" ] ; then  
 echo "文件设置了 SGID 位"  
fi  
if [ -k "/root/123.wav" ] ; then  
 echo "文件设置了粘着位(Sticky Bit)"  
fi  
mkfifo /root/test\_pipe  
if [ -p "/root/test\_pipe" ] ; then  
 echo "文件是有名管道"  
fi  
rm /root/test\_pipe  
if [ -u "/root/123.wav" ] ; then  
 echo "文件设置了 SUID 位"  
fi  
if [ -r "/root/123.wav" ] ; then  
 echo "文件可读"  
fi  
if [ -w "/root/123.wav" ] ; then  
 echo "文件可写"  
fi  
if [ -x "/root/123.wav" ] ; then  
 echo "文件可执行"  
fi  
if [ -s "/root/123.wav" ] ; then  
 echo "文件为空"  
fi  
if [ -e "/root/123.wav" ] ; then  
 echo "文件（包括目录）存在"  
fi

**流程控制:**

#！/bin/bash  
  
# if else 判断  
if true ; then  
 echo "条件是 true"  
else  
 echo "条件是 false"  
fi  
  
# for循环  
for loop in 1 2 3 4 5  
do  
 echo "The value is: $loop"  
done  
  
# while循环  
int=1  
while(( $int<=5 ))  
do  
 echo $int  
 let "int++"  
done  
  
# 多分支结构  
color="red"  
case $color in  
 "red")  
 echo "You chose red."  
 ;;  
 "green")  
 echo "You chose green."  
 ;;  
 "blue")  
 echo "You chose blue."  
 ;;  
 \*)  
 echo "Invalid color."  
 ;;  
esac  
  
# 单选结构  
echo "What is your favourite?"  
select var in "apple" "orange" "banana"; do  
 break;  
done  
echo "You have selected $var"

**子函数:**

#!/bin/bash  
testParam(){  
 echo "第一个参数为 $1 !"  
 shift  
 echo "剩下的参数为 $@ !"  
}  
testParam 1 2 3 4

IO 流的重定向:

每一个 Linux 程序都会需要输入输出，io 重定向就是将 Linux 程序的输入输出重新定位到文件。

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

1、标准输入文件 (stdin)：stdin的文件描述符为 0，Unix 程序默认从 stdin 读取数据。  
2、标准输出文件 (stdout)：stdout 的文件描述符为 1，Unix 程序默认向 stdout 输出数据。  
3、标准错误文件 (stderr)：stderr 的文件描述符为 2，Unix 程序会向 stderr 流中写入错误信息。

#!/bin/bash  
  
# 输入重定向  
command1 < file  
  
# 输出重定向  
command1 > file  
  
# 标准错误文件重定向  
command 2 > file  
  
# 完全输出文件重定向  
command > file 2>&1

/dev/null 这是一个特殊的设备文件，名字为“文件黑洞“，它只能写入文件，不能读出文件。常用于忽略程序的所有输出。

### （3）、Linux 实用

**定时任务**

Crontab 是 Linux 和 Unix 中用于定时任务非常实用的工具。它相当于 Linux 系统中的闹钟，能够按照设定的时间执行预定的程序。

Crontab 运行时是由一个名叫 Cron 的守护程序来定时读取时刻表中的条目，并与现在时刻进行比较，当时间符合后就调用时刻表中需要执行的命令。

Crontab 的基本命令有四个：

* crontab -l：列出当前用户的所有Crontab任务
* crontab -e：编辑当前用户的Crontab任务
* crontab -r：删除当前用户的所有Crontab任务
* crontab -i：在删除 Crontab 任务前提醒用户确认

一个 crontab 时刻条目如下：

0 5 \* \* 1 tar -zcf /var/backups/home.tgz /home/  
minute hour day month week command

编写规则如下：

星号（\*）：代表所有可能的值，例如month字段如果是星号，则表示在满足其它字段的制约条件后每月都执行该命令操作。

逗号（,）：可以用逗号隔开的值指定一个列表范围，例如，“1,2,5,7,8,9”。

中杠（-）：可以用整数之间的中杠表示一个整数范围，例如“2-6”表示“2,3,4,5,6”。

正斜线（/）：可以用正斜线指定时间的间隔频率，例如“0-23/2”表示每两小时执行一次。同时正斜线可以和星号一起使用，例如\*/10，如果用在minute字段，表示每十分钟执行一次。

艾特（@）：可以用 @ 符号表示出现什么状态时运行脚本，比如@reboot：开机时执行；@yearly == 0 0 1 1 \* ：每年执行一次；@monthly == 0 0 1 \* \*:每月执行一次；@weekly == 0 0 \* \* 0：每周执行一次；@daily == 0 0 \* \* \*: 每天执行一次；@hourly == 0 \* \* \* \*：没小时执行一次

每天 18：30执行每日任务：

root@M5Core135:~# crontab -l  
# m h dom mon dow command  
30 18 \* \* \* /home/nihao/.everyday\_task.sh &

**开机任务**

Linux 的开机任务没有一个固定的方式：可以通过 crontab，也可以通过 systemd 服务脚本，在这里我们介绍一个通用的 /etc/rc.load 的方式。

/etc/rc.local 是一个默认约定的当系统启动完成后执行的脚本，大多数系统都会默认这个文件的存在。如果使用 systemd 作为 init 系统没有这个脚本时，我们需要手动添加：

#!/bin/bash  
# 创建 /etc/rc.local   
sudo cat < EOF > /etc/rc.local   
#!/bin/bash  
# you command  
  
exit 0  
EOF  
  
# 添加执行权限  
sudo chmod +x /etc/rc.local  
  
# 创建 systemd 服务脚本  
sudo cat < EOF > /lib/systemd/system/rc-local.service  
[Unit]  
Description=/etc/rc.local Compatibility  
Documentation=man:systemd-rc-local-generator(8)  
ConditionFileIsExecutable=/etc/rc.local  
After=network.target  
  
[Service]  
Type=forking  
ExecStart=/etc/rc.local start  
TimeoutSec=0  
RemainAfterExit=yes  
GuessMainPID=no  
  
[Install]  
WantedBy=multi-user.target  
EOF  
  
# 使能开机服务  
sudo systemctl enable rc-local

**条件任务**

inotify 软件可用于监控 Linux 文件或者目录的变换，这对于当目录发生变化后执行某些动作很有用。

安装：

sudo apt-get install inotify-tools

监视文件或目录的变化:

inotifywait /path/to/directory

监视特定事件类型：

inotifywait -e modify,create,delete /path/to/directory

通过递归监视子目录:

inotifywait -r /path/to/directory

当监视的文件被修改时，执行自定义脚本：

#!/bin/bash  
  
directory="/path/to/directory"  
  
inotifywait -m -e modify --format '%w%f' "$directory" | while read file  
do  
 echo "File $file was modified. Performing custom action."  
 # 在这里添加你要执行的自定义操作  
done

### （4）、Linux 网络配置

Linux 中的网络配置有很多中形式，在这里介绍如何手动方法去配置无线网络。

需要的安装的软件包:net-tools, udhcpc

#!/bin/bash  
sudo apt install net-tools udhcpc

相关命令

#!/bin/bash  
ifconfig # 显示所有网卡和接口信息  
ifconfig -a # 显示所有网卡（包括开机没启动的）信息  
ifconfig eth0 # 指定设备显示信息  
ifconfig eth0 up # 激活网卡  
ifconfig eth0 down # 关闭网卡  
ifconfig eth0 192.168.120.56 # 给网卡配置 IP 地址  
ifconfig eth0 10.0.0.8 netmask 255.255.255.0 up # 配置 IP 并启动  
ifconfig eth0 hw ether 00:aa:bb:cc:dd:ee # 修改 MAC 地址  
dhclient -i eth0 # 获取 dhcp 动态ip

注意：修改网口的 mac 地址时，一定要让网卡接口处于关闭状态 ( ifconfig eth0 down )。

如果网口获取到 ip 地址，能 ping 通外网的 ip 地址但是不能 ping 网址的时，大概率是没有设置系统 dns 服务器。修改过程如下：

echo 'nameserver 8.8.8.8  
nameserver 114.114.114.114' > /etc/resolv.conf

# 第二章：Linux 应用开发

## 1. 操作系统内核

Linux 应用是建立在 Linux操作系统内核上的。从软件的整体运行来看，内核给应用程序提供标准二进制 api 接口，这些接口是内核将底层硬件抽象出来的结果，无论底层硬件如何变换，应用程序都可以通过标准的接口进行工作。这些二进制 api 接口由和操作系统的特性有关，所以想要使用二进制 api 接口，需要了解操作系统特性。

### （1）、 操作系统有四个基本特性

操作系统的基本特征有四个即：并发性、共享性、虚拟性、异步性

**并发性：** 指的是两个或者多个事件在同一时间间隔内发生，即这个设备一会执行这个事件一会执行那个文件，多个事件共同使用一个设备。

操作系统的并发性是指计算机系统中同时存在多个运行着的程序，因此它应该具有处理和调度多个程序同时执行的能力。在这种多道程序环境下，一段时间内，宏观上有多个程序在同时运行，而在每一时刻，单处理器环境下实际仅能有一道程序执行，故微观上这些程序还是在分时地交替执行。操作系统的并发性是通过分时得以实现的。而并行则是指两个或者多个事件在同一时刻发生，即两个或者多个事件互不影响，计算机在同一时间完成两种或两种以上的工作。

**共享性：** 即资源共享，指系统中的资源供内存中的多道程序所共同使用。

共享有两种方式：互斥共享和同时访问。

* 互斥共享方式：系统中的某些资源，虽然可以提供给多个进程使用，但是 一个时间段内，只允许一个进程访问
* 同时共享方式：系统中的某些资源，允许一个时间段内，多个进程"同时"对该资源进行访问
* 这个同时是从宏观上来说的，在微观上，依然是多个进程交替访问该资源。

注意：并发、共享是多用户OS的最基本特征；资源共享以进程并发执行为条件；系统对资源共享的有效管理，需协调诸进程对其访问

并发和共享的关系

* 两个进程正在并发的执行(并发性)
* 需要共享的访问硬盘资源(共享性)

如果失去并发性，则系统只有一个进程在运行，那么共享性就没有意义。

如果失去共享性，则多个应用程序不能同时访问硬盘资源，就无法同时发送文件，即不能并发。

这就是并发性和共享性互为存在条件的原因。

**虚拟性 ：**指通过某种技术把一个物理实体变成若干个逻辑上的对应物。

虚拟性是一种管理技术，把物理上的一个实体变成逻辑上的多个对应物，或把物理上的多个实体变成逻辑上的一个对应物的技术。采用虚拟技术的目的是为用户提供易于使用、方便高效的操作环境。

例如分时系统将一机虚拟为多机就是未来使得用户资源共享更加方便。具体表现为两种虚拟：

* 1）虚拟内存：当前要处理的作业所占的内存比计算机的内存小时，先调入部分作业，当这部分作业处理完之后再调入接下来要处理的部分作业，因而程序认为计算机内存足够大，此即虚拟了更大内存
* 2）虚拟外设：当计算机连接多个外部设备时，让多个外设工作的时间差在可接受范围内，近似于同时工作，此即虚拟外设。

**异步性：** 指的是系统中并发执行的多道程序“走走停停”即随时可能面对中断，以不可预知的速度向前推进。

在多道程序环境下，允许多个程序并发执行，但由于资源有限，进程的执行不是一贯到底；而是走走停停，以不可预知的速度向前推进，也就是进程的执行顺序和执行时间不确定。这就是进程的异步性。

### （2）、 操作系统分类

操作系统会给运行在其上的应用程序分配虚拟化的硬件资源，不同的资源分配方案对应用程序的运行效果影响是巨大的。而人们在不同场景下对计算机运行的要求是不同的，于是便根据资源分配方案划分了操作系统额的分类。

#### 分时操作系统

计算机的硬件承载能力是有限的，在一段时间内，硬件能做的事情也是有限的。应用程序工作的时候需要对硬件有持续的需求，这种需求也包括 cpu 运行资源的使用上。在没有操作系统的时候，采用人工的方法划分 cpu 使用时段，程序会一直运行到结束。有了操作系统之后，操作系统会根据算法自动的为程序划分 cpu 使用时段，操作系统分配资源是十分快速的，这给应用程序划分更小的运行资源提供了可能，也让多个用户同时使用计算机提供了可能。借鉴多道批处理流水线，将运行资源划分到人无法觉察的地步，然后让程序分时段轮流使用运行资源，只要人无法觉察，看起来就像是计算机同时运行了多个应用程序。这种能够自动划分操作系统分时运行资源的操作系统被叫做分时操作系统。

分时操作系统的特点：

* 交互性：用户能与系统实现交互；
* 独立性：多个用户间可以彼此独立运行；

分时系统也有缺点，分时系统让多个用户同时联机共享主机，但也使得系统变得更加复杂，且无法选择具体用户作业的先后顺序。

分时系统的代表是：Unix、Linux、windows 等主要直接为人提供交互服务的系统。

#### 实时操作系统

实时体系指体系的核算正确性不只取决于核算的逻辑正确性，还取决于发生成果的时刻。假如未满意体系的时刻束缚，则以为体系失效。  
——Donal Gillies 在 Realtime Computing FAQ 中提出的界说。

分时系统的缺点无法选择具体用户作业的先后顺序，也就导致了无法保证应用程序对外服务的实时性。为了解决这个问题，研发人员更改了操作系统的运行资源分配方式，添加应用程序的紧急性，让拥有紧急事情的应用程序更快的享受运行资源，也就是更快的执行，从而保证应用程序的实时，这样的操作系统就被叫做实时操作系统。

实时操作系统，当外界事件和数据产生时，系统能以足够快的速度予以处理，其处理结果能在规定的时间内控制生产结果或对系统做出响应，并控制所有实时任务协调一致运行的操作系统。

实时操作系统，一般分为硬实时和软实时，硬实时指的是在规定的时间内必须完成任务操作，在操作系统设计时保证；软实时只要按照任务的优先级，尽可能地完成任务操作即可

硬实时操作系统：完全满足在指定时间内完成关键行为。

硬实时操作系统严格规定了限定时间内必须完成任务，否则会导致严重后果发生。如在自动驾驶场景下，车辆前置摄像头模块，短距长距雷达及助力转向等模块需要连续采集处理数据，并通过算法输出决策行为。其中，系统必须在指定时间内完成对输入的采集数据完成运算及处理，并完成输出。假设方向盘转角，节气门等驾驶相关控制信号若无法确保实时性，自动驾驶系统的安全性将无法保证。

常见的硬实时操作系统有 VxWorks，ThreadX、FreeRTOS、ucOS。

值得提一句的是，VxWorks是世界范围内现役航天，航海，飞机，坦克自动化控制部分使用范围最广的实时操作系统，包括国内网上热度最高的几种军事设备也同使用了此系统。

软实时操作系统：大多数情况下在指定时间内完成关键行为。

软实时系统对处理事件的时间要求是统计学预期，即使在处理过程中偶发出现了时间偏差，也不会对系统造成致命后果。如IP电视解码视频流数据时，可能会出现数据帧的丢失，但即使了，仅是用户视觉上的体验影响，或通过抖动处理，就可以解决图像上的视觉感知问题。

总结就是实时操作系统对操作系统的特性提出了时间上的要求。

* 1. 响应时间：实时操作系统要求任务的响应时间尽可能短，通常在毫秒级别以下。而非实时操作系统对任务响应时间没有严格要求。
  2. 任务调度方式：实时操作系统通常采用抢占式调度方式，即高优先级任务可以强制中断低优先级任务执行。而非实时操作系统通常采用协作式调度方式，即任务执行完后才会主动让出CPU资源。
  3. 可靠性：实时操作系统要求系统具有高可靠性，能够保证任务执行的正确性和稳定性。而非实时操作系统对于系统可靠性要求较低，只要程序不崩溃即可。
  4. 硬件支持：实时操作系统通常需要硬件的支持，如中断控制器、定时器等，以实现对实时任务的快速响应和调度。而非实时操作系统则对硬件支持要求较低。
  5. 功能模块：实时操作系统提供了一系列专门针对实时应用需求而设计的功能模块，如实时任务调度算法、中断处理机制、进程间通信方式等。而非实时操作系统则更加注重通用性，提供了更加丰富的功能模块，如文件系统、网络协议栈等。

非实时操作系统与实时操作系统相比，**最直接的差别**：

**实时操作系统严格规定了限定时间内必须完成任务，否则会导致严重后果发生。当内核处于相对消耗状态时，非实时操作系统处理延时会增加，甚至无法保证最高优先级任务执行的最后时限。**

### （3）、 操作系统的内核结构

操作系统内核按照提供服务的大小分为宏内核和微内核。

#### 宏内核（Monolithic Kernel）

在操作系统诞生的时候，Unix 的设计就是宏内核，宏内核是一种将大部分系统功能都直接实现在内核中的操作系统内核设计。这种设计的优点在于，它具有非常高的执行效率和良好的操作系统资源管理能力，因为操作系统资源可以直接在内核中实现。另外，宏内核可以减少进程间通信带来的性能损失，并提高系统的执行速度。然而，宏内核的缺点在于，由于所有组件都在内核中实现，内核的代码量非常大，因此宏内核的可靠性和安全性也存在问题。

内核管理着用户程序和硬件之间的系统资源，在宏内核架构中，用户服务和内核服务在同一空间中实现。具体一点，就是内核可以代表内核进程运行代码，就是通常的内核进程；当用户进程经过系统调用或者中断进入到内核态时，内核也可以代表它运行代码。这样一来，宏内核需要管理的资源多于微内核，其大小就相对大一些了。

在宏内核架构当中，内核管理着CPU调度，内存管理，文件管理和系统调用等各模块的的工作，由于用户服务和内核服务被实现在同一空间中，这样在执行速度上要比微内核快。然而，宏内核的劣势也是显而易见的，那就是当内核中的某个服务崩溃了，整个内核也会崩溃。另一点，想要在内核中添加新的功能就意味着内核中的各个模块需要做相应的修改，因此其扩展性很弱。

#### 微内核（Micro kernel）

为了解决安全性和稳定性，那就需要对内核态运行的程序进行剥离和隔离，所以微内核只实现最基本的功能，其他功能则通过进程间通信来实现。

微内核则是一种只实现最基本功能的操作系统内核设计。这种设计的优点在于，它可以避免操作系统内核越来越大的问题，减少内核的复杂性、提高内核的安全性，使得内核更加稳定、可靠。但是微内核的缺点在于，它往往需要使用进程间通信等方式来实现某些功能，这些方式往往会降低系统的执行效率。

内核管理着所有的系统资源，在微内核中用户服务和内核服务在不同的地址空间中实现。在应用程序和硬件的通信中，内核进程和内存管理的极小的服务，而客户端程序和运行在用户空间的服务通过消息的传递来建立通信，它们之间不会有直接的交互，这样一来，微内核中的执行速度相对就比较慢了，这是微内核架构的一个缺点。

在内核架构中，用户服务是独立于内核服务的，因此任何用户服务崩溃都不会影响到内核服务，这就加强了操作系统的健壮性，这是微内核的优势所在。另一点，微内核的扩展性强，添加一个功能，只需要建立一个新的服务到用户空间当中，而内核空间不需要任何的修改。因此，微内核可移植性强、安全并且易于扩展。

#### 宏内核和微内核的差异

除了两者在基本内核服务上的差异之外，由于设计角度的不同，在其它服务上的设计上也有很大的区别。

微内核将很多的操作系统服务移动到了用户空间，我们来列张表来对比一下微内核和宏内核的具体区别：

内存管理的区别：

宏内核在内核空间就实现了系统所有内存管理所需的一切业务，包括内存分配策略、虚拟内存管理分页算法等，如下图所示：

对于微内核的内存管理实现，有一个发展过程。在第一代的微内核架构中，内核代理了用户空间的内存管理，控制着内存访问的权限，如下图所示。

内核中的某个服务负责管理缺页异常并保存新分配的页，只要有缺页异常发生，请求就经过内核通知页管理器。页管理器必须进入特权模式下来获取内存的访问，然后回到用户模式下。然后发送一个返回结果来触发进程，当然这个过程也是需要经过内核的。处理缺页异常或者保存新分配页的整个过程是繁复而耗时的。

为了弥补性能上的损失，之后的微内核构架在内存管理上作了相应的改变。每个进程有3个内存管理原语(Primitves)：map、grant和flush。如果进程想要共享它的内存页那么它的可以通过map来映射内存页给其它进程。当进程通过grant来让渡它的内存页给其它进程之后，该进程就丧失了这些内存页的访问权，直到让渡进程不再刷新它们。如此一来，整个系统运行过程就成了这样：在系统启动时，内核将所有内存保存到一个叫做基本的系统进程 (basic system process) 当中，当然这个所谓有基本进程是运行在用户空间中的，如果其它进程需要内存，就是再需要从内核中获取，它可以直接向这个基本的系统进程索取。

其它模块的区别：

微内核和宏内核在设计上还存在一些其它的区别，比方IO管理方面，宏内核的设备驱动直接在内核中实现，硬件中断也直接在内核中处理；但是在微内核中，内核是不直接处理IO中断管理的，来自硬件的请求将被重定向到用户服务中去，比方内核捕获了一个中断，那么内核发送给设备驱动服务就完事了，设备驱动服务会去处理这个中断。

总结：

微内核效率比宏内核慢，但在安全性、可靠性方面要比宏内核好，在扩展性方面微内核也有优势。

从内核构架发展趋势来说，将来或许会有比Linux更为强劲的基于微内核架构的OS出现，让我们拭目以待吧！

## 2. 开发环境

计算机只认得二进制指令，也就是 01010101 这种的数字高低电平，它非常善于处理数字计算。开发就是将想让计算机做的计算用特定的语言写出来，然后用编译器翻译成二进制指令，让计算机执行。

### （1）、 GCC

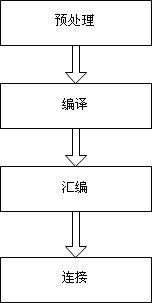
#### 编译步骤

GCC（GNU C Compiler）全名 CNU project C and C++，为 GCC 编译套件，最初由开源 GNU 事业领袖理查德·斯托曼（Richard Stallman） 于 1985 开发。最初的 gcc 只能处理 C 语言，经过扩展，很快可处理 C++ 。经过现代多语言的发展，Fortran、Pascal、Objective -C、Java、Ada、Go等也在 gcc 的支持列表中。目前已经成为 Linux 发行版默认的编译器，以 gcc 命令的形式在终端（Shell）中提供。

如果 Debian 环境中没有提供 gcc 程序，可以通过下面的命令进行安装：

sudo apt install gcc

gcc 对源文件的处理是四个步骤，第一步：预处理，第二步：编译，第三步：汇编，第四步：链接。



从功能上分，预处理、编译、汇编是三个不同的阶段，但 GCC 的实际操作上，它可以把这三个步骤合并为一个步骤来执行。下面我们以 C 语言为例来谈一下不同阶段的输入和输出情况。

我们新建一个 helloworld 程序！

root@M5Core135:~/hello# cat << EOF > helloworld.c  
#include <stdio.h>  
int main(int argc, char \*argv[])  
{  
 printf("hello world !\n");  
 return 0;  
}  
EOF

预处理：

root@M5Core135:~/hello# gcc -E helloworld.c -o helloworld.i

查看 helloworld.i 文件，它会将所有的 #include 的头文件展开，然后替换 C 文件中所有的宏。

编译：

root@M5Core135:~/hello# gcc -S helloworld.i -o helloworld.s

生成的 helloworld.s 文件就是 gcc 将 C 的源码编译成了汇编文件。

汇编：

gcc -c helloworld.s -o helloworld.o

这一步是将汇编文件翻译成机器的二进制指令，查看 helloworld.o 就知道，这是一个二进制文件。此时该文件还是不能执行的，它还没链接上系统库，它现在还是一个独立的机器二进制程序，没有系统库的帮助，它是无法运行的。

链接：

root@M5Core135:~/hello# gcc helloworld.o -o helloworld

这一步会生成一个 helloworld 的可执行文件，它是一个完整的 elf 文件，可以由系统加载运行。

执行:

root@M5Core135:~/hello# ./helloworld   
hello world !

在真正的编译工程的时候，通常会将预编译，编译，汇编合成一步，然后在多个源文件的情况下将他们链接起来形成一个工程。

一步编译：gcc helloworld.c -o helloworld

#### 多文件编译

单个文件可以直接使用 gcc 一步到位生成一个可执行文件，多个文件就需要分两个阶段进行编译了。

创建多个源码文件：

# 创建 second.c  
root@M5Core135:~/hello# cat << EOF > second.c  
#include <stdio.h>  
void say\_hi(void)  
{   
 printf("hi , i am second!\n");  
}   
EOF   
  
# 创建 second.h  
root@M5Core135:~/hello# cat << EOF > second.h  
#ifndef \_\_SECOND\_H\_\_  
#define \_\_SECOND\_H\_\_  
void say\_hi(void);  
#endif  
EOF  
  
# 创建 main.c  
root@M5Core135:~/hello# cat << EOF > main.c  
#include <stdio.h>  
#include "second.h"  
int main(int argc, char \*argv[])  
{  
 say\_hi();  
 printf("hi , i am main!\n");  
return 0;  
}  
EOF  
  
# 编译  
root@M5Core135:~/hello# gcc -c second.c -o second.o  
root@M5Core135:~/hello# gcc -c main.c -o main.o  
root@M5Core135:~/hello# gcc main.o second.o -o main  
  
# 执行  
root@M5Core135:~/hello# ./main   
hi , i am second!  
hi , i am main!

#### GCC 编译静态库和动态库

首先我们创建多个文件源码：

// add\_minus.c  
#include "add\_minus.h"  
  
int add(int a, int b)  
{  
 return a + b;  
}  
  
int minus(int a, int b)  
{  
 return a - b;  
}  
  
// add\_minus.h  
#ifndef \_\_ADD\_MINUS\_H\_\_  
#define \_\_ADD\_MINUS\_H\_\_  
  
int add(int a, int b);  
int minus(int a, int b);  
  
#endif /\*\_\_ADD\_MINUS\_H\_\_\*/  
  
// main.c  
#include <stdio.h>  
#include "add\_minus.h"  
#include "multi\_div.h"  
  
int main(void)  
{  
 int rst;  
  
 printf("Hello Cacu!\n");  
  
 rst = add(3, 2);  
 printf("3 + 2 = %d\n", rst);  
  
 rst = minus(3, 2);  
 printf("3 - 2 = %d\n", rst);  
  
 rst = multi(3, 2);  
 printf("3 \* 2 = %d\n", rst);  
  
 rst = div(6, 2);  
 printf("6 / 2 = %d\n", rst);  
  
 return 0;  
}  
  
// multi\_div.c  
#include "multi\_div.h"  
  
int multi(int a, int b)  
{  
 return a \* b;  
}  
  
int div(int a, int b)  
{  
 return a / b;  
}  
  
// multi\_div.h  
#ifndef \_\_MULTI\_DIV\_H\_\_  
#define \_\_MULTI\_DIV\_H\_\_  
  
int multi(int a, int b);  
int div(int a, int b);  
  
#endif /\*\_\_MULTI\_DIV\_H\_\_\*/

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ ls  
add\_minus.c add\_minus.h main.c multi\_div.c multi\_div.h

编译静态库：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ gcc -c add\_minus.c -o add\_minus.o  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ ar rcs libadd\_minus.a add\_minus.o

编译动态库：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ gcc -fPIC -c multi\_div.c -o multi\_div.o  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ gcc -shared multi\_div.o -o libmulti\_div.so

编译 main.c 源码:

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ gcc -c main.c -o main.o

查看当前文件：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── add\_minus.o  
├── libadd\_minus.a  
├── libmulti\_div.so  
├── main.c  
├── main.o  
├── multi\_div.c  
├── multi\_div.h  
└── multi\_div.o  
  
0 directories, 10 files

使用动态库和静态库生成可执行文件：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ gcc -o main main.o -L. -I. -ladd\_minus -lmulti\_div -Wl,-rpath,./  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── add\_minus.o  
├── libadd\_minus.a  
├── libmulti\_div.so  
├── main  
├── main.c  
├── main.o  
├── multi\_div.c  
├── multi\_div.h  
└── multi\_div.o  
  
0 directories, 11 files  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ ldd main  
 linux-vdso.so.1 (0x00007fff3d309000)  
 libmulti\_div.so => ./libmulti\_div.so (0x00007f8130312000)  
 libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007f8130000000)  
 /lib64/ld-linux-x86-64.so.2 (0x00007f813031e000)

执行：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ ./main   
Hello Cacu!  
3 + 2 = 5  
3 - 2 = 1  
3 \* 2 = 6  
6 / 2 = 3

编译动态库的时候一定要添加 -fPIC 选项让代码运行与位置无关，否则运行可能会出错。

-Wl 是将后面的指令传递给链接器，-rpath 是设置运行时动态库的搜索路径， ./ 执行命令的当前目录。

#### GCC 常用编译指令

gcc 的基本用法是：gcc [options] [filenames],[options] 表示参数，[filenames] 表示相关文件的名称，一些最基本的参数及含义下表所示：

|  |  |
| --- | --- |
| 参数 | 含义 |
| -c | 编译程序，但不链接成为可执行文件，生成二进制文件 .o |
| -E | 编译器经过预处理阶段后停止工作，生成预编译文件 .i |
| –S | 编译器经过预处理阶段后停止工作，生成汇编文件.s |
| -o | 指定输出文件的名称 |
| -D | 设置编译的宏定义 |
| -O/-O1/-O2/O3 | 代码优化等级 |
| -Os | 优化代码体积到最小 |
| -Og -g | 生成调试信息，方便 gdb 调试，会增大可执行文件体积 |
| -O0 | 不使用优化 |
| -w | 不输出任何警告信息 |
| -Wall | 将警告视为错误，编译过程一有警告便停止编译 |
| -v | 输出 gcc 工作的详细过程 |
| -ggdb | 加入 GDB 调试器能识别的格式 |
| -static | 链接静态库生成目标文件，不使用动态库 |
| -share | 尽量使用动态库,但是需要系统存在动态库 |
| -shared | 生成适用于共享库的与地址无关的代码（PIC） |
| -fPIC | 生成与位置无关的的代码，适用于使用动态库，与 -fpic的区别在于去除去全局偏移表的任何限制 |
| -fPIE | 使用与地址无关的代码生成可执行文件 |
| -Wl,option | 将 option 作为选项传递给链接器。如果选项包含逗号，则将其在逗号处拆分为多个选项。如：-Wl,-Map,output.map 传递 -Map output.map 给链接器（ GNU的链接器也可这样表示：-Wl,-Map=output.map ） |
| -T script | 使用 script 作为链接器的脚本 |
| -I dir | 将 dir 作为包含的头文件路径 |
| -L dir | 将 dir 作为 -l 操作的文件路径 |
| -sysroot=dir | 指定 dir 作为头文件和库文件寻找的根目录，这个选项在交叉编译时很有用 |

gcc 的相关知识非常多，比如控制 ELF 结构的链接脚本，控制各种警告的选项等等，读者可以自行研究。

<https://zhuanlan.zhihu.com/p/265099138>

### （3）、 Makefile

#### Makefile 简介

1977年，斯图亚特·菲尔德曼在贝尔实验室里制作了这个软件，并将其包含在了Unix系统中。这款软件是一个工具程序（Utility software），通过读取叫做 “Makefile” 的文件，自动化构建软件。大多数情况下，他被用来编译代码，生成结果代码，然后把结果代码链接起来生成可执行文件或者哭文件。Make出现之后，很快的就普及开来，在众多的依赖关系检查工具中，make 是应用最为广泛的一个。2003年，斯图亚特·菲尔德曼因为发明了这样一款重要的工具而接受了美国计算机协会（ACM）颁发的软件系统奖。直到现在，make 仍然被用来编译很多完整的操作系统。最著名的操作系统内核 Linux 依然选用 make 作为制作工具。

Makefile 是 make 的脚本文件，Makefile 中会存放目标和制作目标需要的步骤和顺序。

make 的工作原则：

第一、确定最终目标。

第二、确定中间步骤。

第三、确定步骤之间的依赖顺序。

第四、根据依赖顺序，判断依赖的步骤是否需要重新执行，一步步执行中间步骤并完成最终目标的制作。

Makefile 对编程的帮助，在编译程序的过程中，我们经常需要重复执行各种 gcc 命令操作。这样不仅费时费力，当有成千上万的源文件时，我们根本没办法通过手动输入完成这些操作。而 而 make 会监控我们的 c 源文件，并且自动的输出编译指令，从而把我们从不断的执行编译命令的工作中解救出来。

Makefile 和 IDE 的关系，Makefile 属于制作工具，现代 IDE 一般会自动处理制作过程，也有些 IDE 会生成 Makefile 然后用 make 完成制作过程。

除此之外，你甚至还可以在 Makefile 中储存一些你想调用的系统终端的命令，像一个 Shell 脚本一样使用它。

#### 编写 Makefile

Makefile 的主要的编写规则有三个：

target: dependencies ...  
 commands  
 ...

* target 就是我们需要让 make 制作出来的目标。
* dependencies 是制作 target 需要的依赖，用空格隔开，这个 dependencies 可以是文件，也可以是其他的 target。
* commands 是制作 target 需要执行的命令。

非常值的注意的是，Makefile 的 commands 前面是用 tab 隔开的，并不是空格，如果用空格替代了就会出现错误。

现在我们来编写一个 Makefile 的 hello world！

# makefile  
say\_hello:  
 echo "hello world!"

在这个 Makefile 中，定义了一个 say\_hello 的目标，该目标没有依赖。制作这个目标需要执行一个 echo "hello world!" 的指令。

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make  
echo "hello world!"  
hello world!

现在我们执行这个 Makefile。make 会自动搜索执行 make 命令目录下的 "Makefile" 文件，并读取 Makefile 文件中的第一个目标作为默认目标，然后输出制作命令。

Makefile 中的目标只是一个声明而以，不用担心 target 的起名问题。由于 Makefile 中的目标具有可提示性，你可以用 Makefile 做一个简单的有提示命令选择脚本。

比如：

# makefile  
say\_hello:  
 echo "hello world!"  
  
say\_hi:  
 echo "hi! How are you?"

当你输入 make 命令后，就可以通过 tab 键进行补全目标。当 make 指定目标后，它只会构建指定的目标。

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make say\_hi   
echo "hi! How are you?"  
hi! How are you?

如果不希望 make 回显，可以在命令前面添加 "**@**" 符号。

# makefile  
say\_hello:  
 @echo "hello world!"  
  
say\_hi:  
 @echo "hi! How are you?"

再次执行：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make  
hello world!  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make say\_hi  
hi! How are you?

我们以 gcc 中编译静态库和动态库为例，编写一个 Makefile 用于介绍：

# 定义目标的版本  
VERSION = 1.00  
  
# 定义目标的名字  
TARGET = main-$(VERSION)  
  
# 使用 ?= 定义编译器，如果系统中没有定义 CC 编译器，就把 CC 设定成 gcc  
CC ?= gcc  
  
# 使用 ?= 定义打包器，如果系统中没有定义 AR 打包器，就把 AR 设定成 ar  
AR ?= ar  
  
# 使用 ?= 添加一个编译的宏定义  
DEBUG ?= -DUSE\_DEBUG  
  
# 使用 := 定义需要编译的源文件， := 强制将后面的值赋值到 SOURCES   
SOURCES := add\_minus.c main.c multi\_div.c  
  
# 使用 += 添加头文件寻找目录  
INCLUDES += -I.  
  
# 使用 += 添加需要的库  
LIB\_NAMES += -lmulti\_div -ladd\_minus  
  
# 使用 += 添加库寻找路径  
LIB\_PATH += -L.  
  
# 使用 patsubst 函数将 SOURCES 中以 .c 结尾的文件名替换成 .o 的文件名  
OBJ += $(patsubst %.c, %.o, $(SOURCES))  
  
# 添加编译选项  
CFLAGS += -Wall  
  
# 添加链接选项  
LDFLAGS += -Wl,-rpath,./  
  
# 链接成可执行文件  
$(TARGET):$(OBJ) libmulti\_div.so libadd\_minus.a  
 $(CC) $(OBJ) $(LIB\_PATH) $(LIB\_NAMES) -o $(TARGET) $(LDFLAGS)  
   
# 制作 libmulti\_div.so 动态库  
libmulti\_div.so:multi\_div.o  
 $(CC) -shared multi\_div.o -o libmulti\_div.so  
  
# 制作 libadd\_minus.a 静态库  
libadd\_minus.a:add\_minus.o  
 $(AR) rcs libadd\_minus.a add\_minus.o  
  
# 通过自动推导，将所有的 .c 文件编译成二进制 .o 文件  
%.o: %.c  
 $(CC) $(INCLUDES) $(DEBUG) -c $(CFLAGS) $< -o $@  
  
# 清理编译出的文件  
clean:  
 @echo "Remove compiled files......"  
 $(RM) $(OBJ)  
  
# 删除所有编译出的文件  
distclean:  
 @echo "Remove linked and compiled files......"  
 $(RM) $(OBJ) $(TARGET) libmulti\_div.so libadd\_minus.a

下面执行这个 makefile ：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── main.c  
├── Makefile  
├── multi\_div.c  
└── multi\_div.h  
  
0 directories, 6 files  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make  
cc -I. -DUSE\_DEBUG -c -Wall add\_minus.c -o add\_minus.o  
cc -I. -DUSE\_DEBUG -c -Wall main.c -o main.o  
cc -I. -DUSE\_DEBUG -c -Wall multi\_div.c -o multi\_div.o  
cc -shared multi\_div.o -o libmulti\_div.so  
ar rcs libadd\_minus.a add\_minus.o  
cc add\_minus.o main.o multi\_div.o -L. -lmulti\_div -ladd\_minus -o main-1.00 -Wl,-rpath,./  
# make 会自动检测文件是否发生了改动，只有改动的文件才会被重新编译  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make  
make: “main-1.00”已是最新。  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── add\_minus.o  
├── libadd\_minus.a  
├── libmulti\_div.so  
├── main-1.00  
├── main.c  
├── main.o  
├── Makefile  
├── multi\_div.c  
├── multi\_div.h  
└── multi\_div.o  
  
0 directories, 12 files  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make clean   
Remove compiled files......  
rm -rf add\_minus.o main.o multi\_div.o  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── libadd\_minus.a  
├── libmulti\_div.so  
├── main-1.00  
├── main.c  
├── Makefile  
├── multi\_div.c  
└── multi\_div.h  
  
0 directories, 9 files  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make distclean   
Remove linked and compiled files......  
rm -rf add\_minus.o main.o multi\_div.o main-1.00 libmulti\_div.so libadd\_minus.a  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── main.c  
├── Makefile  
├── multi\_div.c  
└── multi\_div.h  
  
0 directories, 6 files

#### Makefile 变量

Makefile 中的变量和 C 语言中的宏定义类似，主要起的作用是字符替换。值得注意的是，Makefile 中的变量是大小写敏感的。

Makefile 中的变量主要有三类：

* 用户自定义变量，用户自己定义的变量，可随意更改，比如上方的 SOURCES
* 环境变量，make 工作时提供的变量，谨慎更改，比如上方的 CC 、 AR
* 自动变量，make 工作时自动推导出的变量，由 make 完成更改，比如上方的 @、$<

Makefile 中常用的环境变量，更多的请查看 make 手册：

|  |  |
| --- | --- |
| 变量名 | 说明 |
| $@ | **表示目标文件的名字。**如果规则有多个目标，它代表的是触发规则被执行的文件名。 |
| $% | 当目标文件是一个静态库文件时，代表静态库的一个成员名。 |
| $< | **表示第一个依赖文件的名字。**如果是一个目标文件使用隐含的规则来重建，则它代表由隐含规则加入的第一个依赖文件。 |
| $? | **表示所有比目标文件更新的依赖文件列表**，以空格分隔。 |
| $^ | **表示所有依赖文件列表**，以空格分隔。一个文件可重复的出现在目标的依赖中，变量“^”会去掉重复的依赖文件。 |
| $+ | 与“$^”类似，但它保留了依赖文件中重复出现的文件。 |
| $\* | 在模式规则和静态模式规则中，代表“茎”。“茎”是目标模式中“%”所代表的部分（当文件名中存在目录时，“茎”也包含目录部分）。这啥意思？ |
| CC | 用于指定C语言的编译器。 |
| CXX | 用于指定C++语言的编译器。 |
| AR | 用于指定归档文件的命令。 |
| LD | 用于指定链接器的命令。 |
| RM | 用于指定删除文件的命令。 |
| CFLAGS | 用于指定C语言编译器的选项，例如优化级别、警告级别、编译器标志等。 |
| CXXFLAGS | 用于指定C++语言编译器的选项，与CFLAGS类似，但专门针对C++代码。 |
| LDFLAGS | 用于指定链接器的选项，例如库文件路径和链接器标志。 |

Makefile 的环境变量也可以由 make 显式指定：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make TARGET=main  
cc -I. -DUSE\_DEBUG -c -Wall add\_minus.c -o add\_minus.o  
cc -I. -DUSE\_DEBUG -c -Wall main.c -o main.o  
cc -I. -DUSE\_DEBUG -c -Wall multi\_div.c -o multi\_div.o  
cc -shared multi\_div.o -o libmulti\_div.so  
ar rcs libadd\_minus.a add\_minus.o  
cc add\_minus.o main.o multi\_div.o -L. -lmulti\_div -ladd\_minus -o main -Wl,-rpath,./  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$ tree  
.  
├── add\_minus.c  
├── add\_minus.h  
├── add\_minus.o  
├── libadd\_minus.a  
├── libmulti\_div.so  
├── main  
├── main.c  
├── main.o  
├── Makefile  
├── multi\_div.c  
├── multi\_div.h  
└── multi\_div.o  
  
0 directories, 12 files  
nihao@nihao-z690:~/work/test/make\_example-master/tmp$

当 make 指定了 TARGET=main ，编译出的可执行文件变成了 main 。

Makfile 可以使用 include 指令导入其他 makefile 的变量。

#### Makefile 函数

Makefile 会提供一些函数供我们使用，调用方式是 $(fun args)

LOG 输出函数：

$(error "this is error")  
$(warning "this is warning")  
$(info "this is info")

如果我们想要查看 SOURCES 有多少个源文件，我们可以在 Makefile 中添加 $(info $(SOURCES))

# $(error "this is error")  
$(warning "this is warning")  
$(info "this is info")  
test:  
 @echo "test"

执行：

nihao@nihao-z690:~/work/test/make\_example-master/tmp$ make  
Makefile:68: "this is warning"  
"this is info"  
test

能看出有源文件名被打印了出来。

(abs\_objtree)) 能够打印出行号信息，方便定位问题。

$(error "this is error") 执行后会退出制作过程。

常用的字符串操作函数 ：

$(subst <from>,<to>,<text>)  
名称：字符串替换函数——subst。  
功能：把字串<text>中的<from>字符串替换成<to>。  
返回：函数返回被替换过后的字符串。  
示例：$(subst hel,HEL,hello world!)  
HELlo world!  
  
  
$(patsubst <pattern>,<replacement>,<text>)   
名称：模式字符串替换函数——patsubst。  
功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符%，表示任意长度的字串。如果<replacement>中也包含%，那么，中的这个%将是<pattern>中的那个%所代表的字串。（可以用/来转义，以/%来表示真实含义的%字符）  
返回：函数返回被替换过后的字符串。  
示例：$(patsubst %.c,%.o,add\_minus.c main.c multi\_div.c)  
add\_minus.o main.o multi\_div.o  
  
  
$(strip <string>)  
名称：去空格函数——strip。  
功能：去掉<string>字串中开头和结尾的空字符。  
返回：返回被去掉空格的字符串值。  
示例：$(strip a b c)  
abc  
  
  
$(findstring <find>,<in>)  
名称：查找字符串函数——findstring。  
功能：在字串<in>中查找<find>字串。  
返回：如果找到，那么返回<find>，否则返回空字符串。  
示例：$(findstring a,a b c)  
a  
$(findstring a,b c)  
""  
  
  
$(filter <pattern...>,<text>)  
名称：过滤函数——filter。  
功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。  
返回：返回符合模式<pattern>的字串。  
示例：$(filter %.c %.s,foo.c bar.c baz.s ugh.h)  
foo.c bar.c baz.s  
  
  
$(sort <list>)  
名称：排序函数——sort。  
功能：给字符串<list>中的单词排序（升序）。  
返回：返回排序后的字符串。  
sort函数会去掉<list>中相同的单词。  
示例：$(sort foo bar lose foot)  
bar foo foot lose  
  
  
$(words <text>)  
名称：单词个数统计函数——words。  
功能：统计<text>中字符串中的单词个数。  
返回：返回<text>中的单词数。  
示例：$(words, foo bar baz)  
3

常用的文件名操作函数 ：

$(dir <names...>)  
名称：取目录函数——dir。  
功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（/）之前的部分。如果没有反斜杠，那么返回./。  
返回：返回文件名序列<names>的目录部分。  
示例：$(dir src/foo.c hacks)  
src/ ./  
  
  
$(notdir <names...>)  
名称：取文件函数——notdir。  
功能：从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（/）之后的部分。  
返回：返回文件名序列<names>的非目录部分。  
示例：$(notdir src/foo.c hacks)  
foo.c hacks  
  
  
$(suffix <names...>)  
名称：取后缀函数——suffix。  
功能：从文件名序列<names>中取出各个文件名的后缀。  
返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。  
示例：$(suffix src/foo.c src-1.0/bar.h hacks)  
.c .h  
  
  
$(basename <names...>)  
名称：取前缀函数——basename。  
功能：从文件名序列<names>中取出各个文件名的前缀部分。  
返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。  
示例：$(basename src/foo.c src-1.0/bar.c hacks)  
src/foo src-1.0/bar hacks  
  
  
$(addsuffix <suffix>,<names...>)  
名称：加后缀函数——addsuffix。  
功能：把后缀<suffix>加到<names>中的每个单词后面。  
返回：返回加过后缀的文件名序列。  
示例：$(addsuffix .c,foo bar)  
foo.c bar.c  
  
  
$(join <list1>,<list2>)  
名称：连接函数——join。  
功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。  
返回：返回连接过后的字符串  
示例：$(join aaa bbb , 111 222 333)  
aaa111 bbb222 333  
  
$(abspath <names>)  
名称：绝对路径-abspath  
功能：用于将路径转换为绝对路径。  
返回：绝对路径。  
示例：$(abspath .)  
/home/nihao/work/test/make\_example-master/tmp

功能函数 ：

$(foreach <var>,<list>,<text>)  
把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。  
<var>最好是一个变量名，<list>可以是一个表达式，而<text>中一般会使用<var>这个参数来依次枚举<list>中的单词。  
示例：   
names := a b c d  
$(foreach n,$(names),$(n).o)  
a.o b.o c.o d.o  
  
  
$(call <expression>,<parm1>,<parm2>,<parm3>...)  
call函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用call函数来向这个表达式传递参数。  
当make执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是call函数的返回值。  
示例：   
reverse = $(1) $(2)  
foo = $(call reverse,a,b)  
foo的值就是a b  
参数的次序是可以自定义的，不一定是顺序的  
reverse = $(2) $(1)  
foo = $(call reverse,a,b)  
foo的值就是b a  
  
  
$(shell <cmd>)  
名称：shell 命令执行函数  
功能：执行 shell 命令。  
返回：执行 shell 命令的返回值。  
示例：$(shell echo "hello world !")  
hello world !

自定义函数 ：

define myfunc  
 $(if $(filter $(1), yes), \  
 @echo "My function param is Yes", \  
 @echo "My function param is Not Yes" \  
 )  
endef  
  
test:  
 $(call myfunc,yes) # 调用 myfunc 函数，输出 "My function param is Yes"  
 $(call myfunc,no) # 调用 myfunc 函数，输出 "My function param is Not Yes"

#### Makefile 判断

Makefile 需要用到判断控制条件制作，Makefile 的判断和宏定义类似，主要有四个条件判断：ifeq、ifneq、ifdef、ifndef。

条件表达式的语法:

<conditional-directive>;   
 <text-if-true>;   
else   
 <text-if-false>;   
endif

ifeq：表示条件语句的开始，并指定一个条件表达式。表达式包含两个参数，以逗号分隔，表达式以圆括号括起，表示判断表达式与某个值是否相等。

OS = linux  
ifeq ($(OS),linux)   
 CFLAGS += -DLINUX  
else   
 CFLAGS += -DNO\_OS  
endif   
  
# ifneq（与 ifeq 相反）

ifdef：如果变量 variable-name; 的值 **非空**，表达式为**真**。否则，表达式为假。当然，variable-name; 同样可以是一个函数的返回值。

OS = linux  
ifdef OS  
 CFLAGS += -DLINUX  
else   
 CFLAGS += -DNO\_OS  
endif   
  
# ifndef（与 ifdef 相反）

判断文件夹是否存在，如果不存在则创建文件夹:

# 文件夹名  
FOLDER\_NAME := my\_folder  
  
# 查询文件夹是否存在  
ifeq ($(wildcard $(FOLDER\_NAME)),)  
 $(shell mkdir $FOLDER\_NAME)  
endif

<https://zhuanlan.zhihu.com/p/150176358>

## 3. Linux c

C 语言是汤普逊和里奇对当时的 B 语言改进而来的，其诞生之初就被用来编写 UNIX 系统，是非常强大的一门语言。

C 具有非常鲜明的特性：

* **过程化编程：** C是一种过程化编程语言，程序由一系列函数组成。这使得程序的结构相对清晰，易于理解和维护。
* **低级语言特性：** C语言提供了对计算机底层硬件的直接访问，允许进行位级操作和内存管理。这使得C语言非常适合系统编程和嵌入式系统开发。
* **可移植性：** C语言编写的程序可以在不同的平台上进行编译和运行，这是因为C语言对硬件的依赖性相对较低。这种可移植性使得C语言成为跨平台开发的理想选择。
* **中级语言：** C语言被称为中级语言，因为它同时具有高级语言和低级语言的特性。它提供了高级语言的抽象特性，同时允许直接访问底层硬件。
* **面向过程：** C语言是一种面向过程的编程语言，程序的执行是按照一系列定义好的步骤（过程或函数）顺序执行的。
* **简洁而高效：** C语言的语法相对简单，但足够强大。它允许程序员以紧凑的方式表达想法，并且编写的代码通常执行效率很高。  
  **丰富的标准库：** C语言提供了丰富的标准库，包含大量的函数，涵盖了文件操作、字符串处理、数学计算等各个方面。这些库可以帮助程序员更容易地完成任务，同时提高了代码的可移植性。
* **指针和内存管理：** C语言允许直接使用指针，使得程序员可以更灵活地操作内存。但与此同时，这也带来了需要谨慎管理内存的责任。
* **丰富的运算符：** C语言提供了丰富的运算符，包括算术运算符、逻辑运算符、关系运算符等，使得程序员可以进行各种复杂的计算和逻辑操作。
* **支持系统级编程：** 由于C语言提供了对底层硬件的直接访问，它非常适合进行系统级编程，如编写操作系统、驱动程序等。

嵌入式编程 C 语言是首选，C 语言的基本语法和编程方法在此便不在赘述，本章节主要介绍 C 语言标准库中常用的头文件和函数。

### C 语言标准

C 语言最开始没有官方标准，布莱恩·柯林汉（Brian Kernighan） 和 丹尼斯·里奇（Dennis Ritchie） 出版了一本书，名叫《 [The C Programming Language](http://baike.baidu.com/view/5012996.htm)》。这本书被 C语言开发者们称为 **K&R**，很多年来被当作 C语言的非正式的标准说明。人们称这个版本的 C语言为 **K&R C**。这个标准初步规定了 C 语言的变量，语法，程序结构。

1989年，美国国家标准协会（ANSI）通过了C语言标准，被称为 **ANSI X3.159-1989 "Programming Language C"**。因为这个标准是1989年通过的，所以一般简称 **C89标准**。有些人也简称 **ANSI C**，因为这个标准是美国国家标准协会（ANSI）发布的。

1990年， [国际标准化组织](http://baike.baidu.com/view/42488.htm)（ISO）和 [国际电工委员会](http://baike.baidu.com/view/159311.htm)（IEC）把C89标准定为C语言的国际标准，命名为 **ISO/IEC 9899:1990 - Programming languages -- C** [5] 。因为此标准是在1990年发布的，所以有些人把简称作 **C90标准**。不过大多数人依然称之为 **C89标准**，因为此标准与ANSI C89 标准完全等同。它规定了C语言的基本语法、数据类型、运算符等，并定义了一个基本的标准库。该标准奠定了C语言的基础，并广泛应用于早期的C编译器和开发工具。

1999 年1月，国际标准化组织（ISO）和国际电工委员会（IEC）发布了C语言的新标准，名叫 **ISO/IEC 9899:1999 - Programming languages -- C** [8] ，简称 **C99 标准**。这是C语言的第二个官方标准, 也是目前使用人数最多的标准。它引入了新的语言特性，如单行注释、布尔类型、变长数组、复合字面量等。此外，C99标准还增加了更多的库函数，包括对复数数学、格式化输入输出、浮点数环境等的支持。然而，C99的广泛采用相对较晚，因此某些编译器和开发环境对其支持有限。

2011年12月8日，国际标准化组织（ISO）和国际电工委员会（IEC）再次发布了C语言的新标准，名叫 **ISO/IEC 9899:2011 - Information technology -- Programming languages -- C** [9] ，简称 **C11标准**，原名 **C1X**。这是C语言的第三个官方标准，也是C语言的最新标准。它在C99的基础上进行了进一步改进，并引入了一些新的特性，如类型泛型、多线程支持、匿名结构体和联合体、对 Unicode 字符的支持等。C11还扩展了标准库，增加了新的头文件和函数，如对多线程编程的支持、原子操作、时间和日期处理等。

### C 编译器

C 编译器负责对 C 语言语法编译的实现，C 语言想要转换成 C 程序能够执行，那就需要 C 编译器帮忙编译。

C语言有许多不同的编译器可供使用，每个编译器都有其独特的特性、性能和支持程度。以下是一些常见的C语言编译器：

* GCC（GNU Compiler Collection）：GCC 是一个广泛使用的开源编译器套件，支持多种编程语言，包括C语言。它是许多Linux发行版的默认编译器，并且在其他平台上也有广泛的支持。GCC 提供了丰富的优化选项和严格的符合标准的支持，被认为是一个强大而可靠的编译器。
* Clang：是 LLVM 项目下的 C/C++/Objective-C/Objective-C++ 编译器，，支持C、C++ 和Objective-C。它设计注重灵活性、可扩展性和良好的诊断能力。Clang 具有较好的标准兼容性，并提供了先进的静态分析和代码检查功能。Clang 的静态分析能力使其成为许多代码编辑器和集成开发环境（IDE）的首选。
* Visual C++：Visual C++ 是微软公司提供的专有编译器，它是 Visual Studio 集成开发环境的一部分。Visual C++ 提供了一套完整的工具链，包括编辑器、调试器和性能分析器。它在 Windows 平台上具有良好的集成和支持，并且提供了一些扩展功能，如Windows API 的直接支持。
* Intel C++ Compiler：Intel C++ Compiler 是由英特尔公司提供的优化编译器，针对 Intel 架构和处理器进行了优化。它提供了强大的优化选项和针对特定硬件的优化技术，可生成高性能的代码。Intel C++ Compiler 在科学计算和高性能计算领域有较广泛的应用。

当然，还有一些特殊的编译器可供选择。TCC（Tiny C Compiler）非常小的编译器，可在内存非常小的地方工作，支持将 C 语言当成脚本执行。IBM XL C/C++、Digital Mars C/C++ Compiler 特殊厂家编译器。

编译器可以指定对 C 标准的支持版本，比如 GCC 指定 C 标准为 C11：

gcc -std=c11 -o exm exm,c

### C 标准库

C 语言标准库是一组**预定义函数、宏和文件的集合**，这些函数和文件提供了一些操作系统基本的功能和操作，可供 C 语言程序调用使用。

最开始 C 标准库是 ANSI 发布的 libc，后来 GNU 组织发布了自己的 glibc，glibc 兼容 libc 并扩展了一些函数。随这嵌入式的需求，有 newlibc，microlibc 等精简的 C 库。

**C语言标准库包含以下三部分**：

* **头文件（Header Files）**：C语言标准库包含一系列的头文件，用于声明函数原型、定义常量和数据类型等。开发者可以通过包含适当的头文件来使用标准库中提供的函数和类型。
* **静态库（Static Libraries）**：C语言标准库还以静态库的形式提供，其中包含了标准库函数和类型的实现代码。静态库是在编译时与应用程序一起链接，使得应用程序可以调用标准库提供的函数和类型。
* **动态库（Dynamic Libraries）**：一些操作系统和编译器还提供了C语言标准库的动态库版本。动态库是在运行时被应用程序动态加载和链接的库。应用程序可以共享同一个动态库的实例，从而节省内存，并且可以在库的更新时实现简单的升级。

函数的名字在头文件中声明，调用的函数程序则包含在库中，比如 C 语言的开始 “hello world！” 用的就是 stdio.h 标准输入输出头文件，需要静态或者动态的链接到标准库中。

#### 标准库头文件

C 库有很多个头文件，头文件的数量和编译器，标准库都有关系，下面将会介绍 15 个常用的标准头文件,例出常用的函数名：

**<stdio.h>**

提供了标准输入输出函数的声明、定义和相关宏定义。

* 输入输出
  + printf：格式化输出到标准输出流。
  + scanf：从标准输入流中读取格式化输入。
  + puts：输出字符串到标准输出流并追加换行符。
  + gets：从标准输入流中读取一行字符串。
  + fgets：从指定的输入流中读取一行字符串，包括换行符。
  + fputc 和 fgetc：将字符写入和从指定的文件流读取字符。
* 文件操作
  + fopen：打开一个文件。
  + fclose：关闭一个文件。
  + fread 和 fwrite：从文件读取数据和将数据写入文件。
  + fprintf：将格式化数据写入文件。
* 宏定义
  + stdin、stdout 和 stderr：表示标准输入、标准输出和标准错误输出流的文件指针。
  + EOF：表示文件结束符。
  + BUFSIZ：定义了标准 I/O 缓冲区的大小。
  + NULL：表示空指针。

**<stdlib.h>**

提供了一些常用的函数、类型和宏定义，用于内存管理、随机数生成、字符串转换、动态内存分配等操作。

* 内存管理
  + malloc：分配指定大小的内存块。
  + calloc：分配并清零指定数量的内存块。
  + realloc：重新分配已分配内存块的大小。
  + free：释放先前分配的内存块。
* 字符串转换
  + atoi：将字符串转换为整数。
  + atol：将字符串转换为长整数。
  + atof：将字符串转换为浮点数。
  + itoa：将整数转换为字符串。
* 伪随机数生成
  + rand：生成一个伪随机整数。
  + srand：设置随机数生成的种子值。
* 程序控制
  + exit：终止程序的执行。
  + abort：异常终止程序的执行。
* 动态内存分配
  + malloc ：用于动态地申请堆内存。
  + calloc：用于按照数量动态地申请堆内存。
  + realloc ：将扩展原来的内存区。
  + free ：释放申请的堆内存。

**<string.h>**

提供了字符串处理相关的函数和宏定义。

* 字符串操作
  + strcpy：将一个字符串复制到另一个字符串。
  + strncpy：复制指定长度的字符串到另一个字符串。
  + strcat：将一个字符串追加到另一个字符串。
  + strncat：将指定长度的字符串追加到另一个字符串。
  + strcmp：比较两个字符串的大小。
  + strncmp：比较指定长度的两个字符串的大小。
  + strlen：计算字符串的长度。
  + strchr：在字符串中查找指定字符的第一次出现。
  + strstr：在字符串中查找指定子字符串的第一次出现。
  + strtok：将字符串分解为一系列标记。
* 内存操作
  + memset：将指定的内存块设置为指定的值。
  + memcpy：将一个内存块的内容复制到另一个内存块。
  + memmove：将一个内存块的内容移动到另一个内存块。

**<math.h>**

提供了数学相关的函数和常量。

* 常见数学函数
  + sqrt：计算平方根。
  + pow：计算幂运算。
  + exp：计算指数函数。
  + log：计算自然对数。
  + sin、cos、tan：计算三角函数（正弦、余弦、正切）。
  + asin、acos、atan：计算反三角函数。
  + ceil：向上取整。
  + floor：向下取整。
  + round：四舍五入。
  + abs：计算绝对值。
  + fmod：计算浮点数的余数。
* 常量
  + M\_PI：圆周率 π 的近似值。
  + M\_E：自然对数的底数 e 的近似值。

**<time.h>**

提供了时间和日期相关的函数和类型。

* 时间获取和转换：
  + time：获取当前的系统时间，返回从 1970 年 1 月 1 日 00:00:00 UTC 到当前时间的秒数。
  + difftime：计算两个时间之间的差值，以秒为单位。
  + mktime：将时间和日期转换为从 1970 年 1 月 1 日 00:00:00 UTC 开始的秒数。
  + localtime：将时间表示为本地时间的结构体。
  + gmtime：将时间表示为协调世界时（UTC）的结构体。
* 时间格式化：
  + strftime：将时间转换为指定格式的字符串。
  + asctime：将时间转换为可读的字符串表示。
  + ctime：将时间转换为可读的字符串表示，与 asctime 类似。
* 延时函数：
  + sleep：使程序暂停指定的秒数。
* 时间类型：
  + time\_t：表示从 1970 年 1 月 1 日 00:00:00 UTC 开始的秒数。
  + truct tm：表示日期和时间的结构体。

**<ctype.h>**

提供了**字符**处理相关的函数和宏定义。

* 字符分类函数：
* isalnum：检查字符是否是字母或数字。
* isalpha：检查字符是否是字母。
* isdigit：检查字符是否是数字。
* islower：检查字符是否是小写字母。
* isupper：检查字符是否是大写字母。
* isspace：检查字符是否为空格字符。
* ispunct：检查字符是否是标点符号。
* isxdigit：检查字符是否是十六进制数字。
* isblank：检查字符是否为空白字符（空格或制表符）。
* 字符转换函数：
  + tolower：将字符转换为小写字母。
  + toupper：将字符转换为大写字母。
* 其他函数和宏：
  + isascii：检查字符是否是 ASCII 字符。
  + isgraph：检查字符是否是可打印字符。
  + iscntrl：检查字符是否是控制字符。
  + isprint：检查字符是否是可打印字符（包括空格）。

**<stdbool.h>**

提供了布尔类型和相关操作的定义和支持。

* 类型定义：
  + bool：布尔类型，表示真（true）或假（false）的逻辑值。
* 常量：
  + true：布尔类型的真值常量，等价于非零值。
  + false：布尔类型的假值常量，等价于零值。

**<errno.h>**

提供了错误处理相关的宏定义和全局变量。

* 全局变量：
  + errno：表示当前错误码的全局变量。它是一个整数类型的变量，在发生错误时会被设置为对应的错误码。
* 宏定义：
  + EDOM：表示数学域错误。
  + ERANGE：表示结果溢出或超出范围错误。

**<float.h>**

用于定义浮点数类型的特定限制和属性。它提供了有关浮点数类型的一些重要信息和常量。下面是一些常用的常量和属性：

* 浮点数类型属性：
  + FLT\_RADIX：指定浮点数的基数，通常为 2。
  + FLT\_MANT\_DIG：指定 float 类型的尾数位数。
  + FLT\_DIG：指定 float 类型的十进制精度。
  + FLT\_MIN\_EXP：float 类型的最小负指数。
  + FLT\_MAX\_EXP：float 类型的最大正指数。
  + FLT\_MIN\_10\_EXP：float 类型的最小负十进制指数。
  + FLT\_MAX\_10\_EXP：float 类型的最大正十进制指数。
* float 类型的范围和精度：
  + FLT\_MIN：float 类型的最小正数值。
  + FLT\_MAX：float 类型的最大正数值。
  + FLT\_EPSILON：float 类型的最小可表示的绝对误差。
* double 类型的范围和精度：
  + DBL\_MIN：double 类型的最小正数值。
  + DBL\_MAX：double 类型的最大正数值。
  + DBL\_EPSILON：double 类型的最小可表示的绝对误差。
* long double 类型的范围和精度：
  + LDBL\_MIN：long double 类型的最小正数值。
  + LDBL\_MAX：long double 类型的最大正数值。
  + LDBL\_EPSILON：long double 类型的最小可表示的绝对误差

**<limits.h>**

用于定义各种基本数据类型的实现特定限制。它提供了代表不同类型变量中可存储的最小值和最大值的常量。

* CHAR\_BIT：该常量表示 char 数据类型中的位数，通常为 8。
* 整数类型的限制：
  + CHAR\_MIN 和 CHAR\_MAX：char 数据类型的最小值和最大值。
  + SHRT\_MIN 和 SHRT\_MAX：short (short int) 数据类型的最小值和最大值。
  + INT\_MIN 和 INT\_MAX：int 数据类型的最小值和最大值。
  + LONG\_MIN 和 LONG\_MAX：long (long int) 数据类型的最小值和最大值。
  + LLONG\_MIN 和 LLONG\_MAX：long long (long long int) 数据类型的最小值和最大值。
* 无符号整数类型的限制：
  + UCHAR\_MAX：无符号 char 的最大值。
  + USHRT\_MAX：无符号 short 的最大值。
  + UINT\_MAX：无符号 int 的最大值。
  + ULONG\_MAX：无符号 long 的最大值。
  + ULLONG\_MAX：无符号 long long 的最大值。
* MB\_LEN\_MAX：多字节字符中的最大字节数。

**<stddef.h>**  
提供了一些与指针和宏相关的类型和常量定义。它定义了一些重要的符号常量和类型，用于帮助编写可移植的代码。

* NULL：这是一个宏，用于表示空指针常量。
* size\_t：这是一个无符号整数类型，在 C 语言中用于表示对象的大小。例如，sizeof 运算符返回的结果类型就是 size\_t。
* ptrdiff\_t：这是一个有符号整数类型，用于表示两个指针之间的差值。
* offsetof 宏：这是一个用于计算结构体成员在结构体中的偏移量的宏。它接受一个结构体类型和一个成员名作为参数，并返回该成员在结构体中的偏移量。

**<stdarg.h>**  
于处理可变数量的参数函数。它提供了一组宏和类型，使得函数能够接受不定数量的参数。

* va\_list：这是一个类型，用于声明可变数量参数列表的对象。
* va\_start 宏：它用于初始化可变参数列表。它接受两个参数，第一个是 va\_list 对象，第二个是可变参数列表的前一个已知参数。
* va\_arg 宏：它用于访问可变参数列表中的下一个参数，并指定参数的类型。它接受两个参数，第一个是 va\_list 对象，第二个是参数的类型。每次调用 va\_arg 都会返回下一个参数，并将 va\_list 对象更新为指向列表中的下一个参数。
* va\_end 宏：它用于清理可变参数列表相关的资源。它接受一个参数，即 va\_list 对象，用于标记列表的结束。

**<signal.h>**  
信号处理

* signal 函数：该函数用于注册信号处理器（Signal Handler）以响应特定的信号。它接受两个参数，第一个参数是信号编号，第二个参数是指向信号处理器函数的指针。
* raise 函数：该函数用于向当前进程发送信号。它接受一个参数，即要发送的信号编号。
* SIG\_ERR 宏：用于表示信号处理器注册失败的返回值。
* SIGINT、SIGABRT、SIGSEGV 等，用于表示特定的信号。

**<locale.h>**  
用于处理与本地化（国际化）相关的操作。它提供了一些函数和类型，用于控制程序的本地化设置，包括日期、时间、货币和语言等。

* setlocale 函数：该函数用于设置程序的本地化区域设置。它接受两个参数，第一个参数是 int 类型，用于指定本地化的分类，如日期、时间、货币等；第二个参数是一个字符串，用于指定要设置的区域设置。
* localeconv 函数：该函数返回一个指向 struct lconv 类型的指针，包含了本地化相关的货币和数值格式信息。
* struct lconv：这是一个结构体类型，包含了本地化相关的货币和数值格式信息。它的成员可以用于获取和设置货币符号、小数点符号、千位分隔符等。
* 一些定义的常量：LC\_ALL、LC\_COLLATE、LC\_CTYPE、LC\_MONETARY、LC\_NUMERIC 和 LC\_TIME，用于指定 setlocale 函数中的本地化分类。

**<setjmp.h>**  
用于提供非局部跳转函数，经常被用作异常处理恢复。

* setjmp 函数：将本函数的“调用上下文"保存到参数 env 中，同时该函数返回 0。此函数的作用可以理解为“将当前位置（调用 setjmp() 函数的位置）设置成跳转目标”；
* longjmp 函数: 此函数实现跳转，跳转至 setjmp() 函数设置的“跳转目标”，其中参数 env 就是setjmp() 函数设置的 env。此函数从 setjmp() 函数中返回，参数 val 作为从 setjmp() 函数返回时的返回值。

这些头文件一般跟随编译器提供，直接 include 即可使用。

#### Linux 头文件

主要用于 Linux 系统上进行特定的功能和操作，与 Linux 操作系统的特性和功能紧密相关。  
**<unistd.h>**  
提供了访问操作系统服务的函数，如文件操作、进程管理、系统调用等。

* 系统调用和进程控制：
  + fork：创建一个子进程。
  + exec 系列函数：在当前进程中执行新的程序。
  + exit：终止当前进程的执行。
  + getpid 和 getppid：获取当前进程和父进程的进程 ID。
* 文件和目录操作：
  + open 和 close：打开和关闭文件。
  + read 和 write：从文件中读取和写入数据。
  + lseek：移动文件指针。
  + access：检查文件的访问权限。
  + mkdir 和 rmdir：创建和删除目录。
* 系统资源管理：
  + sleep：使当前进程挂起指定的时间。
  + getcwd：获取当前工作目录。
  + chdir：改变当前工作目录。
  + gethostname：获取主机名。
  + getlogin：获取登录名。
* 进程间通信（IPC）：
  + pipe：创建一个管道用于进程间通信。
  + dup 和 dup2：复制文件描述符。
  + fork 和 exec 系列函数：在进程间传递状态和数据。

**<fcntl.h>**  
提供了对文件描述符的操作，如打开、关闭、读取和写入文件。

* 文件控制：
* open：打开文件并返回一个文件描述符。
* creat：创建一个新文件。
* close：关闭文件。

-文件状态标志（File Status Flags）：

* O\_RDONLY、O\_WRONLY、O\_RDWR：打开文件的读取、写入、读写模式。
* O\_APPEND：追加写入文件。
* O\_CREAT、O\_EXCL：创建文件并验证文件不存在。
* O\_TRUNC：将文件截断为零长度。

-文件控制操作：

* fcntl：对文件描述符进行各种控制操作。
* dup、dup2：复制文件描述符。

-文件锁定：

* flock：对文件进行简单的全局锁定。
* fcntl：使用 F\_SETLK、F\_SETLKW 等命令进行更复杂的文件锁定。

**<sys/types.h>**  
定义了一些基本的系统数据类型，如 size\_t、pid\_t 等。

* 数据类型定义：
* size\_t：用于表示对象的大小。
* ssize\_t：用于表示有符号的对象大小。
* time\_t：用于表示时间值。
* off\_t：用于表示文件偏移量。
* pid\_t：用于表示进程 ID。
* uid\_t 和 gid\_t：用于表示用户 ID 和组 ID。
* 文件类型和模式定义：
  + mode\_t：用于表示文件的权限模式。
  + dev\_t：用于表示设备的标识符。
* 其他常见的数据类型和宏：
  + int8\_t、int16\_t、int32\_t、int64\_t：用于表示固定大小的整数类型。
  + intptr\_t 和 uintptr\_t：用于表示指针类型的整数值。

**<sys/stat.h>**  
包含了文件状态相关的函数和宏，用于获取和修改文件的元数据。

* 文件访问权限：
* S\_IRUSR、S\_IWUSR、S\_IXUSR：用户（所有者）的读、写、执行权限。
* S\_IRGRP、S\_IWGRP、S\_IXGRP：用户组的读、写、执行权限。
* S\_IROTH、S\_IWOTH、S\_IXOTH：其他用户的读、写、执行权限。
* 文件类型和模式：
  + S\_IFMT：用于提取文件类型的位掩码。
  + S\_IFREG、S\_IFDIR、S\_IFLNK 等：表示不同文件类型的位掩码。
  + S\_ISREG()、S\_ISDIR()、S\_ISLNK() 等：用于检查文件类型的宏。
* 文件状态结构体：
  + struct stat：用于描述文件的状态信息，包括文件大小、权限、时间戳等。
* 文件属性获取函数：
  + stat、lstat：获取文件的状态信息。
  + fstat：获取打开文件的状态信息。

**<sys/socket.h>**

* 用于网络编程，提供了与套接字相关的函数和数据结构。
* 函数：
  + socket()：创建一个新的套接字
  + bind()：将套接字绑定到地址指定的端口号
  + listen()：监听套接字上的连接请求
  + accept()：接收客户端的连接请求
  + connect()：建立与远程服务器的连接
  + recv()：从套接字中接收数据
  + send()：将数据发送到套接字
  + close()：关闭一个套接字
  + shutdown()：关闭一个已建立的连接（这两个有区别，本文暂不不讲解）
* 常量：
  + AF\_INET：IPv4 协议的地址族，又称为 Internet 地址族
  + AF\_UNIX：Unix 协议的地址族
  + SOCK\_STREAM：提供面向连接、可靠的数据传输服务的套接字类型
  + SOCK\_DGRAM：提供面向无连接、不可靠的数据传输服务的套接字类型
  + IPPROTO\_TCP：TCP 协议的协议号
  + IPPROTO\_UDP：UDP 协议的协议号

**<sys/time.h>**  
提供了时间和日期相关的函数和定义，如获取当前时间、计时器等。

* 时间结构体：
* struct timeval：表示时间值，包括秒数和微秒数。
* 时间获取函数：
  + gettimeofday：获取当前时间。
* 定时器操作函数：
  + timeradd、timersub、timerclear：对时间进行加法、减法和清零操作。
* 时间转换函数：
  + gmtime、localtime：将时间转换为本地时间或协调世界时（UTC）时间。
  + strftime：将时间格式化为字符串。

**<sys/wait.h>**  
用于进程间通信，提供了等待子进程状态变化的函数和宏。

* 子进程状态常量：
  + WIFEXITED(status)：检查子进程是否正常退出。
  + WEXITSTATUS(status)：获取子进程的退出状态。
  + WIFSIGNALED(status)：检查子进程是否因信号而终止。
  + WTERMSIG(status)：获取导致子进程终止的信号编号。
  + WIFSTOPPED(status)：检查子进程是否暂停。
  + WSTOPSIG(status)：获取导致子进程暂停的信号编号。
* 等待子进程的状态改变：
  + wait：等待任意子进程的状态改变。
  + waitpid：等待指定子进程的状态改变。

**<pthread.h>**  
用于多线程编程，提供了线程相关的函数和定义。

* 线程管理函数：
  + pthread\_create：创建一个新线程。
  + pthread\_exit：终止当前线程。
  + pthread\_join：等待指定线程的结束。
  + pthread\_detach：将线程设置为可分离状态，使其在退出时自动释放资源。
* 线程同步函数：
  + pthread\_mutex\_init、pthread\_mutex\_destroy：初始化和销毁互斥锁。
  + pthread\_mutex\_lock、pthread\_mutex\_unlock：对互斥锁进行加锁和解锁操作。
  + pthread\_cond\_init、pthread\_cond\_destroy：初始化和销毁条件变量。
  + pthread\_cond\_wait、pthread\_cond\_signal：等待和通知条件变量的变化。
* 线程属性操作：
  + pthread\_attr\_init、pthread\_attr\_destroy：初始化和销毁线程属性。
  + pthread\_attr\_getdetachstate、pthread\_attr\_setdetachstate：获取和设置线程的分离状态。
* 线程局部存储：
  + pthread\_key\_create、pthread\_key\_delete：创建和删除线程局部存储键。
  + pthread\_setspecific、pthread\_getspecific：设置和获取线程局部存储的值。

**<signal.h>**  
用于信号处理，提供了处理进程间通信和异常处理的函数和定义。

* 信号处理函数：
  + signal：设置信号处理器（Signal Handler）。
  + raise：向当前进程发送指定信号。
* 信号常量：
  + SIGINT、SIGTERM、SIGQUIT：表示用户发送的中断、终止、退出信号。
  + SIGSEGV、SIGILL、SIGFPE：表示发生的错误信号，如段错误、非法指令、浮点异常等。
  + SIGUSR1、SIGUSR2：表示用户自定义的信号。
* 信号处理器的行为：
  + SIG\_DFL：默认的信号处理器。
  + SIG\_IGN：忽略该信号。
* **<dlfcn.h>**  
  用于动态加载库，提供了动态链接库的加载和使用的函数和宏。
* 动态库加载和卸载函数：
  + dlopen：打开一个动态链接库。
  + dlclose：关闭一个已打开的动态链接库。
* 符号获取函数：
  + dlsym：在动态链接库中查找指定名称的符号（函数、变量等）。
* 错误处理函数：
  + dlerror：获取最近一次动态链接库相关操作的错误信息。

这些头文件由 Linux 提供，直接 include 即可使用。

#### 链接库

链接库是头文件中函数具体的实现，他们已经被编译成二进制形式，当需要的时候链接即可。

* libc：C语言标准库的核心部分，提供了输入输出函数、字符串处理函数、内存管理函数等基本功能。会默认自动被链接到动态库。
* libm：数学库，包含了数学函数，如三角函数、指数函数、对数函数等。在需要的时候使用 -lm 链接。
* libpthread：线程库，提供了线程相关的函数和类型，用于多线程编程。在需要的时候使用 -lpthread 链接。
* libdl：动态链接库支持库，提供了动态链接库的相关函数和类型。在需要的时候使用 -ldl 链接。
* librt：实时库，提供了实时相关的函数和类型，如定时器函数、时间操作函数等。在需要的时候使用 -lrt 链接。
* libcrypt：密码库，提供了密码相关的函数和类型，如加密和解密函数。在需要的时候使用 -lcrypt 链接。

当编译时提示标准函数缺失，很可能就是没有链接相应的库。

<https://blog.csdn.net/xiaorange131/article/details/53292972>

<https://blog.csdn.net/weixin_43764974/article/details/130892554>

## 4. Linux 进程

进程是操作系统中给程序分配资源的基本单位，现在操作系统会同时运行多道程序，每个程序的集合体被命名为进程。一个进程就是一个程序在运行阶段的实体，程序在向操作系统申请资源的时候也是以进程的身份申请的。在现在操作系统中，每个进程都会被以内存分段的方式隔离起来，使得进程之间不能直接相互访问资源，不能相互影像，提高了进程的安全性。

以程序的视角来看，程序运行时，相当于单独有一个世界。这样大大减轻了编程人员的工作量，同时给计算机带来非常大的灵活性。

### Linux 进程模型

进程模型对进程所需的资源进行区分而起的名字叫做进程，在代码中就是建立进程的数据结构。

Linux进程通过一个 task\_struct 结构体描述，在 linux/sched.h 中定义，通过理解该结构，可更清楚的理解 linux 进程模型。包含进程所有信息的 task\_struct 数据结构是比较庞大的，但是该数据结构本身并不复杂，我们将它的所有域按其功能可做如下划分：

* 进程状态（State）
* 进程调度信息（Scheduling Information）
* 各种标识符（Identifiers）
* 进程通信有关信息（IPC：Inter\_Process Communication）
* 时间和定时器信息（Times and Timers）
* 进程链接信息（Links）
* 文件系统信息（File System）
* 虚拟内存信息（Virtual Memory）
* 页面管理信息（page）
* 对称多处理器（SMP）信息
* 和处理器相关的环境（上下文）信息（Processor Specific Context）
* 其他信息

进程在系统的中的基本索引是 PID，是一个 32 位无符号整数。为了保持在 16 位平台上的兼容，PID 默认被限制到最大为 32767，也就是最多允许建立 32767 个进程。这个值可以在编译时通过宏定义指定，或者在运行时使用 ulimit -u 进程设定。

**PID** 的基本数据结构

struct pid  
{  
 atomic\_t count;  
 unsigned int level;  
 /\* lists of tasks that use this pid \*/  
 struct hlist\_head tasks[PIDTYPE\_MAX];  
 struct rcu\_head rcu;  
 struct upid numbers[1];  
};  
\* What is struct pid?  
\*  
\* A struct pid is the kernel's internal notion of a process identifier.  
\* It refers to individual tasks, process groups, and sessions. While  
\* there are processes attached to it the struct pid lives in a hash  
\* table, so it and then the processes that it refers to can be found  
\* quickly from the numeric pid value. The attached processes may be  
\* quickly accessed by following pointers from struct pid.

每个进程有进程标识符、用户标识符、组标识符，如下：

* Pid : 进程标识符
* Uid、gid :用户标识符、组标识符
* Euid、egid : 有效用户标识符、有效组标识符
* Suid、sgid : 备份用户标识符、备份组标识符
* Fsuid、fsgid : 文件系统用户标识符、文件系统组标识符

在系统运行时可以使用 ps 命令查看进程信息：

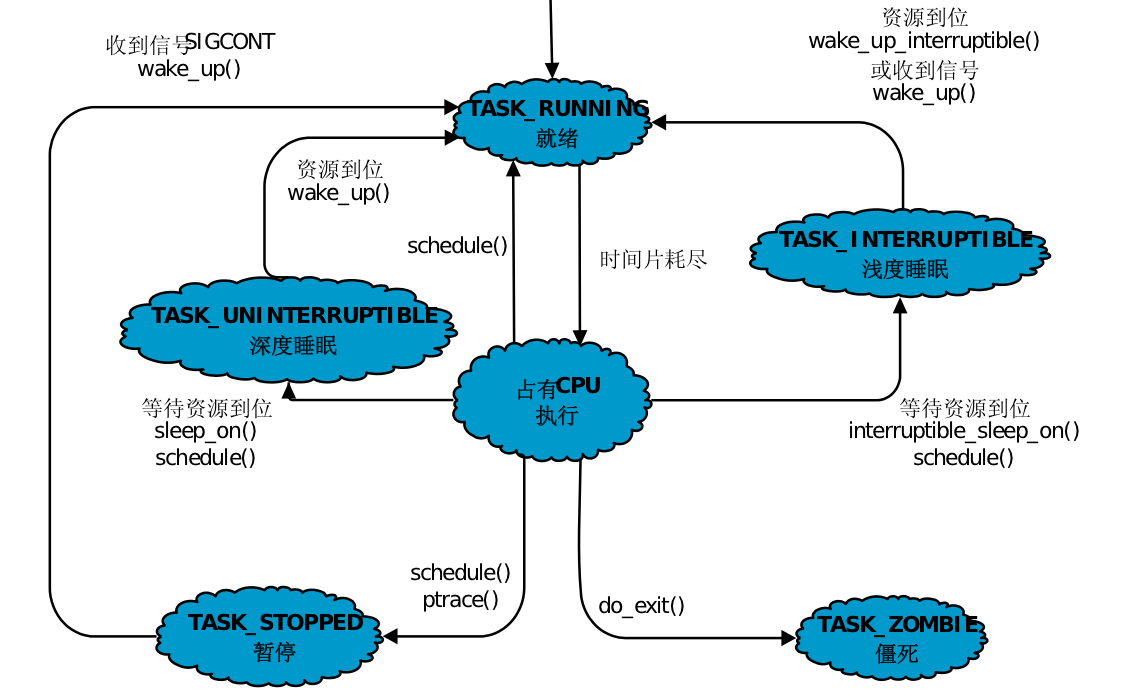
root@M5Core135:~# ps aux  
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND  
root 1 1.8 1.8 30888 8356 ? Ss 00:39 0:06 /sbin/init  
root 2 0.0 0.0 0 0 ? S 00:39 0:00 [kthreadd]  
root 3 0.0 0.0 0 0 ? I< 00:39 0:00 [rcu\_gp]  
root 4 0.0 0.0 0 0 ? I< 00:39 0:00 [rcu\_par\_gp]  
root 5 0.1 0.0 0 0 ? I 00:39 0:00 [kworker/0:0-ev  
root 6 0.1 0.0 0 0 ? I< 00:39 0:00 [kworker/0:0H-m  
root 7 0.4 0.0 0 0 ? I 00:39 0:01 [kworker/u2:0-e  
root 8 0.0 0.0 0 0 ? I< 00:39 0:00 [mm\_percpu\_wq]  
root 9 0.0 0.0 0 0 ? S 00:39 0:00 [rcu\_tasks\_kthr  
root 10 0.0 0.0 0 0 ? S 00:39 0:00 [rcu\_tasks\_trac  
root 11 0.0 0.0 0 0 ? S 00:39 0:00 [ksoftirqd/0]  
root 12 0.0 0.0 0 0 ? I 00:39 0:00 [rcu\_preempt]  
root 13 0.0 0.0 0 0 ? S 00:39 0:00 [kdevtmpfs]  
root 14 0.0 0.0 0 0 ? I< 00:39 0:00 [netns]  
root 15 0.0 0.0 0 0 ? I< 00:39 0:00 [inet\_frag\_wq]  
root 17 0.0 0.0 0 0 ? S 00:39 0:00 [oom\_reaper]  
root 18 0.0 0.0 0 0 ? I< 00:39 0:00 [writeback]  
root 19 0.0 0.0 0 0 ? S 00:39 0:00 [kcompactd0]  
root 39 0.0 0.0 0 0 ? I< 00:39 0:00 [kblockd]  
root 40 0.0 0.0 0 0 ? I< 00:39 0:00 [blkcg\_punt\_bio  
root 41 0.0 0.0 0 0 ? I< 00:39 0:00 [ata\_sff]  
root 42 0.0 0.0 0 0 ? I< 00:39 0:00 [edac-poller]  
root 43 0.0 0.0 0 0 ? I< 00:39 0:00 [devfreq\_wq]  
root 44 0.0 0.0 0 0 ? S 00:39 0:00 [watchdogd]  
root 45 0.8 0.0 0 0 ? I 00:39 0:02 [kworker/u2:1-w  
root 47 0.0 0.0 0 0 ? I< 00:39 0:00 [rpciod]  
root 48 0.0 0.0 0 0 ? I< 00:39 0:00 [kworker/u3:0]  
root 49 0.0 0.0 0 0 ? I< 00:39 0:00 [xprtiod]  
root 50 0.0 0.0 0 0 ? S 00:39 0:00 [kswapd0]  
root 52 0.0 0.0 0 0 ? I< 00:39 0:00 [nfsiod]  
root 53 0.1 0.0 0 0 ? I 00:39 0:00 [kworker/0:2-ev  
root 54 0.0 0.0 0 0 ? I 00:39 0:00 [kworker/0:3-ev  
root 55 0.0 0.0 0 0 ? I< 00:39 0:00 [optee\_pcpu\_not  
root 56 0.0 0.0 0 0 ? S 00:39 0:00 [hwrng]  
root 57 0.0 0.0 0 0 ? I< 00:39 0:00 [kstrp]  
root 58 0.0 0.0 0 0 ? I 00:39 0:00 [kworker/u2:3-e  
root 64 0.0 0.0 0 0 ? S 00:40 0:00 [irq/52-5a00100  
root 65 0.0 0.0 0 0 ? S 00:40 0:00 [irq/53-5a00100  
root 66 0.0 0.0 0 0 ? S 00:40 0:00 [card0-crtc0]  
root 68 0.0 0.0 0 0 ? I< 00:40 0:00 [stmmac\_wq]  
root 70 0.0 0.0 0 0 ? I< 00:40 0:00 [stmmac\_wq]  
root 72 0.0 0.0 0 0 ? S 00:40 0:00 [irq/60-4c00400  
root 73 0.0 0.0 0 0 ? S 00:40 0:00 [irq/62-ft6x36]  
root 74 0.0 0.0 0 0 ? S 00:40 0:00 [irq/63-4c00600  
root 75 0.0 0.0 0 0 ? S 00:40 0:00 [irq/41-stm\_the  
root 76 0.0 0.0 0 0 ? S 00:40 0:00 [irq/43-mmci-pl  
root 77 0.0 0.0 0 0 ? S 00:40 0:00 [irq/22-4400400  
root 78 0.0 0.0 0 0 ? S 00:40 0:00 [spi0]  
root 80 0.0 0.0 0 0 ? I< 00:40 0:00 [mmc\_complete]  
root 81 0.0 0.0 0 0 ? I< 00:40 0:00 [kworker/0:2H-k  
root 82 0.0 0.0 0 0 ? S 00:40 0:00 [jbd2/mmcblk0p1  
root 83 0.0 0.0 0 0 ? I< 00:40 0:00 [ext4-rsv-conve  
root 85 0.0 0.0 0 0 ? I< 00:40 0:00 [mld]  
root 86 0.0 0.0 0 0 ? I< 00:40 0:00 [ipv6\_addrconf]  
root 120 0.3 1.6 27156 7260 ? Ss 00:40 0:01 /lib/systemd/sy  
root 137 0.9 1.1 20076 5036 ? Ss 00:40 0:03 /lib/systemd/sy  
root 203 0.0 0.4 7152 2112 ? Ss 00:40 0:00 /usr/sbin/cron   
message+ 204 0.3 0.6 5832 3024 ? Ss 00:40 0:01 /usr/bin/dbus-d  
polkitd 206 0.0 1.1 35672 5400 ? Ssl 00:40 0:00 /usr/lib/polkit  
root 207 0.1 1.2 10888 5592 ? Ss 00:40 0:00 /lib/systemd/sy  
root 221 0.2 1.9 55352 8860 ? Ssl 00:40 0:00 /usr/sbin/Modem  
root 222 0.5 3.1 55316 14360 ? Ssl 00:40 0:01 /usr/sbin/Netwo  
root 224 0.0 1.7 10820 7736 ? Ss 00:40 0:00 /sbin/wpa\_suppl  
root 227 0.0 0.0 0 0 ? I< 00:40 0:00 [cfg80211]  
root 259 0.0 1.3 10584 6156 ? Ss 00:40 0:00 sshd: /usr/sbin  
root 263 0.0 0.5 7136 2528 ? S 00:40 0:00 bash -c cd /roo  
root 266 0.0 0.2 2128 1252 ? S 00:40 0:00 ./ts\_test  
root 267 0.0 0.3 4004 1464 tty1 Ss+ 00:40 0:00 /sbin/agetty -o  
root 295 0.0 0.0 0 0 ? S 00:40 0:00 [RTW\_CMD\_THREAD  
root 427 1.2 0.6 10072 3008 ttySTM0 Ss 00:44 0:01 /bin/login -p -  
root 428 0.0 0.0 0 0 ? S 00:44 0:00 [irq/57-4400300  
root 438 24.4 1.6 12420 7400 ? Ss 00:45 0:01 /lib/systemd/sy  
root 442 0.0 0.8 32304 3784 ? S 00:45 0:00 (sd-pam)  
root 460 0.5 0.6 7236 2776 ttySTM0 S 00:45 0:00 -bash  
root 465 300 0.8 10660 3660 ttySTM0 R+ 00:45 0:00 ps aux

完整的描述一个进程的结构体需要很长的文字和代码说明，但是我们实际使用时，更多只用了解进程的状态，生命周期以及操作系统如何调度分配资源即可。

### Linux 进程状态

进程是程序运行的集合体，操作系统的资源是共享的，所以进程不能无休止的占用 CPU，同时进程也不是一直在运行。于是操作系统对进程进程状态进行了划分:

* TASK\_RUNNING （可运行状态）：处于这种状态的进程，要么正在运行、要么正准备运行。正在运行的进程就是当前进程（由current所指向的进程），而准备运行的进程只要得到CPU就可以立即投入运行，CPU是这些进程唯一等待的系统资源。在任务管理器中用 R 标识。
* TASK\_INTERRUPTIBLE（可中断的等待状态）：表示进程被阻塞（睡眠），直到某个条件达成，进程的状态就被设置为TASK\_RUNNING。处于该状态的进程正在等待某个事件（event）或某个资源，而被挂起。对应的task\_struct结构被放入对应事件的等待队列中。处于可中断等待态的进程可以被信号（外部中断触发或者其他进程触发）唤醒，如果收到信号，该进程就从等待状态进入可运行状态，并且加入到运行队列中，等待被调度。在任务管理器中用 S 标识。
* TASK\_UNINTERRUPTIBLE（不可中断的等待状态）：该状态与 TASK\_INTERRUPTIBLE 状态类似，也表示进程被阻塞，处于睡眠状态。当进程等待的某些条件被满足了之后，内核也会将该进程的状态设置为 TASK\_RUNNING。但是，处于这个状态下的进程不能在接收到某个信号之后立即被唤醒。这时该状态与 TASK\_INTERRUPTIBLE 状态唯一的区别。在任务管理器中用 D 标识。
* \_\_TASK\_STOPPED（暂停状态）：此时的进程暂时停止运行来接受某种特殊处理。通常当进程接收到SIGSTOP、SIGTSTP、SIGTTIN或 SIGTTOU信号后就处于这种状态。例如，正接受调试的进程就处于这种状态。在任务管理器中用 T 标识。
* \_\_TASK\_TRACED（跟踪状态）：当前进程正在被另一个进程所监视。在任务管理器中用 T 标识。
* EXIT\_ZOMBIE（僵死状态）：进程虽然已经终止，但由于某种原因，父进程还没有执行wait()系统调用，终止进程的信息也还没有回收。顾名思义，处于该状态的进程就是死进程，这种进程实际上是系统中的垃圾，必须进行相应处理以释放其占用的资源。在任务管理器中用 Z 标识。
* EXIT\_DEAD：一个进程的最终状态。在任务管理器中用 X 标识。



### Linux 进程关系

Linux 中的进程主要是父子关系，除了 1 号进程，其他的进程都是由父进程分裂出来的，在继承关系中属于父子继承。

**1号 init 进程**

1 号进程是内核创建出的第一个用户进程 init，也是所有用户进程的祖先，1 号进程通常的任务是按照初始化脚本完成系统初始化和服务进程的创建，然后就进入回收孤儿进程的进程资源状态。现在的发行版使用了更现代的 init 程序 systemd。systemd 相比之前的 init 进程，提供和接管了更多的系统级服务，如进程组管理，守护进程监控，系统事件动作等等。

**僵尸进程**

进程被父进程创建出来并完成任务退出后，进程占用的资源还停留在系统中，此时需要父进程使用 wait 方法进行子进程资源回收，同时获取子进程的返回值。而没有及时回收资源的已经退出的进程被称为僵尸进程。

僵尸进程占用系统的资源，如果父进程不断的产生僵尸进程，那将会造成系统资源的浪费，从而影响其他进程获取资源。

防止僵尸进程的办法就是及时使用 wait 方法和 waitpid 方法进行资源回收。

**孤儿进程**

孤儿进程是相对于父进程而言的，父进程创建了子进程，但是父进程因为某种原因退出了，然后被父进程的父进程回收了。这时，子进程就脱离了父子关系，成为了孤儿进程。

孤儿进程会被系统中的 1 号 init 进程领养，也就是由 1 号 init 进程来完成孤儿进程结束后的资源回收。

### Linux 进程调度

进程调度是为进程分配计算资源的行为，不同调度算法带来实际体验的差异是很大的，特别是在系统负载比较大的情况下。

Linux 的调度算法经过了下面的几个阶段：

第一阶段：O(n)。每次遍历所有的任务，相当于多个任务一个一个的运行。这种调度算法在进程比较多的情况体验是非常差的。

第二阶段：O(1)。采用一个进程动态优先级的数组队列，算法将会按照优先级顺序查找系统中就绪的进程，然后参与调度。这个调度器很快就被后面的 CFS 公平调度器取代了。

目前：CFS 全然公平调度器。CFS 调度程序使用安抚（ appeasement ）策略确保公平性。当某个任务进入运行队列后，将记录当前时间，当某个进程等待 CPU 时，将对这个进程的 wait\_runtime 值加一个数，这个数取决于运行队列当前的进程数。当执行这些计算时，也将考虑不同任务的优先级值。 将这个任务调度到 CPU 后，它的 wait\_runtime 值开始递减，当这个值递减到其他任务成为红黑树的最左侧任务时，当前任务将被抢占。通过这种方式，CFS 努力实现一种理想 状态，即 wait\_runtime 值为 0。  
**CFS调度器类fair\_sched\_class**  
CFS完全公平调度器的调度器类叫 fair\_sched\_class , 其定义在 kernel/sched/fair.c , line 8521, 它是我们熟知的是 struct sched\_class 调度器类类型, 将我们的 CFS 调度器与一些特定的函数关联起来.

/\*  
 \* All the scheduling class methods:  
 \*/  
const struct sched\_class fair\_sched\_class = {  
 .next = &idle\_sched\_class, /\* 下个优先级的调度类, 所有的调度类通过next链接在一个链表中\*/  
 .enqueue\_task = enqueue\_task\_fair,  
 .dequeue\_task = dequeue\_task\_fair,  
 .yield\_task = yield\_task\_fair,  
 .yield\_to\_task = yield\_to\_task\_fair,  
  
 .check\_preempt\_curr = check\_preempt\_wakeup,  
  
 .pick\_next\_task = pick\_next\_task\_fair,  
 .put\_prev\_task = put\_prev\_task\_fair,  
  
#ifdef CONFIG\_SMP  
 .select\_task\_rq = select\_task\_rq\_fair,  
 .migrate\_task\_rq = migrate\_task\_rq\_fair,  
  
 .rq\_online = rq\_online\_fair,  
 .rq\_offline = rq\_offline\_fair,  
  
 .task\_waking = task\_waking\_fair,  
 .task\_dead = task\_dead\_fair,  
 .set\_cpus\_allowed = set\_cpus\_allowed\_common,  
#endif  
  
 .set\_curr\_task = set\_curr\_task\_fair,  
 .task\_tick = task\_tick\_fair,  
 .task\_fork = task\_fork\_fair,  
  
 .prio\_changed = prio\_changed\_fair,  
 .switched\_from = switched\_from\_fair,  
 .switched\_to = switched\_to\_fair,  
  
 .get\_rr\_interval = get\_rr\_interval\_fair,  
  
 .update\_curr = update\_curr\_fair,  
  
#ifdef CONFIG\_FAIR\_GROUP\_SCHED  
 .task\_move\_group = task\_move\_group\_fair,  
#endif  
};

下面就这调度器类成员进行介绍：

* enqueue\_task：向就绪队列中添加一个进程, 某个任务进入可运行状态时，该函数将得到调用。它将调度实体（进程）放入红黑树中，并对 nr\_running 变量加 1。
* dequeue\_task：将一个进程从就就绪队列中删除, 当某个任务退出可运行状态时调用该函数，它将从红黑树中去掉对应的调度实体，并从 nr\_running 变量中减 1。
* yield\_task：在进程想要资源放弃对处理器的控制权的时, 可使用在sched\_yield系统调用, 会调用内核API yield\_task完成此工作. compat\_yield sysctl 关闭的情况下，该函数实际上执行先出队后入队；在这种情况下，它将调度实体放在红黑树的最右端。
* check\_preempt\_curr：该函数将检查当前运行的任务是否被抢占。在实际抢占正在运行的任务之前，CFS 调度程序模块将执行公平性测试。这将驱动唤醒式（wakeup）抢占。
* pick\_next\_task：该函数选择接下来要运行的最合适的进程。
* put\_prev\_task：用另一个进程代替当前运行的进程。
* set\_curr\_task：当任务修改其调度类或修改其任务组时，将调用这个函数。
* task\_tick：在每次激活周期调度器时, 由周期性调度器调用, 该函数通常调用自 time tick 函数；它可能引起进程切换。这将驱动运行时（running）抢占。
* task\_new：内核调度程序为调度模块提供了管理新任务启动的机会, 用于建立fork系统调用和调度器之间的关联, 每次新进程建立后, 则用new\_task通知调度器, CFS 调度模块使用它进行组调度，而用于实时任务的调度模块则不会使用这个函数。

**CFS的就绪队列**

就绪队列是全局调度器许多操作的起点, 但是进程并不是由就绪队列直接管理的, 调度管理是各个调度器的职责, 因此在各个就绪队列中嵌入了特定调度类的子就绪队列(cfs的顶级调度就队列struct cfs\_rq， 实时调度类的就绪队列struct rt\_rq和deadline调度类的就绪队列struct dl\_rq)。

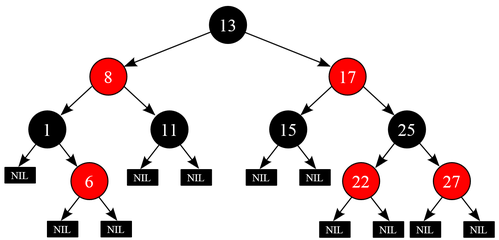
/\* CFS-related fields in a runqueue \*/  
/\* CFS调度的运行队列，每个CPU的rq会包含一个cfs\_rq，而每个组调度的sched\_entity也会有自己的一个cfs\_rq队列 \*/  
struct cfs\_rq {  
 /\* CFS运行队列中所有进程的总负载 \*/  
 struct load\_weight load;  
 /\*  
 \* nr\_running: cfs\_rq中调度实体数量  
 \* h\_nr\_running: 只对进程组有效，其下所有进程组中cfs\_rq的nr\_running之和  
 \*/  
 unsigned int nr\_running, h\_nr\_running;  
  
 u64 exec\_clock;  
  
 /\*  
 \* 当前CFS队列上最小运行时间，单调递增  
 \* 两种情况下更新该值:   
 \* 1、更新当前运行任务的累计运行时间时  
 \* 2、当任务从队列删除去，如任务睡眠或退出，这时候会查看剩下的任务的vruntime是否大于min\_vruntime，如果是则更新该值。  
 \*/  
  
 u64 min\_vruntime;  
#ifndef CONFIG\_64BIT  
 u64 min\_vruntime\_copy;  
#endif  
 /\* 该红黑树的root \*/  
 struct rb\_root tasks\_timeline;  
 /\* 下一个调度结点(红黑树最左边结点，最左边结点就是下个调度实体) \*/  
 struct rb\_node \*rb\_leftmost;  
  
 /\*  
 \* 'curr' points to currently running entity on this cfs\_rq.  
 \* It is set to NULL otherwise (i.e when none are currently running).  
 \* curr: 当前正在运行的sched\_entity（对于组虽然它不会在cpu上运行，但是当它的下层有一个task在cpu上运行，那么它所在的cfs\_rq就把它当做是该cfs\_rq上当前正在运行的sched\_entity）  
 \* next: 表示有些进程急需运行，即使不遵从CFS调度也必须运行它，调度时会检查是否next需要调度，有就调度next  
 \*  
 \* skip: 略过进程(不会选择skip指定的进程调度)  
 \*/  
 struct sched\_entity \*curr, \*next, \*last, \*skip;  
  
#ifdef CONFIG\_SCHED\_DEBUG  
 unsigned int nr\_spread\_over;  
#endif  
  
#ifdef CONFIG\_SMP  
 /\*  
 \* CFS load tracking  
 \*/  
 struct sched\_avg avg;  
 u64 runnable\_load\_sum;  
 unsigned long runnable\_load\_avg;  
#ifdef CONFIG\_FAIR\_GROUP\_SCHED  
 unsigned long tg\_load\_avg\_contrib;  
#endif  
 atomic\_long\_t removed\_load\_avg, removed\_util\_avg;  
#ifndef CONFIG\_64BIT  
 u64 load\_last\_update\_time\_copy;  
#endif  
  
#ifdef CONFIG\_FAIR\_GROUP\_SCHED  
 /\*  
 \* h\_load = weight \* f(tg)  
 \*  
 \* Where f(tg) is the recursive weight fraction assigned to  
 \* this group.  
 \*/  
 unsigned long h\_load;  
 u64 last\_h\_load\_update;  
 struct sched\_entity \*h\_load\_next;  
#endif /\* CONFIG\_FAIR\_GROUP\_SCHED \*/  
#endif /\* CONFIG\_SMP \*/  
  
#ifdef CONFIG\_FAIR\_GROUP\_SCHED  
 /\* 所属于的CPU rq \*/  
 struct rq \*rq; /\* cpu runqueue to which this cfs\_rq is attached \*/  
  
 /\*  
 \* leaf cfs\_rqs are those that hold tasks (lowest schedulable entity in  
 \* a hierarchy). Non-leaf lrqs hold other higher schedulable entities  
 \* (like users, containers etc.)  
 \*  
 \* leaf\_cfs\_rq\_list ties together list of leaf cfs\_rq's in a cpu. This  
 \* list is used during load balance.  
 \*/  
 int on\_list;  
 struct list\_head leaf\_cfs\_rq\_list;  
 /\* 拥有该CFS运行队列的进程组 \*/  
 struct task\_group \*tg; /\* group that "owns" this runqueue \*/  
  
#ifdef CONFIG\_CFS\_BANDWIDTH  
 int runtime\_enabled;  
 u64 runtime\_expires;  
 s64 runtime\_remaining;  
  
 u64 throttled\_clock, throttled\_clock\_task;  
 u64 throttled\_clock\_task\_time;  
 int throttled, throttle\_count;  
 struct list\_head throttled\_list;  
#endif /\* CONFIG\_CFS\_BANDWIDTH \*/  
#endif /\* CONFIG\_FAIR\_GROUP\_SCHED \*/  
};

成员描述：

* nr\_running：队列上可运行进程的数目。
* load：就绪队列上可运行进程的累计负荷权重。
* min\_vruntime：跟踪记录队列上所有进程的最小虚拟运行时间. 这个值是实现与就绪队列相关的虚拟时钟的基础。
* tasks\_timeline：用于在按时间排序的红黑树中管理所有进程。
* rb\_leftmost：总是设置为指向红黑树最左边的节点, 即需要被调度的进程. 该值其实可以可以通过病例红黑树获得, 但是将这个值存储下来可以减少搜索红黑树花费的平均时间。
* curr：当前正在运行的sched\_entity（对于组虽然它不会在cpu上运行，但是当它的下层有一个task在cpu上运行，那么它所在的cfs\_rq就把它当做是该cfs\_rq上当前正在运行的sched\_entity。
* next：表示有些进程急需运行，即使不遵从CFS调度也必须运行它，调度时会检查是否next需要调度，有就调度next。
* skip：略过进程(不会选择skip指定的进程调度)。

**红黑树**

红黑树 (Red–black tree) 是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构。在 CFS 调度器中将 sched\_entity 存储在以时间为顺序的红黑树中，vruntime 最低的进程存储在树的左侧，vruntime 最高的进程存储在树的右侧。 为了公平，调度器每次都会选取红黑树最左端的节点调度以便保持公平性。这样，整颗红黑树最左侧的进程就被给予时间运行了，树的内容从右侧迁移到左侧以保持公平。同时，红黑树的时间复杂度为 O(log *n*)，可以快速高效地执行插入或是删除操作。如下图：



1.<https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/382503?fr=aladdin>

2.<http://blog.chinaunix.net/uid-24203478-id-3130713.html>

3.<https://blog.csdn.net/jinkang_zhao/article/details/71367924>

4.<https://blog.csdn.net/gatieme/article/details/51701149>

5.<https://blog.csdn.net/gatieme/article/details/52067518>

### 进程的创建与运行

#### 进程的创建

只有一个进程的 C 语言代码:

// file : process.c  
#include<stdio.h>  
#include<unistd.h>  
int main(int argc, char argv[])   
{  
 printf("i am process...pid:%d,ppid:%d\n",getpid(),getppid());  
 return 0;  
}

然后用 GCC 编译这个源码并执行，查看效果：

root@M5Core135:~/tmp# gcc process.c -o process  
root@M5Core135:~/tmp# ./process   
i am process...pid:581,ppid:549

可以看出，这个程序执行的时候相当于一个进程了，然后打印出了自己进程的 pid 和 ppid 。

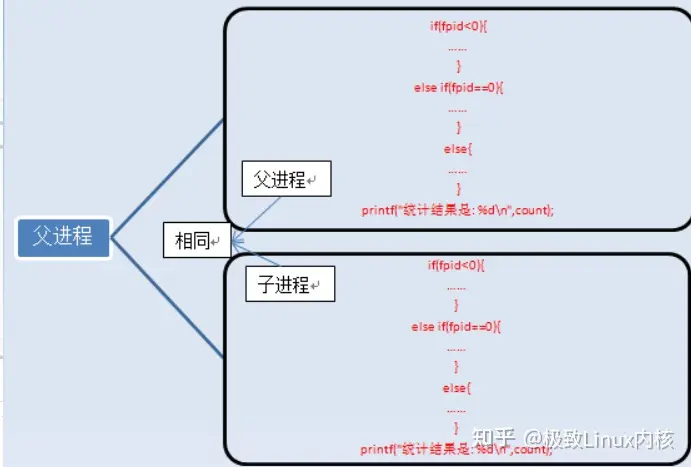
创建一个子进程的 C 语言代码：

// file : fork\_process.c  
#include <stdio.h>  
#include <unistd.h>  
int main(int argc, char argv[])  
{  
 pid\_t id = fork(); // 创建子进程  
  
 if (id == 0)  
 {  
 printf("i am process...child---pid:%d,ppid:%d\n", getpid(), getppid());  
 sleep(1);  
 }  
 else if (id > 0)  
 {  
 printf("i am process..father---pid:%d,ppid:%d\n", getpid(), getppid());  
 sleep(1);  
 }  
 else  
 {  
 perror("fork error!");  
 }  
 return 0;  
}

编译执行后查看效果：

root@M5Core135:~/tmp# gcc fork\_process.c -o fork\_process  
root@M5Core135:~/tmp# ./fork\_process   
i am process..father---pid:628,ppid:549  
i am process...child---pid:629,ppid:628

这段 C 程序是使用 fork 函数来创建一个子进程，fork 是一个在 Unix/Linux 操作系统中创建新进程的系统调用。这个系统调用会创建一个当前进程的副本（子进程）。这两个进程（父进程和子进程）在执行的瞬间是相同的，但是它们有着不同的进程 ID（PID）。在父进程返回的是子进程的 pid 号，而在子进程中会返回 0 。



创建一个守护进程（孤儿进程）：

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <unistd.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
  
void daemond()  
{  
 // 父进程创建子进程,然后退出,形式上脱离控制终端  
 pid\_t pid = fork();  
 if (pid > 0)  
 {  
 exit(0);  
 }  
 // 子进程创建一个新的会话  
 setsid();  
 // 改变当前目录为根目录  
 chdir("/");  
 // 设置umask掩码,防止从父进程继承的过来的屏蔽字拒绝某些权限  
 umask(0);  
 // 关闭文件描述符,节省系统资源  
 // STDIN\_FILENO就是标准输入设备（一般是键盘）的文件描述符  
 // 写入/dev/null的东西会被系统丢掉  
 close(STDIN\_FILENO);  
 // 再一次的保护  
 open("/dev/null", O\_RDWR);  
 dup2(0, 1);  
 dup2(0, 2);  
 // 守护进程的核心逻辑  
 // 退出  
}  
  
int main(int argc, char argv[])  
{  
 daemond();  
 while (1)  
 ;  
 return 0;  
}

#### 进程的输入输出

进程的输入输出其实就是程序的基本输入输出，对于进程来讲，操作系统给进程提供的输入输出有多个通道。

**标准输入输出**

* stdin **标准输入**是程序默认的输入流。它通常关联到键盘或其他输入设备。程序可以从 stdin 中读取数据。在大多数编程语言中，可以使用标准库中的函数（例如，C 语言中的 scanf 或 Python 中的 input）来从 stdin 读取输入。
* stdout **标准输出**是程序默认的输出流。它通常关联到终端窗口或其他输出设备。程序可以通过 stdout 输出结果，供用户或其他程序查看。在大多数编程语言中，可以使用标准库中的函数（例如，C 语言中的 printf 或 Python 中的 print）将输出发送到 stdout。也可使用 fprintf(stdout, ...) 直接将输出发送到 stdout。
* stderr **标准错误输出**用于输出错误消息和诊断信息，通常关联到终端窗口或日志文件。与 stdout 类似，程序可以使用标准库中的函数将错误消息发送到 stderr。在大多数情况下，错误消息应该被输出到 stderr，以便于与正常输出区分。

下面是一个使用 C 语言的简单示例，演示了标准输入输出：

// file : stdio\_test.c  
#include <stdio.h>  
  
int main(int argc, char argv[]) {  
 int number;  
  
 // 从标准输入读取数据  
 printf("Enter a number: ");  
 scanf("%d", &number);  
  
 // 将结果输出到标准输出  
 printf("You entered: %d\n", number);  
  
 // 将结果输出到标准错误输出  
 fprintf(stderr, "This is an error message\n");  
   
   
 return 0;  
}

编译执行：

root@M5Core135:~/tmp# gcc stdio\_test.c -o stdio\_test  
root@M5Core135:~/tmp# ./stdio\_test   
Enter a number: 1  
You entered: 1  
This is an error message

**命令行参数**

命令行参数就是从命令行传入的参数，对于 C 程序入口 main 函数是这样写的 int main(int argc, char argv[])其实里面的参数保存的就是命令行传参。

* argc（argument count）是一个整数，表示命令行参数的数量。
* argv（argument vector）是一个字符指针数组，每个指针指向一个字符串，即命令行参数的实际值。

以下是一个简单的示例，演示如何在 C 语言程序中使用命令行参数：

// file : cmdline\_test.c  
#include <stdio.h>  
  
int main(int argc, char \*argv[]) {  
 // 打印程序的名称  
 printf("Program name: %s\n", argv[0]);  
  
 // 打印命令行参数的数量  
 printf("Number of arguments: %d\n", argc - 1);  
  
 // 打印所有命令行参数  
 for (int i = 1; i < argc; i++) {  
 printf("Argument %d: %s\n", i, argv[i]);  
 }  
  
 return 0;  
}

编译运行：

root@M5Core135:~/tmp# gcc cmdline\_test.c -o cmdline\_test  
root@M5Core135:~/tmp# ./cmdline\_test a b c=2 -d1  
Program name: ./cmdline\_test  
Number of arguments: 4  
Argument 1: a  
Argument 2: b  
Argument 3: c=2  
Argument 4: -d1

显然，命令行参数只有输入，没有输出。

**环境变量**

对于一个程序来说，进程启动的时候是有一些默认的环境参数的，这些环境参数也能被程序所读取。环境变量是一个选项和选项对应的参数值，程序读取后就能知道当前执行环境是什么样子的。

在 C 语言中，可以通过标准库 <stdlib.h> 中的 getenv 函数来获取环境变量的值。环境变量是在操作系统中存储配置信息和其他系统参数的一种机制。

以下是一个简单的例子，演示如何使用 getenv 函数获取和输出环境变量的值：

// file : env\_test.c  
#include <stdio.h>  
#include <stdlib.h>  
  
// 外部变量，定义在 <unistd.h> 或 <stdlib.h> 中  
extern char \*\*environ;  
  
int main() {  
   
 // 遍历环境变量  
 for (char \*\*env = environ; \*env != NULL; env++) {  
 printf("%s\n", \*env);  
 }  
   
 // 获取环境变量 PATH 的值  
 char \*path\_value = getenv("PATH");  
  
 if (path\_value != NULL) {  
 printf("PATH environment variable: %s\n", path\_value);  
 } else {  
 printf("PATH environment variable not found\n");  
 }  
  
 return 0;  
}

编译运行:

root@M5Core135:~/tmp# gcc env\_test.c -o env\_test  
root@M5Core135:~/tmp# ./env\_test   
SHELL=/bin/bash  
LC\_ADDRESS=zh\_CN.UTF-8  
LC\_NAME=zh\_CN.UTF-8  
LC\_MONETARY=zh\_CN.UTF-8  
PWD=/root/tmp  
LOGNAME=root  
XDG\_SESSION\_TYPE=tty  
MOTD\_SHOWN=pam  
HOME=/root  
LANG=en\_US.UTF-8  
LC\_PAPER=zh\_CN.UTF-8  
SSH\_CONNECTION=192.168.31.156 44602 192.168.31.172 22  
LCD\_FRAMEBUFFER=/dev/fb1  
XDG\_SESSION\_CLASS=user  
TERM=xterm-256color  
LC\_IDENTIFICATION=zh\_CN.UTF-8  
USER=root  
SHLVL=1  
LC\_TELEPHONE=zh\_CN.UTF-8  
LC\_MEASUREMENT=zh\_CN.UTF-8  
XDG\_SESSION\_ID=c3  
LD\_LIBRARY\_PATH=/usr/local/lib  
LC\_CTYPE=zh\_CN.UTF-8  
XDG\_RUNTIME\_DIR=/run/user/0  
SSH\_CLIENT=192.168.31.156 44602 22  
LC\_TIME=zh\_CN.UTF-8  
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
DBUS\_SESSION\_BUS\_ADDRESS=unix:path=/run/user/0/bus  
SSH\_TTY=/dev/pts/0  
LC\_NUMERIC=zh\_CN.UTF-8  
\_=./env\_test  
OLDPWD=/root  
PATH environment variable: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

由于环境变量没有统一的描述，所以关于环境变量的每个条目的具体含义可以去搜索引擎的自行搜索，在此不做详细注解。

## 5. 守护进程

### 守护进程概述

守护进程，也就是通常所说的 Daemon 进程，是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性的执行某种任务或等待处理某些发生的事件。守护进程常常在系统启动时开始运行，在系统关闭时终止。

Linux 系统有很多守护进程，大多数服务都是用守护进程实现的。例如常见的常见的守护进程包括系统日志进程syslogd、 web 服务器 httpd、邮件服务器 sendmail 和数据库服务器 mysqld 等。

守护进程一部分是由 init 进程直接创建而来，一部分是由应用进程传建的孤儿进程被 init 进程领养后称为守护进程。

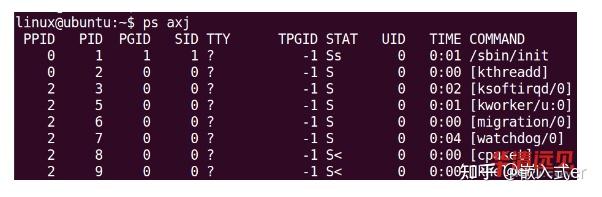
### 进程与终端

在Linux中，每一个系统与用户进行交流的界面称为终端。从该终端开始运行的进程都会依附于这个终端，这个终端称为这些进程的控制终端。当控制终端被关闭时，相应的进程都会被自动关闭。

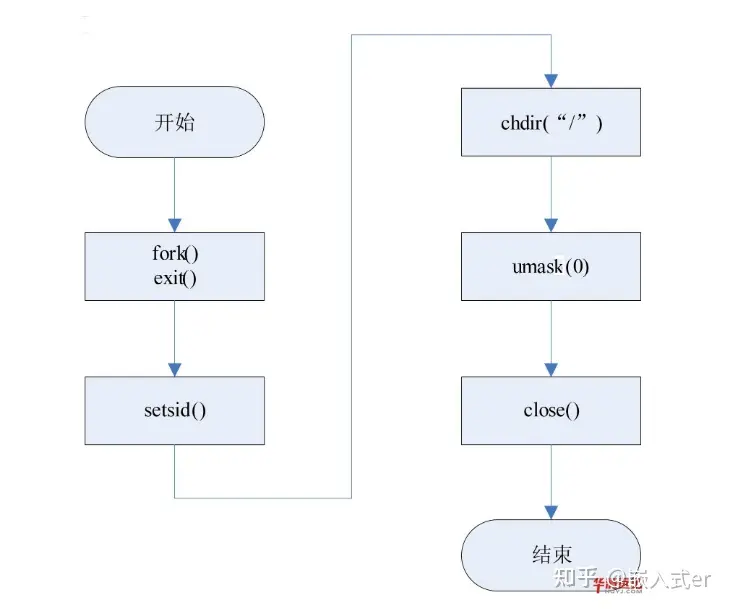
守护进程能够突破这种限制，它从开始运行，直到整个系统关闭才会退出。如果想让某个进程不会因为用户或终端的变化而受到影响，就必须把这个进程变成一个守护进程。

### 查看守护进程

命令： ps axj



### Linux守护进程编写



进程被创建出来后，如果不进行保护性设置，很容易就会受到终端或者父进程的影响而退出工作。创建守护进程的步骤其实就是如何创建出孤儿进程，并进行保护的过程。

第一步，创建孤儿进程。`

// 创建子进程  
pid\_t child\_pid = fork();  
  
// 检查 fork 是否成功  
if (child\_pid < 0) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
}  
  
// 父进程退出，使子进程成为孤儿进程  
if (child\_pid > 0) {  
 exit(EXIT\_SUCCESS);  
}

第一步完成以后，子进程就在形式上做到了与控制终端的脱离

由于父进程已经先于子进程退出，子进程变成孤儿进程

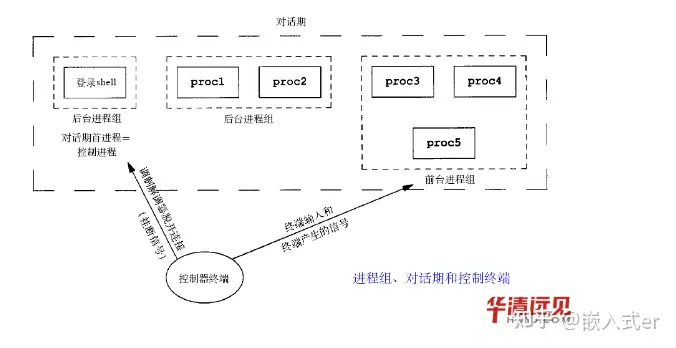
由于守护进程是脱离控制终端的，因此，完成第一步后就会在 shell 终端里造成一程序已经运行完毕的假象。之后的所有后续工作都在子进程中完成，而用户在 shell 终端里则可以执行其他的命令，从而在形式上做到了与控制终端的脱离。

第二步，在子进程中创建新会话。

进程被组合成进程组，进程组号 GID 就是进程组长的进程号 PID 。登录会话可以包含多个进程组。这些进程组共享一个控制终端。这个控制终端通常是创建进程的登录终端。控制终端，登录会话和进程组通常是从父进程继承下来的。我们的目的就是要摆脱它们，使之不受它们的影响。方法是在孤儿进程的基础上，调用相关函数使进程成为会话组长。

* 进程组：一个或多个进程的集合。进程组由进程组ID来唯一标识。每个进程组都有一个组长进程，进程组ID就是组长进程的进程号。
* 会话期：会话组是一个或多个进程组的集合
* 终端：面向用户交互的程序。

setsid 函数用于创建一个新的会话，并使得当前进程成为新会话组的组长 setsid 函数能够使进程完全独立出来，从而脱离所有其他进程的控制。



// 创建一个新会话，并使当前进程成为新会话的领头进程  
if (setsid() == -1) {  
 perror("setsid");  
 exit(EXIT\_FAILURE);  
}

第三步：修改当前工作目录。

在进程运行过程中，当前目录所在的文件系统是不能卸载的。所以会占用启动目录，改变工作目录的原因就是避免占用目录资源。所用的函数是 chdir ，用于改变工作目录。

通常的做法是让“/”或”/tmp”作为守护进程的当前工作目录 ，chdir(“/”); 或 chdir(“/tmp”);。

// 切换工作目录到根目录，防止影响文件系统的卸载  
if (chdir("/") == -1) {  
 perror("chdir");  
 exit(EXIT\_FAILURE);  
}

第四步：重设置文件权限掩码。

文件权限掩码是指文件权限中被屏蔽掉的对应位。把文件权限掩码设置为0，可以增加该守护进程的灵活性。设置文件权限掩码的函数umask();

umask(0);

第五步：关闭文件描述符。

新建的子进程会从父进程那里继承所有已经打开的文件。在创建完新的会话后，守护进程已经脱离任何控制终端，应当关闭用不到的文件。这些被打开的文件可能永远不会被守护进程读或写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法卸载

从终端输入的字符不可能达到守护进程，守护进程中用常规的方法 (如printf) 输出的字符也不可能在终端上显示出来。所以，文件描述符为 0、1 和 2 的三个文件(对应标准输入、标准输出和标准错误这三个流)已经失去了存在的意义，也应被关闭。

// 关闭不需要的文件描述符，防止影响文件操作  
close(STDIN\_FILENO);  
close(STDOUT\_FILENO);  
close(STDERR\_FILENO);

完整代码：

#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
 // 创建子进程  
 pid\_t child\_pid = fork();  
  
 // 检查 fork 是否成功  
 if (child\_pid < 0) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 父进程退出，使子进程成为孤儿进程  
 if (child\_pid > 0) {  
 exit(EXIT\_SUCCESS);  
 }  
  
 // 在子进程中  
  
 // 创建一个新会话，并使当前进程成为新会话的领头进程  
 if (setsid() == -1) {  
 perror("setsid");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 切换工作目录到根目录，防止影响文件系统的卸载  
 if (chdir("/") == -1) {  
 perror("chdir");  
 exit(EXIT\_FAILURE);  
 }  
   
 // 重设置文件权限掩码  
 umask(0);  
   
 // 关闭不需要的文件描述符，防止影响文件操作  
 close(STDIN\_FILENO);  
 close(STDOUT\_FILENO);  
 close(STDERR\_FILENO);  
  
 // 执行守护进程的核心逻辑，这里用 sleep 模拟  
 while (1) {  
 sleep(1);  
 }  
  
 return 0;  
}

<https://zhuanlan.zhihu.com/p/657482216>

## 6. 线程

进程和线程的区别:

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。线程（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。进程之间的代码段是相互隔离独立的，他们之间不会被直接影响。线程是存在进程中的多个同时执行的代码段，他们共享进程内的所有资源。线程相当于轻量进程。其实在 Linux 内核中，创建线程时，其实就是相当于创建一个共享的进程。

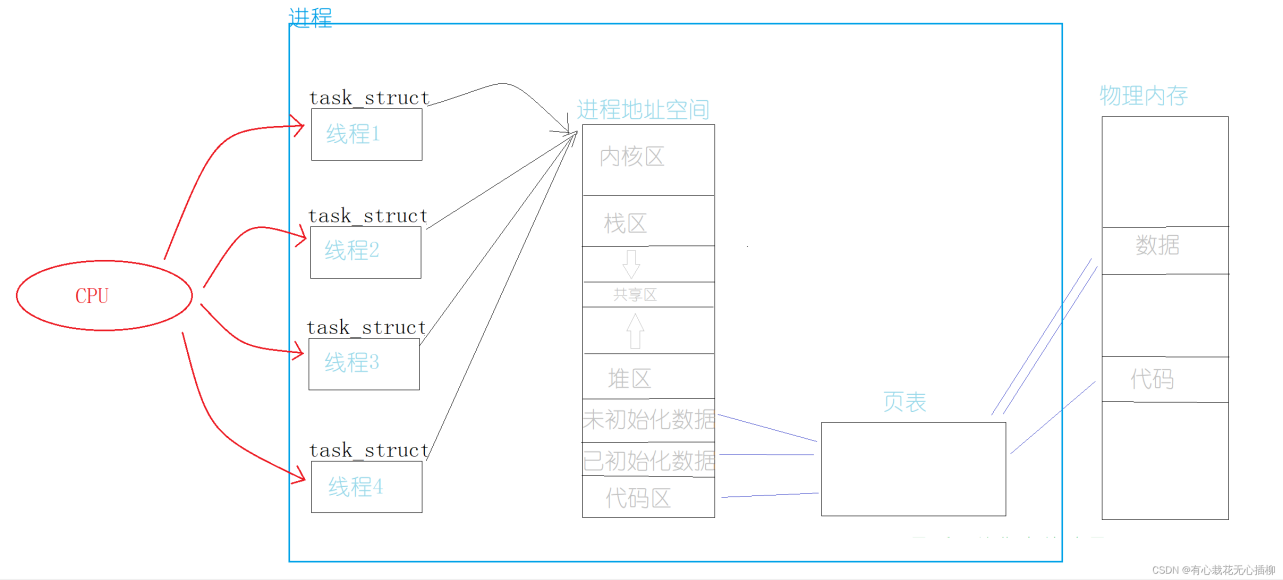
Linux 中的进程：

在Linux中一个进程的创建实际上伴随着进程控制块（PCB或task\_struct），进程地址空间以及页表的创建。

进程控制块（Processing Control Block），是操作系统核心中一种数据结构，主要表示进程状态。其作用是使一个在多道程序环境下不能独立运行的程序（含数据），成为一个能独立运行的基本单位或与其它进程并发执行的进程。或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。 PCB通常是系统内存占用区中的一个连续存区，它存放着操作系统用于描述进程情况及控制进程运行所需的全部信息，它使一个在多道程序环境下不能独立运行的程序成为一个能独立运行的基本单位或一个能与其他进程并发执行的进程。

Linux中的多线程（Linux并不存在真正意义上的线程）：

Linux中每创建一个进程，都要伴随着产生进程控制块，进程地址空间，页表。  
而操作系统中有大量的进程，一个进程内又可以包含多个线程，因此线程的数量一定会远远多于进程的。如果一个操作系统要真正地去支持使用线程，那么就必须有某种结构对线程进行相应的管理。比如：线程的创建，终止，转换，调度和释放回收等。  
但是Linux并没有单独地为线程创建相应的结构去管理，而是复用了进程的结构。而Windows系统单独为线程创建了相应的结构去管理，会比Linux复杂很多。



由上图可以很好地理解线程为什么是 CPU 调度的基本单位，在 CPU 看来，它只关心一个独立的执行流，无论进程内部是一个还是多个执行流，CPU 都是以 task\_struct 为单位来调度的。在 CPU 看来，Linux 中的进程比传统中的进程更加轻量化。进程的执行流我们叫轻量化进程。  
也能很好地理解了为什么进程是分配资源的基本单位，因为进程之间是相互独立的，每个进程都有相应的进程地址空间。

线程的优点:

* 新线程的代价要比创建一个新进程小得多
* 程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多
* 占用的资源要比进程少很多
* 分利用多处理器的可并行数量 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务
* 密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现
* O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。

线程的缺点:

* 性能损失: 少被外部事件阻塞的计算密集型线程往往无法与共它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。
* 性降低: 写多线程需要更全面更深入的考虑，在一个多线程程序里，因时间分配上的细微偏差或者因共享了,该共享的变量而造成不良影响的可能性是很大的，换句话说线程之间是缺乏保护的。
* 访问控制: 是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。
* 难度提高: 与调试一个多线程程序比单线程程序困难得多

线程异常:  
单个线程如果出现除零，野指针问题导致线程崩溃，进程也会随着崩溃  
线程是进程的执行分支，线程出异常，就类似进程出异常，进而触发信号机制，终止进程，进程终止，该进程内的所有线程也就随即退出

线程用途:  
合理的使用多线程，能提高CPU密集型程序的执行效率合理的使用多线程，能提高IO密集型程序的用户体验（如生活中我们一边打游戏一边听音乐，就是多线程运行的一种表现）

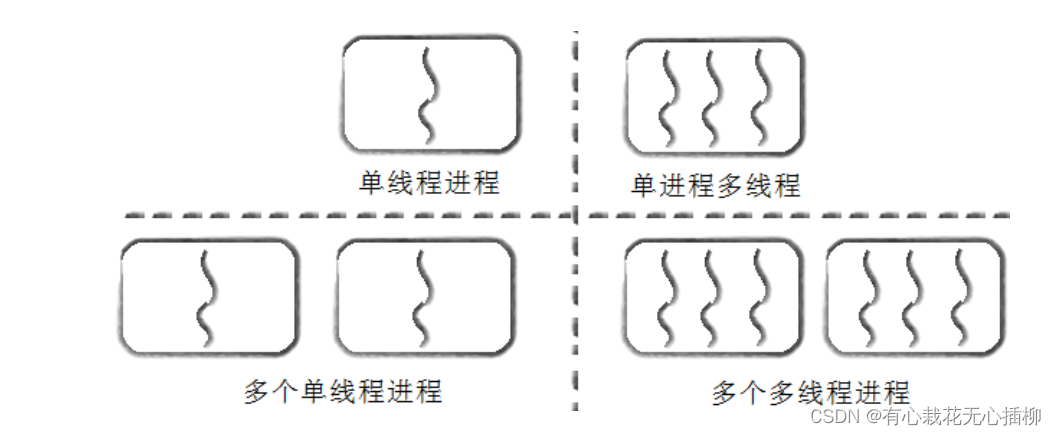
线程 VS 进程

线程共享进程数据，但也拥有自己的一部分数据:

线程ID； 一组寄存器； 栈； errno； 信号屏蔽字； 调度优先级

进程的多个线程共享 同一地址空间,因此Text Segment、Data Segment都是共享的,如果定义一个函数,在各线程中都可以调用,如果定义一个全局变量,在各线程中都可以访问到,除此之外,各线程还共享以下进程资源和环境:

文件描述符表；每种信号的处理方式(SIG\_ IGN、SIG\_ DFL或者自定义的信号处理函数)； 当前工作目录 ；用户id和组id

进程和线程的关系如下图:  
  
Linux 线程控制:

在操作系统的的视角下，Linux下没有真正意义的线程，而是用进程模拟的线程（LWP，轻量级进程），所以Linux不会提供直接创建线程的系统调用，最多提供创建轻量级进程的接口。  
但是对于用户来说，用户需要的是线程接口。  
所以 Linux 提供了用户线程库，对下将 Linux 接口封装，对上给用户提供进行线程控制的接口，也就是 pthread 库（原生线程库）。

与线程有关的函数构成了一个完整的系列，绝大多数函数的名字都是以“pthread\_”打头的  
要使用这些函数库，要通过引入头文<pthread.h>

链接这些线程函数库时要使用编译器命令的“-lpthread”选项

创建线程:

#include <pthread.h>  
  
int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,  
 void \*(\*start\_routine) (void \*), void \*arg);

**参数说明：**

* thread：用于存储新线程标识符的指针，pthread\_t 类型，用于标识新创建的线程。
* attr：用于指定新线程属性的指针，可以为 NULL，表示使用默认属性。
* start\_routine：是一个函数指针，指向新线程将要运行的函数。这个函数应该有一个 void\* 类型的参数，并返回一个 void\* 类型的指针。
* arg：是传递给 start\_routine 函数的参数。

**返回值：**

* 如果成功，返回 0。
* 如果出现错误，返回一个正整数，表示错误代码。

**使用示例：**

// file: pthread\_test.c  
#include <stdio.h>  
#include <pthread.h>  
  
void \*thread\_function(void \*arg) {  
 // 线程的主要逻辑  
 printf("Hello from the new thread! Argument passed: %s\n", (char\*)arg);  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
 const char \*message = "Hello, Thread!";  
  
 // 创建新线程  
 int result = pthread\_create(&my\_thread, NULL, thread\_function, (void\*)message);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 主线程继续执行其他工作  
 printf("Hello from the main thread!\n");  
  
 // 等待新线程结束  
 pthread\_join(my\_thread, NULL);  
  
 return 0;  
}

编译运行:

root@M5Core135:~/tmp# gcc pthread\_test.c -o pthread\_test -lpthread  
root@M5Core135:~/tmp# ./pthread\_test   
Hello from the main thread!  
Hello from the new thread! Argument passed: Hello, Thread!

在这个例子中，pthread\_create 函数用于创建一个新线程，新线程执行 thread\_function 函数，并传递了一个字符串作为参数。主线程继续执行其他工作，然后通过 pthread\_join 函数等待新线程结束。这样可以确保在主线程继续执行之前，新线程已经完成了它的任务。

线程ID及地址空间布局：

pthread\_ create 函数会产生一个线程 ID，存放在第一个参数指向的地址中。该线程 ID 和前面说的线程 ID 不是一回事。  
前面讲的线程 ID 属于进程调度的范畴。因为线程是轻量级进程，是操作系统调度器的最小单位，所以需要一个数值来唯一表示该线程。  
pthread\_ create 函数第一个参数指向一个虚拟内存单元，该内存单元的地址即为新创建线程的线程ID，属于本地线程库的范畴。线程库的后续操作，就是根据该线程ID来操作线程的。本地线程库提供了 pthread\_ self 函数，可以获得线程自身的ID：

#include <pthread.h>  
  
pthread\_t pthread\_self(void);

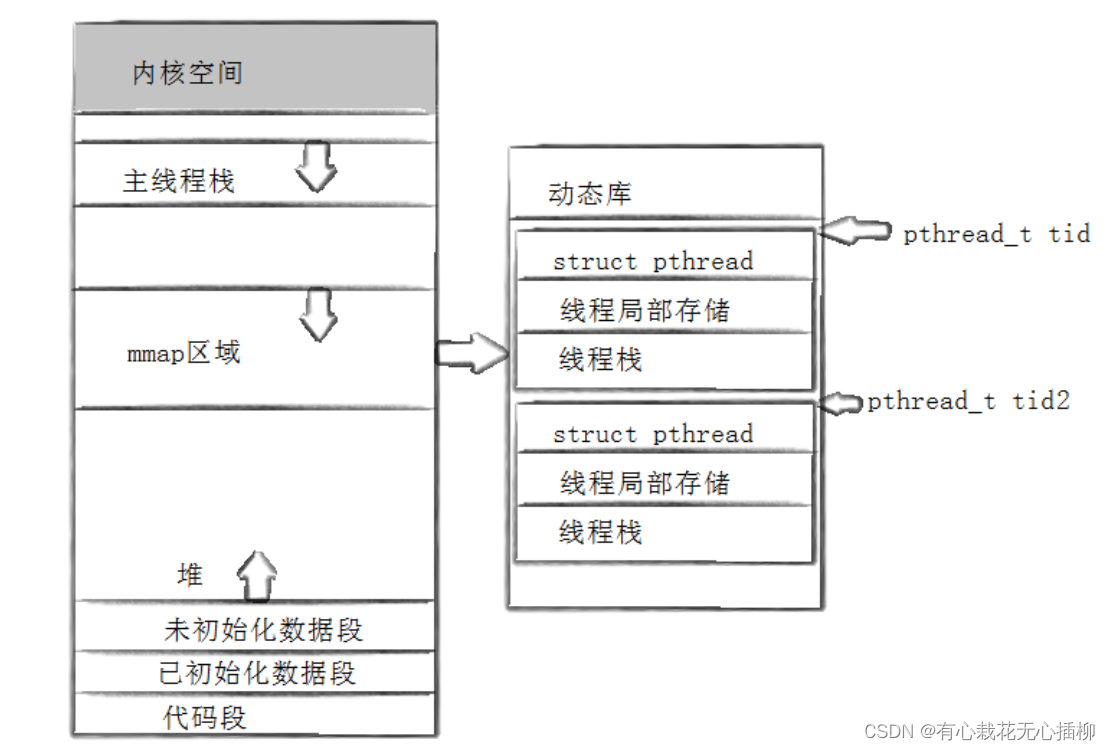
**返回值：**

* 返回调用线程的线程标识符，类型为 pthread\_t。

**使用示例：**

#include <stdio.h>  
#include <pthread.h>  
  
void \*thread\_function(void \*arg) {  
 // 获取当前线程的线程标识符  
 pthread\_t tid = pthread\_self();  
 printf("Thread ID inside the thread function: %lu\n", tid);  
  
 // 线程的主要逻辑  
 // ...  
  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
  
 // 创建新线程  
 int result = pthread\_create(&my\_thread, NULL, thread\_function, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 获取主线程的线程标识符  
 pthread\_t main\_tid = pthread\_self();  
 printf("Thread ID inside the main function: %lu\n", main\_tid);  
  
 // 主线程继续执行其他工作  
 // ...  
  
 // 等待新线程结束  
 pthread\_join(my\_thread, NULL);  
  
 return 0;  
}

pthread\_t 到底是什么类型呢？取决于实现。对于 Linux 目前实现的本地线程库实现而言，pthread\_t 类型的线程ID，本质就是一个进程地址空间上的一个地址。

下图中的mmap区域是共享区  


线程终止:

只终止某个线程而不是终止整个进程，可以有三种方法：

1.线程函数处进行 return 。  
2.线程可以自己调用 pthread\_exit 函数终止自己。  
3.一个线程可以调用 pthread\_cancel 函数终止同一进程中的另一个线程。

注意：在线程中使用return代表该线程退出，而在main函数（主线程）中使用return代表整个进程退出。

#include <pthread.h>  
  
void pthread\_exit(void \*retval);

**参数说明：**

* retval：线程的退出状态，可以是任何指针类型的值。这个状态可以被主线程或其他线程通过 pthread\_join 函数获取。

**使用示例：**

#include <stdio.h>  
#include <pthread.h>  
  
void \*thread\_function(void \*arg) {  
 printf("Thread is executing...\n");  
  
 // 模拟一些工作  
 for (int i = 0; i < 5; ++i) {  
 printf("Working... %d\n", i);  
 sleep(1);  
 }  
  
 // 线程退出，并传递退出状态  
 int \*exit\_status = malloc(sizeof(int));  
 \*exit\_status = 42;  
 pthread\_exit((void \*)exit\_status);  
  
 // 以下的代码不会被执行，因为线程已经退出  
 printf("This will not be printed.\n");  
  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
 void \*thread\_exit\_status;  
  
 // 创建新线程  
 int result = pthread\_create(&my\_thread, NULL, thread\_function, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 主线程等待新线程结束，并获取退出状态  
 pthread\_join(my\_thread, &thread\_exit\_status);  
  
 if (thread\_exit\_status != NULL) {  
 printf("Thread exit status: %d\n", \*((int \*)thread\_exit\_status));  
 free(thread\_exit\_status);  
 } else {  
 printf("Thread exit status is NULL.\n");  
 }  
  
 printf("Main thread exiting...\n");  
  
 return 0;  
}

在这个示例中，新线程执行了一些工作，然后通过 pthread\_exit 函数退出，并传递一个退出状态。主线程通过 pthread\_join 函数等待新线程结束，并获取退出状态。这样，可以在线程退出时传递一些信息，以便主线程了解线程的执行情况。在实际应用中，线程退出状态可以是任何类型的数据，根据需要进行传递。

#include <pthread.h>  
  
int pthread\_cancel(pthread\_t thread);

**参数说明：**

* thread：要取消的目标线程的线程标识符。

**返回值：**

* 成功时返回 0。
* 失败时返回一个正整数，表示错误代码。

**使用示例：**

#include <stdio.h>  
#include <pthread.h>  
#include <unistd.h>  
  
void \*thread\_function(void \*arg) {  
 // 线程的主要逻辑  
 while (1) {  
 printf("Thread is running...\n");  
 sleep(1);  
 }  
  
 // 以下的代码不会被执行，因为线程会在取消请求后终止  
 printf("This will not be printed.\n");  
  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
  
 // 创建新线程  
 int result = pthread\_create(&my\_thread, NULL, thread\_function, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 等待一段时间  
 sleep(3);  
  
 // 请求取消线程  
 result = pthread\_cancel(my\_thread);  
 if (result != 0) {  
 fprintf(stderr, "Thread cancellation failed: %d\n", result);  
 return 1;  
 }  
  
 // 等待新线程结束  
 result = pthread\_join(my\_thread, NULL);  
  
 if (result == 0) {  
 printf("Thread has been canceled.\n");  
 } else {  
 fprintf(stderr, "Thread join failed: %d\n", result);  
 return 1;  
 }  
  
 printf("Main thread exiting...\n");  
  
 return 0;  
}

在这个示例中，主线程创建了一个新线程，然后等待一段时间后发送取消请求给新线程。新线程在主循环中通过 sleep 模拟执行一些任务。当主线程发送取消请求后，新线程会在适当的时候（例如在 sleep 调用结束后）收到取消请求，并终止执行。主线程通过 pthread\_join 函数等待新线程结束。需要注意的是，线程在收到取消请求后，可以选择在取消点（cancellation point）终止执行，而不是立即终止。 sleep 是一个取消点的例子，它会在收到取消请求后立即返回。

线程回收:

已经退出的线程，其空间没有被释放，仍然在进程的地址空间内。创建新的线程不会复用刚才退出线程的地址空间。

```c  
#include <pthread.h>  
  
int pthread\_join(pthread\_t thread, void \*\*retval);  
  
```

**参数说明：**

* thread：要等待的目标线程的线程标识符。
* retval：用于存储目标线程的退出状态的指针。如果不关心退出状态，可以传递 NULL。

**返回值：**

* 成功时返回 0。
* 失败时返回一个正整数，表示错误代码。

**使用示例：**

#include <stdio.h>  
#include <pthread.h>  
#include <unistd.h>  
  
void \*thread\_function(void \*arg) {  
 printf("Thread is executing...\n");  
  
 // 模拟一些工作  
 for (int i = 0; i < 5; ++i) {  
 printf("Working... %d\n", i);  
 sleep(1);  
 }  
  
 // 线程退出，并传递退出状态  
 int \*exit\_status = malloc(sizeof(int));  
 \*exit\_status = 42;  
 pthread\_exit((void \*)exit\_status);  
  
 // 以下的代码不会被执行，因为线程已经退出  
 printf("This will not be printed.\n");  
  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
 void \*thread\_exit\_status;  
  
 // 创建新线程  
 int result = pthread\_create(&my\_thread, NULL, thread\_function, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 主线程等待新线程结束，并获取退出状态  
 result = pthread\_join(my\_thread, &thread\_exit\_status);  
  
 if (result == 0) {  
 if (thread\_exit\_status != NULL) {  
 printf("Thread exit status: %d\n", \*((int \*)thread\_exit\_status));  
 free(thread\_exit\_status);  
 } else {  
 printf("Thread exit status is NULL.\n");  
 }  
 } else {  
 fprintf(stderr, "Thread join failed: %d\n", result);  
 return 1;  
 }  
  
 printf("Main thread exiting...\n");  
  
 return 0;  
}

在这个示例中，主线程创建了一个新线程，然后通过 pthread\_join 函数等待新线程结束，并获取新线程的退出状态。新线程通过 pthread\_exit 函数传递了一个退出状态。主线程通过 pthread\_join 获取新线程的退出状态，并在必要时释放相应的资源。 pthread\_join 会阻塞调用线程，直到指定的线程结束为止。

线程分离:

默认情况下，新创建的线程是 joinable 的，线程退出后，需要对其进行 pthread\_join 操作，否则无法释放资源，从而造成系统泄漏。  
如果不关心线程的返回值，join 是一种负担，这个时候，我们可以告诉系统，当线程退出时，自动释放线程资源。

#include <pthread.h>  
  
int pthread\_detach(pthread\_t thread);

**参数说明：**

* thread：要分离的目标线程的线程标识符。

**返回值：**

* 成功时返回 0。
* 失败时返回一个正整数，表示错误代码。

**使用示例**

#include <stdio.h>  
#include <pthread.h>  
#include <unistd.h>  
  
void \*thread\_function(void \*arg) {  
 printf("Thread is executing...\n");  
  
 // 模拟一些工作  
 for (int i = 0; i < 5; ++i) {  
 printf("Working... %d\n", i);  
 sleep(1);  
 }  
  
 // 由于线程被设置为分离状态，无需显式调用pthread\_exit  
 // 也不需要等待其他线程调用pthread\_join  
  
 printf("Thread is exiting...\n");  
  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
  
 // 创建新线程  
 int result = pthread\_create(&my\_thread, NULL, thread\_function, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 将新线程设置为分离状态  
 result = pthread\_detach(my\_thread);  
 if (result != 0) {  
 fprintf(stderr, "Thread detach failed: %d\n", result);  
 return 1;  
 }  
  
 // 主线程继续执行其他工作  
 sleep(3);  
  
 printf("Main thread exiting...\n");  
  
 return 0;  
}

在这个示例中，主线程创建了一个新线程，并通过 pthread\_detach 函数将新线程设置为分离状态。这样，新线程在结束时会自动释放其相关资源，无需显式调用 pthread\_join 或 pthread\_exit。主线程继续执行其他工作，不需要等待新线程的结束。需要注意的是，一旦线程被设置为分离状态，就不能再将其转换为非分离状态。

<https://blog.csdn.net/iamxiaobai_/article/details/131723986>

## 7. 线程属性

属性对象是不透明的，而且不能通过赋值直接进行修改。系统提供了一组函数，用于初始化、配置和销毁每种对象类型。

初始化和配置属性后，属性便具有进程范围的作用域。使用属性时最好的方法即是在程序执行早期一次配置好所有必需的状态规范。然后，根据需要引用相应的属性对象。

使用属性对象具有两个主要优点。

* 使用属性对象可增加代码可移植性。
* 即使支持的属性可能会在实现之间有所变化，但您不需要修改用于创建线程实体的函数调用。这些函数调用不需要进行修改，因为属性对象是隐藏在接口之后的。
* 如果目标系统支持的属性在当前系统中不存在，则必须显式提供才能管理新的属性。管理这些属性是一项非常容易的移植任务，因为只需在明确定义的位置初始化属性对象一次即可。
* 应用程序中的状态规范已被简化。
* 例如，假设进程中可能存在多组线程。每组线程都提供单独的服务。每组线程都有各自的状态要求。
* 在应用程序执行初期的某一时间，可以针对每组线程初始化线程属性对象。以后所有线程的创建都会引用已经为这类线程初始化的属性对象。初始化阶段是简单和局部的。将来就可以快速且可靠地进行任何修改。

在进程退出时需要注意属性对象。初始化对象时，将为该对象分配内存。必须将此内存返回给系统。pthreads 标准提供了用于销毁属性对象的函数调用。

线程属性可用于控制更加细粒的线程，同时更加细致的控制线程的生命周期。

#include <pthread.h>  
  
int pthread\_attr\_init(pthread\_attr\_t \*attr);

**参数说明：**

* attr：指向线程属性对象的指针，函数将初始化该对象。

|  |  |  |
| --- | --- | --- |
| **属性** | **值** | **结果** |
| scope | PTHREAD\_SCOPE\_PROCESS | 新线程与进程中的其他线程发生竞争。 |
| detachstate | PTHREAD\_CREATE\_JOINABLE | 线程退出后，保留完成状态和线程 ID。 |
| stackaddr | NULL | 新线程具有系统分配的栈地址。 |
| stacksize | 0 | 新线程具有系统定义的栈大小。 |
| priority | 0 | 新线程的优先级为 0。 |
| inheritsched | PTHREAD\_EXPLICIT\_SCHED | 新线程不继承父线程调度优先级。 |
| schedpolicy | SCHED\_OTHER | 新线程对同步对象争用使用 Solaris 定义的固定优先级。线程将一直运行，直到被抢占或者直到线程阻塞或停止为止。 |

**返回值：**

* 成功时返回 0。
* 失败时返回一个正整数，表示错误代码。

#include <pthread.h>  
  
int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

**参数说明：**

* attr：指向线程属性对象的指针，函数将销毁该对象。

**返回值：**

* 成功时返回 0。
* 失败时返回一个正整数，表示错误代码。

**使用示例：**

#include <stdio.h>  
#include <pthread.h>  
  
void \*thread\_function(void \*arg) {  
 // 线程的主要逻辑  
 printf("Thread is executing...\n");  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread;  
 pthread\_attr\_t my\_thread\_attr;  
  
 // 初始化线程属性对象  
 int result = pthread\_attr\_init(&my\_thread\_attr);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread attribute initialization failed: %d\n", result);  
 return 1;  
 }  
  
 // 设置线程属性（例如，设置线程为分离状态）  
 // pthread\_attr\_setdetachstate(&my\_thread\_attr, PTHREAD\_CREATE\_DETACHED);  
  
 // 创建新线程时传递线程属性对象  
 result = pthread\_create(&my\_thread, &my\_thread\_attr, thread\_function, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 等待新线程结束  
 result = pthread\_join(my\_thread, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread join failed: %d\n", result);  
 return 1;  
 }  
  
 // 销毁线程属性对象  
 result = pthread\_attr\_destroy(&my\_thread\_attr);  
  
 if (result != 0) {  
 fprintf(stderr, "Thread attribute destruction failed: %d\n", result);  
 return 1;  
 }  
  
 printf("Main thread exiting...\n");  
  
 return 0;  
}

在这个示例中，主线程创建了一个线程属性对象 my\_thread\_attr，然后通过 pthread\_attr\_init 函数进行初始化。接着，可以通过其他函数如 pthread\_attr\_setdetachstate 来设置线程属性，例如将线程设置为分离状态。在创建新线程时，通过将线程属性对象传递给 pthread\_create 函数，新线程将继承这些属性。在程序结束时，通过 pthread\_attr\_destroy 函数销毁线程属性对象。

## 8. 线程同步异步

在多线程编程中，"同步"和"异步"是两个重要的概念，它们描述了线程或任务之间的交互和执行方式。

线程同步指的是在多线程环境中，确保线程以一种协调一致的方式执行，通常涉及到对共享资源的访问控制。目的防止数据竞争和条件竞争，确保数据的一致性和完整性。可以通过互斥锁（mutexes）、信号量（semaphores）、读写锁（rwlocks）、条件变量（condition variables）等机制来实现同步。特点是同步操作通常会阻塞线程，直到满足某些条件（例如，获取了锁或接收到信号）。

线程异步指的是线程独立于其他线程执行，不需要等待其他线程的操作完成。目的提高程序的响应性和并发性能，通过允许线程独立运行而不互相干扰。比如创建独立的线程来处理任务，使用回调函数、事件循环、消息队列等。异步操作的特点允许线程在等待某些操作（如I/O操作）完成时继续执行其他任务。  
同步例子，如果有一个共享的数据结构，比如一个列表，多个线程需要添加或删除元素，那么这些线程在访问和修改列表时需要通过锁来同步，以确保在任何时刻只有一个线程能修改列表。  
异步例子，如果一个线程负责读取网络数据，而另一个线程处理用户输入，这两个线程可以异步运行。网络线程在等待数据到达时不会阻塞用户输入线程，它们各自独立处理自己的任务。

线程同步关注的是线程间的协调和顺序，通常涉及共享资源的安全访问。  
线程异步则更关注于线程的独立性和非阻塞性，允许线程在后台执行任务，而主线程可以继续执行其他工作。

在设计多线程程序时，合理地使用同步和异步可以提高程序的效率和响应性，但也需要谨慎处理，以避免死锁、活锁或数据不一致等问题。

### 线程同步

承担线程同步的就是信号量。

线程信号量有下面几种形式：

* 互斥锁：使用互斥锁（互斥）可以使线程按顺序执行。通常，互斥锁通过确保一次只有一个线程执行代码的临界段来同步多个线程。互斥锁还可以保护单线程代码。
* 条件变量：使用条件变量可以以原子方式阻塞线程，直到某个特定条件为真为止。条件变量始终与互斥锁一起使用。
* 信号：信号可以是未命名的，也可以是命名的。未命名信号在进程内存中分配，并会进行初始化。未命名信号可能可供多个进程使用，具体取决于信号的分配和初始化的方式。命名信号类似于进程共享的信号，区别在于命名信号是使用路径名而非 pshared 值引用的。命名信号可以由多个进程共享。命名信号具有属主用户 ID、组 ID 和保护模式。
* 读写锁：通过读写锁，可以对受保护的共享资源进行并发读取和独占写入。读写锁是可以在**读取**或**写入**模式下锁定的单一实体。要修改资源，线程必须首先获取互斥写锁。必须释放所有读锁之后，才允许使用互斥写锁。

#### 互斥锁

互斥锁函数说明：

|  |  |  |  |
| --- | --- | --- | --- |
| **操作** | **函数名称** | 函数参数 | 返回值 |
| 初始化互斥锁 | int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr); | mutex：指向互斥锁对象的指针，函数将初始化该对象。attr：指向互斥锁属性对象的指针，可以为 NULL，表示使用默认属性。 | 成功时返回 0。失败时返回一个正整数，表示错误代码。 |
| 锁定互斥锁 | int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex); | mutex：指向互斥锁对象的指针，函数将尝试获取该互斥锁。 | 同上 |
| 解除锁定互斥锁 | int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex); | 同上 | 同上 |
| 使用非阻塞互斥锁锁定 | pthread\_mutex\_trylock | 同上 | 成功时返回 0。 如果互斥锁已经被占用，返回 EBUSY（表示资源忙碌，锁已经被其他线程占用）。 如果 mutex 无效，返回 EINVAL（表示参数无效）。 如果线程不是互斥锁的所有者，返回 EPERM（表示没有权限解锁互斥锁）。 其他错误情况下返回相应的错误代码。 |
| 销毁互斥锁 | int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex); | 同上 | 成功时返回 0。失败时返回一个正整数，表示错误代码。 |
| 初始化互斥锁属性对象 | pthread\_mutexattr\_init | attr：指向互斥锁属性对象的指针，函数将初始化该对象 | 同上 |
| 销毁互斥锁属性对象 | pthread\_mutexattr\_destroy | 同上 | 同上 |

示例:

#include <stdio.h>  
#include <pthread.h>  
  
// 共享资源  
int shared\_data = 0;  
  
// 互斥锁  
pthread\_mutex\_t my\_mutex;  
  
void \*thread\_function(void \*arg) {  
 // 尝试加锁  
 int result = pthread\_mutex\_trylock(&my\_mutex);  
  
 if (result == 0) {  
 // 成功获取锁  
 // 修改共享资源  
 shared\_data++;  
 printf("Thread %ld: Shared data: %d\n", (long)arg, shared\_data);  
  
 // 解锁  
 pthread\_mutex\_unlock(&my\_mutex);  
 } else if (result == EBUSY) {  
 // 锁已经被其他线程占用  
 printf("Thread %ld: Mutex is busy, try again later.\n", (long)arg);  
 } else {  
 // 其他错误情况  
 fprintf(stderr, "Thread %ld: Mutex trylock failed: %d\n", (long)arg, result);  
 }  
  
 return NULL;  
}  
void \*thread\_function1(void \*arg) {  
 // 加锁  
 if (pthread\_mutex\_lock(&my\_mutex) != 0) {  
 fprintf(stderr, "Mutex lock failed.\n");  
 return NULL;  
 }  
  
 // 修改共享资源  
 shared\_data++;  
 printf("Thread %ld: Shared data: %d\n", (long)arg, shared\_data);  
  
 // 解锁  
 if (pthread\_mutex\_unlock(&my\_mutex) != 0) {  
 fprintf(stderr, "Mutex unlock failed.\n");  
 return NULL;  
 }  
  
 return NULL;  
}  
  
int main() {  
 pthread\_t my\_thread1, my\_thread2;  
  
 // 初始化互斥锁  
 int result = pthread\_mutex\_init(&my\_mutex, NULL);  
  
 if (result != 0) {  
 fprintf(stderr, "Mutex initialization failed: %d\n", result);  
 return 1;  
 }  
  
 // 创建两个线程  
 result = pthread\_create(&my\_thread1, NULL, thread\_function, (void \*)1);  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 result = pthread\_create(&my\_thread2, NULL, thread\_function, (void \*)2);  
 if (result != 0) {  
 fprintf(stderr, "Thread creation failed: %d\n", result);  
 return 1;  
 }  
  
 // 等待两个线程结束  
 result = pthread\_join(my\_thread1, NULL);  
 if (result != 0) {  
 fprintf(stderr, "Thread join failed: %d\n", result);  
 return 1;  
 }  
  
 result = pthread\_join(my\_thread2, NULL);  
 if (result != 0) {  
 fprintf(stderr, "Thread join failed: %d\n", result);  
 return 1;  
 }  
  
 // 销毁互斥锁  
 result = pthread\_mutex\_destroy(&my\_mutex);  
  
 if (result != 0) {  
 fprintf(stderr, "Mutex destruction failed: %d\n", result);  
 return 1;  
 }  
  
 printf("Main thread exiting...\n");  
  
 return 0;  
}

#### 条件变量

条件变量函数说明：

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名称** | **操作** | 函数参数 | 返回值 |
| int pthread\_condattr\_init(pthread\_condattr\_t \*cattr); | 初始化条件变量属性 | 参数 attr 是一个指向条件变量属性对象的指针，该指针将被初始化为默认的条件变量属性值。 | 成功时返回 0。失败时返回一个正整数，表示错误代码。 |
| int pthread\_condattr\_destroy(pthread\_condattr\_t \*attr); | 删除条件变量属性 | 参数 attr 是一个指向条件变量属性对象的指针，该指针将被销毁。 | 同上 |
| int pthread\_condattr\_setpshared(pthread\_condattr\_t \*attr, int pshared); | 设置条件变量的范围 | 参数 attr 是一个指向条件变量属性对象的指针。参数 pshared 指定条件变量的共享属性，可以为 PTHREAD\_PROCESS\_PRIVATE（条件变量只能在同一进程内的线程之间共享）或 PTHREAD\_PROCESS\_SHARED（条件变量可在不同进程间的线程之间共享）。 | 同上 |
| int pthread\_condattr\_getpshared(const pthread\_condattr\_t \*attr, int \*pshared); | 获取条件变量的范围 | 参数 attr 是一个指向条件变量属性对象的指针，该对象已经被初始化。参数 pshared 是一个指向整数的指针，用于存储条件变量的共享属性。 | 同上 |

示例：

#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <errno.h>  
  
// 共享资源  
int shared\_variable = 0;  
  
// 条件变量  
pthread\_cond\_t condition\_variable;  
  
// 互斥锁  
pthread\_mutex\_t mutex;  
  
// 示例线程函数  
void\* example\_thread(void\* arg) {  
 int thread\_id = \*(int\*)arg;  
 free(arg);  
  
 // 模拟一些工作  
 sleep(1);  
  
 // 加锁  
 if (pthread\_mutex\_lock(&mutex) != 0) {  
 perror("pthread\_mutex\_lock");  
 return NULL;  
 }  
  
 // 修改共享资源  
 shared\_variable = thread\_id;  
  
 // 发送信号通知其他线程条件变量的状态已改变  
 if (pthread\_cond\_signal(&condition\_variable) != 0) {  
 perror("pthread\_cond\_signal");  
 pthread\_mutex\_unlock(&mutex);  
 return NULL;  
 }  
  
 // 解锁  
 if (pthread\_mutex\_unlock(&mutex) != 0) {  
 perror("pthread\_mutex\_unlock");  
 return NULL;  
 }  
  
 return NULL;  
}  
  
int main() {  
 // 初始化互斥锁  
 if (pthread\_mutex\_init(&mutex, NULL) != 0) {  
 perror("pthread\_mutex\_init");  
 return 1;  
 }  
  
 // 初始化条件变量属性对象  
 pthread\_condattr\_t cond\_attr;  
 if (pthread\_condattr\_init(&cond\_attr) != 0) {  
 perror("pthread\_condattr\_init");  
 return 1;  
 }  
  
 // 设置条件变量属性为可进程间共享  
 if (pthread\_condattr\_setpshared(&cond\_attr, PTHREAD\_PROCESS\_SHARED) != 0) {  
 perror("pthread\_condattr\_setpshared");  
 return 1;  
 }  
  
 // 使用初始化后的属性对象创建条件变量  
 if (pthread\_cond\_init(&condition\_variable, &cond\_attr) != 0) {  
 perror("pthread\_cond\_init");  
 return 1;  
 }  
  
 // 创建两个线程  
 pthread\_t thread1, thread2;  
 int \*thread1\_id = malloc(sizeof(int));  
 int \*thread2\_id = malloc(sizeof(int));  
 \*thread1\_id = 1;  
 \*thread2\_id = 2;  
  
 if (pthread\_create(&thread1, NULL, example\_thread, (void\*)thread1\_id) != 0 ||  
 pthread\_create(&thread2, NULL, example\_thread, (void\*)thread2\_id) != 0) {  
 perror("pthread\_create");  
 return 1;  
 }  
  
 // 加锁  
 if (pthread\_mutex\_lock(&mutex) != 0) {  
 perror("pthread\_mutex\_lock");  
 return 1;  
 }  
  
 // 等待条件变量的状态改变  
 if (pthread\_cond\_wait(&condition\_variable, &mutex) != 0) {  
 perror("pthread\_cond\_wait");  
 pthread\_mutex\_unlock(&mutex);  
 return 1;  
 }  
  
 // 打印共享资源的值  
 printf("Main Thread: Shared Variable = %d\n", shared\_variable);  
  
 // 解锁  
 if (pthread\_mutex\_unlock(&mutex) != 0) {  
 perror("pthread\_mutex\_unlock");  
 return 1;  
 }  
  
 // 等待线程结束  
 if (pthread\_join(thread1, NULL) != 0 || pthread\_join(thread2, NULL) != 0) {  
 perror("pthread\_join");  
 return 1;  
 }  
  
 // 销毁条件变量属性对象  
 if (pthread\_condattr\_destroy(&cond\_attr) != 0) {  
 perror("pthread\_condattr\_destroy");  
 return 1;  
 }  
  
 // 销毁条件变量  
 if (pthread\_cond\_destroy(&condition\_variable) != 0) {  
 perror("pthread\_cond\_destroy");  
 return 1;  
 }  
  
 // 销毁互斥锁  
 if (pthread\_mutex\_destroy(&mutex) != 0) {  
 perror("pthread\_mutex\_destroy");  
 return 1;  
 }  
  
 return 0;  
}

#### 信号量

信号量函数说明:

所需的头文件

#include <semaphore.h>

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名称** | **操作** | 函数参数 | 返回值 |
| int sem\_init(sem\_t \*sem, int pshared, unsigned int value); | 初始化信号 | sem: 指向要初始化的信号量的指针。 pshared: 指定信号量是在进程间共享还是线程间共享。如果为 0，信号量将被线程共享；否则，信号量将在进程间共享。通常使用 0。 value: 指定信号量的初始值。 | 成功时返回 0。失败时返回一个正整数，表示错误代码。 |
| int sem\_post(sem\_t \*sem); | 增加信号 | sem: 指向要增加的信号量的指针。 | 同上 |
| int sem\_wait(sem\_t \*sem); | 基于信号计数阻塞 | sem: 指向要减少的信号量的指针。 | 同上 |
| int sem\_trywait(sem\_t \*sem); | 减小信号计数 | sem: 指向要尝试减少的信号量的指针。 | 同上 |
| int sem\_destroy(sem\_t \*sem); | 销毁信号状态 | sem: 指向要销毁的信号量的指针。 | 同上 |

示例:

#include <stdio.h>  
#include <semaphore.h>  
#include <pthread.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
// 全局变量，作为信号量  
sem\_t mySemaphore;  
  
void\* thread\_wait\_function(void\* arg) {  
 int thread\_id = \*(int\*)arg;  
 free(arg);  
  
 printf("Thread %d is waiting using sem\_wait...\n", thread\_id);  
  
 // 等待信号量  
 if (sem\_wait(&mySemaphore) == 0) {  
 printf("Thread %d acquired the semaphore using sem\_wait.\n", thread\_id);  
  
 // 模拟一些工作  
 sleep(2);  
  
 printf("Thread %d releasing the semaphore.\n", thread\_id);  
  
 // 释放信号量  
 sem\_post(&mySemaphore);  
 } else {  
 perror("sem\_wait");  
 }  
  
 return NULL;  
}  
  
void\* thread\_trywait\_function(void\* arg) {  
 int thread\_id = \*(int\*)arg;  
 free(arg);  
  
 printf("Thread %d is trying to acquire the semaphore using sem\_trywait...\n", thread\_id);  
  
 // 尝试等待信号量，如果信号量的值为正，减少该值并继续执行，否则立即返回  
 if (sem\_trywait(&mySemaphore) == 0) {  
 printf("Thread %d acquired the semaphore using sem\_trywait.\n", thread\_id);  
  
 // 模拟一些工作  
 sleep(2);  
  
 printf("Thread %d releasing the semaphore.\n", thread\_id);  
  
 // 释放信号量  
 sem\_post(&mySemaphore);  
 } else {  
 printf("Thread %d failed to acquire the semaphore using sem\_trywait.\n", thread\_id);  
 }  
  
 return NULL;  
}  
  
int main() {  
 // 初始化信号量  
 if (sem\_init(&mySemaphore, 0, 1) != 0) {  
 perror("sem\_init");  
 return 1;  
 }  
  
 // 创建线程，一个使用sem\_wait，一个使用sem\_trywait  
 pthread\_t thread\_wait, thread\_trywait;  
   
 int\* thread\_id\_wait = malloc(sizeof(int));  
 \*thread\_id\_wait = 1;  
 if (pthread\_create(&thread\_wait, NULL, thread\_wait\_function, (void\*)thread\_id\_wait) != 0) {  
 perror("pthread\_create");  
 return 1;  
 }  
  
 int\* thread\_id\_trywait = malloc(sizeof(int));  
 \*thread\_id\_trywait = 2;  
 if (pthread\_create(&thread\_trywait, NULL, thread\_trywait\_function, (void\*)thread\_id\_trywait) != 0) {  
 perror("pthread\_create");  
 return 1;  
 }  
  
 // 主线程等待所有线程结束  
 if (pthread\_join(thread\_wait, NULL) != 0) {  
 perror("pthread\_join");  
 return 1;  
 }  
  
 if (pthread\_join(thread\_trywait, NULL) != 0) {  
 perror("pthread\_join");  
 return 1;  
 }  
  
 // 销毁信号量  
 if (sem\_destroy(&mySemaphore) != 0) {  
 perror("sem\_destroy");  
 return 1;  
 }  
  
 return 0;  
}

#### 读写锁

所需头文件

#include <pthread.h>

读写锁函数说明:

|  |  |  |  |
| --- | --- | --- | --- |
| **函数名称** | **操作** | 函数参数 | 返回值 |
| int pthread\_rwlockattr\_init(pthread\_rwlockattr\_t \*attr); | 初始化读写锁属性 | attr: 指向要初始化的读写锁属性对象的指针。 | 成功时返回 0。失败时返回一个正整数，表示错误代码。 |
| int pthread\_rwlockattr\_destroy(pthread\_rwlockattr\_t \*attr); | 销毁读写锁属性 | attr: 指向要初始化的读写锁属性对象的指针。 | 同上 |
| int pthread\_rwlockattr\_setpshared(pthread\_rwlockattr\_t \*attr, int pshared); | 设置读写锁属性 | attr: 指向读写锁属性对象的指针。 pshared: 用于指定共享属性的值。可以是 PTHREAD\_PROCESS\_PRIVATE（锁在进程间不共享）或 PTHREAD\_PROCESS\_SHARED（锁在进程间共享）。 | 同上 |
| int pthread\_rwlockattr\_getpshared(const pthread\_rwlockattr\_t \*attr, int \*pshared); | 获取读写锁属性 | attr: 指向读写锁属性对象的指针。 pshared: 用于存储共享属性值的整数指针。 | 同上 |

示例:

#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
pthread\_rwlock\_t rwlock;  
  
void \*reader(void \*arg) {  
 pthread\_rwlock\_rdlock(&rwlock);  
 printf("Reader: acquired the lock\n");  
 pthread\_rwlock\_unlock(&rwlock);  
 printf("Reader: released the lock\n");  
 return NULL;  
}  
  
void \*writer(void \*arg) {  
 pthread\_rwlock\_wrlock(&rwlock);  
 printf("Writer: acquired the lock\n");  
 pthread\_rwlock\_unlock(&rwlock);  
 printf("Writer: released the lock\n");  
 return NULL;  
}  
  
int main() {  
 pthread\_t t1, t2, t3;  
 pthread\_rwlockattr\_t attr;  
 int shared;  
  
 // Initialize rwlock attributes  
 pthread\_rwlockattr\_init(&attr);  
  
 // Set the rwlock to be shared between processes  
 pthread\_rwlockattr\_setpshared(&attr, PTHREAD\_PROCESS\_SHARED);  
  
 // Get the rwlock shared attribute  
 pthread\_rwlockattr\_getpshared(&attr, &shared);  
 if (shared == PTHREAD\_PROCESS\_SHARED) {  
 printf("rwlock is set to be shared between processes.\n");  
 } else {  
 printf("rwlock is set to be shared between threads of a process.\n");  
 }  
  
 // Initialize the rwlock with the attributes  
 pthread\_rwlock\_init(&rwlock, &attr);  
  
 // Create reader and writer threads  
 pthread\_create(&t1, NULL, reader, NULL);  
 pthread\_create(&t2, NULL, writer, NULL);  
 pthread\_create(&t3, NULL, reader, NULL);  
  
 // Wait for threads to finish  
 pthread\_join(t1, NULL);  
 pthread\_join(t2, NULL);  
 pthread\_join(t3, NULL);  
  
 // Destroy rwlock and its attributes  
 pthread\_rwlock\_destroy(&rwlock);  
 pthread\_rwlockattr\_destroy(&attr);  
  
 return 0;  
}

### 线程异步

线程的异步机制只有信号，类似于进程的信号。

线程信号具备以下特点

* 任何线程都可以向其它线程(同一进程下)发送信号；
* 每个线程都具备自己独立的信号屏蔽集，不影响其它线程；
* 线程创建时，不继承原线程的信号屏蔽集；
* 同进程下，所有线程共享对某信号的处理方式，即一个设置，所有有效；
* 多个线程的程序，向某一个线程发送终止信号，则整个进程终止

所需头文件

#include <signal.h>

信号函数说明:

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | **操作** | 函数参数 | 返回值 |
| sighandler\_t signal(int signum, sighandler\_t handler); | 安装信号 | signum：代表要设置处理程序的信号的编号，例如SIGINT、SIGTERM等。 handler：是一个函数指针，指向您希望在接收到指定信号时调用的处理程序函数。处理程序函数必须接受一个整数参数，代表接收到的信号编号，通常命名为signum。 | signal 函数的返回值是一个函数指针类型 sighandler\_t，该函数指针指向之前与该信号相关联的信号处理程序。这是为了让您可以在设置新的信号处理程序之前保存旧的处理程序，以便稍后恢复或查询它。如果之前没有与该信号相关联的处理程序，返回值通常为 SIG\_DFL（默认处理程序）或 SIG\_IGN（忽略信号）。 |
| int pthread\_kill(pthread\_t threadid, int signo); | 发送信号 | 发送信号 threadid：指定目标线程的线程标识符（pthread\_t 类型），即要发送信号的线程。 signo：指定要发送的信号的编号，例如SIGINT、SIGTERM等。 | pthread\_kill 函数成功时返回0，如果发生错误，则返回一个非零的错误代码。您可以使用 pthread\_kill 函数向另一个线程发送信号，以便该线程可以捕获信号并执行相应的处理。 |
| int pthread\_sigmask(int how, const sigset\_t \*set, sigset\_t \*oldset); | 设置屏蔽集 | how：指定要执行的操作。可以采用以下值之一： SIG\_BLOCK：将 set 中的信号添加到线程的信号屏蔽集中。 SIG\_UNBLOCK：从线程的信号屏蔽集中解除 set 中的信号屏蔽。 SIG\_SETMASK：将线程的信号屏蔽集设置为 set 中的信号集。 set：一个指向 sigset\_t 类型的指针，指定要添加、解除或设置的信号集。 oldset：一个指向 sigset\_t 类型的指针，用于存储之前的信号屏蔽集。如果不需要保存之前的信号屏蔽集，可以将此参数设置为 NULL。 | pthread\_sigmask 函数成功时返回0，如果发生错误，则返回一个非零的错误代码。 |

示例:

/\*threadsig.c\*/  
#include <signal.h>  
#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
   
void sighandler(int signo);  
   
void \*  
thr1\_fn(void \*arg)  
{  
 struct sigaction action;  
 action.sa\_flags = 0;  
 action.sa\_handler = sighandler;  
   
 sigaction(SIGINT, &action, NULL);  
   
 pthread\_t tid = pthread\_self();  
 int rc;  
   
 printf("thread 1 with tid:%lu\n", tid);  
 rc = sleep(60);  
 if (rc != 0)  
 printf("thread 1... interrupted at %d second\n", 60 - rc);  
 printf("thread 1 ends\n");  
 return NULL;  
}  
   
void \*  
thr2\_fn(void \*arg)  
{  
 struct sigaction action;  
 pthread\_t tid = pthread\_self();  
 int rc, err;  
   
 printf("thread 2 with tid:%lu\n", tid);  
   
 action.sa\_flags = 0;  
 action.sa\_handler = sighandler;  
   
 err = sigaction(SIGALRM, &action, NULL);  
   
 rc = sleep(60);  
 if (rc != 0)  
 printf("thread 2... interrupted at %d second\n", 60 - rc);  
 printf("thread 2 ends\n");  
 return NULL;  
}  
   
void \*  
thr3\_fn(void \*arg)  
{  
 pthread\_t tid = pthread\_self();  
 sigset\_t mask;  
 int rc, err;  
   
 printf("thread 3 with tid%lu\n", tid);  
   
   
 sigemptyset(&mask); /\* 初始化mask信号集 \*/  
   
 sigaddset(&mask, SIGALRM);  
 err = pthread\_sigmask(SIG\_BLOCK, &mask, NULL);  
 if (err != 0)  
 {  
 printf("%d, %s/n", rc, strerror(rc));  
 return NULL;  
 }  
   
 rc = sleep(10);  
 if (rc != 0)  
 printf("thread 3... interrupted at %d second\n", 60 - rc);  
 err = pthread\_sigmask( SIG\_UNBLOCK,&mask,NULL );  
 if ( err != 0 )  
 {  
 printf("unblock %d, %s/n", rc, strerror(rc));  
 return NULL;  
 }  
   
 rc = sleep(10);  
 if (rc != 0)  
 printf("thread 3... interrupted at %d second after unblock\n", 60 - rc);  
  
 printf("thread 3 ends\n");  
 return NULL;  
   
 return NULL;  
}  
   
int  
main(void)  
{  
 int rc, err;  
 pthread\_t thr1, thr2, thr3, thrm = pthread\_self();  
   
 printf("thread main with pid %lu\n",thrm);  
 err = pthread\_create(&thr1, NULL, thr1\_fn, NULL);  
 if (err != 0) {  
 printf("error in creating pthread:%d\t%s\n",err, strerror(rc));  
 exit(1);  
 }  
   
   
/\* pthread\_kill(thr1, SIGALRM); send a SIGARLM signal to thr1 before thr2 set the signal handler, then the whole process will be terminated\*/  
 err = pthread\_create(&thr2, NULL, thr2\_fn, NULL);  
 if (err != 0) {  
 printf("error in creating pthread:%d\t%s\n",err, strerror(rc));  
 exit(1);  
 }  
   
 err = pthread\_create(&thr3, NULL, thr3\_fn, NULL);  
 if (err != 0) {  
 printf("error in creating pthread:%d\t%s\n",err, strerror(rc));  
 exit(1);  
 }  
   
 sleep(10);  
 //内部产生的信号，只有指定的线程能收到，因此要向所有线程发送  
 pthread\_kill(thr1, SIGALRM);  
 pthread\_kill(thr2, SIGALRM);  
 pthread\_kill(thr3, SIGALRM);  
 pthread\_kill(thr3, SIGALRM);  
 pthread\_kill(thr3, SIGALRM);  
 sleep(5);  
 pthread\_join(thr1, NULL); /\*wait for the threads to complete.\*/  
 pthread\_join(thr2, NULL);  
 pthread\_join(thr3, NULL);  
 printf("main ends\n");  
 return 0;  
}  
   
void  
sighandler(int signo)  
{  
 pthread\_t tid = pthread\_self();  
   
 printf("thread with pid:%lu receive signo:%d\n", tid, signo);  
 return;  
}

在上面的代码中，主线程创建三个线程。线程1注册SIGINT信号(即ctrl+c) ，线程2注册SIGALRM，线程三则是先阻塞SIGALRM，然后解除阻塞。

编译后看运行结果：

xzc@xzc-HP-ProBook-4446s:~/code/test$ gcc -o threadsig threadsig.c -pthread  
xzc@xzc-HP-ProBook-4446s:~/code/test$ ./threadsig   
thread main with pid 139946922108736  
thread 2 with tid:139946905396992  
thread 1 with tid:139946913789696  
thread 3 with tid139946897004288  
^Cthread with pid:139946922108736 receive signo:2  
thread with pid:139946913789696 receive signo:14  
thread 1... interrupted at 4 second  
thread 1 ends  
thread with pid:139946905396992 receive signo:14  
thread 2... interrupted at 4 second  
thread 2 ends  
^Cthread with pid:139946922108736 receive signo:2  
^Cthread with pid:139946922108736 receive signo:2  
thread with pid:139946897004288 receive signo:14  
thread 3 ends  
main ends  
xzc@xzc-HP-ProBook-4446s:~/code/test$

在第一行红色的地方，主线程正在sleep，我按下ctrl+c，只有主线程收到并处理了信号。说明进程会从主线程开始查找不阻塞该信号的线程来处理job control类的信号。

由于主线程sleep被打断，随后向三个线程发送了SIGALRM，线程1、2由于没有阻塞该信号，被迫从sleep中醒来，并结束进程。进程3仍在sleep。

在第二行红色的地方，线程3第一次sleep终于完成，解除了对SIGALRM的阻塞。于是马上收到被阻塞的SIGALRM(发送3次，只收到一次)。

请注意信号阻塞与忽略的区别。

<https://www.cnblogs.com/Jscroop/p/12815466.html>

## 9. IPC

IPC（Inter-Process Communication，进程间通信）是一组机制和技术，允许不同进程之间在同一系统上进行数据传输、共享资源和协作。这些IPC机制用于在多个进程之间进行通信，以便它们可以协调工作和交换数据。

### 管道（Pipe）

* 管道是一种用于在父子进程或兄弟进程之间进行通信的简单IPC机制。
* 有两种类型：无名管道（Anonymous Pipe）和命名管道（Named Pipe，也称为FIFO）。
* 无名管道通常用于具有亲缘关系的进程，它们共享文件描述符。
* 命名管道可用于非亲缘关系进程之间的通信。
* 管道是一种半双工通信机制，管道的数据流是单向的,通常从一个进程（写入端）流向另一个进程（读取端），需要使用多个管道才能实现双向通信。
* 管道中的数据是无格式的字节流,通信两端需要协商好数据格式。
* 写端写满数据后会阻塞,读端需要及时读取以防止写端阻塞。
* 管道有大小限制,一般是页大小,超过限制写入会阻塞。
* 管道中的数据存储在页缓冲区中,读完后会从内核缓冲区释放。
* 无名管道在最后一个读或写端关闭后自动删除,不占用磁盘空间，有名管道需要手动删除。
* 可以使用select/poll检测管道是否就绪以实现同步，不用阻塞线程。
* 管道的实质是一个具有 FIFO 性质的特殊文件。

所需头文件：

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

相关函数说明:

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | **操作** | 函数参数 | 返回值 |
| int pipe(int pipefd[2]); | 创建无名管道 | 参数 pipefd 是一个包含两个整数元素的数组，用于存储管道的文件描述符。pipefd[0] 用于从管道中读取数据，pipefd[1] 用于向管道中写入数据。 | 成功时返回0，失败时返回-1，并设置 errno 来指示错误原因。 |
| int mkfifo(const char \*pathname, mode\_t mode); | 创建命名管道 | pathname：指定要创建的命名管道的路径。这是一个字符串，通常是一个文件系统路径，可以是绝对路径或相对路径。 mode：指定创建的命名管道的权限和权限掩码。通常使用八进制表示，例如 0666 表示读写权限。 | 成功时返回0，失败时返回-1，并设置 errno 来指示错误原因。 |

无名管道的示例:

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
int main() {  
 int pipefd[2];  
 pid\_t child\_pid;  
  
 // 创建管道  
 if (pipe(pipefd) == -1) {  
 perror("pipe");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建子进程  
 child\_pid = fork();  
  
 if (child\_pid == -1) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 if (child\_pid == 0) {  
 // 子进程  
 close(pipefd[1]); // 关闭写端  
 char buffer[100];  
 ssize\_t bytes\_read;  
  
 // 从管道中读取数据  
 bytes\_read = read(pipefd[0], buffer, sizeof(buffer));  
  
 if (bytes\_read == -1) {  
 perror("read");  
 exit(EXIT\_FAILURE);  
 }  
  
 printf("Child received data from parent: %.\*s\n", (int)bytes\_read, buffer);  
 close(pipefd[0]); // 关闭读端  
 } else {  
 // 父进程  
 close(pipefd[0]); // 关闭读端  
 const char \*message = "Hello, child!";  
  
 // 向管道中写入数据  
 if (write(pipefd[1], message, strlen(message)) == -1) {  
 perror("write");  
 exit(EXIT\_FAILURE);  
 }  
  
 close(pipefd[1]); // 关闭写端  
 }  
  
 return 0;  
}

有名管道的示例:

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <sys/wait.h>  
#include <string.h>  
  
int main() {  
 const char \*fifo\_path = "/tmp/my\_fifo"; // 命名管道的路径  
  
 // 创建命名管道，如果已存在则忽略  
 if (mkfifo(fifo\_path, 0666) == -1) {  
 perror("mkfifo");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建子进程  
 pid\_t pid = fork();  
  
 if (pid == -1) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 if (pid == 0) {  
 // 子进程写入数据到命名管道  
 int fd = open(fifo\_path, O\_WRONLY);  
 if (fd == -1) {  
 perror("open");  
 exit(EXIT\_FAILURE);  
 }  
  
 const char \*message = "Hello, Parent!";  
 ssize\_t bytes\_written;  
  
 // 向命名管道中写入数据  
 bytes\_written = write(fd, message, strlen(message));  
  
 if (bytes\_written == -1) {  
 perror("write");  
 exit(EXIT\_FAILURE);  
 }  
  
 printf("Child wrote data to named pipe.\n");  
  
 // 关闭命名管道文件描述符  
 close(fd);  
 } else {  
 // 父进程读取数据从命名管道  
 int fd = open(fifo\_path, O\_RDONLY);  
 if (fd == -1) {  
 perror("open");  
 exit(EXIT\_FAILURE);  
 }  
  
 char buffer[100];  
 ssize\_t bytes\_read;  
  
 // 从命名管道中读取数据  
 bytes\_read = read(fd, buffer, sizeof(buffer));  
  
 if (bytes\_read == -1) {  
 perror("read");  
 exit(EXIT\_FAILURE);  
 }  
  
 printf("Parent received data from child: %.\*s\n", (int)bytes\_read, buffer);  
  
 // 关闭命名管道文件描述符  
 close(fd);  
  
 // 等待子进程退出  
 wait(NULL);  
 }  
  
 return 0;  
}

### 消息队列（Message Queues）：

* 消息队列允许进程通过消息进行通信，每个消息都有一个特定的类型。
* 进程可以通过消息队列发送和接收消息，通常采用FIFO（先进先出）方式处理消息。
* 消息队列可以用于不同进程之间的通信，包括不具有亲缘关系的进程。
* 进程间消息队列允许进程异步通信，发送者和接收者不需要同时准备好。

所需头文件：

#include <fcntl.h> /\*Define 0\_\*constant\*/  
#include <sys/stat.h> /\*Define mode constants\*/  
#include <mqueue.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode, struct mq\_attr \*attr); | 创建消息队列实例 | name: 消息队列名称，oflag：根据传入标识来创建或者打开一个已创建的消息队列，mode：访问权限，attr：消息队列属性地址。 | 成功返回消息队列描述符，失败返回-1，错误码存于error中 |
| mqd\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned \*msg\_prio); | 无限阻塞方式接收消息 | mqdes: 消息队列描述符，msg\_ptr：消息体缓冲区地址，msg\_len：消息体长度，长度必须大于等于消息属性设定的最大值，msg\_prio：消息优先级。 | 成功返回消息长度，失败返回-1，错误码存于error中 |
| mqd\_t mq\_timedreceive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned \*msg\_prio, const struct timespec \*abs\_timeout); | 指定超时时间阻塞方式接收消息 | mqdes: 消息队列描述符，msg\_ptr：消息体缓冲区地址，msg\_len：消息体长度，长度必须大于等于消息属性设定的最大值，msg\_prio：消息优先级，abs\_timeout：超时时间 | 成功返回消息长度，失败返回-1，错误码存于error中 |
| mqd\_t mq\_send(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned msg\_prio); | 无限阻塞方式发送消息 | mqdes: 消息队列描述符，msg\_ptr：待发送消息体缓冲区地址，msg\_len：消息体长度，msg\_prio：消息优先级 | 成功返回 0，失败返回-1 |
| mqd\_t mq\_timedsend(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned msg\_prio, const struct timespec \*abs\_timeout); | 指定超时时间阻塞方式发送消息 | mqdes: 消息队列描述符，msg\_ptr：待发送消息体缓冲区地址，msg\_len：消息体长度，msg\_prio：消息优先级，abs\_timeout：超时时间 | 成功返回 0，失败返回-1 |
| mqd\_t mq\_close(mqd\_t mqdes); | 关闭消息队列 | mqdes: 消息队列描述符 | 成功返回0，失败返回-1 |
| mqd\_t mq\_unlink(const char \*name); | 删除消息队列，一旦消息队列被成功删除，将无法再使用相同的名称创建该消息队列。 | name: 消息队列名称 | 成功返回0，失败返回-1 |

使用示例:

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <mqueue.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
  
int main() {  
 const char \*queue\_name = "/my\_queue"; // 消息队列的名称  
 mqd\_t mq; // 消息队列描述符  
 struct mq\_attr attr; // 消息队列属性  
 pid\_t child\_pid;  
  
 // 设置消息队列属性  
 attr.mq\_flags = 0;  
 attr.mq\_maxmsg = 10; // 最大消息数  
 attr.mq\_msgsize = 256; // 每个消息的最大大小  
 attr.mq\_curmsgs = 0;  
  
 // 创建消息队列  
 mq = mq\_open(queue\_name, O\_CREAT | O\_RDWR, 0666, &attr);  
 if (mq == (mqd\_t)-1) {  
 perror("mq\_open");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建子进程  
 child\_pid = fork();  
  
 if (child\_pid == -1) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 if (child\_pid == 0) {  
 // 子进程重新打开消息队列并发送消息三次  
 mqd\_t child\_mq;  
  
 child\_mq = mq\_open(queue\_name, O\_RDWR);  
 if (child\_mq == (mqd\_t)-1) {  
 perror("mq\_open (child)");  
 exit(EXIT\_FAILURE);  
 }  
  
 for (int i = 0; i < 3; i++) {  
 char message[256];  
 sprintf(message, "Hello from child process, message %d!", i + 1);  
 if (mq\_send(child\_mq, message, strlen(message) + 1, 0) == -1) {  
 perror("mq\_send (child)");  
 exit(EXIT\_FAILURE);  
 }  
 }  
  
 mq\_close(child\_mq); // 关闭子进程的消息队列  
 exit(EXIT\_SUCCESS);  
 } else {  
 // 父进程接收消息  
 char received\_message[256];  
 unsigned int priority;  
  
 // 等待子进程退出  
 wait(NULL);  
  
 // 从消息队列接收消息  
 while (mq\_receive(mq, received\_message, sizeof(received\_message), &priority) != -1) {  
 printf("Parent received: %s\n", received\_message);  
 }  
  
 if (errno != EAGAIN) {  
 perror("mq\_receive (parent)");  
 exit(EXIT\_FAILURE);  
 }  
 }  
  
 // 关闭和删除消息队列  
 mq\_close(mq);  
 mq\_unlink(queue\_name);  
  
 return 0;  
}

<https://zhuanlan.zhihu.com/p/638279470>

<https://zhuanlan.zhihu.com/p/389623919?utm_id=0>

<https://zhuanlan.zhihu.com/p/651152391>

### 共享内存（Shared Memory）：

* 共享内存允许多个进程共享同一块内存区域，以便它们可以直接读写其中的数据。
* 这是一种高效的IPC机制，但需要额外的同步机制来避免竞态条件。
* 共享内存本质上就是内存中的一块区域，用于进程间通信使用。

所需头文件：

#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/shm.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| int shmget(key\_t key, size\_t size, int shmflg); | 创建共享内存 | key：一个用于唯一标识共享内存段的键值。可以使用 ftok 函数生成键值，也可以使用IPC\_PRIVATE来创建一个新的共享内存段。 size：指定要创建或获取的共享内存段的大小（以字节为单位）。 shmflg：指定共享内存的访问权限和选项，通常使用 IPC\_CREAT 标志来创建共享内存段。 | 如果成功，shmget函数返回0。 如果失败，返回-1，并设置errno以指示错误的原因。 |
| void \*shmat(int shmid, const void \*shmaddr, int shmflg); | 创建共享内存后还不能直接使用，需要找到内存地址后才能使用，即连接。 | shmid：共享内存段的标识符，通常是由shmget函数获取的共享内存标识符。 shmaddr：通常设置为NULL，表示让操作系统自动选择合适的地址来连接共享内存。如果需要指定连接地址，可以提供一个非空指针。 shmflg：用于指定连接的选项，通常设置为0。 | 成功，shmat函数返回指向连接共享内存的地址的指针。失败，返回(void\*)-1，并设置errno以指示错误的原因。 |
| int shmdt(const void \*shmaddr); | 当使用完毕后，需要分离挂接的共享内存。 | shmaddr：指向共享内存连接地址的指针，通常是由shmat函数返回的连接共享内存的地址。 | 如果成功，shmdt函数返回0。 如果失败，返回-1，并设置errno以指示错误的原因。 |
| int shmctl(int shmid, int cmd, struct shmid\_ds \*buf); | 销毁共享内存 | shmid：共享内存段的标识符，通常是由shmget函数获取的共享内存标识符。 cmd：用于指定执行的操作命令，可以是以下之一： IPC\_RMID：删除共享内存段。 IPC\_STAT：获取共享内存段的信息。 IPC\_SET：设置共享内存段的信息。 buf：指向struct shmid\_ds结构的指针，用于传递共享内存段的信息。 | 如果成功，shmctl函数根据操作不同返回不同的值： 如果cmd为IPC\_RMID，则返回0表示删除成功。 如果cmd为IPC\_STAT，则返回0表示成功，并将共享内存段的信息填充到buf中。 如果cmd为IPC\_SET，则返回0表示设置成功。 如果失败，返回-1，并设置errno以指示错误的原因。 |

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <sys/ipc.h>  
#include <sys/shm.h>  
#include <unistd.h>  
#include <sys/wait.h>  
  
#define SHM\_SIZE 1024 // 共享内存的大小  
  
int main() {  
 key\_t key = ftok("shmfile", 65);  
 int shmid = shmget(key, SHM\_SIZE, 0666 | IPC\_CREAT); // 创建共享内存  
  
 if (shmid == -1) {  
 perror("shmget");  
 exit(1);  
 }  
  
 char \*shared\_memory = (char \*)shmat(shmid, NULL, 0); // 连接共享内存  
  
 if (shared\_memory == (void \*)-1) {  
 perror("shmat");  
 exit(1);  
 }  
  
 // 创建子进程  
 pid\_t child\_pid = fork();  
  
 if (child\_pid == -1) {  
 perror("fork");  
 exit(1);  
 }  
  
 if (child\_pid == 0) { // 子进程  
 printf("Child Process: Writing to shared memory\n");  
 sprintf(shared\_memory, "Hello from child process!");  
  
 // 分离共享内存  
 if (shmdt(shared\_memory) == -1) {  
 perror("shmdt");  
 exit(1);  
 }  
 } else { // 父进程  
 wait(NULL); // 等待子进程完成  
 printf("Parent Process: Reading from shared memory\n");  
 printf("Data read from shared memory: %s\n", shared\_memory);  
  
 // 分离共享内存  
 if (shmdt(shared\_memory) == -1) {  
 perror("shmdt");  
 exit(1);  
 }  
  
 // 删除共享内存  
 if (shmctl(shmid, IPC\_RMID, NULL) == -1) {  
 perror("shmctl");  
 exit(1);  
 }  
 }  
  
 return 0;  
}

### 信号（Signals）：

* 信号是一种轻量级IPC机制，用于通知进程发生了某个事件或异常。
* 进程可以发送信号给其他进程，也可以定义自己的信号处理函数来响应接收到的信号。

所需头文件：

#include <unistd.h>  
#include <signal.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| sighandler\_t signal(int signum, sighandler\_t handler); | 安装信号 | signum：代表要设置处理程序的信号的编号，例如SIGINT、SIGTERM等。 handler：是一个函数指针，指向您希望在接收到指定信号时调用的处理程序函数。处理程序函数必须接受一个整数参数，代表接收到的信号编号，通常命名为signum。 | signal 函数的返回值是一个函数指针类型 sighandler\_t，该函数指针指向之前与该信号相关联的信号处理程序。这是为了让您可以在设置新的信号处理程序之前保存旧的处理程序，以便稍后恢复或查询它。如果之前没有与该信号相关联的处理程序，返回值通常为 SIG\_DFL（默认处理程序）或 SIG\_IGN（忽略信号）。 |
| int kill(pid\_t pid, int sig); | 给进程发送信号 | pid 参数表示要发送信号的目标进程的进程ID（Process ID）。可以使用正整数来指定特定的进程，也可以使用负数来指定进程组（通常为负数）或会话ID（通常为负数）。 sig 参数表示要发送的信号编号，如SIGTERM、SIGKILL等。 | 如果 "kill" 函数成功发送信号，它将返回0。 如果出现错误，它将返回-1，并设置全局变量 errno 来指示具体的错误原因。 |
| unsigned int alarm(unsigned int seconds); | 进程定时器闹钟信号 | seconds 参数表示定时器的时间间隔，以秒为单位。当 seconds 参数为正整数时，定时器将在指定的秒数后触发SIGALRM信号。如果 seconds 参数为0，表示取消当前的定时器。定时时间到后会给进程发送 SIGALRM 信号。 | 函数在成功设置定时器后返回之前的剩余时间，如果之前没有定时器或者新的定时器覆盖了之前的定时器，返回值为0。如果之前的定时器剩余时间还未到期，则返回之前剩余的秒数。 |

信号编号与说明：

|  |  |  |
| --- | --- | --- |
| 信号名 | 编号 | 说明 |
| SIGHUP | 1 | 如果终端界面检测到一个连接断开， 则将此信号送给与该终端相关的进程。 |
| SIGINT | 2 | 当用户按中断键(一般采用 Delete或Ctrl+C)时，终端驱动程序产生这个信号并将信号送给前台进程组中的每一个进程。当一个进程在运行时失控，特别是它正在屏幕上产生大量不需要的输出时，常用此信号终止它。 |
| SIGQUIT | 3 | 当用户在终端上按退出键(一般采用CtrI+C) 时，产生此信号，并送至前台进程组中的所有进程。 |
| SIGILL | 4 | 此信号指示进程已执行一条非法硬件指令。 |
| SIGTRAP | 5 | 指示一个实现定义的硬件故障。 |
| SIGABRT | 6 | 调用abort()函数时产生此信号，进程异常终止。 |
| SIGEMT | 7 | 指示一个实现定义的硬件故障。 |
| SIGFPE | 8 | 此信号表示一个算术运算异常，例如除以0，浮点溢出等。 |
| SIGKILL | 9 | 强制终止进程。不能被捕捉、阻塞或忽略。 |
| SIGBUS | 10 | 指示一个实现定义的硬件故障。 |
| SIGSEGV | 11 | 指示进程进行了一次无效的存储访问。 |
| SIGSYS | 12 | 指示一个无效的系统调用。由于某种未知原因，某个进程执行了一条系统调用命令，但是调用命令所用的参数无效。 |
| SIGPIPE | 13 | 如果在读进程时已终止写管道，则产生此信号。 |
| SIGALRM | 14 | 超过用alarm()函数设置的时间时产生此信号。 |
| SIGTERM | 15 | 这是由kill命令发送的系统默认终止信号。 |
| SIGURG | 16 | 此信号通知进程已经发生一个紧急情况。在网络连接上，接到非规定波特率的数据时，此信号可选择地产生。 |
| SIGCHLD | 17 | 在一个进程终止或停止时，SIGCHLD信号被送给其父进程。如果希望从父进程中了解其子进程的状态改变，则应捕捉此信号。信号捕捉函数中通常要调用wait()函数以取得子进程ID和其终止状态。 |
| SIGCONT | 18 | 此作业控制信号送给需要继续运行的处于停止状态的进程。如果接收到此信号的进程处于停止状态，则操作系统的默认动作是使该停止的进程继续运行，否则默认动作是忽略此信号。 |
| SIGSTOP | 19 | 这是一个作业控制信号，它停止一个进程。 |
| SIGTSTP | 20 | 交互停止信号，当用户在终端上按挂起键(一般采用Ctrl+Z) 时，终端驱动程序产生此信号。 |
| SIGTTIN | 21 | 当一个后台进程组进程试图读其控制终端时，终端驱动程序产生此信号。 |
| SIGTTOU | 22 | 当一个后台进程组进程试图写其控制终端时产生此信号。 |
| SIGIO | 23 | 此信号指示一个异步IO事件。 |

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <signal.h>  
  
volatile int exit\_flag = 0; // 用于标志子进程是否退出  
  
void signal\_handler(int sig) {  
 if (sig == SIGUSR1) {  
 printf("子进程收到了自定义信号 SIGUSR1\n");  
 // 在这里执行子进程需要的操作  
 } else if (sig == SIGUSR2) {  
 printf("子进程收到了退出信号 SIGUSR2\n");  
 exit\_flag = 1; // 设置退出标志  
 }  
}  
  
int main() {  
 pid\_t child\_pid;  
  
 // 创建子进程  
 if ((child\_pid = fork()) == -1) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 if (child\_pid == 0) {  
 // 子进程  
 signal(SIGUSR1, signal\_handler); // 设置自定义信号处理函数  
 signal(SIGUSR2, signal\_handler); // 设置退出信号处理函数  
 printf("子进程等待信号...\n");  
 while (!exit\_flag) {  
 // 子进程持续运行以等待信号  
 sleep(1);  
 }  
 printf("子进程退出\n");  
 } else {  
 // 父进程  
 sleep(2); // 等待一段时间，确保子进程已经准备好  
 printf("父进程向子进程发送 SIGUSR1 信号\n");  
 if (kill(child\_pid, SIGUSR1) == -1) {  
 perror("kill");  
 exit(EXIT\_FAILURE);  
 }  
 sleep(1); // 等待一秒钟  
 printf("父进程向子进程发送 SIGUSR2 退出信号\n");  
 if (kill(child\_pid, SIGUSR2) == -1) {  
 perror("kill");  
 exit(EXIT\_FAILURE);  
 }  
 sleep(2); // 等待一段时间，以确保子进程退出  
 printf("父进程结束\n");  
 }  
  
 return 0;  
}

在这个示例中，父进程首先发送SIGUSR1自定义信号，然后等待1秒钟后发送SIGUSR2退出信号给子进程。子进程捕捉这两个信号并在处理SIGUSR2信号后退出。父进程等待一段时间以确保子进程退出，然后结束。请注意，子进程在一个循环中不断休眠以等待信号的到来。

请注意，当使用信号和多线程程序时，应格外小心，因为信号处理函数的行为可能会受到限制。此外，某些系统函数在信号处理函数中可能不安全。

### 信号量（Semaphores）：

* 信号量是一种用于控制多个进程对共享资源的访问的IPC机制。
* 它们可以用来解决竞态条件和进程同步问题。

所需头文件:

#include <semaphore.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value); | 创建或打开 POSIX 信号量（Semaphore） | name：信号量的名称，这个名称在系统中必须是唯一的，用于标识信号量。 oflag：打开信号量的标志位，可以是以下之一或它们的组合： O\_CREAT：如果信号量不存在，则创建它。 O\_EXCL：与 O\_CREAT 一起使用时，如果信号量已经存在，则返回错误。 O\_RDWR：允许读写信号量的权限。 O\_RDONLY：只允许读信号量的权限。 O\_WRONLY：只允许写信号量的权限。 mode：创建信号量时，指定的权限位，通常以八进制表示，例如 0666。 value：新创建的信号量的初始值，表示可用资源的数量。 | sem\_open 函数返回一个指向 sem\_t 类型的指针，该指针可以在后续的操作中用于引用信号量。如果创建或打开信号量失败，它将返回 SEM\_FAILED。 |
| int sem\_post(sem\_t \*sem); | 增加信号 | sem: 指向要增加的信号量的指针。 | 同上 |
| int sem\_wait(sem\_t \*sem); | 基于信号计数阻塞 | sem: 指向要减少的信号量的指针。 | 同上 |
| int sem\_trywait(sem\_t \*sem); | 减小信号计数 | sem: 指向要尝试减少的信号量的指针。 | 同上 |
| int sem\_destroy(sem\_t \*sem); | 销毁信号状态 | sem: 指向要销毁的信号量的指针。 | 同上 |

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/wait.h>  
#include <fcntl.h>  
#include <sys/stat.h>  
#include <semaphore.h>  
  
int main() {  
 sem\_t \*semaphore;  
 pid\_t pid1, pid2;  
  
 // 在父进程中创建一个名为 "mysemaphore" 的 POSIX 信号量  
 semaphore = sem\_open("mysemaphore", O\_CREAT, 0644, 1);  
 if (semaphore == SEM\_FAILED) {  
 perror("sem\_open");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建第一个子进程  
 pid1 = fork();  
 if (pid1 < 0) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 if (pid1 == 0) { // 子进程1  
 // 在子进程中打开相同的信号量  
 semaphore = sem\_open("mysemaphore", 0);  
 if (semaphore == SEM\_FAILED) {  
 perror("sem\_open");  
 exit(EXIT\_FAILURE);  
 }  
 sem\_wait(semaphore); // 等待信号量  
 printf("Child process 1 is running\n");  
 sleep(2); // 模拟一些工作  
 sem\_post(semaphore); // 释放信号量  
 exit(0);  
 }  
  
 // 创建第二个子进程  
 pid2 = fork();  
 if (pid2 < 0) {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
  
 if (pid2 == 0) { // 子进程2  
 // 在子进程中打开相同的信号量  
 semaphore = sem\_open("mysemaphore", 0);  
 if (semaphore == SEM\_FAILED) {  
 perror("sem\_open");  
 exit(EXIT\_FAILURE);  
 }  
 sem\_wait(semaphore); // 等待信号量  
 printf("Child process 2 is running\n");  
 sleep(2); // 模拟一些工作  
 sem\_post(semaphore); // 释放信号量  
 exit(0);  
 }  
  
 // 等待两个子进程结束  
 waitpid(pid1, NULL, 0);  
 waitpid(pid2, NULL, 0);  
  
 // 关闭并删除信号量  
 sem\_close(semaphore);  
 sem\_unlink("mysemaphore");  
  
 return 0;  
}

### 互斥锁（Mutexes）：

* 互斥锁用于确保在多个进程之间对共享资源的互斥访问。
* 进程间没有值接的互斥锁，需要手动设置共享内存。

进程间的互斥锁相对于线程中的互斥锁多了一个共享内存映射的操作，就是申请一块共享内存，用于存放锁的数据，然后在就可以在不同进程之间使用同一个互斥锁了。

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <pthread.h> // 添加pthread头文件  
#include <sys/wait.h>  
#include <sys/mman.h>  
#include <fcntl.h>  
  
struct mt  
{  
 int num;  
 pthread\_mutex\_t mutex;  
 pthread\_mutexattr\_t mutexattr;   
};  
  
int main(int argc, const char \*argv[])  
{  
 pid\_t pid;  
 struct mt \*mm;  
 mm = mmap(NULL, sizeof(\*mm), PROT\_READ | PROT\_WRITE, MAP\_SHARED | MAP\_ANON, -1, 0);  
 if (mm == MAP\_FAILED)  
 {  
 perror("FAIL mmap");  
 return -1;  
 }  
 mm->num = 0;  
 pthread\_mutexattr\_init(&mm->mutexattr); // 初始化mutex属性  
 pthread\_mutexattr\_setpshared(&mm->mutexattr, PTHREAD\_PROCESS\_SHARED);  
 pthread\_mutex\_init(&mm->mutex, &mm->mutexattr); // 初始化一把锁  
  
 pid = fork();  
 if (pid < 0)  
 {  
 perror("Fork failed");  
 return -1;  
 }  
 else if (pid == 0) // 子,负责售票  
 {  
 while (mm->num < 10)  
 {  
 pthread\_mutex\_lock(&mm->mutex);  
 mm->num = mm->num + 1;  
 printf("儿子卖出一张，现在卖出个数为%d\n", mm->num);  
 pthread\_mutex\_unlock(&mm->mutex);  
 sleep(1);  
 }  
 exit(1);  
 }  
 else  
 {  
 while (mm->num < 10)  
 {  
 sleep(1);  
 pthread\_mutex\_lock(&mm->mutex);  
 mm->num = mm->num + 1;  
 printf("父亲卖出一张,当前卖出个数为%d\n", mm->num);  
 pthread\_mutex\_unlock(&mm->mutex);  
 }  
 exit(1);  
 }  
 wait(NULL);  
 pthread\_mutexattr\_destroy(&mm->mutexattr); // 销毁锁的属性对象  
 pthread\_mutex\_destroy(&mm->mutex); // 销毁锁  
 return 0;  
}

此代码创建了一个共享内存区域（使用mmap），然后在该共享内存区域中放置一个整数num和一个共享的互斥锁mutex。子进程和父进程都可以通过这个共享内存来访问和修改num的值，但是在修改时需要先获取互斥锁以确保互斥访问。

子进程不断卖票，每次卖一张票就通过互斥锁保护地修改num的值，并打印出卖出的票数。父进程也以类似的方式卖票。两个进程都在while循环中，直到num的值达到10时退出。

最后，代码在程序结束前销毁了互斥锁和互斥锁属性对象，并等待子进程结束。

### 套接字（Sockets）：

* 套接字是计算机网络编程的基础工具之一，用于实现进程之间的通信，无论是在本地计算机上还是跨网络。
* 套接字允许不同主机之间的进程进行通信。这意味着您可以通过网络与远程计算机上运行的进程进行数据交换。
* 套接字也可用于在同一主机上的不同进程之间进行本地通信，这被称为本地套接字（Unix Domain Socket）。
* 套接字可以使用不同的协议进行通信，如TCP、UDP和RAW等。不同协议提供不同的通信特性，例如TCP提供可靠的面向连接的通信，而UDP提供不可靠的面向数据报的通信。
* 套接字可以支持面向连接的通信（如TCP套接字）和面向数据报的通信（如UDP套接字）。面向连接的套接字提供可靠的通信，而面向数据报的套接字提供无连接的通信。
* 套接字支持双向通信，允许进程之间进行双向数据传输。客户端可以向服务器发送数据，并从服务器接收响应。
* 套接字支持全双工通信，这意味着在通信的两端，数据可以同时进行双向传输，而不需要等待对方完成。
* 套接字编程具有高度的可扩展性，允许您同时处理多个连接或请求，以满足不同的应用需求。
* 套接字编程不仅限于C语言，还支持多种编程语言，包括Python、Java、C++等。
* 套接字通信通常可以受到权限和安全性的控制，以确保只有授权的进程或用户能够访问和使用套接字。
* 套接字编程在不同操作系统上都是可用的，尽管在不同操作系统上可能需要不同的配置和设置。

总的来说，套接字是网络编程中非常重要的工具，它们提供了灵活的通信机制，可以用于构建各种不同类型的网络应用程序，从简单的客户端/服务器应用到复杂的分布式系统。不同类型的套接字提供了不同级别的控制和性能，使得开发人员可以根据特定需求选择合适的套接字类型。

所需头文件：

#intclude <sys/socket.h>  
#include <arpa/inet.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| int socket(int domain, int type, int protocol); | 创建一个套接字 | domain：AF\_INET、AF\_INET6、AF\_UNIX,type：SOCK\_STREAM、SOCK\_DGRAM 流式套接字（TCP）/数据报格式(UDP)套接字,protocol:0 | 返回值：成功：新套接字所对应的文件描述符 失败：-1 error |
| int bind(int sockfd, const struct sockaddr \*addr,socklen\_t addrlen); | 给socket绑定一个 地址结构（IP+port） | sockfd，socket 函数返回值,struct sockaddr\_in addr;套接字地址，addrlen，地址结构大小。 | 成功：0 失败：-1 |
| int listen(int sockfd, int backlog); | 设置同时与服务器建立连接的上限数（同时进行3次握手的客户端数量） | sockfd，socket 函数返回值，backlog，上限数值 最大值 128。 | 成功：0 失败：-1 |
| int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); | 阻塞等待客户端连接，成功的话，返回一个与客户端成功连接的socket文件描述符 | sockfd：socket //函数返回值 addr： //传出参数 成功与服务器建立连接的那个客户端的地址结构（IP+port） socklen\_t clit\_addr\_len = sizeof(addr); addrlen： //传入传出 &clit\_addr\_len //入：addr的大小 出：客户端addr实际大小 | 成功：能与服务器进行数据通信的socket对应的文件描述符 失败： -1 |
| int connect(int sockfd, const struct sockaddr \*addr,socklen\_t addrlen); | 使用现有的socket与服务器建立连接 | sockfd：socket //函数返回值 struck socketaddr\_in serv\_addr; //服务器地址结构 serv\_addr.sin\_family = AF\_INET; serv\_addr.sin\_port = 9527 ; //跟服务器bind时设定的port完全一致 inet\_pton(AF\_INET, "服务器地址", &serv\_addr.sin\_addr.s\_addr); addr： //传入参数 服务器地址结构 addrlen： //服务器的地址结构的大小 | 返回值： 成功 0 失败 -1 error |

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <sys/un.h>  
  
#define SOCKET\_PATH "/tmp/my\_socket"  
  
int main() {  
 int sockfd;  
 struct sockaddr\_un server\_addr, client\_addr;  
 char buffer[256];  
 pid\_t pid;  
  
 // 创建本地套接字  
 sockfd = socket(AF\_UNIX, SOCK\_DGRAM, 0);  
 if (sockfd < 0) {  
 perror("Error opening socket");  
 exit(1);  
 }  
  
 // 设置本地套接字地址  
 server\_addr.sun\_family = AF\_UNIX;  
 strcpy(server\_addr.sun\_path, SOCKET\_PATH);  
  
 // 绑定套接字到地址  
 if (bind(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {  
 perror("Error binding socket");  
 exit(1);  
 }  
  
 // 创建子进程  
 pid = fork();  
 if (pid < 0) {  
 perror("Fork failed");  
 exit(1);  
 } else if (pid == 0) {  
 // 子进程，向父进程发送消息  
 printf("Child process sending message...\n");  
 sleep(2); // 等待父进程准备好接收消息  
 snprintf(buffer, sizeof(buffer), "Hello from child process (PID %d)!", getpid());  
 sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr \*)&server\_addr, sizeof(server\_addr));  
 } else {  
 // 父进程，接收子进程发送的消息  
 printf("Parent process waiting for message...\n");  
 socklen\_t client\_len = sizeof(client\_addr);  
 ssize\_t num\_bytes = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr \*)&client\_addr, &client\_len);  
 if (num\_bytes < 0) {  
 perror("Error receiving data");  
 exit(1);  
 }  
 buffer[num\_bytes] = '\0';  
 printf("Received message in parent process: %s\n", buffer);  
  
 // 关闭套接字  
 close(sockfd);  
  
 // 删除套接字文件  
 unlink(SOCKET\_PATH);  
 }  
  
 return 0;  
}

在这个示例中，我们创建了一个本地套接字，并使用SOCK\_DGRAM作为套接字类型，因此它支持面向数据报的通信。然后，我们创建了一个子进程，子进程向父进程发送消息，父进程接收子进程发送的消息。

### 文件锁（File Locking）：

* 文件锁允许多个进程协调对文件的访问，以防止多个进程同时写入或读取同一文件。
* 文件锁是跨进程的，允许不同进程之间使用锁来同步对共享文件的访问。这使得多个独立运行的进程可以协调工作。
* Linux文件锁有两种主要类型：共享锁和排他锁。共享锁允许多个进程同时持有锁并进行读取操作，而排他锁只允许一个进程持有锁进行写操作。这种灵活性允许多进程同时读取相同的文件，但只允许一个进程进行写操作。
* 文件锁可以以阻塞或非阻塞方式进行设置。阻塞模式下，如果进程尝试获得一个已被其他进程持有的锁，它会被阻塞，直到锁可用。在非阻塞模式下，进程会立即返回一个错误，而不会等待锁可用。
* 文件锁可以设置为锁定整个文件，也可以设置为锁定文件的特定区域。这使得进程可以选择性地锁定文件的一部分，而不是整个文件。
* 文件锁可以设置为临时锁或持久锁。临时锁在进程关闭文件描述符或退出时自动释放，而持久锁则会一直保持，直到显式释放或进程终止。
* Linux文件锁通常会被子进程继承。这意味着如果一个进程持有一个文件锁，然后创建了子进程，子进程也会继承相同的锁。
* 文件锁可以设置优先级，允许某些进程获得锁的优先权。例如，高优先级的进程可以在低优先级的进程之前获得锁。
* 如果持有锁的进程异常终止，内核通常会自动释放锁，以确保其他进程能够继续访问文件。

linux锁存在强制锁（mandatory lock）和劝告锁（advisory lock）。所谓强制锁，就是一个进程获取了那把锁（只有一把钥匙），只有一个进程可以操作，别的进程无能为力。所谓劝告锁，本质是一种协议，你访问文件前，先检查锁，这时候锁才其作用，如果不检查就要读写，那么劝告锁没有任何的作用。为了遵守协议，读写前先检查锁的那些进程通常称为合作进程。linux 下可以使用 fcntl，flock，lockf 三个函数对文件上锁。

所需头文件：

#include <unistd.h>  
#include <fcntl.h>  
#include <sys/file.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| int fcntl(int fd, int cmd, struct flock \*lock) | 文件锁操作，系统调用 | fcntl函数用于对文件描述符 fd 执行各种控制操作，其中包括文件锁。 cmd 参数用于指定操作类型，通常是 F\_SETLK（设置锁）、F\_SETLKW（设置锁并等待）或 F\_GETLK（获取锁信息）。 struct flock 结构体包含了锁的信息，如锁的类型、起始偏移量、锁定字节数等。可以使用此结构来指定要设置或获取的锁的属性。 | 成功：0 失败：-1 |
| int flock(int fd, int operation) | 文件锁操作，系统调用 | flock函数用于对文件描述符 fd 执行文件级别的锁操作。operation 参数用于指定操作类型，可以是 LOCK\_SH（共享锁）、LOCK\_EX（独占锁）或 LOCK\_UN（释放锁）等。 | 成功：0 失败：-1 |
| int lockf(int fd, int cmd, off\_t len) | 文件锁操作，C 库函数 | lockf函数用于对文件描述符 fd 执行文件级别的锁操作。cmd 参数用于指定操作类型，通常是 F\_LOCK（锁定文件）、F\_TLOCK（尝试锁定文件）或 F\_ULOCK（解锁文件）。len 参数指定锁定的字节数。如果设置为0，则锁定整个文件。 | 成功：0 失败：-1 |

强制锁示例:

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <errno.h>  
  
int main() {  
 int fd = open("example.txt", O\_RDWR | O\_CREAT, 0666);  
  
 if (fd == -1) {  
 perror("Failed to open file");  
 return 1;  
 }  
  
 struct flock lock;  
 lock.l\_type = F\_WRLCK; // 独占写锁  
 lock.l\_whence = SEEK\_SET;  
 lock.l\_start = 0;  
 lock.l\_len = 0; // 锁定整个文件  
  
 // 尝试设置写锁，如果文件已被锁定，则等待  
 if (fcntl(fd, F\_SETLKW, &lock) == -1) {  
 perror("Failed to set lock");  
 close(fd);  
 return 1;  
 }  
  
 printf("Lock set successfully. Press Enter to release the lock.\n");  
 getchar(); // 等待用户按下 Enter 键  
  
 // 释放锁  
 lock.l\_type = F\_UNLCK;  
 if (fcntl(fd, F\_SETLK, &lock) == -1) {  
 perror("Failed to release lock");  
 } else {  
 printf("Lock released successfully.\n");  
 }  
  
 close(fd);  
 return 0;  
}

劝告锁示例:

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/file.h>  
#include <fcntl.h>  
  
int main(int argc, char \*\*argv) {  
 if (argc > 1) {  
 int fd = open(argv[1], O\_WRONLY);  
 if (fd == -1) {  
 perror("open");  
 exit(1);  
 }  
  
 struct flock lock;  
 lock.l\_type = F\_WRLCK; // 写锁（劝告锁）  
 lock.l\_start = 0;  
 lock.l\_whence = SEEK\_SET;  
 lock.l\_len = 0;  
 lock.l\_pid = getpid();  
  
 int ret = fcntl(fd, F\_SETLK, &lock); // 设置劝告锁  
 if (ret == -1) {  
 perror("fcntl");  
 } else if (ret == 0) {  
 printf("Advisory lock set successfully.\n");  
 }  
  
 // 程序继续执行，但不会阻塞其他进程  
  
 sleep(10); // 模拟程序持有锁的时间  
  
 lock.l\_type = F\_UNLCK; // 解锁  
 ret = fcntl(fd, F\_SETLK, &lock); // 解除劝告锁  
 if (ret == -1) {  
 perror("fcntl");  
 } else if (ret == 0) {  
 printf("Advisory lock released successfully.\n");  
 }  
  
 close(fd);  
 }  
 return 0;  
}

Linux中的劝告（advisory）文件锁是一种锁定文件的方式，它不会阻塞其他进程的访问，而是仅仅向其他进程发出锁定建议。其他进程可以选择是否遵守这个建议。

这些IPC机制提供了不同的抽象层次和用途，可以根据应用程序的需求选择合适的机制。IPC在操作系统和分布式系统中起着重要作用，帮助不同进程或线程之间协作，共享资源，以及实现各种应用程序。

原文链接：<https://blog.csdn.net/weixin_47077788/article/details/128100959>

## 10. 网络编程

网络编程是计算机编程的一个重要分支，它涉及到在网络环境中使不同的计算机设备能够进行数据交换和通讯。下面我们将探索网络编程的基本概念、技术和协议，包括TCP/IP模型、套接字编程以及客户端-服务器架构。理解如何在不同的网络环境中进行有效的数据传输，以及如何构建可靠且高效的网络应用程序。无论是构建网站、开发网络服务还是实现远程通信，网络编程都是一项基础且关键的技能。

### 网络基础概念

计算机的通信是通过通信介质实现的，比如网线，光纤等。以网线为例，两台计算机之间的通信，可以直接拉一条线，三台计算机之间如果想要相互通信就要拉三根线，到了四台计算机的时候就需要拉 6 根线了。如果这样下去的话，所需要的线将会是指数形式增加的，非常消耗资源。于是将计算机分组，组内设置一个通信组长，就这样一级一级的组织起来，形成的类似与网状的连接结构，被称为互联网络。所有连网的计算机都是网络上的是一个节点，这就是互联网现在的基本连接形式的由来。

#### 基本协议

在原始的互联网中设想，计算机想要通信时，直接将信息发送到网络中，网络中的每一台计算机都收到了信息，然后想要回复的计算机，以同样的方式，将信息发送到网络中，让所有的计算机，接受到信息。这样做的有什么不可忍受的缺点呢？第一、你发的信息所有人都知道了，没有私密性。第二、当很多的计算机都想要进行交流的时候，互联网络中的信息将会无限的放大，以致超过了互联网络能能够承受的通信极限，很多计算机因此通信受阻。为了解决这些缺点，互联网的开创者们从门牌号中获得了灵感，给每个设备都设定一个地址，然后消息就按照地址进行传递。同时将交流的信息分割成信息包，分组定点传送。这样既能解决隐私的问题，又解决了信息线路信息爆炸的问题。而约定好的地址，和信息分组的方式官方其名互联协议，计算机必须遵循这个协议才能和其他计算机正常交流。

##### 网络的分层模型：

[OSI](https://so.csdn.net/so/search?q=OSI&spm=1001.2101.3001.7020) 7层模型：

* 应用层：主要就是应用程序使用的协议， ftp ssh email http 协议等等。
* 表示层：进行编解码和翻译工作。
* 会话层：建立会话和保持会话 。
* 传输层：定义了端到端的传输， TCP， UDP 协议等等。
* 网络层：定义了点到点的传输，IP 协议 --路由器。
* 数据链路层：数据校验，定义了数据格式 --帧，ARP 协议 RARP 协议。
* 物理层：通信介质-双绞线，光纤，调制解调器 modemn

##### TCP/IP 协议：

TCP/IP（传输控制协议/网际协议）是一组用于互联网数据通信的核心协议。这个协议栈由两部分主要协议组成：TCP 和 IP，它们共同确保了互联网的可靠、有效运行，后面添加了 UDP 协议，作为 TCP 轻量化的补充。

IP（网际协议）工作在网络层，主要有两个职责。第一、负责设定地址系统，为互联网上的每一台设备都分配一个唯一的地址标识符。第二、负责数据包的分割与传递，根据数据包的头部信息，将数据包传递到对应的连路上去。IP 协议本身不能保证数据包的顺序、完整性或可靠传输。

TCP（传输控制协议）工作在传输层，主要职责有两个。第一、通过三次握手，四次挥手的机制确保双方都准备好进行数据交换，建立一个连接通道。第二、对数据包进行分包，添加顺序号，通过滑动窗口等机制保证数据按顺序、完整无误地传输到接收端。同时 TCP 兼任了流量控制和拥塞控制的职能，保证整个互联网络的正常工作。TCP 是面向可靠的连接。

UDP（传输控制协议）工作在传输层，就是对 IP 协议的简单封装而来。具有轻量化，快速的特点，不可靠，信息容易丢失。

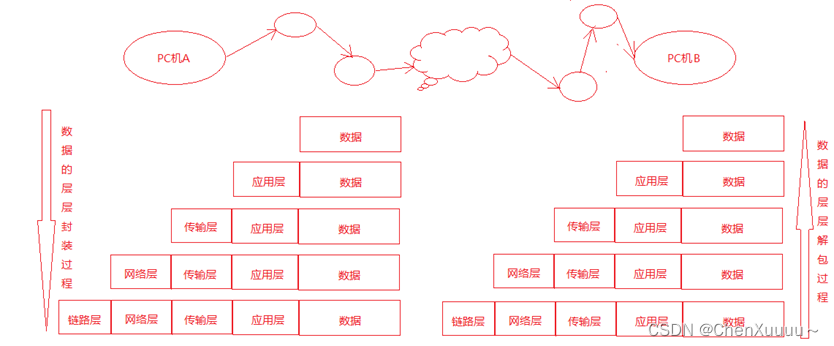
##### TCP/IP 网络简化四层模型：

* 应用层：对应会话层，表示层，应用层
* 传输层：对应传输层
* 网络层：对应网络层
* 网络接口层：对应于物理层和数据链路层

#### 传输模型

在互联网中存在这么多的协议，他们能够正常协同工作的原因就是网络分层。在基本协议中说明了通信分层，通过通信分层，我们能够划分协议在什么时候工作，什么时候开始工作。

数据的加工<->传输<->解析的工作图：



数据从应用层出发时，被格式化成了相关协议规定的形式，比如 HTTP 行条目传输。

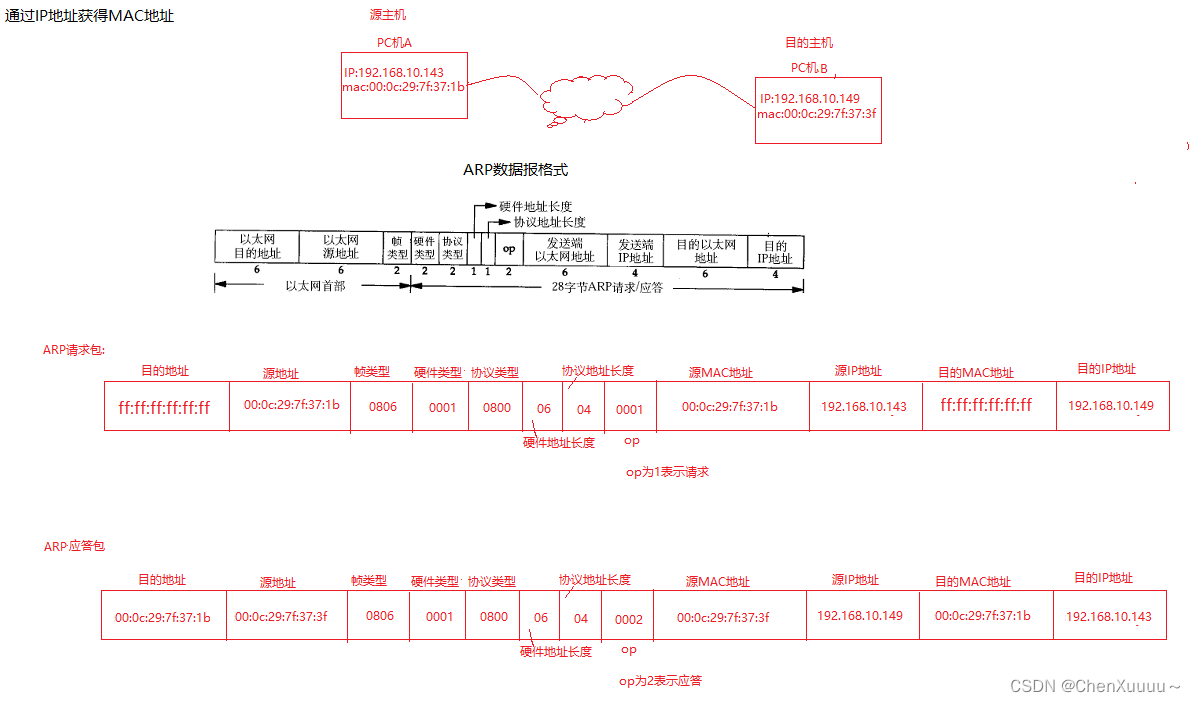
当数据被送到传输层的时候，TCP或UDP 协议会将协议状态信息添加到数据的头部。

当数据经过网络层时，会在头部添加上 IP 协议信息的信息。

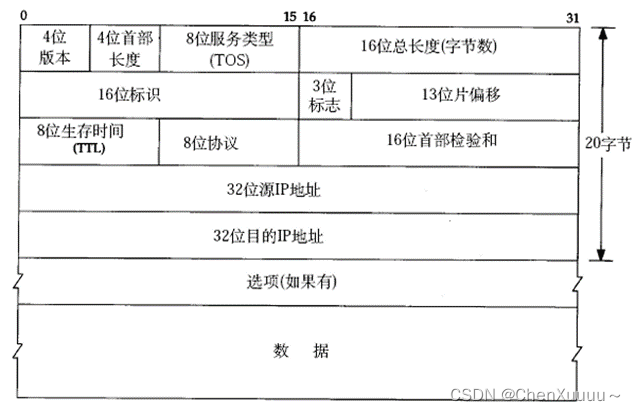
当数据经过链路层时，会被添加上网卡 mac 地址的信息。

同样的，当计算机有数据传来时，每经过一层，就会被去掉当前层的协议信息，直到信息被送到应用程序让应用程序进行处理。

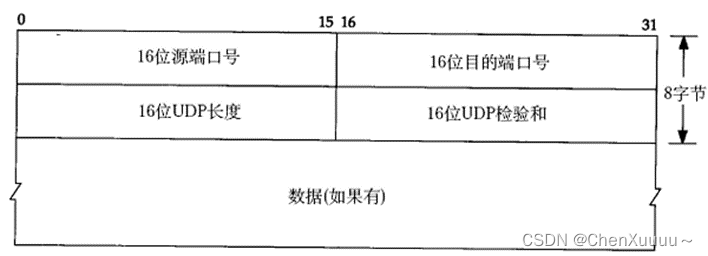
ARP协议信息：



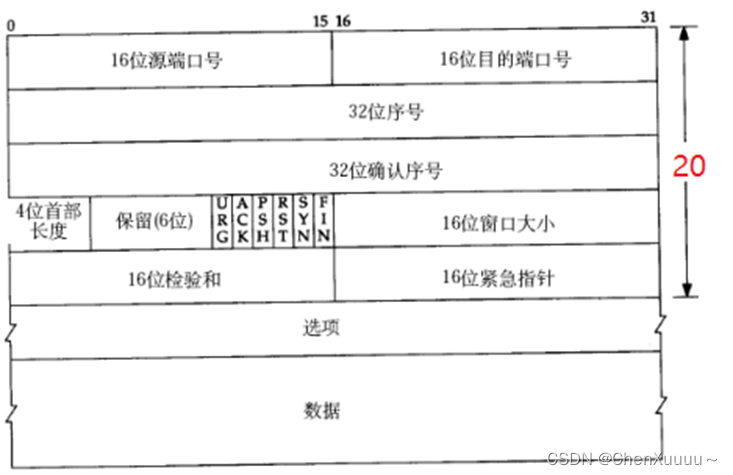
IP协议信息：



UDP 协议信息：



TCP 协议信息:



### 编程方式

网络编程的接口就是套接字，网络传输就是将套接字中的数据经过多个协议和其他的计算机进行交换。

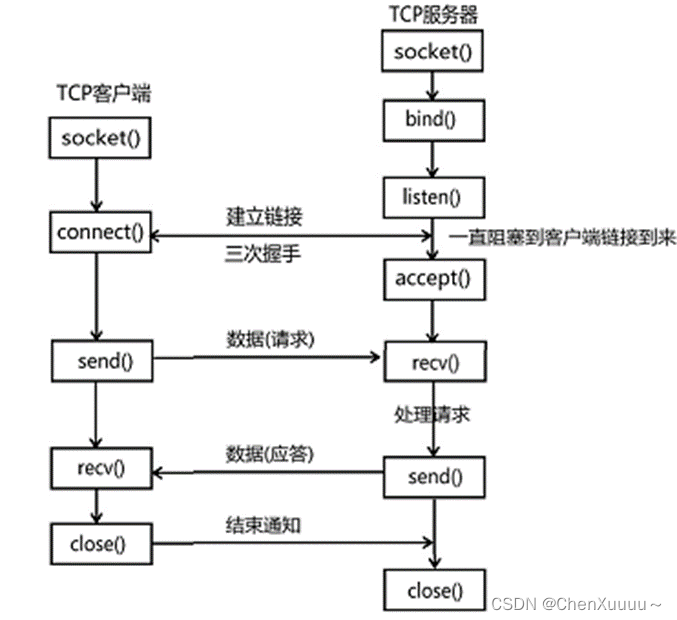
在 IPC 的套接字章节，简要的介绍了网络编程的基本 API 套接字的使用，在这一节中，将会更加详细的介绍套接字的使用。

所需头文件：

#include<arpa/inet.h>  
#include <sys/types.h>  
#include <sys/socket.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| uint32\_t htonl(uint32\_t hostlong); | 将主机字节序转换成网络字节序 | hostlong：要转换为网络字节序的32位整数。 | 返回值是以网络字节序表示的32位整数。 |
| uint16\_t htons(uint16\_t hostshort); | 将主机字节序转换成网络字节序 | hostshort：要转换为网络字节序的16位整数。 | 返回值是以网络字节序表示的16位整数。 |
| uint32\_t ntohl(uint32\_t netlong); | 将网络字节序转换成主机字节序 | netlong：要从网络字节序转换为主机字节序的32位整数。 | 返回值是以主机字节序表示的32位整数。 |
| uint16\_t ntohs(uint16\_t netshort); | 将网络字节序转换成主机字节序 | netshort：要从网络字节序转换为主机字节序的16位整数。 | 返回值是以主机字节序表示的16位整数。 |
| int inet\_pton(int af,const char \*src,woid \*dst); | 用于将IPv4和IPv6地址从文本表示形式（Presentation format）转换为二进制表示形式（Network format）的函数 | af：指定地址族，可以是 AF\_INET 表示IPv4地址，或 AF\_INET6 表示IPv6地址。 src：一个指向以文本形式表示的IP地址的指针 | 如果转换成功，函数返回1（IPv4）或1（IPv6）。 如果输入的地址无效或发生错误，函数返回0。 如果地址族 af 不支持，函数返回-1，并设置 errno 为相应的错误代码。 |
| const char \*inet\_ntop(int af,const void \*src,char \*dst,socklen\_t size); | 用于将二进制表示的IPv4和IPv6地址转换为文本表示形式的函数 | af：指定地址族，可以是 AF\_INET 表示IPv4地址，或 AF\_INET6 表示IPv6地址。 src：一个指向以二进制形式表示的IP地址的指针。 dst：一个指向用于存储转换后的文本形式地址的字符数组的指针。 size：目标字符数组的大小，以确保不会发生缓冲区溢出。 | 如果转换成功，函数返回一个指向目标字符数组 dst 的指针，即返回 dst 的地址。 如果发生错误，函数返回 NULL，并且可以使用 errno 获取错误信息。 |
| int socket(int domain,int type,int protocol); | 用于创建套接字（socket）的系统调用 | domain 参数指定套接字的地址族（Address Family），常见的值包括： AF\_INET：IPv4 地址族 AF\_INET6：IPv6 地址族 AF\_UNIX 或 AF\_LOCAL：本地套接字（Unix 域套接字） type 参数指定套接字的类型（Socket Type），常见的值包括： SOCK\_STREAM：流式套接字，用于面向连接的 TCP 通信 SOCK\_DGRAM：数据报套接字，用于无连接的 UDP 通信 protocol 参数指定套接字的协议，通常为 0，表示根据 domain 和 type 自动选择合适的协议。 | socket 函数返回一个整数套接字描述符（socket descriptor） |
| int bind(int sockfd,const struct sockaddr \*addr,socklen\_t addrlen); | 用于将套接字（socket）与特定的本地地址和端口号绑定的系统调用 | sockfd 参数是套接字描述符，即要进行绑定操作的套接字。 addr 参数是一个指向 struct sockaddr 结构体的指针，其中包含了要绑定的本地地址信息。这个结构体可以是 struct sockaddr\_in（IPv4）或 struct sockaddr\_in6（IPv6），具体取决于套接字的地址族。 addrlen 参数是 addr 结构体的长度。 | 成功：0 失败：-1 |
| int listen(int sockfd,int backlog); | 用于将套接字（socket）标记为一个被动套接字（passive socket），用于监听传入连接请求的系统调用 | sockfd 参数是要进行监听操作的套接字描述符，通常是通过 socket 创建的套接字。 backlog 参数指定了等待连接队列的最大长度，即允许排队等待连接的客户端的数量。通常，这个参数的值是一个正整数，表示队列的最大容量。 | 成功：0 失败：-1 |
| int accept(int sockfd,struct sockaddr \*addr,socklen\_t \*addrlen); | 用于接受传入连接请求并创建新套接字的系统调用 | sockfd 参数是要进行接受连接操作的套接字描述符，通常是通过 socket 创建的套接字，并通过 listen 函数开始监听连接请求。 addr 参数是一个指向 struct sockaddr 结构体的指针，用于存储客户端的地址信息。 addrlen 参数是一个指向 socklen\_t 类型的指针，用于指定 addr 结构体的长度。 | 等待并接受客户端的连接请求，一旦连接请求到达，它会创建一个新的套接字，该套接字与客户端的套接字相连，并返回新套接字的描述符。 |
| int connect(int sockfd,const struct sockaddr \*addr,socklen\_t addrlen); | 用于建立与远程服务器的连接的系统调用 | sockfd 参数是要进行连接操作的套接字描述符，通常是通过 socket 创建的套接字。 addr 参数是一个指向 struct sockaddr 结构体的指针，其中包含了要连接的远程服务器的地址信息。这个结构体可以是 struct sockaddr\_in（IPv4）或 struct sockaddr\_in6（IPv6），具体取决于要连接的服务器的地址族。 addrlen 参数是 addr 结构体的长度。 | 成功：0 失败：-1 |
| ssize\_t read(int fd, void \*buf, size\_t count); | 用于从文件描述符（file descriptor）中读取数据的系统调用 | fd 是要读取数据的文件描述符。可以是文件描述符、套接字描述符或其他文件类型的描述符。 buf 是一个指向要存储读取数据的缓冲区的指针。 count 是要读取的字节数，即希望从文件描述符中读取多少字节的数据。 | 如果成功读取了一些数据（至少一个字节），返回值为已读取的字节数。 如果已经到达文件末尾（EOF），返回值为0。 如果发生错误，返回值为-1，并且可以使用 errno 获取错误代码。 |
| ssize\_t write(int fd, const void \*buf, size\_t count); | 用于向文件描述符（file descriptor）中写入数据的系统调用 | fd 是要写入数据的文件描述符。可以是文件描述符、套接字描述符或其他文件类型的描述符。 buf 是一个指向包含要写入的数据的缓冲区的指针。 count 是要写入的字节数，即希望将多少字节的数据写入文件描述符中。 | 如果成功写入了一些数据（至少一个字节），返回值为已写入的字节数。 如果写入操作被中断（例如，由于信号中断），返回值为-1，并且可以使用 errno 获取错误代码。 |
| ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags); | 用于从套接字（socket）中接收数据的系统调用 | sockfd 参数是要进行接收数据操作的套接字描述符，通常是通过 socket 创建的套接字，用于与远程主机通信。 buf 参数是一个指向用于存储接收数据的缓冲区的指针。 len 参数指定缓冲区的大小，即最多接收多少字节的数据。 flags 参数用于指定接收操作的行为，通常可以设置为0以表示默认行为。 | 如果成功接收了一些数据（至少一个字节），返回值为已接收的字节数。 如果连接已关闭（对方主机关闭了连接），返回值为0，表示已到达数据流的末尾。 如果发生错误，返回值为-1，并且可以使用 perror 或 errno 获取错误代码，以便进一步诊断问题。 |
| ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags); | 用于向套接字（socket）发送数据的系统调用 | sockfd 参数是要进行发送数据操作的套接字描述符，通常是通过 socket 创建的套接字，用于与远程主机通信。 buf 参数是一个指向包含要发送数据的缓冲区的指针。 len 参数指定要发送的字节数，即要发送多少字节的数据。 flags 参数用于指定发送操作的行为，通常可以设置为0以表示默认行为。 | 如果成功发送了一些数据（至少一个字节），返回值为已发送的字节数。 如果连接已关闭（对方主机关闭了连接），返回值为0，表示无法发送更多数据。 如果发生错误，返回值为-1，并且可以使用 perror 或 errno 获取错误代码，以便进一步诊断问题。 |
| int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen); | 用于设置套接字选项的系统调用函数。 | sockfd：要设置选项的套接字文件描述符。 level：选项的级别，通常为 SOL\_SOCKET，表示通用套接字选项。还可以使用其他级别，如 IPPROTO\_TCP 或 IPPROTO\_UDP，以设置与特定协议相关的选项。 optname：要设置的选项的名称，如 SO\_REUSEADDR、SO\_RCVBUF、SO\_SNDBUF 等。 optval：指向存储选项值的缓冲区的指针。 optlen：optval 缓冲区的大小（以字节为单位）。 | 返回0表示成功，返回-1表示失败 |
| int shutdown(int sockfd, int how); | 用于关闭套接字的系统调用函数 | sockfd：要关闭的套接字文件描述符。 how：关闭方式，可以取以下值之一： SHUT\_RD：关闭套接字的读取（接收）端。在调用后，不再能够从套接字接收数据。 SHUT\_WR：关闭套接字的写入（发送）端。在调用后，不再能够向套接字发送数据。 SHUT\_RDWR：同时关闭套接字的读取和写入端。 | 返回0表示成功，返回-1表示失败 |

API 调用顺序：



TCP 连接示例：

// file：tcp\_server.c  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <string.h>  
#include <ctype.h>  
#include <arpa/inet.h>  
#include <netinet/in.h>  
  
int main() {  
 // 创建socket用于和服务端进行通信  
 int lfd = socket(AF\_INET, SOCK\_STREAM, 0);  
 if (lfd < 0) {  
 perror("socket error");  
 return -1;  
 }  
  
 struct sockaddr\_in serv;  
 // 初始化将前n个字节清零  
 bzero(&serv, sizeof(serv));  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 // 标识适用本机任意可用IP  
 serv.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
 int ret = bind(lfd, (struct sockaddr \*)&serv, sizeof(serv));  
 if (ret < 0) {  
 perror("bind error");  
 close(lfd);  
 return -1;  
 }  
 // 最大是128  
 listen(lfd, 128);  
 struct sockaddr\_in client;  
 socklen\_t len = sizeof(client);  
 int cfd = accept(lfd, (struct sockaddr \*)&client, &len);  
  
 // 获取client端的IP和port端口  
 char sIP[16];  
 memset(sIP, 0x00, sizeof(sIP));  
 printf("client-->IP:[%s],PORT:[%d]\n", inet\_ntop(AF\_INET, &client.sin\_addr.s\_addr, sIP, sizeof(sIP)), ntohs(client.sin\_port));  
 printf("lfd==[%d], cfd==[%d]\n", lfd, cfd);  
  
 int n = 0;  
 char buf[1024];  
 while (1) {  
 // 读数据  
 memset(buf, 0x00, sizeof(buf));  
 n = read(cfd, buf, sizeof(buf));  
 if (n <= 0) {  
 if (n == 0) {  
 // 客户端关闭连接  
 printf("Client closed the connection.\n");  
 } else {  
 printf("read error, n==[%d]\n", n);  
 }  
 break;  
 }  
   
 // 处理接收到的数据  
 for (int i = 0; i < n; i++) {  
 if (islower(buf[i])) {  
 buf[i] = toupper(buf[i]);  
 }  
 }  
   
 // 发送数据  
 ssize\_t bytesSent = write(cfd, buf, n);  
 if (bytesSent < 0) {  
 perror("write");  
 break;  
 }  
 }  
  
 // 关闭监听文件描述符和通信文件描述符  
 close(lfd);  
 close(cfd);  
 return 0;  
}

// file: tcp\_client.c  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <string.h>  
#include <arpa/inet.h>  
#include <netinet/in.h>  
  
int main() {  
 // 创建socket用于和服务端进行通信  
 int cfd = socket(AF\_INET, SOCK\_STREAM, 0);  
 if (cfd < 0) {  
 perror("socket error");  
 return -1;  
 }  
 // 连接服务器  
 struct sockaddr\_in serv;  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 inet\_pton(AF\_INET, "127.0.0.1", &serv.sin\_addr);  
 int ret = connect(cfd, (struct sockaddr \*)&serv, sizeof(serv));  
 if (ret < 0) {  
 perror("connect error");  
 close(cfd);  
 return -1;  
 }  
 int n = 0;  
 char buf[256];  
 while (1) {  
 // 读标准输入数据  
 memset(buf, 0x00, sizeof(buf));  
 n = read(STDIN\_FILENO, buf, sizeof(buf));  
 if (n <= 0) {  
 perror("read error");  
 break;  
 }  
 // 发送数据  
 if (write(cfd, buf, n) < 0) {  
 perror("write error");  
 break;  
 }  
 // 读服务端发来的数据  
 memset(buf, 0x00, sizeof(buf));  
 n = read(cfd, buf, sizeof(buf));  
 if (n <= 0) {  
 printf("read error or server closed, n==[%d]\n", n);  
 break;  
 }  
 printf("Received from server: %s\n", buf);  
 }  
 close(cfd);  
 return 0;  
}

# Makefile  
  
all:tcp\_server tcp\_client  
  
server:tcp\_server.c  
 $(CC) tcp\_server.c -o tcp\_server  
  
client:tcp\_client.c  
 $(CC) tcp\_client.c -o tcp\_client  
  
clean:  
 $(RM) tcp\_server tcp\_client

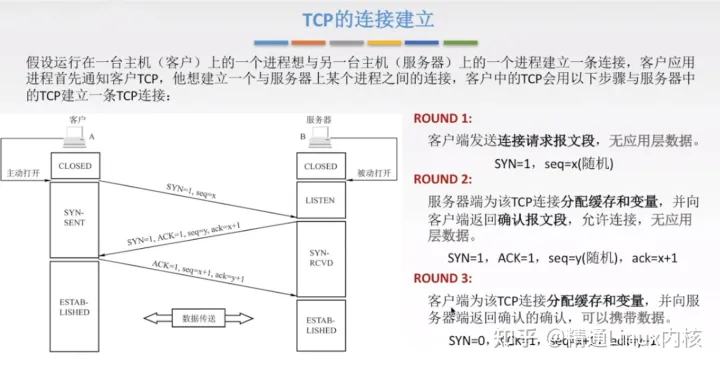
编译运行：

nihao@nihao-z690:~/work/test/socket$ make  
cc tcp\_server.c -o tcp\_server  
cc tcp\_client.c -o tcp\_client  
nihao@nihao-z690:~/work/test/socket$ ./tcp\_server &  
[1] 281222  
nihao@nihao-z690:~/work/test/socket$ ./tcp\_client   
client-->IP:[127.0.0.1],PORT:[33880]  
lfd==[3], cfd==[4]  
this is test!  
Received from server: THIS IS TEST!  
  
^CClient closed the connection.  
  
[1]+ 已完成 ./tcp\_server

在程序中使用 TCP 的 SOCKET API 编程的时候，很多细节都被 SOCKET 框架自动处理了，但是 TCP 的建立过程还是要了解的。

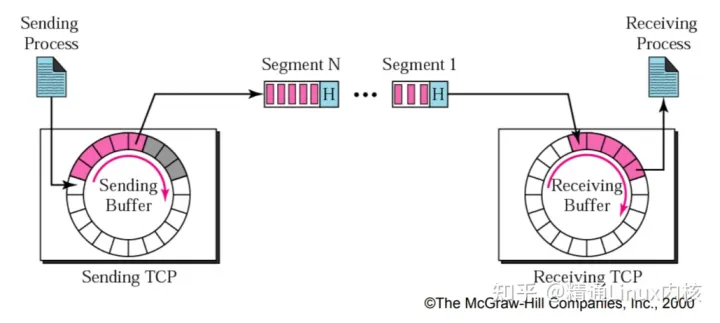
**TCP 通信过程**

TCP 是面向可靠的连接，如果保证连接是可靠的，就需要一些机制来确认状态。

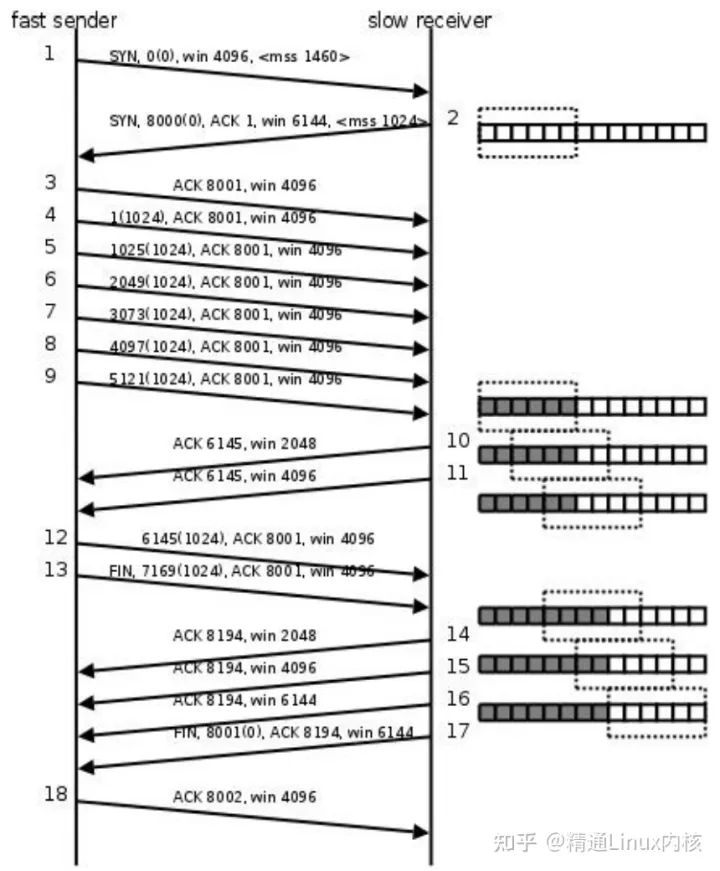


当确认好连接状态后，就可以进行数据发送与接收。

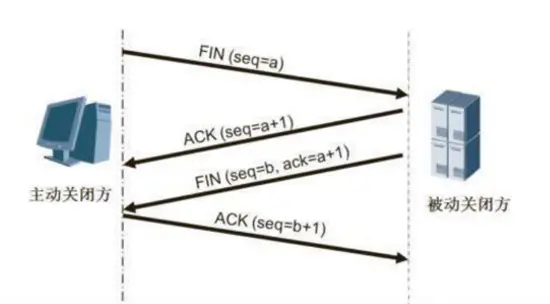
TCP 的数据发送采用的是滑动窗口方式，一种流量控制技术。早期的 TCP 协议中是直接向网络中发送数据的，由于没有流量控制，经常会造成流量阻塞的情况，导致网络环境变差。为了控制流量和网络质量，出现了滑动窗口技术。滑动窗口协议通过交互窗口信息来完成流量控制， 建立连接的双方通过滑动窗口协议知到对方的缓冲区大小，同时通过 ACK 确认机制了解网络的通信质量。而滑动窗口能很好的确认数据包的丢失问题，从而进行重发，保证传输的数据连续和正确。



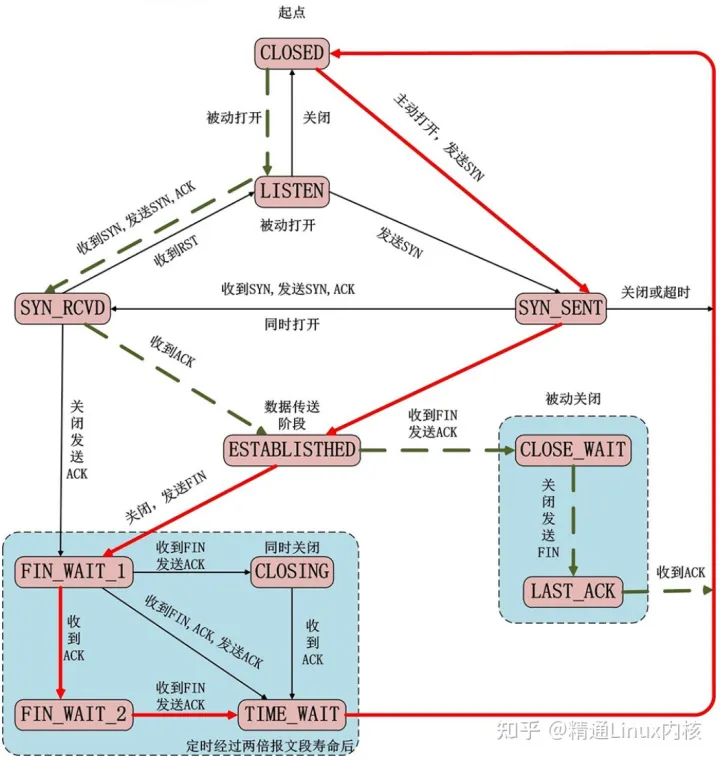
数据通信状态：



有状态的 TCP 端开时，需要进行四次挥手。



编程调用过程：



UDP 连接示例：

// file udp\_server.c  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <netinet/in.h>  
#include <ctype.h>  
  
int main() {  
 // 创建socket用于和客户端进行通信  
 int sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);  
 if (sockfd < 0) {  
 perror("socket error");  
 return -1;  
 }  
  
 struct sockaddr\_in serv;  
 bzero(&serv, sizeof(serv));  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 serv.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
  
 int ret = bind(sockfd, (struct sockaddr \*)&serv, sizeof(serv));  
 if (ret < 0) {  
 perror("bind error");  
 close(sockfd);  
 return -1;  
 }  
  
 struct sockaddr\_in client;  
 socklen\_t len = sizeof(client);  
  
 char buf[1024];  
 while (1) {  
 // 从客户端接收数据  
 memset(buf, 0x00, sizeof(buf));  
 ssize\_t bytesReceived = recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr \*)&client, &len);  
 if (bytesReceived < 0) {  
 perror("recvfrom error");  
 break;  
 }  
  
 // 获取客户端的IP和端口  
 char clientIP[INET\_ADDRSTRLEN];  
 inet\_ntop(AF\_INET, &client.sin\_addr, clientIP, sizeof(clientIP));  
 int clientPort = ntohs(client.sin\_port);  
 printf("Received data from %s:%d\n", clientIP, clientPort);  
  
 // 处理接收到的数据  
 for (ssize\_t i = 0; i < bytesReceived; i++) {  
 buf[i] = toupper(buf[i]);  
 }  
  
 // 发送处理后的数据回客户端  
 ssize\_t bytesSent = sendto(sockfd, buf, bytesReceived, 0, (struct sockaddr \*)&client, len);  
 if (bytesSent < 0) {  
 perror("sendto error");  
 break;  
 }  
 }  
  
 // 关闭套接字  
 close(sockfd);  
 return 0;  
}

// file: udp\_client.c  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <netinet/in.h>  
  
int main() {  
 // 创建socket用于和服务器进行通信  
 int sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);  
 if (sockfd < 0) {  
 perror("socket error");  
 return -1;  
 }  
  
 struct sockaddr\_in serv;  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 inet\_pton(AF\_INET, "127.0.0.1", &serv.sin\_addr);  
  
 int n = 0;  
 char buf[256];  
 while (1) {  
 // 读标准输入数据  
 memset(buf, 0x00, sizeof(buf));  
 n = read(STDIN\_FILENO, buf, sizeof(buf));  
 if (n <= 0) {  
 perror("read error");  
 break;  
 }  
  
 // 发送数据到服务器  
 ssize\_t bytesSent = sendto(sockfd, buf, n, 0, (struct sockaddr \*)&serv, sizeof(serv));  
 if (bytesSent < 0) {  
 perror("sendto error");  
 break;  
 }  
  
 // 接收服务器的响应数据  
 memset(buf, 0x00, sizeof(buf));  
 socklen\_t serv\_len = sizeof(serv);  
 ssize\_t bytesReceived = recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr \*)&serv, &serv\_len);  
 if (bytesReceived < 0) {  
 perror("recvfrom error");  
 break;  
 }  
 printf("Received from server: %s\n", buf);  
 }  
  
 // 关闭套接字  
 close(sockfd);  
 return 0;  
}

# Makefile  
  
all:udp\_server udp\_client  
  
server:udp\_server.c  
 $(CC) udp\_server.c -o udp\_server  
  
client:udp\_client.c  
 $(CC) udp\_client.c -o udp\_client  
  
clean:  
 $(RM) udp\_server udp\_client

编译运行:

nihao@nihao-z690:~/work/test/socket$ make  
cc udp\_server.c -o udp\_server  
cc udp\_client.c -o udp\_client  
nihao@nihao-z690:~/work/test/socket$ ./udp\_server &  
[1] 671929  
nihao@nihao-z690:~/work/test/socket$ ./udp\_client   
nihao  
Received data from 127.0.0.1:37246  
Received from server: NIHAO  
  
this is test  
Received data from 127.0.0.1:37246  
Received from server: THIS IS TEST  
  
^C

#### IO 复用

网络编程分为服务端和连接端两个部分，基本的网络通信比较简单，但是稍微复杂点的就需要用到一些 IO 非阻塞技术。

在读取 IO 描述符的时候，通常会阻塞线程，这时如果想要再做其他的事情可以有两种方法。第一、使用多线程或者多进程技术，另开一个执行流执行任务。第二、使用 IO select 方式，检测 IO 是否可读可写。

##### 多线程 TCP 服务端

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <netinet/in.h>  
#include <pthread.h>  
#include <signal.h>  
  
// 全局变量用于通知线程退出  
volatile sig\_atomic\_t exit\_flag = 0;  
  
// 互斥锁用于线程间同步  
pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  
  
// Ctrl+C信号处理函数  
void sigint\_handler(int signum) {  
 printf("\nReceived Ctrl+C signal. Exiting...\n");  
 exit\_flag = 1;  
}  
  
// 线程处理函数  
void \*client\_handler(void \*arg) {  
 int client\_socket = \*(int \*)arg;  
 char buffer[1024];  
 ssize\_t bytes\_received;  
  
 while (1) {  
 pthread\_mutex\_lock(&mutex);  
  
 if (exit\_flag) {  
 // 如果收到退出信号，退出线程  
 pthread\_mutex\_unlock(&mutex);  
 break;  
 }  
  
 pthread\_mutex\_unlock(&mutex);  
  
 // 读取客户端发送的数据  
 bytes\_received = recv(client\_socket, buffer, sizeof(buffer), 0);  
 if (bytes\_received <= 0) {  
 // 处理客户端断开连接或错误  
 break;  
 }  
  
 // 处理接收到的数据  
 // 发送处理后的数据回客户端  
 send(client\_socket, buffer, bytes\_received, 0);  
 }  
  
 // 关闭客户端套接字  
 close(client\_socket);  
 free(arg);  
 return NULL;  
}  
  
int main() {  
 int server\_socket, client\_socket;  
 struct sockaddr\_in server\_addr, client\_addr;  
 socklen\_t client\_addr\_len = sizeof(client\_addr);  
 pthread\_t thread\_id;  
  
 // 注册Ctrl+C信号处理函数  
 signal(SIGINT, sigint\_handler);  
  
 // 创建套接字  
 server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);  
 if (server\_socket < 0) {  
 perror("socket error");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 绑定套接字到地址和端口  
 memset(&server\_addr, 0, sizeof(server\_addr));  
 server\_addr.sin\_family = AF\_INET;  
 server\_addr.sin\_addr.s\_addr = INADDR\_ANY;  
 server\_addr.sin\_port = htons(8888);  
  
 if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0) {  
 perror("bind error");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 监听连接请求  
 if (listen(server\_socket, 5) < 0) {  
 perror("listen error");  
 exit(EXIT\_FAILURE);  
 }  
  
 printf("Server listening on port 8888...\n");  
  
 while (!exit\_flag) {  
 // 接受客户端连接  
 client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_addr\_len);  
 if (client\_socket < 0) {  
 if (exit\_flag) {  
 // 如果接收到退出信号，退出循环  
 break;  
 }  
 perror("accept error");  
 continue;  
 }  
  
 printf("Accepted connection from %s:%d\n", inet\_ntoa(client\_addr.sin\_addr), ntohs(client\_addr.sin\_port));  
  
 // 创建线程处理客户端通信  
 int \*client\_socket\_ptr = (int \*)malloc(sizeof(int));  
 if (client\_socket\_ptr == NULL) {  
 perror("malloc error");  
 close(client\_socket);  
 continue;  
 }  
 \*client\_socket\_ptr = client\_socket;  
  
 if (pthread\_create(&thread\_id, NULL, client\_handler, client\_socket\_ptr) != 0) {  
 perror("pthread\_create error");  
 close(client\_socket);  
 free(client\_socket\_ptr);  
 }  
 }  
  
 // 关闭服务器套接字  
 close(server\_socket);  
  
 // 等待所有线程退出  
 pthread\_mutex\_destroy(&mutex);  
  
 return 0;  
}

多线程解决了 IO 阻塞的问题，但是当连接数过多的时候，将会创建过多的线程，给系统造成的压力是指数级增加。

##### 高性能 IO-select

select是一种I/O多路复用机制，用于在多个文件描述符（sockets、文件、管道等）上等待事件并进行处理。它允许应用程序同时监视多个文件描述符的可读、可写、异常等事件，而不需要使用阻塞或轮询方式。select通常用于编写网络服务器和其他需要处理多个I/O事件的程序。

select允许应用程序注册多个文件描述符，并等待这些文件描述符上的事件发生。当一个或多个事件发生时，select会通知应用程序，并提供哪些文件描述符上发生了事件。select是跨平台的，它几乎在所有主流操作系统上可用，包括Linux、Windows、UNIX等。select可以用于阻塞或非阻塞模式。在阻塞模式下，select会一直等待事件发生，而在非阻塞模式下，它会立即返回，并告诉应用程序哪些文件描述符上发生了事件。与一些其他I/O多路复用机制相比，select在处理大量文件描述符时可能会有性能问题，因为它需要遍历所有注册的文件描述符来检查事件是否发生。这可能导致性能下降，特别是在需要监视大量文件描述符的情况下。在使用select时，您需要创建一个fd\_set数据结构，用于表示要监视的文件描述符集合，以及该文件描述符上关注的事件。select系统调用会修改fd\_set来指示哪些文件描述符上发生了事件。select的一个限制是它通常有一个最大文件描述符数的限制，这个限制通常是在头文件中定义的。如果要监视的文件描述符超过了这个限制，需要特殊处理。select是一种跨平台的I/O多路复用机制，适用于需要监视和处理多个文件描述符的应用程序。虽然它在性能上可能不如一些其他机制，如epoll（在Linux上），但它仍然是一种有用的工具，特别是在需要跨平台兼容性的情况下。

所需头文件：

#include <sys/select.h>

函数和宏介绍：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| int select(int nfds,fd\_set \*readfds,fd\_set \*writefds,fd\_set \*exceptfds,struct timeval \*timeout); | 委托内核监控该文件描述符对应的读,写或者错误事件的发生 | nfds：监视的最大文件描述符加1。 readfds：指向一个可读文件描述符集合的指针，用于监视是否有可读数据。 writefds：指向一个可写文件描述符集合的指针，用于监视是否可以进行写入操作。 exceptfds：指向一个异常文件描述符集合的指针，用于监视是否有异常情况发生。 timeout：指向一个 struct timeval 结构的指针，用于设置超时时间。如果传递 NULL，select 将一直阻塞等待事件发生。 | select 返回一个正整数，表示就绪的文件描述符的数量，或者返回0（表示超时）或-1（表示发生错误） |
| void FD\_CLR(int fd,fd\_set \*set); | 将fd从set集合中清除的宏 | fd：IO 描述符。set：IO 描述符集合 | 无 |
| int FD\_ISSET(int fd,fd\_set \*set); | 判断fd是否在集合中的宏 | fd：IO 描述符。set：IO 描述符集合 | 如果fd在set集合中,返回1,否则返回0 |
| void FD\_SET(int fd,fd\_set \*set); | 将fd设置到set集合中的宏 | fd：IO 描述符。set：IO 描述符集合 | 无 |
| void FD\_ZERO(fd\_set \*set); | 初始化set集合的宏 | set：IO 描述符集合 | 为 |

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <sys/types.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <netinet/in.h>  
#include <ctype.h>  
#include "wrap.h"  
  
int main() {  
 // 创建socket  
 int lfd = Socket(AF\_INET, SOCK\_STREAM, 0);  
  
 // 设置端口复用  
 int opt = 1;  
 setsockopt(lfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(int));  
  
 // 绑定  
 struct sockaddr\_in serv;  
 bzero(&serv, sizeof(serv));  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 serv.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
 Bind(lfd, (struct sockaddr \*)&serv, sizeof(serv));  
  
 // 监听  
 Listen(lfd, 128);  
  
 // 定义fd\_set类型的变量  
 fd\_set readfds;  
 fd\_set tmpfds; // readfds是传入传出参数，所以需要临时变量来存储  
  
 // 清空readfds和tmpfds集合  
 FD\_ZERO(&readfds);  
 FD\_ZERO(&tmpfds);  
  
 // 将lfd加入到readfds中，委托内核监控  
 FD\_SET(lfd, &readfds);  
  
 int maxfd = lfd;  
 int nready;  
 int cfd;  
 int i = 0;  
 int sockfd;  
 char buf[1024];  
 int n = 0;  
 while (1) {  
 tmpfds = readfds;  
  
 // tmpfds是输入输出参数  
 // 输入:告诉内核要监测哪些文件描述符  
 // 输出:内核告诉应用程序有哪些文件描述符发生了变化  
 nready = select(maxfd + 1, &tmpfds, NULL, NULL, NULL); // select返回发生变化的文件描述符个数  
 if (nready < 0) {  
 // 被信号中断  
 if (errno == EINTR) {  
 continue;  
 }  
 perror("select error");  
 break;  
 }  
  
 // 有客户端连接请求到来  
 if (FD\_ISSET(lfd, &tmpfds)) {  
 // 接受新的客户端连接请求  
 cfd = Accept(lfd, NULL, NULL);  
 if (cfd < 0) {  
 perror("accept error");  
 continue;  
 }  
 printf("Accepted connection from %s:%d\n", GetPeerName(cfd), GetPeerPort(cfd));  
  
 // 将cfd加入到readfds集合中  
 FD\_SET(cfd, &readfds);  
  
 // 修改内核的监控范围  
 if (maxfd < cfd) {  
 maxfd = cfd;  
 }  
  
 if (--nready == 0) {  
 continue;  
 }  
 }  
  
 // 有数据发来的情况  
 for (i = lfd + 1; i <= maxfd; i++) {  
 sockfd = i;  
 // 判断sockfds文件描述符是否发生变化  
 if (FD\_ISSET(sockfd, &tmpfds)) {  
 // 读数据  
 memset(buf, 0x00, sizeof(buf));  
 n = Read(sockfd, buf, sizeof(buf));  
 if (n <= 0) {  
 // 关闭连接  
 Close(sockfd);  
 // 将sockfd从readfds中删除  
 FD\_CLR(sockfd, &readfds);  
 } else {  
 printf("n==[%d], buf==[%s]\n", n, buf);  
 int k = 0;  
 for (k = 0; k < n; k++) {  
 buf[k] = toupper(buf[k]);  
 }  
 Write(sockfd, buf, n);  
 }  
 if (--nready == 0) {  
 break;  
 }  
 }  
 }  
 }  
 // 关闭监听套接字  
 Close(lfd);  
 return 0;  
}

通过 IO 监控调用，可以在一个线程内处理多个网络连接的任务。

##### 高性能 IO-poll

poll是一种I/O多路复用机制，用于在多个文件描述符（sockets、文件、管道等）上等待事件并进行处理。它允许应用程序监视多个文件描述符的可读、可写、异常等事件，而不需要使用阻塞或轮询方式。poll通常用于编写网络服务器和其他需要处理多个I/O事件的程序。

poll允许应用程序注册多个文件描述符，并等待这些文件描述符上的事件发生。当一个或多个事件发生时，poll会通知应用程序，并提供哪些文件描述符上发生了事件。poll是POSIX标准的一部分，因此它在各种UNIX-like操作系统上可用，而不仅仅限于Linux。poll可以用于阻塞或非阻塞模式。在阻塞模式下，poll会一直等待事件发生，而在非阻塞模式下，它会立即返回，并告诉应用程序哪些文件描述符上发生了事件。与某些其他I/O多路复用机制相比，poll在处理大量文件描述符时可能会有一些性能开销，因为它需要遍历所有注册的文件描述符来检查事件是否发生。poll通常用于单线程处理多个连接的情况，因为它允许一个线程监视和处理多个文件描述符上的事件。在使用poll时，您需要创建一个struct pollfd类型的数组，每个数组元素对应一个要监视的文件描述符，以及该文件描述符上关注的事件和发生的事件。poll系统调用会修改数组中的元素来指示哪些文件描述符上发生了事件。poll是一种可用于多平台的I/O多路复用机制，适用于需要监视和处理多个文件描述符的应用程序。虽然它的性能可能不如一些其他机制，如epoll（在Linux上），但它仍然是一种有用的工具，特别是在需要可移植性的情况下。

所需头文件

#include <poll.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| int poll(struct pollfd \*fds,nfds\_t nfds,int timeout); | 监控多路IO | fds：一个指向 struct pollfd 结构数组的指针，每个元素描述了一个文件描述符和你想要监视的事件。 nfds：数组中元素的数量。 timeout：指定超时时间（以毫秒为单位），如果设置为 -1，则表示一直阻塞等待事件发生；如果设置为 0，则表示立即返回，不阻塞；如果设置为正数，表示等待指定的毫秒数后返回。 | -1：发生错误，可以通过检查 errno 获取错误信息。 0：超时，没有文件描述符就绪。 大于 0：就绪的文件描述符数量。 |

使用示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <sys/poll.h>  
#include <errno.h>  
  
#define MAX\_CLIENTS 1024  
#define MAX\_BUFFER\_SIZE 1024  
  
int main() {  
 // 创建 socket  
 int lfd = socket(AF\_INET, SOCK\_STREAM, 0);  
 if (lfd < 0) {  
 perror("Socket creation failed");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 允许端口复用  
 int opt = 1;  
 setsockopt(lfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(int));  
  
 // 绑定 bind  
 struct sockaddr\_in serv;  
 memset(&serv, 0, sizeof(serv));  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 serv.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
  
 if (bind(lfd, (struct sockaddr \*)&serv, sizeof(serv)) < 0) {  
 perror("Bind failed");  
 close(lfd);  
 exit(EXIT\_FAILURE);  
 }  
  
 // 监听 listen  
 if (listen(lfd, 128) < 0) {  
 perror("Listen failed");  
 close(lfd);  
 exit(EXIT\_FAILURE);  
 }  
  
 struct pollfd client[MAX\_CLIENTS];  
  
 // 监听文件描述符委托给内核监控，监控读事件  
 client[0].fd = lfd;  
 client[0].events = POLLIN;  
  
 int i = 0;  
 int nready = 0;  
 int maxi = 0; // 内核监控范围  
  
 for (i = 1; i < MAX\_CLIENTS; i++) {  
 client[i].fd = -1;  
 }  
  
 int n = 0;  
 int cfd = 0;  
 int sockfd = 0;  
 char buf[MAX\_BUFFER\_SIZE];  
 while (1) {  
 nready = poll(client, maxi + 1, -1);  
 if (nready < 0) {  
 if (errno == EINTR) {  
 continue;  
 }  
 perror("Poll failed");  
 break;  
 }  
  
 // 有客户端连接请求  
 if (client[0].revents == POLLIN) {  
 cfd = accept(lfd, NULL, NULL);  
 // 寻找 client 数组中的可用位置  
 for (i = 1; i < MAX\_CLIENTS; i++) {  
 if (client[i].fd == -1) {  
 client[i].fd = cfd;  
 client[i].events = POLLIN;  
 break;  
 }  
 }  
 // 没有可用位置，则关闭连接  
 if (i == MAX\_CLIENTS) {  
 close(cfd);  
 continue;  
 }  
 if (maxi < i) {  
 maxi = i;  
 }  
 if (--nready == 0) {  
 continue;  
 }  
 }  
  
 // 有数据到来的情况  
 for (i = 1; i <= maxi; i++) {  
 sockfd = client[i].fd;  
 memset(buf, 0x00, sizeof(buf));  
 // 若 fd 为 -1，表示连接已经关闭或者没有连接  
 if (client[i].fd == -1) {  
 continue;  
 }  
 if (client[i].revents == POLLIN) {  
 n = read(sockfd, buf, sizeof(buf));  
 if (n <= 0) {  
 if (n == 0) {  
 printf("Client closed the connection\n");  
 } else {  
 perror("Read error or client close");  
 }  
 close(sockfd);  
 client[i].fd = -1; // fd = -1，表示不再让内核监控  
 } else {  
 printf("Received %d bytes from client: %s\n", n, buf);  
 write(sockfd, buf, n);  
 }  
 if (--nready == 0) {  
 break;  
 }  
 }  
 }  
 }  
  
 close(lfd);  
 return 0;  
}

##### 多路IO-epoll

epoll（事件轮询）是Linux操作系统提供的一种高效的I/O事件通知机制，用于处理大量的文件描述符（sockets、文件、管道等）上的事件。它通常用于编写高性能的网络服务器，以监视多个套接字上的事件，如数据可读或可写等，而无需使用阻塞或轮询方式。

epoll在处理大量文件描述符时非常高效，因为它使用了内核中的数据结构来迅速检测和通知发生的事件。使用epoll，您可以将关注的事件注册到一个epoll对象中，然后等待这些事件的发生。当一个或多个事件发生时，epoll会通知应用程序。epoll支持两种不同的工作模式，水平触发（Level Triggered）和边缘触发（Edge Triggered）。水平触发模式会在文件描述符上有待处理事件时反复通知，而边缘触发模式只会在事件状态改变时通知一次，要求应用程序自行处理未读数据。epoll通常用于处理大量连接的网络服务器，允许单个线程有效地处理多个连接，而无需使用多线程或多进程。epoll通常与非阻塞I/O一起使用，这样在一个线程中可以同时处理多个套接字，而不会阻塞整个应用程序。在Linux中，epoll提供了三个主要的系统调用函数来进行操作：epoll\_create（创建epoll对象）、epoll\_ctl（注册、修改或删除事件），以及epoll\_wait（等待事件发生并返回发生的事件列表）。

epoll是Linux下高性能I/O多路复用机制之一，适用于需要处理大量并发连接的网络应用程序，如Web服务器、代理服务器、聊天应用等。通过充分利用操作系统内核的特性，它能够提供出色的性能和可伸缩性。

头文件:

#include <sys/epoll.h>

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | 操作 | 函数参数 | 返回值 |
| int epoll\_create(int size); | 创建一个树根 | size:最大节点数 | 成功:返回一个大于0的文件描述符,代表整个树的树根 失败:返回-1,并设置errno值 |
| int epoll\_ctl(int epfd,int op,int fd,struct epoll\_event \*event); | 将要监听的节点在epoll树上添加,删除和修改 | epfd:epoll树根 op: EPOLL\_CTL\_ADD:添加事件节点到树上 EOPLL\_CTL\_DEL:从树上删除事件节点 EOPLL\_CTL\_MOD:修改树上对应的事件节点 fd:事件节点对应的文件描述符 event:要操作的事件节点 | 返回0表示成功，返回-1表示失败 |
| int epoll\_wait(int pefd,struct epoll\_event \*events,int maxevents,int timeout); | 等待内核返回事件发生 | epfd:epoll树根 events:传出参数,时间结构体数组 maxevents:数组大小 timeout: -1:表示永久阻塞 0:立即返回 >0:表示超时等待事件 | 成功:返回发生事件的个数 失败:若timeout=0,没有事件发生,则返回;返回-1,设置errno值 |

示例：

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <sys/epoll.h>  
#include <errno.h>  
  
#define MAX\_EVENTS 1024  
#define MAX\_BUFFER\_SIZE 1024  
  
int main() {  
 int lfd = socket(AF\_INET, SOCK\_STREAM, 0);  
 if (lfd < 0) {  
 perror("Socket creation failed");  
 exit(EXIT\_FAILURE);  
 }  
  
 int opt = 1;  
 setsockopt(lfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(int));  
  
 struct sockaddr\_in serv;  
 memset(&serv, 0, sizeof(serv));  
 serv.sin\_family = AF\_INET;  
 serv.sin\_port = htons(8888);  
 serv.sin\_addr.s\_addr = htonl(INADDR\_ANY);  
  
 if (bind(lfd, (struct sockaddr \*)&serv, sizeof(serv)) < 0) {  
 perror("Bind failed");  
 close(lfd);  
 exit(EXIT\_FAILURE);  
 }  
  
 if (listen(lfd, 128) < 0) {  
 perror("Listen failed");  
 close(lfd);  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建 epoll 树  
 int epfd = epoll\_create(1024);  
 if (epfd < 0) {  
 perror("Epoll creation failed");  
 close(lfd);  
 exit(EXIT\_FAILURE);  
 }  
  
 // 将 lfd 添加到 epoll 树上  
 struct epoll\_event ev;  
 ev.events = EPOLLIN;  
 ev.data.fd = lfd;  
 if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, lfd, &ev) < 0) {  
 perror("Epoll\_ctl add failed");  
 close(lfd);  
 close(epfd);  
 exit(EXIT\_FAILURE);  
 }  
  
 struct epoll\_event events[MAX\_EVENTS]; // 定义一个传出数组  
 int nready = 0;  
 int i = 0;  
 int n = 0;  
 char buf[MAX\_BUFFER\_SIZE];  
 int cfd = 0;  
 int sockfd = 0;  
 while (1) {  
 // 阻塞，内核等待事件发生，保存修改或者发生事件的文件描述符  
 nready = epoll\_wait(epfd, events, MAX\_EVENTS, -1);  
 if (nready < 0) {  
 if (errno == EINTR) {  
 continue;  
 }  
 perror("Epoll\_wait failed");  
 break;  
 }  
  
 // 有事件发生时，处理客户端连接请求或数据传输  
 for (i = 0; i < nready; i++) {  
 sockfd = events[i].data.fd;  
 // 判断是否是 lfd，是则 accept  
 if (sockfd == lfd) {  
 cfd = accept(lfd, NULL, NULL);  
 if (cfd < 0) {  
 perror("Accept failed");  
 continue;  
 }  
 ev.data.fd = cfd;  
 ev.events = EPOLLIN;  
 // 将 cfd 添加到 epoll 树上  
 if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, cfd, &ev) < 0) {  
 perror("Epoll\_ctl add failed for cfd");  
 close(cfd);  
 }  
 continue;  
 }  
  
 // 有客户端发送数据过来  
 memset(buf, 0x00, sizeof(buf));  
 n = read(sockfd, buf, sizeof(buf));  
 if (n <= 0) {  
 if (n == 0) {  
 printf("Client closed the connection\n");  
 } else {  
 perror("Read error or client close");  
 }  
 // 关闭通信文件描述符，从树上删除  
 close(sockfd);  
 epoll\_ctl(epfd, EPOLL\_CTL\_DEL, sockfd, NULL);  
 continue;  
 }  
 printf("Received %d bytes from client: %s\n", n, buf);  
  
 // 假设服务器直接将收到的数据回发给客户端  
 write(sockfd, buf, n);  
 }  
 }  
  
 close(epfd);  
 close(lfd);  
 return 0;  
}

### 应用协议

* HTTP/HTTPS

http/https 又被叫做超文本协议，是我们网络上最常用的通信协议了，通过这个协议，我们拥有了现在丰富多彩的互联网生活，上网冲浪看到的网页、音频、视频等都是通过这个协议进行传输的，所以是非常重要的一个协议。

HTTP 最开始被用于在互联网上传输超文本 HTML，但经过扩展，很快便能传输音频、视频、各类文件中。后来由于安全性问题，经过扩展后，拥有了加密功能，新的协议被命名成了 HTTPS。

* FTP

FTP（File Transfer Protocol）是一种用于在计算机之间传输文件的标准网络协议。它属于应用层协议，通常用于将文件从一个计算机上传输到另一个计算机，或者从服务器下载文件到客户端。  
FTP 使用客户端-服务器模型，其中有一个 FTP 服务器和一个或多个 FTP 客户端。客户端通过 FTP 协议与服务器建立连接，然后通过命令进行文件操作。常见的 FTP 客户端包括 FileZilla 、WinSCP 、以及内置 FTP 客户端。用户可以使用 FTP 客户端向服务器上传、下载、删除、重命名和列出文件。 FTP 使用两个不同的连接：控制连接和数据连接。控制连接用于传输 FTP 命令和响应。这个连接始终保持打开状态，直到用户结束 FTP 会话。数据连接用于实际的文件传输。每当需要传输文件或目录列表时，客户端会打开一个新的数据连接，并在传输完成后关闭它。FTP 支持多种传输模式，包括 ASCII 模式和二进制模式。ASCII 模式用于传输文本文件，而二进制模式用于传输非文本文件，如图像、音频和视频。除了这两种模式，还有自动模式，它根据文件类型自动选择适当的传输模式。FTP 服务器通常要求用户提供用户名和密码以进行身份验证。这种身份验证方式被称为基本身份验证。另外，FTP 还支持匿名 FTP，允许用户以匿名方式登录到服务器，通常使用 "anonymous" 作为用户名，以便访问公共资源。FTP 是一种不安全的协议，因为它在传输过程中不对数据进行加密。这意味着敏感数据可能会在传输过程中被窃听。为了增加安全性，可以使用 FTPS（FTP Secure）或 SFTP（SSH File Transfer Protocol），这两者都提供了加密传输和身份验证机制。

* SMTP

SMTP（Simple Mail Transfer Protocol）是一种用于电子邮件传输的协议。它是应用层协议，用于将电子邮件从发送者的电子邮件客户端发送到接收者的电子邮件服务器，然后再从接收者的服务器传递到接收者的电子邮件客户端。

SMTP 是一种客户端-服务器协议，它依赖于可信任的 SMTP 服务器来中转和传递电子邮件。发送者使用 SMTP 客户端程序或电子邮件客户端（如 Outlook、Gmail 等）来连接到自己的 SMTP 服务器，并通过 SMTP 协议发送电子邮件。SMTP 服务器接受发送者的电子邮件，并将其传递到接收者的 SMTP 服务器，该服务器负责将电子邮件投递给接收者。SMTP 协议用于传输电子邮件的消息部分，包括邮件文本、附件和邮件头。SMTP 服务器使用消息头中的收件人地址来确定将电子邮件传递到哪个接收者的电子邮件服务器。SMTP 协议使用 TCP 协议进行通信，默认端口号是 25 。然而，为了提供更加安全的传输，SMTP 服务器也可以监听其他端口，如 587（SMTPS）。SMTP 服务器通常要求用户提供身份验证信息，以确认发送者是授权的用户。身份验证可以使用用户名和密码，也可以使用其他方法，如OAuth2.0。SMTP 服务器还可以采用开放式 SMTP（Open Relay SMTP）或者匿名 SMTP ，但这会增加垃圾邮件风险，因此通常不被推荐。SMTP 传输通常是明文的，因此电子邮件内容可以在传输过程中被窃听。为了增加安全性，可以使用 SMTP over TLS/SSL（SMTPS）来加密 SMTP 会话。当 SMTP 客户端与 SMTP 服务器之间建立 TLS/SSL 连接时，通信将得到加密，保护电子邮件内容的隐私

* DNS

DNS（Domain Name System）是互联网上的一种分布式数据库系统，用于将域名转换为 IP 地址以及反向转换，即将 IP 地址转换为域名。DNS是互联网的基础设施之一，负责将用户友好的域名映射到计算机能够识别的 IP 地址，从而实现网络中的通信和服务访问。

DNS的主要功能之一是将用户提供的域名（例如<www.example.com>）转换为对应的 IP 地址（例如192.0.2.1）。这个过程称为域名解析。域名解析使得用户可以使用易记的域名来访问互联网上的资源，而不需要记住复杂的 IP 地址。DNS采用分层结构，由多个DNS服务器组成。最顶层是根域（Root Domain），下面是顶级域（Top-Level Domains，TLDs），例如.com、.org、.net等。然后是二级域（Second-Level Domains），最终是子域名（Subdomains）。每个域都可以拥有自己的DNS服务器，负责管理该域下的域名解析。当用户在浏览器中输入一个域名时，操作系统的DNS客户端会发起域名解析请求。首先，DNS客户端会查询本地缓存中是否有域名对应的IP地址，如果有，则直接返回结果。如果本地缓存没有，DNS客户端会向本地DNS服务器（通常由ISP提供）发起查询请求。如果本地DNS服务器没有缓存结果，它将向更高级别的DNS服务器发起递归查询，直到根域DNS服务器。根域DNS服务器返回顶级域DNS服务器的地址。最终，递归查询会在顶级域、二级域和子域的DNS服务器之间进行，直到找到域名对应的IP地址。这个结果会被返回给本地DNS服务器，然后传递给DNS客户端，最终返回给用户。DNS服务器会缓存域名解析结果，以减少重复的域名解析请求。这有助于提高DNS查询的速度和减轻DNS服务器的负载。除了将域名解析为IP地址，DNS还支持反向解析，即将IP地址解析为域名。这通常用于确定某个IP地址的所有者

* DHCP

DHCP（Dynamic Host Configuration Protocol）是一种用于在计算机网络上分配IP地址和其他网络配置参数的协议。DHCP允许网络管理员自动分配和管理IP地址，从而简化了网络设备的配置和维护工作。

DHCP服务器负责为网络上的客户端设备分配IP地址。当设备连接到网络时，它可以向DHCP服务器发送请求以获取可用的IP地址。DHCP服务器通常维护一个IP地址池，从中分配IP地址给客户端设备。每个IP地址都有一个租期（lease），在租期到期前，客户端设备可以继续使用该IP地址。除了IP地址，DHCP还可以分配其他网络配置参数，如子网掩码、默认网关、DNS服务器的IP地址、NTP服务器的IP地址等。这些配置参数帮助客户端设备正确配置其网络连接，以便与其他设备通信和访问互联网。DHCP允许网络管理员自动配置和管理网络上的设备，而无需手动分配IP地址和配置参数。这对于大型网络尤其有用，因为手动配置每个设备将非常耗时和容易出错。客户端设备在租期结束前可以向DHCP服务器请求续约，以保持其分配的IP地址。如果设备不再需要IP地址，它可以向DHCP服务器发送释放请求，将IP地址返回到IP地址池中，以供其他设备使用。DHCP通信通常使用广播方式进行。当客户端设备首次连接到网络时，它会广播DHCP发现请求，以查找可用的DHCP服务器。DHCP服务器会广播DHCP提供请求，以响应客户端设备的请求。为了提高安全性，网络管理员可以配置DHCP服务器以仅响应授权的客户端设备。此外，可以使用DHCP Snooping等技术来检测和防止恶意DHCP服务器的出现。

* SNMP

SNMP（Simple Network Management Protocol）是一种用于监控和管理网络设备、服务器和应用程序的网络协议。它允许网络管理员远程监视网络设备的性能、状态和配置，以及执行远程管理任务。  
SNMP 遵循客户端-服务器或管理-代理模型。网络管理员充当管理者，而网络设备、服务器和应用程序充当代理。管理者使用 SNMP 协议从代理获取信息，以监视和管理网络设备。SNMP 使用 MIB 数据库来存储有关网络设备和应用程序的信息。MIB 包含了大量的对象，这些对象描述了网络设备的状态、配置和性能参数。MIB 使用层次结构，其中包含多个标准 MIB 和私有 MIB，各种设备和厂商可以根据需要扩展 MIB。有多个 SNMP 版本，最常见的是 SNMPv1、SNMPv2c 和 SNMPv3。SNMPv1 是最早的版本，提供基本的功能，但安全性有限，通信基于明文。SNMPv2c 基于 SNMPv1，增加了一些新的功能，但仍然存在安全性问题。SNMPv3 是最安全的版本，提供加密、认证和访问控制，以保护 SNMP 通信的安全性。SNMP 定义了一组操作，包括 GET（获取单个值）、GETNEXT（获取下一个值）、SET（设置值）和TRAP（主动通知）。管理者使用这些操作来检索和配置代理中的信息。GET 操作允许管理者从代理中检索特定对象的值，而 SET 操作允许管理者更改特定对象的值。TRAP 操作用于代理向管理者发送事件通知，以便及时处理问题。SNMP 的安全性问题在早期版本中较为突出，因为通信是基于明文的，并没有强制的身份验证和加密机制。SNMPv3 添加了安全功能，包括加密、身份验证和访问控制，以提高通信的安全性。SNMP 主要用于监控和管理网络设备，如路由器、交换机、服务器和打印机等。它也可以用于监控操作系统、应用程序和数据库等。

* IMAP

IMAP（Internet Message Access Protocol）是一种用于电子邮件客户端和电子邮件服务器之间的通信协议，用于访问和管理电子邮件。IMAP的主要目的是允许用户在多个设备上访问并同步其电子邮件，以便在任何地方都能够查看相同的邮件内容和状态

IMAP允许电子邮件存储在电子邮件服务器上，而不是下载到本地计算机上。这意味着邮件始终保留在服务器上，用户可以从任何支持IMAP的设备上访问它们。IMAP支持文件夹（也称为邮箱）和标签，使用户能够将邮件组织成不同的类别。用户可以创建文件夹来存储和组织邮件，并将标签附加到邮件以进行分类。由于电子邮件存储在服务器上，因此用户可以在多个设备上访问相同的邮件。无论用户使用电子邮件客户端、Web邮件界面还是移动应用程序，它们都可以同步查看邮件的状态、文件夹结构和已读/未读状态等信息。IMAP允许用户在电子邮件服务器上执行各种操作，如查看、删除、标记、移动和复制邮件。这些操作都可以从客户端应用程序中发出，并反映在服务器上。IMAP客户端通常具有离线访问功能，允许用户在没有互联网连接的情况下查看以前下载的邮件。这些邮件通常以离线模式存储在本地设备上。IMAP通常通过TLS/SSL加密来保护邮件通信的安全性，以防止邮件内容在传输过程中被窃听。

* SSH

SSH（Secure Shell）是一种用于远程访问和安全数据传输的网络协议和工具。SSH的主要目的是提供安全的远程管理和文件传输，以保护敏感数据和维护系统的安全性。

SSH允许用户通过网络远程登录到远程计算机或服务器，以执行命令行操作和管理远程系统。用户可以使用SSH客户端与SSH服务器建立安全连接，然后在远程计算机上执行命令。SSH使用加密技术来保护通信的机密性和完整性。所有传输的数据都经过加密，使得窃听者无法读取或篡改通信内容。SSH要求用户进行身份验证，以确认其身份。最常见的身份验证方法是基于密码的身份验证，用户需要提供用户名和密码。另外，SSH还支持基于公钥的身份验证，其中用户可以使用密钥对进行身份验证，而不是密码。这种方式更安全，因为它不需要在网络上传输密码。SSH通常使用TCP协议，并监听默认端口号22。管理员可以配置SSH服务器以使用不同的端口号，以提高安全性。除了远程登录，SSH还支持安全文件传输。使用SCP（Secure Copy Protocol）或SFTP（SSH File Transfer Protocol）可以在本地计算机和远程服务器之间安全地传输文件。SSH允许用户设置代理和端口转发，以便在远程服务器上访问本地服务或将流量路由到远程服务器。这对于穿越防火墙或访问内部网络资源非常有用。SSH支持密钥对的生成和管理。用户可以生成公钥和私钥，然后将公钥上传到SSH服务器，以便进行基于公钥的身份验证。这种方式比密码更安全，尤其是对于自动化任务和无人值守操作。

* TLS/SSL

TLS（Transport Layer Security）和SSL（Secure Sockets Layer）是用于网络通信中的加密协议，用于保护数据的机密性和完整性。它们是用于安全通信的重要工具，用于加密数据传输和验证通信方的身份。

TLS/SSL使用加密算法来加密通过网络传输的数据，使其在传输过程中无法被窃听者读取。这有助于保护敏感信息，如登录凭据、信用卡号码和个人数据。TLS/SSL还通过使用消息摘要算法来验证传输的数据的完整性，以确保在传输过程中没有被篡改。TLS/SSL允许通信方验证对方的身份，以确保与正确的目标通信。这通常是通过使用数字证书来实现的。数字证书由可信的证书颁发机构（CA）签发，包含了公钥和相关信息。客户端和服务器可以使用证书来验证对方的身份，并确保通信的安全性。当客户端和服务器建立TLS/SSL连接时，它们会执行握手过程，以协商加密算法、生成会话密钥并进行身份验证。握手过程包括多个步骤，包括客户端向服务器发送“客户端Hello”、服务器向客户端发送“服务器Hello”、协商加密算法、服务器身份验证以及生成会话密钥等步骤。SSL有多个版本，包括SSLv2、SSLv3和TLS 1.0。然而，SSLv2和SSLv3存在安全漏洞，不再安全，应该避免使用。目前，TLS 1.2和TLS 1.3是最常用的TLS版本，它们提供了更强大的安全性和性能。TLS/SSL广泛用于保护各种类型的通信，包括Web浏览器与服务器之间的HTTPS通信、电子邮件传输、虚拟私人网络（VPN）连接、VoIP通信和移动应用程序通信等。HTTPS（HTTP Secure）是使用TLS/SSL协议来加密和保护HTTP通信的协议。它是安全的Web通信标准，用于保护网站和用户之间的数据传输。

* MQTT

MQTT（Message Queuing Telemetry Transport）是一种轻量级、发布-订阅消息传输协议，旨在在低带宽和不稳定网络环境中传输数据。MQTT最初由IBM开发，现在是一个开放的标准，广泛用于物联网（IoT）应用程序、传感器网络、实时通信和许多其他领域。

MQTT使用发布-订阅模型，其中有两种主要角色：发布者（Publisher）和订阅者（Subscriber）。发布者负责将消息发布到特定的主题（Topic），而订阅者则订阅一个或多个主题以接收相关消息。这种模型允许消息的生产者和消费者之间实现解耦，发布者不需要知道哪些订阅者在接收消息，而订阅者不需要知道消息来自哪个发布者。MQTT支持不同的消息传递质量服务级别（Quality of Service，QoS），以确保消息可靠性。有三个QoS级别：QoS 0（至多一次交付）、QoS 1（至少一次交付）和QoS 2（恰好一次交付）。用户可以根据应用程序需求选择适当的QoS级别。MQTT被设计为轻量级协议，其协议头较小，占用较少的网络带宽和系统资源。这使得它适用于资源有限的设备和低带宽网络。MQTT支持保留消息（Retained Messages），允许发布者设置一个主题的保留消息。当订阅者订阅该主题时，它会立即接收到最新的保留消息，而不仅仅是最新的发布消息。发布者可以设置遗嘱消息（Last Will and Testament，LWT），以便在其断开连接时通知订阅者。这可以用于检测设备的离线状态或异常情况。MQTT客户端可以选择是否在连接期间维护会话状态，以确保在连接断开后可以恢复。此外，MQTT支持持久性会话，以确保消息在客户端离线期间不会丢失。MQTT可以与其他安全协议（如TLS/SSL）一起使用，以加密通信，确保数据的安全性和保密性。此外，可以配置用户名和密码来进行身份验证。

原文链接：<https://blog.csdn.net/qq_41874746/article/details/130529174>

# 第三章：软件调试

## 1. GDB

### GDB 简介

GDB 全称 “GNU symbolic debugger”，诞生于 GNU 计划（同时诞生的还有 GCC、Emacs 等），是 Linux 下默认的程序调试器。发展至今，GDB 已经迭代了诸多个版本，当下的 GDB 支持调试多种编程语言编写的程序，包括 C、C++、Go、Objective-C、OpenCL、Ada 等。实际场景中，GDB 更常用来调试 C 和 C++程序。它允许开发者在程序运行时查看和修改内存内容、寄存器状态，设置断点，单步执行代码，以及分析程序的行为，帮助找出程序中的错误和问题。GDB 通常用于 UNIX 和 Linux 系统上，但也可以用于其他操作系统。

GDB 具有一些特点：

* **源代码级调试**：GDB允许开发者以源代码级别进行调试，这意味着您可以在程序中的特定源代码行上设置断点，查看变量的值，并逐步执行代码。
* **多种编程语言支持**：GDB支持多种编程语言，包括C、C++、Fortran、Ada等。它可以与多种编译器搭配使用，如GNU Compiler Collection（GCC）。
* **命令行界面**：GDB通常以命令行界面的方式运行，开发者通过输入命令来与GDB进行交互。虽然它没有图形用户界面（GUI），但它提供了强大的功能和灵活性。
* **断点和观察点**：GDB允许您在程序中设置断点，以中断程序的执行，并在断点处进行调试。您还可以设置观察点，以在特定条件满足时中断程序执行。
* **调试信息**：为了进行源代码级的调试，程序必须以调试信息编译。这些调试信息包含了源代码位置、变量名和类型等信息，帮助GDB正确地关联机器代码和源代码。
* **核心转储分析**：GDB可以分析核心转储文件，这是在程序崩溃时生成的文件，其中包含了程序在崩溃时的状态信息。这有助于识别程序中的错误。
* **脚本支持**：GDB支持脚本编程，您可以编写Python或其他脚本来自动化和扩展调试过程。
* **远程调试**：GDB支持远程调试，允许您在远程计算机上调试运行中的程序。

GDB是一个强大的调试工具，用于识别和修复程序中的错误，从而提高软件质量和可靠性。它是开发者在开发和维护复杂程序时的重要工具之一，并在各种 UNIX 和 Linux 开发环境中广泛使用。虽然它的学习曲线可能较陡，但一旦熟悉了 GDB 的基本用法，它将成为一个非常有价值的工具。除 gdb 外，linux 下比较有名的调试器还有 xxgdb , ddd , kgdb , ups 等等。

GDB 的启动界面：

nihao@nihao-z690:~$ gdb  
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86\_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word".  
>>>

通过 gdb 命令启动后，会进入 gdb 终端界面，通过命令与 gdb 进行交互。

### GDB 安装

**通过包管理器进行安装:**

sudo apt install gdb

**源码安装 GDB**

Linux 传统的安装方法就是源码安装，可以在 GDB 的官网上下载 GDB 的源码， GDB 官网地址：<http://www.gnu.org/software/gdb/download/>。在作者写书的时候， GDB 版本已经到了 GDB 14.1 。下载最新的版本 [gdb-14.1.tar.xz](https://ftp.gnu.org/gnu/gdb/gdb-14.1.tar.xz) ，进行编译。

nihao@nihao-z690:~/work/test/gdb$ wget https://ftp.gnu.org/gnu/gdb/gdb-14.1.tar.xz  
--2023-12-21 16:29:38-- https://ftp.gnu.org/gnu/gdb/gdb-14.1.tar.xz  
正在解析主机 ftp.gnu.org (ftp.gnu.org)... 209.51.188.20  
正在连接 ftp.gnu.org (ftp.gnu.org)|209.51.188.20|:443... 已连接。  
已发出 HTTP 请求，正在等待回应... 200 OK  
长度： 24108624 (23M) [application/x-xz]  
正在保存至: ‘gdb-14.1.tar.xz’  
  
gdb-14.1.tar.xz 100%[===================>] 22.99M 5.83MB/s 用时 6.3s   
  
2023-12-21 16:29:45 (3.66 MB/s) - 已保存 ‘gdb-14.1.tar.xz’ [24108624/24108624])  
  
nihao@nihao-z690:~/work/test/gdb$ tar -xJf gdb-14.1.tar.xz   
nihao@nihao-z690:~/work/test/gdb$ cd gdb-14.1/  
nihao@nihao-z690:~/work/test/gdb/gdb-14.1$ mkdir build  
nihao@nihao-z690:~/work/test/gdb/gdb-14.1$ cd build/  
nihao@nihao-z690:~/work/test/gdb/gdb-14.1/build$ ../configure  
checking build system type... x86\_64-pc-linux-gnu  
checking host system type... x86\_64-pc-linux-gnu  
checking target system type... x86\_64-pc-linux-gnu  
checking for a BSD-compatible install... /usr/bin/install -c  
checking whether ln works... yes  
checking whether ln -s works... yes  
checking for a sed that does not truncate output... /usr/bin/sed  
checking for gawk... gawk  
checking for gdbserver support... yes  
checking for gcc... gcc  
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... no  
checking for suffix of object files... o  
checking whether we are using the GNU C compiler... yes  
checking whether gcc accepts -g... yes  
checking for gcc option to accept ISO C89... none needed  
checking for gcc option to accept ISO C99... none needed  
checking for g++... g++  
checking whether we are using the GNU C++ compiler... yes  
checking whether g++ accepts -g... yes  
checking whether g++ accepts -static-libstdc++ -static-libgcc... yes  
checking for gnatbind... no  
checking for gnatmake... no  
checking whether compiler driver understands Ada and is recent enough... no  
checking for gdc... no  
checking whether the D compiler works... no  
checking how to compare bootstrapped objects... cmp --ignore-initial=16 $$f1 $$f2  
checking for objdir... .libs  
checking for the correct version of gmp.h... yes  
checking for the correct version of mpfr.h... yes  
checking for the correct version of the gmp/mpfr libraries... yes  
checking for isl 0.15 or later... no  
required isl version is 0.15 or later  
checking for default BUILD\_CONFIG...   
checking for --enable-vtable-verify... no  
checking for bison... bison -y  
checking for bison... bison  
checking for gm4... no  
checking for gnum4... no  
checking for m4... m4  
checking for flex... flex  
checking for flex... flex  
checking for makeinfo... makeinfo  
checking for expect... expect  
checking for runtest... no  
checking for ar... ar  
checking for as... as  
checking for dlltool... no  
checking for dsymutil... no  
checking for ld... ld  
checking for lipo... no  
checking for nm... nm  
checking for ranlib... ranlib  
checking for strip... strip  
checking for windres... no  
checking for windmc... no  
checking for objcopy... objcopy  
checking for objdump... objdump  
checking for otool... no  
checking for readelf... readelf  
checking for -plugin option... --plugin /usr/lib/gcc/x86\_64-linux-gnu/11/liblto\_plugin.so  
checking for cc... cc  
checking for c++... c++  
checking for gcc... gcc  
checking for gfortran... no  
checking for gccgo... no  
checking for gdc... no  
checking for gm2... no  
checking for ar... ar  
checking for as... as  
checking for dlltool... no  
checking for dsymutil... no  
checking for ld... ld  
checking for lipo... no  
checking for nm... nm  
checking for objcopy... objcopy  
checking for objdump... objdump  
checking for otool... no  
checking for ranlib... ranlib  
checking for readelf... readelf  
checking for strip... strip  
checking for windres... no  
checking for windmc... no  
checking where to find the target ar... host tool  
checking where to find the target as... host tool  
checking where to find the target cc... host tool  
checking where to find the target c++... host tool  
checking where to find the target c++ for libstdc++... host tool  
checking where to find the target dlltool... host tool  
checking where to find the target dsymutil... host tool  
checking where to find the target gcc... host tool  
checking where to find the target gfortran... host tool  
checking where to find the target gccgo... host tool  
checking where to find the target gdc... host tool  
checking where to find the target gm2... host tool  
checking where to find the target ld... host tool  
checking where to find the target lipo... host tool  
checking where to find the target nm... host tool  
checking where to find the target objcopy... host tool  
checking where to find the target objdump... host tool  
checking where to find the target otool... host tool  
checking where to find the target ranlib... host tool  
checking where to find the target readelf... host tool  
checking where to find the target strip... host tool  
checking where to find the target windres... host tool  
checking where to find the target windmc... host tool  
checking whether to enable maintainer-specific portions of Makefiles... no  
configure: creating ./config.status  
config.status: creating Makefile  
nihao@nihao-z690:~/work/test/gdb/gdb-14.1/build$ make  
.....  
make[4]: 离开目录“/home/nihao/work/test/gdb/gdb-14.1/build/gdb/data-directory”  
make[3]: 离开目录“/home/nihao/work/test/gdb/gdb-14.1/build/gdb”  
make[2]: 离开目录“/home/nihao/work/test/gdb/gdb-14.1/build/gdb”  
make[1]: 离开目录“/home/nihao/work/test/gdb/gdb-14.1/build  
nihao@nihao-z690:~/work/test/gdb/gdb-14.1/build$ sudo make install  
nihao@nihao-z690:~/work/test/gdb/gdb-14.1/build$ gdb -v  
GNU gdb (GDB) 14.1  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.

上面是安装过程，当执行 gdb -v 后输出了 gdb 版本，说明安装成功了。

### GDB 基本使用

新编写一个程序用于学习 GDB 调试工具的使用。

// test.c  
#include <stdio.h>  
#include <stdlib.h>  
// 函数声明  
int divide(int a, int b);  
  
int main(int argc, char \*argv[]) {  
 int x,y,result;  
 x = atoi(argv[1]);  
 y = atoi(argv[2]);  
 printf("开始计算...\n");  
   
 result = divide(x, y);  
   
 printf("%d / %d 结果是：%d\n",x, y, result);  
   
 result = divide(x, y);  
   
 printf("%d / %d 结果是：%d\n",x, y, result);  
 return 0;  
}  
  
// 函数定义  
int divide(int a, int b) {  
 if (b == 0) {  
 printf("错误：除数不能为0！\n");  
 return -1;  
 }  
 return a / b;  
}

添加可调试的选项然后编译这个程序：

root@M5Core135:~/tmp# gcc -g test.c -o test  
root@M5Core135:~/tmp# ls -la  
-rwxr-xr-x 1 root root 9740 7月21日 16:14 test  
-rw-r--r-- 1 root root 569 7月21日 16:12 test.c  
root@M5Core135:~/tmp# ./test 5 1  
开始计算...  
5 / 1 结果是：5  
5 / 1 结果是：5  
root@M5Core135:~/tmp# ./test 5 0  
开始计算...  
错误：除数不能为0！  
5 / 0 结果是：-1  
错误：除数不能为0！  
5 / 0 结果是：-1

现在我们有一个可调试的进程用于学习如何使用 gdb 进行调试。  
查看 gdb 版本:

root@M5Core135:~/tmp# gdb -v  
GNU gdb (GDB) 14.1  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.

查看 gdb 基本帮助:

root@M5Core135:~/tmp# gdb --help  
This is the GNU debugger. Usage:  
  
 gdb [options] [executable-file [core-file or process-id]]  
 gdb [options] --args executable-file [inferior-arguments ...]  
  
Selection of debuggee and its files:  
  
 --args Arguments after executable-file are passed to inferior.  
 --core=COREFILE Analyze the core dump COREFILE.  
 --exec=EXECFILE Use EXECFILE as the executable.  
 --pid=PID Attach to running process PID.  
 --directory=DIR Search for source files in DIR.  
 --se=FILE Use FILE as symbol file and executable file.  
 --symbols=SYMFILE Read symbols from SYMFILE.  
 --readnow Fully read symbol files on first access.  
 --readnever Do not read symbol files.  
 --write Set writing into executable and core files.  
  
Initial commands and command files:  
  
 --command=FILE, -x Execute GDB commands from FILE.  
 --init-command=FILE, -ix  
 Like -x but execute commands before loading inferior.  
 --eval-command=COMMAND, -ex  
 Execute a single GDB command.  
 May be used multiple times and in conjunction  
 with --command.  
 --init-eval-command=COMMAND, -iex  
 Like -ex but before loading inferior.  
 --nh Do not read ~/.gdbinit.  
 --nx Do not read any .gdbinit files in any directory.  
  
Output and user interface control:  
  
 --fullname Output information used by emacs-GDB interface.  
 --interpreter=INTERP  
 Select a specific interpreter / user interface.  
 --tty=TTY Use TTY for input/output by the program being debugged.  
 -w Use the GUI interface.  
 --nw Do not use the GUI interface.  
 --tui Use a terminal user interface.  
 -q, --quiet, --silent  
 Do not print version number on startup.  
  
Operating modes:  
  
 --batch Exit after processing options.  
 --batch-silent Like --batch, but suppress all gdb stdout output.  
 --return-child-result  
 GDB exit code will be the child's exit code.  
 --configuration Print details about GDB configuration and then exit.  
 --help Print this message and then exit.  
 --version Print version information and then exit.  
  
Remote debugging options:  
  
 -b BAUDRATE Set serial port baud rate used for remote debugging.  
 -l TIMEOUT Set timeout in seconds for remote debugging.  
  
Other options:  
  
 --cd=DIR Change current directory to DIR.  
 --data-directory=DIR, -D  
 Set GDB's data-directory to DIR.  
  
At startup, GDB reads the following early init files and executes their  
commands:  
 None found.  
  
At startup, GDB reads the following init files and executes their commands:  
 \* system-wide init files: /etc/gdb/gdbinit  
  
For more information, type "help" from within GDB, or consult the  
GDB manual (available as on-line info or a printed manual).  
  
Report bugs to <https://www.gnu.org/software/gdb/bugs/>.  
  
You can ask GDB-related questions on the GDB users mailing list  
(gdb@sourceware.org) or on GDB's IRC channel (#gdb on Freenode).

进入 gdb 命令界面：

root@M5Core135:~/tmp# gdb  
GNU gdb (Debian 13.1-3) 13.1  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "arm-linux-gnueabihf".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word".  
(gdb)

载入我们的可执行文件,也可以在 gdb 运行时直接以gdb test开始 :

(gdb) file test  
Reading symbols from test...

设置命令行参数:

(gdb) set args 1 5

运行程序:

(gdb) run  
Starting program: /root/tmp/test 1 5  
[Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/arm-linux-gnueabihf/libthread\_db.so.1".  
开始计算...  
1 / 5 结果是：0  
1 / 5 结果是：0  
[Inferior 1 (process 856) exited normally]

退出:

(gdb) quit  
root@M5Core135:~/tmp#

**GDB 步调试**

root@M5Core135:~/tmp# gdb test  
GNU gdb (Debian 13.1-3) 13.1  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "arm-linux-gnueabihf".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from test...  
# 设置命令行参数  
(gdb) set args 5 1  
# 设置断点在 main 函数处  
(gdb) b main  
Breakpoint 1 at 0x596: file test.c, line 8.  
# 运行程序，遇到断点停止  
(gdb) r  
Starting program: /root/tmp/test 5 1  
[Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/arm-linux-gnueabihf/libthread\_db.so.1".  
  
Breakpoint 1, main (argc=3, argv=0xbefffb74) at test.c:8  
8 x = atoi(argv[1]);  
# 单步运行进程  
(gdb) n  
9 y = atoi(argv[2]);  
(gdb)   
10 printf("开始计算...\n");  
# 打印出当前变量  
(gdb) info locals   
x = 5  
y = 1  
result = 0  
(gdb) n  
开始计算...  
12 result = divide(x, y);  
# 单步进入运行  
(gdb) s  
divide (a=5, b=1) at test.c:24  
24 if (b == 0) {  
(gdb) n  
28 return a / b;  
(gdb)   
29 }  
(gdb)   
main (argc=3, argv=0xbefffb74) at test.c:14  
14 printf("%d / %d 结果是：%d\n",x, y, result);  
(gdb) info locals   
x = 5  
y = 1  
result = 5  
(gdb) n  
5 / 1 结果是：5  
16 result = divide(x, y);  
# 设置变量 y 的值为0  
(gdb) print y=0  
$1 = 0  
(gdb) info locals   
x = 5  
y = 0  
result = 5  
(gdb) n  
错误：除数不能为0！  
18 printf("%d / %d 结果是：%d\n",x, y, result);  
(gdb)   
5 / 0 结果是：-1  
19 return 0;  
(gdb)   
20 }  
(gdb)   
\_\_libc\_start\_call\_main (main=main@entry=0x40058d <main>, argc=argc@entry=3, argv=0xbefffb74, argv@entry=0xb6fc8000)  
 at ../sysdeps/nptl/libc\_start\_call\_main.h:74  
74 ../sysdeps/nptl/libc\_start\_call\_main.h: No such file or directory.  
(gdb)   
# 退出调试  
(gdb) q   
A debugging session is active.  
  
 Inferior 1 [process 8449] will be killed.  
# 确认是否退出调试，y 为退出，n 为不退出  
Quit anyway? (y or n) y

**更多 GDB 命令**

|  |  |  |
| --- | --- | --- |
| 命令 | 简单输入 | 解释 |
| attach pid |  | 动态链接到正在运行的 pid 进程，链接成功后目标进程将停止执行 |
| file program |  | 载入目标可执行文件 |
| set args arg1 arg2 |  | 设置运行输入的命令行参数 |
| show args |  | 显示设定的程序运行的命令行参数 |
| run | r | 运行程序，调试程序从这个命令开始 |
| list | l | 打印出当前正在运行的源代码 |
| break | b | 设置运行断点，当程序运行到此处时暂停 |
| info break | i b | 打印出已经设置的断点信息 |
| delete breakpoint 1 |  | 删除断点1 |
| disable breakpoint 1 |  | 禁用断点1 |
| enable breakpoint 1 |  | 起用断点1 |
| clean number |  | 删除源文件文件 number 行中的断点 |
| continue | c | 继续运行程序，直到下一个断点 |
| watch |  | 在程序中设置一个监测点（即数据断点） |
| display |  | 打印出观察的变量 |
| undisplay |  | display命令的反命令，不要显示表达式 |
| print | Quit anyway? (y or n) y <-- 确认是否退出调试，y 为退出，n 为不退出p | 查看运行时的变量以及表达式，也可以用print n=10 给变量 |
| print h@10 | p h@10 | 打印变量 h 数组 10 个值 |
| step | s | 单步运行，遇到子函数跳入其中运行 |
| next | n | 单步运行，遇到子函数不进入运行 |
| finish |  | 继续运行程序直到跳出当前栈 |
| until | u | 继续运行直到跳出循环（适合在循环内使用） |
| show convenience |  | 查看当前的所有环境变量 |
| set env env\_name=value |  | 设置环境变量 |
| set var var\_name=value |  | 设置变量，var\_name必须是调试环境中的变量名称 |
| print var\_name=value |  | 修改变量值 |
| info functions |  | 查看程序中函数符号 |
| info args |  | 查看当前函数参数的值 |
| info locals |  | 查看当前局部变量的值 |
| info variables |  | 查看程序中的变量符号 |
| info registers |  | 查看函数寄存器信息 |
| info frame |  | 查看当前函数调用的栈帧信息 |
| frame n |  | 切换到栈编号为N的函数栈帧 |
| backtrace |  | 查看函数调用的顺序，函数调用栈信息 |
| disassemble function |  | 查看函数反汇编代码 |
| set debug entry-values 1 |  | 打印尾调用堆栈帧信息 |
| call function |  | 强制调用某函数，会显示函数返回值（如果函数返回值不是void）。 |
| start function |  | 执行函数，并停在函数开始位置 |
| cd |  | 改变当前工作目录 |
| clear |  | 删除刚才停止处的断点 |
| search text |  | 在源文件中搜索正规表达式 |
| set history expansion on |  | 可使用 set history expansion on 命令 |
| commands |  | 命中断点时，列出将要执行的命令 |
| up |  | 上移栈帧，使另一函数成为当前函数 |
| down |  | 下移栈帧，使得另一个函数成为当前函数 |
| jump |  | 在源程序中的另一点开始运行 |
| kill |  | 异常终止在 gdb 控制下运行的程序 |
| pwd |  | 显示当前工作目录 |
| pype |  | 显示一个数据结构（如一个结构或C++类）的内容 |
| quit |  | 退出gdb |
| signal |  | 将一个信号发送到正在运行的进程 |
| whatis |  | 显示变量或函数类型 |

**GDB 调试 core dump**

Quit anyway? (y or n) y <-- 确认是否退出调试，y 为退出，n 为不退出一个有错误的 C 源程序 bugging.c

// file: bugging.c  
#include<stdio.h>  
static char buff [256];  
static char\* string;  
int main (int argc , char \*argv[])  
{  
 printf ("Please input a string: ");  
 gets (string);  
 printf ("\nYour string is: %s\n", string);  
 return 0;  
}

上面这个程序非常简单，其目的是接受用户的输入，然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串地址 string，因此，编译并运行之后，将出现 Segment Fault 错误：

root@M5Core135:~/tmp# ulimit -c unlimited  
root@M5Core135:~/tmp# gcc -g bugging.c -o bugging  
root@M5Core135:~/tmp# ./bugging   
Please input a string: nihao  
  
Segmentation fault (core dumped)  
root@M5Core135:~/tmp# ls  
bugging bugging.c core  
root@M5Core135:~/tmp# gdb bugging core   
GNU gdb (Debian 13.1-3) 13.1  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "arm-linux-gnueabihf".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from bugging...  
[New LWP 608]  
[Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/arm-linux-gnueabihf/libthread\_db.so.1".  
Core was generated by `./bugging'.  
Program terminated with signal SIGSEGV, Segmentation fault.  
#0 \_\_GI\_strlen () at ../sysdeps/arm/armv6t2/strlen.S:126  
126 ../sysdeps/arm/armv6t2/strlen.S: No such file or directory.  
(gdb) bt  
#0 \_\_GI\_strlen () at ../sysdeps/arm/armv6t2/strlen.S:126  
#1 0xb6ed5dd8 in \_\_vfprintf\_internal (s=0xb6fa0d50 <\_IO\_2\_1\_stdout\_>,   
 format=format@entry=0xb6fd3d80 <\_rtld\_global\_ro> "", ap=..., ap@entry=...,   
 mode\_flags=mode\_flags@entry=0) at vfprintf-process-arg.c:397  
#2 0xb6eceb4e in \_\_printf (format=0x4c3660 "\nYour string is: %s\n")  
 at printf.c:33  
#3 0x004c358a in main (argc=1, argv=0xbe902bb4) at bugging.c:8  
(gdb) info locals   
No locals.  
(gdb) quit

可以看到程序崩溃时运行到了 printf 函数，通过查看栈的形式，我们能检查程序运行在哪里发生了错误，从而让我们能够更快定位问题。

gdb启动时会默认打印一堆免责条款，通过添加 --silent（或者 -q、--quiet）选项，可将这部分信息屏蔽掉。

### GDB 命令介绍

**载入程序**

gdb 可以调试多种状态的程序，未运行的程序，已运行的程序都可以进行调试。不同状态的程序有不同的调试载入方法。

静态装载：

# 第一种，命令行载入。 gdb + <可执行程序> + [选项]  
nihao@nihao-z690:~/work/test/gdb/test$ gdb test  
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86\_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from test...  
>>> quit  
  
# 第二种，在 gdb 中使用 file 命令加载  
nihao@nihao-z690:~/work/test/gdb/test$ gdb  
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86\_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word".  
>>> file test  
Reading symbols from test...  
>>> quit

运行时装载：

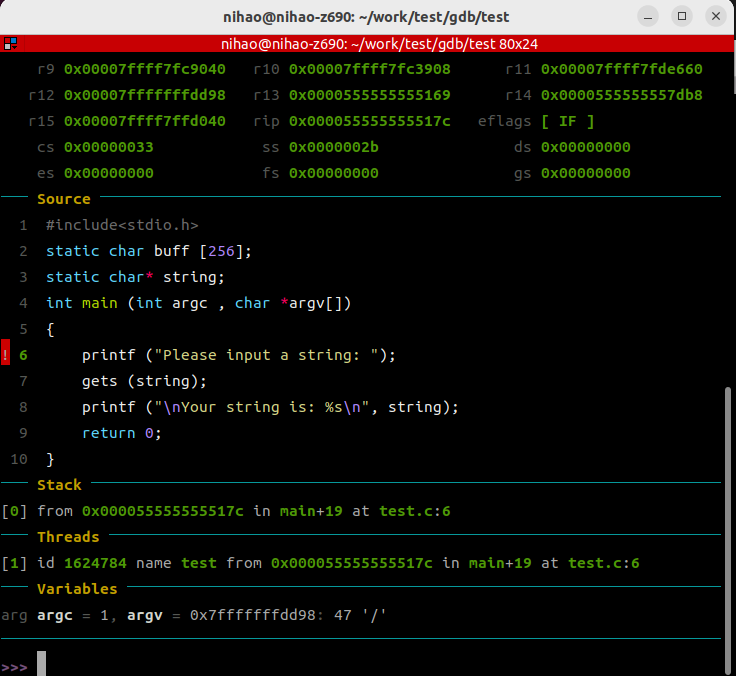
nihao@nihao-z690:~/work/test/gdb/test$ ./test &  
[1] 1617872  
Please input a string:   
[1]+ 已停止 ./test  
nihao@nihao-z690:~/work/test/gdb/test$ ps  
 PID TTY TIME CMD  
1482279 pts/3 00:00:00 bash  
1617872 pts/3 00:00:00 test  
1617876 pts/3 00:00:00 ps  
nihao@nihao-z690:~/work/test/gdb/test$ sudo gdb -p 1617872  
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86\_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word".  
Attaching to process 1617872  
Reading symbols from /home/nihao/work/test/gdb/test/test...  
Reading symbols from /lib/x86\_64-linux-gnu/libc.so.6...  
Reading symbols from /usr/lib/debug/.build-id/20/3de0ae33b53fee1578b117cb4123e85d0534f0.debug...  
Reading symbols from /lib64/ld-linux-x86-64.so.2...  
Reading symbols from /usr/lib/debug/.build-id/cc/cdd41e22e25f77a8cda3d045c57ffdb01a9793.debug...  
[Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".  
  
Program received signal SIGTTIN, Stopped (tty input).  
0x00007f20b2b145f2 in \_\_GI\_\_\_libc\_read (fd=0, buf=0x55aa9decc6b0, nbytes=1024) at ../sysdeps/unix/sysv/linux/read.c:26  
26 ../sysdeps/unix/sysv/linux/read.c: 没有那个文件或目录.  
(gdb) quit  
A debugging session is active.  
  
 Inferior 1 [process 1617872] will be detached.  
  
Quit anyway? (y or n) y  
Detaching from program: /home/nihao/work/test/gdb/test/test, process 1617872  
[Inferior 1 (process 1617872) detached]  
  
[1]+ 已停止 ./test

**启动程序**

GDB 提供了两种方式启动静态装载的程序，分别是 run 指令和 start 指令。他们之间的区别是：

run 会一直执行程序，直到遇到断点停止，或者到程序结束。

start 只会执行到 main() 函数起始位置，就是 main() 函数内的第一行语句停止执行。



可以根据不同的场景选择不同的启动方式。

**断点 break 命令**

break 命令，简化输入（b），输入的语法格式：

# 直接表达式  
(gdb) break location // b location  
  
# 判断表达式  
(gdb) break ... if cond // b .. if cond

location 用于指定打断点的具体位置，其表示方式有多种，如下表所示。

|  |  |
| --- | --- |
| location 的值 | 含义 |
| linenum | linenum 代表的是 gdb 当前代码页面中需打断点的代码所在行数。当前代码页可以用小写的 L 命令看到。 |
| filename:linenum | filename 是源码文件名，linenum 为源码文件中断点所在的行数。 |
| + offset - offset | + offset 表示在当前暂停处加上偏移行数处打上断点。 - offset 表示在当前暂停处减上偏移行数处打上断点。 |
| function | function 为需要在名为 function 函数入口处打一个断点。 |
| filename:function | 和 filename:linenum 相似，在名为 filename 的源码文件中的 function 函数入口打上断点。 |

在判断断点的表达式中，cond 为某个表达式，其含义是，当每次程序执行到 ... 位置的时候判断 cond 表达式是否成立，如果成立，就暂停执行。

示例：

# 直接添加  
(gdb) break main.c:42 if x==5

这行命令会在源文件main.c的第42行设置一个断点，并指定了一个条件，即当变量x的值等于5时，才会触发断点。这意味着程序只有在满足条件时才会在这一行暂停执行，否则会继续执行。

# 间接添加  
(gdb) break main.c:42 # 添加断点  
(gdb) condition x==5 # 为该断点添加条件

这是两步过程。首先，在源文件main.c的第42行设置了一个普通的断点，然后通过condition命令为这个断点添加了条件。条件是x==5，也就是说，只有当变量x的值等于5时，程序才会在这个断点处停止执行。

**tbreak和rbreak命令**

tbreak 命令打的断点被称为一次性断点，就是该断点仅会被暂停一次，下一次就不起作用了，同时该断点在作用后就会自动消失。

rbreak 命令是判断断点的扩展函数，rbreak 后跟的是一个正则表达式，可以在源码中批量添加断点。

(gdb) tbreak regex

regex 是正则表达式，源码中符合正则表达式的函数名都会在函数入口的位置打上断点。

**clear/delete 删除断点**

当前面创建的断点不需要的时候，我们需要删除这个断点。

删除断点命令为 clear 。使用的语法格式如下：

(gdb) clear location

location 可以是源码中的某行号，也可以是一个具体的函数名。执行完命令后，指定的断点将会被删除。

如果想要一次性删除所有的断点，可以使用 delete 命令。使用的语法格式如下：

# 删除所有的断点  
(gdb) d  
# or  
# (gdb) delete  
  
# 删除指定的断点  
(gdb) d [breakpoints] [num]  
# 简写  
# (gdb) d breakpoints 1 # 删除编号为 1 的断点。

breakpoints 为可选参数，num 为断点的编号，断点编号信息可以使用 info break 命令查看。

**disable 禁用断点**

如果不想删除断点，同时也不想让程序在断点处暂停，可以使用 disable 禁用断点，让程序不在该断点处暂停。disable 语法格式如下：

(gdb) disable [breakpoints] [num...]

breakpoints 为可选参数，和前面删除断点的意义一样，命令中有 breakpoints 时，只会禁用编号为 num 的断点。如果没有 breakpoints 选项， disable 会禁用当前所有的断点。

**enableable 激活断点**

对于处于禁用状态的断点，可以使用 enable 命令激活，该命令的语法格式有多种，分别对应有不同的功能：

# 激活用 num... 参数指定的多个断点，如果不设定 num...，表示激活所有禁用的断点  
(gdb) enable [breakpoints] [num...]   
  
# 临时激活以 num... 为编号的多个断点，但断点只能使用 1 次，之后会自动回到禁用状态  
(gdb) enable [breakpoints] once num…   
  
# 临时激活以 num... 为编号的多个断点，断点可以使用 count 次，之后进入禁用状态  
(gdb) enable [breakpoints] count num...   
  
# 激活 num.. 为编号的多个断点，但断点只能使用 1 次，之后会被永久删除。  
(gdb) enable [breakpoints] delete num…

其中，breakpoints 参数可有可无；num...表示可以提供多个断点的编号，enable命令可以同时激活多个断点。

**watch 观察断点**

这是当程序中某个变量发生变换后就停止执行，并打印出变量内容的一类断点，该断点可以帮助我们在变量发生变换的时候及时停下来检查程序。要知道程序正常执行时是非常快的，靠人力观察变量的变化是不可能的。watch 表达式如下：

(gdb) watch cond

cond 为要监控的变量的值。

watch 有两个扩展命令：

# 程序中出现读取目标变量（表达式）的值的操作，程序就会暂停运行  
(gdb) rwatch cond  
  
# 程序中出现读取目标变量（表达式）的值或者改变值的操作，程序就会暂停运行  
(gdb) awatch cond

**print 打印变量值命令**

它的功能就是在 GDB 调试程序的过程中，动态的输出或者修改指定变量或者表达式的值。

print 命令可以缩写为 p，最常用的语法格式如下所示：编译运行

￼  
 ./test  
Segmentation fault (core dumped) <-- 发生段错误，并生成了 core 文件  
可以根据生成时间查找core dump文件

￼  
ls /home/homework/coresave -hl | grep test  
-rw-rw-rw- 1 root root 400K Mar 13 15:08 core.test.27725.1615619332  
-rw-rw-rw- 1 root root 400K Mar 13 15:26 core.test.7791.1615620408  
-rw-rw-rw- 1 root root 540K Mar 11 10:29 core.test.1868.1615429740  
-rw-rw-rw- 1 root root 400K Mar 13 15:07 core.test.26880.1615619264  
-rw-rw-rw- 1 root root 404K Mar 3 19:42 core.test.28802.1614771771  
用gdb进行调试

￼  
$ gdb test /home/homework/coresave/core.test1.7791.1615620408 -q  
Reading symbols from /home/zhudi/project/linux/blog/gdb/test...done.

warning: core file may notELF格式是是UNIX系统实验室作为ABI（Application Binary Interface）而开发和发布的。 match specified executable file.  
[New LWP 7791]  
Core was generated by `./test'.  
Program terminated with signal 11, Segmentation fault.  
#0 0x00000000004005bd in main () at core.cpp:5  
5 \*a = 2;  
由此可见，程序崩溃了在第五行，定位到了出现问题的代码位置。

# 打印变量 variable 的内容  
(gdb) print variable  
# 简化命令  
# (gdb) p variable

其中，参数 variable 用来代指要查看或者修改的目标变量或者表达式。

当程序中包含多个作用域不同但名称相同的变量或表达式时，可以借助::运算符明确指定要查看的目标变量或表达式。::运算符的语法格式如下：

(gdb) print file::variable  
(gdb) print function::variable

其中 file用于指定具体的文件名，funciton 用于指定具体所在函数的函数名，variable表示要查看的目标变量或表达式。

另外，print 也可以打印出类或者结构体变量的值。

**display 命令**

和 print 命令一样，display 命令也用于调试阶段查看某个变量或表达式的值，它们的区别是，使用 display 命令查看变量或表达式的值，每当程序暂停执行（例如单步执行）时，GDB 调试器都会自动帮我们打印出来，而 print 命令则不会。

也就是说，使用 1 次 print 命令只能查看 1 次某个变量或表达式的值，而同样使用 1 次 display 命令，每次程序暂停执行时都会自动打印出目标变量或表达式的值。因此，当我们想频繁查看某个变量或表达式的值从而观察它的变化情况时，使用 display 命令可以一劳永逸。

display 命令没有缩写形式，常用的语法格式如下 2 种：

(gdb) display expr  
(gdb) display/fmt expr

注意，display 命令和 /fmt 之间不要留有空格。

**next 命令**

next 是最常用来进行单步调试的命令，其最大的特点是当遇到包含调用函数的语句时，无论函数内部包含多少行代码，next 指令都会一步执行完。也就是说，对于调用的函数来说，next 命令只会将其视作一行代码。

next 命令可以缩写为n 命令，使用方法也很简单，语法格式如下：

(gdb) next count

**step 命令**

通常情况下，step 命令和 next 命令的功能相同，都是单步执行程序。不同之处在于，当step 命令所执行的代码行中包含函数时，会进入该函数内部，并在函数第一行代码处停止执行。

step 命令可以缩写为 s 命令，用法和 next 命令相同，语法格式如下：

(gdb) step count

**until 命令**

until 命令可以简写为 u 命令，有 2 种语法格式，如下所示：

(gdb) until  
(gdb) until location

其中，参数 location为某一行代码的行号。

不带参数的 until 命令，可以使 GDB 调试器快速运行完当前的循环体，并运行至循环体外停止。注意，until 命令并非任何情况下都会发挥这个作用，只有当执行至循环体尾部（最后一行代码）时，until 命令才会发生此作用；反之，until 命令和 next 命令的功能一样，只是单步执行程序。

**return 命令**

实际调试时，在某个函数中调试一段时间后，可能不需要再一步步执行到函数返回处，希望在当前位置返回并退出函数，这时可以使用 return 命令。

(gdb) return

使用 return 命令时要小心，因为 return 命令并不会改变函数的实际返回值，函数的返回值通常是由函数内部的 `return 语句执行的结果决定，gdb 无法改变这个值。当用户使用 return 命令退出当前函数时，函数的返回值将是函数内部计算或指定的值，或者取决于函数的默认行为。在大多数情况下，函数的返回值将是未定义的或不确定的，因为 GDB 不会模拟函数的实际执行，而只是跳过函数的剩余部分。

如果您想要更改函数的返回值以进行调试，通常需要修改程序的源代码并重新编，或者在返回之后手动设置返回值。

**finish 命令**

finish 命令和 return 命令功能类似，有区别的是，finish 命令会执行函数到正常退出。

(gdb) finish

**jump 命令**

jump 命令的功能是直接跳到指定行继续执行程序，相当于 c 语言的 longjmp 语句。其语法格式为：

(gdb) jump location

其中，location 通常为某一行代码的行号。

值得注意的是，jump 破坏当前运行栈，跳过的一些变量的初始化或者栈初始化，容易导致程序崩溃。推荐在函数内部使用 jump ，相当于使用 goto 语句。location 必须是一个合法的代码地址，如果不合法，命令则执行不成功。

**search 命令**

search 提供代码搜索功能，用于在程序源代码中搜索符合正则表达式的函数，变量。该命令通常会和 list 命令配合使用，搜索和查看运行代码的上下文。

search 命令的语法格式为：

# 向前搜索  
(gdb) search <regexp>  
  
# 向后搜索  
(gdb) reverse-search <regexp>

regexp 源码中的字符串的郑则表达式。

**backtrace 命令**

backtrace 命令用于打印当前调试环境中栈帧的信息，用于查看当前执行位置的函数栈调用关系。常用的语法格式如下：

(gdb) backtrace [-full] [n]

backtrace 命令的示例输出：

(gdb) backtrace  
#0 main (argc=1, argv=0x7fffffffe488) at main.c:6  
#1 0x0000555555555157 in foo () at foo.c:4  
#2 0x0000555555555162 in bar () at bar.c:5

[-full] 为可选参数，用于控制在打印当前运行栈的同时打印出当前栈中的局部变量。

[n] 是一个整数值，当为正数时，会从栈 0 开始打印 n 个栈，当为负数时，会从当前栈向前打印 n 个栈。

注意，当调试多线程程序时，该命令仅用于打印当前线程中所有栈帧的信息。如果想要打印所有线程的栈帧信息，应执行thread apply all backtrace命令。

**frame 命令**

frame 用于打印栈帧信息的命令，默认打印当前栈，也可以打印指定栈。同时可以使用 up/down 两个命令调整当前栈。

# 打印当前栈  
(gdb) frame  
  
# 打印指定栈  
(gdb) frame spec  
  
# 向前调整 n 个栈。  
(gdb) up n  
  
# 向后调整 n 个栈。  
(gdb) down n

**info 命令**

借助 info 命令，我们可以查看当前栈帧中存储的信息：

(gdb) info frame

该命令会依次打印出当前栈帧的如下信息：

* 当前栈帧的编号，以及栈帧的地址。
* 当前栈帧对应函数的存储地址，以及该函数被调用时的代码存储的地址。
* 当前函数的调用者，对应的栈帧的地址。
* 编写此栈帧所用的编程语言。
* 函数参数的存储地址以及值。
* 函数中局部变量的存储地址。
* 栈帧中存储的寄存器变量，例如指令寄存器（64位环境中用 rip 表示，32为环境中用eip 表示）、堆栈基指针寄存器（64位环境用 rbp表示，32位环境用 ebp表示）等。

info args 命令查看程序的启动参数。

info locals 命令查看当前函数中各局部变量的值。

调试示例：

// file: debug\_file.c  
// compile cmd: gcc debug\_file.c -o debug\_file -g  
#include <stdio.h>  
  
void func4() {  
 int a = 4;  
 printf("func4:%d\n", a);  
}  
  
void func3() {  
 int b = 3;  
 printf("func3:%d\n", b);  
 func4();  
}  
  
void func2() {  
 int c = 2;  
 printf("func2:%d\n", c);  
 func3();  
}  
  
void func1() {  
 int d = 1;  
 printf("func1:%d\n", d);  
 func2();  
}  
  
int main() {  
 printf("main\n");  
 func1();  
 return 0;  
}

m5stack@m5stack-MS-7C94:~/work/test$ gcc debug\_file.c -o debug\_file -g  
m5stack@m5stack-MS-7C94:~/work/test$ gdb debug\_file   
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1  
Copyright (C) 2022 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86\_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
 <http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from debug\_file...  
# 启动程序  
(gdb) start  
Temporary breakpoint 1 at 0x124b: file debug\_file.c, line 29.  
Starting program: /home/m5stack/work/test/debug\_file   
[Thread debugging using libthread\_db enabled]  
Using host libthread\_db library "/lib/x86\_64-linux-gnu/libthread\_db.so.1".  
  
Temporary breakpoint 1, main () at debug\_file.c:29  
29 printf("main\n");  
# 打印源代码  
(gdb) l  
24 printf("func1:%d\n", d);  
25 func2();  
26 }  
27   
28 int main() {  
29 printf("main\n");  
30 func1();  
31 return 0;  
32 }  
# 执行一句语句  
(gdb) n  
main  
30 func1();  
# 单步递归执行  
(gdb) s  
func1 () at debug\_file.c:23  
23 int d = 1;  
(gdb) n  
24 printf("func1:%d\n", d);  
(gdb)   
func1:1  
25 func2();  
(gdb) s  
func2 () at debug\_file.c:17  
17 int c = 2;  
(gdb) n  
18 printf("func2:%d\n", c);  
(gdb)   
func2:2  
19 func3();  
(gdb) s  
func3 () at debug\_file.c:11  
11 int b = 3;  
(gdb) n  
12 printf("func3:%d\n", b);  
(gdb)   
func3:3  
13 func4();  
(gdb) s  
func4 () at debug\_file.c:6  
6 int a = 4;  
# 打印栈帧  
(gdb) backtrace  
#0 func4 () at debug\_file.c:6  
#1 0x00005555555551ce in func3 () at debug\_file.c:13  
#2 0x0000555555555207 in func2 () at debug\_file.c:19  
#3 0x0000555555555240 in func1 () at debug\_file.c:25  
#4 0x0000555555555264 in main () at debug\_file.c:30  
(gdb) backtrace 2  
#0 func4 () at debug\_file.c:6  
#1 0x00005555555551ce in func3 () at debug\_file.c:13  
(More stack frames follow...)  
(gdb) backtrace -2  
#3 0x0000555555555240 in func1 () at debug\_file.c:25  
#4 0x0000555555555264 in main () at debug\_file.c:30  
# 打印当前栈的同时，打印栈中的变量。  
(gdb) backtrace -full  
#0 func4 () at debug\_file.c:6  
 a = 32767  
#1 0x00005555555551ce in func3 () at debug\_file.c:13  
 b = 3  
#2 0x0000555555555207 in func2 () at debug\_file.c:19  
 c = 2  
#3 0x0000555555555240 in func1 () at debug\_file.c:25  
 d = 1  
#4 0x0000555555555264 in main () at debug\_file.c:30  
No locals.  
# 查看当前栈  
(gdb) fram  
#0 func4 () at debug\_file.c:6  
6 int a = 4;  
(gdb) fram 3  
#3 0x0000555555555240 in func1 () at debug\_file.c:25  
25 func2();  
# info 查看栈帧信息  
(gdb) info frame  
Stack level 3, frame at 0x7fffffffe1b0:  
 rip = 0x555555555240 in func1 (debug\_file.c:25); saved rip = 0x555555555264  
 called by frame at 0x7fffffffe1c0, caller of frame at 0x7fffffffe190  
 source language c.  
 Arglist at 0x7fffffffe1a0, args:   
 Locals at 0x7fffffffe1a0, Previous frame's sp is 0x7fffffffe1b0  
 Saved registers:  
 rbp at 0x7fffffffe1a0, rip at 0x7fffffffe1a8

[GDB调试命令详解](https://link.zhihu.com/?target=https%3A//blog.csdn.net/qq_28351609/article/details/114855630%3Fspm%3D1001.2101.3001.6650.2%26utm_medium%3Ddistribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-114855630-blog-80102076.235%5Ev38%5Epc_relevant_sort_base2%26depth_1-utm_source%3Ddistribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-114855630-blog-80102076.235%5Ev38%5Epc_relevant_sort_base2%26utm_relevant_index%3D3)*[小桥流水人家](https://link.zhihu.com/?target=https%3A//blog.csdn.net/qq_28351609/article/details/114855630%3Fspm%3D1001.2101.3001.6650.2%26utm_medium%3Ddistribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-114855630-blog-80102076.235%5Ev38%5Epc_relevant_sort_base2%26depth_1-utm_source%3Ddistribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-114855630-blog-80102076.235%5Ev38%5Epc_relevant_sort_base2%26utm_relevant_index%3D3)*[的博客-CSDN博客](https://link.zhihu.com/?target=https%3A//blog.csdn.net/qq_28351609/article/details/114855630%3Fspm%3D1001.2101.3001.6650.2%26utm_medium%3Ddistribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-114855630-blog-80102076.235%5Ev38%5Epc_relevant_sort_base2%26depth_1-utm_source%3Ddistribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-2-114855630-blog-80102076.235%5Ev38%5Epc_relevant_sort_base2%26utm_relevant_index%3D3)

<https://zhuanlan.zhihu.com/p/652799381>

## 2. Linux 栈回溯

栈回溯（stack trace）是指在程序执行期间记录和显示函数调用的堆栈信息。堆栈信息包括调用栈中的每个函数及其参数、局部变量和返回地址。栈回溯通常用于调试和分析程序的运行时行为，特别是在程序崩溃或产生错误时，它可以帮助定位问题的根本原因。学会使用栈回溯技能对分析程序错误非常有帮助。

栈回溯属于接近计算机底层基本原理，需要分析可执行程序的结构，以及 CPU 代码执行行为。

### ELF

ELF, Executable and Linking Format 是 Linux 二进制可执行文件、目标文件、共享库和核心转储的标准文件格式，在赋予可执行权限之后，计算机程序可以直接加载执行。ELF格式是是UNIX系统实验室作为ABI（Application Binary Interface）而开发和发布的。

最初的 UNIX 系统采用的是 a.out 二进制指令组织格式，有一个简单的头部，包括入口地址，代码长度，数据长度等基本信息，可被加载器加载到系统中执行。a.out 与平台有关，每种 CPU 都有一个特定的格式变种，不具备通用性。随着发展，被 System V 系统中的 COFF 二进制格式取代。最后被更加具备通用的性的 SVR4 中的 ELF 格式所取代，并一直沿用至今。但并不是所有的系统都采用这种二进制组织形式。

windows 中采用的是 COFF 格式的变种 PE 格式。

UNIX 变种 MAC OS X 使用的则是自定义的 Mach-O 格式。

ELF 在实际过程中主要有四种文件存在形式：

* 可重定位文件(Relocatable)。主要在 gcc 编译过程中产生的 .o 文件，此时的二进制代码处于散碎状态，还需被 Linker 进一步组织链接后才能被执行。
* 可执行文件(Executable)。这是二进制指令的可执行文件格式，内部包含进程影像，被系统加载后即可运行。
* 共享对象文件(Shared Object)。这种主要是动态库 so 文件， 当代码被 gcc 编译成二进制动态库时，将会以 elf 格式保存在 .so 文件中，用于多进程共享代码，减小内存使用。
* 核心转储文件(Core File)。也就是 linux 进程运行出错时产生的 core dump 文件，保存了程序运行时出现致命错误的现场。栈回溯主要就是分析这个文件。

### CPU 代码执行方式

CPU 的代码执行方式和 CPU 的设计架构有关，最基本的架构又有两种，冯诺依曼体系架构和哈佛结构体系架构。

#### 冯诺依曼体系架构

冯诺依曼体系架构，也被称为冯诺依曼计算机体系架构，是计算机体系结构的一种基本范式，它是现代计算机体系结构的基础。这个架构由匈牙利数学家约翰·冯·诺伊曼（John von Neumann）于20世纪40年代提出，它包括了一系列的设计原则和概念，主要特点如下：

* 存储器存储程序和数据：冯诺依曼体系架构的核心概念是将程序指令和数据存储在同一存储器中，这就意味着程序和数据都以二进制形式存储在内存中。这种方式使得计算机能够像数据一样处理程序，同时也可以通过修改存储器中的指令来改变程序的执行流程，从而实现了程序的存储和执行的灵活性。
* 顺序执行：冯诺依曼计算机按照程序中指令的顺序一步一步地执行，每次执行一条指令，然后继续执行下一条指令。这种顺序执行是计算机的基本执行模型。
* 存储程序概念：冯诺依曼体系中引入了存储程序的概念，即程序本身也存储在内存中，并且可以被计算机读取和修改。这使得计算机可以灵活地执行各种不同的程序，而不需要物理改变硬件。
* 单一总线结构：冯诺依曼计算机通常使用单一总线结构，将CPU、内存、输入输出设备等各个组件连接在一起。CPU通过总线与其他组件进行通信，从内存中读取指令和数据，执行指令后将结果写回内存。
* 存储器访问速度：冯诺依曼计算机的性能受到存储器访问速度的限制，因为指令和数据都存储在内存中，而内存的访问速度相对较慢。为了解决这个问题，现代计算机通常使用高速缓存（Cache）来提高存储器访问速度。

冯诺依曼体系架构奠定了现代计算机的基础，几乎所有的通用计算机都采用了这一架构。它的优势在于它的通用性和灵活性，允许计算机执行各种不同的任务，只需改变存储在内存中的程序即可。因此，冯诺依曼计算机架构被认为是计算机科学和计算机工程领域的重要里程碑之一。

#### 哈佛结构体系架构

哈佛结构体系架构，也称为哈佛架构，是一种计算机体系结构，与冯诺依曼体系结构相对立。哈佛结构体系架构具有以下主要特点：

* 分离的存储器：
  + 在哈佛结构中，程序指令和数据存储在不同的存储器中。通常有一个用于存储指令的指令存储器（Instruction Memory）和一个用于存储数据的数据存储器（Data Memory）。
  + 这种分离的存储器架构允许并行地访问指令和数据，从而提高了计算机的性能，尤其在需要频繁的指令和数据访问时。
* 并行性：
  + 由于指令和数据存储在不同的存储器中，哈佛结构计算机通常具有更好的并行性能。它可以同时从指令存储器读取指令并从数据存储器读取数据，而不会发生冲突。
  + 这有助于加速计算机的执行速度，特别是在需要高吞吐量的应用中。
* 更快的指令获取速度：
  + 哈佛结构使得指令存储器的访问速度通常比数据存储器更快。这是因为指令存储器可以被优化为更快的访问，以确保CPU能够高效地获取和执行指令。
* 程序加载：
  + 哈佛结构允许程序直接从指令存储器中执行，无需像冯诺依曼结构那样首先将程序加载到内存中。这可以加速程序的启动和执行。
* 地址空间分离：
  + 指令存储器和数据存储器通常具有不同的地址空间，因此不需要特殊的地址区分机制，程序和数据的地址可以独立管理。

哈佛结构体系架构的主要优势在于它提供了更好的并行性和更快的指令获取速度，适用于需要高性能和高吞吐量的应用，例如嵌入式系统、信号处理等。然而，它也可能增加了复杂性和成本，因为需要额外的硬件来管理分离的存储器。冯诺依曼结构体系架构在通用计算环境中更为常见，因为它更简单且灵活，适用于各种应用。

#### CPU 对于二进制指令的依赖

前面介绍了 CPU 的两种结构形式，冯诺依曼结构和哈佛结构。从两种 CPU 对二进制指令的需求分析来看，CPU 的二进制指令应该是顺序的，紧凑的，数据也应该是顺序的，紧凑的。但同时，由于指令对数据的依赖形式，两者之间是分离的。基于这种需求，可执行文件的基本结构应该是指令和数据分部分排列的。

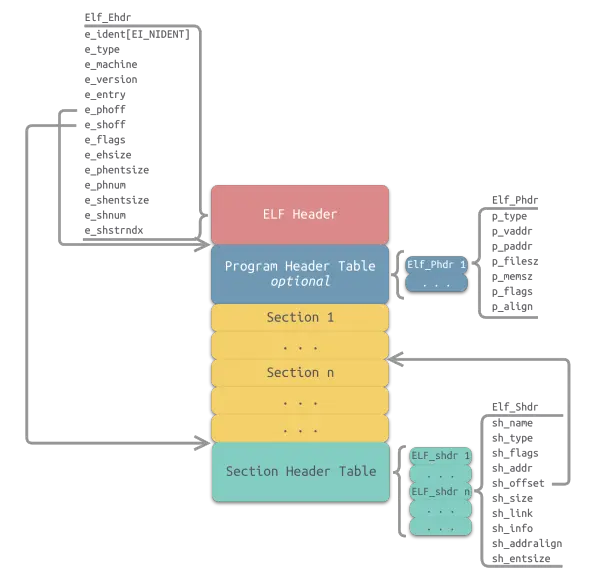
### ELF 文件格式分析

在编程过程中，根据数据在源代码中的作用，又可以分为局部变量，全局变量，有初始值的变量，不变的常量。在组织形式中，引入段（Sections）来描述，根据二进制数据的不同作用被放在不同段中。所以 ELF 文件主要以段的方式来组织二进制可执行文件。

Sections 有“段”和“节“的意思，所以用无论用”段”来描述 Sections 还是用“节”来描述 Sections 都是一个意思。一般称 Sections 为段。

ELF 的文件主要有八部分:

* **文件头部（ELF Header）：** ELF文件的开头包含一个固定长度的头部，包含了文件的基本信息，如文件类型、目标体系结构、入口点地址、程序头表和节头表的偏移等。这个头部是ELF文件的标识和入口点。
* **程序头表（Program Header Table）：** 程序头表描述了文件中各个段（segment）的信息，包括段的类型（加载、动态链接、TLS等）、在文件中的偏移、在内存中的虚拟地址、段的大小、权限（可读、可写、可执行等）等。程序头表用于加载和执行文件。
* **节头表（Section Header Table）：** 节头表描述了文件中各个节（section）的信息，包括节的名称、类型（代码、数据、符号表等）、在文件中的偏移、大小、在内存中的虚拟地址、与其他节的关系等。节头表对于链接器、调试器和其他工具非常重要。
* **节（Sections）：** ELF文件中的数据和代码通常被组织成多个节。常见的节包括.text（包含可执行代码）、.data（包含初始化的全局和静态数据）、.bss（包含未初始化的全局和静态数据）、.rodata（包含只读数据）等。每个节都有一个相关的节头表条目。
* **符号表（Symbol Table）：** ELF文件中包含一个符号表，用于存储全局符号和局部符号的信息。符号表包含了符号的名称、值（地址）、大小、绑定属性（全局、局部、弱等）和类型信息。符号表对于动态链接和调试非常重要。
* **字符串表（String Table）：** ELF文件中通常包含一个字符串表，用于存储符号表和节名字节的字符串。字符串表使得符号表和节头表中的字符串可以通过偏移来引用。
* **动态链接信息（Dynamic Linking Information）：** 可执行文件和共享库中可以包含动态链接信息，其中包括动态链接器的名称、共享库的需求和依赖关系等信息。这些信息允许操作系统在运行时解析和加载共享库。
* **重定位信息（Relocation Information）：** 如果文件需要在加载时进行地址重定位（例如，当共享库被加载时），则文件中可能包含重定位信息，用于指示需要修正的地址引用。



使用 readelf 能够查看到 elf 文件中的信息。

// file: elf\_example.c  
// compile cmd: gcc elf\_example.c -o elf\_example -g  
#include <stdio.h>  
int a;  
int b = 1;  
int main()  
{  
 int c;  
 c = 10;  
 const char \*str = "Hello";  
 printf("%s, World! \na:%d b:%d c:%d", str, a, b, c);  
 return 0;  
}

nihao@nihao-z690:~/work/test/gdb/test$ readelf -a elf\_example  
ELF 头：  
 Magic： 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00   
 类别: ELF64  
 数据: 2 补码，小端序 (little endian)  
 Version: 1 (current)  
 OS/ABI: UNIX - System V  
 ABI 版本: 0  
 类型: DYN (Position-Independent Executable file)  
 系统架构: Advanced Micro Devices X86-64  
 版本: 0x1  
 入口点地址： 0x1060  
 程序头起点： 64 (bytes into file)  
 Start of section headers: 14944 (bytes into file)  
 标志： 0x0  
 Size of this header: 64 (bytes)  
 Size of program headers: 56 (bytes)  
 Number of program headers: 13  
 Size of section headers: 64 (bytes)  
 Number of section headers: 37  
 Section header string table index: 36  
  
节头：  
 [号] 名称 类型 地址 偏移量  
 大小 全体大小 旗标 链接 信息 对齐  
 [ 0] NULL 0000000000000000 00000000  
 0000000000000000 0000000000000000 0 0 0  
 [ 1] .interp PROGBITS 0000000000000318 00000318  
 000000000000001c 0000000000000000 A 0 0 1  
 [ 2] .note.gnu.pr[...] NOTE 0000000000000338 00000338  
 0000000000000030 0000000000000000 A 0 0 8  
 [ 3] .note.gnu.bu[...] NOTE 0000000000000368 00000368  
 0000000000000024 0000000000000000 A 0 0 4  
 [ 4] .note.ABI-tag NOTE 000000000000038c 0000038c  
 0000000000000020 0000000000000000 A 0 0 4  
 [ 5] .gnu.hash GNU\_HASH 00000000000003b0 000003b0  
 0000000000000024 0000000000000000 A 6 0 8  
 [ 6] .dynsym DYNSYM 00000000000003d8 000003d8  
 00000000000000a8 0000000000000018 A 7 1 8  
 [ 7] .dynstr STRTAB 0000000000000480 00000480  
 000000000000008f 0000000000000000 A 0 0 1  
 [ 8] .gnu.version VERSYM 0000000000000510 00000510  
 000000000000000e 0000000000000002 A 6 0 2  
 [ 9] .gnu.version\_r VERNEED 0000000000000520 00000520  
 0000000000000030 0000000000000000 A 7 1 8  
 [10] .rela.dyn RELA 0000000000000550 00000550  
 00000000000000c0 0000000000000018 A 6 0 8  
 [11] .rela.plt RELA 0000000000000610 00000610  
 0000000000000018 0000000000000018 AI 6 24 8  
 [12] .init PROGBITS 0000000000001000 00001000  
 000000000000001b 0000000000000000 AX 0 0 4  
 [13] .plt PROGBITS 0000000000001020 00001020  
 0000000000000020 0000000000000010 AX 0 0 16  
 [14] .plt.got PROGBITS 0000000000001040 00001040  
 0000000000000010 0000000000000010 AX 0 0 16  
 [15] .plt.sec PROGBITS 0000000000001050 00001050  
 0000000000000010 0000000000000010 AX 0 0 16  
 [16] .text PROGBITS 0000000000001060 00001060  
 000000000000013b 0000000000000000 AX 0 0 16  
 [17] .fini PROGBITS 000000000000119c 0000119c  
 000000000000000d 0000000000000000 AX 0 0 4  
 [18] .rodata PROGBITS 0000000000002000 00002000  
 0000000000000025 0000000000000000 A 0 0 4  
 [19] .eh\_frame\_hdr PROGBITS 0000000000002028 00002028  
 0000000000000034 0000000000000000 A 0 0 4  
 [20] .eh\_frame PROGBITS 0000000000002060 00002060  
 00000000000000ac 0000000000000000 A 0 0 8  
 [21] .init\_array INIT\_ARRAY 0000000000003db8 00002db8  
 0000000000000008 0000000000000008 WA 0 0 8  
 [22] .fini\_array FINI\_ARRAY 0000000000003dc0 00002dc0  
 0000000000000008 0000000000000008 WA 0 0 8  
 [23] .dynamic DYNAMIC 0000000000003dc8 00002dc8  
 00000000000001f0 0000000000000010 WA 7 0 8  
 [24] .got PROGBITS 0000000000003fb8 00002fb8  
 0000000000000048 0000000000000008 WA 0 0 8  
 [25] .data PROGBITS 0000000000004000 00003000  
 0000000000000014 0000000000000000 WA 0 0 8  
 [26] .bss NOBITS 0000000000004014 00003014  
 000000000000000c 0000000000000000 WA 0 0 4  
 [27] .comment PROGBITS 0000000000000000 00003014  
 000000000000002b 0000000000000001 MS 0 0 1  
 [28] .debug\_aranges PROGBITS 0000000000000000 0000303f  
 0000000000000030 0000000000000000 0 0 1  
 [29] .debug\_info PROGBITS 0000000000000000 0000306f  
 00000000000000ee 0000000000000000 0 0 1  
 [30] .debug\_abbrev PROGBITS 0000000000000000 0000315d  
 000000000000009d 0000000000000000 0 0 1  
 [31] .debug\_line PROGBITS 0000000000000000 000031fa  
 0000000000000063 0000000000000000 0 0 1  
 [32] .debug\_str PROGBITS 0000000000000000 0000325d  
 00000000000000e0 0000000000000001 MS 0 0 1  
 [33] .debug\_line\_str PROGBITS 0000000000000000 0000333d  
 0000000000000042 0000000000000001 MS 0 0 1  
 [34] .symtab SYMTAB 0000000000000000 00003380  
 0000000000000390 0000000000000018 35 18 8  
 [35] .strtab STRTAB 0000000000000000 00003710  
 00000000000001e5 0000000000000000 0 0 1  
 [36] .shstrtab STRTAB 0000000000000000 000038f5  
 000000000000016a 0000000000000000 0 0 1  
Key to Flags:  
 W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
 L (link order), O (extra OS processing required), G (group), T (TLS),  
 C (compressed), x (unknown), o (OS specific), E (exclude),  
 D (mbind), l (large), p (processor specific)  
  
There are no section groups in this file.  
  
程序头：  
 Type Offset VirtAddr PhysAddr  
 FileSiz MemSiz Flags Align  
 PHDR 0x0000000000000040 0x0000000000000040 0x0000000000000040  
 0x00000000000002d8 0x00000000000002d8 R 0x8  
 INTERP 0x0000000000000318 0x0000000000000318 0x0000000000000318  
 0x000000000000001c 0x000000000000001c R 0x1  
 [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]  
 LOAD 0x0000000000000000 0x0000000000000000 0x0000000000000000  
 0x0000000000000628 0x0000000000000628 R 0x1000  
 LOAD 0x0000000000001000 0x0000000000001000 0x0000000000001000  
 0x00000000000001a9 0x00000000000001a9 R E 0x1000  
 LOAD 0x0000000000002000 0x0000000000002000 0x0000000000002000  
 0x000000000000010c 0x000000000000010c R 0x1000  
 LOAD 0x0000000000002db8 0x0000000000003db8 0x0000000000003db8  
 0x000000000000025c 0x0000000000000268 RW 0x1000  
 DYNAMIC 0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8  
 0x00000000000001f0 0x00000000000001f0 RW 0x8  
 NOTE 0x0000000000000338 0x0000000000000338 0x0000000000000338  
 0x0000000000000030 0x0000000000000030 R 0x8  
 NOTE 0x0000000000000368 0x0000000000000368 0x0000000000000368  
 0x0000000000000044 0x0000000000000044 R 0x4  
 GNU\_PROPERTY 0x0000000000000338 0x0000000000000338 0x0000000000000338  
 0x0000000000000030 0x0000000000000030 R 0x8  
 GNU\_EH\_FRAME 0x0000000000002028 0x0000000000002028 0x0000000000002028  
 0x0000000000000034 0x0000000000000034 R 0x4  
 GNU\_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000  
 0x0000000000000000 0x0000000000000000 RW 0x10  
 GNU\_RELRO 0x0000000000002db8 0x0000000000003db8 0x0000000000003db8  
 0x0000000000000248 0x0000000000000248 R 0x1  
  
 Section to Segment mapping:  
 段节...  
 00   
 01 .interp   
 02 .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version\_r .rela.dyn .rela.plt   
 03 .init .plt .plt.got .plt.sec .text .fini   
 04 .rodata .eh\_frame\_hdr .eh\_frame   
 05 .init\_array .fini\_array .dynamic .got .data .bss   
 06 .dynamic   
 07 .note.gnu.property   
 08 .note.gnu.build-id .note.ABI-tag   
 09 .note.gnu.property   
 10 .eh\_frame\_hdr   
 11   
 12 .init\_array .fini\_array .dynamic .got   
  
Dynamic section at offset 0x2dc8 contains 27 entries:  
 标记 类型 名称/值  
 0x0000000000000001 (NEEDED) 共享库：[libc.so.6]  
 0x000000000000000c (INIT) 0x1000  
 0x000000000000000d (FINI) 0x119c  
 0x0000000000000019 (INIT\_ARRAY) 0x3db8  
 0x000000000000001b (INIT\_ARRAYSZ) 8 (bytes)  
 0x000000000000001a (FINI\_ARRAY) 0x3dc0  
 0x000000000000001c (FINI\_ARRAYSZ) 8 (bytes)  
 0x000000006ffffef5 (GNU\_HASH) 0x3b0  
 0x0000000000000005 (STRTAB) 0x480  
 0x0000000000000006 (SYMTAB) 0x3d8  
 0x000000000000000a (STRSZ) 143 (bytes)  
 0x000000000000000b (SYMENT) 24 (bytes)  
 0x0000000000000015 (DEBUG) 0x0  
 0x0000000000000003 (PLTGOT) 0x3fb8  
 0x0000000000000002 (PLTRELSZ) 24 (bytes)  
 0x0000000000000014 (PLTREL) RELA  
 0x0000000000000017 (JMPREL) 0x610  
 0x0000000000000007 (RELA) 0x550  
 0x0000000000000008 (RELASZ) 192 (bytes)  
 0x0000000000000009 (RELAENT) 24 (bytes)  
 0x000000000000001e (FLAGS) BIND\_NOW  
 0x000000006ffffffb (FLAGS\_1) 标志： NOW PIE  
 0x000000006ffffffe (VERNEED) 0x520  
 0x000000006fffffff (VERNEEDNUM) 1  
 0x000000006ffffff0 (VERSYM) 0x510  
 0x000000006ffffff9 (RELACOUNT) 3  
 0x0000000000000000 (NULL) 0x0  
  
重定位节 '.rela.dyn' at offset 0x550 contains 8 entries:  
 偏移量 信息 类型 符号值 符号名称 + 加数  
000000003db8 000000000008 R\_X86\_64\_RELATIVE 1140  
000000003dc0 000000000008 R\_X86\_64\_RELATIVE 1100  
000000004008 000000000008 R\_X86\_64\_RELATIVE 4008  
000000003fd8 000100000006 R\_X86\_64\_GLOB\_DAT 0000000000000000 \_\_libc\_start\_main@GLIBC\_2.34 + 0  
000000003fe0 000200000006 R\_X86\_64\_GLOB\_DAT 0000000000000000 \_ITM\_deregisterTM[...] + 0  
000000003fe8 000400000006 R\_X86\_64\_GLOB\_DAT 0000000000000000 \_\_gmon\_start\_\_ + 0  
000000003ff0 000500000006 R\_X86\_64\_GLOB\_DAT 0000000000000000 \_ITM\_registerTMCl[...] + 0  
000000003ff8 000600000006 R\_X86\_64\_GLOB\_DAT 0000000000000000 \_\_cxa\_finalize@GLIBC\_2.2.5 + 0  
  
重定位节 '.rela.plt' at offset 0x610 contains 1 entry:  
 偏移量 信息 类型 符号值 符号名称 + 加数  
000000003fd0 000300000007 R\_X86\_64\_JUMP\_SLO 0000000000000000 printf@GLIBC\_2.2.5 + 0  
No processor specific unwind information to decode  
  
Symbol table '.dynsym' contains 7 entries:  
 Num: Value Size Type Bind Vis Ndx Name  
 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND   
 1: 0000000000000000 0 FUNC GLOBAL DEFAULT UND \_[...]@GLIBC\_2.34 (2)  
 2: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterT[...]  
 3: 0000000000000000 0 FUNC GLOBAL DEFAULT UND [...]@GLIBC\_2.2.5 (3)  
 4: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_  
 5: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_registerTMC[...]  
 6: 0000000000000000 0 FUNC WEAK DEFAULT UND [...]@GLIBC\_2.2.5 (3)  
  
Symbol table '.symtab' contains 38 entries:  
 Num: Value Size Type Bind Vis Ndx Name  
 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND   
 1: 0000000000000000 0 FILE LOCAL DEFAULT ABS Scrt1.o  
 2: 000000000000038c 32 OBJECT LOCAL DEFAULT 4 \_\_abi\_tag  
 3: 0000000000000000 0 FILE LOCAL DEFAULT ABS crtstuff.c  
 4: 0000000000001090 0 FUNC LOCAL DEFAULT 16 deregister\_tm\_clones  
 5: 00000000000010c0 0 FUNC LOCAL DEFAULT 16 register\_tm\_clones  
 6: 0000000000001100 0 FUNC LOCAL DEFAULT 16 \_\_do\_global\_dtors\_aux  
 7: 0000000000004014 1 OBJECT LOCAL DEFAULT 26 completed.0  
 8: 0000000000003dc0 0 OBJECT LOCAL DEFAULT 22 \_\_do\_global\_dtor[...]  
 9: 0000000000001140 0 FUNC LOCAL DEFAULT 16 frame\_dummy  
 10: 0000000000003db8 0 OBJECT LOCAL DEFAULT 21 \_\_frame\_dummy\_in[...]  
 11: 0000000000000000 0 FILE LOCAL DEFAULT ABS elf\_example.c  
 12: 0000000000000000 0 FILE LOCAL DEFAULT ABS crtstuff.c  
 13: 0000000000002108 0 OBJECT LOCAL DEFAULT 20 \_\_FRAME\_END\_\_  
 14: 0000000000000000 0 FILE LOCAL DEFAULT ABS   
 15: 0000000000003dc8 0 OBJECT LOCAL DEFAULT 23 \_DYNAMIC  
 16: 0000000000002028 0 NOTYPE LOCAL DEFAULT 19 \_\_GNU\_EH\_FRAME\_HDR  
 17: 0000000000003fb8 0 OBJECT LOCAL DEFAULT 24 \_GLOBAL\_OFFSET\_TABLE\_  
 18: 0000000000000000 0 FUNC GLOBAL DEFAULT UND \_\_libc\_start\_mai[...]  
 19: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_deregisterT[...]  
 20: 0000000000004000 0 NOTYPE WEAK DEFAULT 25 data\_start  
 21: 0000000000004010 4 OBJECT GLOBAL DEFAULT 25 b  
 22: 0000000000004014 0 NOTYPE GLOBAL DEFAULT 25 \_edata  
 23: 000000000000119c 0 FUNC GLOBAL HIDDEN 17 \_fini  
 24: 0000000000000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC\_2.2.5  
 25: 0000000000004000 0 NOTYPE GLOBAL DEFAULT 25 \_\_data\_start  
 26: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_\_gmon\_start\_\_  
 27: 0000000000004008 0 OBJECT GLOBAL HIDDEN 25 \_\_dso\_handle  
 28: 0000000000002000 4 OBJECT GLOBAL DEFAULT 18 \_IO\_stdin\_used  
 29: 0000000000004020 0 NOTYPE GLOBAL DEFAULT 26 \_end  
 30: 0000000000001060 38 FUNC GLOBAL DEFAULT 16 \_start  
 31: 0000000000004018 4 OBJECT GLOBAL DEFAULT 26 a  
 32: 0000000000004014 0 NOTYPE GLOBAL DEFAULT 26 \_\_bss\_start  
 33: 0000000000001149 82 FUNC GLOBAL DEFAULT 16 main  
 34: 0000000000004018 0 OBJECT GLOBAL HIDDEN 25 \_\_TMC\_END\_\_  
 35: 0000000000000000 0 NOTYPE WEAK DEFAULT UND \_ITM\_registerTMC[...]  
 36: 0000000000000000 0 FUNC WEAK DEFAULT UND \_\_cxa\_finalize@G[...]  
 37: 0000000000001000 0 FUNC GLOBAL HIDDEN 12 \_init  
  
Histogram for `.gnu.hash' bucket list length (total of 2 buckets):  
 Length Number % of total Coverage  
 0 1 ( 50.0%)  
 1 1 ( 50.0%) 100.0%  
  
Version symbols section '.gnu.version' contains 7 entries:  
 地址：0x0000000000000510 Offset: 0x000510 Link: 6 (.dynsym)  
 000: 0 (\*本地\*) 2 (GLIBC\_2.34) 1 (\*全局\*) 3 (GLIBC\_2.2.5)  
 004: 1 (\*全局\*) 1 (\*全局\*) 3 (GLIBC\_2.2.5)  
  
Version needs section '.gnu.version\_r' contains 1 entry:  
 地址：0x0000000000000520 Offset: 0x000520 Link: 7 (.dynstr)  
 000000: Version: 1 文件：libc.so.6 计数：2  
 0x0010: Name: GLIBC\_2.2.5 标志：无 版本：3  
 0x0020: Name: GLIBC\_2.34 标志：无 版本：2  
  
Displaying notes found in: .note.gnu.property  
 所有者 Data size Description  
 GNU 0x00000020 NT\_GNU\_PROPERTY\_TYPE\_0  
 Properties: x86 feature: IBT, SHSTK  
 x86 ISA needed: x86-64-baseline  
  
Displaying notes found in: .note.gnu.build-id  
 所有者 Data size Description  
 GNU 0x00000014 NT\_GNU\_BUILD\_ID (unique build ID bitstring)  
 Build ID: e9cd436268169a676222190cb0efbd1a74a72363  
  
Displaying notes found in: .note.ABI-tag  
 所有者 Data size Description  
 GNU 0x00000010 NT\_GNU\_ABI\_TAG (ABI version tag)  
 OS: Linux, ABI: 3.2.0

ELF 文件源码描述：

//在这个include/linux/elf.h文件中，根据系统使用的框架结构来决定使用 32 位还是 64 位的elf header结构。  
#if ELF\_CLASS == ELFCLASS32  
 extern Elf32\_Dyn \_DYNAMIC [];  
 #define elfhdr elf32\_hdr  
 #define elf\_phdr elf32\_phdr  
 #define elf\_shdr elf32\_shdr  
 #define elf\_note elf32\_note  
 #define elf\_addr\_t Elf32\_Off  
 #define Elf\_Half Elf32\_Half  
#else  
 extern Elf64\_Dyn \_DYNAMIC [];  
 #define elfhdr elf64\_hdr  
 #define elf\_phdr elf64\_phdr  
 #define elf\_shdr elf64\_shdr  
 #define elf\_note elf64\_note  
 #define elf\_addr\_t Elf64\_Off  
 #define Elf\_Half Elf64\_Half  
#endif  
  
//详细描述如下：  
typedef struct elf32\_hdr{  
 unsigned char e\_ident[EI\_NIDENT]; // 16 个字节信息  
 Elf32\_Half e\_type; // 目标文件类型  
 Elf32\_Half e\_machine; // 目标机器类型、体系结构  
 Elf32\_Word e\_version; // 目标文件版本  
 Elf32\_Addr e\_entry; // 程序入口的虚拟地址  
 Elf32\_Off e\_phoff; // 程序头部表偏移  
 Elf32\_Off e\_shoff; // 节区头部表偏移  
 Elf32\_Word e\_flags; // 处理器标记  
 Elf32\_Half e\_ehsize; // elf\_hdr 的大小  
 Elf32\_Half e\_phentsize; // 程序头部表（段头部表）一个表项的大小  
 Elf32\_Half e\_phnum; // 程序头部表表项数量  
 Elf32\_Half e\_shentsize; // 节区头部表一个表项的大小  
 Elf32\_Half e\_shnum; // 节区头部表表项数量  
 Elf32\_Half e\_shstrndx; // 节名字表的表项在节区头部表中的索引  
} Elf32\_Ehdr;  
  
/\* ELF "magic number" \*/ // ELF 文件头部识别区  
#define ELFMAG0 0x7f /\* EI\_MAG \*/  
#define ELFMAG1 'E'  
#define ELFMAG2 'L'  
#define ELFMAG3 'F'  
#define ELFMAG "\177ELF"  
#define SELFMAG 4

**Section**

段是对根据二进制在 CPU 运行时的作用划分的，指令，数据，和函数符等都会被归类并集中存放在特定的内存段中，方便 CPU 执行时读取。

程序段有默认的定义，用户也可以通过链接脚本自行定义。下面是是默认的定义解释。

* .text : 可执行代码段，该段内保存的是 CPU 的二进制代码，由 CPU 识别并执行。
* .data: 具有初始值的全局变量，执行时会被加载到内存到指定区域内。
* .rodata： 只读区域，通常在 C 语言中经过 const 修饰的全局常量会被放在这里。
* .bss: 没有初始值的全局变量，在 Linux 加载后会被置 0。
* .symtab: 符号表，记录了函数和变量名和对应的地址。在链接时被需要。
* .strtab：字符串表，代码中的字符串会被统一收集并存放到这里。
* .init, .fini：程序初始化代码段和结束代码段，在 C++ 静态类构建代码会被放在这里，在类构建时被初始化代码自动调用。

其中 ELF 文件中段内容可以使用 objdump 工具进行查看：

nihao@nihao-z690:~/work/test/gdb/test$ objdump -s elf\_example  
  
elf\_example： 文件格式 elf64-x86-64  
  
Contents of section .interp:  
 0318 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-  
 0328 7838362d 36342e73 6f2e3200 x86-64.so.2.   
Contents of section .note.gnu.property:  
 0338 04000000 20000000 05000000 474e5500 .... .......GNU.  
 0348 020000c0 04000000 03000000 00000000 ................  
 0358 028000c0 04000000 01000000 00000000 ................  
Contents of section .note.gnu.build-id:  
 0368 04000000 14000000 03000000 474e5500 ............GNU.  
 0378 e9cd4362 68169a67 6222190c b0efbd1a ..Cbh..gb"......  
 0388 74a72363 t.#c   
Contents of section .note.ABI-tag:  
 038c 04000000 10000000 01000000 474e5500 ............GNU.  
 039c 00000000 03000000 02000000 00000000 ................  
Contents of section .gnu.hash:  
 03b0 02000000 06000000 01000000 06000000 ................  
 03c0 00008100 00000000 06000000 00000000 ................  
 03d0 d165ce6d .e.m   
Contents of section .dynsym:  
 03d8 00000000 00000000 00000000 00000000 ................  
 03e8 00000000 00000000 10000000 12000000 ................  
 03f8 00000000 00000000 00000000 00000000 ................  
 0408 4a000000 20000000 00000000 00000000 J... ...........  
 0418 00000000 00000000 22000000 12000000 ........".......  
 0428 00000000 00000000 00000000 00000000 ................  
 0438 66000000 20000000 00000000 00000000 f... ...........  
 0448 00000000 00000000 75000000 20000000 ........u... ...  
 0458 00000000 00000000 00000000 00000000 ................  
 0468 01000000 22000000 00000000 00000000 ...."...........  
 0478 00000000 00000000 ........   
Contents of section .dynstr:  
 0480 005f5f63 78615f66 696e616c 697a6500 .\_\_cxa\_finalize.  
 0490 5f5f6c69 62635f73 74617274 5f6d6169 \_\_libc\_start\_mai  
 04a0 6e007072 696e7466 006c6962 632e736f n.printf.libc.so  
 04b0 2e360047 4c494243 5f322e32 2e350047 .6.GLIBC\_2.2.5.G  
 04c0 4c494243 5f322e33 34005f49 544d5f64 LIBC\_2.34.\_ITM\_d  
 04d0 65726567 69737465 72544d43 6c6f6e65 eregisterTMClone  
 04e0 5461626c 65005f5f 676d6f6e 5f737461 Table.\_\_gmon\_sta  
 04f0 72745f5f 005f4954 4d5f7265 67697374 rt\_\_.\_ITM\_regist  
 0500 6572544d 436c6f6e 65546162 6c6500 erTMCloneTable.   
Contents of section .gnu.version:  
 0510 00000200 01000300 01000100 0300 ..............   
Contents of section .gnu.version\_r:  
 0520 01000200 29000000 10000000 00000000 ....)...........  
 0530 751a6909 00000300 33000000 10000000 u.i.....3.......  
 0540 b4919606 00000200 3f000000 00000000 ........?.......  
Contents of section .rela.dyn:  
 0550 b83d0000 00000000 08000000 00000000 .=..............  
 0560 40110000 00000000 c03d0000 00000000 @........=......  
 0570 08000000 00000000 00110000 00000000 ................  
 0580 08400000 00000000 08000000 00000000 .@..............  
 0590 08400000 00000000 d83f0000 00000000 .@.......?......  
 05a0 06000000 01000000 00000000 00000000 ................  
 05b0 e03f0000 00000000 06000000 02000000 .?..............  
 05c0 00000000 00000000 e83f0000 00000000 .........?......  
 05d0 06000000 04000000 00000000 00000000 ................  
 05e0 f03f0000 00000000 06000000 05000000 .?..............  
 05f0 00000000 00000000 f83f0000 00000000 .........?......  
 0600 06000000 06000000 00000000 00000000 ................  
Contents of section .rela.plt:  
 0610 d03f0000 00000000 07000000 03000000 .?..............  
 0620 00000000 00000000 ........   
Contents of section .init:  
 1000 f30f1efa 4883ec08 488b05d9 2f000048 ....H...H.../..H  
 1010 85c07402 ffd04883 c408c3 ..t...H....   
Contents of section .plt:  
 1020 ff359a2f 0000f2ff 259b2f00 000f1f00 .5./....%./.....  
 1030 f30f1efa 68000000 00f2e9e1 ffffff90 ....h...........  
Contents of section .plt.got:  
 1040 f30f1efa f2ff25ad 2f00000f 1f440000 ......%./....D..  
Contents of section .plt.sec:  
 1050 f30f1efa f2ff2575 2f00000f 1f440000 ......%u/....D..  
Contents of section .text:  
 1060 f30f1efa 31ed4989 d15e4889 e24883e4 ....1.I..^H..H..  
 1070 f0505445 31c031c9 488d3dca 000000ff .PTE1.1.H.=.....  
 1080 15532f00 00f4662e 0f1f8400 00000000 .S/...f.........  
 1090 488d3d81 2f000048 8d057a2f 00004839 H.=./..H..z/..H9  
 10a0 f8741548 8b05362f 00004885 c07409ff .t.H..6/..H..t..  
 10b0 e00f1f80 00000000 c30f1f80 00000000 ................  
 10c0 488d3d51 2f000048 8d354a2f 00004829 H.=Q/..H.5J/..H)  
 10d0 fe4889f0 48c1ee3f 48c1f803 4801c648 .H..H..?H...H..H  
 10e0 d1fe7414 488b0505 2f000048 85c07408 ..t.H.../..H..t.  
 10f0 ffe0660f 1f440000 c30f1f80 00000000 ..f..D..........  
 1100 f30f1efa 803d092f 00000075 2b554883 .....=./...u+UH.  
 1110 3de22e00 00004889 e5740c48 8b3de62e =.....H..t.H.=..  
 1120 0000e819 ffffffe8 64ffffff c605e12e ........d.......  
 1130 0000015d c30f1f00 c30f1f80 00000000 ...]............  
 1140 f30f1efa e977ffff fff30f1e fa554889 .....w.......UH.  
 1150 e54883ec 10c745f4 0a000000 488d05a1 .H....E.....H...  
 1160 0e000048 8945f88b 0da32e00 008b15a5 ...H.E..........  
 1170 2e00008b 75f4488b 45f84189 f04889c6 ....u.H.E.A..H..  
 1180 488d0583 0e000048 89c7b800 000000e8 H......H........  
 1190 bcfeffff b8000000 00c9c3 ...........   
Contents of section .fini:  
 119c f30f1efa 4883ec08 4883c408 c3 ....H...H....   
Contents of section .rodata:  
 2000 01000200 48656c6c 6f002573 2c20576f ....Hello.%s, Wo  
 2010 726c6421 200a613a 25642062 3a256420 rld! .a:%d b:%d   
 2020 633a2564 00 c:%d.   
Contents of section .eh\_frame\_hdr:  
 2028 011b033b 34000000 05000000 f8efffff ...;4...........  
 2038 68000000 18f0ffff 90000000 28f0ffff h...........(...  
 2048 a8000000 38f0ffff 50000000 21f1ffff ....8...P...!...  
 2058 c0000000 ....   
Contents of section .eh\_frame:  
 2060 14000000 00000000 017a5200 01781001 .........zR..x..  
 2070 1b0c0708 90010000 14000000 1c000000 ................  
 2080 e0efffff 26000000 00440710 00000000 ....&....D......  
 2090 24000000 34000000 88efffff 20000000 $...4....... ...  
 20a0 000e1046 0e184a0f 0b770880 003f1a3a ...F..J..w...?.:  
 20b0 2a332422 00000000 14000000 5c000000 \*3$"........\...  
 20c0 80efffff 10000000 00000000 00000000 ................  
 20d0 14000000 74000000 78efffff 10000000 ....t...x.......  
 20e0 00000000 00000000 1c000000 8c000000 ................  
 20f0 59f0ffff 52000000 00450e10 8602430d Y...R....E....C.  
 2100 0602490c 07080000 00000000 ..I.........   
Contents of section .init\_array:  
 3db8 40110000 00000000 @.......   
Contents of section .fini\_array:  
 3dc0 00110000 00000000 ........   
Contents of section .dynamic:  
 3dc8 01000000 00000000 29000000 00000000 ........).......  
 3dd8 0c000000 00000000 00100000 00000000 ................  
 3de8 0d000000 00000000 9c110000 00000000 ................  
 3df8 19000000 00000000 b83d0000 00000000 .........=......  
 3e08 1b000000 00000000 08000000 00000000 ................  
 3e18 1a000000 00000000 c03d0000 00000000 .........=......  
 3e28 1c000000 00000000 08000000 00000000 ................  
 3e38 f5feff6f 00000000 b0030000 00000000 ...o............  
 3e48 05000000 00000000 80040000 00000000 ................  
 3e58 06000000 00000000 d8030000 00000000 ................  
 3e68 0a000000 00000000 8f000000 00000000 ................  
 3e78 0b000000 00000000 18000000 00000000 ................  
 3e88 15000000 00000000 00000000 00000000 ................  
 3e98 03000000 00000000 b83f0000 00000000 .........?......  
 3ea8 02000000 00000000 18000000 00000000 ................  
 3eb8 14000000 00000000 07000000 00000000 ................  
 3ec8 17000000 00000000 10060000 00000000 ................  
 3ed8 07000000 00000000 50050000 00000000 ........P.......  
 3ee8 08000000 00000000 c0000000 00000000 ................  
 3ef8 09000000 00000000 18000000 00000000 ................  
 3f08 1e000000 00000000 08000000 00000000 ................  
 3f18 fbffff6f 00000000 01000008 00000000 ...o............  
 3f28 feffff6f 00000000 20050000 00000000 ...o.... .......  
 3f38 ffffff6f 00000000 01000000 00000000 ...o............  
 3f48 f0ffff6f 00000000 10050000 00000000 ...o............  
 3f58 f9ffff6f 00000000 03000000 00000000 ...o............  
 3f68 00000000 00000000 00000000 00000000 ................  
 3f78 00000000 00000000 00000000 00000000 ................  
 3f88 00000000 00000000 00000000 00000000 ................  
 3f98 00000000 00000000 00000000 00000000 ................  
 3fa8 00000000 00000000 00000000 00000000 ................  
Contents of section .got:  
 3fb8 c83d0000 00000000 00000000 00000000 .=..............  
 3fc8 00000000 00000000 30100000 00000000 ........0.......  
 3fd8 00000000 00000000 00000000 00000000 ................  
 3fe8 00000000 00000000 00000000 00000000 ................  
 3ff8 00000000 00000000 ........   
Contents of section .data:  
 4000 00000000 00000000 08400000 00000000 .........@......  
 4010 01000000 ....   
Contents of section .comment:  
 0000 4743433a 20285562 756e7475 2031312e GCC: (Ubuntu 11.  
 0010 342e302d 31756275 6e747531 7e32322e 4.0-1ubuntu1~22.  
 0020 30342920 31312e34 2e3000 04) 11.4.0.   
Contents of section .debug\_aranges:  
 0000 2c000000 02000000 00000800 00000000 ,...............  
 0010 49110000 00000000 52000000 00000000 I.......R.......  
 0020 00000000 00000000 00000000 00000000 ................  
Contents of section .debug\_info:  
 0000 ea000000 05000108 00000000 04130000 ................  
 0010 001d1f00 00000000 00004911 00000000 ..........I.....  
 0020 00005200 00000000 00000000 00000108 ..R.............  
 0030 07a10000 00010407 a6000000 010108b3 ................  
 0040 00000001 02070000 00000101 06b50000 ................  
 0050 00010205 d6000000 05040569 6e740001 ...........int..  
 0060 0805c600 00000101 06bc0000 00066600 ..............f.  
 0070 00000261 00025800 00000903 18400000 ...a..X......@..  
 0080 00000000 02620003 58000000 09031040 .....b..X......@  
 0090 00000000 000007cf 00000002 64010c58 ............d..X  
 00a0 000000ae 00000008 ae000000 09000a08 ................  
 00b0 6d000000 0bc10000 00010405 58000000 m...........X...  
 00c0 49110000 00000000 52000000 00000000 I.......R.......  
 00d0 019c0363 00060958 00000002 91640373 ...c...X.....d.s  
 00e0 74720008 11ae0000 00029168 0000 tr.........h..   
Contents of section .debug\_abbrev:  
 0000 0124000b 0b3e0b03 0e000002 34000308 .$...>......4...  
 0010 3a21013b 0b392105 49133f19 02180000 :!.;.9!.I.?.....  
 0020 03340003 083a2101 3b0b390b 49130218 .4...:!.;.9.I...  
 0030 00000411 01250e13 0b031f1b 1f110112 .....%..........  
 0040 07101700 00052400 0b0b3e0b 03080000 ......$...>.....  
 0050 06260049 13000007 2e013f19 030e3a0b .&.I......?...:.  
 0060 3b05390b 27194913 3c190113 00000805 ;.9.'.I.<.......  
 0070 00491300 00091800 00000a0f 000b0b49 .I.............I  
 0080 1300000b 2e013f19 030e3a0b 3b0b390b ......?...:.;.9.  
 0090 49131101 12074018 7c190000 00 I.....@.|....   
Contents of section .debug\_line:  
 0000 5f000000 05000800 33000000 010101fb \_.......3.......  
 0010 0e0d0001 01010100 00000100 00010101 ................  
 0020 1f020000 00002d00 00000201 1f020f03 ......-.........  
 0030 1f000000 001f0000 00003a00 00000105 ..........:.....  
 0040 01000902 49110000 00000000 160507bc ....I...........  
 0050 05117505 05ad050c 022d1305 01590202 ..u......-...Y..  
 0060 000101 ...   
Contents of section .debug\_str:  
 0000 73686f72 7420756e 7369676e 65642069 short unsigned i  
 0010 6e740047 4e552043 31372031 312e342e nt.GNU C17 11.4.  
 0020 30202d6d 74756e65 3d67656e 65726963 0 -mtune=generic  
 0030 202d6d61 7263683d 7838362d 3634202d -march=x86-64 -  
 0040 67202d66 6173796e 6368726f 6e6f7573 g -fasynchronous  
 0050 2d756e77 696e642d 7461626c 6573202d -unwind-tables -  
 0060 66737461 636b2d70 726f7465 63746f72 fstack-protector  
 0070 2d737472 6f6e6720 2d667374 61636b2d -strong -fstack-  
 0080 636c6173 682d7072 6f746563 74696f6e clash-protection  
 0090 202d6663 662d7072 6f746563 74696f6e -fcf-protection  
 00a0 006c6f6e 6720756e 7369676e 65642069 .long unsigned i  
 00b0 6e740075 6e736967 6e656420 63686172 nt.unsigned char  
 00c0 006d6169 6e006c6f 6e672069 6e740070 .main.long int.p  
 00d0 72696e74 66007368 6f727420 696e7400 rintf.short int.  
Contents of section .debug\_line\_str:  
 0000 2f686f6d 652f6e69 68616f2f 776f726b /home/nihao/work  
 0010 2f746573 742f6764 622f7465 73740065 /test/gdb/test.e  
 0020 6c665f65 78616d70 6c652e63 002f7573 lf\_example.c./us  
 0030 722f696e 636c7564 65007374 64696f2e r/include.stdio.  
 0040 6800 h.

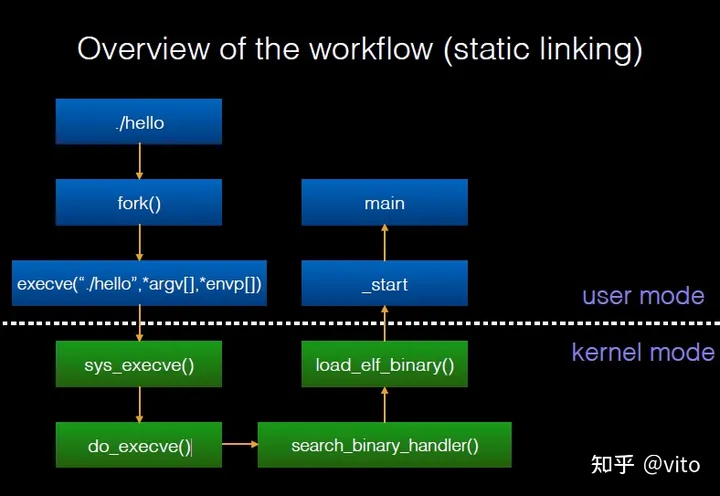
**loader 过程**

一般程序的手动执行是由 bash 来完成的，主要经过了 fork() 系统调用和 exec() 系统调用。

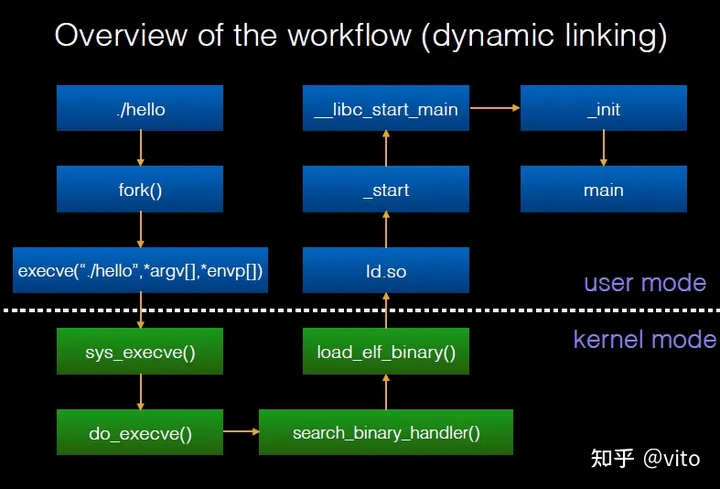
当 bash 收到用户的执行程序命令后，首先调用 fork() 通过复制一份自己来创建一个子程序，然后使用 exec() 系统调用，清空进程中所有的段，识别新程序 ELF ，根据 ELF 头部的描述信息，重新分配进程内存并将对应的段复制到内存中，最后将 PC 程序执行指针指向新的代码段中的首地址，新的程序便开始执行了。

当然，着中间需要一些准备工作。

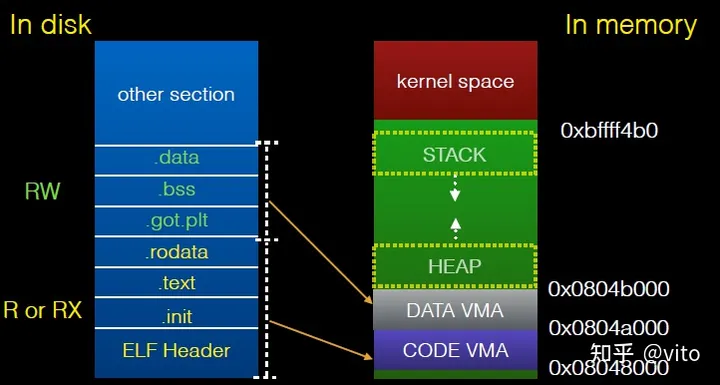
程序静态执行(没有链接动态库)：



程序动态执行(链接了动态库)：



ELF 文件执行时在内存中的映射：



<https://zhuanlan.zhihu.com/p/628432429?utm_id=0>

### binutils 工具集

binutils 是基本的编程工具集，它提供了一整套操作二进制 ELF 文件的工具，可用于生成，查看，操作，和加载 ELF 文件。GCC 在编译的最后两个阶段会调用 binutils 中的工具来完成最后的汇编编译和链接过程。binutils 中包含了 addr2line、nm、readelf、size、objcopy、objdump、strings、strip、ar、ld、ranlib。

测试 C 程序的源码：

// file: binutils\_test.c  
#include <stdio.h>  
  
int hello\_world()  
{  
 printf("func addr is:%p\n", hello\_world);  
 return 0;  
}  
  
int main(int argc,char \*argv[])  
{  
 hello\_world();  
 return 0;  
}

gcc binutils\_test.c -o binutils\_test -g

**nm**

打印出 ELF 文件中的符号，以及符号在内存中的地址。

nihao@nihao-z690:~/work/test/gdb/test$ nm -n binutils\_test   
 w \_\_cxa\_finalize@GLIBC\_2.2.5  
 w \_\_gmon\_start\_\_  
 w \_ITM\_deregisterTMCloneTable  
 w \_ITM\_registerTMCloneTable  
 U \_\_libc\_start\_main@GLIBC\_2.34  
 U printf@GLIBC\_2.2.5  
000000000000038c r \_\_abi\_tag  
0000000000001000 T \_init  
0000000000001060 T \_start  
0000000000001090 t deregister\_tm\_clones  
00000000000010c0 t register\_tm\_clones  
0000000000001100 t \_\_do\_global\_dtors\_aux  
0000000000001140 t frame\_dummy  
0000000000001149 T hello\_world  
0000000000001176 T main  
000000000000119c T \_fini  
0000000000002000 R \_IO\_stdin\_used  
0000000000002018 r \_\_GNU\_EH\_FRAME\_HDR  
0000000000002120 r \_\_FRAME\_END\_\_  
0000000000003db8 d \_\_frame\_dummy\_init\_array\_entry  
0000000000003dc0 d \_\_do\_global\_dtors\_aux\_fini\_array\_entry  
0000000000003dc8 d \_DYNAMIC  
0000000000003fb8 d \_GLOBAL\_OFFSET\_TABLE\_  
0000000000004000 D \_\_data\_start  
0000000000004000 W data\_start  
0000000000004008 D \_\_dso\_handle  
0000000000004010 B \_\_bss\_start  
0000000000004010 b completed.0  
0000000000004010 D \_edata  
0000000000004010 D \_\_TMC\_END\_\_  
0000000000004018 B \_end  
nihao@nihao-z690:~/work/test/gdb/test$ nm --format=sysv binutils\_test   
  
  
来自 binutils\_test 的符号：  
  
Name Value Class Type Size Line Section  
  
\_\_abi\_tag |000000000000038c| r | OBJECT|0000000000000020| |.note.ABI-tag  
\_\_bss\_start |0000000000004010| B | NOTYPE| | |.bss  
completed.0 |0000000000004010| b | OBJECT|0000000000000001| |.bss  
\_\_cxa\_finalize@GLIBC\_2.2.5| | w | FUNC| | |\*UND\*  
\_\_data\_start |0000000000004000| D | NOTYPE| | |.data  
data\_start |0000000000004000| W | NOTYPE| | |.data  
deregister\_tm\_clones|0000000000001090| t | FUNC| | |.text  
\_\_do\_global\_dtors\_aux|0000000000001100| t | FUNC| | |.text  
\_\_do\_global\_dtors\_aux\_fini\_array\_entry|0000000000003dc0| d | OBJECT| | |.fini\_array  
\_\_dso\_handle |0000000000004008| D | OBJECT| | |.data  
\_DYNAMIC |0000000000003dc8| d | OBJECT| | |.dynamic  
\_edata |0000000000004010| D | NOTYPE| | |.data  
\_end |0000000000004018| B | NOTYPE| | |.bss  
\_fini |000000000000119c| T | FUNC| | |.fini  
frame\_dummy |0000000000001140| t | FUNC| | |.text  
\_\_frame\_dummy\_init\_array\_entry|0000000000003db8| d | OBJECT| | |.init\_array  
\_\_FRAME\_END\_\_ |0000000000002120| r | OBJECT| | |.eh\_frame  
\_GLOBAL\_OFFSET\_TABLE\_|0000000000003fb8| d | OBJECT| | |.got  
\_\_gmon\_start\_\_ | | w | NOTYPE| | |\*UND\*  
\_\_GNU\_EH\_FRAME\_HDR |0000000000002018| r | NOTYPE| | |.eh\_frame\_hdr  
hello\_world |0000000000001149| T | FUNC|000000000000002d| |.text  
\_init |0000000000001000| T | FUNC| | |.init  
\_IO\_stdin\_used |0000000000002000| R | OBJECT|0000000000000004| |.rodata  
\_ITM\_deregisterTMCloneTable| | w | NOTYPE| | |\*UND\*  
\_ITM\_registerTMCloneTable| | w | NOTYPE| | |\*UND\*  
\_\_libc\_start\_main@GLIBC\_2.34| | U | FUNC| | |\*UND\*  
main |0000000000001176| T | FUNC|0000000000000024| |.text  
printf@GLIBC\_2.2.5 | | U | FUNC| | |\*UND\*  
register\_tm\_clones |00000000000010c0| t | FUNC| | |.text  
\_start |0000000000001060| T | FUNC|0000000000000026| |.text  
\_\_TMC\_END\_\_ |0000000000004010| D | OBJECT| | |.data

第一列是符号的地址，第二列是符号存在的段简名，第三列是符号名称。

第二列的段简名：

|  |  |
| --- | --- |
| 名称 | 含义 |
| A | 表示符号所对应的地址是绝对地址，在链接和执行过程中都不会变。 |
| B 或 b | 表示符号位于 .bss 段中。 |
| C | 表示没有被初始化的公共符号。 |
| D 或 d | 表示符号位于.data 段中。 |
| N | 表示符号是调试用的。 |
| P | 表示符号位于一个栈回溯段中。 |
| R 或r | 表示符号位于 .rdata 段中。 |
| T 或 t | 表示符号位于 .text 段中。 |
| U | 表示符号没有被定义。 |

**addr2line**

打印地址对应的符号信息。

使用演示：

nihao@nihao-z690:~/work/test/gdb/test$ nm -n binutils\_test | grep hello\_world  
0000000000001149 T hello\_world  
nihao@nihao-z690:~/work/test/gdb/test$ addr2line 0x0000000000001149 -f -e binutils\_test   
hello\_world  
/home/nihao/work/test/gdb/test/addr2line\_test.c:5

该工具原本是用于查看程序运行出错函数的地址对应的符号信息，新的操作系统起用了地址空间随机化（ASLR）安全特性。ASLR 会在每次运行程序时随机分配函数的地址，以增加系统的安全性，使得恶意攻击更加困难。但也让 addr2line 工具接近失效，但是在调试嵌入式设备时，addr2line 还是很有用的。

**objdump**

用来显示目标文件的信息，可以通过选项控制显示那些特定信息，objdump 还具有反汇编功能。

nihao@nihao-z690:~/work/test/gdb/test$ objdump -d addr2line\_test  
  
addr2line\_test： 文件格式 elf64-x86-64  
  
  
Disassembly of section .init:  
  
0000000000001000 <\_init>:  
 1000: f3 0f 1e fa endbr64   
 1004: 48 83 ec 08 sub $0x8,%rsp  
 1008: 48 8b 05 d9 2f 00 00 mov 0x2fd9(%rip),%rax # 3fe8 <\_\_gmon\_start\_\_@Base>  
 100f: 48 85 c0 test %rax,%rax  
 1012: 74 02 je 1016 <\_init+0x16>  
 1014: ff d0 call \*%rax  
 1016: 48 83 c4 08 add $0x8,%rsp  
 101a: c3 ret   
  
Disassembly of section .plt:  
  
0000000000001020 <.plt>:  
 1020: ff 35 9a 2f 00 00 push 0x2f9a(%rip) # 3fc0 <\_GLOBAL\_OFFSET\_TABLE\_+0x8>  
 1026: f2 ff 25 9b 2f 00 00 bnd jmp \*0x2f9b(%rip) # 3fc8 <\_GLOBAL\_OFFSET\_TABLE\_+0x10>  
 102d: 0f 1f 00 nopl (%rax)  
 1030: f3 0f 1e fa endbr64   
 1034: 68 00 00 00 00 push $0x0  
 1039: f2 e9 e1 ff ff ff bnd jmp 1020 <\_init+0x20>  
 103f: 90 nop  
  
Disassembly of section .plt.got:  
  
0000000000001040 <\_\_cxa\_finalize@plt>:  
 1040: f3 0f 1e fa endbr64   
 1044: f2 ff 25 ad 2f 00 00 bnd jmp \*0x2fad(%rip) # 3ff8 <\_\_cxa\_finalize@GLIBC\_2.2.5>  
 104b: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)  
  
Disassembly of section .plt.sec:  
  
0000000000001050 <printf@plt>:  
 1050: f3 0f 1e fa endbr64   
 1054: f2 ff 25 75 2f 00 00 bnd jmp \*0x2f75(%rip) # 3fd0 <printf@GLIBC\_2.2.5>  
 105b: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)  
  
Disassembly of section .text:  
  
0000000000001060 <\_start>:  
 1060: f3 0f 1e fa endbr64   
 1064: 31 ed xor %ebp,%ebp  
 1066: 49 89 d1 mov %rdx,%r9  
 1069: 5e pop %rsi  
 106a: 48 89 e2 mov %rsp,%rdx  
 106d: 48 83 e4 f0 and $0xfffffffffffffff0,%rsp  
 1071: 50 push %rax  
 1072: 54 push %rsp  
 1073: 45 31 c0 xor %r8d,%r8d  
 1076: 31 c9 xor %ecx,%ecx  
 1078: 48 8d 3d f7 00 00 00 lea 0xf7(%rip),%rdi # 1176 <main>  
 107f: ff 15 53 2f 00 00 call \*0x2f53(%rip) # 3fd8 <\_\_libc\_start\_main@GLIBC\_2.34>  
 1085: f4 hlt   
 1086: 66 2e 0f 1f 84 00 00 cs nopw 0x0(%rax,%rax,1)  
 108d: 00 00 00   
  
0000000000001090 <deregister\_tm\_clones>:  
 1090: 48 8d 3d 79 2f 00 00 lea 0x2f79(%rip),%rdi # 4010 <\_\_TMC\_END\_\_>  
 1097: 48 8d 05 72 2f 00 00 lea 0x2f72(%rip),%rax # 4010 <\_\_TMC\_END\_\_>  
 109e: 48 39 f8 cmp %rdi,%rax  
 10a1: 74 15 je 10b8 <deregister\_tm\_clones+0x28>  
 10a3: 48 8b 05 36 2f 00 00 mov 0x2f36(%rip),%rax # 3fe0 <\_ITM\_deregisterTMCloneTable@Base>  
 10aa: 48 85 c0 test %rax,%rax  
 10ad: 74 09 je 10b8 <deregister\_tm\_clones+0x28>  
 10af: ff e0 jmp \*%rax  
 10b1: 0f 1f 80 00 00 00 00 nopl 0x0(%rax)  
 10b8: c3 ret   
 10b9: 0f 1f 80 00 00 00 00 nopl 0x0(%rax)  
  
00000000000010c0 <register\_tm\_clones>:  
 10c0: 48 8d 3d 49 2f 00 00 lea 0x2f49(%rip),%rdi # 4010 <\_\_TMC\_END\_\_>  
 10c7: 48 8d 35 42 2f 00 00 lea 0x2f42(%rip),%rsi # 4010 <\_\_TMC\_END\_\_>  
 10ce: 48 29 fe sub %rdi,%rsi  
 10d1: 48 89 f0 mov %rsi,%rax  
 10d4: 48 c1 ee 3f shr $0x3f,%rsi  
 10d8: 48 c1 f8 03 sar $0x3,%rax  
 10dc: 48 01 c6 add %rax,%rsi  
 10df: 48 d1 fe sar %rsi  
 10e2: 74 14 je 10f8 <register\_tm\_clones+0x38>  
 10e4: 48 8b 05 05 2f 00 00 mov 0x2f05(%rip),%rax # 3ff0 <\_ITM\_registerTMCloneTable@Base>  
 10eb: 48 85 c0 test %rax,%rax  
 10ee: 74 08 je 10f8 <register\_tm\_clones+0x38>  
 10f0: ff e0 jmp \*%rax  
 10f2: 66 0f 1f 44 00 00 nopw 0x0(%rax,%rax,1)  
 10f8: c3 ret   
 10f9: 0f 1f 80 00 00 00 00 nopl 0x0(%rax)  
  
0000000000001100 <\_\_do\_global\_dtors\_aux>:  
 1100: f3 0f 1e fa endbr64   
 1104: 80 3d 05 2f 00 00 00 cmpb $0x0,0x2f05(%rip) # 4010 <\_\_TMC\_END\_\_>  
 110b: 75 2b jne 1138 <\_\_do\_global\_dtors\_aux+0x38>  
 110d: 55 push %rbp  
 110e: 48 83 3d e2 2e 00 00 cmpq $0x0,0x2ee2(%rip) # 3ff8 <\_\_cxa\_finalize@GLIBC\_2.2.5>  
 1115: 00   
 1116: 48 89 e5 mov %rsp,%rbp  
 1119: 74 0c je 1127 <\_\_do\_global\_dtors\_aux+0x27>  
 111b: 48 8b 3d e6 2e 00 00 mov 0x2ee6(%rip),%rdi # 4008 <\_\_dso\_handle>  
 1122: e8 19 ff ff ff call 1040 <\_\_cxa\_finalize@plt>  
 1127: e8 64 ff ff ff call 1090 <deregister\_tm\_clones>  
 112c: c6 05 dd 2e 00 00 01 movb $0x1,0x2edd(%rip) # 4010 <\_\_TMC\_END\_\_>  
 1133: 5d pop %rbp  
 1134: c3 ret   
 1135: 0f 1f 00 nopl (%rax)  
 1138: c3 ret   
 1139: 0f 1f 80 00 00 00 00 nopl 0x0(%rax)  
  
0000000000001140 <frame\_dummy>:  
 1140: f3 0f 1e fa endbr64   
 1144: e9 77 ff ff ff jmp 10c0 <register\_tm\_clones>  
  
0000000000001149 <hello\_world>:  
 1149: f3 0f 1e fa endbr64   
 114d: 55 push %rbp  
 114e: 48 89 e5 mov %rsp,%rbp  
 1151: 48 8d 05 f1 ff ff ff lea -0xf(%rip),%rax # 1149 <hello\_world>  
 1158: 48 89 c6 mov %rax,%rsi  
 115b: 48 8d 05 a2 0e 00 00 lea 0xea2(%rip),%rax # 2004 <\_IO\_stdin\_used+0x4>  
 1162: 48 89 c7 mov %rax,%rdi  
 1165: b8 00 00 00 00 mov $0x0,%eax  
 116a: e8 e1 fe ff ff call 1050 <printf@plt>  
 116f: b8 00 00 00 00 mov $0x0,%eax  
 1174: 5d pop %rbp  
 1175: c3 ret   
  
0000000000001176 <main>:  
 1176: f3 0f 1e fa endbr64   
 117a: 55 push %rbp  
 117b: 48 89 e5 mov %rsp,%rbp  
 117e: 48 83 ec 10 sub $0x10,%rsp  
 1182: 89 7d fc mov %edi,-0x4(%rbp)  
 1185: 48 89 75 f0 mov %rsi,-0x10(%rbp)  
 1189: b8 00 00 00 00 mov $0x0,%eax  
 118e: e8 b6 ff ff ff call 1149 <hello\_world>  
 1193: 48 8d 05 dc ff ff ff lea -0x24(%rip),%rax # 1176 <main>  
 119a: 48 89 c6 mov %rax,%rsi  
 119d: 48 8d 05 71 0e 00 00 lea 0xe71(%rip),%rax # 2015 <\_IO\_stdin\_used+0x15>  
 11a4: 48 89 c7 mov %rax,%rdi  
 11a7: b8 00 00 00 00 mov $0x0,%eax  
 11ac: e8 9f fe ff ff call 1050 <printf@plt>  
 11b1: b8 00 00 00 00 mov $0x0,%eax  
 11b6: c9 leave   
 11b7: c3 ret   
  
Disassembly of section .fini:  
  
00000000000011b8 <\_fini>:  
 11b8: f3 0f 1e fa endbr64   
 11bc: 48 83 ec 08 sub $0x8,%rsp  
 11c0: 48 83 c4 08 add $0x8,%rsp  
 11c4: c3 ret

**readelf**

用于分析 ELF（Executable and Linkable Format）文件的结构和内容。

# readelf -S -W  
-a 查看所有  
-h ELF头  
-l 程序表头  
-S 节头表  
-s 符号表  
  
nihao@nihao-z690:~/work/test/gdb/test$ readelf -S -W binutils\_test   
There are 37 section headers, starting at offset 0x3a88:  
  
节头：  
 [Nr] Name Type Address Off Size ES Flg Lk Inf Al  
 [ 0] NULL 0000000000000000 000000 000000 00 0 0 0  
 [ 1] .interp PROGBITS 0000000000000318 000318 00001c 00 A 0 0 1  
 [ 2] .note.gnu.property NOTE 0000000000000338 000338 000030 00 A 0 0 8  
 [ 3] .note.gnu.build-id NOTE 0000000000000368 000368 000024 00 A 0 0 4  
 [ 4] .note.ABI-tag NOTE 000000000000038c 00038c 000020 00 A 0 0 4  
 [ 5] .gnu.hash GNU\_HASH 00000000000003b0 0003b0 000024 00 A 6 0 8  
 [ 6] .dynsym DYNSYM 00000000000003d8 0003d8 0000a8 18 A 7 1 8  
 [ 7] .dynstr STRTAB 0000000000000480 000480 00008f 00 A 0 0 1  
 [ 8] .gnu.version VERSYM 0000000000000510 000510 00000e 02 A 6 0 2  
 [ 9] .gnu.version\_r VERNEED 0000000000000520 000520 000030 00 A 7 1 8  
 [10] .rela.dyn RELA 0000000000000550 000550 0000c0 18 A 6 0 8  
 [11] .rela.plt RELA 0000000000000610 000610 000018 18 AI 6 24 8  
 [12] .init PROGBITS 0000000000001000 001000 00001b 00 AX 0 0 4  
 [13] .plt PROGBITS 0000000000001020 001020 000020 10 AX 0 0 16  
 [14] .plt.got PROGBITS 0000000000001040 001040 000010 10 AX 0 0 16  
 [15] .plt.sec PROGBITS 0000000000001050 001050 000010 10 AX 0 0 16  
 [16] .text PROGBITS 0000000000001060 001060 00013a 00 AX 0 0 16  
 [17] .fini PROGBITS 000000000000119c 00119c 00000d 00 AX 0 0 4  
 [18] .rodata PROGBITS 0000000000002000 002000 000015 00 A 0 0 4  
 [19] .eh\_frame\_hdr PROGBITS 0000000000002018 002018 00003c 00 A 0 0 4  
 [20] .eh\_frame PROGBITS 0000000000002058 002058 0000cc 00 A 0 0 8  
 [21] .init\_array INIT\_ARRAY 0000000000003db8 002db8 000008 08 WA 0 0 8  
 [22] .fini\_array FINI\_ARRAY 0000000000003dc0 002dc0 000008 08 WA 0 0 8  
 [23] .dynamic DYNAMIC 0000000000003dc8 002dc8 0001f0 10 WA 7 0 8  
 [24] .got PROGBITS 0000000000003fb8 002fb8 000048 08 WA 0 0 8  
 [25] .data PROGBITS 0000000000004000 003000 000010 00 WA 0 0 8  
 [26] .bss NOBITS 0000000000004010 003010 000008 00 WA 0 0 1  
 [27] .comment PROGBITS 0000000000000000 003010 00002b 01 MS 0 0 1  
 [28] .debug\_aranges PROGBITS 0000000000000000 00303b 000030 00 0 0 1  
 [29] .debug\_info PROGBITS 0000000000000000 00306b 0000f5 00 0 0 1  
 [30] .debug\_abbrev PROGBITS 0000000000000000 003160 0000a7 00 0 0 1  
 [31] .debug\_line PROGBITS 0000000000000000 003207 000067 00 0 0 1  
 [32] .debug\_str PROGBITS 0000000000000000 00326e 0000f6 01 MS 0 0 1  
 [33] .debug\_line\_str PROGBITS 0000000000000000 003364 000045 01 MS 0 0 1  
 [34] .symtab SYMTAB 0000000000000000 0033b0 000378 18 35 18 8  
 [35] .strtab STRTAB 0000000000000000 003728 0001f2 00 0 0 1  
 [36] .shstrtab STRTAB 0000000000000000 00391a 00016a 00 0 0 1  
Key to Flags:  
 W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
 L (link order), O (extra OS processing required), G (group), T (TLS),  
 C (compressed), x (unknown), o (OS specific), E (exclude),  
 D (mbind), l (large), p (processor specific)

**strip**

用于剥离（或删除）可执行文件、目标文件和共享库中的调试信息和符号信息的命令行工具，可执行文件进行瘦身。

nihao@nihao-z690:~/work/test/gdb/test$ ls binutils\_test -lh  
-rwxrwxr-x 1 nihao nihao 17K 12月 25 12:11 binutils\_test  
nihao@nihao-z690:~/work/test/gdb/test$ strip binutils\_test   
nihao@nihao-z690:~/work/test/gdb/test$ ls binutils\_test -lh  
-rwxrwxr-x 1 nihao nihao 15K 12月 25 14:42 binutils\_test

**objcopy**

用于复制、转换和修改目标文件中的数据。objcopy 提供了许多功能，使您可以执行多种操作，包括创建裁剪或转换二进制文件、从一个目标文件中提取特定的部分、修改目标文件的格式以及生成新的目标文件等。

# --only-keep-debug Strip everything but the debug information  
# --strip-debug Remove all debugging symbols & sections  
# --add-gnu-debuglink=<file> Add section .gnu\_debuglink linking to <file>  
  
# 1、生成调试信息文件out.symbol  
objcopy --only-keep-debug out out.symbol  
  
# 2、生成不含调试信息的可执行文件out.strip  
objcopy --strip-debug out out.strip  
  
# 3、为不含调试信息的可执行文件,添加调试信息.   
objcopy --add-gnu-debuglink=out.symbol out.strip  
  
# 4、生成嵌入式二进制固件  
objcopy -I elf32-little -O binary my\_firmware.elf my\_firmware.bin

**ar**

用于建立、修改、提取静态库文件(archive)。archive是一个包含多个被包含文件的单一文件（也称之为库文件），其结构保证了可以从中检索并得到原始的被包含文件（称之为archive中的member）。member的原始文件内容、模式（权限）、时间戳、所有者和组等属性都被保存在 archive中。member被提取后，他们的属性被恢复到初始状态。ar 主要用于创建C库文件。典型用法如：$ ar rv libNAME.a file1.o file2.o

ar [-]p[mod] [--plugin name] [--target bfdname] [relpos] [count] archive [member...]  
ar -M [ <mri-script ]  
The gnu ar program creates, modifies, and extracts from archives.   
An archive is a single file holding a collection of other files in a structure that   
makes it possible to retrieve the original individual files (called members of the archive).   
  
#创建静态库：  
ar -rcs test.a \*.o  
#查看静态库：  
ar -tv test.a  
#解压静态库：  
ar -x test.a  
   
#查看程序依赖的动态库：  
readelf -a xxx|grep library  
#如：可以看到，下面的交叉程序hello执行依赖于如下两个动态库。  
rebi@ubuntu:~/test$ arm-none-linux-gnueabi-readelf -a hello|grep "library"  
 0x00000001 (NEEDED) Shared library: [libgcc\_s.so.1]  
 0x00000001 (NEEDED) Shared library: [libc.so.6]

**ld**

ld 是链接器，通常由GCC/G++调用。外部基本不可见。

**c++filt**

用于反混淆c++变量或函数名称的，可以还原为人类可读的格式。通常这个过程称为反混淆-demangle。

# 00000000000011a9 <\_Z11print\_hellov>:  
$ c++filt \_Z11print\_hellov  
print\_hello()

**size**

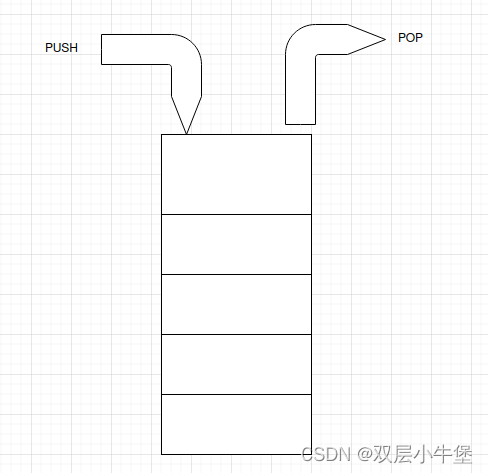
用于查看可执行文件、共享库和目标文件的节（section）大小和符号表信息。size 命令用于分析二进制文件的大小和内存占用，以帮助开发人员了解程序的结构和资源使用情况。

# 可以看到out的分布，对裁剪flash有参考价值；  
nihao@nihao-z690:~/work/test/gdb/test$ size binutils\_test   
 text data bss dec hex filename  
 1472 600 8 2080 820 binutils\_test

### 栈回溯原理

栈回溯在调试中是一个重要的概念，尤其是在调试程序崩溃或异常行为时。栈回溯（Stack Backtrace）或栈跟踪在调试过程中用于确定程序崩溃或异常行为发生时的函数调用序列。这种方法对于理解程序的执行流程和定位问题的根本原因非常有帮助。

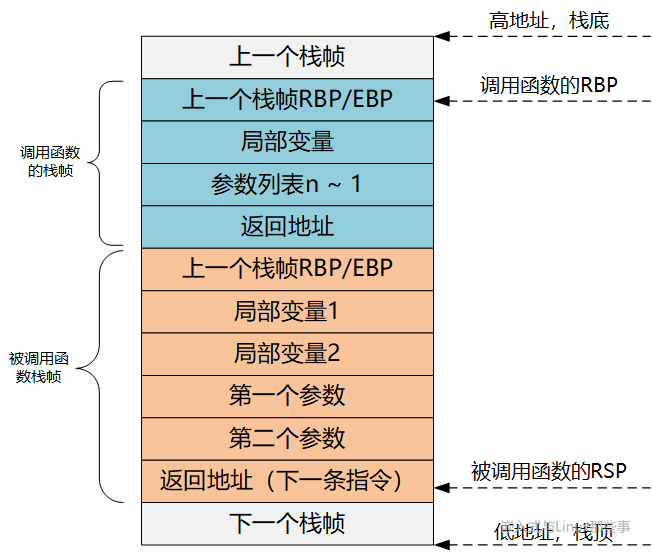
#### C 语言调用栈

C 语言的运行是基于栈的，栈是一种先入后出的内存模型。通过 PUSH 和 POP 指令，能够自动的向内存中写入数据和读出输出。  


C 语言的一个函数对应的就是内存中的一个栈帧，C 语言程序的主体就是由一个个的栈帧组成，栈帧的建立和消除的细节跟操作系统平台及编译器的实现有关，C 语言的标准并没有描述实现的方式，所以，不同的编译器，处理器，操作系统都可能有自己的建立栈帧的方式。但是栈帧的结构基本相似，下面的描述是针对运行在 Intel 奔腾芯片上 Linux 的 gcc 编译器而言。

int foo(int arg1, int arg2, int arg3)  
{  
 int a = 1;  
 int b = 2;   
 return 0;  
}

下图是 C 函数的典型栈帧：



在这个 foo 简化的函数栈帧中， foo 中有两个局部的 int 变量（4个字节），在 main 调用 foo。在这里， main 是调用者（caller），foo 是被调用者（callee）。ESP 只是栈顶寄存器，用于保存栈顶地址，EBP 栈指针寄存器，用于指示栈当前地址，相当于函数基准指针。从 main 传递到 foo 的参数以及 foo 本身的局部变量都可以通过这个基准指针为参考，加上偏移量找到。

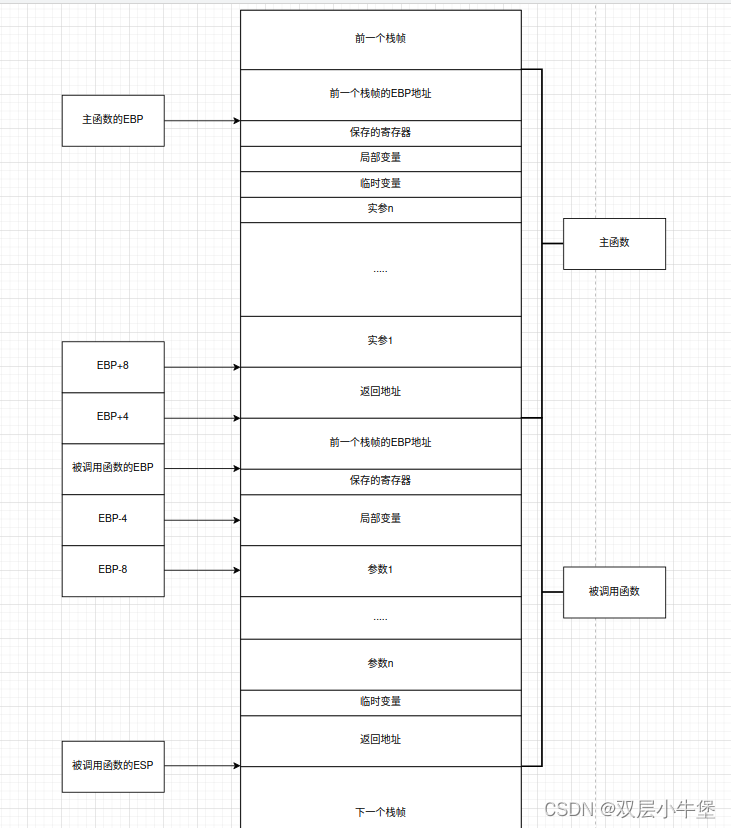
Intel CPU 中的 8 个通用寄存器：

|  |  |
| --- | --- |
| 寄存器 | 作用 |
| EAX | 累加器，用于算数计算，在 C 语言中用于存放返回地址。 |
| EBX | 基地址寄存器，用于间接寻址。 |
| ECX | 计数寄存器，通常用于监测程序的性能和指令执行速度等。 |
| EDX | 数据寄存器，被处理器用来读取或者写入特定数据的寄存器。 |
| EDI | 变址寄存器，用来存放字符串/内存操作的目的地址。 |
| ESI | 变址寄存器，用于存放字符串/内存操作的源地址。 |
| EBP | 栈内数据指针，栈帧的基地址，用于为函数调用创建栈帧，栈底。 |
| ESP | 栈指针，用于存入栈顶位置。 |

通常函数被调用时，只被允许使用 EAX，ECX 和 EDX 寄存器，这三个寄存器会先被调用者先保存在栈中。如果被调用者期望使用其他的寄存器，需要先将使用的寄存器保存在栈中，然后在使用完后，把寄存器的值从栈中恢复出来。

#### C 语言函数调用过程

main 函数和 foo 函数栈在内存中的排列如下：



被调者 foo 函数执行前调用者 main 的动作：

调用者先将 EAX， ECX， EDX 压栈保存，（实际中，这是可选的，取决于调用者是否需要这三个寄存器）。

如果有参数，按照参数顺序，将调用 foo 函数的参数一一进栈。

如果有返回值，将增加栈顶指针，预留返回值内存。

使用 call 指令调用函数，call 指令是一个组合功能，首先会将返回地址压进栈中，然后将栈底指针指压进栈中，再将栈底指针设置到栈顶指针的位置，调整 PC 代码指针指向子程序的首地址，下一步，子程序便开始执行了。

被调者 foo 函数执行动作：

保存自己用到的寄存器的值，也就是将寄存器中的值压栈。

调整栈顶指针，开辟局部变量内存。

开始执行函数中的语句。

被调者 foo 函数返回动作：

当函数执行完后，先恢复用到的寄存器的值，将寄存器的值出栈。

使用 ret 指令，设置返回值，恢复栈底指针，恢复栈顶指针，设置 PC 指针到返回地址，返回到调用者 main 函数。

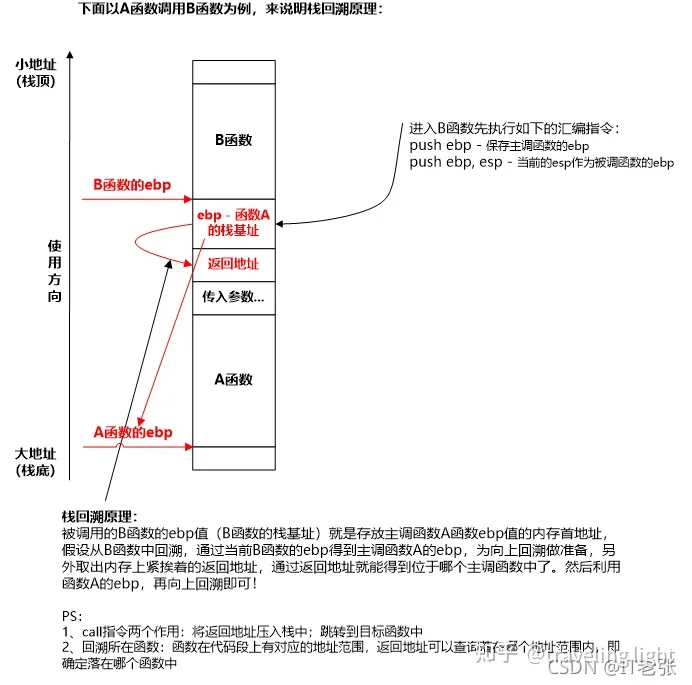
至此，C 语言的一个函数调用便完成了，C 语言程序就是在不断的函数调用中完成所用的代码执行。

这是C语言函数的大致调用过程，在实际的各种 CPU 平台中，会优先使用 CPU 寄存器作为内存传递参数，当参数超过寄存器数量后，才会使用栈内存。

int main() {  
 print\_hello();  
}  
//MOV AX，BX；将BX寄存器的16位数据传送到AX寄存器  
00000000000011a9 <\_Z11print\_hellov>:  
 11a9: f3 0f 1e fa endbr64  
 11ad: 55 push %rbp  
 11ae: 48 89 e5 mov %rsp,%rbp  
...  
 11da: 5d pop %rbp  
 11db: c3 retq  
00000000000011dc <main>:  
 11dc: f3 0f 1e fa endbr64  
 11e0: 55 push %rbp  
 11e1: 48 89 e5 mov %rsp,%rbp  
 11e4: e8 c0 ff ff ff callq 11a9 <\_Z11print\_hellov>  
 11e9: b8 00 00 00 00 mov $0x0,%eax  
 11ee: 5d pop %rbp  
 11ef: c3 retq

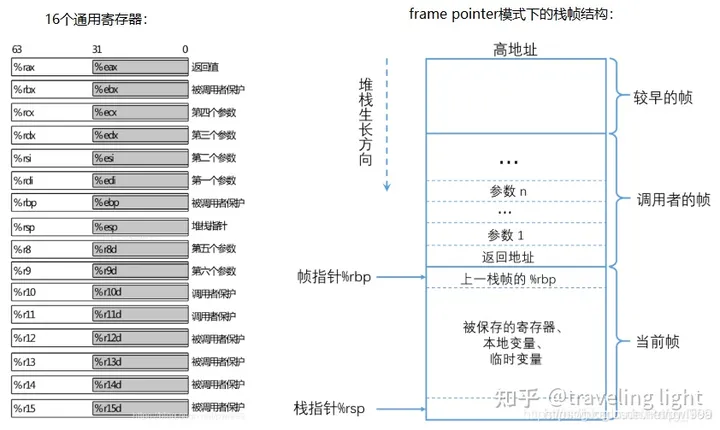
#### 栈回溯

C 语言的运行是基于栈的，栈回溯就是顺着 C 语言执行时前进的栈，逐步的后退并且打印出后退时的情况，让我们知道程序执行时发生了什么情况。在嵌入式系统C语言开发调试过程中，常会遇到各类异常情况。一般可按需添加打印信息，以便观察程序执行流或变量值是否异常。然而，打印操作会占用 CPU 时间，而且代码中添加过多打印信息时会显得很凌乱。此外，即使出错打印已非常详尽，但仍难以完全预防和处理段违例 (Segment Violation) 等错误。在没有外部调试器(如 gdb server )可用或无法现场调试的情况下，若程序能在突发崩溃时自动输出函数的调用堆栈信息(即堆栈回溯)，那么对于排错将会非常有用。

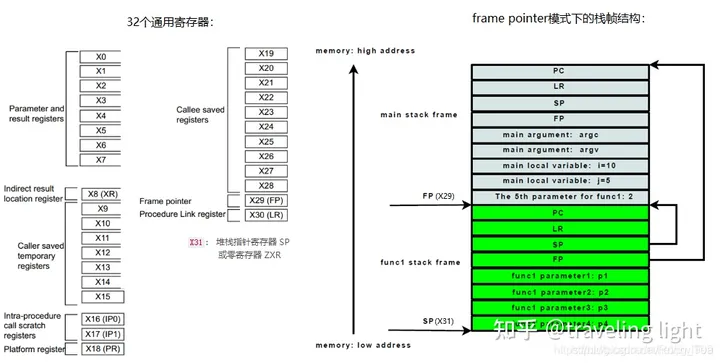


**frame pointer**

* 在调试的时候经常需要进行堆栈回溯。最简单的方式是使用一个独立的寄存器(ebp)来保存每层函数调用的堆栈栈顶(frame pointer)



x86\_64的frame point模式



arm64的frame point模式

这种方式在堆栈回溯时非常方便快捷。但是这种方法也有自己的不足：

* 需要一个专门寄存器ebp来保存frame poniter。
* 保存ebp寄存器即保存回溯信息(unwind info)的动作会被编译成代码，有指令开销。
* 在回溯堆栈时，除了恢复sp，不知道怎么恢复其他的寄存器。(例如gdb中的 frame n, info reg)
* 没有源语言信息。

**debug\_frame (DWARF)**

调试信息标准DWARF(Debugging With Attributed Record Formats)定义了一个.debug\_frame section用来解决上述的难题。

* 可以把ebp当成常规寄存器使用。
* 但是当保存esp时，它必须在.debug\_frame节中产生一条注释，告知调试器它将其保存在什么位置以及存放在何处。
* 这种机制还有的好处是它不仅仅是用来恢复ebp，还可以用来恢复其他寄存器。
* 而且是带外的，不消耗任何指令周期，没有任何性能开销。

这种机制也有其不足：

* 没有源语言信息。
* 不支持在程序加载时同时加载调试信息。

**eh\_frame (LSB)**

现代Linux操作系统在LSB(Linux Standard Base)标准中定义了一个.eh\_frame section来解决上述的难题。这个section和.debug\_frame非常类似，但是它解决了上述难题：

* 拥有源语言信息。
* 编码紧凑，并随程序一起加载。

但是.debug\_frame和.eh\_frame同时面临一个难题：怎么样生成堆栈信息表？

**CFI directives**

* 为了解决上述难题，GAS(GCC Assembler)汇编编译器定义了一组伪指令来协助生成调用栈信息CFI(Call Frame Information)。
* CFI directives伪指令是一组生成CFI调试信息的高级语言
* 关于汇编器利用这些伪指令来生成.debug\_frame还是.debug\_frame，在.cfi\_sections指令中定义。如果只是调试需求可以生成.debug\_frame，如果需要在运行时调用需要生成.eh\_frame。

**eh\_frame 基础**

* 使用gcc -g生成的DWARF信息存储在debug\_\*类型的section，
* 我们可以使用readelf -wi xxx查看debug\_info段，或者dwarfdump xxx查看debug信息。
* 无论是否有-g选项，gcc默认都会生成.eh\_frame和.eh\_frame\_hdr section。
* -fno-asynchronous-unwind-tables选项可以禁止生成.eh\_frame和.eh\_frame\_hdr section。

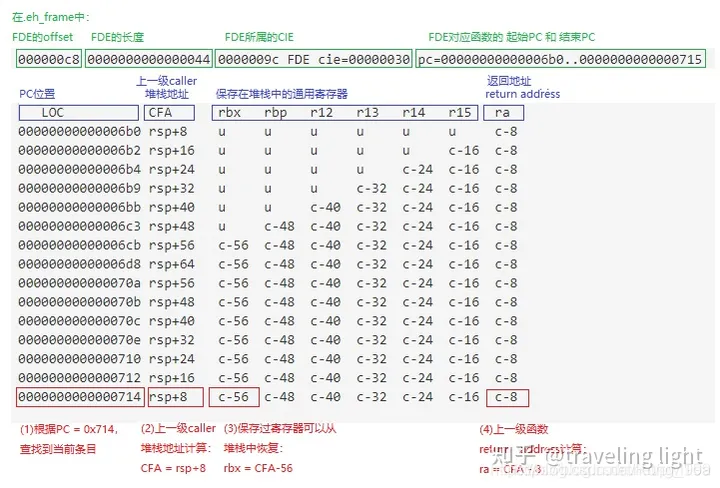
$ readelf -S -W out  
...  
 [19] .eh\_frame\_hdr PROGBITS 0000000000002014 002014 00005c 00 A 0 0 4  
 [20] .eh\_frame PROGBITS 0000000000002070 002070 000168 00 A 0 0 8  
...

* 在LSB(Linux Standard Base)中对.eh\_frame格式有详细的描述。
* .eh\_frame section 包含一个或者多个CFI(Call Frame Information)记录。每个CFI包含一个CIE(Common Information Entry Record)记录，每个CIE包含一个或者多个FDE(Frame Description Entry)记录。
* 通常情况下，CIE对应一个文件，FDE对应一个函数。

**eh\_frame 解析信息**

* 可以看到.eh\_frame总体架构就是由CIE和FDE组成的。
* 其中最核心的就是FDE的组织，读懂它条目的所有字段基本就理解了unwind的含义
* CFA (Canonical Frame Address, which is the address of %rsp in the caller frame)，CFA就是上一级调用者的堆栈指针。

$ readelf -wF a.out   
Contents of the .eh\_frame section:  
  
00000000 0000000000000014 00000000 CIE "zR" cf=1 df=-8 ra=16  
 LOC CFA ra   
0000000000000000 rsp+8 u   
...  
000000c8 0000000000000044 0000009c FDE cie=00000030 pc=00000000000006b0..0000000000000715  
 LOC CFA rbx rbp r12 r13 r14 r15 ra   
00000000000006b0 rsp+8 u u u u u u c-8   
00000000000006b2 rsp+16 u u u u u c-16 c-8   
00000000000006b4 rsp+24 u u u u c-24 c-16 c-8   
00000000000006b9 rsp+32 u u u c-32 c-24 c-16 c-8   
...   
0000000000000714 rsp+8 c-56 c-48 c-40 c-32 c-24 c-16 c-8



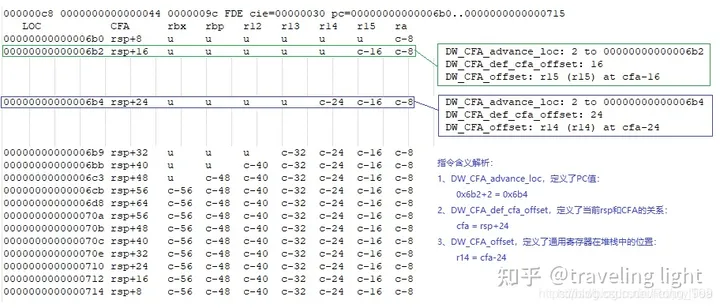
上图详细说明了怎么样利用.eh\_frame来进行栈回溯：

* 根据当前的PC在.eh\_frame中找到对应的条目，根据条目提供的各种偏移计算其他信息。
* 首先根据CFA = rsp+8，把当前rsp+8得到CFA的值。再根据CFA的值计算出通用寄存器和返回地址在堆栈中的位置。
* 通用寄存器栈位置计算。例如：rbx = CFA-56。
* 返回地址ra的栈位置计算。ra = CFA-8。
* 根据ra的值，重复步骤1到4，就形成了完整的栈回溯。

**eh\_frame 原始信息**

# 也可以使用readelf -wf xxx命令来查看elf文件中的.eh\_frame原始信息  
$ readelf -wf a.out   
...  
   
000000c8 0000000000000044 0000009c FDE cie=00000030 pc=00000000000006b0..0000000000000715  
 DW\_CFA\_advance\_loc: 2 to 00000000000006b2  
 DW\_CFA\_def\_cfa\_offset: 16  
 DW\_CFA\_offset: r15 (r15) at cfa-16  
...

* 使用readelf -wF xxx命令解析了这些信息，我们初步看看对应关系：



**eh\_frame 的加载**

* 内核vmlinux也是一个elf文件，它编译完成后默认也生成了.eh\_frame和.eh\_frame\_hdrsection。这两个段运行的时候被一起加载到内存，运行的时候需要有方法能找到它们。
* 内核在链接脚本vmlinux.lds.h中制定了定义：\_\_start\_unwind\_hdr和\_\_end\_unwind\_hdr变量用来标识.eh\_frame\_hdr的位置，\_\_start\_unwind和\_\_end\_unwind变量用来标识.eh\_frame的位置。
* 在系统启动时，把内核的.eh\_frame和.eh\_frame\_hdrsection当成一张table管理起来
* 驱动模块在加载的时候也需要把自己的.eh\_frame加进来，接口函数unwind\_add\_table()，这样就形成了一张unwind信息table链表
* 通常我们调用dump\_stack()来打印出内核的当前调用栈

//kernel\include\asm-generic\vmlinux.lds.h  
#ifdef CONFIG\_STACK\_UNWIND  
#define EH\_FRAME \  
 /\* Unwind data binary search table \*/ \  
 . = ALIGN(8); \  
 .eh\_frame\_hdr : AT(ADDR(.eh\_frame\_hdr) - LOAD\_OFFSET) { \  
 VMLINUX\_SYMBOL(\_\_start\_unwind\_hdr) = .; \  
 \*(.eh\_frame\_hdr) \  
 VMLINUX\_SYMBOL(\_\_end\_unwind\_hdr) = .; \  
 } \  
 /\* Unwind data \*/ \  
 . = ALIGN(8); \  
 .eh\_frame : AT(ADDR(.eh\_frame) - LOAD\_OFFSET) { \  
 VMLINUX\_SYMBOL(\_\_start\_unwind) = .; \  
 \*(.eh\_frame) \  
 VMLINUX\_SYMBOL(\_\_end\_unwind) = .; \  
 }  
#else  
#define EH\_FRAME  
#endif  
  
//start\_kernel()  
void \_\_init unwind\_init(void)  
{  
 init\_unwind\_table(&root\_table, "kernel",  
 \_text, \_end - \_text,  
 NULL, 0,  
 \_\_start\_unwind, \_\_end\_unwind - \_\_start\_unwind,  
 \_\_start\_unwind\_hdr, \_\_end\_unwind\_hdr - \_\_start\_unwind\_hdr);  
}

#### 栈回溯方式

* gcc的-g，应该没有人不知道它是一个调试选项，因此在一般需要进行程序调试的场景下，我们都会加上该选项，并且根据调试工具的不同，还能直接选择更有针对性的说明，比如-g gdb。-g是一个编译选项，即在源代码编译的过程中起作用，让gcc把更多调试信息（也就包括符号信息）收集起来并将存放到最终的可执行文件内。
* -rdynamic却是一个连接选项，它将指示连接器把所有符号（而不仅仅只是程序已使用到的外部符号，但不包括静态符号，比如被static修饰的函数）都添加到动态符号表（即.dynsym表）里，以便那些通过dlopen()或backtrace()（这一系列函数使用.dynsym表内符号）这样的函数使用。

##### 注册信号

* 关于信号处理signal()、sigaction()等的使用。 一个常见陷阱：**信号处理函数必须是可重入函数。如果信号处理函数不可重入，那么可能导致很多诡异问题**。

《UNIX环境高级编程》可重入函数章节中这样写道：  
但在信号处理程序中，不能判断捕捉到信号时进程在何处执行。如果进程正在执行malloc，在其堆中分配另外的存储空间，而此时由于捕捉到信号而插入执行该信号处理程序，其中又调用malloc，这时会发生什么？

* 关于可重入函数相信其概念并不难理解，但真正使用信号时，很多人都忽略了这一点，特别是一些比较隐晦的不可重入函数。在项目中可能遇到信号处理函数中调用不可重入函数导致的[死锁](https://link.zhihu.com/?target=https%3A//so.csdn.net/so/search%3Fq%3D%E6%AD%BB%E9%94%81%26spm%3D1001.2101.3001.7020)：某项目运行一段时间后，进程基本停止响应各种外界命令，日志也基本停止打印(只有个别简单轮询线程定时打印些信息)，但ps命令看到进程还在运行。看到这个问题，第一反应就是进程死锁，gdb attach到进程上，查看各个线程的堆栈，果然, 很多线程都卡在malloc调用上.

#include <signal.h>  
  
//忽略部分信息信号  
static void signalIgnore(void)  
{  
 (void)signal(SIGPIPE, SIG\_IGN);   
 (void)signal(SIGTTIN, SIG\_IGN);  
 (void)signal(SIGTTOU, SIG\_IGN);  
}  
  
//回调处理  
static void stop\_proc(int SIGNO, siginfo\_t \*info, void \*ctx)  
{  
 do\_backtrace();  
 ...  
}  
  
//拦截信号处理  
static void signalCap(void)  
{  
 struct sigaction sa;  
 sa.sa\_sigaction = stop\_proc;  
 sa.sa\_flags = SA\_SIGINFO;  
 sigemptyset(&sa.sa\_mask);  
   
 sigaction(SIGINT, &sa, nullptr);  
 sigaction(SIGSEGV, &sa, nullptr);  
 sigaction(SIGTERM, &sa, nullptr);  
 sigaction(SIGBUS, &sa, nullptr);   
}  
int main()  
{  
 signalIgnore();  
 signalCap();  
 ...  
 return 0;  
}  
  
//SIGINT 2 /\* Interrupt (ANSI). \*/ 终止进程 中断进程 Ctrl+C  
//SIGBUS 7 /\* BUS error (4.2 BSD). \*/ 建立CORE文件,总线错误  
//SIGFPE 8 /\* Floating-point exception (ANSI). \*/ 建立CORE文件,浮点异常  
//SIGSEGV 11 /\* Segmentation violation (ANSI). \*/ 建立CORE文件,段非法错误  
//SIGPIPE 13 /\* Broken pipe (POSIX). \*/ 终止进程 向一个没有读进程的管道写数据  
//SIGTERM 15 /\* Termination (ANSI). \*/ 终止进程 软件终止信号  
//SIGTTIN 21 /\* Background read from tty (POSIX). \*/ 停止进程 后台进程读终端  
//SIGTTOU 22 /\* Background write to tty (POSIX). \*/ 停止进程 后台进程写终端

##### gcc

//gcc提供了\_\_builtin\_return\_address() 宏来做栈的回溯  
void do\_backtrace()  
{  
 void \*pc0 = \_\_builtin\_return\_address(1);  
 void \*pc1 = \_\_builtin\_frame\_address(0);  
 ...  
}

##### glibc

//glibc提供了一对函数backtrace()和backtrace\_symbols()来回溯栈信息：  
//backtrace函数是通过读取操作系统的一个全局信息区，在多线程并发调用时，会造成严重的锁冲突。  
#include <execinfo.h>  
   
#define BACKTRACE\_SIZ 20  
void do\_backtrace()  
{  
 void \*array[BACKTRACE\_SIZ];  
 size\_t size;  
 char \*\*strings = nullptr;  
   
 size = backtrace(array, BACKTRACE\_SIZ);  
  
 //推荐直接写文件：backtrace\_symbols\_fd(array, size, 2); (dladdr)  
 strings = backtrace\_symbols(array, size);  
   
 for (size\_t i = 0; i < size; i++) {  
 printf("%p : %s\n", array[i], strings[i]);  
 }  
   
 free(strings); // malloced by backtrace\_symbols  
}

##### libunwind

//https://github.com/ehsan/libunwind  
//./configure --prefix=./install --host=aarch64-linux-gnu  
//make  
//make install  
// FLAGS += -lunwind  
// FLAGS += -lunwind-aarch64   
#include <libunwind.h>  
   
void do\_backtrace()  
{  
 unw\_cursor\_t cursor;  
 unw\_context\_t uc;  
 unw\_word\_t offset, ip, sp;  
   
 char name[256] = {0};  
  
 unw\_getcontext(&uc);  
 unw\_init\_local(&cursor, &uc);  
  
 do  
 {  
 unw\_get\_reg(&cursor, UNW\_REG\_IP, &ip);  
 unw\_get\_reg(&cursor, UNW\_REG\_SP, &sp);  
 buf[0] = '\0';  
  
 (void) unw\_get\_proc\_name(&cursor, name, sizeof(name), &offset);  
 printf ("%p : (%s+0x%x) [%p]\n", ip, name, offset, sp);  
  
 (void) unw\_get\_proc\_info(&cursor, &pi);  
 printf("start(0x%x) end(0x%x) handler(0x%x) lsda(0x%x) gp(0x%x)\n",  
 (long)pi.start\_ip, pi.end\_ip, pi.handler, pi.lsda, pi.gp);  
 } while (unw\_step(&cursor) > 0);  
}

##### 自定义回溯

//如果源代码编译时使用了-O1或-O2优化选项，可执行代码会把ebp/rbp/rsp寄存器当作普通寄存器使用，导致backtrace失败。  
//为了防止这种情况发生，可以在编译时使用-O2 -fno-omit-frame-pointer 或-Og 来避免优化中使用上述寄存器。  
#define STACKCALL \_\_attribute\_\_((regparm(1),noinline))   
void \*\* STACKCALL getEBP(void){   
 void \*\*ebp=NULL;   
 \_\_asm\_\_ \_\_volatile\_\_("mov %%rbp, %0;\n\t"   
 :"=m"(ebp) /\* 输出 \*/   
 : /\* 输入 \*/   
 :"memory"); /\* 不受影响的寄存器 \*/   
 return (void \*\*)(\*ebp);   
}   
   
int my\_backtrace(void \*\*buffer,int size)  
{   
 int frame=0;   
 void \*\* ebp;   
 void \*\*ret=NULL;   
 unsigned long long func\_frame\_distance=0;   
  
 if (buffer != NULL && size > 0)   
 {   
 ebp = getEBP();   
 func\_frame\_distance = (unsigned long long)(\*ebp) - (unsigned long long)ebp;   
 while(ebp && frame < size &&  
 (func\_frame\_distance < (1ULL << 24)) && //assume function ebp more than 16M   
 (func\_frame\_distance > 0))   
 {   
 ret = ebp + 1;   
 buffer[frame++] = \*ret;   
 ebp = (void\*\*)(\*ebp);   
 func\_frame\_distance = (unsigned long long)(\*ebp) - (unsigned long long)ebp;   
 }   
 }   
 return frame;   
}

##### 根据地址找到函数

# 1、拿到地址  
  
# 2、获取maps  
cat /proc/<pid>/maps  
  
# 3、根据地址找到对应的库(如libapp.so), 并计算偏移地址<offset>  
  
# 4-1、根据地址在符号表(程序)中找到函数  
addr2line -C -f -e libapp.so.symbol <offset>  
  
# 4-2、或获取符号表的消息，全局搜<offset>  
nm -a libapp.so.symbol | grep <offset>  
  
# 4-3、或者反汇编程序app，然后全局搜<offset>  
aarch64-linux-gun-objdump -d app | grep <offset>

**参考**

[GNU Binutils简介及基本用法](https://link.zhihu.com/?target=https%3A//www.cnblogs.com/tocy/p/gnu-binutils-simple-usage.html)

[www.cnblogs.com/tocy/p/gnu-binutils-simple-usage.html](https://link.zhihu.com/?target=https%3A//www.cnblogs.com/tocy/p/gnu-binutils-simple-usage.html)

[GNU Binary Utilities](https://link.zhihu.com/?target=https%3A//sourceware.org/binutils/docs-2.28/binutils/index.html)

[sourceware.org/binutils/docs-2.28/binutils/index.html](https://link.zhihu.com/?target=https%3A//sourceware.org/binutils/docs-2.28/binutils/index.html)

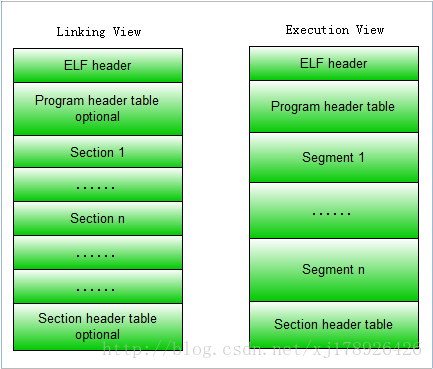
[http://www.skyfree.org/linux/references/ELF\_Format.pdf](https://link.zhihu.com/?target=http%3A//www.skyfree.org/linux/references/ELF_Format.pdf)

[www.skyfree.org/linux/references/ELF\_Format.pdf](https://link.zhihu.com/?target=http%3A//www.skyfree.org/linux/references/ELF_Format.pdf)

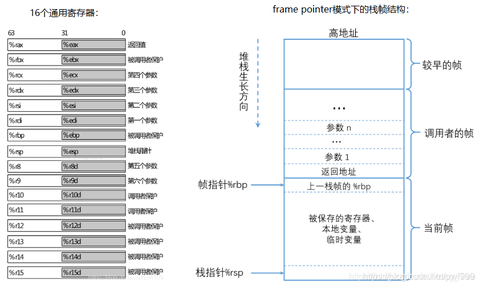
[Linux ELF格式分析 - zzfx - 博客园](https://link.zhihu.com/?target=https%3A//www.cnblogs.com/feng9exe/p/6899351.html)

[www.cnblogs.com/feng9exe/p/6899351.html![img](https://pic1.zhimg.com/v2-2d619cde2ab01846aaa0d6c262a40c0c\_180x120.jpg)](https://link.zhihu.com/?target=https%3A//www.cnblogs.com/feng9exe/p/6899351.html)

[Linux ELF文件格式分析](https://link.zhihu.com/?target=https%3A//blog.csdn.net/xj178926426/article/details/72825630)*[谢健的专栏-CSDN博客](https://link.zhihu.com/?target=https%3A//blog.csdn.net/xj178926426/article/details/72825630)*[elf linux](https://link.zhihu.com/?target=https%3A//blog.csdn.net/xj178926426/article/details/72825630)

[blog.csdn.net/xj178926426/article/details/72825630](https://link.zhihu.com/?target=https%3A//blog.csdn.net/xj178926426/article/details/72825630)

[Unwind 栈回溯详解：libunwind](https://link.zhihu.com/?target=https%3A//blog.csdn.net/Rong_Toa/article/details/110846509)

[blog.csdn.net/Rong\_Toa/article/details/110846509](https://link.zhihu.com/?target=https%3A//blog.csdn.net/Rong_Toa/article/details/110846509)

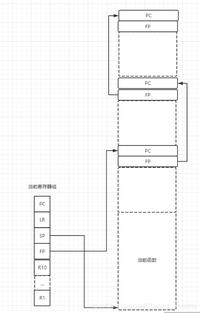
[高效获得Linux函数调用栈/backtrace的方法\_颇锐克-CSDN博客](https://link.zhihu.com/?target=https%3A//blog.csdn.net/prike/article/details/49965113)

[blog.csdn.net/prike/article/details/49965113](https://link.zhihu.com/?target=https%3A//blog.csdn.net/prike/article/details/49965113)

[The libunwind project](https://link.zhihu.com/?target=https%3A//www.nongnu.org/libunwind/)

[www.nongnu.org/libunwind/](https://link.zhihu.com/?target=https%3A//www.nongnu.org/libunwind/)

[ARM Linux 手写实现栈回溯](https://link.zhihu.com/?target=https%3A//blog.csdn.net/Chasing_Chasing/article/details/87161427)

[blog.csdn.net/Chasing\_Chasing/article/details/87161427](https://link.zhihu.com/?target=https%3A//blog.csdn.net/Chasing_Chasing/article/details/87161427)

[谈谈Linux的栈回溯与妙用\_函数](https://link.zhihu.com/?target=https%3A//www.sohu.com/a/256793414_467784)

[www.sohu.com/a/256793414\_467784![img](https://pic1.zhimg.com/v2-885bc8ca5f9a7618065509db856ae8d8\_180x120.jpg)](https://link.zhihu.com/?target=https%3A//www.sohu.com/a/256793414_467784)

<https://zhuanlan.zhihu.com/p/460686470>

C编程之堆栈回溯 <https://www.cnblogs.com/blogernice/articles/13041607.html>

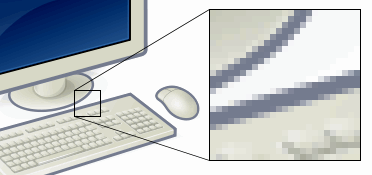
# 第四章：图形显示

## 1. Framebuffer 概念

Framebuffer 是 Linux 下一种图形显示子系统，它是对显示设备抽象出的一个内存映像，允许用户对这个内存映像进行低级别的访问，便于在屏幕上显示图形。FrameBuffer 支持许多不同类型的图形硬件，并提供了一个称为 /dev/fbX 的设备节点，用户可以通过该节点与硬件进行交互。FrameBuffer 控制屏幕上的每个像素，允许图形应用程序在屏幕上绘制图像。由于 FrameBuffer 在硬件级别操作，因此它比传统的X窗口系统更快和更稳定，特别是在较慢的嵌入式设备上。

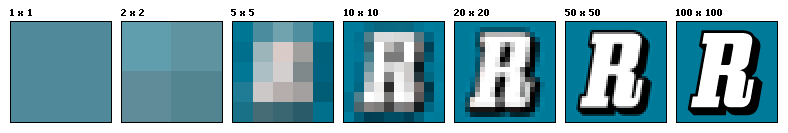
#### 像素

屏幕上显示颜色的最小单位，英文叫 pixel。注意，位图（如jpg、bmp等格式的常见图片）也是由一个个的像素点构成的，跟屏幕的像素点的概念一样。原理上讲，将一张位图显示到屏幕上，就是将图片上的像素点一个个复制到屏幕像素点上。



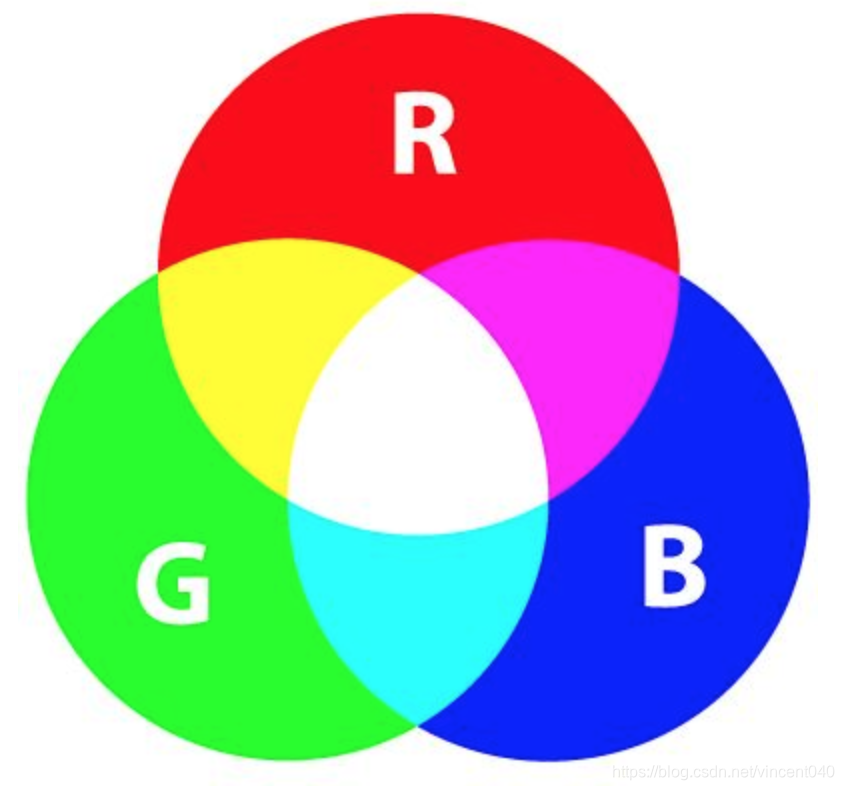
#### 分辨率

分辨率就是一幅画面中有多少个像素点的描述。

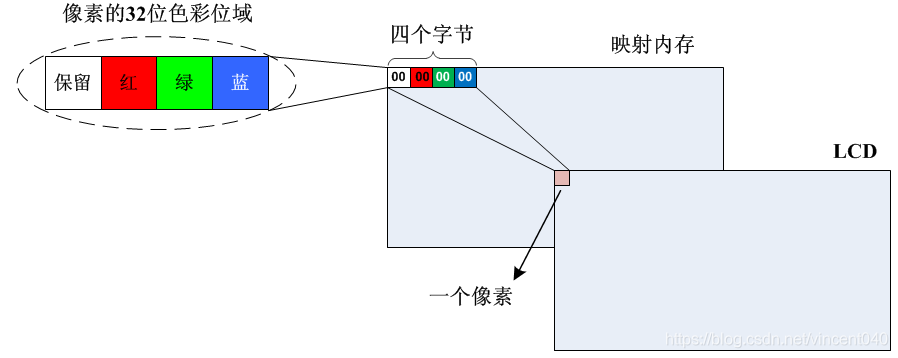


#### 色深

色深是像素的数值大小，同时意味着需要多少个内存字节来描述一个像素。像素有 1 位、8 位、16 位、24 位或 32 位。通过色深我们就能知道屏幕能显示出什么颜色。1 位只有黑白，8 位的颜色是有阴影的黑白，真彩色就是 24 位，有显示三原色红（R）、绿（G）、蓝（B）组成，占用三个字节。16 位颜色是 24 位颜色的简化版，相对的，显示的图像不如 24 颜色的图像鲜艳。32位是对 24 位颜色的扩充，多了一个透明通道，主要用于和其他图像进行混合时使用。



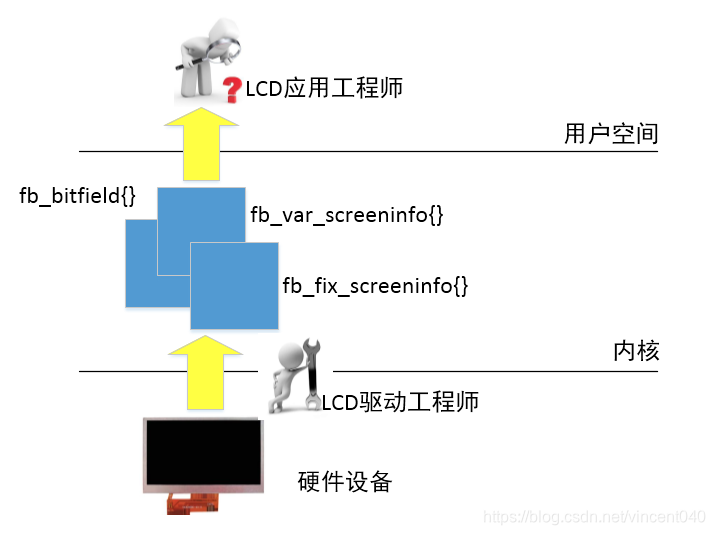
Framebuffer 的本质就是抽象出一块内存，被叫做显存，驱动会自动的将这块显存内的像素数据显示到显示设备上，比如屏幕，或者 LCD。开发者只需要关心要显示什么内容就好了。



#### LCD 的显示分工

驱动工程师会在 Linux 内核中完成显示设备的驱动工作，将显示设备的信息数据填充到固定的结构体中，并在用户空间内生成一个显示设备文件。

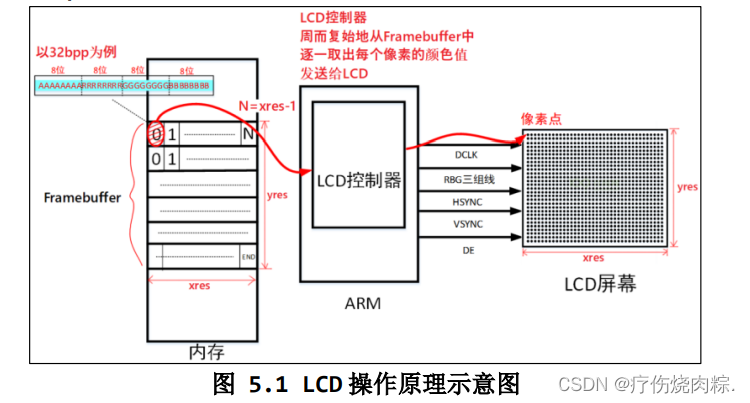
应用工程师在应用层面打开显示设备文件，然后获取显示设备信息，按照显示设备的像素规则将要显示的内容写到驱动分配的内存中，驱动便会自动的将要显示的内容运送到显示设备上，显示出来。



### FrameBuffer 的操作原理

Framebuffer 在编程模型中，就是一块显示内存，显示内存中的数值对应的就是显示硬件 LCD 中的像素点。显存是由驱动申请的连续的物理内存，用户可以将这块内存映射到用户空间，直接读写内存就像于操作 LCD 上的像素点。通过 Framebuffer 设备的 ioctl 操作，可以获取到屏幕显存有关的描述信息。

图片说明：



### FrameBuffer 编程

所需头文件：

#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <sys/mman.h>  
#include <sys/ioctl.h>  
#include <stdio.h>

函数说明：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | **操作** | 函数参数 | 返回值 |
| int open(const char \*pathname, int flags); | open 标准调用。 | pathname 表示打开文件的路径；Flags表示打开文件的方式，常用的有以下6种， ①：O\_RDWR表示可读可写方式打开; ②：O\_RDONLY表示只读方式打开; ③：O\_WRONLY表示只写方式打开; ④：O\_APPEND 表示如果这个文件中本来是有内容的，则新写入的内容会接续到原来内容的后面; ⑤：O\_TRUNC表示如果这个文件中本来是有内容的，则原来的内容会被丢弃，截断； ⑥：O\_CREAT表示当前打开文件不存在，我们创建它并打开它，通常与O\_EXCL结合使用，当没有文件时创建文件，有这个文件时会报错提醒我们； | 打开成功返回文件描述符，失败将返回-1。 |
| int ioctl(int fd, unsigned long request, …); | ioctl 系统调用 | fd 表示文件描述符；request表示与驱动程序交互的命令，用不同的命令控制驱动程序输出我们需要的数据；… 表示可变参数arg，根据request命令，设备驱动程序返回输出的数据。 | 打开成功返回文件描述符，失败将返回-1。 |
| void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset); | mmap系统调用 | addr表示指定映射的內存起始地址，通常设为 NULL表示让系统自动选定地址，并在成功映射后返回该地址；length表示将文件中多大的内容映射到内存中；prot 表示映射区域的保护方式，可以为以下4种方式的组合 ①PROT\_EXEC 映射区域可被执行 ②PROT\_READ 映射区域可被读写 ③PROT\_WRITE 映射区域可被写入 ④PROT\_NONE 映射区域不能存取；Flags 表示影响映射区域的不同特性，常用的有以下两种 ①MAP\_SHARED 表示对映射区域写入的数据会复制回文件内，原来的文件会改变。 ②MAP\_PRIVATE 表示对映射区域的操作会产生一个映射文件的复制，对此区域的任何修改都不会写回原来的文件内容中。 | 若成功映射，将返回指向映射的区域的指针，失败将返回-1。 |

#### FrameBuffer 显存描述结构

#include <linux/fb.h>  
  
// 通过该结构体可以获取显存的 xoffset ,yoffset 的偏移量，和屏幕可见行列像素点 (xres, yres)，以及一个像素所占用的位数 bits\_per\_pixel 的主要信息。   
struct fb\_var\_screeninfo {  
 \_\_u32 xres; /\* 水平可见像素 \*/  
 \_\_u32 yres; /\* 垂直可见像素 \*/  
 \_\_u32 xres\_virtual; /\* 水平虚拟像素 \*/  
 \_\_u32 yres\_virtual; /\* 垂直虚拟像素 \*/  
 \_\_u32 xoffset; /\* 水平偏移量 \*/  
 \_\_u32 yoffset; /\* 垂直偏移量 \*/  
 \_\_u32 bits\_per\_pixel; /\* 每个像素所占位数 \*/  
 \_\_u32 grayscale; /\* 灰度值 \*/  
 struct fb\_bitfield red; /\* 如果为真色彩，表示帧缓冲内存中的位字段，否则仅长度有效 \*/  
 struct fb\_bitfield green;   
 struct fb\_bitfield blue;   
 struct fb\_bitfield transp; /\* 透明度 \*/  
 \_\_u32 nonstd; /\* 非标准像素格式 \*/  
 \_\_u32 activate; /\* 参见 FB\_ACTIVATE\_\* \*/  
 \_\_u32 height; /\* 图像高度 \*/  
 \_\_u32 width; /\* 图像宽度 \*/  
 \_\_u32 accel\_flags; /\* （已过时）参见 fb\_info.flags \*/  
 \_\_u32 pixclock; /\* 像素时钟，以皮秒（pico seconds）为单位 \*/  
 \_\_u32 left\_margin; /\* 从同步到图像的时间 \*/  
 \_\_u32 right\_margin; /\* 从图像到同步的时间 \*/  
 \_\_u32 upper\_margin; /\* 从同步到图像的时间 \*/  
 \_\_u32 lower\_margin;  
 \_\_u32 hsync\_len; /\* 水平同步长度 \*/  
 \_\_u32 vsync\_len; /\* 垂直同步长度 \*/  
 \_\_u32 sync; /\* 参见 FB\_SYNC\_\* \*/  
 \_\_u32 vmode; /\* 参见 FB\_VMODE\_\* \*/  
 \_\_u32 rotate; /\* 逆时针旋转角度 \*/  
 \_\_u32 reserved[5]; /\* 保留字段，用于未来的兼容性 \*/  
};  
  
// 通过该结构体可以获取显存的内存空间大小 smem\_len，每行占用的字节数 line\_length 等。   
struct fb\_fix\_screeninfo {  
 char id[16]; /\* 标识字符串，例如 "TT Builtin" \*/  
 unsigned long smem\_start; /\* 帧缓冲内存的起始地址 \*/  
 \_\_u32 smem\_len; /\* 帧缓冲内存的长度 \*/  
 \_\_u32 type; /\* 类型，参见 FB\_TYPE\_\* \*/  
 \_\_u32 type\_aux; /\* 用于交错平面的附加类型 \*/  
 \_\_u32 visual; /\* 视觉类型，参见 FB\_VISUAL\_\* \*/  
 \_\_u16 xpanstep; /\* 如果没有硬件平移，则为零 \*/  
 \_\_u16 ypanstep; /\* 如果没有硬件平移，则为零 \*/  
 \_\_u16 ywrapstep; /\* 如果没有硬件y轴包装，则为零 \*/  
 \_\_u32 line\_length; /\* 行的长度（以字节为单位） \*/  
 unsigned long mmio\_start; /\* 内存映射I/O的起始地址 \*/  
 \_\_u32 mmio\_len; /\* 内存映射I/O的长度 \*/  
 \_\_u32 accel; /\* 指示驱动程序的加速方式 \*/  
 \_\_u16 reserved[3]; /\* 保留字段，用于未来的兼容性 \*/  
};

#### FrameBuffer 的信息获取

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <stdbool.h>  
#include <errno.h>  
  
#include <sys/types.h>  
#include <sys/mman.h>  
#include <sys/ioctl.h>  
#include <linux/fb.h>  
#include <fcntl.h>  
  
int lcd;  
struct fb\_fix\_screeninfo fixinfo; // 固定属性  
struct fb\_var\_screeninfo varinfo; // 可变属性  
  
void get\_fixinfo()  
{  
 if(ioctl(lcd, FBIOGET\_FSCREENINFO, &fixinfo) != 0)  
 {  
 perror("获取LCD设备固定属性信息失败");  
 return;  
 }  
  
}  
  
void get\_varinfo()  
{  
 if(ioctl(lcd, FBIOGET\_VSCREENINFO, &varinfo) != 0)  
 {  
 perror("获取LCD设备可变属性信息失败");  
 return;  
 }  
}  
  
void show\_info()  
{  
 // 获取LCD设备硬件fix属性  
 get\_fixinfo();  
 printf("\n获取LCD设备固定属性信息成功：\n");  
 printf("[ID]: %s\n", fixinfo.id);  
 printf("[FB类型]: ");  
 switch(fixinfo.type)  
 {  
 case FB\_TYPE\_PACKED\_PIXELS: printf("组合像素\n");break;  
 case FB\_TYPE\_PLANES: printf("非交错图层\n");break;  
 case FB\_TYPE\_INTERLEAVED\_PLANES: printf("交错图层\n");break;  
 case FB\_TYPE\_TEXT: printf("文本或属性\n");break;  
 case FB\_TYPE\_VGA\_PLANES: printf("EGA/VGA图层\n");break;  
 }  
 printf("[FB视觉]: ");  
 switch(fixinfo.visual)  
 {  
 case FB\_VISUAL\_MONO01: printf("灰度. 1=黑;0=白\n");break;  
 case FB\_VISUAL\_MONO10: printf("灰度. 0=黑;1=白\n");break;  
 case FB\_VISUAL\_TRUECOLOR: printf("真彩色\n");break;  
 case FB\_VISUAL\_PSEUDOCOLOR: printf("伪彩色\n");break;  
 case FB\_VISUAL\_DIRECTCOLOR: printf("直接彩色\n");break;  
 case FB\_VISUAL\_STATIC\_PSEUDOCOLOR: printf("只读伪彩色\n");break;  
 }  
 printf("[行宽]: %d 字节\n", fixinfo.line\_length);  
  
 // 获取LCD设备硬件var属性  
 get\_varinfo();  
 printf("\n获取LCD设备可变属性信息成功：\n");  
 printf("[可见区分辨率]: %d×%d\n", varinfo.xres, varinfo.yres);  
 printf("[虚拟区分辨率]: %d×%d\n", varinfo.xres\_virtual, varinfo.yres\_virtual);  
 printf("[从虚拟区到可见区偏移量]: (%d,%d)\n", varinfo.xoffset, varinfo.yoffset);  
 printf("[色深]: %d bits\n", varinfo.bits\_per\_pixel);  
 printf("[像素内颜色结构]:\n");  
 printf(" [红] 偏移量:%d, 长度:%d bits\n", varinfo.red.offset, varinfo.red.length);  
 printf(" [绿] 偏移量:%d, 长度:%d bits\n", varinfo.green.offset, varinfo.green.length);  
 printf(" [蓝] 偏移量:%d, 长度:%d bits\n", varinfo.blue.offset, varinfo.blue.length);  
 printf(" [透明度] 偏移量:%d, 长度:%d bits\n", varinfo.transp.offset, varinfo.transp.length);  
 printf("\n");  
}  
  
int main()  
{  
 lcd = open("/dev/fb0", O\_RDWR);  
 if(lcd == -1)  
 {  
 perror("打开 /dev/fb0 失败");  
 exit(0);  
 }  
  
 // 显示LCD设备属性信息  
 show\_info();  
  
 return 0;  
}

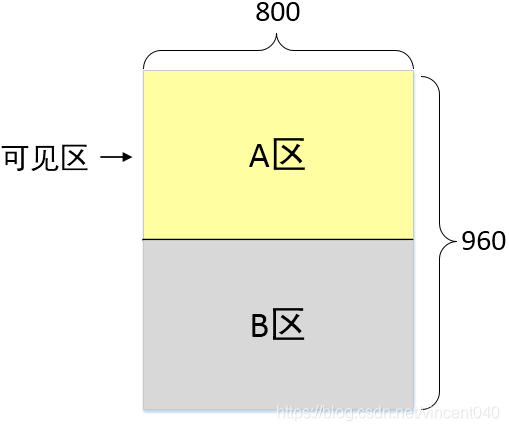
#### FrameBuffer 的多缓冲机制

显示内存对应着显示设备中的像素点。从整体来看，像素点是一个一个的被搬运到显示上的，是串行的，驱动要对显存进行扫描，用于搬运像素点，用户要对显存写入，展示图片。两者是异步的，相当于各自干各自的活。所以就会出现一个现象，当用户将像素写入显存后，要经过驱动扫描到之后屏幕才会显示出来，扫描是有周期的，所以我们能看到像素在屏幕中变化。



在屏幕颜色切换时，能看到颜色切换的过程。期望的是用户能看到一帧完整的画面，而不是画面逐步的显示出来，这显然体验不佳。为了解决这个问题，驱动层引入多缓冲的机制。

多缓冲就是扩大显存区域，让申请到的虚拟显存是可见区域的倍数，通过控制显示区域实现的多缓冲。举个例子：申请的内存为可见内存的两倍，划分它们为 A 页、B 页。在当前可见区域为 A 页的时候，将要显示的画面写在 B 页中，写完之后，再将显示的区域调整到 B 页中，这样就能保证显示的是一个完整的画面，不会出现画面撕裂的情况。



#### FrameBuffer 的使用

FrameBuffer 的设备文件在 Linux 下为 /dev/fbX ，X 从 0 开始，逐渐递增。在 Android 下为 /dev/graphics/fbX ，设备的名字不同，但功能确实一样的。

操作 FrameBuffer 设备主要有五个步骤：

**第一步：打开设备文件**

int fbdev = open("/dev/fb0", O\_RDWR);  
if (fbdev < 0)  
{  
 perror("Error: cannot open framebuffer device");  
 exit(-1);  
}

**第二步：获取屏幕描述信息**

struct fb\_fix\_screeninfo finfo;  
struct fb\_var\_screeninfo vinfo;  
// fb\_fix\_screeninfo 通过 fbdev 获取屏幕固定的相关信息  
if (ioctl(fbdev, FBIOGET\_FSCREENINFO, &finfo) == -1) {  
 perror("Error reading fixed information");  
 close(fbdev);  
 exit(-2);  
}  
// fb\_var\_screeninfo 通过 fbdev 获取可变的信息，可以调用参数为`FBIOPUT\_VSCREENINFO`的重新进行设置  
if (ioctl(fbdev, FBIOGET\_VSCREENINFO, &vinfo) == -1) {  
 perror("Error reading variable information");  
 close(fbdev);  
 exit(-3);  
}

**第三步：映射显存**

// 计算显存的总大小  
long int screensize = vinfo.yres\_virtual \* finfo.line\_length;  
  
// 将显示内存映射到用户空间内  
char \*fbp = (char \*)mmap(NULL, screensize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fbdev, 0);  
if ((int)fbp == -1) {  
 perror("Error: failed to map framebuffer device to memory");  
 exit(-4);  
}

**第四步：操作操作显存显示**

// 计算当前缓冲区偏移  
vinfo.yoffset = vinfo.yoffset + vinfo.yres >= vinfo.yres\_virtual ? 0 : vinfo.yoffset + vinfo.yres;  
  
// 设置画面为黑色  
memset(fbp + finfo.line\_length \* vinfo.yoffset, 0, vinfo.yres \* finfo.line\_length);  
  
// 如果使用了多缓冲，切换显示界面  
if(vinfo.yres\_virtual != vinfo.yres)  
{  
 if (ioctl(fbdev, FBIOPAN\_DISPLAY, &vinfo) == -1) {  
 perror("Error display frame");   
 }  
}

**第五步：程序退出前的清理工作**

munmap(fbp , screensize);  
close(fd\_fb);

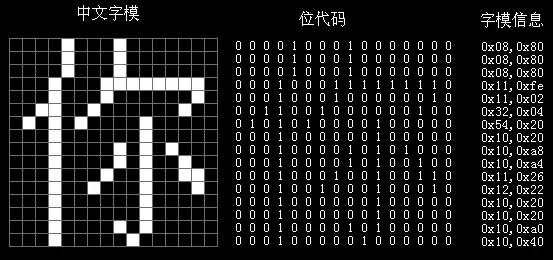
完整程序：

// file: framebuffer.c  
#include <linux/fb.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <sys/mman.h>  
#include <sys/ioctl.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <fcntl.h>  
  
struct fb\_fix\_screeninfo finfo;  
struct fb\_var\_screeninfo vinfo;  
  
int main(int argc, char \*argv[])  
{  
 int fbdev = open("/dev/fb0", O\_RDWR);  
 if (fbdev < 0)  
 {  
 perror("Error: cannot open framebuffer device");  
 exit(-1);  
 }  
   
 // fb\_fix\_screeninfo 通过 fbdev 获取屏幕固定的相关信息  
 if (ioctl(fbdev, FBIOGET\_FSCREENINFO, &finfo) == -1) {  
 perror("Error reading fixed information");  
 close(fbdev);  
 exit(-2);  
 }  
 // fb\_var\_screeninfo 通过 fbdev 获取可变的信息，可以调用参数为`FBIOPUT\_VSCREENINFO`的重新进行设置  
 if (ioctl(fbdev, FBIOGET\_VSCREENINFO, &vinfo) == -1) {  
 perror("Error reading variable information");  
 close(fbdev);  
 exit(-3);  
 }  
  
 // 计算显存的总大小  
 long int screensize = vinfo.yres\_virtual \* finfo.line\_length;  
  
 // 将显示内存映射到用户空间内  
 char \*fbp = (char \*)mmap(NULL, screensize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fbdev, 0);  
 if ((size\_t)fbp == -1) {  
 perror("Error: failed to map framebuffer device to memory");  
 exit(-4);  
 }  
  
   
 // 计算当前缓冲区偏移  
 vinfo.yoffset = vinfo.yoffset + vinfo.yres >= vinfo.yres\_virtual ? 0 : vinfo.yoffset + vinfo.yres;  
  
 // 设置画面为黑色  
 memset(fbp + finfo.line\_length \* vinfo.yoffset, 0, vinfo.yres \* finfo.line\_length);  
  
 // 如果使用了多缓冲，切换显示界面  
 if(vinfo.yres\_virtual != vinfo.yres)  
 {  
 if (ioctl(fbdev, FBIOPAN\_DISPLAY, &vinfo) == -1) {  
 perror("Error display frame");   
 }  
 }  
   
 sleep(1);  
 // 计算当前缓冲区偏移  
 vinfo.yoffset = vinfo.yoffset + vinfo.yres >= vinfo.yres\_virtual ? 0 : vinfo.yoffset + vinfo.yres;  
  
 // 设置画面为白色  
 memset(fbp + finfo.line\_length \* vinfo.yoffset, 0xff, vinfo.yres \* finfo.line\_length);  
  
 // 如果使用了多缓冲，切换显示界面  
 if(vinfo.yres\_virtual != vinfo.yres)  
 {  
 if (ioctl(fbdev, FBIOPAN\_DISPLAY, &vinfo) == -1) {  
 perror("Error display frame");   
 }  
 }  
   
 // 解除内存映射  
 munmap(fbp , screensize);  
 // 关闭设备  
 close(fbdev);  
   
 return 0;  
}

上面的程序会打开 /dev/fb0 设备，设置屏幕为黑色，等待 1 秒后，设置屏幕为白色。并自动检测是否使用多缓冲。

#### 在 FrameBuffer 写字

FrameBuffer 是一块连续的内存区，代表的是一个个像素点。在 FrameBuffer 上写字，就相当于在一张有格子的纸上将字一个一个的涂出来。就像下面图片示意的那样：



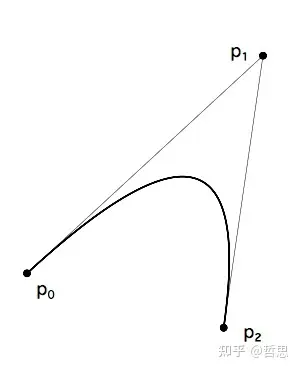
上面是一个相对固定的字符“你”显示出来的样子，但是在实际的使用中，电脑需要显示各种形状的字符，同时还要放大或者缩小。上面的这种字型的保存方法，不单需要占用较大的内存，同时，很难进行缩放。针对字符，工程师开发出了矢量字形表示方法。

##### TrueType 字符格式

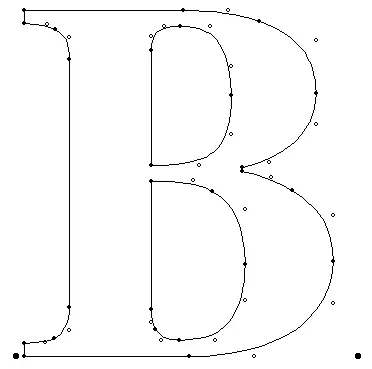
TrueType 是 Apple 公司开发的轮廓字体标准，以为字体开发人员提供高度控制，可在不同字体大小下正确显示著称。现已成为 mac os、windows 等操作系统上最常见的字体格式。一般的文件拓展名为“.ttf”。

为了提高不同字体间相同字形的复用率，拓展 TrueType 字体格式，将多种字体组合到一个文件中，称为 TrueType Collection，常见的拓展名为“.ttc”。

矢量字体轮廓是用曲线来描述的，这种曲线被称为贝塞尔曲线。贝塞尔曲线 (B6zier curve)，又称贝兹曲线或贝济埃曲线，是应用于二维图形应用程序的数学曲线。曲线的定义有四个点：起始点、终止点(也称锚点)以及两个相互分离的中间点，滑动两个中间点，贝塞尔曲线的形状会发生变化。依据四个位置任意的点坐标可绘制出一条光滑曲线。



通过组合贝塞尔曲线，就能得到一个字符的外形。



这种矢量图像非常方便缩放，只需将对应的坐标乘以放大系数便可以了。而想要根据上面数据画出一个字符也只需要三步：

第一步、将坐标缩放到需要的大小；第二步、通过贝塞尔曲线算法，填充出外形；第三步、使用填充算法，将内部填充上需要的颜色。这样一个字符的点阵便诞生了，将其送到显存中，我们便能看到显示屏上的字符。

##### FreeType 解析库

FreeType 库是一个完全免费（开源）的、高质量的且可移植的字体引擎，它提供统一的接口来访问多种字体格式文件，包括 TrueType, OpenType, Type1, CID, CFF, Windows FON/FNT, X11 PCF 等。通过这个库，能够很方便的将想要表达的字符用想要的字体在显示设备上显示出来，从而避免了手工去解析 ttf 文件，画字体图等一系列复杂的操作。

FreeType 库帮我们做了很多事情，有兴趣的可以去官网查看 FreeType 做了哪些细致的工作，在这里主要讲述怎么使用 FreeType 库在屏幕上画出字符。

FreeType 安装：

FreeType 的官网是 <https://freetype.org/>，在官网中有该库的文档，也能下载到 FreeType 的源码库。

# 下载并解压源码  
wget https://download.savannah.gnu.org/releases/freetype/freetype-2.13.2.tar.xz  
tar -xvf freetype-2.13.2.tar.xz  
  
# 进入目录  
cd freetype-2.13.2  
  
# 由于 freetype 有很多额外的功能，在这里只做最小编译，减小依赖。  
./configure --with-zlib=no --with-bzip2=no --with-png=no --with-brotli=no --with-librsvg=no --with-harfbuzz=no --prefix=`pwd`/install  
  
# 编译  
make   
  
# 安装  
make install   
# 安装命令执行成功后，freetype 会被安装到当前的 install 目录下。  
# 在 gcc 编译时，使用 -L 选项将 install/lib 目录加入库寻找目录，使用 -I 选项将 install/include/freetype2 添加到头文件寻找目录

##### FreeType 文字的显示过程

参考 freetype 官方帮助文档（ freetype-doc-2.10.2/docs/tutorial 中的例程和 step1/step2/step3 文档），可以知道程序中，使用 freetype 库显示文字流程为：

1）包含头文件及API头文件：ft2build.h；

#include <ft2build.h>  
#include FT\_FREETYPE\_H // 基础的FreeType 2 API  
#include FT\_GLYPH\_H // Glyph Images 管理  
#include FT\_BITMAP\_H // 位图管理

2）初始化： FT\_InitFreetype

FT\_Library library;  
FT\_Face face;  
FT\_GlyphSlot slot;  
FT\_Error error;  
error = FT\_Init\_FreeType(&library);  
if (error) {  
 printf("Error initializing FreeType library\n");  
 return 1;  
}

3）加载（打开）字体Face： FT\_New\_Face

error = FT\_New\_Face(library, "fangsong\_GB2312.ttf", 0, &face);  
if (error) {  
 printf("Error loading font\n");  
 FT\_Done\_FreeType(library);  
 return 1;  
}

4）设置字体大小：FT\_Set\_Char\_Size 或 FT\_Set\_Pixel\_Size

FT\_Set\_Char\_Size：设置标称尺寸（nominal size in points，1pt=0.376mm）；

FT\_Set\_Pixel\_Size：设置标称尺寸（nominal size (in pixels)，单位是像素点）；

error = FT\_Set\_Char\_Size(face, 0, 16 \* 64, 300, 300);  
if (error) {  
 printf("Error setting character size\n");  
 FT\_Done\_Face(face);  
 FT\_Done\_FreeType(library);  
 return 1;  
}

6）根据编码值 charcode 找到 glyph\_index： glyph\_index = FT\_Get\_Char\_Index(face, charcode)；  
 获取指定字符编码 charcode 的 glyph 索引。加载 glyph 图像到 slot 的函数 FT\_Load\_Glyph，会用到该索引。

7）根据 glyph\_index 取出 glyph ： FT\_Load\_Glyph(face, glyph\_index, load\_flags)；

8）转换为位图：FT\_Render\_Glyph；  
 将给定的glyph 图像转换为bitmap（位图）。

上面步骤6、7、8可以用一个函数替换：FT\_Load\_Char(face, charcode, FT\_LOAD\_RENDER)，可以直接得到glyph对应的bitmap。

slot = face->glyph;  
  
error = FT\_Load\_Char(face, 'a', FT\_LOAD\_RENDER);  
if (error)  
{  
 continue; // Ignore errors and continue.  
}  
int height = slot->bitmap.rows;  
int width = slot->bitmap.width;  
int bitlen = slot->bitmap.pitch;  
for (int row = 0; row < height; ++row)  
{  
 for (int col = 0; col < width; ++col)  
 {  
 int index = row \* bitlen + col;  
  
 // 对于灰度位图，值越高表示越亮  
 unsigned char pixel = slot->bitmap.buffer[index];  
 printf("%c", pixel > 128 ? '#' : '-');  
 }  
 printf("\n");  
}

9）释放位图内存；

FT\_Bitmap\_Done(library, &face->glyph->bitmap);

10）释放字体资源；

FT\_Done\_Face(face);  
FT\_Done\_FreeType(library);

一个简单完整的示例：

// 引入所需的头文件  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <ft2build.h>  
#include FT\_FREETYPE\_H  
#include FT\_BITMAP\_H  
  
int main(int argc, char \*argv[])  
{  
 FT\_Library library; // FreeType 库对象  
 FT\_Face face; // 字体对象  
 FT\_GlyphSlot slot; // 字形槽对象，用于加载和渲染字符  
 FT\_Error error; // 错误码  
  
 // 初始化 FreeType 库  
 error = FT\_Init\_FreeType(&library);  
 if (error)  
 {  
 printf("Error initializing FreeType library\n");  
 return 1;  
 }  
  
 // 加载字体文件并创建字体对象  
 error = FT\_New\_Face(library, "fangsong\_GB2312.ttf", 0, &face);  
 if (error)  
 {  
 printf("Error loading font\n");  
 FT\_Done\_FreeType(library); // 释放 FreeType 库资源  
 return 1;  
 }  
  
 // 设置字符大小  
 error = FT\_Set\_Char\_Size(face, 0, 16 \* 64, 300, 300);  
 if (error)  
 {  
 printf("Error setting character size\n");  
 FT\_Done\_Face(face); // 释放字体对象  
 FT\_Done\_FreeType(library); // 释放 FreeType 库资源  
 return 1;  
 }  
  
 slot = face->glyph; // 获取字形槽对象  
  
 const wchar\_t text[] = {'h','e','l','l','o',' ','w','o','r','l','d'};  
 int num\_chars = sizeof(text) / sizeof(text[0]);  
  
 for (int n = 0; n < num\_chars; n++)  
 {  
 // 加载字符的字形数据  
 error = FT\_Load\_Char(face, text[n], FT\_LOAD\_RENDER);  
 if (error)  
 {  
 continue; // 忽略错误并继续循环  
 }  
  
 int height = slot->bitmap.rows;  
 int width = slot->bitmap.width;  
 int bitlen = slot->bitmap.pitch;  
  
 for (int row = 0; row < height; ++row)  
 {  
 for (int col = 0; col < width; ++col)  
 {  
 int index = row \* bitlen + col;  
  
 // 对于灰度位图，值越高表示越亮  
 unsigned char pixel = slot->bitmap.buffer[index];  
 printf("%c", pixel > 128 ? '#' : '-');  
 }  
 printf("\n");  
 }  
  
 // 释放字符的字形位图数据  
 FT\_Bitmap\_Done(library, &face->glyph->bitmap);  
 }  
  
 FT\_Done\_Face(face); // 释放字体对象  
 FT\_Done\_FreeType(library); // 释放 FreeType 库资源  
  
 return 0;  
}

执行效果

nihao@nihao-z690:~$ ./freetype\_example   
-----------------------------  
-------#---------------------  
-----###---------------------  
---#####---------------------  
-#######---------------------  
-#######---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####---------------------  
----####------########-------  
----####-----##########------  
----####----############-----  
----####---##------#####-----  
----####--#---------####-----  
----######----------#####----  
----#####------------####----  
----#####------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----####-------------####----  
----#####-----------######---  
-###########------##########-  
-###########-----############  
-----------------------------  
.......

##### FreeType 在 Framebuffer 中协议 "哈" 字

#include <linux/fb.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <sys/mman.h>  
#include <sys/ioctl.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <fcntl.h>  
  
#include <stdlib.h>  
#include <string.h>  
#include <ft2build.h>  
#include FT\_FREETYPE\_H  
#include FT\_BITMAP\_H  
  
#include <stdio.h>  
#include <stdint.h>  
  
struct fb\_fix\_screeninfo finfo;  
struct fb\_var\_screeninfo vinfo;  
  
int main(int argc, char \*argv[])  
{  
 int fbdev = open("/dev/fb1", O\_RDWR);  
 if (fbdev < 0)  
 {  
 perror("Error: cannot open framebuffer device");  
 exit(-1);  
 }  
  
 // fb\_fix\_screeninfo 通过 fbdev 获取屏幕固定的相关信息  
 if (ioctl(fbdev, FBIOGET\_FSCREENINFO, &finfo) == -1)  
 {  
 perror("Error reading fixed information");  
 close(fbdev);  
 exit(-2);  
 }  
 // fb\_var\_screeninfo 通过 fbdev 获取可变的信息，可以调用参数为`FBIOPUT\_VSCREENINFO`的重新进行设置  
 if (ioctl(fbdev, FBIOGET\_VSCREENINFO, &vinfo) == -1)  
 {  
 perror("Error reading variable information");  
 close(fbdev);  
 exit(-3);  
 }  
  
 // 计算显存的总大小  
 long int screensize = vinfo.yres\_virtual \* finfo.line\_length;  
  
 // 将显示内存映射到用户空间内  
 char \*fbp = (char \*)mmap(NULL, screensize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fbdev, 0);  
 if ((size\_t)fbp == -1)  
 {  
 perror("Error: failed to map framebuffer device to memory");  
 exit(-4);  
 }  
  
 // 计算当前缓冲区偏移  
 vinfo.yoffset = vinfo.yoffset + vinfo.yres >= vinfo.yres\_virtual ? 0 : vinfo.yoffset + vinfo.yres;  
 uint16\_t \*fb\_mem = (uint16\_t \*)(fbp + finfo.line\_length \* vinfo.yoffset);  
  
 FT\_Library library;  
 FT\_Face face;  
 FT\_GlyphSlot slot;  
 FT\_Error error;  
  
 error = FT\_Init\_FreeType(&library);  
 if (error)  
 {  
 printf("Error initializing FreeType library\n");  
 return 1;  
 }  
  
 error = FT\_New\_Face(library, "fangsong\_GB2312.ttf", 0, &face);  
 if (error)  
 {  
 printf("Error loading font\n");  
 FT\_Done\_FreeType(library);  
 return 1;  
 }  
  
 error = FT\_Set\_Char\_Size(face, 0, 16 \* 64 \* 3, 300, 300);  
 if (error)  
 {  
 printf("Error setting character size\n");  
 FT\_Done\_Face(face);  
 FT\_Done\_FreeType(library);  
 return 1;  
 }  
  
 slot = face->glyph;  
  
 // 哈  
 const FT\_ULong text = 0x54c8;  
  
 error = FT\_Load\_Char(face, text, FT\_LOAD\_RENDER);  
 if (error)  
 {  
 goto error\_end; // Ignore errors and continue.  
 }  
 int height = slot->bitmap.rows;  
 int width = slot->bitmap.width;  
 int bitlen = slot->bitmap.pitch;  
 for (int row = 0; row < height; ++row)  
 {  
 for (int col = 0; col < width; ++col)  
 {  
 int index = row \* bitlen + col;  
  
 // 对于灰度位图，值越高表示越亮  
 unsigned char pixel = slot->bitmap.buffer[index];  
 fb\_mem[row \* vinfo.xres + col] = pixel > 128 ? 0xffff : 0;  
 }  
 }  
  
 FT\_Bitmap\_Done(library, &face->glyph->bitmap);  
error\_end:  
  
 FT\_Done\_Face(face);  
 FT\_Done\_FreeType(library);  
  
 // 如果使用了多缓冲，切换显示界面  
 if (vinfo.yres\_virtual != vinfo.yres)  
 {  
 if (ioctl(fbdev, FBIOPAN\_DISPLAY, &vinfo) == -1)  
 {  
 perror("Error display frame");  
 }  
 }  
  
 // 解除内存映射  
 munmap(fbp, screensize);  
 // 关闭设备  
 close(fbdev);  
  
 return 0;  
}



##### FreeType 主要函数和结构体介绍

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | **操作** | 函数参数 | 返回值 |
| FT\_Error FT\_Init\_FreeType(FT\_Library \*alibrary); | 用于初始化 FreeType 库的核心数据结构，使您能够使用 FreeType 来处理字体文件并执行字形渲染操作。 | alibrary：一个指向 FT\_Library 结构指针的指针，用于存储初始化后的 FreeType 库的句柄。FT\_Library 是 FreeType 库的核心结构，它包含了许多与库相关的设置和信息。 | FT\_Error 类型，如果初始化成功，则返回 0，否则返回错误码。 |
| void FT\_Done\_FreeType(FT\_Library library); | 用于在不再需要使用 FreeType 库时，释放与库相关的资源，以防止内存泄漏。 | library：一个 FT\_Library 结构，它是通过 FT\_Init\_FreeType 初始化得到的 FreeType 库句柄。 | 无 |
| FT\_Error FT\_New\_Face(FT\_Library library, const char\* filepathname, FT\_Long face\_index, FT\_Face \*aface); | 用于加载字体文件并创建一个新的字体对象，以便后续进行字形渲染和其他字体相关操作。 | library：一个 FT\_Library 结构，是 FreeType 库的句柄，通常是通过 FT\_Init\_FreeType 初始化得到的。 filepathname：要加载的字体文件的路径和名称。可以是绝对路径或相对路径。 face\_index：用于指定字体文件中的字体索引。字体文件可以包含多个字体，这个参数允许您选择加载哪一个字体。通常，face\_index 为 0 表示加载第一个字体。 aface：一个指向 FT\_Face 结构指针的指针，用于存储加载后的字体对象。 | FT\_Error 类型，如果加载成功，则返回 0，否则返回错误码。 |
| void FT\_Done\_Face(FT\_Face face); | 用于释放已加载的字体对象（FT\_Face），以便释放与该字体相关的资源并防止内存泄漏。 | face：一个 FT\_Face 结构，代表了一个已加载的字体对象。这个对象通常是通过 FT\_New\_Face 函数加载字体文件后获得的。 | 无 |
| FT\_Error FT\_Set\_Char\_Size(FT\_Face face, FT\_F26Dot6 char\_width, FT\_F26Dot6 char\_height, FT\_UInt horz\_resolution, FT\_UInt vert\_resolution); | 用于设置字符大小和像素大小，以便在渲染字形时控制字形的大小。该函数通常在加载字体后并在渲染之前调用。 | face：一个 FT\_Face 结构，代表了已加载的字体对象，您可以使用此字体对象来设置字符大小。 char\_width：指定字符的宽度（水平方向上的大小），以 26.6 固定点数表示。通常情况下，您可以将其设置为 0，以便自动计算宽度。 char\_height：指定字符的高度（垂直方向上的大小），以 26.6 固定点数表示。 horz\_resolution：水平方向上的分辨率，通常以每英寸像素数（DPI）为单位。这个参数影响字符宽度的转换。 vert\_resolution：垂直方向上的分辨率，通常以每英寸像素数（DPI）为单位。这个参数影响字符高度的转换。 | FT\_Error 类型，如果设置成功，则返回 0，否则返回错误码。 |
| FT\_Error FT\_Load\_Char(FT\_Face face, FT\_ULong char\_code, FT\_Int32 load\_flags); | 于加载指定字符的字形信息到字体对象中，以便后续进行字形渲染和其他字体相关操作。 | face：一个 FT\_Face 结构，代表了已加载的字体对象，您可以使用此字体对象加载字符的字形信息。 char\_code：要加载的字符的 Unicode 编码或字符编码。通常情况下，您可以使用字符的 Unicode 编码来加载字形信息。 load\_flags：加载标志，用于指定加载字形信息的选项。可以通过按位或操作来组合多个标志。一些常用的标志包括： FT\_LOAD\_DEFAULT：默认标志，加载标准字形。 FT\_LOAD\_NO\_BITMAP：不加载位图字形。 FT\_LOAD\_FORCE\_AUTOHINT：强制启用自动提示（hinting）。 FT\_LOAD\_TARGET\_NORMAL：用于正常分辨率渲染。 FT\_LOAD\_TARGET\_MONO：用于单色位图渲染。 | FT\_Error 类型，如果加载成功，则返回 0，否则返回错误码。 |
| void FT\_Bitmap\_Done(FT\_Library library, FT\_Bitmap \*bitmap); | 用于释放位图（bitmap）对象的内存，以防止内存泄漏。位图对象通常包含在 FT\_GlyphSlot 结构中，用于存储字形的位图数据。 | library：一个 FT\_Library 结构，是 FreeType 库的句柄，通常是通过 FT\_Init\_FreeType 初始化得到的。 bitmap：一个 FT\_Bitmap 结构，代表了要释放内存的位图对象。 | 无 |

typedef struct FT\_Bitmap\_  
{  
 int rows; /\* 位图的行数，表示位图的高度。 \*/  
 int width; /\* 位图的列数，表示位图的宽度。 \*/  
 int pitch; /\* 位图的行字节数，通常是位图宽度的整数倍。 \*/  
 unsigned char\* buffer; /\* 存储位图像素数据的缓冲区。 \*/  
 short num\_grays; /\* 位图的灰度级别，通常为 256。 \*/  
 char pixel\_mode; /\* 位图像素模式，通常为 FT\_PIXEL\_MODE\_GRAY。 \*/  
 char palette\_mode; /\* 调色板模式，通常为 FT\_PALETTE\_MODE\_NONE。 \*/  
 void\* palette; /\* 调色板数据。 \*/  
} FT\_Bitmap;

版权声明：本文为CSDN博主「QRS\_HL」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/m0_65525149/article/details/131145814>

## 2. GUI

GUI 本质上就是虚拟化的显示设备 ，显示设备复用技术。显示设备只有一个，但是用户程序却有很多个，如果让这么多程序都能在显示设备上展示出来，怎么组织和管理都是由 GUI 框架来管理。引用百度百科的描述如下：

图形用户界面（Graphical User Interface，简称 GUI，又称图形用户接口）是指采用图形方式显示的计算机操作用户界面。  
图形用户界面是一种人与[计算机通信](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E9%80%9A%E4%BF%A1/8082711%3FfromModule%3Dlemma_inlink)的界面显示[格式](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/%E6%A0%BC%E5%BC%8F/2406%3FfromModule%3Dlemma_inlink)，允许用户使用[鼠标](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/%E9%BC%A0%E6%A0%87/122323%3FfromModule%3Dlemma_inlink)等输入设备操纵[屏幕](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/%E5%B1%8F%E5%B9%95/3750314%3FfromModule%3Dlemma_inlink)上的[图标](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/%E5%9B%BE%E6%A0%87/3823326%3FfromModule%3Dlemma_inlink)或菜单选项，以选择命令、调用文件、启动程序或执行其它一些[日常任务](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/%E6%97%A5%E5%B8%B8%E4%BB%BB%E5%8A%A1/10744783%3FfromModule%3Dlemma_inlink)。与通过键盘输入文本或字符命令来完成例行任务的字符界面相比，图形用户界面有许多优点。图形用户界面由窗口、下拉菜单、对话框及其相应的控制机制构成，在各种新式应用程序中都是标准化的，即相同的操作总是以同样的方式来完成，在图形用户界面，用户看到和操作的都是图形对象，应用的是计算机图形学的技术。  
——《[百度百科](https://link.zhihu.com/?target=https%3A//baike.baidu.com/item/GUI/479966)》

GUI 关联的技术是非常多的，从计算机拥有可用于展示图像画面的屏幕后，计算机视觉几乎占到了计算机技术的一半以上，而计算机的 GUI 显示又占到计算机视觉技术的一大半以上。计算机要完成人类很难完成的图像计算任务，同时也需要将处理后的图像展示出来，这是计算机视觉诞生后一直发展的路线。 GUI 正是承担如何展示计算机视觉的任务，所以在 GUI 发展的过程中，逐步的涌现出非常多的库。由于这些库每个单独拿出来都能写数本书，所以在此只做简单介绍，对 GUI 中一些技术有一些了解。

### GUI 有关名词简介

**OpenGL**

OpenGL (Open Graphics Library) 是一个跨语言，跨平台的应用程序接口，提供 2D，3D 图形渲染接口，操作在 GPU 之上，实现硬件加速渲染。OpenGL 只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。但是在你真正能够在程序中使用 OpenGL 之前，你需要对他进行初始化，但是由于 OpenGL 是跨平台的，所以也没有一个标准的方式进行初始化。OpenGL 初始化分为两个阶段：第一个阶段、你需要创建一个 OpenGL 上下文环境；第二个阶段、你需要定位所有需要在 OpenGL 中使用的函数；

**OpenGL ES**

penGL ES（OpenGL for Embedded Systems）是 OpenGL 三维图形API的子集，针对手机、PDA和游戏主机等嵌入式设备而设计，各显卡制造商和系统制造商来实现这组 API。

**GLEW**

GLEW(The OpenGL Extension Wrangler Library ) 是一个基于 OpenGL 图形接口的跨平台的 C++ 扩展库。由于 OpenGL 驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。任务就落在了开发者身上，开发者需要在运行时获取函数地址并将其保存在一个函数指针中供以后使用。GLEW 能自动识别当前平台所支持的全部 OpenGL 高级扩展涵数。只要包含 glew.h 头文件，就能使用 gl,glu,glext,wgl,glx 的全部函数。GLEW 支持目前流行的各种操作系统。

不过目前还有一个类似的最流行的库 GLAD。

**GLFW**

GLFW (Graphics Library Framework) 是一个专门针对 OpenGL 的 C 语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建 OpenGL 上下文，定义窗口参数以及处理用户输入。简单来说，GLFW 负责创建窗口，处理窗口相关的事件（如键盘和鼠标输入），并提供一个OpenGL上下文供你的程序使用。相似的库还有 GLUT，FreeGLUT。

**vulkan**

Vulkan是一个跨平台的2D和3D绘图应用程序接口（API），最早由科纳斯组织（Khronos Group）在2015年游戏开发者大会（GDC）上发表。科纳斯最先把 VulkanAPI 称为“下一代 OpenGL 行动”（next generation OpenGL initiative）或 “glNext”， 但在正式宣布 Vulkan 之后这些名字就没有再使用了。就像OpenGL，Vulkan 针对实时 3D 程序（如电子游戏）设计，Vulkan 并计划提供高性能和低 CPU 管理负担（overhead），这也是 Direct3D12 和 AMD 的 Mantle 的目标。Vulkan 兼容 Mantle 的一个分支，并使用了 Mantle 的一些组件。Vulkan 旨在提供更低的 CPU 开销与更直接的 GPU 控制，其理念大致与 Direct3D 12 和 Mantle 类似。

**DirectFB**

DirectFB 项目是由德国 Convergence 公司推动的 Open source 计划的一部分，它是专门为满足嵌入式设备要求而开发的小巧、强大、灵活和易于使用的图形系统，并且试图成为一个建构于 Linux Framebuffer Device 之上的新图形标准。它在 FrameBuffer 的基础上提供了图形加速、输入设备处理提取、透明窗口和多重显示层的功能，过程能够对嵌入式系统 GUI 有较好的支持。与那些通用的嵌入式 GUI 系统相比，它具有非常简洁、高效的体系结构和硬件图形加速功能。

**X Window**

X Window 系统（X Window System，也常称为 X11或 X ）是一种以位图方式显示的软体视窗系统。最初是 1984 年麻省理工学院的研究，之后变成 UNIX、类UNIX、以及 OpenVMS 等操作系统所一致适用的标准化软体工具套件及显示架构的运作协定。X Window 系统透过软体工具及架构协定来建立作业系统所用的图形用户界面，此后则逐渐扩展适用到各形各色的其他作业系统上。现在几乎所有的作业系统都能支援与使用 X 。更重要的是，今日知名的桌面环境 GNOME 和 KDE 也都是以 X Window 系统为基础建构成的。

由于 X 只是工具套件及架构规范，本身并无实际参与运作的实体，所以必须有人依据此标准进行开发撰写。如此才有真正可用、可执行的实体，始可称为实现体。目前依据 X 的规范架构所开发撰写成的实现体中，以 X.Org 最为普遍且最受欢迎。X.Org 所用的协定版本，X11，是在 1987 年 9 月所发布。而今最新的参考实作（参考性、示范性的实作体）版本则是 X11 Release 7.7（简称：X11R7.7），而此专案由 X.Org 基金会所领导，且是以 MIT 授权和相似的授权许可的自由软体。

X Window 除 windows 和 Quartz 之外的非常流行的窗口系统，如果你使用的是 Linux 或者类 UNIX 系统，大概率会用到 X Window。

**Wayland**

Wayland 由X.Org开发人员 Kristian Hogsberg 于 2008 年作为个人项目开始。它是一种通信协议 ，用于指定显示服务器与其客户端之间的通信。Wayland 是作为一个免费的开源社区驱动的项目而开发的，目的是用现代，安全和简单的窗口系统代替 X Window System（也称为X11或Xorg）。但由于缺乏支持，Wayland 运行时稳定性不高。截至目前，已经成为 Ubuntu 的默认窗口系统。

**GTK**

GTK，全称为 GIMP Toolkit，是一个开源的图形用户界面（GUI）工具集。它最初是为图像处理程序GIMP（GNU Image Manipulation Program）开发的，但后来演变成为一个广泛用于开发跨平台应用程序的库。

**GNOME**

GNOME是一套纯粹自由的计算机软件，运行在操作系统上，提供图形桌面环境。 GNOME 包含了 Panel （用来启动此程式和显示目前的状态）、桌面 （应用程式和资料放置的地方）及一系列的标准桌面工具和应用程式，并且能让各个应用程式都能正常地运作。 GNOME是Linux操作系统上最常用的图形桌面环境之一。

**KDE**

KDE，即 K 桌面环境，全称 K Desktop Environment。它是一种著名的运行于 Linux、Unix 以及 BSD 等类 UNIX 操作系统上的自由图形桌面环境，整个系统采用的都是 TrollTech 公司所开发的 Qt 程序库。 KDE 是 Linux 操作系统上流行的桌面环境之一。  
K 桌面项目始建于 1996 年 10 月，确切的公布日期是 1996 年 10 月 14 日。K 桌面项目是由图形排版工具 Lyx 的开发者、一位名为 Matthias Ettrich 的德国人发起的，目的是为满足普通用户也能够通过简单易用的桌面来管理 Unix 工作站上的各种应用软件以及完成各种任务。

**MiniGUI**

MiniGUI 是由北京飞漫软件技术有限公司创办的开源 Linux 图形用户界面支持系统，经过近些年的发展，MiniGUI 已经发展成为比较成熟的性能优良的、功能丰富的跨操作系统的嵌入式图形界面支持系统。“小” 是 MiniGUI 的特色，它已经广泛应用于通讯、医疗、工控、电子、机顶盒、多媒体等领域。MiniGUI 的最新版本为MiniGUI 5.0 。MiniGUI 对中文的支持很友好。它支持 GB2312 与 BIG5 字符集，其他字符集也可以轻松加入。

**QT**

Qt 是一个 1991 年由 Qt Company 开发的跨平台 C++ 图形用户界面应用程序开发框架。它既可以开发 GUI 程序，也可用于开发非 GUI 程序，比如控制台工具和服务器。Qt 是面向对象的框架，使用特殊的代码生成扩展（称为元对象编译器( Meta Object Compiler, moc )）以及一些宏，Qt 很容易扩展，并且允许真正的组件编程。  
2008 年，Qt Company 科技被诺基亚公司收购，Qt 也因此成为诺基亚旗下的编程语言工具。2012 年，Qt 被 Digia 收购。  
2014 年 4 月，跨平台集成开发环境 Qt Creator 3.1.0 正式发布，实现了对于 iOS 的完全支持，新增 WinRT、Beautifier 等插件，废弃了无 Python 接口的 GDB 调试支持，集成了基于 Clang 的 C/C++ 代码模块，并对 Android 支持做出了调整，至此实现了全面支持 iOS、Android、WP,它提供给应用程序开发者建立艺术级的图形用户界面所需的所有功能。基本上，Qt 同 X Window 上的 Motif，Openwin，GTK 等图形界面库和 Windows 平台上的 MFC，OWL，VCL，ATL 是同类型的东西。

### MiniGUI

#### 简介

MiniGUI 是一款面向嵌入式系统的高级窗口系统（Windowing System）和图形用户界面（Graphical User Interface，GUI）支持系统。MiniGUI 旨在资源受限的嵌入式设备提供现代窗口系统，让嵌入式设备也能体验到和电脑上类似的 GUI 窗口体验。

从技术上讲，MiniGUI 为嵌入式系统提供如下功能：

* 完整的多窗口系统。在支持多进程的 Linux 操作系统上，MiniGUI 提供了完善的多进程环境下的窗口系统；在诸如 VxWorks 这样的操作系统上，MiniGUI 提供多线程模式下的窗口系统。不同进程（或者线程）创建的窗口，可以协调共存于同一个屏幕之上，就像 X Window 或者我们熟悉的 Windows 那样。
* 完备的图形功能。MiniGUI 为应用程序提供了丰富的图形绘制功能，开发者可以使用 MiniGUI 开发复杂的图形应用程序，典型的如 GIS 系统、浏览器软件等。
* 完备的图形用户界面构件（Widget）集。为方便应用程序的开发，MiniGUI 为应用程序提供了近三十种控件（在 X Window 系统中，称为构件，即 Widget），从而大大降低了图形用户界面应用程序的开发难度。

#### 基本概念

要理解 MiniGUI 工作原理，首先要理解 GUI 中的六个基本概念。

**界面**

界面的本质就是一块画板，也是用户看见的基本单位。界面可以直接由一张图片填充而成，也可以由程序像画画一样在界面上生成一些图形。如果画面一直保持不动，那就是静态界面，如果这个画面可以根据外部事件作出改变，那就是动态界面。

**组件**

在程序在界面中绘画时，一些具备通用性元素（例如一个标签，一个按键等）可以被抽象出来，作为单独组件存在，当界面需要的时候，就可以调用这些组件在界面中显示出来。

**事件**

事件是计算机对外界感知的抽象，就像人的五感一样，计算机会将键盘，鼠标，传感器，定时闹钟等封装成事件放在计算机的特定程序中，等待或者主动的发送给其他程序，对某样事件感兴趣的程序对这个事件作出反应。举个例子，你点击了一下鼠标，计算机收到鼠标点击的信号后会将这个信号封装成一个事件，然后将事件发送到某个程序中，程序会根据这个事件作出预规划的动作，这个动作或许是打开一个网页，或许是播放一个视频等等。

**生命周期**

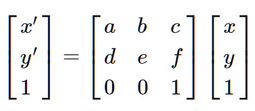
生命周期是对程序的生命过程的抽象，同时生命周期抽象能够更方便开发者编写程序。

**窗口**

窗口是对界面的组织和管理，类似于进程一样，是 GUI 划分资源的基本独立单位。GUI 系统中的 GPU 算力，可视画面中的界面位置，都是以窗口为单位管理和分配的。在一个窗口系统中，一般会有一个主窗口，主窗口也负责建立整个 GUI 系统的可视界面和可视背景。

**渲染**

画图对人来说是一个体力活，对于计算机来说也不例外。在理想情况下讨论变换一个最常见的 1080p 图像需要的算力，以图像学的基础算法仿射变换为例：



这个矩阵对应的是一个像素点的计算，需要 4 次乘法，4 次加法。计算出两张图二维像素坐标在一维空间内的索引，需要两次乘法四次加法。经过 RGB 三个字节搬运。也就是处理一个像素需要 6 次乘法，8 次加法，三次内存搬运。一张 1080p 的图有 1920 × 1080 = 2073600 个像素，也就是需要 35251200 个指令。作为参考，嵌入式常用的 Arm Cortex-A7 内核设计性能是 1.9 DMIPS/MHz range（DMIPS：Dhrystone Million Instructions executed Per Second，每秒执行百万条指令，用来计算同一秒内系统的处理能力，即每秒执行了多少百万条指令）。设定 CPU 频率是 650 Mhz 时，处理这张图大约需要 27 ms。这只是单纯的核心处理需要的耗时，在实际中，IO 延迟，C 语言子函数的调用开销等等加起来的耗时是远远大于 27 ms 的。所以靠 CPU 去绘画非常耗时占用 CPU 的。对于绘图任务来说，就是单纯的计算加内存搬运，是简单的机械劳动。为了应对这种巨量的简单机械劳动，一种新的 GPU 处理器被开发出来。GPU 强化了 CPU 的计算和内存搬运功能，减弱或者削减了控制功能，能够在 CPU 的指导下完成图像学中简单重复的功能。同时这种指导 GPU 绘画的过程被称为渲染。

#### 安装

在 ubuntu 22.04 平台上编译运行 MiniGUI 5.0 。

安装软件依赖包

sudo apt install git g++ binutils autoconf automake libtool make cmake pkg-config  
sudo apt install libgtk2.0-dev  
sudo apt install libjpeg-dev libpng-dev libwebp-dev libfreetype-dev libharfbuzz-dev  
sudo apt install libinput-dev libdrm-dev libsqlite3-dev libxml2-dev libssl-dev

下载编译包

git clone https://github.com/VincentWei/build-minigui-5.0.git

下载源码

./fetch-all.sh

编译构建依赖

./build-deps.sh

构建 MiniGUI 5.0

./build-minigui.sh

运行 demo

cd cell-phone-ux-demo/  
./mginit

运行效果：



编译过程中遇到的问题：

问题一：

witty@ubuntu:~/work/minigui/build-minigui-5.0/cell-phone-ux-demo$ ./mginit   
Gtk-Message: 17:40:16.842: Failed to load module "canberra-gtk-module"  
load\_font\_data: failed to open the font file: font/Helvetica.ttf (1)  
make\_devfont: error when loading font ttf-helvetica-rrncnn-0-0-ISO8859-1,GB2312-0,UTF-8 from font/Helvetica.ttf file  
load\_font\_data: failed to open the font file: /usr/local/share/minigui/res//font/Helvetica.ttf (1)  
make\_devfont: error when loading font ttf-helvetica-rrncnn-0-0-ISO8859-1,GB2312-0,UTF-8 from /usr/local/share/minigui/res//font/Helvetica.ttf file  
mg\_InitGDI: Can not initialize fonts defined in section truetypefonts!  
KERNEL>InitGUI (step 8): Initialization of GDI resource failure!

这是没有安装 gvfb ，需要在 gvfb 目录手动安装。

witty@ubuntu:~/work/minigui/build-minigui-5.0/gvfb/src$ sudo make install  
[100%] Built target gvfb  
Install the project...  
-- Install configuration: ""  
-- Up-to-date: /usr/local/bin/gvfb

问题二：

witty@ubuntu:~/work/minigui/build-minigui-5.0/cell-phone-ux-demo$ ./mginit   
Gtk-Message: 17:44:13.876: Failed to load module "canberra-gtk-module"  
load\_font\_data: failed to open the font file: font/Helvetica.ttf (1)  
make\_devfont: error when loading font ttf-helvetica-rrncnn-0-0-ISO8859-1,GB2312-0,UTF-8 from font/Helvetica.ttf file  
load\_font\_data: failed to open the font file: /usr/local/share/minigui/res//font/Helvetica.ttf (1)  
make\_devfont: error when loading font ttf-helvetica-rrncnn-0-0-ISO8859-1,GB2312-0,UTF-8 from /usr/local/share/minigui/res//font/Helvetica.ttf file  
mg\_InitGDI: Can not initialize fonts defined in section truetypefonts!  
KERNEL>InitGUI (step 8): Initialization of GDI resource failure!

这是缺少 Helvetica.ttf 文件。

witty@ubuntu:~/work/minigui/build-minigui-5.0/cell-phone-ux-demo$ sudo cp res/font/Helvetica.ttf /usr/local/share/minigui/res/font

#### 工作原理

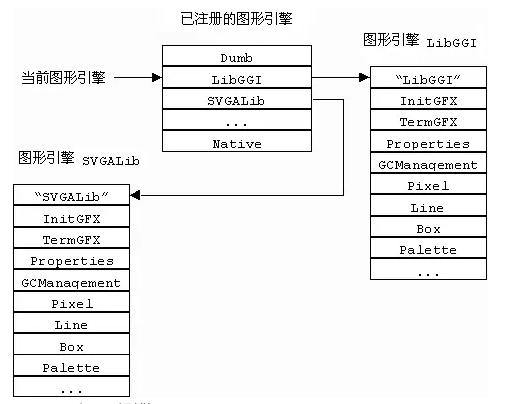
**软件抽象层**

软件是硬件既定行为的蓝图，而不同的硬件也有着不同的行为蓝图。不同的硬件厂商有着不同的硬件技术标准，不同的软件也有着自己的行为标准。这让期望能够用一套代码跑到所有设备或系统上的想法变得不可能，或多或少都要有少量调整。为了解决硬件软件的差异化的问题，软件开发者根据软件大的差异点进行切割，分为主要业务逻辑和适配逻辑，后来便演变成了应用层和兼容层。兼容层为应用层提供较为统一的软件调用，应用层负责整个的业务逻辑就可以了。

在 MiniGUI 上这个兼容层被叫做输入抽象层（IAL）和图像抽象层（GAL），分别负责图形处理引擎和 IO 的兼容。

GAL 和 IAL 的结构是类似的，以 GAL 为例说明 MiniGUI GAL 和 IAL 抽象层的结构。

下图是 GAL 图形引擎的接入解释：



系统维护了一个已注册图形引擎组，保存了每个图形引擎数据结构的指针。系统利用一个指针指向当前使用的图形引擎，从而使用图形引擎的兼容调用。对于 MiniGUI 的运行来说，系统中要存在两个图形引擎，一个是空引擎，保证系统运行不会出错，当然也不进行任何实际的图形输出。另一个是实际要使用的图形引擎，例如 LibGGI 、SVGALib 、Native Engine 等，这是 MiniGUI 工作而使用的引擎。 每个已经注册的图形引擎数据结构中都描述了这个引擎的必要信息，同时定义了一套同名的任务接口，包括生命周期的初始化和终止、图形的上下文管理、画点任务、画线任务、矩形框填充任务、调色板任务等等供 MiniGUI 进行调用。

当然，如果运行的平台所使用的图形硬件比较特殊，现有的图形引擎均不支持。这时只需要我们按照 GAL 所定义的接口实现自己的图形引擎，并指定 MiniGUI 使用这种私有的图形引擎即可。

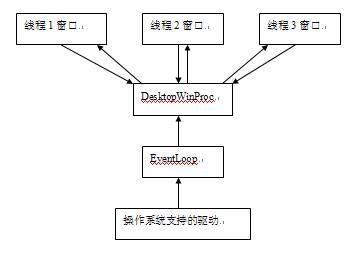
通过 GAL 和 IAL 兼容层，提高了 MiniGUI 的可移植性和快速开发。开发者可以在桌面平台 X window 上开发和调试 MiniGUI 程序，只需要重新编译并指定底层引擎便能在特殊的嵌入式平台上运行。

**基本结构**

MiniGUI 有三种工作模式，对应的多种不同的编程模型。

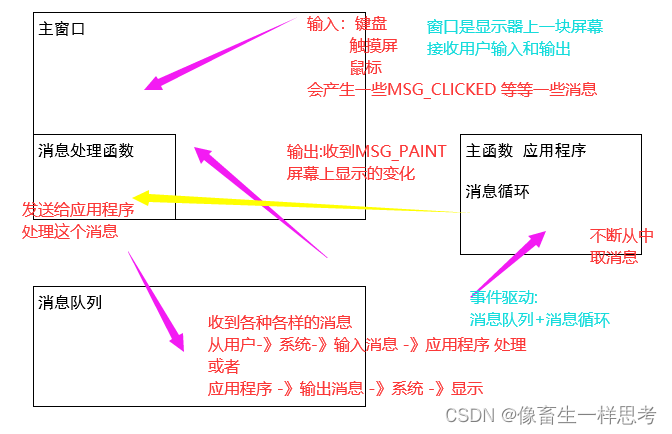
* MiniGUI-Threads 线程模式：可以工作在 RTOS 系统中，如 uC/OS、VxWorks、uClinux、Linux
* MiniGUI-Processes 进程模式：主要工作在嵌入式 Linux 系统中，优点是提高窗口的稳定性。
* MiniGUI-Standalone 独立模式：主要工作在裸机的单片机系统中。

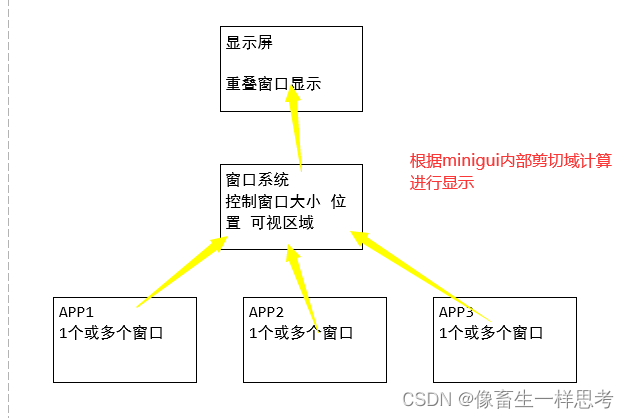
系统是由不同的线程协作完成。系统启动时通过 SystemThreads() 开启两个核心任务线程 DesktopMain()、EventLoop()。



在 EventLoop() 中，通过 Linux 系统调用 select() 监控底层驱动的输入，将键盘、鼠标等外设输入信号封装成时间，然后以消息的形式发送到线程 DesktopMain。 DesktopMain 线程的 DesktopWinProc() 函数负责读取这些事件，然后根据事件发生的位置等信息，转发到对应的线程窗口。DesktopMain 线程是一个服务线程，主要的任务是完成系统任务的处理，接收底层的事件并把事件转发到相应的窗口线程，由窗口线程来完成事件任务。

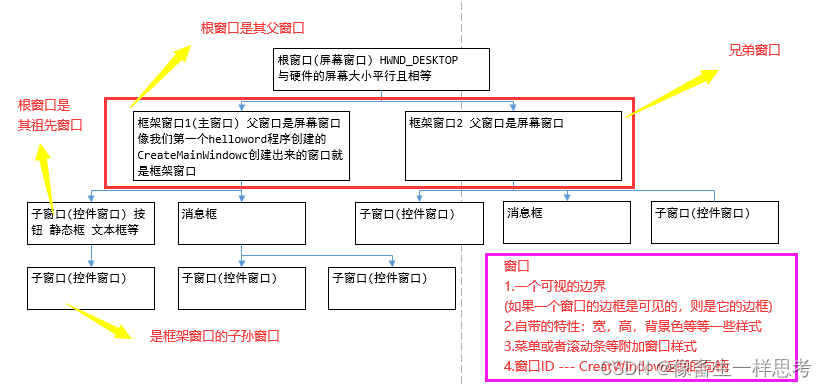
**窗口和消息**





MiniGUI 窗口的特性：

* 窗口的组织:层次结构形式(树的行书)
* 根窗口(桌面窗口,占满整个屏幕 HWND\_DESKTOP 这是minigui程序自己创建出来的 用于处理一些系统事务) --- 所有窗口的祖先
* 除了根窗口，所有窗口都有父窗口
* 父窗口可见，子窗口才可见
* 框架窗口(一般为根窗口的子窗口) = 客户区+非客户区(窗口系统管理的修饰区)
* 窗口的种类:主窗口 + 控件(子窗口) + 对话框

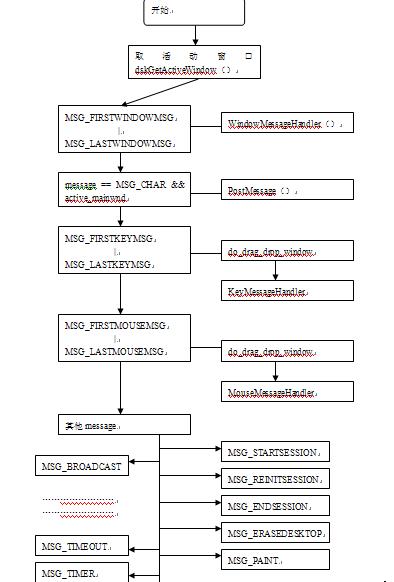


主窗口特性：



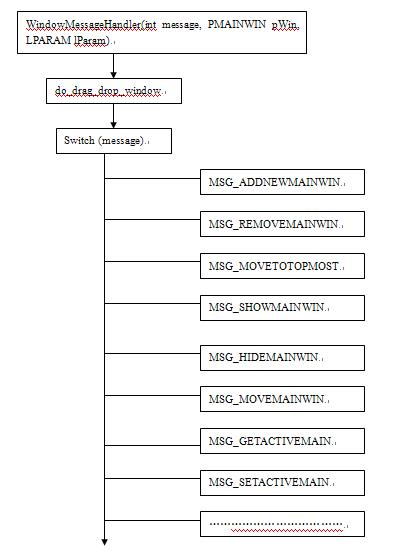
**DesktopMain 主服务函数**

DesktopMain 的主循环函数是 DesktopWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)。下面是它的执行流程图：



**WindowMessageHandler 消息服务函数**

该函数的主要功能是检查消息队列中是否有消息，然后根据消息类型进行相应的处理。



**MiniGUI 系统的启动过程**

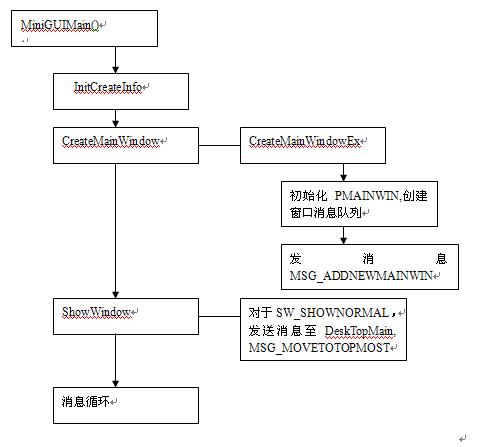
#define MiniGUIMain \  
 MiniGUIAppMain (int args, const char\* argv[]);  
  
int main\_entry (int args, const char\* argv[])  
{  
 int iRet = 0;  
 if (InitGUI (args, argv) != 0) {  
 return 1;  
 }  
 iRet = MiniGUIAppMain (args, argv);  
 TerminateGUI (iRet);  
 return iRet;  
}

InitGUI 的具体过程：介绍介绍

// 初始化 GUI  
int GUIAPI InitGUI(int args, const char \*argv[]) {  
 char engine[LEN\_ENGINE\_NAME + 1]; // 图形引擎名称  
 char mode[LEN\_VIDEO\_MODE + 1]; // 视频模式名称  
 int step = 0; // 初始化步骤计数器  
 MSGQUEUE \*msg\_queue; // 消息队列指针  
  
#ifdef HAVE\_SETLOCALE  
 setlocale(LC\_ALL, "C"); // 设置本地化为 "C"  
#endif  
  
#if defined(\_USE\_MUTEX\_ON\_SYSVIPC) || defined(\_USE\_SEM\_ON\_SYSVIPC)  
 step++;  
 if (\_sysvipc\_mutex\_sem\_init())  
 return step;  
#endif  
  
#ifndef \_\_NOUNIX\_\_  
 // 保存原始 termio  
 tcgetattr(0, &savedtermio);  
#endif  
  
 /\* 初始化默认窗口过程 \*/  
 \_\_mg\_def\_proc[0] = PreDefMainWinProc;  
 \_\_mg\_def\_proc[1] = PreDefDialogProc;  
 \_\_mg\_def\_proc[2] = PreDefControlProc;  
#ifdef \_MGHAVE\_VIRTUAL\_WINDOW  
 \_\_mg\_def\_proc[3] = PreDefVirtualWinProc;  
#endif  
  
 step++;  
 if (!mg\_InitSliceAllocator()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 初始化切片分配器失败！\n");  
 return step;  
 }  
  
 step++;  
 if (!mg\_InitFixStr()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 初始化固定字符串堆失败！\n");  
 return step;  
 }  
  
 step++;  
 /\* 初始化其他模块 \*/  
 if (!mg\_InitMisc()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 初始化杂项模块失败！\n");  
 return step;  
 }  
  
 step++;  
 switch (mg\_InitGAL(engine, mode)) {  
 case ERR\_CONFIG\_FILE:  
 \_ERR\_PRINTF("KERNEL>InitGUI: 读取配置文件失败！\n");  
 return step;  
  
 case ERR\_NO\_ENGINE:  
 \_ERR\_PRINTF("KERNEL>InitGUI: 未定义图形引擎！\n");  
 return step;  
  
 case ERR\_NO\_MATCH:  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法获取图形引擎信息！\n");  
 return step;  
  
 case ERR\_GFX\_ENGINE:  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化图形引擎！\n");  
 return step;  
 }  
  
#ifndef \_\_NOUNIX\_\_  
 InstallSEGVHandler();  
#endif  
  
 /\* 初始化 GDI \*/  
 step++;  
 if (!mg\_InitGDI()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: GDI 初始化失败！\n");  
 goto failure1;  
 }  
  
 /\* 初始化主屏幕 DC \*/  
 step++;  
 if (!mg\_InitScreenDC()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化屏幕 DC！\n");  
 goto failure1;  
 }  
  
 /\*  
 \* 在此加载系统资源。  
 \*/  
 step++;  
 if (!mg\_InitSystemRes()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化系统资源！\n");  
 goto failure1;  
 }  
  
 \_\_mg\_license\_create();  
 \_\_mg\_splash\_draw\_framework();  
  
 /\* 初始化鼠标光标 \*/  
 step++;  
 if (!mg\_InitCursor()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化鼠标光标！\n");  
 goto failure1;  
 }  
 \_\_mg\_splash\_progress();  
  
 /\* 初始化低级事件 \*/  
 step++;  
 if (!mg\_InitLWEvent()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化低级事件！\n");  
 goto failure1;  
 }  
 \_\_mg\_splash\_progress();  
  
 /\*\* 初始化 LF 管理器 \*/  
 step++;  
 if (!mg\_InitLFManager()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化 LF 管理器！\n");  
 goto failure;  
 }  
 \_\_mg\_splash\_progress();  
  
#ifdef \_MGHAVE\_MENU  
 /\* 初始化菜单 \*/  
 step++;  
 if (!mg\_InitMenu()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化菜单模块！\n");  
 goto failure;  
 }  
#endif  
  
 /\* 初始化控件类 \*/  
 step++;  
 if (!mg\_InitControlClass()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化控件类！\n");  
 goto failure;  
 }  
 \_\_mg\_splash\_progress();  
  
 /\* 初始化加速器 \*/  
 step++;  
 if (!mg\_InitAccel()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化加速器！\n");  
 goto failure;  
 }  
 \_\_mg\_splash\_progress();  
  
 step++;  
 if (!mg\_InitDesktop()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化桌面！\n");  
 goto failure;  
 }  
 \_\_mg\_splash\_progress();  
  
 step++;  
 if (!mg\_InitFreeQMSGList()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化空闲 QMSG 堆！\n");  
 goto failure;  
 }  
 \_\_mg\_splash\_progress();  
  
 step++;  
 if (!createThreadInfoKey()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化线程键！\n");  
 goto failure;  
 }  
  
 \_\_mg\_splash\_delay();  
  
 \_is\_minigui\_running = 1;  
  
 step++;  
 if (!SystemThreads()) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法初始化系统线程！\n");  
 goto failure;  
 }  
  
 /\* 初始化主 GUI 线程的消息队列 \*/  
 step++;  
 if (!(msg\_queue = mg\_AllocMsgQueueForThisThread(TRUE))) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法分配消息队列！\n");  
 goto failure;  
 }  
  
 /\* 初始化用于计时器的计数器 \*/  
 step++;  
 if (!mg\_InitTimer(FALSE)) {  
 \_ERR\_PRINTF("KERNEL>InitGUI: 无法启动计时器！\n");  
 goto failure;  
 }  
  
 SetKeyboardLayout("default");  
  
 SetCursor(GetSystemCursor(IDC\_ARROW));  
  
 SetCursorPos(g\_rcScr.right >> 1, g\_rcScr.bottom >> 1);  
  
 mg\_TerminateMgEtc();  
 return 0;  
  
failure:  
 mg\_TerminateLWEvent();  
failure1:  
 mg\_TerminateGAL();  
 \_ERR\_PRINTF("KERNEL>InitGUI: 初始化失败，请检查您的 MiniGUI 配置或资源。\n");  
 return step;  
}

**线程窗口模型**

下面是线程窗口的主要运行结构：

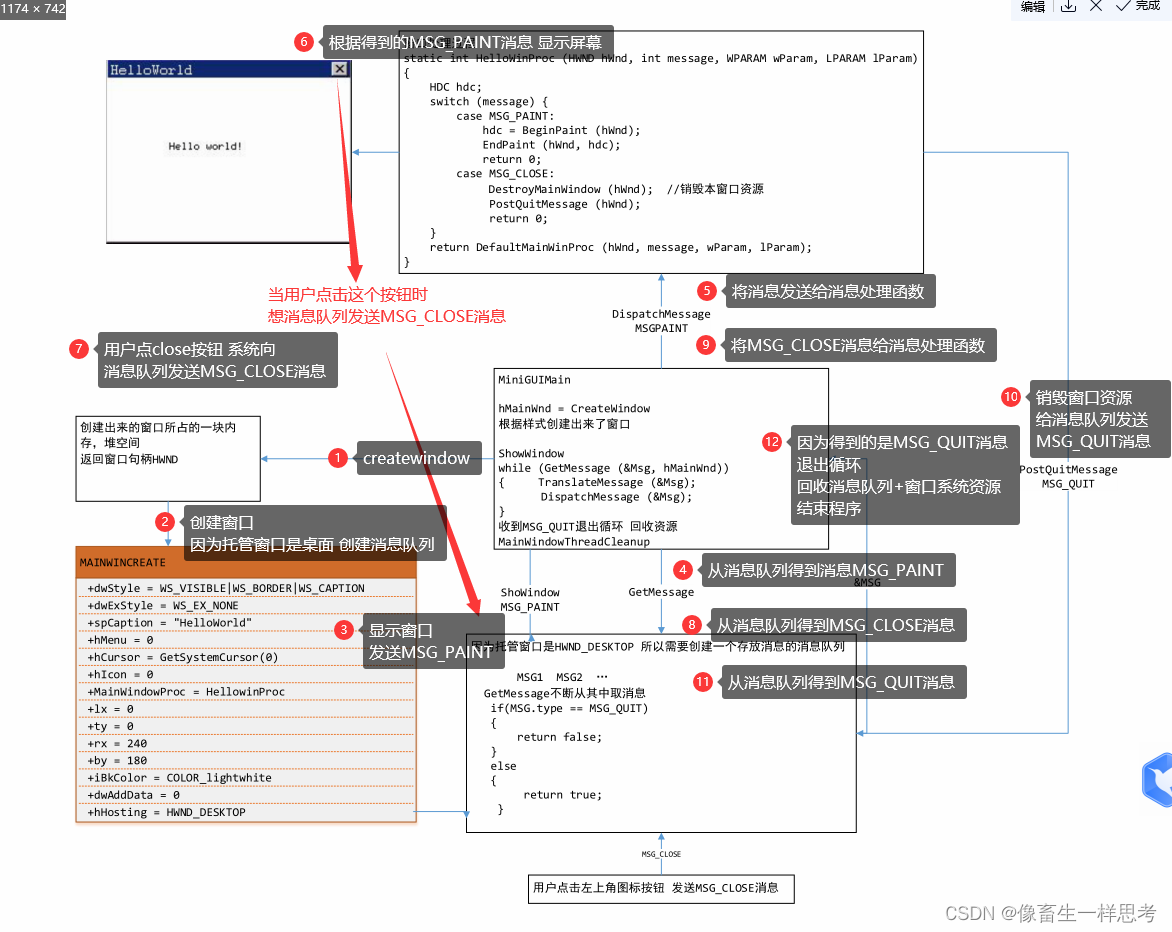


线程串口启动后，先出初始化创建信息，在 CreateMainWindow 中完成窗口的创建。窗口创建完成后，使用 ShowWindows 函数将窗口展示出来，最后进行消息处理循环。

#### hello word 程序示例介绍

#include <stdio.h> //c库函数  
#include <minigui/common.h> //minigui常用的宏和定义  
#include <minigui/minigui.h> //minigui通用的函数和杂项的函数  
#include <minigui/gdi.h> //绘图函数接口定义  
#include <minigui/window.h> //窗口相关的宏 数据结构 数据类型 以及接口函数的说明  
  
//消息过程处理函数  
static int HelloWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)  
{  
 HDC hdc;  
 switch (message) {  
 case MSG\_PAINT: //绘图消息 比如说ShowWindow这个函数调用时会这个消息  
 hdc = BeginPaint (hWnd);//获取绘图上下文  
 TextOut (hdc, 60, 60, "Hello world!");//将文本输出到屏幕上  
 EndPaint (hWnd, hdc);//释放绘图上下文  
 return 0;  
 case MSG\_CLOSE://关闭消息 像用户点击右上角图标后会发送该消息  
 DestroyMainWindow (hWnd);//销毁窗口本身用户创建出来的内存等等  
 PostQuitMessage (hWnd); //将MSG\_QUIT消息投放到消息队列中  
 return 0;  
 }  
 return DefaultMainWinProc (hWnd, message, wParam, lParam);//默认的消息处理函数  
}  
  
//像main一样的程序入口点  
int MiniGUIMain (int argc, const char\* argv[])  
{  
 MSG Msg;  
 HWND hMainWnd;  
 MAINWINCREATE CreateInfo;  
   
 #ifdef \_MGRM\_PROCESSES //进程模式下加入层的概念  
 JoinLayer (NAME\_DEF\_LAYER , "helloworld" , 0 , 0);  
 #endif  
   
 CreateInfo.dwStyle = WS\_VISIBLE | WS\_BORDER | WS\_CAPTION;//可见 有标题栏 有边框  
 CreateInfo.dwExStyle = WS\_EX\_NONE; //无扩展风格  
 CreateInfo.spCaption = "HelloWorld"; //标题栏文本  
 CreateInfo.hMenu = 0; //无菜单  
 CreateInfo.hCursor = GetSystemCursor (0); //系统默认光标  
 CreateInfo.hIcon = 0; //无图标 像windows窗口左上角的那个小图标  
 CreateInfo.MainWindowProc = HelloWinProc; //默认窗口处理函数  
 CreateInfo.lx = 0; //左上x left x  
 CreateInfo.ty = 0; //左上y top y  
 CreateInfo.rx = 240;//右下x right x  
 CreateInfo.by = 180;//右下y bottom y 这四个值可表示一个矩形 左上坐标(0,0) w=240 h=180的矩形  
 CreateInfo.iBkColor = COLOR\_lightwhite; //背景颜色 白色  
 CreateInfo.dwAddData = 0; //窗口数据 无  
 CreateInfo.hHosting = HWND\_DESKTOP; //托管窗口是桌面  
   
 hMainWnd = CreateMainWindow (&CreateInfo); //创建一个窗口 用户程序资源+系统资源  
 if (hMainWnd == HWND\_INVALID)  
 return -1;  
 ShowWindow (hMainWnd, SW\_SHOWNORMAL);//显示一个窗口  
   
 //消息循环队列  
 while (GetMessage (&Msg, hMainWnd)) {//从消息队列中获取消息  
 TranslateMessage (&Msg); //将键盘消息转换为字符消息  
 DispatchMessage (&Msg); //将消息发送给消息处理函数  
 }  
   
 MainWindowThreadCleanup (hMainWnd); //回收窗口的资源 像消息队列+系统资源等等  
 return 0;  
}  
#ifndef \_MGRM\_PROCESSES  
#include <minigui/dti.c>  
#endif

流程图介绍：



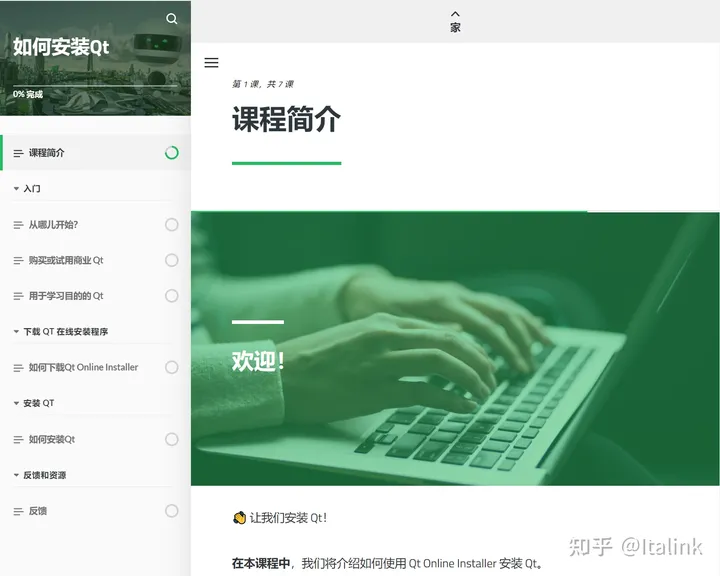
## 3. QT

### 简介

Qt 是一个跨平台的 C++ 框架（C++ 库），主要用来开发图形用户界面（Graphical User Interface，GUI）程序，也可以开发不带界面的命令行（Command User Interface，CUI）程序。

Qt 虽然是一个 GUI 库，基本功能是用来开发图形界面应用程序，但这并不是 Qt 的全部；Qt 除了可以绘制漂亮的界面（包括控件、布局、交互），还包含很多其它功能，比如多线程、访问数据库、图像处理、音频视频处理、网络通信、文件操作等，这些 Qt 都已经内置了，也就是说使用 Qt API 就可以调用不同平台下的这些接口。

### 安装



这是 Qt 的官方安装教程，访问 <https://courses.qt.io/how-to-install-qt/index.html#/lessons/WDj_gIccDJa6gmhH_-3P7uC2QDaZHbYA> 便可以看到。

### 开始

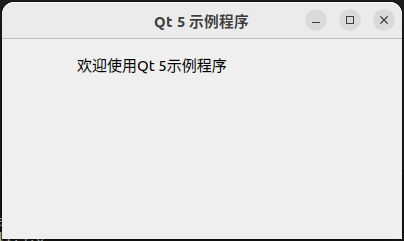
下面是一个 Qt 5 的示例程序，在同一个目录下创建 main.cpp 文件和 CMakeLists.txt 文件。

// file : main.cpp  
#include <QApplication>  
#include <QWidget>  
#include <QLabel>  
  
int main(int argc, char \*argv[]) {  
 QApplication app(argc, argv);  
  
 // 创建主窗口  
 QWidget window;  
 window.setWindowTitle("Qt 5 示例程序");  
  
 // 创建标签  
 QLabel \*label = new QLabel("欢迎使用Qt 5示例程序", &window);  
 label->setAlignment(Qt::AlignCenter);  
  
 // 调整标签的大小  
 label->resize(300, 50);  
  
 // 将标签添加到窗口中  
 label->show();  
  
 // 设置窗口的大小  
 window.resize(400, 200);  
 window.show();  
  
 return app.exec();  
}

# file : CMakeLists.txt  
cmake\_minimum\_required(VERSION 3.12)  
project(QtExample)  
  
add\_executable(QtExample main.cpp)  
  
find\_package(Qt5 COMPONENTS Core Widgets REQUIRED) #查找Qt5的模块  
  
target\_link\_libraries(QtExample #将Core和Widget链接到QtExample中  
 PRIVATE   
 Qt5::Core   
 Qt5::Widgets  
)

这是一个 Cmake 工程，所以使用 Cmake 进行编译即可：

nihao@nihao-z690:~/work/test/QT\_test$ tree  
.  
├── CMakeLists.txt  
└── main.cpp  
  
0 directories, 2 files  
nihao@nihao-z690:~/work/test/QT\_test$ cmake .  
-- The C compiler identification is GNU 11.4.0  
-- The CXX compiler identification is GNU 11.4.0  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Check for working C compiler: /usr/bin/cc - skipped  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Check for working CXX compiler: /usr/bin/c++ - skipped  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/nihao/work/test/QT\_test  
nihao@nihao-z690:~/work/test/QT\_test$ make   
[ 50%] Building CXX object CMakeFiles/QtExample.dir/main.cpp.o  
[100%] Linking CXX executable QtExample  
[100%] Built target QtExample



以下是 QApplication 的主要任务和职责：

* **应用程序的启动和初始化：** 在应用程序的main函数中，首先会创建一个 QApplication 对象，这会初始化Qt库并处理一些应用程序级别的设置，如全局样式、语言设置等。
* **事件循环管理：** QApplication 创建了一个事件循环，它负责接收、分发和处理各种事件，如鼠标点击、键盘输入、定时器事件等。这个事件循环是Qt应用程序的核心，使应用程序可以响应用户输入和系统事件。
* **窗口系统集成：** QApplication 处理与窗口系统的交互，包括创建主窗口、管理应用程序图标、处理窗口系统事件（如关闭、最小化、最大化）等。
* **全局设置和配置：** QApplication 可以设置和管理全局应用程序的一些属性，如默认字体、样式表、全局光标、应用程序名称等。
* **命令行参数处理：** QApplication 可以解析命令行参数，并将它们传递给应用程序，以便应用程序可以根据需要执行不同的操作。
* **国际化和本地化支持：** QApplication 可以处理应用程序的国际化和本地化需求，支持多语言翻译和区域设置。
* **退出处理：** 当应用程序要退出时，QApplication 负责清理资源、发送退出信号，并终止事件循环，确保应用程序正常关闭。
* **系统托盘图标支持：** QApplication 允许应用程序在系统托盘区域显示图标，以便用户可以最小化应用程序到后台运行。
* **剪贴板管理：** QApplication 允许应用程序读取和写入剪贴板内容，以支持剪贴板操作。

QApplication 继承自 QGuiApplication，它来自于 **Widgets模块** ，如果仅需要控制台程序，而非窗口，那么使用 **Core模块** 中的 **[QCoreApplication](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qcoreapplication.html)** 就够了

在实际开发应用时，可以派生QApplication，在其中添加一些全局的管理操作

QApplication::exec()将启动Qt的事件循环，它等价于：

// file : main.cpp  
#include <QApplication>  
#include <QWidget>  
#include <QLabel>  
  
int main(int argc, char \*argv[]) {  
 QApplication app(argc, argv);  
  
 // 创建主窗口  
 QWidget window;  
 window.setWindowTitle("Qt 5 示例程序");  
  
 // 创建标签  
 QLabel \*label = new QLabel("欢迎使用Qt 5示例程序", &window);  
 label->setAlignment(Qt::AlignCenter);  
  
 // 调整标签的大小  
 label->resize(300, 50);  
  
 // 将标签添加到窗口中  
 label->show();  
  
 // 设置窗口的大小  
 window.resize(400, 200);  
 window.show();  
 while (true) { // QApplication内部有一个退出信号，可惜它是私有的，这里就只能作为死循环了  
 QApplication::processEvents(); // 注释它,你会发现窗口无法再响应交互事件  
 }  
 return 0;  
}

在开发过程中，可能经常会执行一些比较慢的操作，导致界面卡顿，这个时候要么将耗时操作放在其他线程中或者也可以在这些操作中间手动调用一下QApplication::processEvents()让界面能够刷新。

### Qt 主要模块

Qt 开发框架中有 14 个主要模块，对应着抽象应用中 14 种不同的功能。

* Qt Core：提供了基本的非GUI功能，包括字符串处理、文件和目录操作、事件处理、多线程支持、定时器等。
* Qt GUI：用于创建丰富的图形用户界面，提供了窗口、布局、绘图、事件处理、输入处理等基本的图形界面功能。
* Qt Widgets：提供了一套丰富的用户界面控件，如按钮、文本框、列表框、表格、菜单等，可以用于构建传统的桌面应用程序。
* Qt Quick：提供了一种声明性的语言（QML）和相应的C++ API，用于创建现代化的用户界面，支持动画、效果、多点触摸等。
* Qt Network：提供了网络编程相关的类和功能，包括HTTP、TCP、UDP、FTP等协议的支持，可以进行网络通信和数据传输。
* Qt SQL：提供了数据库访问的API，支持主流的关系型数据库，如MySQL、SQLite、PostgreSQL等。
* Qt Multimedia：用于处理多媒体内容，包括音频、视频的播放和录制，支持多种格式和编解码器。
* Qt WebEngine：基于Chromium的Web引擎，用于在应用程序中嵌入Web内容，支持HTML5、CSS、JavaScript等。
* Qt XML：提供了XML解析和生成的功能，用于处理和操作XML格式的数据。
* Qt Bluetooth：提供了蓝牙通信的功能，可以在应用程序中实现蓝牙设备的连接和数据交互。
* Qt QML（Qt Meta-Object Language）是一种基于声明性语法的编程语言，用于构建跨平台的用户界面。它是Qt框架的一部分，用于开发富有交互性和动态性的应用程序界面。
* Qt Concurrent，提供了一套用于并行编程的工具和类。它简化了多线程和并行任务的处理，使开发者能够更轻松地编写并行代码。
* QT OpenGL：提供了在应用程序中使用OpenGL和OpenGL ES进行3D图形渲染的功能。它允许开发者在Qt应用程序中创建和显示OpenGL场景，以及与图形进行交互。
* QT Test：提供了单元测试框架和模拟鼠标和键盘事件的功能。它可以帮助开发者编写和执行单元测试，以确保代码的正确性和稳定性。此外，Qt Test还提供了与Visual Studio和KDevelop等集成开发环境的整合支持。

**Qt Core**

Qt Core 是 Q 中提供许多基本的非 GUI 功能和工具的核心模块。它包含了一系列的类和函数，用于处理字符串、日期和时间、文件和目录操作、事件处理、多线程支持、定时器等。

以下是 Qt Core 模块的一些主要特性和功能：

1. 字符串和文本处理：Qt Core提供了 QString 类，用于处理 Unicode 字符串，支持字符串的拼接、查找、替换等操作，以及字符串的编码和解码。
2. 容器类：Qt Core 提供了许多容器类，如 QList、QVector、QMap、QHash 等，用于存储和管理数据集合，支持快速查找、插入和删除等操作。
3. 文件和目录操作：Qt Core 提供了 QFile 和 QDir 类，用于进行文件和目录的读写、复制、移动、删除等操作，以及获取文件和目录的属性信息。
4. 日期和时间：Qt Core 提供了 QDate、QTime 和 QDateTime 类，用于处理日期和时间，支持日期和时间的格式化、比较、计算等操作，以及时区的处理。
5. 事件处理和信号与槽机制：Qt Core 提供了事件处理机制，允许开发者对事件进行捕获和处理。此外，Qt Core 还引入了信号与槽机制，用于实现对象之间的通信和交互。
6. 多线程支持：Qt Core 提供了多线程编程的支持，包括线程的创建、同步和通信等功能。通过 QThread 类和 QMutex 类等，可以方便地实现多线程应用程序。
7. 定时器和延时：Qt Core 提供了 QTimer 类，用于实现定时器功能，可以定时触发事件或执行特定的任务。此外，还提供了 QThread::sleep() 函数，用于实现延时操作。

**Qt GUI**

Qt GUI是Qt中用于创建丰富的图形用户界面（GUI）应用程序的重要模块。它提供了一系列的类和函数，用于窗口管理、布局、绘图、事件处理、输入处理等。

以下是Qt GUI模块的一些主要特性和功能：

1. 窗口和部件（Widgets）：Qt GUI提供了QWidget类作为所有GUI部件的基类，开发者可以使用QWidget及其子类创建窗口和部件，如QMainWindow、QDialog、QPushButton、QLabel等。
2. 布局管理：Qt GUI提供了多种布局管理器，用于自动调整和排列部件，包括QHBoxLayout、QVBoxLayout、QGridLayout等。开发者可以利用这些布局管理器来实现灵活的用户界面布局。
3. 绘图和绘制：Qt GUI提供了绘图相关的类和功能，用于在窗口上进行绘图和绘制，包括QPainter、QPen、QBrush等。开发者可以使用这些类来实现自定义的绘图和图形效果。
4. 事件处理：Qt GUI提供了事件处理机制，允许开发者对窗口和部件的事件进行捕获和处理。通过重写事件处理函数，开发者可以实现对鼠标事件、键盘事件、焦点事件等的响应。
5. 输入处理：Qt GUI提供了输入处理的支持，包括键盘输入和鼠标输入。开发者可以通过重写键盘事件和鼠标事件的处理函数，实现对用户输入的响应和处理。
6. 样式和主题：Qt GUI提供了样式和主题的支持，允许开发者自定义应用程序的外观和风格。通过QStyleSheet类和QStyle类，开发者可以定义应用程序的外观属性、颜色、字体等。
7. 国际化和本地化：Qt GUI提供了国际化和本地化的支持，允许开发者将应用程序进行多语言支持。通过QTranslator类和国际化相关函数，开发者可以实现对不同语言的切换和翻译。

**Qt Widgets**

Qt Widgets是Q中用于创建传统的桌面应用程序的图形用户界面（GUI）的模块。它提供了一系列的部件（Widgets），如按钮、文本框、列表框、表格、菜单等，供开发者使用。

以下是Qt Widgets模块的一些主要特性和功能：

1. 部件（Widgets）：Qt Widgets模块提供了许多常用的GUI部件类，如QPushButton、QLineEdit、QListWidget、QTableWidget、QMenuBar等。开发者可以使用这些部件类来构建应用程序的用户界面。
2. 布局管理器：Qt Widgets提供了多种布局管理器，如QHBoxLayout、QVBoxLayout、QGridLayout等，用于自动调整和排列部件。开发者可以利用这些布局管理器来实现灵活的用户界面布局。
3. 事件处理：Qt Widgets提供了事件处理机制，允许开发者对部件的事件进行捕获和处理。通过重写事件处理函数，开发者可以实现对鼠标事件、键盘事件、焦点事件等的响应。
4. 样式和主题：Qt Widgets提供了样式和主题的支持，允许开发者自定义应用程序的外观和风格。通过QStyleSheet类和QStyle类，开发者可以定义应用程序的外观属性、颜色、字体等。
5. 信号与槽机制：Qt Widgets支持信号与槽机制，用于实现部件之间的通信和交互。开发者可以连接部件的信号（如按钮点击）到槽函数，实现对应的响应和处理。
6. 绘图和绘制：Qt Widgets提供了绘图相关的类和功能，用于在部件上进行绘图和绘制，如QPainter、QPen、QBrush等。开发者可以使用这些类来实现自定义的绘图和图形效果。

**Qt Quick**

Qt Quick是Qt框架中用于快速创建现代、流畅的用户界面（UI）的模块。它基于Qt的核心技术，并结合了声明式语法和可编程的JavaScript，使开发者能够更加高效地构建跨平台的应用程序。

以下是Qt Quick模块的一些主要特性和功能：

1. QML语言：Qt Quick使用QML（Qt Meta-Object Language）作为界面描述语言，它是一种声明式语言，类似于JSON和JavaScript。开发者可以使用QML来描述应用程序的用户界面、交互逻辑和视觉效果。
2. 快速创建UI：Qt Quick提供了一系列的可重用的UI组件，如按钮、文本框、列表视图、图像视图等。开发者可以通过组合这些组件来快速创建和定制应用程序的UI。
3. 动态视觉效果：Qt Quick支持丰富的动画和过渡效果，开发者可以通过简单的声明式语法来实现界面元素的平滑动画、渐变效果、旋转等，为应用程序增加流畅的交互体验。
4. 事件处理和信号槽机制：Qt Quick支持事件处理和信号槽机制，开发者可以通过JavaScript来编写响应事件的代码，并与QML界面进行交互。这使得开发者能够实现用户交互和应用逻辑的连接。
5. 2D图形和渲染：Qt Quick内置了强大的2D图形和渲染引擎，支持绘制矢量图形、图像处理、图形效果等。开发者可以利用这些功能来创建精美的用户界面和图形效果。
6. 跨平台支持：Qt Quick可用于多个平台，包括桌面、移动设备和嵌入式系统。开发者可以使用相同的代码和UI描述来构建适应不同平台的应用程序。

**Qt Network**

Qt NetworkQt中用于处理网络通信和网络操作的模块。它提供了一系列的类和功能，使开发者能够轻松地进行网络编程，包括网络请求、服务器和客户端的实现、数据传输等。

以下是Qt Network模块的一些主要特性和功能：

1. 网络通信：Qt Network模块提供了QTcpSocket和QUdpSocket等类，用于实现TCP和UDP协议的网络通信。开发者可以使用这些类来建立网络连接、发送和接收数据，以及处理网络错误和异常。
2. HTTP请求：Qt Network模块支持HTTP协议，提供了QNetworkAccessManager类，用于发送和接收HTTP请求。开发者可以使用该类来执行GET、POST等HTTP请求，并处理服务器返回的响应数据。
3. FTP和文件下载：Qt Network模块支持FTP协议，开发者可以使用QFtp类来实现FTP客户端，进行文件的上传和下载。此外，Qt Network还提供了QNetworkAccessManager类的下载功能，可以方便地进行大文件的下载操作。
4. SSL加密：Qt Network模块支持SSL和TLS协议，允许开发者进行安全的网络通信。通过QSslSocket类，开发者可以建立安全的加密连接，并实现对数据的加密传输和服务器身份验证。
5. DNS解析：Qt Network模块提供了QHostInfo类，用于进行DNS解析。开发者可以使用该类来查询主机名对应的IP地址，或者反向查询IP地址对应的主机名。
6. 网络代理：Qt Network支持网络代理，允许开发者通过代理服务器进行网络连接。开发者可以配置代理设置，以便在需要时使用代理进行网络通信。

**Qt SQL**

Qt SQL是Qt中用于在应用程序中进行数据库操作的模块。它提供了一系列的类和功能，使开发者能够方便地连接、查询和操作各种不同类型的数据库。

以下是Qt SQL模块的一些主要特性和功能：

1. 支持的数据库：Qt SQL模块支持多种流行的数据库系统，包括SQLite、MySQL、PostgreSQL和Oracle等。开发者可以根据需要选择所需的数据库，并使用相应的数据库驱动程序进行连接和操作。
2. 数据库连接管理：Qt SQL提供了QSqlDatabase类，用于管理数据库连接。开发者可以使用该类来建立和关闭数据库连接，设置连接参数，如主机名、用户名、密码等。
3. SQL查询和事务：Qt SQL模块提供了QSqlQuery类，用于执行SQL查询语句。开发者可以使用该类来执行查询、插入、更新和删除操作，并通过结果集获取查询结果。此外，Qt SQL还支持事务处理，开发者可以通过QSqlDatabase类来管理和控制事务的提交和回滚。
4. 数据库模型和视图：Qt SQL提供了QSqlTableModel和QSqlQueryModel等类，用于在Qt的视图组件（如QTableView）中显示数据库表格数据。这些类可以方便地将数据库中的数据与界面进行绑定，并支持数据的排序、过滤和编辑。
5. SQL语句预处理：Qt SQL支持SQL语句的预处理，通过使用绑定参数的方式，可以提高查询的效率和安全性。开发者可以使用QSqlQuery类的绑定函数来设置查询参数，避免SQL注入等安全问题。
6. 数据库驱动程序插件：Qt SQL模块通过插件系统支持各种不同类型的数据库驱动程序。开发者可以根据需要选择和加载所需的数据库驱动程序，以便与特定的数据库系统进行通信。

**Qt Multimedia**

Qt Multimedia是Qt中用于处理多媒体内容的模块，包括音频和视频的播放、录制和处理。它提供了一系列的类和功能，用于多媒体的处理和控制。

以下是Qt Multimedia模块的一些主要特性和功能：

1. 音频和视频播放：Qt Multimedia模块提供了QMediaPlayer类，用于播放音频和视频文件。开发者可以使用该类来控制媒体的播放、暂停、停止等操作，以及获取媒体的播放状态和元数据信息。
2. 音频和视频录制：Qt Multimedia模块提供了QAudioRecorder和QVideoRecorder类，用于录制音频和视频。开发者可以使用这些类来控制录制设备的选择、开始录制、停止录制等操作，以及获取录制的音频和视频数据。
3. 音频和视频处理：Qt Multimedia模块提供了一些功能类，用于音频和视频的处理和变换。例如，QAudioProbe类可以用于分析音频数据，QVideoFilterRunnable类可以用于实现自定义的视频滤镜。
4. 媒体格式和编解码器支持：Qt Multimedia模块支持多种常见的媒体格式和编解码器，包括但不限于MP3、WAV、OGG、H.264、MPEG等。开发者可以使用Qt Multimedia模块来播放和录制这些格式的媒体内容。
5. 音频和视频设备访问：Qt Multimedia模块提供了QAudioDeviceInfo和QCameraInfo类，用于获取音频和视频设备的信息，如设备名称、支持的采样率、分辨率等。开发者可以使用这些类来选择和配置合适的设备。

**Qt WebEngine**

Qt WebEngine是Qt中用于在应用程序中集成Web浏览器功能的模块。它基于Chromium项目，提供了用于呈现和交互Web内容的API和工具。

以下是Qt WebEngine模块的一些主要特性和功能：

1. 嵌入式Web浏览器：Qt WebEngine模块允许开发者在Qt应用程序中嵌入一个完整的Web浏览器。开发者可以使用Qt的UI组件（如QWebEngineView）来显示Web页面，并与页面进行交互。
2. 支持HTML5和CSS3：Qt WebEngine支持HTML5和CSS3标准，使开发者能够在应用程序中展示和操作现代的Web内容。这包括播放音频和视频、绘制2D和3D图形、处理Web表单等功能。
3. JavaScript交互：Qt WebEngine允许开发者通过JavaScript与Web页面进行交互。开发者可以通过Qt提供的API将JavaScript代码嵌入到页面中，或者从页面中获取和处理JavaScript事件和数据。
4. Web页面导航和控制：Qt WebEngine提供了API来控制Web页面的导航、加载和渲染。开发者可以加载URL、后退和前进页面、重新加载页面，以及处理页面加载进度和状态的变化。看做是一个轻量级的Window，它
5. Cookie和存储管理：Qt WebEngine模块提供了管理和操作Web页面的Cookie和本地存储的功能。开发者可以读取和设置Cookie，以及管理Web页面使用的本介绍地存储（如Web Storage和IndexedDB）。
6. 安全性和隐私：Qt WebEngine模块集成了Chromium的安全性和隐私功能，包括安全的网络通信（如HTTPS）、安全证书验证、跨站点脚本保护等。这有助于保护用户数据和提供安全的Web浏览体验。

**Qt XML**

Qt XML是Qt框架中用于处理XML（可扩展标记语言）的解析和生成模块。它提供了一组类和函数，使开发者能够方便地读取和写入XML文档，并进行XML数据的处理和操作。

以下是Qt XML模块的一些主要特性和功能：

1. XML解析：Qt XML模块提供了QXmlStreamReader和QXmlStreamWriter等类，用于解析和读取XML文档。开发者可以使用QXmlStreamReader类逐行读取XML文档，并从中提取元素、属性和文本等内容。而QXmlStreamWriter类则用于生成XML文档，开发者可以使用它来创建XML元素、属性和文本，并将其写入到XML文件或其他输出设备中。
2. SAX和DOM解析：Qt XML模块支持SAX（Simple API for XML）和DOM（Document Object Model）两种常用的XML解析方式。通过使用QXmlStreamReader，开发者可以实现SAX方式的解析，以事件驱动的方式逐行解析XML文档。而通过使用QDomDocument类，开发者可以实现DOM方式的解析，将整个XML文档加载到内存中，并以树状结构进行访问和操作。
3. XML验证：Qt XML模块支持XML文档的验证，可以根据预定义的XML模式（如DTD或XML Schema）对XML文档进行验证。开发者可以使用QXmlSchema类加载和解析XML模式，并使用QXmlSchemaValidator对XML文档进行验证，以确保其符合定义的结构和规则。
4. 命名空间支持：Qt XML模块提供了对XML命名空间的支持。开发者可以使用QXmlStreamReader和QXmlStreamWriter类来处理具有命名空间的XML文档，包括读取和写入带有命名空间的元素和属性。

**Qt Bluetooth**

Qt Bluetooth是Qt中用于在应用程序中实现蓝牙通信功能的模块。它提供了一组类和函数，使开发者能够方便地与蓝牙设备进行连接、数据交换和控制。

以下是Qt Bluetooth模块的一些主要特性和功能：

1. 蓝牙设备发现：Qt Bluetooth模块提供了类似于扫描的功能，可以发现附近的蓝牙设备。开发者可以使用QBluetoothDeviceDiscoveryAgent类来搜索并获取蓝牙设备的信息，如设备名称、地址和服务列表等。
2. 蓝牙设备连接：Qt Bluetooth模块支持与蓝牙设备的连接和断开连接。开发者可以使用QBluetoothSocket类来创建蓝牙连接，并通过该连接发送和接收数据。
3. 数据交换：Qt Bluetooth模块提供了QBluetoothSocket类用于在蓝牙设备之间进行数据交换。开发者可以使用该类中的函数发送和接收数据，包括传输文件、传递命令和接收传感器数据等。
4. 服务和特性：蓝牙设备通常会提供一组服务和特性，用于定义其功能和行为。Qt Bluetooth模块提供了QBluetoothServiceDiscoveryAgent类来发现设备提供的介绍服务和特性，并使用QBluetoothServiceInfo类来获取和操作这些服务和特性。
5. BLE（低功耗蓝牙）支持：Qt Bluetooth模块支持BLE设备（低功耗蓝牙设备），包括BLE设备的发现、连接和数据交换。开发者可以使用QBluetoothLowEnergy类和相关类来与BLE设备进行通信。

**Qt QML**

Qt QML（Qt Meta-Object Language）是一种基于声明性语法的编程语言，用于构建跨平台的用户界面。它是Qt框架的一部分，用于开发富有交互性和动态性的应用程序界面。

以下是Qt QML的一些主要特点和概念：

1. 声明性语法：Qt QML使用一种声明性语法，类似于JavaScript和JSON。开发者可以使用QML语言描述应用程序的界面结构、布局和交互行为，而不需要编写大量的代码。
2. QML对象：QML中的界面元素被组织为对象的层次结构。每个QML对象都可以具有属性、信号和方法。开发者可以通过在QML中创建对象并设置属性来构建应用程序界面。
3. 模型-视图分离：QML支持模型-视图分离的编程模式。通过将数据模型与界面视图分离，开发者可以实现动态数据绑定、数据驱动的界面更新和交互。
4. 信号与槽机制：QML中的对象可以通过信号与槽机制进行通信。当对象的属性或状态发生变化时，它可以发出信号，其他对象可以通过连接到这些信号的槽函数来获取状态变化的通知并执行相应的操作。
5. 动画与过渡效果：QML提供了丰富的动画和过渡效果的支持。开发者可以使用内置的动画类型和过渡函数，或自定义动画来实现界面元素的平滑动态效果。
6. 与C++的集成：Qt QML可以与C++代码无缝集成。开发者可以在QML中调用C++函数、访问C++对象和属性，并使用Qt提供的信号槽机制进行跨语言的通信。

**Qt Concurrent**

Qt Concurrent是Qt中提供了一套用于并行编程的工具和类的模块。它简化了多线程和并行任务的处理，使开发者能够更轻松地编写并行代码。

以下是Qt Concurrent模块的一些主要特性和功能：

1. QFuture和QFutureWatcher：QFuture类表示一个异步操作的结果或状态，并提供了一组函数来获取结果、等待操作完成以及监视操作的状态。QFutureWatcher类用于监视一个或多个QFuture对象，并在操作完成时发出信号。
2. QtConcurrent命名空间：Qt Concurrent模块提供了一个名为QtConcurrent的命名空间，其中包含了一些用于并行编程的函数和算法。例如，开发者可以使用QtConcurrent::run函数在后台线程中执行函数，使用QtConcurrent::mapped函数对容器中的元素进行并行映射操作，使用QtConcurrent::filter函数对容器中的元素进行并行筛选等等。
3. QThreadPool：QThreadPool类是一个线程池，用于管理和调度线程的执行。开发者可以将任务提交到线程池中，线程池会自动分配线程来执行这些任务。通过使用线程池，可以更好地管理线程资源和提高并行任务的执行效率。
4. QFutureSynchronizer：QFutureSynchronizer类用于同步多个QFuture对象的执行。开发者可以使用它等待多个并行任务完成，并在所有任务完成后继续执行后续操作。
5. 并行容器算法：Qt Concurrent模块提供了一些并行容器算法，如QParallelMap、QParallelReduce、QParallelForEach等。这些算法可以在多个线程上并行执行操作，从而提高处理大量数据的效率。

**Qt OpenGL**

Qt OpenGL 是 Qt 中提供在应用程序中使看做是一个轻量级的Window，它用 OpenGL 和 OpenGL ES 进行3D图形渲染的功能的模块。它允许开发者在Qt应用程序中创建和显示OpenGL场景，以及与图形进行交互。

以下是 Qt OpenGL 模块的一些主要特性和功能：

1. OpenGL 集成：Qt OpenGL模块提供了与OpenGL和OpenGL ES的集成，使开发者能够在Qt应用程序中利用OpenGL进行3D图形渲染。开发者可以使用Qt提供的OpenGL类（如QOpenGLWidget）创建OpenGL上下文，并在其中实现自定义的渲染逻辑。
2. 跨平台支持：Qt OpenGL模块在多个平台上提供对OpenGL的支持，包括Windows、Linux和macOS等。它还支持OpenGL ES，适用于嵌入式设备和移动平台上的图形渲染。
3. 与Qt的整合：Qt OpenGL模块与Qt框架的其他模块无缝集成，开发者可以将Qt的UI组件与OpenGL场景进行结合。例如，可以在Qt的窗口中嵌入OpenGL视图，并在其中显示和操作图形。
4. 交互和事件处理：Qt OpenGL模块提供了基于Qt事件系统的交互和事件处理机制。开发者可以通过重写事件处理函数来响应鼠标、键盘和触摸等输入事件，并与OpenGL场景进行交互。
5. 其他辅助功能：QtOpenGL模块还提供了一些辅助功能，如纹理管理、着色器支持、渲染缓冲区和帧缓冲区等。这些功能可以帮助开发者更方便地进行图形渲染和效果实现。

需要注意的是，在Windows平台上，QtOpenGL模块还支持与Direct3D的集成，即可以使用Qt的API与Direct3D进行交互和渲染。

**Qt Test**

Qt Test是Qt中提供单元测试框架和模拟鼠标和键盘事件的功能的模块。它可以帮助开发者编写和执行单元测试，以确保代码的正确性和稳定性。此外，Qt Test还提供了与Visual Studio和KDevelop等集成开发环境的整合支持。

以下是Qt Test模块的一些主要特性和功能：

1. 单元测试框架：Qt Test提供了一个用于编写和执行单元测试的框架。开发者可以使用该框架定义测试用例、断言和测试函数，并运行这些测试以验证代码的正确性。Qt Test框架支持常见的测试概念，如测试夹具（test fixtures）、测试套件（test suites）和测试运行器（test runners）等。
2. 模拟鼠标和键盘事件：Qt Test提供了一组类和函数，用于模拟鼠标和键盘事件。开发者可以使用这些功能来编写测试用例，模拟用户的输入行为，以测试应用程序的交互逻辑和用户界面响应。
3. 整合支持：Qt Test与多个集成开发环境（IDE）进行了整合，包括Visual Studio和KDevelop等。这使得开发者可以方便地在这些环境中编写、运行和调试单元测试。Qt Test提供了相应的插件和工具，以便在这些IDE中集成和使用单元测试框架。
4. 自动化测试和持续集成：Qt Test支持自动化测试和持续集成流程。开发者可以将Qt Test与自动化测试工具和持续集成系统集成，以实现代码提交时的自动测试和报告生成。

### Object

在 Qt 中有一个基础元对象 QObject ，它是所有绝大多数 Qt 派生类的基类，便以统一和管理所有的子对象。 Qt 中的基础 QObject 类提供了反射、Signal-Slot、垃圾回收、事件等机制。

一个基本的 QObject 派生类定义的示例如下：

class QExmapleClass : public QObject {  
 Q\_OBJECT //Q\_OBJECT是Qt反射数据的入口，如果无需反射功能，可以不写  
};

### QMetaObject

QMetaObject 主要为 Qt 的源对象提供反射功能。反射，就是指对象成员之间的自我检查。Qt 的反射机制是指在运行时获取对象的属性、方法和信号等信息的能力。这种机制使得开发者可以在运行时动态地获取和修改对象的属性和方法，从而实现更加灵活的编程。

反射是 Qt 编程框架中一个重要的内容。由于 C++ 编程语言的限制， C++ 自身不支持反射编程。Qt 通过使用编程语言模板 moc 和 QMetaObject 类来实现。

使用 Qt 的反射机制需要在代码中满足三个条件：

* 该类需要继承自 QMetaObject类、Qobject 基类或者其派生类。
* 必须在类的私有声明区添加 Q\_OBJECT 宏，用于起用元对象功能。
* 要使用元对象编译器 Meta-Object Compiler（moc）编译源文件生成 moc 模板文件，为元对象特性提供必要的元代码。 moc 模板文件会和源文件一起被编译进应用程序。

下面是一个简单的 Qt 反射编程示例：

// file : main.cpp  
#include <QCoreApplication>  
#include <QDebug>  
#include <QMetaObject>  
#include <QMetaProperty>  
  
class ReflectDemo : public QObject  
{  
 Q\_OBJECT // 使用Qt的元对象宏定义  
  
 Q\_PROPERTY(QString name READ getName WRITE setName NOTIFY nameChanged) // 声明属性"name"，可读、可写，并带有通知信号  
  
public:  
 ReflectDemo(QObject \*parent = nullptr) : QObject(parent), m\_name("Hello, World!") {}  
  
 QString getName() const { return m\_name; } // 读取属性值的函数  
 void setName(const QString &name) { m\_name = name; emit nameChanged(); } // 设置属性值的函数，并发射属性变化通知信号  
  
signals:  
 void nameChanged(); // 属性变化通知信号  
  
private:  
 QString m\_name; // 私有成员变量，用于存储属性值  
};  
  
int main(int argc, char \*argv[])  
{  
 QCoreApplication app(argc, argv);  
  
 ReflectDemo obj;  
 const QMetaObject \*metaObj = obj.metaObject(); // 获取对象的元对象  
  
 int propIdx = metaObj->indexOfProperty("name"); // 获取属性"name"的索引  
 if (propIdx != -1) {  
 QMetaProperty prop = metaObj->property(propIdx); // 获取属性元数据  
 qDebug() << "Property name:" << prop.name(); // 输出属性名  
 qDebug() << "Property value:" << prop.read(&obj); // 读取属性值  
 }  
  
 return app.exec();  
}  
#include "main.moc" //必须包含 main.moc 才能触发 moc 编译

# 设置CMake的最低版本要求  
cmake\_minimum\_required(VERSION 3.5)  
  
# 定义项目名称  
project(ReflectionDemo)  
  
# 在相应的构建目录中查找包含文件  
set(CMAKE\_INCLUDE\_CURRENT\_DIR ON)  
  
# 告诉CMake在需要时自动运行moc（元对象编译器）  
set(CMAKE\_AUTOMOC ON)  
  
# 查找Qt5的Widgets库  
find\_package(Qt5 COMPONENTS Widgets REQUIRED)  
  
# 告诉CMake创建ReflectionDemo可执行文件  
add\_executable(ReflectionDemo main.cpp)  
  
# 使用Qt 5的Widgets模块  
target\_link\_libraries(ReflectionDemo Qt5::Widgets)

编译运行结果：

nihao@nihao-z690:~/workdir/qttest$ ls  
CMakeLists.txt main.cpp  
nihao@nihao-z690:~/workdir/qttest$ cmake .  
-- The C compiler identification is GNU 11.4.0  
-- The CXX compiler identification is GNU 11.4.0  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Check for working C compiler: /usr/bin/cc - skipped  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Check for working CXX compiler: /usr/bin/c++ - skipped  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/nihao/workdir/qttest  
nihao@nihao-z690:~/workdir/qttest$ make  
[ 25%] Automatic MOC for target ReflectionDemo  
[ 25%] Built target ReflectionDemo\_autogen  
[ 50%] Building CXX object CMakeFiles/ReflectionDemo.dir/ReflectionDemo\_autogen/mocs\_compilation.cpp.o  
[ 75%] Building CXX object CMakeFiles/ReflectionDemo.dir/main.cpp.o  
[100%] Linking CXX executable ReflectionDemo  
[100%] Built target ReflectionDemo  
nihao@nihao-z690:~/workdir/qttest$ ./ReflectionDemo  
Property : 10  
Invoke Method Begin:  
Hello Boy  
Invoke Method End  
Meta Type : int ID: 2  
Meta Property : Var QVariant(int, 15)  
"Meta Method : void sayHello(QString inName)"  
Meta Enum : Number  
-- Zero : 0  
-- One : 1  
-- Two : 2  
-- Three : 3  
^C

上述代码是一个基本的Qt反射使用示例，更详细的内容，请参阅：

* Qt Meta Object System：[https://doc.qt.io/qt-6/metaobjects.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/metaobjects.html)
* QMetaObject ：[https://doc.qt.io/qt-6/qmetaobject.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qmetaobject.html)
* Qt Property System：[https://doc.qt.io/qt-6/properties.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/properties.html)

### Signal-Slot

信号槽机制属于 Qt 特有的机制，信号槽的本质是高级回调。信号槽机制被用于 Qt 对象之间的通信，是 Qt 的核心特性。

#### 概念

**1、信号（Signal）就是在特定情况下被发射的事件。**

例如 PushButton 最常见的信号就是鼠标单击时发射的 clicked() 信号，一个 ComboBox 最常见的信号是选择的列表项变化时发射的 CurrentIndexChanged() 信号。GUI 程序设计的主要内容就是对界面上各组件的信号的响应，只需要知道什么情况下发射哪些信号，合理地去响应和处理这些信号就可以了。

**2、槽（Slot）就是对信号响应的函数。槽就是一个函数。**

与一般的C++函数是一样的，可以定义在类的任何部分（public、private 或 protected），可以具有任何参数，也可以被直接调用。槽函数与一般的函数不同的是，槽函数可以与一个信号关联，当信号被发射时，关联的槽函数被自动执行。

信号与槽之间的关联：是用 QObject::connect() 函数实现的，其基本格式是：

QObject::connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));  
// 信号发出者，处理的信号， 信号接收者，处理动作方法（槽函数）。  
// sender 是发射信号的对象的名称  
// signal() 是信号名称。信号可以看做是特殊的函数，需要带括号，有参数时还需要指明参数。  
// receiver 是接收信号的对象名称，slot() 是槽函数的名称，需要带括号，有参数时还需要指明参数。  
// SIGNAL 和 SLOT 是 Qt 的宏，用于指明信号和槽，并将它们的参数转换为相应的字符串。

#### 注意项

**1、一个信号可以连接多个槽, 当一个信号与多个槽函数关联时，槽函数按照建立连接时的顺序依次执行。**

connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(addFun(int));  
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(updateStatus(int));

**2、多个信号可以连接同一个槽。**

connect(ui->rBtnBlue,SIGNAL(clicked()),this,SLOT(setTextFontColor()));  
connect(ui->rBtnRed,SIGNAL(clicked()),this,SLOT(setTextFontColor()));  
connect(ui->rBtnBlack,SIGNAL(clicked()),this,SLOT(setTextFontColor()));

**3、一个信号可以连接另外一个信号。**

**4、严格的情况下，信号与槽的参数个数和类型需要一致，至少信号的参数不能少于槽的参数。如果不匹配，会出现编译错误或运行错误。**

**5、使用 signals/slots 必须要加入宏 Q\_OBJECT 。**

**6、当一个信号被发射时，与其关联的槽函数通常被立即执行，就像正常调用一个函数一样。只有当信号关联的所有槽函数执行完毕后，才会执行发射信号处后面的代码。**

**7、一定要有signals关键字，定义信号时这个关键字不可或缺，比如我们定义一个信号void signal()，一定要在前面加上关键字“signals:” 。**

signals:  
 void signal();  
// 就像类中的 public、protected、pravate 一样，但是一定不能在 signals 前面加上 public、protected、private，publi signals: 这样写是错误的。

**8、slots 可以写可以不写，一般的函数也可以与signals下的信号关联，我们定义槽函数时可以像signals那样加上slots关键字，也可以不加，但是需要注意的是，如果加上了，那就必须加上public、protected、paivate 。**

public slots:  
 void func();  
// 当然不加上slots的一般函数也可以与信号关联。

**9、signals下的函数必须是void类型，而且只需要给出声明即可，具体实现QT内部自己处理，但是槽函数一定要实现，从我们角度思考是这种信号处理是一致的，但是槽函数的功能确实根据我们需要自己设计，所以有了这种差异。**

**10、信号与槽的参数不能是宏和函数指针。**

**11、信号一般与 emit 配合使用，使用 emit 发射信号给关联的槽。**

**12、connect 若触发，它后面的不会再运行。**

#### emit 发射信号

emit 是 Qt 关键字，像其他有关 Qt 扩展一样，它也会被 C++ 预处理器转换成标准的 C++ 代码。

#include <QCoreApplication>  
#include <QDebug>  
#include <QMetaObject>  
#include <QMetaProperty>  
class ReflectDemo : public QObject  
{  
 Q\_OBJECT // 使用Qt的元对象宏定义  
  
 Q\_PROPERTY(QString name READ getName WRITE setName NOTIFY nameChanged) // 声明属性"name"，可读、可写，并带有通知信号  
  
public:  
 ReflectDemo(QObject \*parent = nullptr) : QObject(parent), m\_name("Hello, World!") {}  
  
 QString getName() const { return m\_name; } // 读取属性值的函数  
 void setName(const QString &name)   
 {   
 m\_name = name;   
 emit nameChanged(); // emit 发射信号到 nameChanged。  
 } // 设置属性值的函数，并发射属性变化通知信号  
  
signals:  
 void nameChanged(); // 属性变化通知信号  
  
private:  
 QString m\_name; // 私有成员变量，用于存储属性值  
};

#### 示例

#include "widget.h"  
#include <QApplication>  
  
int main(int argc, char \*argv[])  
{  
 // 创建Qt应用程序对象  
 QApplication a(argc, argv);  
  
 // 创建主窗口对象  
 Widget w;  
  
 // 显示主窗口  
 w.show();  
  
 // 进入Qt应用程序的事件循环，等待用户交互和事件处理  
 return a.exec();  
}

// file: widget.h  
#ifndef WIDGET\_H  
#define WIDGET\_H  
  
#include <QWidget>  
  
class Widget : public QWidget  
{  
 Q\_OBJECT  
  
public:  
 Widget(QWidget \*parent = 0);  
 ~Widget() {}  
public slots:  
 void click\_fun();  
 void all\_click\_fun();  
};  
  
#endif // WIDGET\_H

// file: widget.cpp  
#include "widget.h"  
  
#include <QPushButton>  
#include <QGridLayout>  
#include <QDebug>  
  
// 点击按钮时的槽函数  
void Widget::click\_fun()  
{  
 qDebug() << "点击按钮";  
}  
// 点击按钮时的槽函数  
void Widget::all\_click\_fun()  
{  
 qDebug() << "发生一次点击";  
}  
Widget::Widget(QWidget \*parent)  
 : QWidget(parent)  
{  
 /\* 创建控件 \*/  
 QPushButton \*btnClick = new QPushButton("click", this); // 创建一个按钮  
 QPushButton \*btnBlock = new QPushButton("block", this);  
 QPushButton \*btnUnblock = new QPushButton("unblock", this);  
 QPushButton \*btnDisconnect = new QPushButton("disconnect", this);  
 QPushButton \*btnConnect = new QPushButton("connect", this);  
  
 // 创建一个网格布局并将按钮添加到布局中  
 QGridLayout \*pLayout = new QGridLayout();  
 pLayout->addWidget(btnClick, 0, 0);  
 pLayout->addWidget(btnBlock, 0, 1);  
 pLayout->addWidget(btnUnblock, 0, 2);  
 pLayout->addWidget(btnDisconnect, 0, 3);  
 pLayout->addWidget(btnConnect, 0, 4);  
 this->setLayout(pLayout);  
  
 /\* 信号槽 \*/  
 // 按钮点击信号连接到槽函数 click\_fun  
 connect(btnClick, &QPushButton::clicked, this, &Widget::all\_click\_fun);  
 connect(btnClick, &QPushButton::clicked, this, &Widget::click\_fun);  
 btnUnblock->setEnabled(false);  
 btnConnect->setEnabled(false);  
  
 // 按钮点击信号连接到Lambda表达式，用于阻止按钮点击信号  
 connect(btnBlock, &QPushButton::clicked, this, [=]()  
 {  
 b介绍tnClick->blockSignals(true);  
 btnBlock->setEnabled(false);  
 btnUnblock->setEnabled(true);  
 qDebug() << "阻止信号"; });  
  
 // 按钮点击信号连接到Lambda表达式，用于解除按钮点击信号的阻止  
 connect(btnUnblock, &QPushButton::clicked, this, [=]()  
 {  
 btnClick->blockSignals(false);  
 btnBlock->setEnabled(true);  
 btnUnblock->setEnabled(false);  
 qDebug() << "解除阻止"; });  
  
 // 按钮点击信号连接到Lambda表达式，用于断开按钮点击信号的连接  
 connect(btnDisconnect, &QPushButton::clicked, this, [=]()  
 {  
 btnConnect->setEnabled(true);  
 btnDisconnect->setEnabled(false);  
 disconnect(btnClick, &QPushButton::clicked, this, &Widget::click\_fun);  
 qDebug() << "断开信号链接"; });  
  
 // 按钮点击信号连接到Lambda表达式，用于重新连接按钮点击信号  
 connect(btnConnect, &QPushButton::clicked, this, [=]()  
 {  
 btnDisconnect->setEnabled(true);  
 btnConnect->setEnabled(false);  
 connect(btnClick, &QPushButton::clicked, this, &Widget::click\_fun);  
 qDebug() << "连接信号"; });  
}

cmake\_minimum\_required(VERSION 3.5)  
  
project(SignalSlotDemo)  
  
# Find includes in corresponding build directories  
set(CMAKE\_INCLUDE\_CURRENT\_DIR ON)  
  
# Instruct CMake to run moc automatically when needed  
set(CMAKE\_AUTOMOC ON)  
  
# Find the QtWidgets library  
find\_package(Qt5 COMPONENTS Widgets REQUIRED)  
  
# Tell CMake to create the helloworld executable  
add\_executable(SignalSlotDemo main.cpp widget.cpp)  
  
# Use the Widgets module from Qt 5  
target\_link\_libraries(SignalSlotDemo Qt5::Widgets)

编译运行：

nihao@nihao-z690:~/workdir/qttest1$ ls  
CMakeLists.txt main.cpp widget.cpp widget.h  
nihao@nihao-z690:~/workdir/qttest1$ mkdir build  
nihao@nihao-z690:~/workdir/qttest1$ cd build/  
nihao@nihao-z690:~/workdir/qttest1/build$ cmake ..  
-- The C compiler identification is GNU 11.4.0  
-- The CXX compiler identification is GNU 11.4.0  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Check for working C compiler: /usr/bin/cc - skipped  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Check for working CXX compiler: /usr/bin/c++ - skipped  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/nihao/workdir/qttest1/build  
nihao@nihao-z690:~/workdir/qttest1/build$ make   
[ 20%] Automatic MOC for target SignalSlotDemo  
[ 20%] Built target SignalSlotDemo\_autogen  
[ 40%] Building CXX object CMakeFiles/SignalSlotDemo.dir/SignalSlotDemo\_autogen/mocs\_compilation.cpp.o  
[ 60%] Building CXX object CMakeFiles/SignalSlotDemo.dir/main.cpp.o  
[ 80%] Building CXX object CMakeFiles/SignalSlotDemo.dir/widget.cpp.o  
[100%] Linking CXX executable SignalSlotDemo  
[100%] Built target SignalSlotDemo  
nihao@nihao-z690:~/workdir/qttest1/build$ ./SignalSlotDemo  
发生一次点击  
点击按钮  
阻止信号  
解除阻止  
发生一次点击  
点击按钮  
断开信号链接  
发生一次点击  
发生一次点击  
连接信号  
发生一次点击  
点击按钮



效果：

1、点击 "click" 按钮，“click\_fun” 和 “all\_click\_fun” 信号槽收到信号，在终端打印出字符串。

2、点击 "block" 按钮，"click" 按钮的信号被阻止发出，这时再点击 "click" 按钮，两个信号槽函数都无法接收到信号，无法执行终端字符串打印。点击 “unblock” 按钮解除对 “click” 按钮的信号阻止。

3、点击 "disconnect" 按钮，"click" 按钮的信号将会和 "click\_fun" 信号槽函数断开，当再次点击该函数时，"click\_fun" 信号槽函数将不会被执行。点击 "connect" 按钮后恢复连接。

对于更细节的说明，请查阅：

* [https://doc.qt.io/qt-6/signalsandslots.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/signalsandslots.html)

### **Event System**

在Qt中，事件是指抽象类 [QEvent](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qevent.html) 子类的实例对象，它用于描述应用程序内部或外部活动的结果。在作用上它可以看做是重量级的Single-Slot，QWigdet中的交互机制都是通过事件实现的。

当一个事件发生时，Qt通过构造适当的QEvent子类实例来表示它，并调用：

* [static bool QCoreApplication::sendEvent(QObject\* receiver, QEvent event)](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qcoreapplication.%3Ci%3Ehtml%23sen%3C/i%3EdEvent)

来向对应的QObject发送事件，而QObject拥有一个虚函数：

* [virtual bool QObject::event( QEvent\* e )](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qobje%3Ci%3Ec%3C/i%3Et.html%23event)

通过覆写event可以在其中处理事件或者进一步分发到其他函数

此外，Qt还支持事件监听器机制，可以覆写QObject的虚函数：

* [>bool QObject:: eventFilter ( QObject \* watched , QEvent\* event )](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qobject%3Ci%3E.html%23e%3C/i%3EventFilter)

使之可以作为filterObj，再调用：

* [void QObject:: installEventFilter ( QObject\* filterObj )](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qobject.html%23%3Ci%3EinstallEv%3C/i%3EentFilter)

从而让该对象能作为其他的对象的事件监听者

关于事件系统，详见：

* [https://doc.qt.io/qt-6/eventsandfilters.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/eventsandfilters.html)

### **Garbage Collection**

Qt 中有一个非常简单的 GC 内存回收，就是 QObject 可以指定唯一的 parent ，当 parent 释放时，还会释放 children ，此外，可以还可以使用 [QObject::deleteLater](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qobject.html%23deleteLater) 来延迟QObject的释放。

在 QWidget 中，加入子控件会隐式设置 parent ，绝大多数情况下都不需要手动去 delete ，但需要注意控制 parent 的生命周期，对象树中的某个节点如果出现纰漏，那么整个子树的对象都无法被释放。

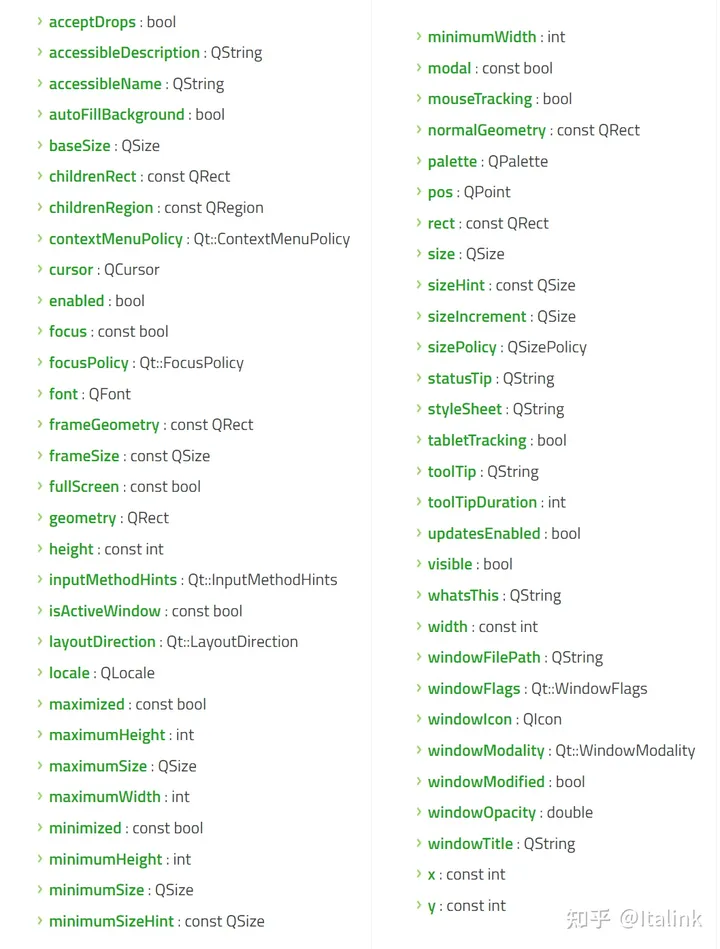
### **Widget**

**[QWidget](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html)** 是Qt GUI的核心，Qt中的绝大多数的 可视部件 都派生于它

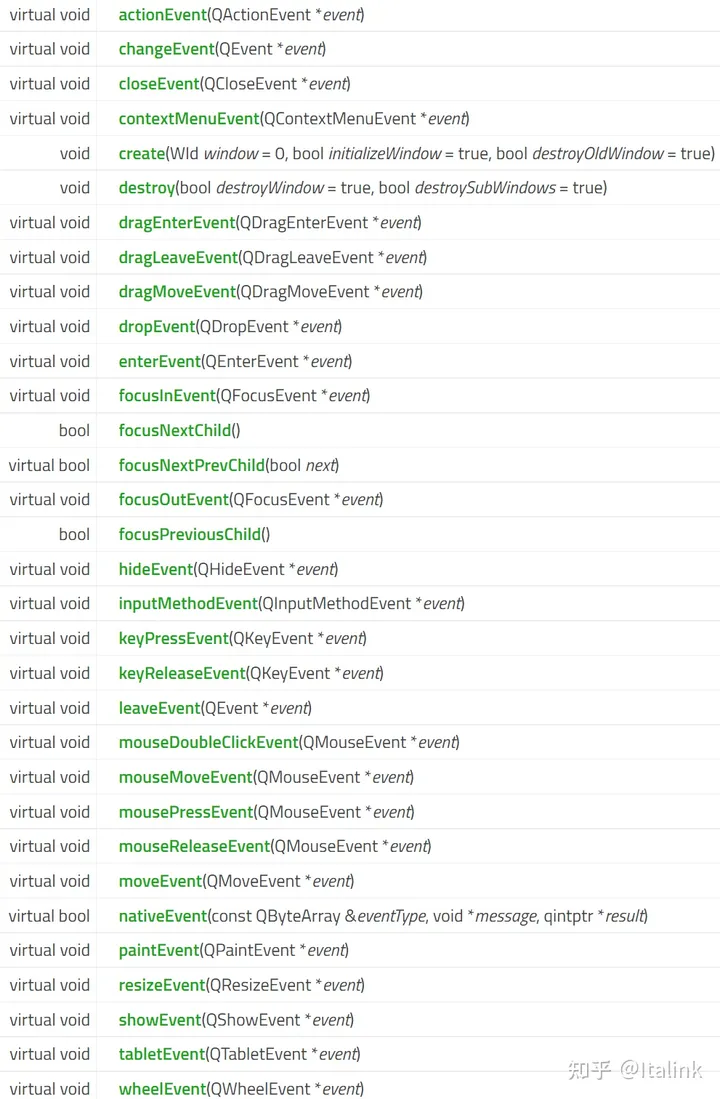
它的创建也很简单：

#include <QApplication>  
#include <QWidget>  
  
int main(int argc, char\* argv[]) {  
 QApplication a(argc, argv);  
 QWidget widget;  
 widget.setWindowTitle("Hello QWigdet!");  
 widget.show();  
 return a.exec();  
}

单个部件的开发，主要是设置 QWidget 的 [Properties](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23protected-functions)：



和覆写 [Event](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23protected-functions)：



来实现各种复杂的界面效果和交互逻辑。

### **WindowOption**

在一般情况下，开发者甚至都不会接触到 QWindow，因为最外层 （or 最顶级）的 QWidget 会自动生成 QWindow，我们可以通过对这个QWidget做一些设置，来影响Window的生成：

* [WindowFlag](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qt.html%23WindowType-enum)：调整Window的标识，比如弹窗，对话框，提示框，无边框...
* [WindowModality](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qt.html%23WindowModality-enum)：设置模态窗口，模特窗口会阻塞其他窗口的输入，一般用于让用户在当前强制选择某些选项，才能继续执行
* [WindowState](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qt.html%23WindowState-enum)：Window的状态有：默认，最小化，最大化，全屏（无边框），激活，可以调用setWindowState，Qt也提供了一些便捷API，例如 [activateWindow()](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23activateWindow)

比如这样的代码：

QWidget widget;  
widget.setWindowFlags(Qt::FramelessWindowHint|Qt::Tool);  
widget.resize(200,200);  
widget.show();

你能在屏幕上看到一个200\*200的灰色方块，并且你在下方的任务栏中，它的图标也被隐藏了

需要注意的是，在创建Widget的时候并没有创建Window，我们对Widget的设置绝大多数时候只是在调整它的属性状态机，只有在事件执行时，才会根据这些状态来进行实际的操作，就比如说，你得在show之后才能拿到Window的句柄

### **[Attribute](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qt.html%23WidgetAttribute-enum)**

在Qt命名空间下，介绍介绍有一组以WA\_开头的枚举值可以用来设置[WidgetAttribute](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qt.html%23WidgetAttribute-enum)，你可以给QWidget设置很多有用的属性，其中常用的：

* Qt::WA\_DeleteOnClose：关闭窗口时，销毁该对象
* Qt::WA\_TranslucentBackground：使窗口的背景变透明

### **Event**

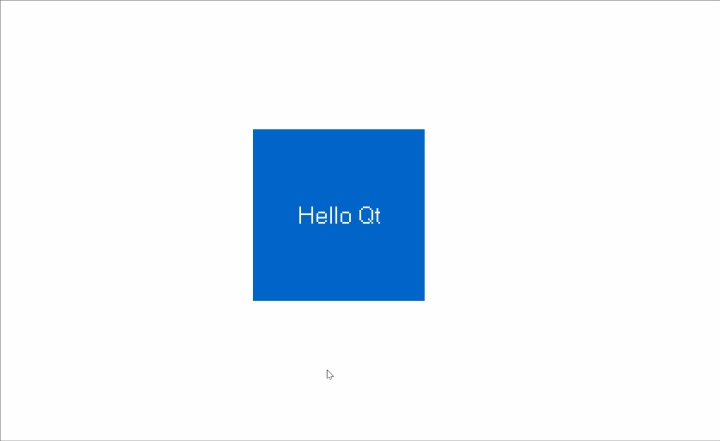
GUI的显示在于 QWidget 的 [paintEvent](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23paintEvent) ，我们可以通过覆写它，在里面使用通过 [QPainter](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qpainter.html) 绘制我们想要的任何图形

而 GUI 的本质无非就是借助于各类交互事件，调整属性状态机，切换 paintEvent 的绘制结果

这是一个很好的例子：

#include <QApplication>  
#include <QWidget>  
#include <QMouseEvent>  
#include <QPainter>  
  
class QExmapleWidget:public QWidget {  
 Q\_OBJECT  
private:  
 QPoint mMousePressPos;  
 bool bIsHovered = false;  
protected:  
 void mousePressEvent(QMouseEvent\* event) override {  
 if (event->button() == Qt::LeftButton) //记录鼠标左键按下时的位置  
 mMousePressPos = event->pos();   
 }  
 void mouseMoveEvent(QMouseEvent\* event) override {  
 if (event->buttons() & Qt::LeftButton) //鼠标左键按下拖拽时，调整Widget的位置  
 move(event->pos() + this->pos() - mMousePressPos);  
 }  
  
 void enterEvent(QEnterEvent\* event) override {  
 bIsHovered = true; //鼠标进入Widget中  
 setCursor(Qt::ClosedHandCursor); //设置鼠标光标的形状  
 update(); //请求刷新界面  
 }  
  
 void leaveEvent(QEvent\* event) override {  
 bIsHovered = false; //鼠标离开Widget  
 setCursor(Qt::ArrowCursor); //设置鼠标光标的形状  
 update(); //请求刷新界面  
 }  
  
 void paintEvent(QPaintEvent\* event) override {  
 QPainter painter(this);  
  
 painter.fillRect(this->rect(),bIsHovered ? Qt::green : QColor(0,100,200)); //在区域内填充颜色  
  
 painter.setPen(QPen(Qt::white));  
 painter.setFont(QFont("",20,90)); //使用缺省字体并设置字号和权重  
 painter.drawText(this->rect(), Qt::AlignCenter, "Hello Qt"); //绘制文字  
 }  
};  
  
int main(int argc, char\* argv[]) {  
 QApplication a(argc, argv);  
 QExmapleWidget widget;  
 widget.setWindowFlags(Qt::FramelessWindowHint); //无边框  
 widget.setAttribute(Qt::WA\_TranslucentBackground); //背景透明   
 widget.setAttribute(Qt::WA\_AlwaysStackOnTop); //置顶  
 widget.resize(200,200);  
 widget.show();  
 return a.exec();  
}  
  
#include "main.moc"

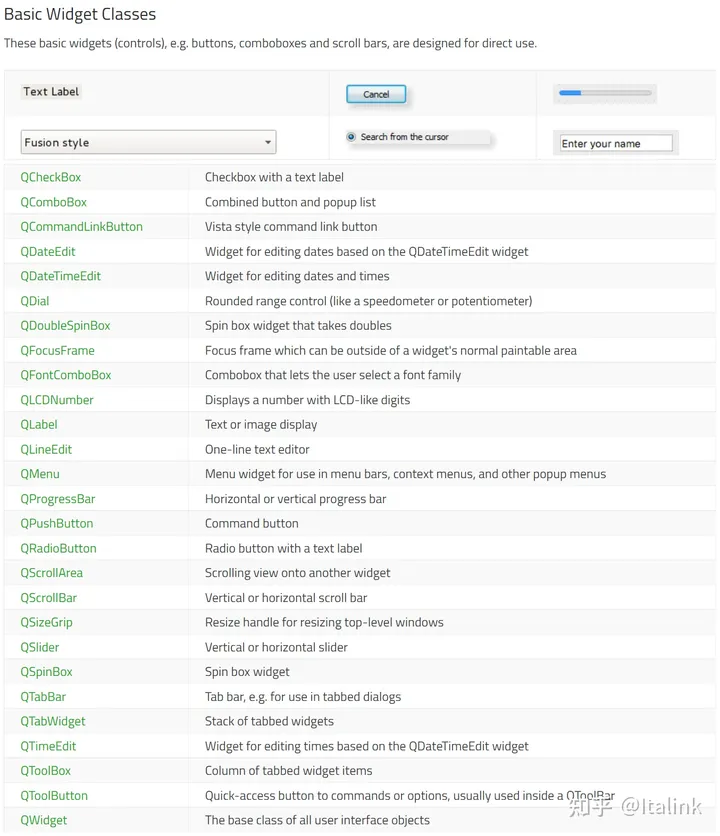
运行它你能看到如下效果：



使用QWidget，可以实现我们想要的任何效果。

Qt官方的提供了很多基础控件供开发者使用：

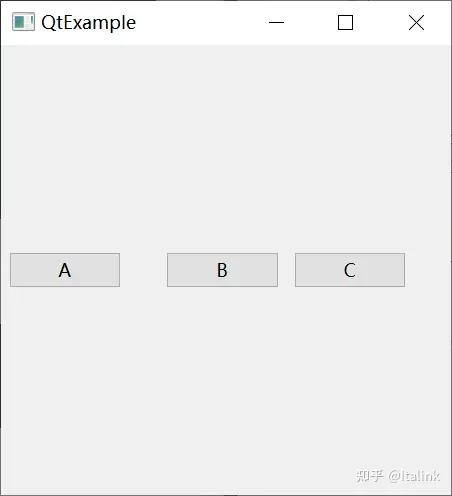
* [https://doc.qt.io/qt-6/widget-classes.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/widget-classes.html)



### Layout

在一个程序中，往往不会只有一个QWidget，所以我们还需要组织这些Widget的布局，Qt中常用的布局是BoxLayout，这里有一个简单的示例：

#include <QApplication>  
#include <QWidget>  
#include <QHBoxLayout>  
#include <QPushButton>  
  
class QMainWidget :public QWidget {  
 Q\_OBJECT  
public:  
 QMainWidget() {  
 createUI();  
 }  
private:  
 void createUI() {  
 QHBoxLayout\* hLayout = new QHBoxLayout(this);  
 // QHBoxLayout\* hLayout = new QHBoxLayout(); //等价  
 //setLayout(hLayout);  
  
 hLayout->setContentsMargins(5, 5, 5, 5); //设置外边距  
 hLayout->setSpacing(10); //设置内部元素间隔  
 hLayout->setAlignment(Qt::AlignLeft|Qt::AlignVCenter); //设置左对齐且垂直居中  
 hLayout->addWidget(new QPushButton("A"));  
 hLayout->addSpacing(20); //加入空白填充  
 hLayout->addWidget(new QPushButton("B"));  
 hLayout->addWidget(new QPushButton("C"));  
 }  
};  
  
int main(int argc, char\* argv[]) {  
 QApplication a(argc, argv);  
 QMainWidget widget;  
 widget.resize(300, 300);  
 widget.show();  
 return a.exec();  
}  
  
#include "main.moc"



布局的完整介绍，请查阅：

* [https://doc.qt.io/qt-6/layout.html](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/layout.html)

对于：

* 最外层的Widget
* 没有设置Layout，但具有层级关系的Widget

可以通过相对与parent左上角坐标来调整以下属性

* pos：[https://doc.qt.io/qt-6/qwidget.html#pos-prop](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23pos-prop)
* geometry：[https://doc.qt.io/qt-6/qwidget.html#geometry-prop](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23geometry-prop)

在同一层级的Widget，可以调用以下接口来调整Z Order：

* [raise()](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23raise)：置顶
* [lower()](https://link.zhihu.com/?target=https%3A//doc.qt.io/qt-6/qwidget.html%23lower)：置底

### **Style**

Qt的默认样式并不美观，但它支持使用 QSS 进行美化控件，详见：

* <https://doc.qt.io/qt-5/stylesheet.html>

### 更进一步

Qt 是一个复杂的编程框架，前面也只不过介绍了它的皮毛，想要很好驾驭 Qt 的开发，是需要日积月累不断学习的。只有不断的实践才能掌握 Qt 开发的精髓。

在学习框架的过程中，最最最重要的是：

* 官方示例：<https://doc.qt.io/qt-5/qtexamplesandtutorials.html>
* 官方文档：<https://doc.qt.io/qt-5>
* 源码

在遇到问题的时候，往往能帮上你的是：

* 社区

<https://zhuanlan.zhihu.com/p/605656730?utm_id=0>

Github Pages：[Modern Graphics Engine Guide](https://link.zhihu.com/?target=https%3A//italink.github.io/ModernGraphicsEngineGuide/00-C%2B%2B/6.GUI/)

<https://zhuanlan.zhihu.com/p/649041904>

# 第五章：大容量数据存储

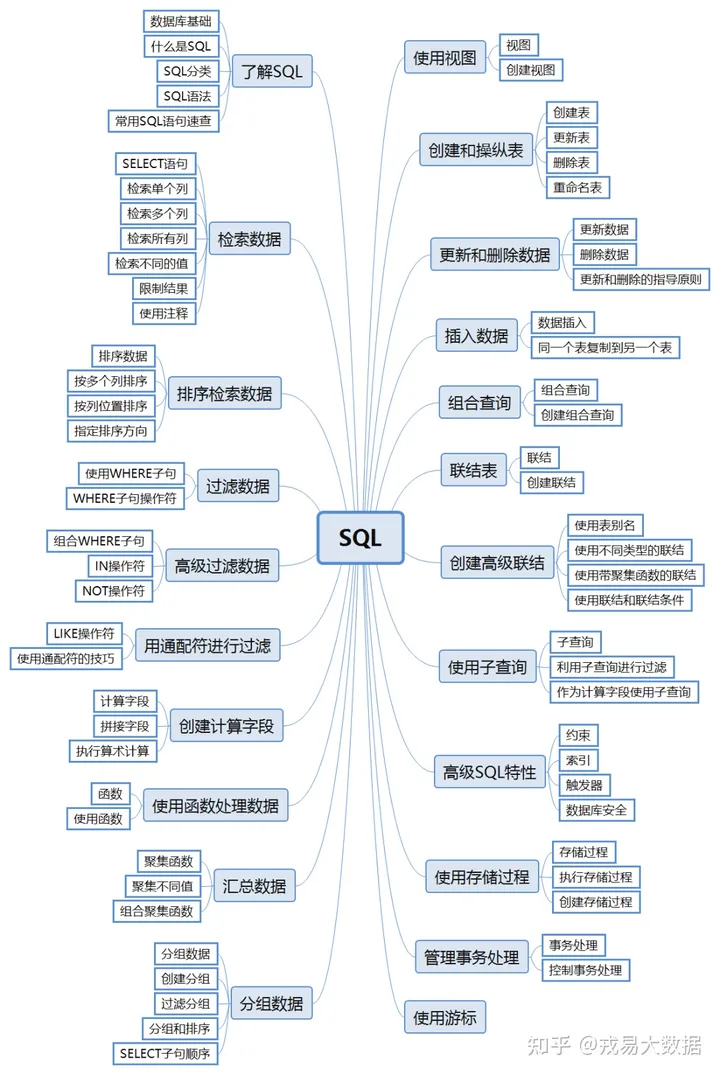
## 1. SQL

SQL 在百度词条上的解释是：

SQL (Structured Query Language) 是具有数据操纵和数据定义等多种功能的数据库语言，这种语言具有交互性特点，能为用户提供极大的便利，数据库管理系统应充分利用 SQL 语言提高计算机应用系统的工作质量与效率。SQL 语言不仅能独立应用于终端，还可以作为子语言为其他程序设计提供有效助力，该程序应用中，SQL 可与其他程序语言一起优化程序功能，进而为用户提供更多更全面的信息。

从计算机被发明以来，计算和储存成为了计算机两个最重要的用途。计算体现了计算机为人类提供智能自动化的服务，而存储是将人类带入现代文明的基石，是现代信息技术的承载体。SQL ”结构化查询语言数据库“ 是专用于储存领域的技术，提供优秀的数据存储、删除、修改、查寻服务。

下图是数据库需要技术导图：



### 数据库产品

常用的数据库软件有多个，下面介绍常见的数据软件：

* oracle

Oracle是一款面向企业级应用的数据库管理系统，被广泛应用于寻求经济高效的DBMS团队。它具有自动配置、扩展和调整的功能，保护、修补和修复数据库，并且没有用户干预。Oracle还支持快速部署和扩展，适合寻求经济高效的DBMS团队。因此，Oracle是市场上最优秀的数据库软件之一。

* mysql

mysql是一种免费的数据库管理系统，它小巧实用，很多人选择它作为学习数据库编程的起点。虽然免费更新比较慢，不适合大型应用，但它是免费且可以快速更新的。MySQL是一个开源数据库管理系统，由瑞典MySQLAB公司开发。它提供了直观的图形用户界面，可用于创建、修改和执行SQL查询，并可以备份和还原数据库。

* sqlserver

sqlserver是微软的数据库管理系统，适用于小中大型应用。它被广泛应用于网络游戏等场景，很多网络游戏就是用sqlserver作为数据库的。

* access

是微软推出的商业数据库软件，主要运行在 windows 系统中，为客户提供集成的数据库服务。

* Sysbase

一个老牌数据库。Sysbase主要有三种版本，一是UNIX操作系统下运行的版本，二是NovellNetware环境下运行的版本，三是WindowsNT环境下运行的版本。其中，UNIX \*\*\* 作系统下运行的版本应用最广泛，而Sybase则是基于客户和服务器体系结构的数据库，被广泛应用于企业信息系统中。Sybase开发工具较少，经验丰富的人员很少。

* Redis

Redis 是一种适合跨本地、混合或云数据库工作的软件。它允许用户实现亚毫秒级的数据处理，并允许用户在不同的数据模型中进行编程。Redis还提供了一些高级功能，如持久化、备份和恢复等，使得开发人员可以更好地管理和保护数据。通过 Redis，开发人员可以快速地创建、读取、写入和查询数据，并轻松地与其他系统进行集成。因此，Redis 是开发人员可以使用和部署在本地和云环境中的理想选择。

* MongoDB

mongoDB是一个介于关系数据库和非关系数据库之间的开源产品，是最接近于关系型数据库的NoSQL数据库。它在轻量级JSON交换基础之上进行了扩展，即称为BSON的方式来描述其无结构化的数据类型。尽管如此它同样可以存储较为复杂的数据类型。

* DB2

IBM DB2 是美国IBM公司开发的一套关系型数据库管理系统，它主要的运行环境为UNIX（包括IBM自家的AIX）、Linux、IBM i（旧称OS/400）、z/OS，以及Windows服务器版本。

DB2主要应用于大型应用系统，具有较好的可伸缩性，可支持从大型机到单用户环境，应用于所有常见的服务器操作系统平台下。 DB2提供了高层次的数据利用性、完整性、安全性、可恢复性，以及小规模到大规模应用程序的执行能力，具有与平台无关的基本功能和SQL命令。DB2采用了数据分级技术，能够使大型机数据很方便地下载到LAN数据库服务器，使得客户机/服务器用户和基于LAN的应用程序可以访问大型机数据，并使数据库本地化及远程连接透明化。 DB2以拥有一个非常完备的查询优化器而著称，其外部连接改善了查询性能，并支持多任务并行查询。 DB2具有很好的网络支持能力，每个子系统可以连接十几万个分布式用户，可同时激活上千个活动线程，对大型分布式应用系统尤为适用。

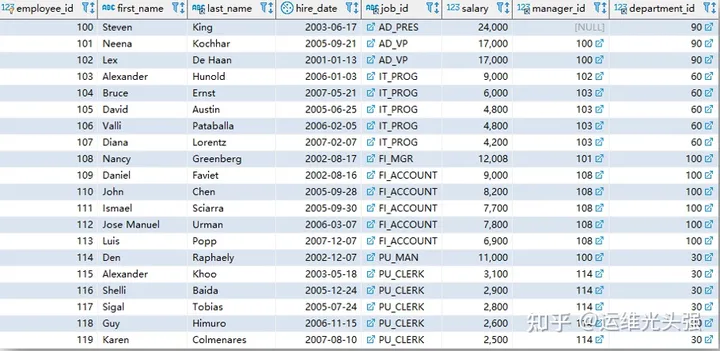
* SQLite

SQLite，是一款轻型的数据库，是遵守ACID的关系型数据库管理系统，它包含在一个相对小的 C 库中。它是 D.RichardHipp 建立的公有领域项目。它的设计目标是嵌入式的，而且已经在很多嵌入式产品中使用了它，它占用资源非常的低，在嵌入式设备中，可能只需要几百 K 的内存就够了。它能够支持 Windows/Linux/Unix 等等主流的操作系统，同时能够跟很多程序语言相结合，比如 Tcl、C#、PHP、Java 等，还有 ODBC 接口，同样比起 Mysql、PostgreSQL 这两款开源的世界著名数据库管理系统来讲，它的处理速度比他们都快。SQLite 第一个 Alpha 版本诞生于 2000 年 5 月。 至 2021 年已经接近有 21 个年头，SQLite 也迎来了一个版本 SQLite 3 已经发布。

### SQL 储存结构

最开始的数据储存方式类似图书的目录文件形式，随着数学集合论的发展，以及数据相互关连概念的提出，新的储存模型关系模型被提出，主要用于解决数据存在的相互关连的问题。

关系模型（Relational model）由 E.F.Codd 博士于 1970 年提出，以集合论中的关系概念为基础；无论是现实世界中的实体对象还是它们之间的联系都使用关系表示。我们在数据库系统中看到的关系就是二维表（Table），由行（Row）和列（Column）组成。因此，也可以说关系表是由数据行构成的集合。



数据库的诞生来源关系模型的实用化，所以说在数据库中一切皆关系。

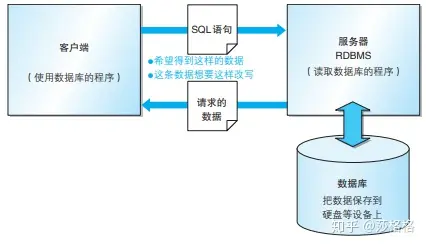
以一条空气质量数据描述：

|  |  |  |
| --- | --- | --- |
| 时间 | 温度/摄氏度 | 湿度/饱和度 |
| 2024年 01月 03日 星期三 18:00:45 CST | 17 | 50% |

时间，温度，湿度三个条目共同描述空气质量，他们之间存在相互关连。同时多条数据能组成时间序列，可以用来详细描述当天的空气质量变化。 这是便是关系的由来。

### SQL 结构化语言

对于操作数据，最开始也是由不同的命令来完成的。在不断的演化中，各种操作逐渐被抽象了出来，形成了结构化语言，SQL 结构化语言也便成为了操作关系数据库的标准语言。SQL 语言在形式上接近正常的英语语法，主要是为了照顾非技术人员的使用需求，简单易用。在使用思想上，用户只需描述好想要的结果，然后将数据处理的过程交给数据系统便可以。所以，SQL 结构化语言是给人设计的操作语言。

介绍

虽然 SQL 是一门 ANSI（American National Standards Institute 美国国家标准化组织）标准的计算机语言，但是仍然存在着多种不同版本的 SQL 语言。为了与 ANSI 标准相兼容，它们必须以相似的方式共同地来支持一些主要的命令（比如 SELECT、UPDATE、DELETE、INSERT、WHERE 等等）。

#### 操作集合

在使用 SQL 语言之前，我们首先要明确我们操作的是什么。

数据库中的一条数据条目由一个或多个列属性值构成，操作数据库也是由基本数据条目为单位进行操作的。还是以前面的空气质量数据为例，一条空气质量就是一个集合，操作时可以一次只操作一个条目，也可以一次操作多个条目，无论一次操作多少个条目，他们统一被称为集合。

从处理的过程来看，数据库收到用户的 SQL 语句后，在库中进行逐级筛选，在不同层次上过滤每一条数据，然后将符合要求的数据组成一个集合再根据 SQL 进行后续的各种操作。

#### 操作方法

SQL 语句是由关键字、表名、列名等组合而成的一条语句。根据对数据库服务器赋予的指令种类的不同，SQL 语句可以分为以下三类：

* DDL

DDL（Data Definition Language，数据定义语言） 用来创建或者删除存储 数据用的数据库以及数据库中的表等对象。DDL 包含以下几种指令：

CREATE：创建数据库和表等对象

CREATE TABLE 表名 (列名 数值类型，...);  
-- 创建一个数据表，表中有三列，分别是日期、温度、湿度  
-- 常见的数值类型：  
-- 整数类型：`INT`、`BIGINT`、`SMALLINT`，用于存储整数值  
-- 浮点类型：`FLOAT`、`DOUBLE`、`REAL`，用于存储小数类型  
-- 字符类型：`CHAR`、`VARCHAR`、`TEXT`，用于储存文本数据  
-- 日期和时间类型：`DATE`、`TIME`、`DATETIME`、`TIMESTAMP`，用于储存日期和时间值  
-- 布尔类型：`BOOLEAN`，用于储存布尔值  
-- 二进制类型：`BLOB`、`BINARY`、`VARBINARY`，用于储存二进制数据

ALTER： 修改数据库和表等对象的结构

ALTER TABLE 表名 DROP COLUMN 列名 ;  
-- 删除列  
  
ALTER TABLE 表名 ADD COLUMN 列名 数值类型 ;  
-- 增加列  
  
ALTER TABLE 表名 CHANGE COLUMN 列名 新列名 ;  
-- 修改列的类型信息  
  
ALTER TABLE 表名字 CHANGE COLUMN 列名 新列名 属性 ;  
-- 重命名列  
  
ALTER TABLE 表名 RENAME TO 表新名 ;  
-- 重命名表  
  
ALTER TABLE 表名 DROP primary key ;  
-- 删除表中主键  
  
ALTER TABLE 表名 ADD CONSTRAINT 约束名 PRIMARY KEY (添加列) ;  
-- 添加主键  
  
ALTER TABLE 表名 ADD index 索引名 (列名);   
-- 添加索引  
  
ALTER TABLE 表名 MODIFY COLUMN 要修改属性的列名 新属性;  
-- 修改列的属性

DROP： 删除数据库和表等对象

DROP TABLE 表名;  
-- 删除表  
  
DROP INDEX 索引名 ON 表名;  
-- 删除索引  
  
DROP VIEW 视图名;  
-- 删除视图  
  
DROP DATABASE 数据库名;  
-- 删除数据库

* DML

DML（Data Manipulation Language，数据操纵语言） 用来查询或者变更 表中的记录。DML 包含以下几种指令：

INSERT：向表中插入新数据

INSERT INTO 表名 VALUES(值1，值2...);  
-- 注意：这种插入数据行的的值必须与表的字段名一一对应，否则数据会插入失败。

UPDATE：更新表中的数据

UPDATE 表名 SET 字段名 = 新值 WHERE 字段名 = 旧值  
-- 更新表中的 WHERE 限定的条目的列值

DELETE：删除表中的数据

DELETE FROM 表名 WHERE 字段名 = 值  
-- 删除表中的一条数据

SELECT：查询表中的数据

SELECT 是数据库中最复杂也是最重要的操作了，由于查寻需求的多样性，查寻功能也被设计的非常灵活。这导致了查寻要学的东西非常多，在这里只介绍基础的查寻功能：

SELECT \* FROM 表名;  
-- 从表中获取所有的条目  
  
SELECT 字段名1, 字段名2, 字段名N FROM 表名;  
-- 从表中获取所有条目，条目中只包含特定的子段。

* DCL

DCL（Data Control Language，数据控制语言） 用来确认或者取消对数据 库中的数据进行的变更。除此之外，还可以对 RDBMS 的用户是否有权限 操作数据库中的对象（数据库表等）进行设定。DCL 包含以下几种指令：

COMMIT： 确认对数据库中的数据进行的变更，多事务时使用。

ROLLBACK：取消对数据库中的数据进行的变更 。

GRANT： 赋予用户操作权限 。

REVOKE： 取消用户的操作权限。

#### SQL 的基本书写规则

1、SQL 语句以分号（;）结尾。

2、SQL 命令语句不区分大小写，但是建议大写，用于和数据进行区分。

3、字符串和日期常数需要使用单引号（'）括起来。 数字常数无需加注单引号（直接书写数字即可）。

4、单词之间需要使用半角空格或者换行符进行分隔。

#### SQL 数据库中的数据运算

数据条目在数据库中会以集合的进行进行条目运算的，集合之间有交集，并集，交叉连接等等。集合条目之间可以获取统计数值，也可以进行加减乘除等运算。这些灵活的数据处理方式都可以通过 SQL 语句进行描述然后交于数据库服务器进行运行。由于多种操作下的 SQL 语句会变的复杂难以维护，在新的 AI 技术面前，可以向 AI 描述好需求，让 AI 生成 SQL 运算语句再交给数据执行，大大减少了操作人员的工作量。

### SQLite

SQLite 是一个非常轻量化的数据库，它自身是由 C 语言编写的，包含大约 13 万行 C 代码，最开始它只支持 C 语言，经过扩展，它很快变得可以支持其他各种类型的编程语言。SQLite 和其他数据库服务器不同的是，SQLite 并不是单独运行的，所以说 SQLite 是一个服务软件并不准确，SQLite 在使用时更像一个完备编程软件库。SQLite 会和使用它的软件编译在一起，为调用它的软件提供 SQL 语句 API 服务。

这是 SQLite 的优点，可以运行在任意的平台上，单也成为了束缚 SQLite 的限制。SQLite 运行的稳定性受软件进程的影响，SQLite 的性能也不会太高。但在单个软件中能够提供如此优秀的服务，对于软件自身来讲也非常足够了。

和 SQLite 相似的基于嵌入式的数据库有： Firebird 、 Berkeley DB、eXtremeDB 等

Firebird 是关系型数据库，功能强大，支持存储过程， SQL 兼容等。

Berkeley DB 并没有数据库服务器的概念，他的程序直接链接到应用程序中 。

eXtremeDB 是内存数据库，运行效率高 。

#### SQLite 安装

SQLite 源码可以 SQLite 官网下载得到：<https://www.sqlite.org/download.html>



源码安装如下：

# 下载源码  
wget https://www.sqlite.org/2023/sqlite-autoconf-3440200.tar.gz  
  
# 解压源码  
unizp sqlite-autoconf-3440200.tar.gz  
  
# 进入源码，并编译  
cd sqlite-autoconf-3440200  
mkdir build  
cd build  
../configure  
make  
  
# 安装  
sudo make install

ubuntu 软件库安装：

sudo apt install libsqlite3-dev

#### SQLite C 程序调用

sqlite3 主要函数介绍：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称 | **操作** | 函数参数 | 返回值 |
| int sqlite3\_exec( sqlite3\* db, /\* 数据库连接句柄 */ const char* sql, /\* 要执行的SQL语句 */ int (*callback)(void*,int,char****,char****), /* 结果处理回调函数 */ void* data, /\* 回调函数的上下文数据 */ char\*\* errmsg /* 出错信息存储位置 \*/ ); | sqlite3\_exec函数用于执行SQL语句，可以是查询语句（SELECT）也可以是修改数据的语句（INSERT、UPDATE、DELETE等）。执行结果可以通过回调函数进行处理。 | db：SQLite数据库连接句柄。 sql：要执行的SQL语句。 callback：一个回调函数，用于处理SQL语句的结果。回调函数通常被调用一次或多次，取决于SQL语句的性质。它的原型应该符合 int (\*)(void\*, int, char\*\*, char\*\*)，其中参数依次是回调函数的上下文数据、列数、结果行的每一列的值（以字符串数组的形式）以及列名（如果有的话）。 data：传递给回调函数的上下文数据，通常用于在回调函数内部传递额外的信息。 errmsg：用于存储出错信息的指针。如果发生错误，会将错误信息存储在这个指针指向的位置。 | 如果操作成功，该函数返回SQLITE\_OK，如果出现错误，将返回相应的错误代码，并且出错信息将存储在errmsg指向的位置。 |

主要头文件：

#include <stdio.h>  
#include <sqlite3.h>

**学生管理系统**

#include <stdio.h>  
#include <stdlib.h>  
#include <sqlite3.h>  
#define DATABASE "stu.db"  
int do\_insert(sqlite3 \*db) // 插入新一条学生数据  
{  
 char \*errmsg;  
 int id;  
 char name[32] = {};  
 int score;  
 char sql[128] = {};  
 printf("Input id:");  
 scanf("%d", &id);  
 getchar();  
 printf("Input name:");  
 scanf("%s", name);  
 getchar();  
 printf("Input score:");  
 scanf("%d", &score);  
 getchar();  
 sprintf(sql, "INSERT INTO stu(id,name,score) VALUES(%d,'%s',%d);", id, name, score);  
 if (sqlite3\_exec(db, sql, NULL, NULL, &errmsg) != SQLITE\_OK)  
 {  
 printf("%s\n", errmsg);  
 return -1;  
 }  
 else  
 {  
 printf("Insert succesful\n");  
 }  
 return 0;  
}  
int do\_delete(sqlite3 \*db) // 删除一条学生数据  
{  
 char \*errmsg;  
 int id;  
 char sql[128] = {};  
 printf("Input id:");  
 scanf("%d", &id);  
 getchar();  
 sprintf(sql, "DELETE FROM stu WHERE id=%d;", id);  
 if (sqlite3\_exec(db, sql, NULL, NULL, &errmsg) != SQLITE\_OK)  
 {  
 printf("%s\n", errmsg);  
 return -1;  
 }  
 else  
 {  
 printf("Delete succesful\n");  
 }  
 return 0;  
}  
int do\_update(sqlite3 \*db) // 更新一个学生的成绩  
{  
 char \*errmsg;  
 int id, score;  
 char sql[128] = {};  
 printf("Input id:");  
 scanf("%d", &id);  
 getchar();  
 printf("Input new score:");  
 scanf("%d", &score);  
 getchar();  
 sprintf(sql, "UPDATE stu SET score=%d WHERE id=%d;", score, id);  
 if (sqlite3\_exec(db, sql, NULL, NULL, &errmsg) != SQLITE\_OK)  
 {  
 printf("%s\n", errmsg);  
 return -1;  
 }  
 else  
 {  
 printf("Update succesful\n");  
 }  
 return 0;  
}  
int callback(void \*para, int columncount, char \*\*columnvalue, char \*\*columnname)  
{  
 for (int i = 0; i < columncount; i++)  
 {  
 printf("%-11s", columnvalue[i]);  
 }  
 printf("\n");  
 return 0;  
}  
int do\_query(sqlite3 \*db) // 打印所有学生数据  
{  
 char \*errmsg;  
 if (sqlite3\_exec(db, "SELECT \* FROM stu;", callback, NULL, &errmsg) != SQLITE\_OK)  
 {  
 printf("%s\n", errmsg);  
 return -1;  
 }  
 else  
 {  
 printf("Update succesful\n");  
 }  
 return 0;  
}  
int main(int argc, char \*\*argv)  
{  
 sqlite3 \*db;  
 char \*errmsg;  
 int option;  
 // 打开或新建数据库  
 if (sqlite3\_open(DATABASE, &db) != SQLITE\_OK)  
 {  
 printf("%s\n", sqlite3\_errmsg(db));  
 exit(-1);  
 }  
 else  
 {  
 printf("Open DATABASE success!\n");  
 }  
 // 创建一个表格，如果表存在，执行错误  
 if (sqlite3\_exec(db, "CREATE TABLE stu(id INTEGER,name CHAR,score INTEGER);", NULL, NULL, &errmsg) != SQLITE\_OK)  
 {  
 printf("%s\n", errmsg);  
 }  
 else  
 {  
 printf("create table successful!\n");  
 }  
 while (1)  
 {  
 printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 printf("1.Insert 2.delete 3.query 4.update 5.quit\n");  
 printf("Input Option:\n");  
 scanf("%d", &option);  
 getchar();  
 switch (option)  
 {  
 case 1:  
 do\_insert(db);  
 break;  
 case 2:  
 do\_delete(db);  
 break;  
 case 3:  
 do\_query(db);  
 break;  
 case 4:  
 do\_update(db);  
 break;  
 case 5:  
 sqlite3\_close(db);  
 exit(1);  
 default:  
 printf("Input error!\n");  
 }  
 }  
  
 return 0;  
}

编译运行：

nihao@nihao-z690:~/workdir$ gcc sqlitedemo.c -o sqlitedemo -lsqlite3  
nihao@nihao-z690:~/workdir$ ./sqlitedemo   
Open DATABASE success!  
create table successful!  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
1  
Input id:1  
Input name:test1  
Input score:95  
Insert succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
1  
Input id:2  
Input name:test2  
Input score:60  
Insert succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
3  
1 test1 95   
2 test2 60   
Update succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
2  
Input id:1  
Delete succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
3  
2 test2 60   
Update succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
4  
Input id:2  
Input new score:99  
Update succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
3  
2 test2 99   
Update succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
5  
nihao@nihao-z690:~/workdir$ ./sqlitedemo   
Open DATABASE success!  
table stu already exists  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
3  
2 test2 99   
Update succesful  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
1.Insert 2.delete 3.query 4.update 5.quit  
Input Option:  
5  
原文链接：https://blog.csdn.net/weixin\_63303786/article/details/130335195

# 第六章：软件框架设计

## 1. C 程序的背后

### 1.1 C 函数的汇编结构

我们知道 C 语言需要被编译成汇编代码才能被计算机识别并执行，但是 C 语言是如何被编译成汇编的，今天来揭开这个谜题。

C 语言的基本结构是由函数组成，这是 C 语言面向过程编程的主要特征。通过编写不同的函数，描述不同的过程，最后将他们组合起来，形成完整的运行链路。这是 C 语言的主要编程思想， 所以在进行 C 语言编程时，首先要把想做的事情拆分成一个一个的步骤，对每个步骤都用函数描述好，最后组合起来就完成了 C 语言的编程。

在 C 语言中没有调用子函数的函数（简称为“非调用函数”）和调用子函数的函数（简称为“调用函数”）在内存栈的操作上有显著的影响，这个影响主要集中在栈内存的分配上。

以一个简单的 C 程序示例：

void test()  
{  
 int a = 1;  
 int b = 1;  
 int c;  
 c = a + b;  
}  
int main(int argc, char \*argv[])  
{  
 int a = 1;  
 int b = 1;  
 int c;  
 c = a + b;  
 return 0;  
}

nihao@nihao-z690:~/workdir/cfun$ gcc -c nihao.c -o nihao.o -g  
nihao@nihao-z690:~/workdir/cfun$ objdump -d -M intel nihao.o  
  
nihao.o： 文件格式 elf64-x86-64  
  
  
Disassembly of section .text:  
  
0000000000000000 <test>:  
 0: f3 0f 1e fa endbr64 ; 强制 64 位模式下的跳转目标  
 4: 55 push rbp ; 将 rbp 寄存器的值压入栈中  
 5: 48 89 e5 mov rbp,rsp ; 设置栈帧基址  
 8: c7 45 f4 01 00 00 00 mov DWORD PTR [rbp-0xc],0x1 ; 将值 1 存储到 [rbp-0xc]  
 f: c7 45 f8 01 00 00 00 mov DWORD PTR [rbp-0x8],0x1 ; 将值 1 存储到 [rbp-0x8]  
 16: 8b 55 f4 mov edx,DWORD PTR [rbp-0xc] ; 将 [rbp-0xc] 的值加载到 edx 寄存器  
 19: 8b 45 f8 mov eax,DWORD PTR [rbp-0x8] ; 将 [rbp-0x8] 的值加载到 eax 寄存器  
 1c: 01 d0 add eax,edx ; 将 edx 和 eax 寄存器中的值相加，结果存储在 eax 中  
 1e: 89 45 fc mov DWORD PTR [rbp-0x4],eax ; 将 eax 寄存器的值存储到 [rbp-0x4]  
 21: 90 nop ; 空操作指令  
 22: 5d pop rbp ; 恢复 rbp 寄存器的值  
 23: c3 ret ; 返回  
  
0000000000000024 <main>:  
 24: f3 0f 1e fa endbr64 ; 强制 64 位模式下的跳转目标  
 28: 55 push rbp ; 将 rbp 寄存器的值压入栈中  
 29: 48 89 e5 mov rbp,rsp ; 设置栈帧基址  
 2c: 48 83 ec 20 sub rsp,0x20 ; 分配 32 字节的本地堆栈空间  
 30: 89 7d ec mov DWORD PTR [rbp-0x14],edi ; 将 edi 寄存器的值存储到 [rbp-0x14]  
 33: 48 89 75 e0 mov QWORD PTR [rbp-0x20],rsi ; 将 rsi 寄存器的值存储到 [rbp-0x20]  
 37: c7 45 f4 01 00 00 00 mov DWORD PTR [rbp-0xc],0x1 ; 将值 1 存储到 [rbp-0xc]  
 3e: c7 45 f8 01 00 00 00 mov DWORD PTR [rbp-0x8],0x1 ; 将值 1 存储到 [rbp-0x8]  
 45: 8b 55 f4 mov edx,DWORD PTR [rbp-0xc] ; 将 [rbp-0xc] 的值加载到 edx 寄存器  
 48: 8b 45 f8 mov eax,DWORD PTR [rbp-0x8] ; 将 [rbp-0x8] 的值加载到 eax 寄存器  
 4b: 01 d0 add eax,edx ; 将 edx 和 eax 寄存器中的值相加，结果存储在 eax 中  
 4d: 89 45 fc mov DWORD PTR [rbp-0x4],eax ; 将 eax 寄存器的值存储到 [rbp-0x4]  
 50: b8 00 00 00 00 mov eax,0x0 ; 将值 0 存储到 eax 寄存器  
 55: e8 00 00 00 00 call 5a <main+0x36> ; 调用函数 main+0x36  
 5a: b8 00 00 00 00 mov eax,0x0 ; 将值 0 存储到 eax 寄存器  
 5f: c9 leave ; 恢复栈帧并弹出栈顶  
 60: c3 ret ; 返回

当函数被被编译成汇编时，主要的操作就是将对应的 C 操作展开，并转换成对应的汇编指令。由于 C 的基本语法只有逻辑操作，运算操作和函数调用，这些都可以被固定的翻译成汇编。唯一的自动化操作就是栈内存的分配，对于编译器来说就是调整栈指针寄存器的操作。在非调用函数中，由于没有后续的子函数调用需求，可以省略栈内存的分配，直接使用后续的所有内存。但是在调用函数中需要调整栈指针，进行栈内存分配，在汇编中的体现就是增加了 sub rsp,0x20

分配栈内存指令。

有上面的汇编代码可见，C 语言被翻译成汇编时是以固定的格式来进行的，无论多么复杂的程序都会遵循这个规则，即编译 C 函数的步骤：

* 函数参数传递内存分配。
* 保存寄存器状态。
* 设置栈帧指针 (Frame Pointer)，保存当前的栈。
* 分配局部变量的空间。
* 保存调用者保存的寄存器。
* 函数体编译。
* 恢复寄存器状态。
* 释放局部变量的空间。
* 恢复栈帧指针。
* 函数返回。

### 1.2 C 程序库

C 语言的运行是基于栈帧的。在理想情况下，假如有一个准备好栈的图灵机，是可以直接将上面的汇编指令装机运行的。但是现实情况却是，第一：机器不会自动的准备好 C 语言环境，它只会准备好汇编运行环境；第二：一个程序的运行不仅仅需要他自己的代码，还需要一些共用的代码；第三：这段程序对于现实运转的机器环境来说，还是过于简陋，比如，没有异常处理代码，没有维持机器本身运行的代码。在现实中，一个机器想要运行起来，是需要很多维持自身运行的代码的。如果我们写程序时，每次都要写这么多维持自身运行的代码，那工作量岂不是非常的大。

针对这些问题，经过抽象，共享代码诞生了。所有的程序都能调用的代码就是共享代码，在实际使用中，这部分代码会在不同阶段来参与我们编译的程序的运行。而这些共享的代码被叫做代码库，在为 linux 中一般为 .a、.so 结尾的文件，而在 windows 中是以 .lib、.DLL 结尾的文件的存在。

首先在编译阶段，直接将这些代码参与进来进行编译，当编译完成后，这些共享代码就和程序绑定了一起，成为静态文件。而在编译阶段就参与的共享代码库就被称为静态库，在 Linux 中一般以 .a 结尾。静态库由于是在编译阶段就被和程序绑定到了一块，所以这样生成的程序体积会相应的大一些。

如果不想将共享代码加入到自己的程序中，这个时候还想调用共享代码怎么办？解决办法就是将一部分代码经过特殊编译载入到内存中，而程序只需要在操作系统的帮助上，寻找导共享代码部分，运行它就好了。而这部分代码是动态的被运行的，于是便称为的动态库。在 Linux 中一般以 .so 结尾。

### 1.3 C 程序编译连接

在前面的 GCC 章节说过，源码变成二进制可执行程序是需要经过 4 个步骤：预编译、编译、汇编、连接。但是实际的 GCC 处理过程中预编译、编译、汇编这三个步骤会被合并到一块，而最后的连接却要被单独的运行。这并不是最后一步不能和前面的合在一块，而是因为最后一步的连接面对的需求和前面的面临的需求更加复杂。

参考 1.1 中的源码，查看连接后的汇编代码：

nihao@nihao-z690:~/workdir/cfun$ gcc nihao.o -o nihao  
nihao@nihao-z690:~/workdir/cfun$ objdump -d -M intel nihao  
  
nihao： 文件格式 elf64-x86-64  
  
  
Disassembly of section .init:  
  
0000000000001000 <\_init>:  
 1000: f3 0f 1e fa endbr64   
 1004: 48 83 ec 08 sub rsp,0x8  
 1008: 48 8b 05 d9 2f 00 00 mov rax,QWORD PTR [rip+0x2fd9] # 3fe8 <\_\_gmon\_start\_\_@Base>  
 100f: 48 85 c0 test rax,rax  
 1012: 74 02 je 1016 <\_init+0x16>  
 1014: ff d0 call rax  
 1016: 48 83 c4 08 add rsp,0x8  
 101a: c3 ret   
  
Disassembly of section .plt:  
  
0000000000001020 <.plt>:  
 1020: ff 35 a2 2f 00 00 push QWORD PTR [rip+0x2fa2] # 3fc8 <\_GLOBAL\_OFFSET\_TABLE\_+0x8>  
 1026: f2 ff 25 a3 2f 00 00 bnd jmp QWORD PTR [rip+0x2fa3] # 3fd0 <\_GLOBAL\_OFFSET\_TABLE\_+0x10>  
 102d: 0f 1f 00 nop DWORD PTR [rax]  
  
Disassembly of section .plt.got:  
  
0000000000001030 <\_\_cxa\_finalize@plt>:  
 1030: f3 0f 1e fa endbr64   
 1034: f2 ff 25 bd 2f 00 00 bnd jmp QWORD PTR [rip+0x2fbd] # 3ff8 <\_\_cxa\_finalize@GLIBC\_2.2.5>  
 103b: 0f 1f 44 00 00 nop DWORD PTR [rax+rax\*1+0x0]  
  
Disassembly of section .text:  
  
0000000000001040 <\_start>:  
 1040: f3 0f 1e fa endbr64   
 1044: 31 ed xor ebp,ebp  
 1046: 49 89 d1 mov r9,rdx  
 1049: 5e pop rsi  
 104a: 48 89 e2 mov rdx,rsp  
 104d: 48 83 e4 f0 and rsp,0xfffffffffffffff0  
 1051: 50 push rax  
 1052: 54 push rsp  
 1053: 45 31 c0 xor r8d,r8d  
 1056: 31 c9 xor ecx,ecx  
 1058: 48 8d 3d ee 00 00 00 lea rdi,[rip+0xee] # 114d <main>  
 105f: ff 15 73 2f 00 00 call QWORD PTR [rip+0x2f73] # 3fd8 <\_\_libc\_start\_main@GLIBC\_2.34>  
 1065: f4 hlt   
 1066: 66 2e 0f 1f 84 00 00 cs nop WORD PTR [rax+rax\*1+0x0]  
 106d: 00 00 00   
  
0000000000001070 <deregister\_tm\_clones>:  
 1070: 48 8d 3d 99 2f 00 00 lea rdi,[rip+0x2f99] # 4010 <\_\_TMC\_END\_\_>  
 1077: 48 8d 05 92 2f 00 00 lea rax,[rip+0x2f92] # 4010 <\_\_TMC\_END\_\_>  
 107e: 48 39 f8 cmp rax,rdi  
 1081: 74 15 je 1098 <deregister\_tm\_clones+0x28>  
 1083: 48 8b 05 56 2f 00 00 mov rax,QWORD PTR [rip+0x2f56] # 3fe0 <\_ITM\_deregisterTMCloneTable@Base>  
 108a: 48 85 c0 test rax,rax  
 108d: 74 09 je 1098 <deregister\_tm\_clones+0x28>  
 108f: ff e0 jmp rax  
 1091: 0f 1f 80 00 00 00 00 nop DWORD PTR [rax+0x0]  
 1098: c3 ret   
 1099: 0f 1f 80 00 00 00 00 nop DWORD PTR [rax+0x0]  
  
00000000000010a0 <register\_tm\_clones>:  
 10a0: 48 8d 3d 69 2f 00 00 lea rdi,[rip+0x2f69] # 4010 <\_\_TMC\_END\_\_>  
 10a7: 48 8d 35 62 2f 00 00 lea rsi,[rip+0x2f62] # 4010 <\_\_TMC\_END\_\_>  
 10ae: 48 29 fe sub rsi,rdi  
 10b1: 48 89 f0 mov rax,rsi  
 10b4: 48 c1 ee 3f shr rsi,0x3f  
 10b8: 48 c1 f8 03 sar rax,0x3  
 10bc: 48 01 c6 add rsi,rax  
 10bf: 48 d1 fe sar rsi,1  
 10c2: 74 14 je 10d8 <register\_tm\_clones+0x38>  
 10c4: 48 8b 05 25 2f 00 00 mov rax,QWORD PTR [rip+0x2f25] # 3ff0 <\_ITM\_registerTMCloneTable@Base>  
 10cb: 48 85 c0 test rax,rax  
 10ce: 74 08 je 10d8 <register\_tm\_clones+0x38>  
 10d0: ff e0 jmp rax  
 10d2: 66 0f 1f 44 00 00 nop WORD PTR [rax+rax\*1+0x0]  
 10d8: c3 ret   
 10d9: 0f 1f 80 00 00 00 00 nop DWORD PTR [rax+0x0]  
  
00000000000010e0 <\_\_do\_global\_dtors\_aux>:  
 10e0: f3 0f 1e fa endbr64   
 10e4: 80 3d 25 2f 00 00 00 cmp BYTE PTR [rip+0x2f25],0x0 # 4010 <\_\_TMC\_END\_\_>  
 10eb: 75 2b jne 1118 <\_\_do\_global\_dtors\_aux+0x38>  
 10ed: 55 push rbp  
 10ee: 48 83 3d 02 2f 00 00 cmp QWORD PTR [rip+0x2f02],0x0 # 3ff8 <\_\_cxa\_finalize@GLIBC\_2.2.5>  
 10f5: 00   
 10f6: 48 89 e5 mov rbp,rsp  
 10f9: 74 0c je 1107 <\_\_do\_global\_dtors\_aux+0x27>  
 10fb: 48 8b 3d 06 2f 00 00 mov rdi,QWORD PTR [rip+0x2f06] # 4008 <\_\_dso\_handle>  
 1102: e8 29 ff ff ff call 1030 <\_\_cxa\_finalize@plt>  
 1107: e8 64 ff ff ff call 1070 <deregister\_tm\_clones>  
 110c: c6 05 fd 2e 00 00 01 mov BYTE PTR [rip+0x2efd],0x1 # 4010 <\_\_TMC\_END\_\_>  
 1113: 5d pop rbp  
 1114: c3 ret   
 1115: 0f 1f 00 nop DWORD PTR [rax]  
 1118: c3 ret   
 1119: 0f 1f 80 00 00 00 00 nop DWORD PTR [rax+0x0]  
  
0000000000001120 <frame\_dummy>:  
 1120: f3 0f 1e fa endbr64   
 1124: e9 77 ff ff ff jmp 10a0 <register\_tm\_clones>  
  
0000000000001129 <test>:  
 1129: f3 0f 1e fa endbr64   
 112d: 55 push rbp  
 112e: 48 89 e5 mov rbp,rsp  
 1131: c7 45 f4 01 00 00 00 mov DWORD PTR [rbp-0xc],0x1  
 1138: c7 45 f8 01 00 00 00 mov DWORD PTR [rbp-0x8],0x1  
 113f: 8b 55 f4 mov edx,DWORD PTR [rbp-0xc]  
 1142: 8b 45 f8 mov eax,DWORD PTR [rbp-0x8]  
 1145: 01 d0 add eax,edx  
 1147: 89 45 fc mov DWORD PTR [rbp-0x4],eax  
 114a: 90 nop  
 114b: 5d pop rbp  
 114c: c3 ret   
  
000000000000114d <main>:  
 114d: f3 0f 1e fa endbr64   
 1151: 55 push rbp  
 1152: 48 89 e5 mov rbp,rsp  
 1155: 48 83 ec 20 sub rsp,0x20  
 1159: 89 7d ec mov DWORD PTR [rbp-0x14],edi  
 115c: 48 89 75 e0 mov QWORD PTR [rbp-0x20],rsi  
 1160: c7 45 f4 01 00 00 00 mov DWORD PTR [rbp-0xc],0x1  
 1167: c7 45 f8 01 00 00 00 mov DWORD PTR [rbp-0x8],0x1  
 116e: 8b 55 f4 mov edx,DWORD PTR [rbp-0xc]  
 1171: 8b 45 f8 mov eax,DWORD PTR [rbp-0x8]  
 1174: 01 d0 add eax,edx  
 1176: 89 45 fc mov DWORD PTR [rbp-0x4],eax  
 1179: b8 00 00 00 00 mov eax,0x0  
 117e: e8 a6 ff ff ff call 1129 <test>  
 1183: b8 00 00 00 00 mov eax,0x0  
 1188: c9 leave   
 1189: c3 ret   
  
Disassembly of section .fini:  
  
000000000000118c <\_fini>:  
 118c: f3 0f 1e fa endbr64   
 1190: 48 83 ec 08 sub rsp,0x8  
 1194: 48 83 c4 08 add rsp,0x8  
 1198: c3 ret

经过连接处理的程序被扩展出了多段代码，这些代码段其实就是添加的支撑程序运行的代码。

更重要的是，源码的二进制文件需要经过连接才能成为真正能够运行二进制应用程序。程序主要的变化如下：

* **生成可执行文件或共享库**：连接器将多个编译后的目标文件（object files）合并成一个可执行文件（executable）或共享库（shared library）。这个文件包含了程序的全部代码和数据，以便在运行时执行。
* **符号解析**：连接器会解析程序中使用的所有符号引用，包括函数和变量。这包括将这些符号引用与它们的定义关联起来，以便程序可以正确找到它们。
* **地址重定位**：连接器会根据每个目标文件的地址信息，调整它们在合并后的可执行文件中的位置，以确保它们能够正确地访问和引用其他符号。这包括调整代码和数据的内存地址，以便它们不会发生冲突。
* **符号重命名**：如果多个目标文件中具有相同名称的符号（例如，多个文件中的相同函数名或全局变量名），连接器会对它们进行重命名，以避免冲突。这通常涉及到生成唯一的符号名称或重定位表。
* **删除未使用的符号**：连接器通常会删除未使用的函数和变量，以减小最终可执行文件的大小。这有助于减少程序的存储空间和加载时间。
* **生成重定位表**：连接器会生成一个包含需要在运行时进行地址重定位的信息的重定位表。这些表指示了哪些位置需要在程序加载时进行调整，以便正确地解析符号引用。
* **生成入口点**：对于可执行文件，连接器会确定程序的入口点（通常是 main 函数），从这里开始执行程序。连接器会设置程序的起始地址以执行入口点。
* **生成段和节**：连接器会将目标文件中的段和节合并成一个整体。这包括代码段、数据段、BSS 段等。每个段包含不同类型的代码和数据，连接器负责将它们组合到最终的可执行文件中。
* **处理共享库依赖**：如果程序依赖于共享库（动态链接库），连接器会解析这些依赖关系，并确保在运行时可以正确加载所需的共享库。
* **生成符号表**：可执行文件通常包含一个符号表，其中包含程序中的各个符号及其地址信息。这有助于调试和符号查找。

总之，连接是将动态库，静态库，多个源码文件的二进制源码文件的编译文件按照链接脚本进行组织起来，形成一个完整的应用程序。

### 1.4 extern C 关键字

extern 属于存储类型关键字，存储类型关键字有四个：auto、extern、register、static。其中 auto 一般是默认的，例如函数内部的局部变量如果没有其他的声明，那就属于 auto 修饰的。register 声明这个变量会被存放字寄存器中。static 声明这个是一个静态变量，存在于程序运行的整个周期，同时引用周期只在当前的范围内。

但 extern 属于比较特殊的一个关键字，extern 不定义变量，自然也不会修饰变量，extern 更像一个约定符，extern 的主要作用告诉编译器这个变量在其他地方属于全局变量，需要把他们引用过来。

#### C 中的引用

众所周知 C 程序中的源码文件不止一个，但是 C 语言的全局变量的限定是相对单个文件来说的。也就是一个源码中的全局变量默认情况下只能在本文件内部直接使用，在别的源码文件中是无法直接使用这个全局变量的。这个限制提高了变量的安全性，让不同文件内部的全局变量不会相互干扰。

但是在要跨文件引用同一个变量的需求下，就可以用 extern 来声明这个变量，这样就能在不同文件中共享同一个全局变量。同样的函数也是这样的操作，但是函数名声明的默认修饰就是 extern。

以一个简单的多文件源码为例：

// file: test.c  
  
int all\_test = 0;  
  
void test()  
{  
 int a = 1;  
 int b = 1;  
 all\_test = a + b;  
}

// file: main.c  
#include<stdio.h>  
void test();  
// extern void test();  
extern int all\_test;  
int main(int argc, char \*argv[])  
{  
 int a = 1;  
 int b = 1;  
 int c;  
 c = a + b;  
 printf("before all\_test:%d\n", all\_test);  
 test();  
 printf("after all\_test:%d\n", all\_test);  
 return 0;  
}

# file: Makefile  
nihao: main.o test.o  
 gcc main.o test.o -o nihao  
  
main.o:main.c  
 gcc -c main.c -o main.o  
  
test.o:test.c  
 gcc -c test.c -o test.o  
  
clean:  
 rm \*.o nihao

运行效果：

nihao@nihao-z690:~/workdir/cfun$ ls  
main.c Makefile test.c  
nihao@nihao-z690:~/workdir/cfun$ make  
gcc -c main.c -o main.o  
gcc -c test.c -o test.o  
gcc main.o test.o -o nihao  
nihao@nihao-z690:~/workdir/cfun$ ./nihao   
before all\_test:0  
after all\_test:2

在上面的 main.c 文件中，通过使用 extern 关键字来声明 all\_test 变量，就能在 main.c 文件中使用 test.c 的全局变量。

#### C++ 中的引用

extern 关键字的另一个作用就是当在 C++ 中期望使用 C 中的函数时，用来到声明导出 C 的函数。

C++ 只是在语法层面上兼容 C 的语法，但由于 C++ 要支持面向对象编程，类的重载和多态等技术，所以 C++ 在编译过程中和 C 还是有一定的区别的。这一明显的特征就是，C 文件用 gcc 编译，但 C++ 文件需要用 g++ 进行编译。

当 C++ 的源码中想要调用 C 的函数时，是不能直接进行调用的，需要用 extern 修饰导出后才能被 C++ 程序正常使用。

如下例子：

// file: main.cpp  
#include <stdio.h>  
  
extern int all\_test;  
extern "C"  
{  
 void test();  
}  
  
int main(int argc, char \*argv[])  
{  
 int a = 1;  
 int b = 1;  
 int c;  
 c = a + b;  
 printf("before all\_test:%d\n", all\_test);  
 test();  
 printf("after all\_test:%d\n", all\_test);  
 return 0;  
}

// file: test.c  
int all\_test = 0;  
  
void test()  
{  
 int a = 1;  
 int b = 1;  
 all\_test = a + b;  
  
}

nihao: main.o test.o  
 g++ main.o test.o -o nihao  
  
main.o:main.cpp  
 g++ -c main.cpp -o main.o  
  
test.o:test.c  
 gcc -c test.c -o test.o  
  
clean:  
 rm \*.o nihao

运行效果：

nihao@nihao-z690:~/workdir/cfun$ make  
g++ -c main.cpp -o main.o  
gcc -c test.c -o test.o  
g++ main.o test.o -o nihao  
nihao@nihao-z690:~/workdir/cfun$ ./nihao   
before all\_test:0  
after all\_test:2

## 2. 面向对象编程

面向对象编程（Object-Oriented Programming，简称OOP）和面向过程编程（Procedural Programming）是编程过程中两大编程范式，它们在代码组织和实现上有很大的区别。

### 2.1 概念解释

**面向过程（Procedure Oriented 简称 PO ：如 C 语言）：**

面向过程编程将问题分解为一系列的步骤或过程，每个过程执行特定的任务。主要的编程构建块是函数或过程，这些函数接受输入数据，执行操作，然后返回结果。程序的控制流程通常由顺序执行、条件语句和循环构成。

**面向对象（Object Oriented简称 OO ：如 C++，JAVA 等语言）：**

OOP 是一种基于对象的编程范式，它将数据和操作数据的方法封装到对象中。程序中的主要构建块是类和对象。类定义了对象的属性（数据）和方法（行为）。OOP 中的关键概念包括封装、继承和多态。封装：将数据和操作数据的方法封装在一个对象内部，隐藏了内部的实现细节，提供了接口供其他对象使用。继承，允许一个类继承另一个类的属性和方法，实现代码的重用和扩展。多态，允许不同的对象对相同的方法做出不同的响应，提高了代码的灵活性和可扩展性。

### 2.2 举个例子解释

面向对象和面向过程编程是编程中的两种思想，事实上，C 也能进行面向对象编程，其他语言也能面向过程编程，这取决于你需要做。面向对象的语言和 C 语言不同的，它们都简化了面向对象编程的操作步骤，自动的开发人员做了很多对象通用的操作。

#### 过程编程

以字符串编程为例，在编程中，经常做的事情就是字符串分割，字符串拼接，字符串中的参数提取，以及将数字字符串和数字进行相互转换。当然，这没什么问题，在标准库中就有字符串查找、分割、拼接、转换函数。

一个简单的题目：

从字符串 "name:xiaoming;age:15;sex:man;mathematical\_grade:A" 中提取出 name,age,sex,mathematical\_grade 的信息，然后按照特定格式打印出来。

输入：

mathematical\_grade:A;age:15;name:xiaoming;sex:man

输出：

name----age----sex----mathematical\_grade  
xiaoming----15----man----A

针对这个问题，我们使用 C 语言面向过程来编写：

// 包含标准C库头文件  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
  
int main(int argc, char \*argv[])  
{  
 char src\_str[128]; // 声明一个字符串数组，用于存储输入的源字符串  
 char name[32]; // 声明一个字符串数组，用于存储姓名  
 char sex[32]; // 声明一个字符串数组，用于存储性别  
 char age\_s[32]; // 声明一个字符串数组，用于存储年龄的字符串形式  
 int age; // 用于存储年龄的整数形式  
 char mathematical\_grade[32]; // 声明一个字符串数组，用于存储数学分数  
 char \*find\_ptr = NULL; // 用于存储 strtok\_r 函数的分隔状态指针  
 gets(src\_str); // 从标准输入中获取一行字符串，存储在 src\_str 中  
  
 // 使用 strtok\_r 函数分割输入字符串  
 char \*token = strtok\_r(src\_str, ";", &find\_ptr);  
 while (token != NULL)  
 {  
 char \*result = strstr(token, "name");  
 if (result = strstr(token, "name"), result != NULL)  
 {  
 strcpy(name, token + 5); // 复制姓名信息到 name 数组  
 }  
 else if (result = strstr(token, "sex"), result != NULL)  
 {  
 strcpy(sex, token + 4); // 复制性别信息到 sex 数组  
 }  
 else if (result = strstr(token, "age"), result != NULL)  
 {  
 strcpy(age\_s, token + 4); // 复制年龄信息到 age\_s 数组  
 age = atoi(age\_s); // 将年龄的字符串形式转换为整数  
 }  
 else if (result = strstr(token, "mathematical\_grade"), result != NULL)  
 {  
 strcpy(mathematical\_grade, token + 19); // 复制数学分数信息到 mathematical\_grade 数组  
 }  
  
 token = strtok\_r(NULL, ";", &find\_ptr); // 继续分割下一个字段  
 }  
  
 // 打印解析结果  
 printf("name----age----sex----mathematical\_grade\n");  
 printf("%s----%d----%s----%c\n", name, age, sex, mathematical\_grade[0]);  
  
 return 0;  
}

nihao@nihao-z690:~/workdir/cfun$ gcc main.c -o main  
nihao@nihao-z690:~/workdir/cfun$ ./main   
mathematical\_grade:A;age:15;name:xiaoming;sex:man  
name----age----sex----mathematical\_grade  
xiaoming----15----man----A

上面就是面向过程的编程，程序接收到参数，一步一步的不断的调用各种函数，最终得到想要的结果。

这个程序并不完美。第一、它限制了字符串的长度，第二、它限制了各项参数的长度。在这样的编程过程中，要小心翼翼的管理着指针，管理着内存，虽然如此，可还是可能出现错误的情况。如果想要更加完美，可能需要添加更多的代码，这样的编程是非常繁琐的，重复的。

#### 对象编程

还是刚才的那个问题，现在我们使用 C++ 的面向对象编程试试：

#include <iostream>  
#include <string>  
using namespace std;  
  
class stus  
{  
private:  
 string src\_str; // 保存包含键值对的字符串  
  
public:  
 stus(string &s) : src\_str(s) {} // 构造函数，接受一个字符串引用并初始化 src\_str 成员变量  
 string get\_val(string key)  
 {  
 string val; // 存储键对应的值  
 size\_t str\_len; // 用于确定子字符串的长度  
 size\_t start = src\_str.find(key); // 查找键的起始位置  
 if (string::npos != start) // 如果找到了键  
 {  
 size\_t end = src\_str.substr(start).find(';'); // 查找分号的位置  
 str\_len = string::npos != end ? end : src\_str.length() - start; // 如果找到了分号,找到的位置就是字符串的长度，如果没有找到分号，使用剩余字符串的长度  
 string item = src\_str.substr(start, str\_len); // 提取包含键值对的子字符串  
 val = item.substr(item.find(':') + 1); // 提取值部分  
 }  
 return val; // 返回键对应的值  
 }  
 ~stus() {} // 析构函数（在这个示例中未使用）  
};  
  
int main(int argc, char \*argv[])  
{  
 string src\_str ;  
 getline(cin, src\_str);  
 stus stu(src\_str); // 创建 stus 类的对象并初始化 src\_str 成员变量  
 stu.get\_val("name"); // 调用 get\_val 方法  
 cout << "name----age----sex----mathematical\_grade\n";  
 cout << stu.get\_val("name") << "---" // 输出姓名  
 << stu.get\_val("age") << "---" // 输出年龄  
 << stu.get\_val("sex") << "---" // 输出性别  
 << stu.get\_val("mathematical\_grade") // 输出数学分数  
 << endl;  
 return 0;  
}

nihao@nihao-z690:~/workdir/cfun$ g++ main.cpp -o main  
nihao@nihao-z690:~/workdir/cfun$ ./main   
mathematical\_grade:A;age:15;name:xiaoming;sex:man  
name----age----sex----mathematical\_grade  
xiaoming---15---man---A

我来看这段 C++ 程序，这段程序是采用面向对象的写法。构建了一个 stus 类，stus 对象中保存了要解析的字符串，有一个 get\_val 方法。这个类接受一个字符串来创建一个对象，然后就能根据对象去调用类中的方法。在面向对象的编程范式中，类内代码只用专著于类内对象的操作方法。而类的使用者只用专著于构造一个类，然后调用类内方法即可。类就是对这一类的操作对象抽象出来，写成通用代码，减少编程者的工作量。在对象内部可以自行管理内存占用，生命周期等等。大大节省了编程者的工作量。

#### 面向过程与面向对象的优缺点

总结来说：

**面向过程**

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、 Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护、易复用、易扩展

**面向对象**

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统 更加灵活、更加易于维护

缺点：性能比面向过程低

### 2.3 面向对象编程的特性

三大基本特性：封装，继承，多态

**封装**

封装，就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

**继承**

继承，指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过 “继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用父类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。

**多态**

多态，是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

### 2.4 五大基本原则：SPR, OCP, LSP, DIP, ISP

**单一职责原则SRP(Single Responsibility Principle)**

是指一个类的功能要单一，不能包罗万象。如同一个人一样，分配的工作不能太多，否则一天到晚虽然忙忙碌碌的，但效率却高不起来。

**开放封闭原则OCP(Open－Close Principle)**

一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。比如：一个网络模块，原来只服务端功能，而现在要加入客户端功能，那么应当在不用修改服务端功能代码的前提下，就能够增加客户端功能的实现代码，这要求在设计之初，就应当将服务端和客户端分开，公共部分抽象出来。

**里式替换原则LSP(the Liskov Substitution Principle LSP)**

子类应当可以替换父类并出现在父类能够出现的任何地方。比如：公司搞年度晚会，所有员工可以参加抽奖，那么不管是老员工还是新员工，也不管是总部员工还是外派员工，都应当可以参加抽奖，否则这公司就不和谐了。

**依赖倒置原则DIP(the Dependency Inversion Principle DIP)**

具体依赖抽象，上层依赖下层。假设B是较A低的模块，但B需要使用到A的功能，这个时候，B不应当直接使用A中的具体类： 而应当由B定义一个抽象接口，并由A来实现这个抽象接口，B只使用这个抽象接口：这样就达到了依赖倒置的目的，B也解除了对A的依赖，反过来是A依赖于B定义的抽象接口。通过上层模块难以避免依赖下层模块，假如B也直接依赖A的实现，那么就可能造成循环依赖。一个常见的问题就是编译A模块时需要直接包含到B模块的cpp文件，而编译B时同样要直接包含到A的cpp文件。

**接口分离原则ISP(the Interface Segregation Principle ISP)**

模块间要通过抽象接口隔离开，而不是通过具体的类强耦合起来。

<https://zhuanlan.zhihu.com/p/75265007>  
原文链接：<https://blog.csdn.net/m0_51233386/article/details/125877039>

# 第七章：交叉开发

交叉开发是指在通用高性能主机中开发、编译、调试目标嵌入式平台的程序，并部署到目标嵌入式平台中。

## 1、嵌入式 Linux 环境

嵌入式 Linux 的基本运行环境和主机 Linux 基本运行环境在结构上是相同的，区别在于在嵌入式平台中，运行环境被精简到只满足目标服务的运行。

### 1.1、Linux 完整操作系统的整体架构

作为现代操作系统内核，Linux 要承担向下操作硬件，向上服务程序的任务。由于 Linux 内核大部分是由 C 编写的，在之前的章节讲过，C 的运行是基于栈的，而 CPU 启动时是没有栈环境的。所以 Linux 需要一段建立 C 运行环境的汇编代码来完成 Linux 系统的引导。随着内核的升级迭代，计算机需要自由选择需要引导的内核。这时当初的那段引导 Linux 程序的需要承担更多的功能，于是它们慢慢的演变成今天的 Bootloader 内核引导程序。Bootloader 的最终目的是要引导内核启动，出与不同的引导需求，Bootloader 也拥有了更多与引导有关的功能。

内核被引导启动后，内核会自动的完成自己的初始化，然后将所有的硬件都调整到合适的状态等待应用调用。内核的最后一项事务是启动一个应用程序进程，这个应用程序将会是新环境下的首个

## 2、嵌入式 mcu 环境

### （1）、 交叉编译环境

一、什么是交叉编译

在一种计算机环境中运行的编译程序，能编译出在另外一种环境下运行的代码，我们就称这种编译器支持交叉编译。这个编译过程就叫交叉编译。

简单地说，就是在一个平台上生成另一个平台上的可执行代码。这里需要注意的是所谓平台，实际上包含两个概念：

1. 体系结构（Architecture）
2. 操作系统（OperatingSystem）。

同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。举例来说:

我们常说的x86 Linux平台实际上是Intel x86体系结构和Linux for x86操作系统的统称；而x86 WinNT平台实际上是Intel x86体系结构和Windows NT for x86操作系统的简称。  
  
要进行交叉编译，我们需要在主机平台上安装对应的交叉编译工具链（crosscompilation tool chain），然后用这个交叉编译工具链编译我们的源代码，最终生成可在目标平台上运行的代码。常见的交叉编译例子如下：

1、在Windows PC上，利用ADS（ARM 开发环境），使用armcc编译器，则可编译出针对ARM CPU的可执行代码。

2、在Linux PC上，利用arm-linux-gcc编译器，可编译出针对Linux ARM平台的可执行代码。

3、在Windows PC上，利用cygwin环境，运行arm-elf-gcc编译器，可编译出针对ARM CPU的可执行代码。

1.1、为什么要使用交叉编译

有时是因为目的平台上不允许或不能够安装我们所需要的编译器，而我们又需要这个编译器的某些特征；  
  
有时是因为目的平台上的资源贫乏，无法运行我们所需要编译器；  
  
有时又是因为目的平台还没有建立，连操作系统都没有，根本谈不上运行什么编译器。

1.2 、本地编译和交叉编译的比较

本地编译：本地编译可以理解为，在当前编译平台下，编译出来的程序只能放到当前平台下运行。平时我们常见的软件开发，都是属于本地编译。比如，我们在 x86 平台上，编写程序并编译成可执行程序。这种方式下，我们使用 x86 平台上的工具，开发针对 x86 平台本身的可执行程序，这个编译过程称为本地编译。  
  
交叉编译：交叉编译可以理解为，在当前编译平台下，编译出来的程序能运行在体系结构不同的另一种目标平台（该平台自己不能干，所以让其它平台来干）上，但是编译平台本身却不能运行该程序。比如，我们在 x86 平台上，编写程序并编译成能运行在 ARM 平台的程序，编译得到的程序在 x86 平台上是不能运行的，必须放到 ARM 平台上才能运行。

二、交叉编译环境搭建

NDK官网地址：NDK 下载 | Android NDK | Android Developers

本人采用ubuntu系统环境搭建，其他版本linux指令请自行百度参照，这里贴上我使用的ubuntu的下载连接 提取码：nbc3

注:谷歌在ndk-r18以后采用clang编译器，已经不支持gcc了，所以下载时要选择r19及以上的版本进行下载

1. 下载NDK，我这里选择当前最新版(北京时间:2021-01-23)

sudo wget <https://dl.google.com/android/repository/android-ndk-r22-linux-x86_64.zip>

1. 解压,执行以下命令，解压ndk（注:系统需已安装unzip）

unzip android-ndk-r22-linux-x86\_64.zip

1. 添加系统环境变量

* 3.1 执行sudo vim etc/profile命令打开配置文件  
    
  3.2 点击按键 i 进入编辑，通过键盘方向键，将光标移动到文件最下面  
    
  3.3 写入以下配置  
    
  3.4 修改NDK\_HOME的路径为你解压的真实路径  
    
  3.5 修改完毕，点击左上角按键Esc退出编辑，输入:wq保存  
    
  3.6 执行source /etc/profile命令使配置生效
* #ndk home  
  export NDK\_HOME=/root/ndk/android-ndk-r21d  
  export SYSROOT="$NDK\_HOME/toolchains/llvm/prebuilt/linux-x86\_64/sysroot/"
  export ANDROID\_GCC="$NDK\_HOME/toolchains/llvm/prebuilt/linux-x86\_64/bin/x86\_64-linux-android24-clang"  
  export PATH=NDK\_HOME

其中:

NDK\_HOME为ndk根路径  
  
ANDROID\_GCC为手机编译路径，稍后会用到

4.验证配置

执行$NDK\_HOME命令，如响应出NDK目录，即代表NDK环境配置成功

三、测试

1. 写main.cpp文件

执行sudo vim main.cpp命令，进入编辑模式，编辑内容如下，编辑完成，保存退出

#include <stdio.h>  
   
int main()  
{  
 printf("hello world\n");  
 return 0;  
}

保存完毕后，使用ls查看指令可以看到已经生成了main.cpp文件，如图

1. 写交叉编译脚本 generate.sh

由于这里命令比较短，也可直接在命令行里写，这里还是将内容写到脚本中，步骤如下:

2.1 执行sudo touch generate.sh命令创建generate.sh文件  
  
 2.2 执行sudo chmod +x generate.sh赋予执行权限  
 2.3 执行sudo vim generate.sh命令，进入编辑模式，编辑内容如下，编辑完成，保存退出  
  
$NDK\_HOME/toolchains/llvm/prebuilt/linux-x86\_64/bin/aarch64-linux-android26-clang++ main.cpp -o hello  
   
或者  
buidl(){  
 $NDK\_HOME/toolchains/llvm/prebuilt/linux-x86\_64/bin/aarch64-linux-android26-clang++ main.cpp -o hello   
}  
build

其中:

NDK\_HOME为自己解压的目录。  
 编译器要选择目的终端的架构，这里用的是arm64，所以是aarch64-linux-android。  
 编译器要选择android的api版本，这里用的是anroid 8.0.0，对应api是26。

按照官网说明，NDK在r21之后，NDK 默认安装的工具链可供使用。  
 可以不需要使用 make\_standalone\_toolchain.py 脚本生成独立工具链来使用。  
 这样使用自带的工具链就比较方便，不用再配置sysroot等编译选项。

1. 交叉编译出可执行程序hello  
   执行./generate.sh即可

注：此时编译生成的文件是无法在x86架构的linux上运行的

1. push到手机上执行

执行adb push hello /data/local/tmp

1. 给hello执行权限  
   adb shell  
   cd /data/local/tmp  
   chmod +x hello

6.执行hello

./hello

可以看到输出  
hello liera  
————————————————  
版权声明：本文为CSDN博主「程序课代表」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/qq_34508943/article/details/113032733>

#### STM32 交叉编译环境

常用的STM32的软件开发方法都是基于MDK keil或IAR集成开发环境，但以上两个集成开发环境软件都是需要收费的，且价格较为昂贵。本节介绍一种在ubuntu上安装arm gcc（arm-none-eabi）的方式，用于编译STM32的程序。

1.在arm官网下载gcc-arm 8.2的版本(注意:下载arm-none-eabi的版本，由于stm32属于arm cortex -M架构)，网址如下：Downloads | GNU Arm Embedded Toolchain Downloads – Arm Developer  
图1 从ARM官方下载gcc编译器

2.拷贝下载的交叉编译器到ubuntu系统的/opt/pkg/gcc\_linaro目录，如下图所示：  
图2 拷贝gcc交叉编译器到ubuntu交叉编译器安装目录

3.解压gcc到工作目录

tar -xjvf gcc-arm-none-eabi-8-2018-q4-major-linux.tar.bz2  
图3 gcc压缩包解压

4.运行动态配置环境变量，再查看gcc的版本，如下所示，说明gcc安装成功。

export PATH=/opt/pkg/gcc\_linaro/gcc-arm-none-eabi-8-2018-q4-major/bin:$PATH

arm-eabi-gcc -v

————————————————  
版权声明：本文为CSDN博主「少林and叔叔」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/q544344318/article/details/129211437>

## 1. 嵌入式 Linux 系统环境组成

Linux内核由三部分构成：

* Bootloader：启动引导系统（可执行文件）
* Kernel：内核（可执行文件）
* Root File System：根文件系统

<https://www.cnblogs.com/bruce1992/p/17574939.html>

## 2. 网络下载 rftp nfs dnw

## 3. ext4 文件系统

[Linux文件系统详解](https://zhuanlan.zhihu.com/p/581250390)

[linux进程管理---实时调度](https://zhuanlan.zhihu.com/p/580503043)

[linux内核内存管理-缺页异常](https://zhuanlan.zhihu.com/p/579482414)

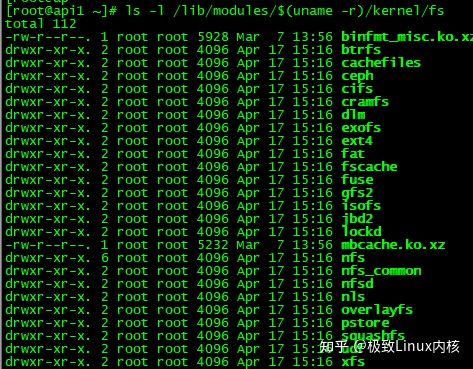
[linux内核内存管理-brk系统调用](https://zhuanlan.zhihu.com/p/579405096)

## **Ext**

全称Linux extended file system, extfs，即Linux扩展文件系统，Ext2就代表第二代文件扩展系统，Ext3/Ext4以此类推，它们都是Ext2的升级版，只不过为了快速恢复文件系统，减少一致性检查的时间，增加了日志功能，所以Ext2被称为**索引式文件系统**，而Ext3/Ext4被称为**日志式文件系统**。

备注：Linux支持很多文件系统，包括网络文件系统(NFS)、Windows的Fat文件系统。

查看Linux支持的文件系统：ls -l /lib/modules/$(uname -r)/kernel/fs



查看Linux支持的文件系统(已载入到内存中)：cat /proc/filesystems



## **核心设计**

### **数据存放区**

这些元素相对稳定，磁盘格式化后，就固定下来了。

* inode(索引节点)  
  记录文件的权限、属性和数据所在块block的号码，每个文件都有且仅有一个的inode，每个inode都有自己的编号，可以把inode简单地理解为**文档索引**。

备注：在磁盘格式化后，inode的大小和数量都已经固定了，大小均为128Bytes(新的Ext4和xfs为258Bytes)。读取文件时，先读取inode里面记录的文件属性和权限，匹配正确后，才会读取文件内容(block)。**在Linux系统中，实际使用inode来识别文件，而不是文件名，类似于用户标识和昵称的设计**。

* inode table  
  存储文件系统的所有inode编号的表格
* block(数据区块)  
  存储的文件内容，也叫数据区块(data block)，每个block都有自己的编号，Ext2支持的单位block容量仅为**1k、2k、4k**。

备注：为了方便inode的记录，在磁盘格式化后，block的大小都已经固定了。每一个块只能存放一个文件的数据，若文件太大，将占用多个block；若文件太小，block剩余空间就不能被使用了，就会导致**磁盘空间浪费**，所以**在磁盘分区后，文件系统格式化前，请先仔细想想文件系统的预计使用情况**。

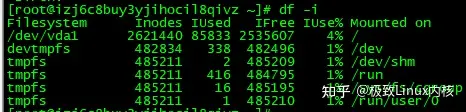
### **查看文件或者文件系统的状态**

stat [options] [filename]



**查看系统各个文件系统的inode使用情况**

df -i



### **中介数据(metadata)**

这些元素是为了维持文件系统状态而设计出来的，当新增、编辑、删除文档时，都需要变更这些状态信息。

* superblock(超级块)  
  记录文件系统(filesystem)的整体信息，包括inode/block的总量、使用量、剩余量、大小、以及文件系统的格式和相关信息。

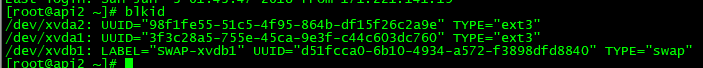
备注：整个文件系统的基本信息全部记录在superblock，它的大小一般为1024Bytes，如果它死掉，将会花费大量的时间去补救哦！！！

* block group(区块群组)  
  试想一下，假如我们的磁盘容量高达数百G，当我们格式后，inode和block会非常庞大，为了便于管理，Ext文件系统在格式化时，引入了区块群组(block group)的概念，每个区块群组都保持独立inode/block/superblock，拥有固定数量的block，这样就分成了一群一群最基础的子文件系统。

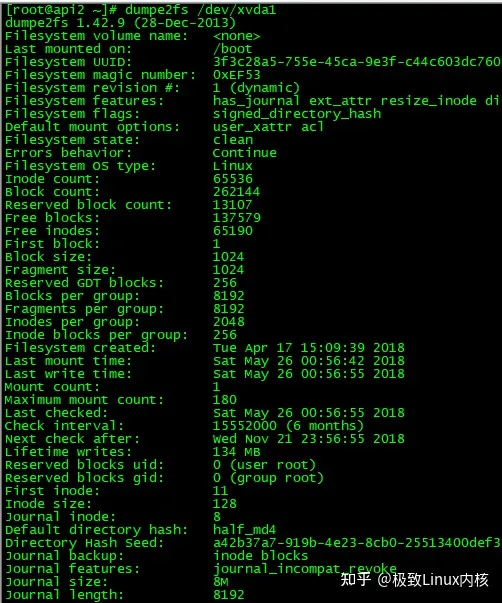
备注：superblock对于文件系统太重要了，但是文件系统的superblock又只有一个，所以除了第一个block group含有superblock外，后续block group都可能会含有备份的superblock，目的就是为了避免superblock单点无法救援的问题。

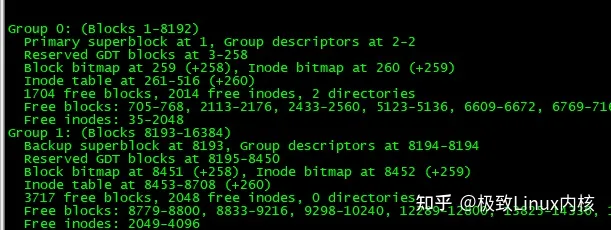
* block bitmap(区块对照表)  
  一个block只能被一个文件使用，当我们新增文件时，肯定需要使用新的block来记录文件数据。那么如何快速地知道，哪些block是新的？哪些block是已经使用了的？block bitmap就是这样被设计出来，记录所有使用和未使用的block号码。同样的，当我们删除文件时，先从block bitmap中找到对应的block号码，然后更新标志为未使用，最后释放block。
* inode bitmap(inode 对照表)  
  和block bitmap一样的设计理念，只不过它记录地是已使用和未使用的inode号码，这里就不再敖述了。
* group descriptor  
  描述每个区段(block group)开始和结束的block号码，以及说明每个区段(inodemap、blockmap、inode table)分别介于哪些block号码之间。

列出目前系统所有被格式化的设备：blkid



挑选一个已格式化好的设备，查看文件系统的详细信息：dumpe2fs /dev/vda1





备注：通过上面的Magic签名为0xEF53，说明我们的磁盘分区是一个标准的ext2和ext3文件系统。类似于通过文件开头的Magic，可以判断文件类型一样。

## **示例说明**

### **1. inode的作用**

当用户搜索或者访问一个文件时，UNIX 系统通过 inode 表查找正确的 inode 编号。在找到 inode 编号之后，相关的命令才可以访问该 inode ，并对其进行适当的更改。

### **示例**

例如使用vi来编辑一个文件。当您键入vi<filename>时，在inode表中找到inode编号之后，才允许您打开该inode 。在 vi 的编辑会话期间，更改了该inode中的某些属性，当您完成操作并键入 :wq 时，将关闭并释放该 inode 。通过这种方式，如果两个用户试图对同一个文件进行编辑， inode 已经在第一个编辑会话期间分配给了另一个用户 ID (UID)，因此第二个编辑任务就必须等待，直到该 inode 释放为止。

备注：大家可以参考百度百科。

### **2. block的重要性**

通过上面的分析，我们知道block是文件数据存储的原子单位，且每一个block只能存储一个文件的数据。当格式化一个文件系统时，如果选择不当，就会造成大量的磁盘空间浪费。

### **示例**

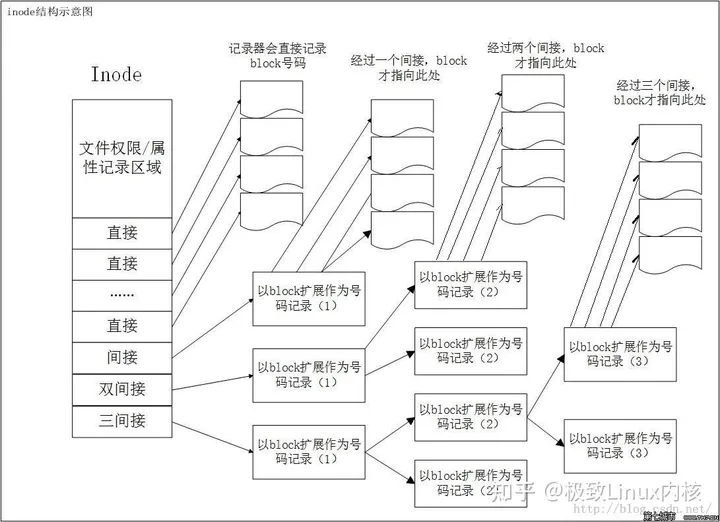
假如文件系统选择的block为4k，存储10000个小文件，每个500bytes，请问此时浪费了多少磁盘空间容量？  
每个文件浪费的磁盘容量 = 4096 - 500 = 3596bytes，10000个文件浪费的磁盘容量 = 10000 \* 3596 ~=34M，实际文件容量 = 10000 \* 500 ~=4.7M，没有对比就没有伤害啊，实际存储容量不到5M，就浪费了34M，浪费率680%，而且文件越多浪费越严重。

备注：从原理上分析，只有当实际文件容量刚好等于系统最小存储单位容量时，磁盘不会存在浪费的情况，但这是理想情况，那么我们选择最小的block不就行了，没毛病。不过，此时又有新的问题产生了，大型文件会占用过多的block，造成inode记录过多的block号码，文件系统的读写性能就会下降，所以说凡事都要有个度，把握好这个度，才能从整体上提高文件系统的性能和利用率。

### **3. inode和block与文件大小的关系**

数据实际存储在block，为了能够快速地读取文件，每个文件都对应一个inode索引文件，记录所有的block编号，但是inode的大小只有128bytes或256bytes(ext4)，如果一个文件太大，block数量很有可能会超过inode可记录的数量，为此，inode记录block号码的区域被设计为12个直接、一个间接、一个双间接、一个三间接记录区。

备注：所谓的间接就是拿一个block来作为block号码记录区，只有最后一个间接才会真正用来记录block号码，其他的间接层，都只是依次引用。

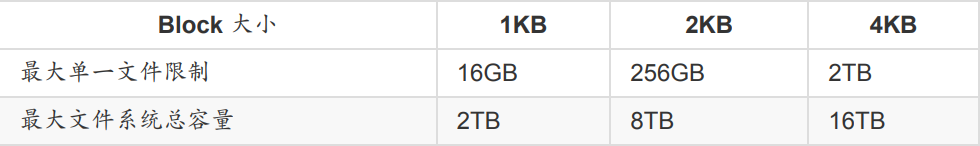


### **计算单文件最大容量**

每个block号码为数字，需要占据4bytes。

* **假设block的单位容量为1K，每个block能记录的block号码为1k/4=256**。
  + 12个直接容量 = 12 \* 1k =12k
  + 单间接容量 = 256 \* 1k = 256k
  + 双间接容量 = 256 \* 256 \* 1k = 65536k
  + 三间接容量 = 256 \* 256 \* 256 \* 1k = 16777216k
  + 单文件最大总量 = 12个直接容量 + 单间接容量 + 双间接容量 + 三间接容量 = (12 + 256 + 65536 + 16777216) / (1024 \* 1024) = 16.06G
* **假设block的单位容量为2K，每个block能记录的block号码为2k/4=512**。
  + 12个直接容量 = 12 \* 2k =24k
  + 单间接容量 = 512 \* 2k = 1024k
  + 双间接容量 = 512 \* 512 \* 2k = 524288k
  + 三间接容量 = 512 \* 512 \* 512 \* 2k = 268435456k
  + 单文件最大总量 = 12个直接容量 + 单间接容量 + 双间接容量 + 三间接容量 = (24 + 1024 + 524288 + 268435456) / (1024 \* 1024) = 256.50G
* **假设block的单位容量为4K，每个block能记录的block号码为4k/4=1024**。
  + 同理，单文件最大总量 = 12个直接容量 + 单间接容量 + 双间接容量 + 三间接容量 = 4.00T

Linux标准的文件系统限制表



备注：当block单位容量为4K时，由于文件系统本身的限制(2T)，所以才与计算的结果不太吻合。

## **查看磁盘和文档的容量**

### **1. 查看文件系统的整体磁盘容量**

df [-ahikHTm] [目录或文件名]



**2. 查看目录和文件容量**

du [options] []

**查看目录geekbuying下所有目录的容量**

du -sm geekbuying/\*



**统计当前目录容量**

du -sm 单位M



## **总结**

Ext家族是Linux支持度最广、最完整的文件系统，当我们格式化磁盘后，就已经为我们规划好了所有的inode/block/metadate等数据，这样系统可以直接使用，不需要再进行动态的配置，这也是它最优秀的特点，不过这也是它最显著的缺点，磁盘容量越大，格式化越慢，centos7.x已经选用xfs作为默认文件系统，xfs是一种适合大容量磁盘和处理巨型文件的文件系统。

<https://zhuanlan.zhihu.com/p/649920886>

# 第八章：系统内核

## 1. 嵌入式操作系统的构建方法

spl-》uboot-》kernel-》rootfs

## 2. uboot 的功能和作用

## uboot简介

uboot是从FADSROM、8xxROM、PPCBOOT逐步发展演化而来的。uboot发展至今，已经可以实现非常多的功能， 在操作系统方面，它不仅支持嵌入式Linux系统的引导，还支持NetBSD,VxWorks, QNX, RTEMS, ARTOS, LynxOS, Android等嵌入式操作系统的引导。在CPU架构方面， uboot支持PowerPC、MIPS、x86、ARM、NIOS、XScale等诸多常用系列的处理器。

一般来说BootLoader必须提供系统上电时的初始化代码，在系统上电时初始化相关环境后， BootLoader需要引导完整的操作系统，然后将控制器交给操作系统。 简单来说BootLoader是一段小程序，它在系统上电时执行，通过这段小程序可以将硬件 设备进行初始化，如CPU、SDRAM、Flash、串口、网络等，初始化完毕后调用操作系统内核。

## 启动uboot

以野火imx6ull 2020.10版本uboot为例，介绍uboot的使用， 在开发板上电uboot启动kernel之前按下键盘的空格或回车键, 进入uboot的命令模式。如下所示

uboot 2020.10-g4f79a1a2 (Feb 20 2021 - 11:25:18 +0800)  
  
CPU: Freescale i.MX6ULL rev1.1 792 MHz (running at 396 MHz)  
CPU: Industrial temperature grade (-40C to 105C) at 49C  
Reset cause: WDOG  
Model: Freescale i.MX6 UltraLiteLite 14x14 EVK Board  
Board: MX6ULL 14x14 EVK  
DRAM: 512 MiB  
MMC: FSL\_SDHC: 0, FSL\_SDHC: 1  
Loading Environment from MMC... OK  
In: serial  
Out: serial  
Err: serial  
Net: eth1: ethernet@20b4000 [PRIME]Could not get PHY for FEC0: addr 2  
  
Hit any key to stop autoboot: 0  
=>

可以看出uboot打印出了板子的一些基本信息，包括CPU、内存等信息。

## uboot命令

当不清楚uboot支持什么命令时， 可输入**help**或**?**可查看uboot支持的命令列表，如下所示

? - alias for 'help'  
base - print or set address offset  
bdinfo - print Board Info structure  
blkcache - block cache diagnostics and control  
bmode - sd1|sd2|qspi1|normal|usb|sata|ecspi1:0|ecspi1:1|ecspi1:2|ecspi1:3|esdhc1|esdhc2|esdhc3|esdhc4  
bmode - getprisec  
boot - boot default, i.e., run 'bootcmd'  
bootd - boot default, i.e., run 'bootcmd'  
bootefi - Boots an EFI payload from memory  
bootelf - Boot from an ELF image in memory  
bootm - boot application image from memory  
bootp - boot image via network using BOOTP/TFTP protocol  
bootvx - Boot vxWorks from an ELF image  
bootz - boot Linux zImage image from memory  
clocks - display clocks  
cmp - memory compare  
coninfo - print console devices and information  
cp - memory copy  
crc32 - checksum calculation  
dcache - enable or disable data cache  
dhcp - boot image via network using DHCP/TFTP protocol  
dm - Driver model low level access  
dtfile - dtoverlay utility commands  
echo - echo args to console  
editenv - edit environment variable  
env - environment handling commands  
erase - erase FLASH memory  
exit - exit script  
ext2load - load binary file from a Ext2 filesystem  
ext2ls - list files in a directory (default /)  
ext4load - load binary file from a Ext4 filesystem  
ext4ls - list files in a directory (default /)  
ext4size - determine a file's size  
ext4write - create a file in the root directory  
false - do nothing, unsuccessfully  
fatinfo - print information about filesystem  
fatload - load binary file from a dos filesystem  
fatls - list files in a directory (default /)  
fatmkdir - create a directory  
fatrm - delete a file  
fatsize - determine a file's size  
fatwrite - write file into a dos filesystem  
fdt - flattened device tree utility commands  
flinfo - print FLASH memory information  
fstype - Look up a filesystem type  
fstypes - List supported filesystem types  
fuse - Fuse sub-system  
go - start application at address 'addr'  
gpio - query and control gpio pins  
help - print command description/usage  
i2c - I2C sub-system  
icache - enable or disable instruction cache  
iminfo - print header information for application image  
imxtract - extract a part of a multi-image  
itest - return true/false on integer compare  
ln - Create a symbolic link  
load - load binary file from a filesystem  
loadb - load binary file over serial line (kermit mode)  
loads - load S-Record file over serial line  
loadx - load binary file over serial line (xmodem mode)  
loady - load binary file over serial line (ymodem mode)  
loop - infinite loop on address range  
ls - list files in a directory (default /)  
md - memory display  
mm - memory modify (auto-incrementing address)  
mmc - MMC sub system  
mmcinfo - display MMC info  
mtest - simple RAM read/write test  
mw - memory write (fill)  
nfs - boot image via network using NFS protocol  
nm - memory modify (constant address)  
panic - Panic with optional message  
ping - send ICMP ECHO\_REQUEST to network host  
pinmux - show pin-controller muxing  
printenv - print environment variables  
protect - enable or disable FLASH write protection  
random - fill memory with random pattern  
reset - Perform RESET of the CPU  
run - run commands in an environment variable  
save - save file to a filesystem  
saveenv - save environment variables to persistent storage  
setenv - set environment variables  
setexpr - set environment variable as the result of eval expression  
sf - SPI flash sub-system  
showvar - print local hushshell variables  
size - determine a file's size  
sleep - delay execution for some time  
source - run script from memory  
test - minimal test like /bin/sh  
tftpboot - boot image via network using TFTP protocol  
true - do nothing, successfully  
version - print monitor, compiler and linker version

可看到uboot支持很多的命令，功能十分强大，与linux类似，在执行某条uboot命令时， 可使用 *tab* 自动补全命令，在没有命令名冲突的情况下可以使用命令的前几个字母作为命令的输入， 例如想要执行 **reset** 命令，输入 *res* 或 *re* 即可。

当需要具体使用哪个命令时，可使用 **“help 命令”** 或 **“? 命令”** 的方式查看具体命令的使用说明。以 **“help printenv”** 为例，

=> help printenv  
printenv - print environment variables  
  
Usage:  
printenv [-a]  
 - print [all] values of all environment variables  
printenv name ...  
 - print value of environment variable 'name'

可以看到printenv命令的说明以及使用方法。

关于uboot命令的使用可参考uboot官方链接： [http://www.denx.de/wiki/DULG/Manual](https://link.zhihu.com/?target=http%3A//www.denx.de/wiki/DULG/Manual) **5.9. uboot Command Line Interface** 部分。

## **uboot常见命令**

uboot命令众多，下面介绍常用的uboot命令,详细的uboot命令使用方式请使用**help [命令]**查看。



以上为用户较为常用使用的部分命令，具体的使用方式可使用**help [命令]**查看。

## **mmc命令**

mmc命令能够对如sd卡以及emmc类的存储介质进行操作，以下进行简单说明， 对于mmc命令不熟悉可使用**help mmc**查看相关命令的帮助，常用功能如下所示



### **查看mmc设备**

使用**mmc list**查看板子上相关设备，本人使用的是emmc版本的开发板，并插入了sd卡， 可看到打印信息如下。

=> mmc list  
FSL\_SDHC: 0 (SD)  
FSL\_SDHC: 1

使用**mmc dev**查看当前使用的mmc设备，打印信息如下，可看到当前设备为mmc0即sd卡。

=> mmc dev  
switch to partitions #0, OK  
mmc0 is current device

可使用**mmc dev 1**命令切换当前设备为emmc设备

使用**mmc info**查看当前使用的sd卡设备的信息。

=> mmc info  
Device: FSL\_SDHC  
Manufacturer ID: 3  
OEM: 5344  
Name: SC16G  
Bus Speed: 50000000  
Mode: SD High Speed (50MHz)  
Rd Block Len: 512  
SD version 3.0  
High Capacity: Yes  
Capacity: 14.8 GiB  
Bus Width: 4-bit  
Erase Group Size: 512 Bytes

### **查看分区信息**

使用**mmc part**列出当前mmc设备分区

=> mmc part  
  
 Partition Map for MMC device 0 -- Partition Type: DOS  
  
 Part Start Sector Num Sectors UUID Type  
 1 8192 81920 2ba51413-01 0e Boot  
 2 90112 31026176 2ba51413-02 83

### **mmc操作**

可使用以下指令对mmc存储介质以block为操作单位进行读、写、擦除操作，根据上面sd的信息可知， 一个block为512字节。

mmc read addr blk# cnt *#读*  
 mmc write addr blk# cnt *#写*  
 mmc erase blk# cnt *#擦除*

简单实例：将mmc设备的的前10个block读取到0x80000000地址处:**mmc read 0x80000000 0 10**

## 文件系统操作命令

uboot能够对ext2/3/4以及fat文件系统设备进行访问， 可使用fstype命令判断存储介质分区使用的是什么类型的文件系统。 以mmc介质为例，判断sd的两个分区的文件系统类型

=> fstype mmc 0:1  
fat  
=> fstype mmc 0:2  
ext4

野火linux开发板具有U盘功能，能够通过PC以访问U盘的形式访问/boot目录下的文件， /boot目录对应的即是 mmc 0:1 分区。

而ext4分区对应的则是Debian根文件系统。

知道了文件系统的类型即可使用相对应的命令对分区内容进行操作了。

### FAT格式文件系统

uboot提供了能够对于FAT格式文件系统操作的各个指令， 如下所示，详细可通过**help [命令]**查看



**查看文件系统信息**

使用fatinfo查看文件系统信息，打印信息如下

=> fatinfo mmc 0:1  
Interface: MMC  
 Device 0: Vendor: Man 000003 Snr 6d8b9601 Rev: 1.0 Prod: SC16G▒  
 Type: Removable Hard Disk  
 Capacity: 15193.5 MB = 14.8 GB (31116288 x 512)  
Filesystem: FAT16 "BOOT "

**查看分区下的文件目录**

使用fatls查看分区下的文件目录，打印信息如下

=> fatls mmc 0:1  
 38 ID.txt  
 kernel/  
 2418 uEnv.txt  
 577 SOC.sh  
 boot/  
 34 autorun.inf  
 System Volume Information/  
 dtbs/  
 1472 BOOTEX.LOG  
  
 5 file(s), 4 dir(s)

若想要查看其它目录下的文件列表，只要加上文件路径即可，如下

=> fatls mmc 0:1 kernel/  
 ./  
 ../  
 9187912 vmlinuz-4.19.35-imx6  
 143341 config-4.19.35-imx6  
 3104898 System.map-4.19.35-imx6  
 5160838 initrd.img-4.19.35-imx6  
  
 4 file(s), 2 dir(s)

**读取文件内容**

使用fatload将FAT文件系统的文件加载到内存中，如下所示

=> fatload mmc 0:1 0x80000000 uEnv.txt  
 2418 bytes read in 13 ms (181.6 KiB/s)

可使用md命令查看0x80000000内存中的部分数据内容

=> md.b 0x80000000 0x80  
80000000: 23 44 6f 63 73 3a 20 68 74 74 70 73 3a 2f 2f 65 #Docs: https://e  
80000010: 6d 62 65 64 2d 6c 69 6e 75 78 2d 74 75 74 6f 72 mbed-linux-tutor  
80000020: 69 61 6c 2e 72 65 61 64 74 68 65 64 6f 63 73 2e ial.readthedocs.  
80000030: 69 6f 2f 7a 68 5f 43 4e 2f 6c 61 74 65 73 74 2f io/zh\_CN/latest/  
80000040: 52 45 41 44 4d 45 2e 68 74 6d 6c 0a 0a 75 6e 61 README.html..una  
80000050: 6d 65 5f 72 3d 34 2e 31 39 2e 33 35 2d 69 6d 78 me\_r=4.19.35-imx  
80000060: 36 0a 23 75 75 69 64 3d 0a 6d 6d 63 5f 64 74 62 6.#uuid=.mmc\_dtb  
80000070: 3d 69 6d 78 36 75 6c 6c 2d 6d 6d 63 2d 6e 70 69 =imx6ull-mmc-npi

读取到的内容就是我们日常在/boot/uEnv.txt文件中的看到的内容了。

**FAT文件系统其他操作**

uboot还提供了FAT文件系统的其他操作命令，可用于创建目录、写入、删除等操作， 通常情况下在uboot中需要使用这类命令的场景很少，简单介绍如下：

* fatmkdir：创建目录
* fatrm：删除文件

### ext4格式文件系统

ext4文件系统的命令使用方式和FAT使用方式相似，仅命令名不同， uboot提供的ext文件系统命令如下



ext4文件系统操作

下面以将/etc/apt/sources.list的内容读取到内存实例，简单说明uboot对ext4文件系统操作。

1.查看/etc/apt目录中的文件内容，

=> ext4ls mmc 0:2 /etc/apt/  
 <DIR> 1024 .  
 <DIR> 5120 ..  
 <DIR> 3072 trusted.gpg.d  
 865 sources.list  
 2175 trusted.gpg  
 <DIR> 1024 auth.conf.d  
 <DIR> 1024 apt.conf.d  
 <DIR> 1024 sources.list.d  
 <DIR> 1024 preferences.d  
 382 trusted.gpg~

1. 将/etc/apt/sources.list 文件读取到内存地址0x8000 0000处

=> ext4load mmc 0:2 0x80000000 /etc/apt/sources.list  
865 bytes read in 30 ms (27.3 KiB/s)

查看内存0x8000 0000的部分数据内存

=> md.b 0x80000000 0x80  
80000000: 64 65 62 20 68 74 74 70 3a 2f 2f 6d 69 72 72 6f deb http://mirro  
80000010: 72 73 2e 74 75 6e 61 2e 74 73 69 6e 67 68 75 61 rs.tuna.tsinghua  
80000020: 2e 65 64 75 2e 63 6e 2f 64 65 62 69 61 6e 20 62 .edu.cn/debian b  
80000030: 75 73 74 65 72 20 6d 61 69 6e 20 63 6f 6e 74 72 uster main contr  
80000040: 69 62 20 6e 6f 6e 2d 66 72 65 65 0a 23 64 65 62 ib non-free.#deb  
80000050: 2d 73 72 63 20 68 74 74 70 3a 2f 2f 6d 69 72 72 -src http://mirr  
80000060: 6f 72 73 2e 74 75 6e 61 2e 74 73 69 6e 67 68 75 ors.tuna.tsinghu  
80000070: 61 2e 65 64 75 2e 63 6e 2f 64 65 62 69 61 6e 20 a.edu.cn/debian

## uboot启动内核过程

bootcmd与bootargs可以说是uboot最重要的两个环境参数， uboot执行完毕之后，如果没有按下回车，则会自动执行bootcmd命环境参数里的内容， 而bootargs则是传递给内核的启动参数。

使用 **printenv bootcmd** 可查看bootcmd的内容。

=> printenv bootcmd  
 bootcmd=run distro\_bootcmd

bootcmd执行了distro\_bootcmd，同样可以使用 printenv distro\_bootcmd 查看distro\_bootcmd的内容如下

=> printenv distro\_bootcmd  
distro\_bootcmd=for target in ${boot\_targets}; do run bootcmd\_${target}; done  
  
#boot\_targets的值如下  
boot\_targets=mmc0 mmc1

也就是说distro\_bootcmd会执行**bootcmd\_mmc0、bootcmd\_mmc1**这两个环境参数， 在前面我们知道，mmc0表示的sd卡的存储设备，mmc1表示的emmc设备， 也就是说当sd卡插在板子时，若sd卡装有系统则会优先从sd卡内启动。

=> printenv bootcmd\_mmc0  
 bootcmd\_mmc0=setenv devtype mmc; setenv mmcdev 0; setenv bootpart 0:1 ; setenv rootfpart 0:2 ; run boot  
 => printenv bootcmd\_mmc1  
 bootcmd\_mmc1=setenv devtype mmc; setenv mmcdev 1; setenv bootpart 1:1 ; setenv rootfpart 1:2 ; run boot

bootcmd\_mmc0与bootcmd\_mmc1均设置各自**devtype、mmcdev、bootpart、rootfpart**环境参数的值， 最后运行 boot 环境参数，boot内容如下

"boot=mmc check;${devtype} dev ${mmcdev};mmc rescan; " \  
 "echo loading [${devtype} ${bootpart}] /uEnv.txt ...; "\  
 "if run loaduEnv; then " \  
 "run importbootenv;" \  
 "if test ${second\_flash} = emmc; then " \  
 "setenv dtb ${mmc\_dtb};" \  
 "setenv storage\_media init=/opt/scripts/tools/eMMC/init-eMMC-flasher-v3.sh;" \  
 "else " \  
 "setenv dtb ${nand\_dtb};" \  
 "setenv storage\_media init=/opt/scripts/tools/Nand/init-Nand-flasher-v1.sh;" \  
 "fi; " \  
 "if test -n ${flash\_firmware}; then " \  
 "echo setting flash firmware...;" \  
 "setenv flashtype ${storage\_media};" \  
 "fi;" \  
 "run args\_mmc\_old;" \  
 "echo loading vmlinuz-${uname\_r} ...; "\  
 "load ${devtype} ${bootpart} 0x80800000 /kernel/vmlinuz-${uname\_r};"\  
 "echo loading ${dtb} ...; "\  
 "load ${devtype} ${rootfpart} 0x83000000 /usr/lib/linux-image-${uname\_r}/${dtb};"\  
 "dtfile 0x83000000 0x87000000 /uEnv.txt ${loadaddr};" \  
 "load ${devtype} ${bootpart} 0x88000000 /kernel/initrd.img-${uname\_r};"\  
 "echo debug: [${bootargs}] ... ;" \  
 "echo debug: [bootz] ... ;" \  
 "bootz 0x80800000 0x88000000:${filesize} 0x83000000;" \  
 "fi;\0" \

若从uboot中直接使用printenv查看boot的内容会显得格式很乱，推荐在uboot源include/configs/mx6ullfire.h 中查看。

第3行，运行loading，loading的内容是将uEnv.txt文件的内容读取到内存中， 如下所示

#其中loadaddr的值为0x8200 0000  
loaduEnv=load ${devtype} ${bootpart} ${loadaddr} /uEnv.txt;

第4行，运行importbootenv，从内存地址中导入环境参数。

"importbootenv=echo Importing environment from ${devtype} ...; " \  
"env import -t ${loadaddr} ${filesize}\0" \

* 第5-11行，判断启动介质类型，设置 **dtb、storage、init** 环境参数。
* 第12-15行，判断是否需要使用利用现有镜像烧录固件到其他介质。
* 第16行，args\_mmc\_old 的作用主要用于设置 **bootargs** 环境参数。如下所示

"args\_mmc\_old=setenv bootargs console=ttymxc0 " \  
"root=/dev/mmcblk${mmcdev}p2 rw " \  
"rootfstype=ext4 " \  
"rootwait ${cmdline} ${flashtype}\0" \

* 第18行，将kernel内核加载到内存地址0x8080 0000处。
* 第20行，将主设备树加载到内存地址 0x8300 0000处。
* 第21行，将设备树插件的内容解析合成到主设备树上，dtfile命令并不是原来uboot就有的， 为了方便用户使用/boot/uEnv.txt文件使用设备树插件而添加的，有兴趣的读者可自行查看相关源码。
* 第22行，将虚拟文件系统加载到0x8800 0000中。
* 第25行，启动linux内核。

## uboot环境参数介绍

uboot中环境参数为我们提供一种不修改uboot源码的情况下， 能够修改kernel启动倒计时、ip地址、以及向内核传递不同的参数等。

在板子上使用 **printenv** 可查看板子上所有的环境参数， 使用 **setenv** 添加/修改/删除环境参数，具体说明如下所示

#设置新的环境参数名为abc，值为100  
=> setenv abc 100  
=> echo $abc  
= 100  
  
#将值修改为200  
=> setenv abc 200  
=> echo $abc  
200  
  
#删除abc环境参数  
=> setenv abc  
=> echo $abc  
  
=>

默认情况下使用setenv命令修改环境参数重启后就会消失， 若想要掉电保存需要执行**saveenv**将环境参数保存到存储介质。

uboot上有一些官方规定的环境变量，这些环境变量在uboot有着特殊的作用， 可通过以下链接查看：[https://www.denx.de/wiki/view/DULG/UBootEnvVariables](https://link.zhihu.com/?target=https%3A//www.denx.de/wiki/view/DULG/UBootEnvVariables)

<https://zhuanlan.zhihu.com/p/656662971>

## 3. uboot 编译与配置

## Linux U-Boot 开发指南

### 1 前言

### 1.1 编写目的

介绍 U-Boot 的编译打包、基本配置、常用命令的使用、基本调试方法等, 为 U-BOOT 的移植及应用开发提供了基础。

### 1.2 适用范围

本文档适用于 brandy2.0, 即 U-Boot-2018 平台。

### 1.3 相关人员

U-Boot 开发/维护人员，内核开发人员。

### 2 LICHEE 类宏关键字解释

请到 longan 目录下的.buildconfig 查看目前使用了以下 LICHEE 类宏。

LICHEE\_IC ——> IC名\  
LICHEE\_CHIP ——> 平台名\  
LICHEE\_BOARD ——> 板级名\  
LICHEE\_ARCH ——> 所属架构\  
LICHEE\_BOARD\_CONFIG\_DIR ——> 板级目录\  
LICHEE\_BRANDY\_OUT\_DIR ——> bin文件所在目录\  
LICHEE\_PLAT\_OUT ——> 平台临时bin所在目录\  
LICHEE\_CHIP\_CONFIG\_DIR ——> IC目录

### 3 编译方法介绍

### 3.1 准备编译工具链

准备编译工具链接执行步骤如下：

1）cd longan/brandy/brandy-2.0/\  
2）./build.sh -t

### 3.2 快速编译 boot0 及 U-Boot

在longan/brandy/brandy-2.0/目录下，执行 ./build.sh -p 平台名称，可以快速完成整个 boot 编译动作。这个平台名称是指，LICHEE\_CHIP。

./build.sh -p {LICHEE\_CHIP} //快速编译spl/U-Boot  
./build.sh -o spl-pub -p {LICHEE\_CHIP} //快速编译spl-pub  
./build.sh -o uboot -p {LICHEE\_CHIP} //快速编译U-Boot

### 3.3 编译 U-Boot

cd longan/brandy/brandy-2.0/u-boot-2018/进入 u-boot-2018 目录。以{LICHEE\_CHIP}为例，依次执行如下操作即可。

1）make {LICHEE\_CHIP}\_defconfig  
2）make -j

### 3.4 编译 boot0/fes/sboot

cd longan/brandy/brandy-2.0/spl-pub进入spl-pub目录，需设置平台和要编译的模块参数。以{LICHEE\_CHIP}为例，编译 nand/emmc 的方法如下：

1. 编译boot0

make distclean  
make p={LICHEE\_CHIP} m=nand  
make boot0  
  
make distclean  
make p={LICHEE\_CHIP} m=emmc  
make boot0

1. 编译fes

make distclean  
make p={LICHEE\_CHIP} m=fes  
make fes

1. 编译sboot

make distclean  
make p={LICHEE\_CHIP} m=sboot  
make sboot

### 4 U-Boot 功能及其配置方法/文件介绍

### 4.1 U-Boot 功能介绍

在嵌入式操作系统中，BootLoader/U-Boot 是在操作系统内核运行之前运行。可以初始化硬件设备、建立内存空间映射图，从而将系统的软硬件环境带到一个合适状态，以便为最终调用操作系统内核准备好正确的环境。在 sunxi 平台中，除了必须的引导系统启动功能外，BOOT 系统还提供烧写、升级等其它功能。

U-Boot 主要功能可以分为以下几类

1. 引导内核

能从存储介质（nand/mmc/spinor）上加载内核镜像到 DRAM 指定位置并运行。

1. 量产 & 升级

包括卡量产，USB 量产，私有数据烧录，固件升级

1. 开机提示信息

开机能显示启动 logo 图片（BMP 格式)

1. Fastboot 功能

实现 fastboot 的标准命令，能使用 fastboot 刷机

### 4.2 U-Boot 功能配置方法介绍

U-Boot 中的各项功能可以通过 defconfig 或配置菜单 menuconfig 进行开启或关闭, 具体配置

方法如下:

### 4.2.1 通过 defconfig 方式配置

1. vim /longan/brandy/brandy-2.0/u-boot-2018/configs/{LICHEE\_CHIP}\_defconfig
2. 开{LICHEE\_CHIP}*defconfig或{LICHEE\_CHIP}*nor\_defconfig后，在相应的宏定义前去掉或添加"#"即可将相应功能开启或关闭。如下图，只要将CONFIG\_SUNXI\_NAND前的#去掉即可支持 NAND 相关功能，其他宏定义的开启关闭也类似。修改后需要运行make xxx\_defconfig使修改后的配置生效。

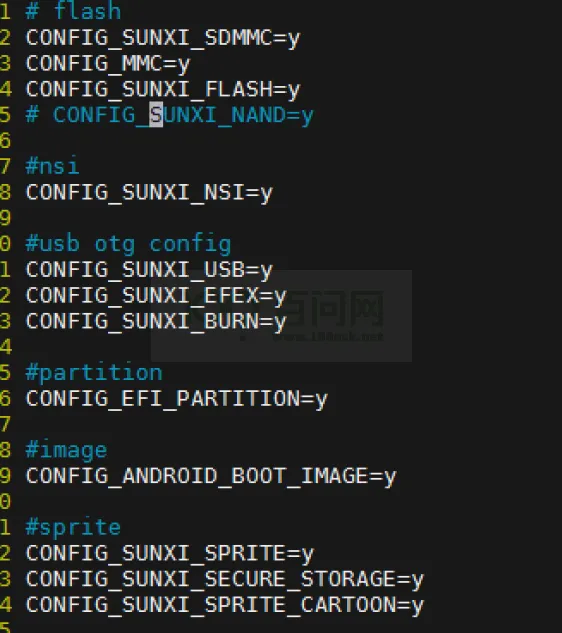


图 4-1: defconfig 配置图

### 4.2.2 通过 menuconfig 方式配置

通过 menuconfig 方式配置的方法步骤如下：

1. cd brandy/brandy-2.0/u-boot-2018/
2. 执行make menuconfig命令，会弹出 menuconfig 配置菜单窗口，如下图所示。此时即可对各模块功能进行配置，配置方法 menuconfig 配置菜单窗口中有说明。
3. 修改后配置已经生效，直接 make 即可生成对应 bin。如果重新运行make xxx\_defconfig，通过menuconfig 方式修改的配置会在运行make xxx\_defconfig后被xxx\_defconfig中的配置覆盖。

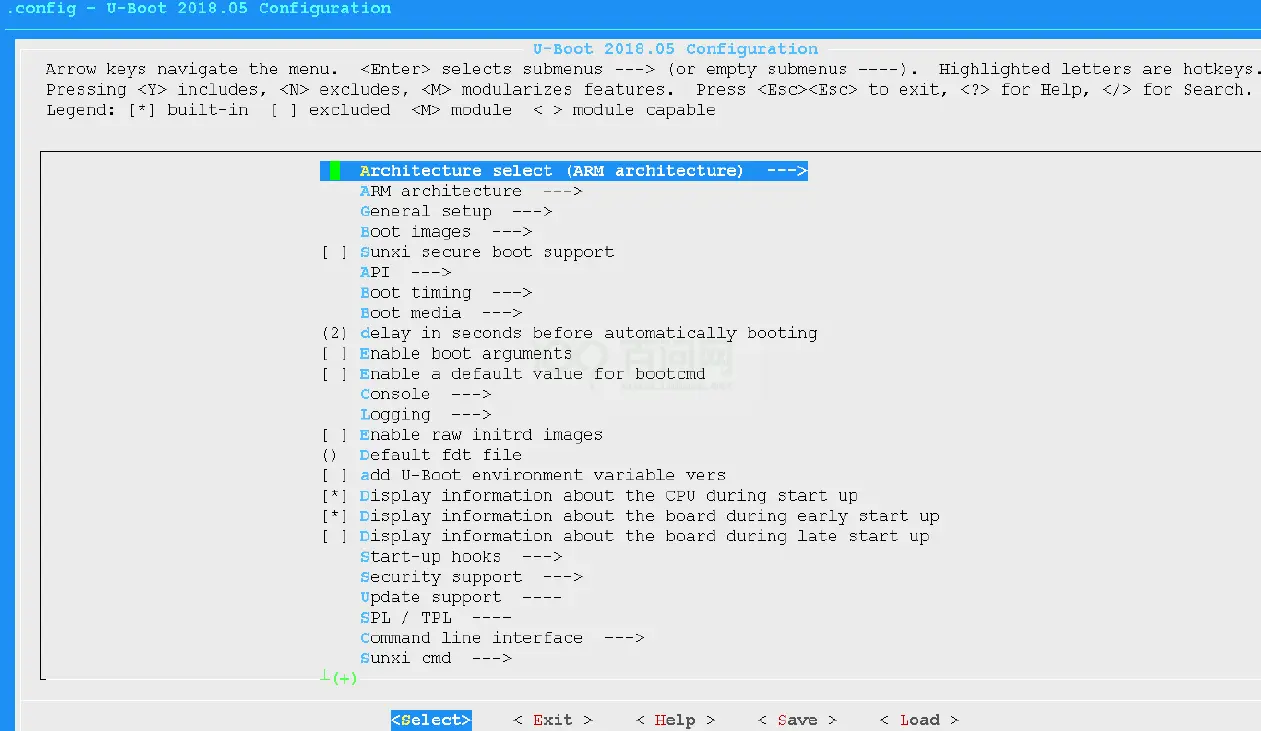


图 4-2: menuconfig 配置菜单图

### 4.3 U-Boot 配置参数文件介绍

U-Boot 自 linux-5.4 以后不再使用 sysconfig 和内核 dts 作为配置文件，而是使用 U-Boot 自带的 dts 来配置参数。kernel-dts 与 U-Boot-dts 完全独立。

### 4.3.1 U-Boot-dts 路径

U-Boot-dts 路径为：vim longan/brandy/brandy-2.0/u-boot-2018/arch/arm/dts

### 4.3.2 U-Boot-dts，defconfig 配置

|  |  |
| --- | --- |
| 配置项 | 配置项含义 |
| CONFIG\_OF\_SEPARATE | 构建 U-Boot 设备树成为 U-Boot 的一部分 |
| CONFIG\_OF\_BOARD | 关闭使用外部 dts |
| CONFIG\_DEFAULT\_DEVICE\_TREE | 选择构建的 dts 文件文件名 |
| CONFIG\_SUNXI\_NECESSARY\_REPLACE\_FDT | 开启选项, 实现内部 dts 换成外部 dts |
| 配置项 | 选项 |
| CONFIG\_OF\_SEPARATE | y |
| CONFIG\_OF\_BOARD | n |
| CONFIG\_DEFAULT\_DEVICE\_TREE | “{LICHEE\_CHIP}-soc-system” |
| CONFIG\_SUNXI\_NECESSARY\_REPLACE\_FDT | y |

### 4.3.3 U-Boot-dts 注意事项

### 4.3.3.1 编译注意事项

1.dts 分为板级 dts，和系统 dts。

系统 dts 由 CONFIG\_DEFAULT\_DEVICE\_TREE 决定，可以在 $(CONFIG\_SYS\_CONFIG\_NAME)\_defconfig找到该宏的定义。

系统 dts 最终会 include 板级 dts，文件路径 {LICHEE\_BOARD\_CONFIG\_DIR}，文件名:uboot-board.dts。

1. 我们可以通过编译时的打印判断启动的 dts

OBJCOPY examples/standalone/hello\_world.srec  
OBJCOPY examples/standalone/hello\_world.bin  
LD u-boot  
OBJCOPY u-boot.srec  
OBJCOPY u-boot-nodtb.bin  
‘{LICHEE\_BOARD\_CONFIG\_DIR}/uboot-board.dts’ -> ‘~/longan/brandy/brandy-2.0/u-boot-2018/  
arch/{LICHEE\_ARCH}/dts/.board-uboot.dts’  
DTC arch/{LICHEE\_ARCH}/dts/{LICHEE\_CHIP}-soc-system.dtb  
SYM u-boot.sym  
SHIPPED dts/dt.dtb  
FDTGREP dts/dt-spl.dtb  
COPY u-boot.dtb  
CAT u-boot-dtb.bin  
COPY u-boot.bin  
‘u-boot.bin’ -> ‘{LICHEE\_CHIP}.bin’ ‘u-boot-g{LICHEE\_CHIP}.bin’ -> ‘{LICHEE\_BRANDY\_OUT\_DIR}/bin/u-boot-g{LICHEE\_CHIP}.bin’ ‘u-boot-g{LICHEE\_CHIP}.bin’ -> ‘{LICHEE\_PLAT\_OUT}/u-boot-g{LICHEE\_CHIP}.bin’  
CFGCHK u-boot.cfg

### 4.3.3.2 语法注意事项

当系统 dts 与板级 dts 存在同路径下同名节点时，板级 dts 将会覆盖系统 dts。

### 4.3.3.3 运行时注意事项

1. 为了在启动内核前更新参数到内核 dts 和可以在 U-Boot 控制台查看修改 dts。按阶段划分可以分为使用内部 dts 阶段和使用内核 dts 阶段，如下图所示。

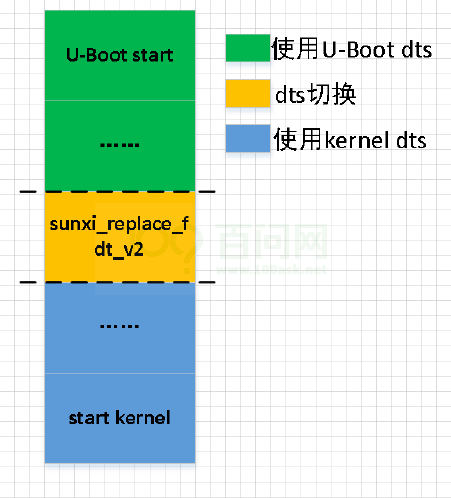


图 4-3: dts 变化图

1. 可以通过命令set\_working\_fdt来切换当前生效的 fdt。

[04.562]update bootcmd  
[04.576]change working\_fdt 0x7bebee58 to 0x7be8ee58  
[04.587]update dts  
Hit any key to stop autoboot: 0  
=> set  
 set\_working\_fdt setenv setexpr  
=> set\_working\_fdt 0x7bebee58  
change working\_fdt 0x7be8ee58 to 0x7bebee58  
=>

### 5 U-Boot 常用命令介绍

### 5.1 env 命令说明

通过env命令可以对{LICHEE\_CHIP\_CONFIG\_DIR}/configs/default/env.cfg中的环境变量进行查看及更改。在小机启动过程中按任意键进入 U-Boot shell 命令状态，输入命令"env"即可查看命令帮助信息。

具体示例如下：

1. 输入命令"env print"，可查看当前所有的环境变量信息，如下：

=> pri  
ab\_partition\_list=bootloader,env,boot,vendor\_boot,dtbo,vbmeta,vbmeta\_system,vbmeta\_vendor  
android\_trust\_chain=true  
boot\_fastboot=fastboot  
boot\_normal=sunxi\_flash read 45000000 boot;bootm 45000000  
boot\_recovery=sunxi\_flash read 45000000 recovery;bootm 45000000  
bootcmd=run setargs\_mmc boot\_normal  
bootdelay=0  
bootreason=charger  
bt\_mac=20:A1:11:12:13:44  
cma=8M  
console=ttyAS0,115200  
earlyprintk=sunxi-uart,0x05000000  
fdtcontroladdr=7bed0e60  
fileaddr=40000000  
filesize=15cf6  
force\_normal\_boot=1  
init=/init  
initcall\_debug=0  
keybox\_list=widevine,ec\_key,ec\_cert1,ec\_cert2,ec\_cert3,rsa\_key,rsa\_cert1,rsa\_cert2,rsa\_cert3  
loglevel=8  
mac=10:14:15:15:9A:CA  
mmc\_root=/dev/mmcblk0p4  
nand\_root=/dev/nand0p4  
partitions=bootloader\_a@mmcblk0p1:bootloader\_b@mmcblk0p2:env\_a@mmcblk0p3:env\_b@mmcblk0p4:  
boot\_a@mmcblk0p5:boot\_b@mmcblk0p6:vendor\_boot\_a@mmcblk0p7:vendor\_boot\_b@mmcblk0p8:  
super@mmcblk0p9:misc@mmcblk0p10:vbmeta\_a@mmcblk0p11:vbmeta\_b@mmcblk0p12:  
vbmeta\_system\_a@mmcblk0p13:vbmeta\_system\_b@mmcblk0p14:vbmeta\_vendor\_a@mmcblk0p15:  
vbmeta\_vendor\_b@mmcblk0p16:frp@mmcblk0p17:empty@mmcblk0p18:metadata@mmcblk0p19:  
private@mmcblk0p20:dtbo\_a@mmcblk0p21:dtbo\_b@mmcblk0p22:media\_data@mmcblk0p23:  
UDISK@mmcblk0p24  
rotpk\_status=0  
setargs\_mmc=setenv bootargs earlyprintk=${earlyprintk} clk\_ignore\_unused initcall\_debug=${  
initcall\_debug} console=${console} loglevel=${loglevel} root=${mmc\_root} init=${init}  
cma=${cma} snum=${snum} mac\_addr=${mac} wifi\_mac=${wifi\_mac} bt\_mac=${bt\_mac}  
specialstr=${specialstr} gpt=1 androidboot.force\_normal\_boot=${force\_normal\_boot}  
androidboot.slot\_suffix=${slot\_suffix}  
setargs\_nand=setenv bootargs earlyprintk=${earlyprintk} clk\_ignore\_unused initcall\_debug=${  
initcall\_debug} console=${console} loglevel=${loglevel} root=${nand\_root} init=${init}  
cma=${cma} snum=${snum} mac\_addr=${mac} wifi\_mac=${wifi\_mac} bt\_mac=${bt\_mac}  
specialstr=${specialstr} gpt=1 androidboot.force\_normal\_boot=${force\_normal\_boot}  
androidboot.slot\_suffix=${slot\_suffix}  
slot\_suffix=\_a  
snum=A100B3N041  
wifi\_mac=10:A1:11:12:13:44  
Environment size: 2078/131068 bytes  
=>

1. 输入命令"env set bootdelay 3"，可更改环境变量bootdelay（即 boot 启动时 log 中的倒计时延迟时间）值的大小。
2. 输入命令"env save"，即可将上述更改进行保存，保存后重新上电，或输入命令"reset"，即可看到上述更改bootdelay的延时时间被更改生效。
3. 其他env命令请查看env帮助信息。

### 5.2 sunxi\_flash read 命令说明

### 5.2.1 使用方法

用以下命令将 flash 指定地址中数据读到 DRAM 的指定地址处：

sunxi\_flash read dram\_addr flash\_addr

### 5.2.2 使用示例

sunxi\_flash read 0x45000000 env—将env分区数据读到DRAM的0x45000000地址处  
sunxi\_flash read 45000000 boot;bootm 45000000—将flash中boot分区数据读到DRAM的0x45000000地 址,并 从0x45000000处启动。

### 5.3 fastboot 命令说明

fastboot 是 Android 平台上一个通用的刷机工具，也是一个很好的开发调试工具，以下介绍 fastboot 的基本使用方法。

### 5.3.1 使用前提

fastboot PC 端工具可以从 Google Android SDK(Android-sdk-windows/tools) 中获得，也可以在 Android 源代码编译过后的生成文件获得 (out/host/linux-x86/bin)。

在 Linux 系统中，使用 fastboot 不需要安装驱动。但在 Windows 系统中，使用 fastboot 前需安装 fastboot 相关驱动。adb 的驱动在 fastboot 模式下也可以安装成功，但是无法使用，请使用我们提供的驱动，并手动安装。

### 5.3.2 使用步骤

1. 小机上电启动，按任意键进入 U-Boot 命令状态；
2. 串口端输入"fastboot"命令；
3. 打开 PC 端 fastboot 工具，并输入"fastboot devices"命令，看是否有 fastboot 设备显示；
4. 在正确获取 fastboot 设备的前提下，输入命令"fastboot flash env /path/to/env.fex"，将env.fex写到env分区（/path/to/目录下的env.fex中bootdelay值应该与 flash 中原有env中bootdelay值不同，这样可根据bootdelay值不同来确定 fastboot 烧写是否成功）, 同下载env.fex分区一样, 输入命令“fastboot flash boot /path/to/boot.img”将内核下载到内存中；
5. 输入"fastboot reboot"命令重启，查看启动倒计时即bootdelay的值是否改变；

### 5.3.3 fastboot 基本命令使用示例

1. fastboot 几个基本命令示例如下:

fastboot devices ：显示 fastboot 的设备。

fastboot erase ：擦除分区，例如fastboot erase boot，擦除boot分区。

fastboot flash：旧分区（待写分区），例如fastboot flash boot/path/to/boot.img，将boot.img写到boot分区。

1. 注意事项:

fastboot 中使用的分区和sys\_partition.fex中分区一致，具体的分区信息可以从小机上电启动进入 U-Boot shell 命令状态，输入命令"part list sunxi\_flash 0"中获取，分区信息如下：

=> part list sunxi\_flash 0  
  
Partition Map for UNKNOWN device 0 -- Partition Type: EFI  
  
Part Start LBA End LBA Name  
 Attributes  
 Type GUID  
 Partition GUID  
1 0x00008000 0x00017fff "bootloader"  
 attrs: 0x8000000000000000  
 type: ebd0a0a2-b9e5-4433-87c0-68b6b72699c7  
 guid: a0085546-4166-744a-a353-fca9272b8e45  
2 0x00018000 0x0001ffff "env"  
 attrs: 0x8000000000000000  
 type: ebd0a0a2-b9e5-4433-87c0-68b6b72699c7  
 guid: a0085546-4166-744a-a353-fca9272b8e46  
3 0x00020000 0x0002ffff "boot"  
 attrs: 0x8000000000000000  
 type: ebd0a0a2-b9e5-4433-87c0-68b6b72699c7  
 guid: a0085546-4166-744a-a353-fca9272b8e47  
4 0x00030000 0x0032ffff "super"  
 attrs: 0x8000000000000000  
 type: ebd0a0a2-b9e5-4433-87c0-68b6b72699c7  
 guid: a0085546-4166-744a-a353-fca9272b8e48  
5 0x00330000 0x00337fff "misc"  
 attrs: 0x8000000000000000  
 type: ebd0a0a2-b9e5-4433-87c0-68b6b72699c7  
 guid: a0085546-4166-744a-a353-fca9272b8e49  
6 0x00338000 0x00347fff "recovery"  
 attrs: 0x8000000000000000  
 type: ebd0a0a2-b9e5-4433-87c0-68b6b72699c7  
 guid: a0085546-4166-744a-a353-fca9272b8e4a

### 5.4 fat 命令说明

fat命令可以对 FAT 文件系统的相关存储设备进行查询及文件读写操作，在打包固件的时候, 我们会制作启动资源分区镜像, 把指定的目录下的文件按照文件系统的格式排布，文件中包括了原来目录中的所有文件，并完全按照目录结构排列。当把这个镜像文件烧写到存储设备上的某一个分区的时候，可以看到这个分区和原有目录的内容一样。使用fat可以方便地以文件和目录的方式对小机 flash 进行数据访问，如显示 logo。这些指令基本上要和 U 盘或者 SD 卡同时使用，主要用于读取这些移动存储器上的 FAT 分区。其相关操作命令如下：

1. fatls : 列出相应设备目录上的所有文件，示例如下图：

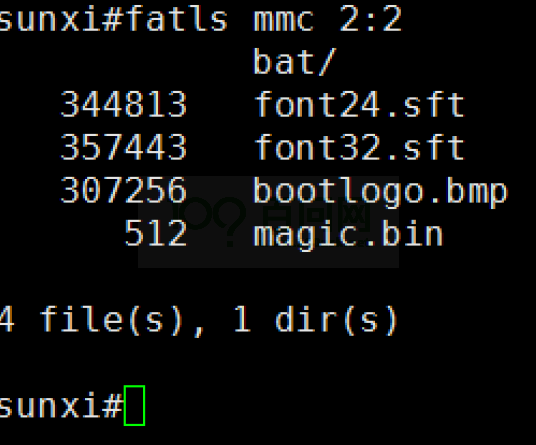


图 5-1: fatls 命令执行示例图

说明

**补充说明，fatls mmc 2:2 中的第一个 2 表示的是 emmc 设备，2 表示其分区号，其说明如下图：**



图 5-2: fatls 命令参数说明图

1. fatinfo: 打印出相应设备目录的文件系统信息，示例如下图：

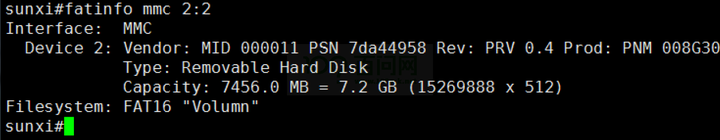


图 5-3: fatinfo 命令执行示例图

1. fatload: 从 FAT 文件系统中读取二进制文件到 RAM 存储中，示例如下：

sunxi#usb start  
(Re)start USB...  
USB0: start sunxi ehci1...  
config usb pin success  
config usb clk ok  
sunxi ehci1 init ok...  
USB EHCI 1.00  
scanning bus 0 for devices... 3 USB Device(s) found  
scanning usb for storage devices... 1 Storage Device(s) found  
sunxi#fatls usb 0:1 /  
16024600 sandisksecureaccessv3\_win.exe  
sandisk secureaccess/  
lost.dir/  
Android/  
test/  
video test/  
amapauto/  
0 vid\_20161017\_160818.ts  
phoenixsuit/  
system volume information/  
0 vid\_20161017\_160919.ts  
video/  
156672 wifi pro\_com su.exe  
495 sys.ini  
1035 pr\_80211g\_all.ini  
config/  
158208 wifi pro\_new.exe  
158208 wifi pro.exe  
0 vid\_20161017\_164822.ts  
0 vid\_20161017\_164906.ts  
sunxi-tvd/  
71149 sys\_config.fex  
vga/  
397836884 system.img  
14180352 boot.img  
13 file(s), 13 dir(s)  
sunxi#fatload usb 0:1 0x42000000 boot.img  
reading boot.img  
14180352 bytes read in 1149 ms (11.8 MiB/s)  
sunxi#mmc dev 2  
mmc2(part 0) is current device  
sunxi#mmc write 0x42000000 0x15000 5000  
MMC write: dev # 2, block # 86016, count 20480 ... 20480 blocks written: OK

说明：以上操作即将 U 盘的boot.img写到对应的 mmc 分区地址处。

1. fatwrite: 从内存中将对应的文件写到设备文件系统中。

### 5.5 md 命令说明

md命令可以对指定内存的数据进行查看，方便了解内存的数据情况及调试工作。其使用方法如下：

md 0xF0000000： 即用md命令查看内存DRAM 0xF0000000处内容

### 5.6 FDT 命令说明

FDT：flattened device tree 的缩写在 U-Boot 控制台停下后，输入fdt，可以查看fdt命令帮助。

sunxi#fdt  
fdt - flattened device tree utility commands  
Usage:  
fdt addr [-c] <addr> [<length>] - Set the [control] fdt location to <addr>  
fdt move <fdt> <newaddr> <length> - Copy the fdt to <addr> and make it active  
fdt resize - Resize fdt to size + padding to 4k addr  
fdt print <path> [<prop>] - Recursive print starting at <path>  
fdt list <path> [<prop>] - Print one level starting at <path>  
fdt get value <var> <path> <prop> - Get <property> and store in <var>  
fdt get name <var> <path> <index> - Get name of node <index> and store in <var>  
fdt get addr <var> <path> <prop> - Get start address of <property> and store in <var>  
fdt get size <var> <path> [<prop>] - Get size of [<property>] or num nodes and store in <var>  
fdt set <path> <prop> [<val>] - Set <property> [to <val>]  
fdt mknode <path> <node> - Create a new node after <path>  
fdt rm <path> [<prop>] - Delete the node or <property>  
fdt header  
fdt bootcpu <id> - Set boot cpuid  
fdt memory <addr> <size> - Add/Update memory node  
fdt rsvmem print - Show current mem reserves  
fdt rsvmem add <addr> <size> - Add a mem reserve  
fdt rsvmem delete <index> - Delete a mem reserves  
fdt chosen [<start> <end>] - Add/update the /chosen branch in the tree  
<start>/<end> - initrd start/end addr  
NOTE: Dereference aliases by omiting the leading '/', e.g. fdt print ethernet0。  
sunxi#

说明

**其中常用的命令就是fdt list 和 fdt set,fdt list 用来查询节点配置,fdt set 用来修改节点配置。**

### 5.6.1 查询配置

首先确定要查询的字段在 device tree 的路径，如果不知道路径，则需要用fdt命令按以下步骤进

行查询。1. 在根目录下查找。

sunxi#fdt list /  
/ {  
 model = "{LICHEE\_CHIP}";  
 compatible = "arm,{LICHEE\_CHIP}", "arm,{LICHEE\_CHIP}";  
 interrupt-parent = <0x00000001>;  
 #address-cells = <0x00000002>;  
 #size-cells = <0x00000002>;  
 ......................  
 cpuscfg {  
 };  
 ion {  
 };  
 dram {  
 };  
 memory@40000000 {  
 };  
 interrupt-controller@1c81000 {  
 };  
 sunxi-chipid@1c14200 {  
 };  
 timer {  
 };  
 pmu {  
 };  
 dvfs\_table {  
 };  
 dramfreq {  
 };  
 gpu@0x01c40000 {  
 };  
 wlan {  
 };  
 bt {  
 };  
 btlpm {  
 };  
};

如果找到需要的配置，比如wlan的配置，运行如下命令即可。

sunxi#fdt list /wlan //注意路径中的 /  
wlan {  
 compatible = "allwinner,sunxi-wlan";  
 clocks = <0x00000096>;  
 wlan\_power = "vcc-wifi";  
 wlan\_io\_regulator = "vcc-wifi-io";  
 wlan\_busnum = <0x00000001>;  
 status = "okay";  
 device\_type = "wlan";  
 wlan\_regon = <0x00000077 0x0000000b 0x00000002 0x00000001 0xffffffff 0xffffffff 0  
 x00000000>;  
 wlan\_hostwake = <0x00000077 0x0000000b 0x00000003 0x00000006 0xffffffff 0xffffffff  
 0x00000000>;  
};

1. 在 soc目录下找。如果在第一步中没有发现要找的配置，比如nand0的配置，则该配置可能在soc目录下。

sunxi#fdt list /soc  
soc@01c00000 {  
 compatible = "simple-bus";  
 #address-cells = <0x00000002>;  
 #size-cells = <0x00000002>;  
 ranges;  
 device\_type = "soc";  
 ......................  
 hdmi@01ee0000 {  
 };  
 tr@01000000 {  
 };  
 pwm@01c21400 {  
 };  
 nand0@01c03000 {  
 };  
 thermal\_sensor {  
 };  
 cpu\_budget\_cool {  
 };  
 .......................  
};

然后用如下命令显示即可:

sunxi#fdt list /soc/nand0  
nand0@01c03000 {  
 compatible = "allwinner,sun50i-nand";  
 device\_type = "nand0";  
 reg = <0x00000000 0x01c03000 0x00000000 0x00001000>;  
 interrupts = <0x00000000 0x00000046 0x00000004>;  
 clocks = <0x00000004 0x0000007e>;  
 pinctrl-names = "default", "sleep";  
 pinctrl-1 = <0x00000081>;  
 nand0\_regulator1 = "vcc-nand";  
 nand0\_regulator2 = "none";  
 nand0\_cache\_level = <0x55aaaa55>;  
 nand0\_flush\_cache\_num = <0x55aaaa55>;  
 nand0\_capacity\_level = <0x55aaaa55>;  
 nand0\_id\_number\_ctl = <0x55aaaa55>;  
 nand0\_print\_level = <0x55aaaa55>;  
 nand0\_p0 = <0x55aaaa55>;  
 nand0\_p1 = <0x55aaaa55>;  
 nand0\_p2 = <0x55aaaa55>;  
 nand0\_p3 = <0x55aaaa55>;  
 status = "disabled";  
 nand0\_support\_2ch = <0x00000000>;  
 pinctrl-0 = <0x000000a9 0x000000aa>;  
};

1. 使用路径别名查找。别名是 device tree 中完整路径的一个简写，有一个专门的节点 ( /aliases) 来表示别名的相关信息，用如下命令可以查看系统中别名的配置情况：

sunxi#fdt list /aliases  
aliases {  
 serial0 = "/soc@01c00000/uart@01c28000";  
 ..............  
 mmc0 = "/soc@01c00000/sdmmc@01c0f000";  
 mmc2 = "/soc@01c00000/sdmmc@01C11000";  
 nand0 = "/soc@01c00000/nand0@01c03000";  
 disp = "/soc@01c00000/disp@01000000";  
 lcd0 = "/soc@01c00000/lcd0@01c0c000";  
 hdmi = "/soc@01c00000/hdmi@01ee0000";  
 pwm = "/soc@01c00000/pwm@01c21400";  
 boot\_disp = "/soc@01c00000/boot\_disp";  
};  
sunxi#

由于配置了nand0节点的路径别名，因此可以用如下命令来显示nand0的配置信息。

sunxi#fdt list nand0  
nand0@01c03000 {  
 compatible = "allwinner,sun50i-nand";  
 device\_type = "nand0";  
 reg = <0x00000000 0x01c03000 0x00000000 0x00001000>;  
 ..................  
 pinctrl-names = "default", "sleep";  
 pinctrl-1 = <0x00000081>;  
};

注：在fdt的所有命令中，alias 可以用作path参数。

fdt list <path> [<prop>] - Print one level starting at <path>  
fdt set <path> <prop> [<val>] - Set <property> [to <val>]

### 5.6.2 修改配置

### 5.6.2.1 修改整数配置

命令格式：fdt set path prop 示例：fdt set /wlan wlan\_busnum <0x2>

sunxi#fdt list /wlan  
wlan {  
 compatible = "allwinner,sunxi-wlan";  
 clocks = <0x00000096>;  
 wlan\_power = "vcc-wifi";  
 wlan\_io\_regulator = "vcc-wifi-io";  
 wlan\_busnum = <0x00000001>;  
 status = "disable";  
 device\_type = "wlan";  
};  
sunxi#fdt set /wlan wlan\_busnum <0x2>  
sunxi#fdt list /wlan  
wlan {  
 compatible = "allwinner,sunxi-wlan";  
 clocks = <0x00000096>;  
 wlan\_power = "vcc-wifi";  
 wlan\_io\_regulator = "vcc-wifi-io";  
 wlan\_busnum = <0x00000002>; //修改后  
 status = "disable";  
 device\_type = "wlan";  
};

注：修改整数时，根据需要也可配置为数组形式，需要用空格来分隔。命令格式：fdt set path prop <0x1 0x2 0x3>

### 5.6.2.2 修改字符串配置

命令格式：fdt set path prop "xxxxx" 示例：fdt set /wlan status "disable"

sunxi#fdt list /wlan  
wlan {  
 compatible = "allwinner,sunxi-wlan";  
 clocks = <0x00000096>;  
 wlan\_power = "vcc-wifi";  
 wlan\_io\_regulator = "vcc-wifi-io";  
 wlan\_busnum = <0x00000001>;  
 status = "okay";  
 device\_type = "wlan";  
};  
sunxi#fdt set /wlan status "disable"  
sunxi#fdt list /wlan  
wlan {  
 compatible = "allwinner,sunxi-wlan";  
 clocks = <0x00000096>;  
 wlan\_power = "vcc-wifi";  
 wlan\_io\_regulator = "vcc-wifi-io";  
 wlan\_busnum = <0x00000001>;  
 status = "disable"; //修改后  
 device\_type = "wlan";  
};  
sunxi#

注：修改字符串时，根据需要也可配置为数组形式，需要用空格来分隔。命令格式：fdt set path prop "string1" "string2"

### 5.6.3 GPIO 或者 PIN 配置特殊说明

接口对应的数字编号说明如下：

#define PA 0  
#define PB 1  
#define PC 2  
#define PD 3  
#define PE 4  
#define PF 5  
#define PG 6  
#define PH 7  
#define PI 8  
#define PJ 9  
#define PK 10  
#define PL 11  
#define PM 12  
#define PN 13  
#define PO 14  
#define PP 15  
#define default 0xffffffff

Sysconfig 中描述 gpio 的形式如下：Port:端口+组内序号<功能分配><内部电阻状态><驱动能力><输出电平状态>

### 5.6.3.1 Pin 配置说明

Pinctrl 节点分为 cpux 和 cpus，对应的节点路径如下：Cpux : /soc/pinctrl@01c20800 Cpus:/soc/pinctrl@01f02c00

### 5.6.3.2 查看 PIN 配置

PIN 配置属性字段说明:

|  |  |
| --- | --- |
| 属性字段 | 含义 |
| allwinner,function | 对应于 sysconfig 中的主键名 |
| allwinner,pins | 对应于 sysconfig 中每个 gpio 配置中的端口名 |
| allwinner,pname | 对应于 sysconfig 中主键下面子键名字 |
| allwinner,muxsel | 功能分配 |
| allwinner,pull | 内部电阻状态 |
| allwinner,drive | 驱动能力 |
| allwinner,data | 输出电平状态 |

说明

**其中0xffffffff表示使用默认值。**

按以下方法查看cpux的 PIN 配置。

sunxi#fdt list /soc/pinctrl@01c20800/lcd0  
lcd0@0 {  
 linux,phandle = <0x000000ab>;  
 phandle = <0x000000ab>;  
 allwinner,pins = "PD12", "PD13", "PD14", "PD15", "PD16", "PD17", "PD18", "PD19", "PD20", "PD21";  
 allwinner,function = "lcd0";  
 allwinner,pname = "lcdd0", "lcdd1", "lcdd2", "lcdd3", "lcdd4", "lcdd5", "lcdd6", "lcdd7", "lcdd8", "lcdd9";  
 allwinner,muxsel = <0x00000003>;  
 allwinner,pull = <0x00000000>;  
 allwinner,drive = <0xffffffff>;  
 allwinner,data = <0xffffffff>;  
};  
sunxi#

按以下方法查看cpus的 PIN 配置。

sunxi# fdt list /soc/pinctrl@01f02c00/s\_uart0  
s\_uart0@0 {  
 linux,phandle = <0x000000b4>;  
 phandle = <0x000000b4>;  
 allwinner,pins = "PL2", "PL3";  
 allwinner,function = "s\_uart0";  
 allwinner,pname = "s\_uart0\_tx", "s\_uart0\_rx";  
 allwinner,muxsel = <0x00000002>;  
 allwinner,pull = <0xffffffff>;  
 allwinner,drive = <0xffffffff>;  
 allwinner,data = <0xffffffff>;  
};  
sunxi#

### 5.6.3.3 修改 PIN 配置

使用fdt set命令可以修改 PIN 中相关属性字段

sunxi#fdt set /soc/pinctrl@01c20800/lcd0 allwinner,drive <0x1>  
sunxi#fdt list /soc/pinctrl@01c20800/lcd0  
lcd0@0 {  
 linux,phandle = <0x000000ab>;  
 phandle = <0x000000ab>;  
 allwinner,pins = "PD12", "PD13", "PD14", "PD15", "PD16", "PD17", "PD18", "PD19", "PD20", "PD21";  
 allwinner,function = "lcd0";  
 allwinner,pname = "lcdd0", "lcdd1", "lcdd2", "lcdd3", "lcdd4", "lcdd5", "lcdd6", "lcdd7", "lcdd8", "lcdd9";  
 allwinner,muxsel = <0x00000003>;  
 allwinner,pull = <0x00000000>;  
 allwinner,drive = <0x00000001>;  
 allwinner,data = <0xffffffff>;  
};

说明

**示例中该处修改会影响allwinner,pins表示的所有端口的驱动能力配置，修改allwinner,muxsel, allwinner,pull,allwinner,data的值也会产生类似效果。**

### 5.6.3.4 GPIO 配置说明

Device tree 中 GPIO 对应关系，以 usb 中usb\_id\_gpio为例

sunxi#fdt list /soc/usbc0  
usbc0@0 {  
 test = <0x00000002 0x00000003 0x12345678>;  
 device\_type = "usbc0";  
 compatible = "allwinner,sun50i-otg-manager";  
 ........  
 usb\_serial\_unique = <0x00000000>;  
 usb\_serial\_number = "20080411";  
 rndis\_wceis = <0x00000001>;  
 status = "okay";  
 usb\_id\_gpio = <0x00000030 0x00000007 0x00000009 0x00000000 0x00000001 0xffffffff 0xffffffff>;  
};

对应于 device tree 中 usb\_id\_gpio = <0x00000030 0x00000007 0x00000009 0x00000000 0x00000001 0xffffffff 0xffffffff>，解释如下：

|  |  |
| --- | --- |
| 属性数值 | 含义 |
| 0x00000030 | device tree 内部一个节点相关信息，这里可以略过 |
| 0x00000007 | 端口 PH, 即 #define PH 7 |
| 0x00000009 | 组内序号, 即 PH09 |
| 0x00000000 | 功能分配, 即将 PH09 配为输入 |
| 0x00000001 | 内部电阻状态, 即配为上拉 |
| 0xffffffff | 驱动能力, 默认值 |
| 0xffffffff | 输出电平, 默认值 |

如果需要修改 usb\_id\_gpio的配置，可按如下方式（示例修改了驱动能力，输出电平两项）：

sunxi#fdt set /soc/usbc0 usb\_id\_gpio <0x00000030 0x00000007 0x00000009 0x00000000 0x00000001 0x2 0x1>  
sunxi#fdt list  
usbc0@0 {  
 test = <0x00000002 0x00000003 0x12345678>;  
 device\_type = "usbc0";  
 compatible = "allwinner,sun50i-otg-manager";  
 ........  
 usb\_serial\_unique = <0x00000000>;  
 usb\_serial\_number = "20080411";  
 rndis\_wceis = <0x00000001>;  
 status = "okay";  
 usb\_id\_gpio = <0x00000030 0x00000007 0x00000009 0x00000000 0x00000001 0x00000002 0x00000001>; //修改ok  
};  
sunxi#

5.7 其他命令说明（boot, reset, efex）

1. boot : 启动内核
2. reset: 复位重启系统
3. efex: 进入烧录状态

说明

**注：其他更多 U-Boot 命令介绍，请进入 U-Boot shell 命令状态后输入"help"进行了解。**

### 6 基本调试方法介绍

debug 调试信息介绍如下：

1. debug\_mode

debug\_mode 可以控制 Boot0 的打印等级，打开文件{LICHEE\_BOARD\_CONFIG\_DIR}/sys\_config.fex，在主键 [platform] 下添加子键"debug\_mode = 8"即表示开启所有打印，debug\_mode=0 表示关闭启动时 Boot0 的打印 log,未显式配置 debug\_mode 时，按 debug\_mode=8 处理。目前常用的打印等级有 0（关闭所有打印）、1（只显示关键节点打印）、4（打印错误信息）、8（打印所有 log 信息）。

debug\_mode 可以控制 U-Boot 的打印等级，打开文件{LICHEE\_BOARD\_CONFIG\_DIR}/b3/uboot-board.dts，在 platform 节点下添加子键"debug\_mode = 8"即表示开启所有打印，debug\_mode=0 表示关闭启动时 U-Boot 的打印log, 未显式配置 debug\_mode 时，按 debug\_mode=8 处理。目前常用的打印等级有 0（关闭所有打印）、1（只显示关键节点打印）、4（打印错误信息）、8（打印所有 log 信息）。

1. usb\_debug 在烧录或启动过程中，若遇到烧录失败或启动失败大致挂死在 usb 相关模块，但又不确定具体位置，这时可以打开usb\_debug进行调试，开启usb\_debug后有关 usb 相关的运行信息会被较详细打印出来。打开usb\_debug的方式：打开usb\_base.h文件，将其中的#defineSUNXI\_USB\_DEBUG宏定义打开，打开后重新编译 U-Boot 并打包烧录即可。

### 7 进入烧写的方法

1. 开机时按住 fel 键
2. 开机时打开串口按住键盘数字’2’
3. 进入 U-Boot 控制台输入efex
4. 进入 Android 控制台输入 reboot efex

### 8 常用接口函数

### 8.1 fdt 相关接口

1. const void \*fdt\_getprop(const void \*fdt, int nodeoffset, const char \*name, int \*lenp)

*•* 作用：检索指定属性的值

*•* 参数：

*•* fdt: 工作 flattened device tree

*•* nodeoffset: 待修改节点的偏移

*•* name: 待检索的属性名

*•* lenp: 检索属性值的长度（会被覆盖）或者为 NULL

*•* 返回：

*•* 非空（属性值的指针）：成功

*•* NULL（lenp 为空）：失败

*•* 失败代码（lenp 非空）：失败

1. int fdt\_set\_node\_status(void \*fdt, int nodeoffset, enum fdt\_status status, unsigned int error\_code)

*•* 作用：设置节点状态

*•* 参数：

*•* fdt: 工作 flattened device tree

*•* nodeoffset: 待修改节点的偏移

*•* status:FDT\_STATUS\_OKAY, FDT\_STATUS\_DISABLED, FDT\_STATUS\_FAIL, FDT\_STATUS\_FAIL\_ERROR\_CODE

*•* error\_code:optional, only used if status is FDT\_STATUS\_FAIL\_ERROR\_CODE

*•* 返回：

*•* 0: 成功

*•* 非 0: 失败

1. int fdt\_path\_offset(const void \*fdt, const char \*path)

*•* 作用：通过全路径查找节点的偏移量

*•* 参数：

*•* fdt: 工作 fdt

*•* path: 全路径名称

*•* 返回：

*•* >=0(节点的偏移量): 成功

*•* <0: 失败代码

1. static inline int fdt\_setprop\_u32(void \*fdt, int nodeoffset, const char \*name, uint32\_t val)

*•* 作用：将属性值设置为一个 32 位整型数值，如果属性值不存在，则新建该属性

*•* 参数：

*•* fdt: 工作 flattened device tree

*•* nodeoffset: 待修改节点的偏移

*•* name: 待修改的属性名

*•* val:32 位目标值

*•* 返回：

*•* 0: 成功

*•* <0: 失败代码

1. static inline int fdt\_setprop\_u64(void \*fdt, int nodeoffset, const char \*name, uint64\_t val)

*•* 作用：与fdt\_setprop\_u32类似，将属性值设置为一个 64 位整型数值，如果属性值不存在，则新建该属性

*•* 参数：

*•* fdt: 工作 flattened device tree

*•* nodeoffset: 待修改节点的偏移

*•* name: 待修改的属性名

*•* val:64 位目标值

*•* 返回：

*•* 0: 成功

*•* <0: 失败代码

1. #define fdt\_setprop\_string(fdt, nodeoffset, name, str) fdt\_setprop((fdt), (nodeoffset), (name), (str), strlen(str)+1)

*•* 作用：将属性值设置为一个字符串，如果属性值不存在，则新建该属性

*•* 参数：

*•* fdt: 工作 flattened device tree

*•* nodeoffset: 待修改节点的偏移

*•* name: 待修改的属性名

*•* str: 目标值

*•* 返回：

*•* 0: 成功

*•* <0: 失败代码

注意：在sys\_config.fex的配置中，节点的启用状态为 0 或 1。转换到 fdt 中对应的 status 属性为disable或okay。

1. int save\_fdt\_to\_flash(void \*fdt\_buf, size\_t fdt\_size)

*•* 作用：保存修改到 flash

*•* 参数：

*•* fdt\_buf: 当前工作 flattened device tree

*•* fdt\_size: 当前工作 flattened device tree 的大小，可以通过fdt\_totalsize(fdt\_buf )获取

*•* 返回：

*•* 0: 成功

*•* <0: 失败

1. 应用参考

U-Boot 中 fdt 命令行的实现：cmd/fdt.c

### 8.2 env 相关接口函数

1. int env\_set(const char \*varname, const char \*varvalue)

*•* 作用：将环境变量 varname 的值设置为 varvalue，重启失效

*•* 参数：

*•* varname: 待设置环境变量的名称

*•* varvalue: 将指定的环境变量修改为该值

*•* 返回：

*•* 0: 成功

*•* 非 0: 失败

1. char \*env\_get(const char \*name)

*•* 作用：获取指定环境变量的值

*•* 参数：

*•* name: 变量名称

*•* 返回：

*•* NULL: 失败

*•* 非空（环境变量的值）：成功

1. int env\_save(void)

*•* 作用：保存环境变量，重启仍保存

*•* 参数: 无

*•* 返回：

*•* 0: 成功

*•* 非 0: 失败

1. 应用参考

board/sunxi/sunxi\_bootargs.c update\_bootargs通过 cmdline 向 kernel 提供信息，主要是通过更新bootargs变量实现env\_set("bootargs", cmdline)。

### 8.3 调用 U-Boot 命令行

1. int run\_command\_list(const char \*cmd, int len, int flag)

*•* 作用：执行 U-Boot 命令行

*•* 参数：

*•* cmd: 命令字符指针

*•* len: 命令行长度，设置为-1 则自动获取

*•* flag: 任意，因为 sunxi 中没有用到

*•* 返回：

*•* 0: 成功

*•* 非 0: 失败

1. 应用参考：

common/autoboot.c autoboot\_command实现了 U-Boot 的自动启动命令

s = env\_get("bootcmd");

run\_command\_list(s, -1, 0)。

### 8.4 Flash 的读写

1. int sunxi\_flash\_read(uint start\_block, uint nblock, void \*buffer)

*•* 作用：将指定起始位置start\_block的nblock读取到buffer

*•* 参数：

*•* start\_block: 起始地址

*•* nblock:block 个数

*•* buffer: 内存地址

*•* 返回：

*•* 0: 成功

*•* 非 0: 失败

1. int sunxi\_flash\_write(uint start\_block, uint nblock, void \*buffer)

*•* 作用：将buffer写入指定起始位置start\_block的nblock中

*•* 参数：

*•* start\_block: 起始地址

*•* nblock:block 个数

*•* buffer: 内存地址

*•* 返回：

*•* 0: 成功

*•* 非 0: 失败

1. int sunxi\_sprite\_read(uint start\_block, uint nblock, void \*buffer)

*•* 作用与sunxi\_flash\_read相似

1. int sunxi\_sprite\_write(uint start\_block, uint nblock, void \*buffer)

*•* 作用与sunxi\_flash\_write相似

1. 应用参考

common/sunxi/board\_helper.c sunxi\_set\_bootcmd\_from\_mis实现了对 misc 分区的读写操作

### 8.5 获取分区信息

1. int sunxi\_partition\_get\_partno\_byname(const char \*part\_name)

*•* 作用：根据分区名称获取分区号

*•* 参数：

*•* part\_name: 分区名称

*•* 返回：

*•* <0: 失败

*•* >0（分区号）：成功

1. int sunxi\_partition\_get\_info\_byname(const char \*part\_name, uint \*part\_offset, uint \*part\_size)

*•* 作用：根据分区名称获取分区的偏移量和大小

*•* 参数：

*•* part\_name: 分区名称

*•* part\_offset: 分区的偏移量

*•* part\_size: 分区的大小

*•* 返回：

*•* 0: 成功

*•* -1: 失败

1. uint sunxi\_partition\_get\_offset\_byname(const char \*part\_name)

*•* 作用：根据分区名称获取偏移量

*•* 参数：

*•* part\_name: 分区名称

*•* 返回：

*•* <=0 : 失败

*•* >0 : 成功

1. int sunxi\_partition\_get\_info(const char \*part\_name, disk\_partition\_t \*info)

*•* 作用：根据part\_name获取分区信息

*•* 参数：

*•* part\_name: 分区名称

*•* info: 分区信息

*•* 返回：

*•* 非 0: 失败

*•* 0: 成功

1. lbaint\_t sunxi\_partition\_get\_offset(int part\_index)

*•* 作用：card sprite 模式下获取分区的偏移量

*•* 参数：

*•* part\_index: 分区号

*•* 返回：

*•* >=0（偏移量）：成功

*•* -1: 失败

1. 应用参考

启动时加载图片：drivers/video/sunxi/logo\_display/sunxi\_load\_bmp.c

### 8.6 GPIO 相关操作

1. int fdt\_get\_one\_gpio(const char\* node\_path, const char\* prop\_name,user\_gpio\_set\_t\* gpio\_list)

*•* 作用：根据路径node\_path和 gpio 名称prop\_name获取 gpio 配置

*•* 参数：

*•* node\_path：fdt 路径

*•* prop\_name：gpio 名称

*•* gpio\_list：待获取的 gpio 信息

*•* 返回：

*•* 0：成功

*•* -1：失败

1. ulong sunxi\_gpio\_request(user\_gpio\_set\_t \*gpio\_list, \_\_u32 group\_count\_max)

*•* 作用：根据 gpio 配置获取 gpio 操作句柄

*•* 参数：

*•* gpio\_list：gpio 配置列表，可以由fdt\_get\_one\_gpio获得

*•* group\_count\_max: gpio\_list中最大的 gpio 配置个数

*•* 返回：

*•* 0：失败

*•* >0（gpio 操作句柄）：成功

1. **s32 gpio\_write\_one\_pin\_value(ulong p\_handler,** u32 value\_to\_gpio, const char \*gpio\_name)

*•* 作用：根据 gpio 操作句柄写数据

*•* 参数：

*•* p\_handler：gpio 操作句柄，可由sunxi\_gpio\_request获取

*•* value\_to\_gpio：待写入数据，0 或 1

*•* gpio\_name：gpio 名称

*•* 返回：

*•* EGPIO\_SUCCESS：成功

*•* EGPIO\_FAIL：失败

1. 应用参考

操作 led 状态：

ssprite/sprite\_led.c  
  
user\_gpio\_set\_t gpio\_init;  
  
fdt\_get\_one\_gpio("/soc/card\_boot", "sprite\_gpio0", &gpio\_init); //获取/soc/card\_boot中sprite\_gpio0的gpio配置  
  
sprite\_led\_hd = sunxi\_gpio\_request(&gpio\_init, 1); //获取gpio操作句柄  
  
gpio\_write\_one\_pin\_value(sprite\_led\_hd, sprite\_led\_status, "sprite\_gpio0"); //操作led状态

### 9 常用资源的初始化阶段

*•* env ：环境变量初始化后可以访问

*•* fdt ：在 U-Boot 运行开始即可访问

*•* malloc ：在重定位后才能访问

<https://zhuanlan.zhihu.com/p/609754486?utm_id=0>

## 4. Linux 内核配置

一、Linux内核的配置

驱动进入内核有两种方式：模块和直接编译进内核  
Linux内核的配置系统由三个部分组成，分别是：  
1、Makefile：分布在 Linux 内核源代码根目录及各层目录中，定义 Linux 内核的编译规则  
2、配置文件：.config可执行的脚本文件，给用户提供配置选择的功能  
3、配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面以及基于 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）  
1.1 配置用户界面

主要分为三种方式：

1、make config（基于文本的最为传统的配置界面，不推荐使用）   
2、make menuconfig（基于文本选单的配置界面，字符终端下推荐使用）   
3、make xconfig（基于图形窗口模式的配置界面，Xwindow下推荐使用）

1  
2  
3

目的都是生成一个.config文件，有三种选择，它们分别代表的含义如下：  
Y－-将该功能编译进内核  
N－-不将该功能编译进内核  
M－-将该功能编译成可以在需要时动态插入到内核中的模块  
二、make menuconfig

Menuconfig 配置内核原理： 在Linux里面我们所看到的 menuconfig 界面是通过配置内核顶层的Kconfig产生的，而当输入make menuconfig命令的时候系统会读取Makefile来解析Kconfig。 通常会在Kconfig里面编写以下四项：

1、模块的名字，用module开头  
2、选项，通常设为bool(二选一)或者trastate(三选一)  
3、默认选项  
4、帮助说明

1  
2  
3  
4

2.1 Loadable module support

含义：对模块的支持  
2.2 Networking options

含义：网络选项

Packet Socket：该选项用来与网卡进行通信而不需要在内核中实现网络协议  
TCP/IP networking：内核将支持TCP/IP协议

2.3 Console drivers

含义：配置控制台驱动

VGA text console：在VGA模式下启动字符模式。多数人都需要它  
video mode selection support：使启动的时候能够使用字符模式的分辨率  
Sound sound：声卡驱动，可以配置声卡

————————————————  
版权声明：本文为CSDN博主「future\_sky\_word」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/future_sky_word/article/details/128512412>

## 5. 添加一个 Linux 源码文件

参考用书：《操作系统实践：基于Linux的应用与内核编程》

一．添加一个内核模块

1.1需求分析

对于一个应用程序而言，源代码经编译后与标准运行库链接，通过系统调用执行操作系统内核中的特权指令，指令返回的结果通过系统调用返回给用户，完成程序。

由于Linux是单内核多模块的操作系统，整个操作系统只包含一个由多个不同功能模块组成的内核，通过Linux系统的这种特性,可以动态添加自己编写的内核模块，并由系统每次启动时进行加载。

通过内核模块的动态添加和删除，可以直观的感受到内核模块的编写以及动态添加和删除的过程。可以通过此次实验理解用户态（User Model）和内核态（Kernel Model）的关系。并掌握内核模块代码中的一些常见宏和参数以及内核模块程序和应用程序的差异。

实验流程为：编写kello.c文件和Makefile文件，并使用insmod和rmmmod命令对内核模块进行安装和卸载。观察终端输出的不同信息。同时编写代码时应注意安全问题，不要对内核进行不当访问从而使系统出现错误。

输入：使用insmod安装所编写的内核模块。

输出：通过dmesg命令观察安装内核模块和卸载内核模块所输出的信息。

1.2概要设计

本次实验包含两个文件：kello.c和编译所需的Makefile文件，kello.c文件包含模块初始化函数hello\_init，在模块初始化时被调用，以及模块回收函数hello\_exit，在模块被撤销时使用。

在该文件的最后，需要添加两行声明module\_init(hello\_init)和module\_exit(hello\_exit)，使模块被更高效的加载执行和撤销，加快内存的分配和回收

对于Makefile文件内容，KERNELRELEASE定义在linux内核源代码中的顶层makefile，boj-m表示所安装内核模块的名称，需要和c文件名字相同。

KDIR和PWD目录分别代表内核的源代码目录和当前的工作路径，defult中为编译命令以及一些文件的删除命令。

1.3详细设计

Kello.c的hello\_init调用printk输出一段初始化信息，hello\_exit调用printk输出模块回收时打印的信息。

对于Makefile文件，KERNELRELEASE是在Linux内核源代码顶层Makelile中定义的一个变量，在第一次读取执行此 Makefile文件时，KERNELRELEASE没有被定义，所以make将读取执行else之后的内容。

如果make的目标是clean. 直接执行clean操作,然后结束。

当没有声明make的目标时,make执行默认操作,即default 后的指令，此时-C (PWD) 表明需要返回到当前目录继续读入并执行当前的Makefile.当从内核源代码目录返回时, KERNELRELEASE已被定义,kbuild也被启动去解析kbuild语法的语句，make将继续读取else之前的内容。

实验过程：

在当前文件夹下执行make命令

观察当前文件夹是否生成所需文件

sudo insmod 安装LKM

观察安装LKM后dmesg输出的内核初始化信息

sudo rmmod 卸载LKM

观察卸载LKM后dmesg输出的内核初始化和撤销信息

1.4调试分析

分析一:make命令提示空变量名错误

首先遇到的错误为，make命令报告空变量名。是make文件的格式问题，注意在(KDIR) SUBDIRS =$(shell pwd) modules，SUBDIRS的后没有空格，去掉空格后问题解决。

分析二：mkdir创建目录权限不够

mkdir创建目录权限不够，错误定位在fixdep.c文件的404行，fatal error打开依赖文件的权限不够，并显示compilation terminated（编译结束）。

fixdep.c文件是make的一个工具，由make在编译时调用。问题出在make -C SUBDIRS=处，应修改为make -C M=。二者的区别在于，$(MAKE)相当于make，-C 选项的作用是指将当前工作目录转移到所指定的位置。“M=”表示当用户需要以某个内核为基础编译一个外部模块的话，需要在make modules 命令中加入“M=dir”，程序会到所指定的dir目录中查找模块源码，将其编译，生成.ko文件。

M并非随意命名，在新的内核模块编程中的make命令中有个M选项，是makefile脚本中的一个变量，(KDIR) M=(MAKE) -C (PWD)的作用是一样的，不过SUBDIRS是过时的使用方法，修改后问题得以解决。

make文件和c文件如下：

ifneq ($(KERNELRELEASE),)  
obj-m := kello.o  
else   
KDIR := /lib/modules/$(shell uname -r)/build  
PWD := $(shell pwd)  
default:  
 $(MAKE) -C $(KDIR) M=$(shell pwd) modules  
 rm -r -f .tmp\_versions \*.mod.c .\*.cmd \*.o \*.symvers  
endif  
  
#include<linux/module.h>  
   
/\*  
此函数为模块初始化实例，  
将在模块初始化时被调用  
\*/  
int hello\_init(void)  
{  
 printk("\nI AM HERE IN KERNEL:)\n");  
 return 0;  
}  
   
/\*  
此函数为模块回收实例，  
将在模块被撤销时使用  
\*/  
void hello\_exit(void){  
 printk("\n I AM OUT BYE BYE :)\n");  
}  
   
MODULE\_AUTHOR("zhuoge04");//描述模块作者为本人  
MODULE\_LICENSE("GPL");  
   
/\*  
以下两行声明可以使模块被更高效的加载执行和撤销，  
加快内存的分配和回收  
\*/  
module\_init(hello\_init);  
module\_exit(hello\_exit);

————————————————  
版权声明：本文为CSDN博主「日赴一日」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/m0_49071428/article/details/128577284>

## 6. busybox 功能和作用

在嵌入式系统构建中，Busybox可用于构建轻量级的[根文件系统](https://www.zhihu.com/search?q=根文件系统&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType"%3A"answer"%2C"sourceId"%3A2714183182})

，本文从源码结构和源码入口角度分析busybox，了解其背后的运作机制。

busybox版本：1.35.0

## **一、Busybox简介**

### **（1-1）开源项目**

Busybox是一个开源项目，遵循GPL v2协议。Busybox将众多的UNIX命令集合进了一个很小的可执行程序中，可以用来替代GNU fileutils、shellutils等工具集。Busybox中各种命令与相应的GNU工具相比，所能提供的选项比较少，但是对于一般的应用场景也足够了，特别是在嵌入式系统的设计中。

### **（1-2）程序本体较小**

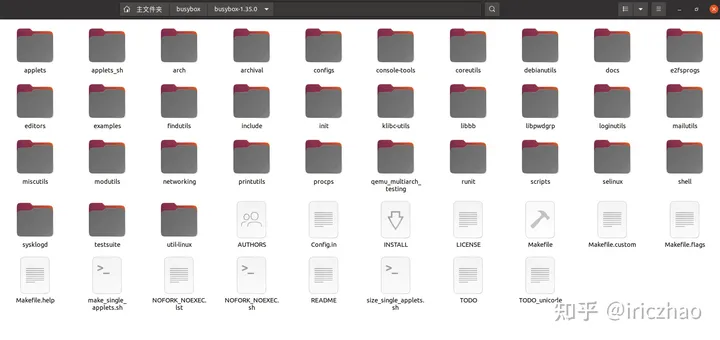
Busybox在设计过程中对文件大小进行了优化，并考虑了系统资源有限(比如内存等)的情况。与一般的GNU工具集动辄几M的体积相比，动态链接的Busybox只有几百K，即使是采用静态链接也只有1M左右。除此之外，Busybox按模块设计，可以很容易地加入、去除某些命令，或增减命令的某些选项。

### **（1-3）使用简单**

如果使用Busybox来创建根文件系统，使用起来比较方便，只需要在/dev目录下创建必要的设备节点，在/etc目录下增加一些配置文件即可，当然如果Busybox是动态链接的，那么还需要在/lib目录下包含相关的运行库文件。

## **二、Busybox源码目录结构**

在较老版本的Busybox中，对于Busybox的多个程序是全部塞进了一个名为utility.c的文件中，后来更改了Busybox的整体源码结构和设计，将这些程序拆分成了各个工具模块。其目录结构如下图所示：



|  |  |  |
| --- | --- | --- |
| 序号 | 目录名称 | 功能说明 |
| 1 | applets | 实现applets框架的文件。目录中包含了几个main()的文件 |
| 2 | applets\_sh | 此目录包含了几个作为shell脚本实现的applet示例。在“make install”时不会被自动安装，需要使用时，手动处理 |
| 3 | arch | 包含用于不同体系架构的makefile文件。约束busybox在不同架构体系下的编译构建过程 |
| 4 | archival | 与压缩相关命令的实现源文件。 |
| 5 | configs | busybox自带的默认配置文件 |
| 6 | console-tools | 与控制台相关的一些命令 |
| 7 | coreutils | 常用的一些核心命令。例如chgrp、rm等 |
| 8 | debianutils | 针对Debian的套件。 |
| 9 | e2fsprogs | 针对Linux Ext2 FS prog的命令。例如chattr、lsattr |
| 10 | editors | 常用的编辑命令。例如diff、vi等 |
| 11 | findutils | 用于查找的命令 |
| 12 | include | busybox项目的头文件 |
| 13 | init | init进程的实现源码目录 |
| 14 | klibc-utils | klibc命令套件 |
| 15 | libbb | 与busybox实现相关的库文件 |
| 16 | libpwdgrp | libpwdgrp相关的命令 |
| 17 | loginutils | 与用户管理相关的命令 |
| 18 | mailutils | 与mail相关的命令套件 |
| 19 | miscutils | 该文件下是一些杂项命令，针对特定应用场景 |
| 20 | modutils | 与模块相关的命令 |
| 21 | networking | 与网络相关的命令，例如arp |
| 22 | printutils | Print相关的命令 |
| 23 | procps | 与内存、进程相关的命令 |
| 24 | runit | 与Runit实现相关的命令 |
| 25 | shell | 与shell相关的命令 |
| 26 | sysklogd | 系统日志记录工具相关的命令 |
| 27 | util-linux | Linux下常用的命令，主要与文件系统操作相关的命令。 |

## **三、Busybox程序主体**

Busybox是在linux内核启动后加载运行的用户空间程序，在源码设计上是基于C语言完成设计和开发的。与常规程序一样，Busybox的入口同样是main()，定义在libbb/appletlib文件的末尾处。在函数开始处，使用ENABLE\_BUILD\_LIBBUSYBOX对函数名称进行了条件分支处理：如果ENABLE\_BUILD\_LIBBUSYBOX为真，则表示将Busybox以库的方式进行构建。

在函数体中，以条件宏定义进行代码的编译逻辑控制：

/\* Tweak malloc for reduced memory consumption \*/  
#ifdef M\_TRIM\_THRESHOLD  
 /\* M\_TRIM\_THRESHOLD是释放的最顶层内存的最大数量  
 \* 默认值太大，是256k  
 \*/  
 mallopt(M\_TRIM\_THRESHOLD, 8 \* 1024);  
#endif  
#ifdef M\_MMAP\_THRESHOLD  
 /\* M\_MMAP\_THRESHOLD是使用mmap()的请求大小阈值。  
 \* 默认值是256k  
 \*/  
 mallopt(M\_MMAP\_THRESHOLD, 32 \* 1024 - 256);  
#endif

上述代码都调用了mallopt()函数，该函数用于设置内存的分配参数，由于默认值太大（为256KB），故此处调整内存分配大小，让出多余的内存。

接着，是一个由#if -- #elif -- #else -- #endif控制的条件宏多分支判断结构语句，此处以Busybox的一般运行情况为例（在Linux内核启动后期，加载并运行Busybox构建出的init程序）。其执行逻辑如下：

首先Busybox是一个linux下的工具集合，本质则是一个个的命令，例如：ls、mv、cp等，在命令行我们输入想要执行的操作时，例如：mkdir iriczhao，则会将参数传递给Busybox，然后由他完成对应的操作。

在源码中，使用char \* applet\_name表示工具的名称（本质是字符串），首先会调用lbb\_prepare()函数：

lbb\_prepare("busybox" IF\_FEATURE\_INDIVIDUAL(, argv));

将会设置applet\_name的值为“busybox“，用于执行ENABLE\_FEATURE\_INDIVIDUAL为真时的逻辑操作：

void lbb\_prepare(const char \*applet  
 IF\_FEATURE\_INDIVIDUAL(, char \*\*argv))  
{  
#ifdef bb\_cached\_errno\_ptr  
 ASSIGN\_CONST\_PTR(&bb\_errno, get\_perrno());  
#endif  
 applet\_name = applet;  
  
 if (ENABLE\_LOCALE\_SUPPORT)  
 setlocale(LC\_ALL, "");  
  
#if ENABLE\_FEATURE\_INDIVIDUAL  
 /\* Redundant for busybox (run\_applet\_and\_exit covers that case)  
 \* but needed for "individual applet" mode \*/  
 if (argv[1] && !argv[2] && strcmp(argv[1], "--help") == 0 && !is\_prefixed\_with(applet, "busybox"))  
 {  
 /\* Special cases. POSIX says "test --help"  
 \* should be no different from e.g. "test --foo".  
 \*/  
 if (!(ENABLE\_TEST && strcmp(applet\_name, "test") == 0) && !(ENABLE\_TRUE && strcmp(applet\_name, "true") == 0) && !(ENABLE\_FALSE && strcmp(applet\_name, "false") == 0) && !(ENABLE\_ECHO && strcmp(applet\_name, "echo") == 0))  
 bb\_show\_usage();  
 }  
#endif  
}

接着，会解析命令行传递的第一个参数：

applet\_name = argv[0];  
 if (applet\_name[0] == '-')  
 applet\_name++;  
 applet\_name = bb\_basename(applet\_name);

例如，在命令行输入mkdir iriczhao命令，则会解析到mkdir命令传递给applet\_name，至于后面的参数（此处是iriczhao）是如何传递的，后文会描述到。

如果配置了FEATURE\_SUID\_CONFIG宏定义，在parse\_config\_file()函数中还将从/etc/busybox.conf文件中解析关于busybox的配置参数。

在最后，则是busybox的重要函数：run\_applet\_and\_exit()，该函数定义如下：

static NORETURN void run\_applet\_and\_exit(const char \*name, char \*\*argv)  
{  
#if ENABLE\_BUSYBOX  
 //检查是否是带有busybox前缀的字符串，如果不是，则返回NULL。  
 //如果在命令行下输入具体的命令，则不是带有busybox前缀的命令字符串，则不会执行该条件下的语句  
 if (is\_prefixed\_with(name, "busybox"))  
 exit(busybox\_main(/\*unused:\*/ 0, argv));  
#endif  
#if NUM\_APPLETS > 0  
 /\* find\_applet\_by\_name() search is more expensive, so goes second \*/  
 {  
 int applet = find\_applet\_by\_name(name);  
 if (applet >= 0)  
 run\_applet\_no\_and\_exit(applet, name, argv);  
 }  
#endif  
  
 /\*bb\_error\_msg\_and\_die("applet not found"); - links in printf \*/  
 full\_write2\_str(applet\_name);  
 full\_write2\_str(": applet not found\n");  
 /\* POSIX: "If a command is not found, the exit status shall be 127" \*/  
 exit(127);  
}

如果NUM\_APPLETS大于0，则会执行对应的命令操作，并退出；否则，busybox将会报错：

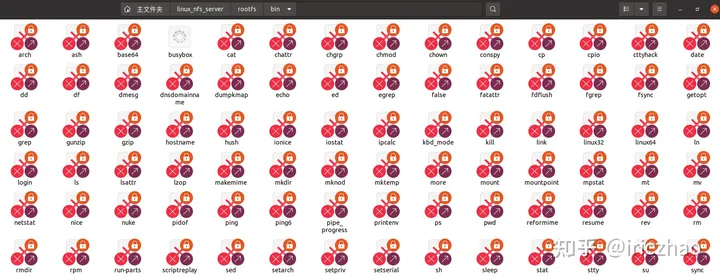
#if NUM\_APPLETS > 0  
 /\* find\_applet\_by\_name() search is more expensive, so goes second \*/  
 {  
 int applet = find\_applet\_by\_name(name);  
 if (applet >= 0)  
 run\_applet\_no\_and\_exit(applet, name, argv);  
 }  
#endif  
   
 //正常情况下（NUM\_APPLETS > 0），不会执行下述代码。  
 /\*bb\_error\_msg\_and\_die("applet not found"); - links in printf \*/  
 full\_write2\_str(applet\_name);  
 full\_write2\_str(": applet not found\n");  
 /\* POSIX: "If a command is not found, the exit status shall be 127" \*/  
 exit(127);

从上述代码可知，在命令行键入命令后，实则起关键作者的函数是：find\_applet\_by\_name()和run\_applet\_no\_and\_exit()。下文将继续分析。

## **四、Busybox程序运行剖析和实列**

在上一小节中，已经知道当我们在busybox的命令行下，键入命令后，执行具体操作的函数是：find\_applet\_by\_name()和run\_applet\_no\_and\_exit()。

在编译构建源码并安装busybox后，在安装目录下的文件结构则是一个名为busybox的可执行程序和很多的链接，这些链接实则是我们在命令行键入的命令名称。如下图所示：



从源码角度看，busybox中的命令都有一一对应的执行函数，其函数命名格式为xxx\_main()，在源码设计上，其内部在/include/applet\_tabls.h头文件中维护了一张**命令表**，定义如下（代码太长，有省略）：

int (\*const applet\_main[])(int argc, char \*\*argv) = {  
test\_main,  
test\_main,  
acpid\_main,  
add\_remove\_shell\_main,  
addgroup\_main,  
adduser\_main,  
adjtimex\_main,  
uname\_main,  
arp\_main,  
arping\_main,  
ascii\_main,  
ash\_main,  
awk\_main,  
baseNUM\_main,  
baseNUM\_main,  
basename\_main,  
//省略大量内容  
//...  
}

上述[函数指针](https://www.zhihu.com/search?q=函数指针&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType"%3A"answer"%2C"sourceId"%3A2714183182})

数组中的元素则是分布于busybox源码各个目录下命令入口函数。在代码执行逻辑中，首先会调用find\_applet\_by\_name()函数，通过传入的命令名称获取在命令表中的数组下标。并将命令对应的下标applet、命令名称name和命令行参数字符串argv传递给run\_applet\_no\_and\_exit()函数（注：解释了上一小节中，命令行对应命令后面的参数是如何传递的），该函数定义如下：

void FAST\_FUNC run\_applet\_no\_and\_exit(int applet\_no, const char \*name, char \*\*argv)  
{  
 int argc;  
  
 /\*  
 \* We do not use argv[0]: do not want to repeat massaging of  
 \* "-/sbin/halt" -> "halt", for example.  
 \*/  
 applet\_name = name;  
  
 show\_usage\_if\_dash\_dash\_help(applet\_no, argv);  
  
 if (ENABLE\_FEATURE\_SUID)  
 check\_suid(applet\_no);  
  
 argc = string\_array\_len(argv);  
 xfunc\_error\_retval = applet\_main[applet\_no](argc, argv);  
  
 /\* Note: applet\_main() may also not return (die on a xfunc or such) \*/  
 xfunc\_die();  
}  
#endi

在上述代码中，执行命令下的对应具体操作函数的语句是：

xfunc\_error\_retval = applet\_main[applet\_no](argc, argv);

applet\_main是命令表数组，applet\_no是对应命令的数组下标，本质则是调用对应的applet\_main命令表数组中的元素（函数指针），并将argc和argv作为参数给了对应的命令执行函数。

<https://www.zhihu.com/question/32018633/answer/2714183182>

# 第九部分：驱动程序

## 1. Linux 平台设备驱动

前言

要满足 Linux 设备模型，就必须有总线、设备和驱动。但是有的设备并没有对应的物理总线，比如 LED、RTC 和蜂鸣器等。为此，内核专门开发了一种虚拟总线一-platfomm总线，用来连接这些没有物理总线的设备或者一些不支持热插拔的设备，DM9000 网卡

设备就是挂接在这条总线上的。  
一、平台设备

平台设备是用structplatform device 结构来表示的，它的定义如下

驱动开发者关心的主要成员如下。  
  
 name: 设备的名字，在平台总线的 match 函数中可用于同平台驱动的匹配  
  
 id:设备的ID 号，用于区别同类型的不同平台设备。  
 dev:内的 struct device。  
 num\_resources:平台设备使用的资源个数。  
 resource: 平台设备的资源列表 (数组)，指向资源数组中的首元素.  
  
 id\_entry:用于同平台驱动匹配的 ID,在平台总线的 match 函数中首先尝试匹配该 ID,如果不成功再尝试用 name 成员来匹配。  
  
 在平台设备中，最关键的就是设备使用的资源信息的描述，这是实现设备和驱动分离的关键。struct resource 的定义如下

驱动开发者关心的主要成员如下。  
 start: 资源的开始，对于 IO 内存来说就是起始的内存地址，对于中断资源来说就是起始的中断号，对于 DMA 资源来说就是起始的 DMA 通道号。  
 end:资源的结束。  
 flags:资源的标志，定义在“include/linux/ioport.h”文件中，最常见的有如下几种.

IORESOURCE\_MEM:资源的类型是内存资源，也包括I/O内存。  
 IORESOURCE\_IRO:资源的类型是中断资源。  
 IORESOURCE\_DMA:资源的类型是 DMA 通道资源。  
 资源可以组成一个树形结构，由成员parent、sibling和child来完成.  
  
 平台设备及其资源通常存在于BSP(Board Support Package，板级支持包)文件中,该文件通常包含和目标板相关的一些代码。例如对于 QT2410 目标板，其对应的 BSP文件为arch/arm/mach-s3c24xx/mach-gt2410.c，现将其描述CS8900网卡的平台设备摘录如下。

CS8900 平台设备有两个资源，分别是 IORESOURCE\_MEM 和 IORESOURCE\_IRQ两种类型的，并用宏 DEFINE\_RES\_MEM 和 DEFINE\_RES\_IRQ 来定义。对于 DEFINERES\_MEM 宏，里面的两个参数分别是内存的起始地址和大小:对于 DEFINE\_RES\_IRQ宏，里面的参数则是中断号。读者可以自行查看这两个宏的定义，最终是对 start、end和flags成员进行了赋值。最终定义的平台设备是 qt2410\_cs89x0，ARRAY\_SIZE 是用于获取数组元素个数的宏。

向平台总线注册和注销的平台设备的主要函数如下。

platform\_add\_devices用于一次注册多个平台设备，platform\_device\_register 一次只注册一个平台设备。其实，platform\_add\_devices 是通过多次调用 platform\_device\_register来实现的。platform\_device\_unregister 用于注销平台设备。  
 当平台总线发现有和平台设备匹配的驱动时，就会调用平台驱动内的一个函数，并传递匹配的平台设备结构地址，平台驱动就可以从中获取设备的资源信息。关于资源操作的主要函数如下。

platform\_get\_resource:从平台设备 dev 中获取类型为type、序号为num的资源.  
  
 resource\_size:返回资源的大小，其值为 end-start +1。  
  
 例如，在 CS8900 网卡驱动中就有如下的代码来获取资源及其大小。

代码第1857行获取了IORESOURCE\_MEM资源，序号为0。代码第1858 行获取了IORESOURCE\_IRQ 资源，序号也为0。所以，当资源类型不同后，序号重新开始编号.代码第1865行获取了内存资源的大小。代码第1872行使用ioremap将内存资源进行映射得到映射后的虚拟地址。

二、平台驱动

平台驱动是用struct platform\_driver 结构来表示的，它的定义如下

驱动开发者关心的主要成员如下。  
 probe: 总线发现有匹配的平台设备时调用。

remove:所驱动的平台设备被移除时或平台驱动注销时调用。  
  
 shutdown、suspend 和 resume: 电源管理函数，在要求设备电、挂起和恢复时被调用。内嵌的 struct device\_driver 的 pm 成员也有对应的电源管理函数。  
  
 id\_table: 平台驱动可以驱动的平台设备ID 列表，可用于和平台设备匹配。  
  
 向平台总线注册和注销的平台驱动的主要函数如下。

因为在驱动中，经常在模块初始化函数中注册一个平台驱动，在清除函数中注销一个平台驱动，所以内核定义了一个宏来简化这些代码，宏的定义如下。

三、平台驱动简单实例

在前面的基础之上，我们可以先来编写一个简单的平台驱动，再编写一个模块来注册两个设备，代码如下

在 pltdev.c 文件中，代码第 7 行至第 29 行分别定义了两个平台设备，id为0和1以示区别，名字都为 pdev，没有使用任何资源。在模块的初始化函数和清除函数中分别注册和注销了这两个平台设备。  
 在 pltdrv.c 文件中，代码第 34 行至第42 行定义了一个平台驱动，名字也为pdev,这样才能和平台设备匹配。pm 是电源管理函数的集合，实现了挂起和恢复两个电源管理操作。因为是虚拟设备，所以并没有做任何电源管理相关的操作。为了简单,probe 和remove函数也只是返回成功而已。代码第44行使用module\_platform\_driver这个宏来简化模块初始化函数和卸载函数的编写。  
 编译和测试的命令如下

从上面的测试结果可以看到，平台驱动驱动了两个设备 pdev.0 和 pdev.1，这是设备名字加过构成的名字。

四、 电源管理

在平台驱动里面实现了挂起和恢复两个电源管理函数，从而可以管理设备的电源状态。 /sys/devices/platform/pdev.0/power/control 和 /sys/devices/platform/pdev.1/power/control两个文件可以用来管理两个设备的电源控制方式，如果文件的内容为 auto，那么设备的电源会根据系统的状态自动进行管理，为on则表示打开。我们首先确定电源控制方式为自动，可以使用下面的命令进行确认。

貌似只跑一个linux的内核不行捏

没有挂起命令，只使用串口貌似也不能挂起，那这样的话就还是用ubuntu吧

接下来将 Ubuntu 系统挂起

喵的恢复不了了

试试用vm的挂起能不能起到效果

还是不行下面我修改一下驱动程序看看能不能在挂起后立刻自动恢复

看看上面的选项是不是y并且确定他有唤醒方式

API和版本貌似对不上换一下试试

我又修改了一下代码

这次没挂起直接关机了

使用命令后挂起成功了但是又是黑屏状态估计哪没配置好

服了，这里我环境不行就当成功验证了吧，后面我看看怎么修改一下环境或者直接在板子上跑这样很容易恢复。

系统挂起后,再重新恢复系统,使用 dmesg 命令可以看到,驱动中的 suspend 和 resume函数先后都被调用了两次。

# dmesg

[ 171.396323] pdev: suspend  
[ 171.396325] pdev; suspend  
......

176.699954] pdev:resume

[ 176.699959] pdev: resume

五、udev 和驱动的自动加载

在上面的例子中，我们可以通过加载模块来向系统添加两个设备，也可以通过移除模块来删除这两个设备。对于这样的操作，我们想使设备被添加到系统后，其驱动能够自动被加载，这对于实际的可支持热插拔的硬件来说更有必要。比如，我们插入一个USB无线网卡，那么对应的驱动就应该自动加载，而不是由用户来手动加载。要做到这一点,就必须利用到一个工具-udev，在入式系统中通常使用 mdev，其功能比 udev 要弱很多，但也可以移植 udev 到嵌入式系统上。  
 使用了 Linux 设备模型后，任何设备的添加、删除或状态修改都会导致内核向用户空间发送相应的事件，这个事件叫 uevent，和 kobiect 密切关联。这样用户空间就可以捕获这些事件来自动完成某些操作，如自动加载驱动、自动创建和删除设备节点、修改权限、创建软链接、修改网络设备的名字等。目前实现这个功能的工具就是 udev(或 mdev)这是一个用户空间的应用程序，捕获来自内核空间发来的事件，然后根据其规则文件进行操作。udev的规则文件为/etc/udev/rules.d 目录下后缀为.rules 的文件。  
 udev 规则文件用#来注释，除此之外的就是一条一条的规则。每条规则至少包含一个键值对，键分为匹配和赋值两种类型。如果内核发来的事件匹配了规则中的所有匹配键的值，那么这条规则就可以得到应用，并且赋值键被赋予指定的值。一条规则包含了一个或多个键值对，这些键值对用逗号隔开，每个键由操作符规定一个操作，合法的操作符如下。  
 ==和!= :判等，用于匹配键。  
 =、+=和:= : 赋值，用于赋值键，=和:=的区别是前者允许用新值来覆盖原来的值后者则不允许。+=则是追加赋值。  
 常见的键如下。  
 ACTION:事件动作的名字，如add 表示添加  
 DEVPATH:事件设备的路径。  
 KERNEL:事件设备的名字。  
 NAME:节点或网络接口的名字  
 SUBSYSTEM:事件设备子系统  
 DRIVER:事件设备驱动的名字。  
 ENV{key}:设备的属性。  
 OWNER、GROUP、MODE:设备节点的权限。  
 RUN:添加一个和设备相关的命令到一个命令列表中。  
  
 IMPORT{type):导入一组设备属性的变量，依赖于类型 type。  
  
 上面的键有的是匹配键，有的是赋值键，还有的既是匹配键又是赋值键。另外，还有很多其他的键，在此不一一罗列，详细信息请参见 udev 的 man 手册。

值还可以使用?、\*和来[]进行通配，这和正则表达式中的含义是一样的。接下来来看一个例子。

它表示当向 SCSI子系统添加任意设备后都要添加一个命令“/sbin/modprobe sg”到命令列表中，这个命令就是为相应的设备加载 sg 驱动模块。  
 在 Ubuntu 中自动加载驱动的规则如下，请将这条规则添加到/etc/udev/rules.d/40-modprobe.rules 文件中，如果没有这个文件请新建一个。

它表示根据模块的别名信息，用 modprobe 命令加载对应的内核模块。为此，我们要给平台驱动一个别名，如 pltdrv.c 文件中代码的第 49 行。pdev 要和驱动中用于匹配平台设备的名字保持一致。

添加了这一条规则后，加载 pltdev 模块就可以自动加载平台 pltdrv 驱动

#IsmodI grep plt

# modprobe pltdev

#lsmod I grep plt  
pltdrvpltdev

但是我的ubuntu的modprobe不能用，并且开发板还不支持udev所以后面再说。

六、使用平台设备的LED 驱动

前面我们说过，之前的驱动最大的问题就是没有把设备和驱动分离开，这使得驱动的通用性很差。只要硬件有任何改动(比如换一个管脚，增加或删除 LED 灯)，都会导致驱动代码的修改。有了 Linux 设备模型以及平台总线后，我们可以把设备的信息用平台设备来实现，这就大大提高了驱动的通用性。接下来的任务就是把前面的 LED 驱动改造成基于平台总线的设备和驱动。首先是平台设备，代码如下

由上可知，我们分别定义了 4 个平台设备，每一个平台设备代表一个LED 灯，之所以要这样做，是因为可以任意增加或删除一个 LED 灯。4 个平台设备都有一个IORESOURCE\_MEM资源，用来描述2个寄存器所占用的内存空间;名字都为 fsled，用来和平台驱动匹配;id 分别为 2、3、4、5，用来区别不同的设备。还给每个平台设备的platform\_data成员赋了值，platform\_data 的类型是 void\*，用来向驱动传递更多的信息，在这里传递的是每个LED 灯使用的管脚号，因为只有I/O内存是不能够控制一个具体的管脚的。这些平台设备放在 fsled\_devices 数组中，在模块初始化函数中使用platform\_add\_devices 一次注册到平台总线上。在模块的清除函数中，则使用 platform\_device\_unregister 来注销。  
 再来看看平台驱动。

fsled.c

fsled.h

代码第 152 行至第 159 行定义了一个平台驱动 fsled\_drv，名字叫 fsled，和平台设备匹配。代码第 161 行是平台驱动注册和注销的简化宏。  
 在 fsled\_probe 函数中，代码第 86 行首先通过 platform\_data 取了管脚号。代码第88 行以平台设备中的 id 为次设备号。代码第 93 行动态分配了 struct fsled\_dev 结构对象代码第 105行使用 platform\_get\_resource 获取了I/O内存的资源，这样要操作 GPIO管脚的两个信息就都获得了，一个是管脚号，一个是 I/O 内存地址。代码第 122 行使用platform set\_drvdata 将动态分配得到的 fsled 保存到了平台设备中，便于之后的代码能从平台设备中获取 struct fsled\_dev 结构对象的地址，是经常会使用到的一种技巧，也是个驱动支持多个设备的关键。  
 函数 fsled\_remove 中使用了 platform\_get\_drvdata 得到了对应的 struct fsled\_dev 结构对象的地址，其他操作则是函数 fsled\_probe 的反操作。  
 函数fsled\_open 也使用了 container\_of宏得到了对应的struct fsled\_dev 结构对象的地址，并保存在 filp->private\_data 中，这也是我前面谈到的一个驱动支持多个设备的技巧。  
 函数 fsled\_ioctl 相比于以前则要简单一些，因为只控制一个对应的LED灯。  
 测试的应用代码则是分别打开了 4个 LED 设备文件，然后再分别控制，代码比较简单，这里就不再赘述。测试方法和前面基本一致，只是要创建 4 个设备文件，用到 4个不同的次设备号 2、3、4、5。

七、自动创建设备节点

前面谈到，内核中设备的添加、删除或修改都会向应用层发送热插拔事件，应用程序可以捕获这些事件来自动完成某些操作，如自动加载驱动、自动创建设备节点等。接下来以mdev为例，来说明如何自动创建设备节点。  
 mdev 创建设备节点有两种方法，一种是运行 mdev -s 命令，一种是实时捕获热插拔事件。mdev-s 命令通常在根文件系统挂载完成后运行一次，它将递归扫描/sys/block 目录和/sys/class 目录下的文件，根据文件的内容来调用 make device 自动创建设备文件，这在busybox中的 mdev 源码中展现得非常清楚。

另外一种情况则是当内核发生了热插拔事件后，mdev会自动被调用，这体现在根文件系统中的/etc/init.d/reS 初始化脚本文件中。

内核有一种在发生热插拔事件后调用应用程序的方式，那就是执行/proc/sys/kernel/hotplug 文件中的程序，因为这种方式比较简单，所以常用在嵌入式系统之中。而之前说的udev 使用的则是 netlink 机制。发生热插拔事件时，调用 mdev 程序会将热插拔信息放在环境变量和参数当中，mdev 程序利用这些信息就可以自动创建设备节点，在 mdev 的源码中也有清晰的体现。

上面的代码的总体思路是根据 ACTION 键的值来决定 op 是增加还是移除操作,最终调用 make\_device 来自动创建或删除设备节点。  
  
 了解了应用层自动创建设备节点的方式后，接下来就需要讨论在驱动中如何实现了.既然自动设备节点的创建要依靠热插拔事件和 sysfs 文件系统，那这和我们之前讨论的kobjet 就是分不开的，mdev 扫描/sys/class 目录暗示我们要创建类，并且在类下面应该有具体的设备。为此，内核提供了相应的 API。

class\_create: 创建类，owner 是所属的模块对象指针，name 是类的名字，返回 struct

class 对象指针，返回值通过IS\_ERR 宏来判断是否失败,通过 PTR\_ERR 宏来获得错误码。  
 class\_destroy:销毁cls 类。  
 device\_create:在类class 下创建设备，parent 是父设备，没有则为 NULL。devt 是设备的主次设备号，drvdata 是驱动数据，没有则为 NULL。fmt 是格式化字符串，使用方法类似于printk。  
 device\_destroy:销毁 class 类下面主次设备号为 devt 的设备。返回值的检查方式同class\_create。  
 添加了自动创建设备的驱动的主要代码如下

代码第177行使用class\_create 创建了名叫 fsled的类。代码第127行使用device\_create在 fsled 类下面创建了 led%d 的设备，%d 用平台设备的id 来替代。在创建过程中，内核会发送热插拔事件给 mdev，mdev 利用这些信息就可以创建设备节点，因为设备的名字和设备号都传递给了 device\_create，而内核又会利用这些参数生成热插拔信息。  
  
 使用上面的驱动且驱动加载成功后，设备节点就自动被创建了，不需要再手动创建.整个测试过程和前面的例子类似，这里就不再重复了

测试程序是个流水灯不多说了。

这周感觉更累了，已经没有太多的精力继续学习驱动了，但是周末我们软件部门聚餐时听着那些四五十的大佬说着人生经历讲着做人道理时我幡然醒悟。一时的强弱代表不了什么态度才是最重要的。只要你有终身学习的态度不会成为不了强者的，在公司做工作也是一种学习，即便有时做起来不是很愉快，但往往向着难受的方向学习，才能发展的更加全面。现在来公司已经三周了，我刚刚敢叫大家的名字，因为大家都是英文花名，我是农村的从小英语就不好，一直不敢张口叫大家的名字，只能尴尬的这个同学那个同学，后来发现没人在意你的，大概意思对就行了。  
  
 以前在学校听说能画四层板的就能找到工作能画六层板就是大佬，偶然和硬件部的大佬交流了一下，喵的我们公司底板28层，我丢啊，玩个锤子。晚上吃饭的时候，听着大家都是各大名牌大学的研究生。其中不乏浙大这种顶尖学府的人才。顿时一个二本还没毕业的大三学生的我自卑感油然而生。突然萌生了考研的念头。可是真的很讨厌英语。如果以后有机会一定要试试另一种方式读研。去国外读研只需要一封介绍信。但是需要你是某个领域的顶尖人才。我们公司有个华三挖来的大佬。什么都会，整个FPGA都是他做的，这么多年问题不超过7个，或许大家绝对7个很多，但是真正工作时你就知道这个含金量了，我们软件的问题单貌似有70多个还是10多个人哦。重要的是他还精通python、通信、uboot。其它的我就不了解了，一个人可以全方位都牛，成为一个架构师我想这是每个专心搞技术的人的最高成就了吧。  
  
 我们组有个很阳光的大哥，人巨好，我们老大太严厉了有时我都不敢问，但是这个邻家哥哥一样的同事每次问他东西都给我耐心解决。这周刚好赶上版本验证，第一次经历这种大规模代码发布新版本前的验证工作。挺有意思的，因为我是实习生可能活少吧，大半天就完事了，他们几个忙的都见不到人了，过了巨无聊的一天，第二天他们还在验证，我就自己去解决一些新的问题单。其实看着那些署了自己名字的代码进入公司的版本成为其中的一部分，还是很有成就感的，以后可以吹牛说你用的手机芯片是我写的程序验证出来的啦哈哈。我的老大之前有点不理解他，后来吃饭才知道他原来已经结婚了，怪不得赚那么多，还每天那么节俭，在领导面前被迫改变自己的性格。结婚后要学会的第一件事或许就是责任吧。  
  
 我还发现了一种名叫领导力和人格魅力的东西。曾经在学校时我在一个大我两届的学长身上见到过。如今在这家公司我又见到了。一个长者，大概是公司的二号人物。真的很有人格魅力。有种即便做错了也想跟着他的感觉。  
  
 吃完饭去唱歌时玩游戏输了，不得不喝了两口酒，我根本就不能喝酒，吃饭时领导敬酒我都是喝水的，有点不好意思哈哈。但是没办法输了就要接受惩罚。有个也是东北的姐姐问到我为什么会来杭州。我楞了一下，回了两个字，舔狗。是啊，北京一个月实习期能拿一万多的不去，深圳一个月5000供吃住，常州4500管住都没去，来了杭州一个月也就4000块还不管吃住。最后呢，人家还不是很愿意搭理你，不是舔狗又是什么呢。  
  
 她并不是很惊艳的那种美丽，也没有很好的身材。但是就有一种奇特的魔力吸引我，或许是种别样的人格魅力吧。智慧型吸引？又或许是因为曾在我最需要帮助时帮了我。呼，累。但是说出来舒服了很多。喵的在CSDN写日常不会被和谐吧哈哈。后面技术更扎实了解决问题速度提高后就开始写写小说，现在已经看完两千多本了，感觉能看下去的很少了，所以以后自己写哈哈。就到这里咯，以后尽量保证每周更新一篇博客，学习些新东西，顺便记录一下一周的故事，就当是日记了。以后也有可以翻一翻的东西。

————————————————  
版权声明：本文为CSDN博主「宇努力学习」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/qq_52479948/article/details/131949898>

## 2. Linux 内核模块的加载卸载

insmod

rmmod

modper

depmod

## 3. Linux 字符设备驱动模板

一、字符设备驱动简介

字符设备是 Linux 驱动中最基本的一类设备驱动，字符设备就是一个一个字节，按照字节流进行读写操作的设备，读写数据是分先后顺序的。比如我们最常见的点灯、按键、 IIC、 SPI，LCD 等等都是字符设备，这些设备的驱动就叫做字符设备驱动。  
  
 在详细的学习字符设备驱动架构之前，我们先来简单的了解一下 Linux 下的应用程序是如何调用驱动程序的， Linux 应用程序对驱动程序的调用流程如下图所示：  
  
 在 Linux 中一切皆为文件，驱动加载成功以后会在“/dev”目录下生成一个相应的文件，应用程序通过对这个名为“/dev/xxx” (xxx 是具体的驱动文件名字)的文件进行相应的操作即可实现对硬件的操作。  
  
 比如现在有个叫做/dev/led 的驱动文件，此文件是 led 灯的驱动文件。应用程序使用 open 函数来打开文件/dev/led，使用完成以后使用 close 函数关闭/dev/led 这个文件。 open和 close 就是打开和关闭 led 驱动的函数，如果要点亮或关闭 led，那么就使用 write 函数来操作，也就是向此驱动写入数据，这个数据就是要关闭还是要打开 led 的控制参数。如果要获取led 灯的状态，就用 read 函数从驱动中读取相应的状态。  
  
 应用程序运行在用户空间，而 Linux 驱动属于内核的一部分，因此驱动运行于内核空间。当我们在用户空间想要实现对内核的操作，比如使用 open 函数打开/dev/led 这个驱动，因为用户空间不能直接对内核进行操作，因此必须使用一个叫做“系统调用”的方法来实现从用户空间“陷入” 到内核空间，这样才能实现对底层驱动的操作。  
  
 open、 close、 write 和 read 等这些函数是由 C 库提供的，在 Linux 系统中，系统调用作为 C 库的一部分。当我们调用 open 函数的时候流程如下图所示：  
  
 其中关于 C 库以及如何通过系统调用“陷入” 到内核空间这个我们不用去管，我们重点关注的是应用程序和具体的驱动，应用程序使用到的函数在具体驱动程序中都有与之对应的函数，比如应用程序中调用了 open 这个函数，那么在驱动程序中也得有一个名为 open 的函数。每一个系统调用，在驱动中都有与之对应的一个驱动函数，在 Linux 内核文件 include/linux/fs.h 中有个叫做 file\_operations 的结构体，此结构体就是 Linux 内核驱动操作函数集合，内容如下所示：  
  
/\* file\_operations 结构体 \*/  
struct file\_operations {  
 struct module \*owner;  
 loff\_t (\*llseek) (struct file \*, loff\_t, int);  
 ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);  
 ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);  
 ssize\_t (\*read\_iter) (struct kiocb \*, struct iov\_iter \*);  
 ssize\_t (\*write\_iter) (struct kiocb \*, struct iov\_iter \*);  
 int (\*iopoll)(struct kiocb \*kiocb, struct io\_comp\_batch \*,  
 unsigned int flags);  
 int (\*iterate) (struct file \*, struct dir\_context \*);  
 int (\*iterate\_shared) (struct file \*, struct dir\_context \*);  
 \_\_poll\_t (\*poll) (struct file \*, struct poll\_table\_struct \*);  
 long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);  
 long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long);  
 int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  
 unsigned long mmap\_supported\_flags;  
 int (\*open) (struct inode \*, struct file \*);  
 int (\*flush) (struct file \*, fl\_owner\_t id);  
 int (\*release) (struct inode \*, struct file \*);  
 int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync);  
 int (\*fasync) (int, struct file \*, int);  
 int (\*lock) (struct file \*, int, struct file\_lock \*);  
 ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  
 unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);  
 int (\*check\_flags)(int);  
 int (\*flock) (struct file \*, int, struct file\_lock \*);  
 ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);  
 ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);  
 int (\*setlease)(struct file \*, long, struct file\_lock \*\*, void \*\*);  
 long (\*fallocate)(struct file \*file, int mode, loff\_t offset,  
 loff\_t len);  
 void (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f);  
#ifndef CONFIG\_MMU  
 unsigned (\*mmap\_capabilities)(struct file \*);  
#endif  
 ssize\_t (\*copy\_file\_range)(struct file \*, loff\_t, struct file \*,  
 loff\_t, size\_t, unsigned int);  
 loff\_t (\*remap\_file\_range)(struct file \*file\_in, loff\_t pos\_in,  
 struct file \*file\_out, loff\_t pos\_out,  
 loff\_t len, unsigned int remap\_flags);  
 int (\*fadvise)(struct file \*, loff\_t, loff\_t, int);  
 int (\*uring\_cmd)(struct io\_uring\_cmd \*ioucmd, unsigned int issue\_flags);  
 int (\*uring\_cmd\_iopoll)(struct io\_uring\_cmd \*, struct io\_comp\_batch \*,  
 unsigned int poll\_flags);  
}  
  
 在字符设备驱动开发中最常用的就是上面这些函数，关于其他的函数大家可以查阅相关文档。我们在字符设备驱动开发中最主要的工作就是实现上面这些函数，不一定全部都要实现，但是像 open、 release、 write、 read 等都是需要实现的，当然了，具体需要实现哪些函数还是要看具体的驱动要求。

二、字符设备驱动开发步骤

下面是开发字符设备驱动的基本步骤：  
1.确定设备号

如果使用静态分配设备号，可以在代码中指定一个固定的设备号。  
  
如果使用动态分配设备号，可以调用 'alloc\_chrdev\_region' 函数在模块初始化时分配设备号。  
  
如果使用 'udev' 等工具进行设备号的动态分配和管理，则需要在驱动程序中声明一个 'dev\_t' 类型的变量，并使用 'MKDEV' 宏将主设备号和次设备号合并为一个设备号。

2.定义 file\_operations 结构体

创建一个结构体，用于定义字符设备驱动程序对外提供的操作接口。常见的函数包括 'open'、'release'、'read'、'write'、'ioctl' 等。  
  
在驱动程序的初始化函数中，将这些操作函数与对应的函数指针关联起来。

3.实现操作函数

实现字符设备驱动中定义的操作函数，根据设备的需求来进行相应的操作。  
  
在 'open' 函数中可以进行设备的初始化工作，例如分配内存、初始化设备状态等。  
  
在 'release' 函数中可以进行设备的清理工作，例如释放内存、关闭设备等。  
  
在 'read' 函数中可以从设备读取数据，并将数据传递给用户空间。  
  
在 'write' 函数中可以接收用户空间传递的数据，并将数据写入设备。  
  
在 'ioctl' 函数中可以处理设备的特殊控制命令。

4.注册和注销字符设备

在驱动程序的初始化函数中，调用 'alloc\_chrdev\_region' 函数分配设备号。  
  
使用 'cdev\_init' 初始化 'cdev' 结构体，并将 file\_operations 结构体指针赋值给 'cdev' 的成员。  
  
使用 'cdev\_add' 将设备添加到内核中，使其可用。  
  
在驱动程序的退出函数中，使用 'cdev\_del' 和 'unregister\_chrdev\_region' 函数注销设备。

5.编译和加载模块

将驱动程序的源代码编译为内核模块，生成对应的 .ko 文件。  
  
使用 'insmod' 命令加载模块到内核中。

6.用户空间交互：

用户程序可以通过系统调用来访问字符设备，例如使用 'open'、'read'、'write'、'ioctl' 等函数来打开、读取和写入设备。  
  
用户程序可以使用文件描述符来标识打开的设备，通过文件描述符进行读写操作。  
  
 以上是字符设备驱动开发的基本步骤。在实际开发中，还需要进行错误处理、设备管理、内存分配和释放等工作，具体的实现细节会根据设备的特性和需求而有所不同。为了开发出高质量的驱动程序，建议仔细阅读相关的文档、示例代码和最佳实践，并进行充分的测试和验证。

三、字符设备驱动示例模板

字符驱动框架是一种在Linux内核中实现设备驱动程序的方法。它允许开发者编写基于字符设备的驱动程序，以便与用户空间中的字符设备进行通信。下面是一个简单的字符驱动框架模板，包括了必要的函数和数据结构。  
  
#include <linux/module.h>  
#include <linux/fs.h>  
#include <linux/cdev.h>  
#include <linux/uaccess.h>  
   
#define DEVICE\_NAME "mydevice"  
#define BUF\_SIZE 1024  
   
static dev\_t dev;  
static struct cdev cdev;  
static char buffer[BUF\_SIZE];  
static int buffer\_len = 0;  
   
static int device\_open(struct inode \*inode, struct file \*filp)  
{  
 // 设备打开时的操作  
 return 0;  
}  
   
static int device\_release(struct inode \*inode, struct file \*filp)  
{  
 // 设备关闭时的操作  
 return 0;  
}  
   
static ssize\_t device\_read(struct file \*filp, char \*user\_buf, size\_t count, loff\_t \*f\_pos)  
{  
 // 从设备读取数据  
 size\_t to\_copy = min(count, (size\_t)buffer\_len);  
 if (copy\_to\_user(user\_buf, buffer, to\_copy) != 0)  
 return -EFAULT;  
   
 return to\_copy;  
}  
   
static ssize\_t device\_write(struct file \*filp, const char \*user\_buf, size\_t count, loff\_t \*f\_pos)  
{  
 // 向设备写入数据  
 size\_t to\_copy = min(count, (size\_t)BUF\_SIZE);  
 if (copy\_from\_user(buffer, user\_buf, to\_copy) != 0)  
 return -EFAULT;  
   
 buffer\_len = to\_copy;  
 return to\_copy;  
}  
   
static struct file\_operations fops = {  
 .owner = THIS\_MODULE,  
 .open = device\_open,  
 .release = device\_release,  
 .read = device\_read,  
 .write = device\_write,  
};  
   
static int \_\_init chardev\_init(void)  
{  
 // 模块初始化函数  
 if (alloc\_chrdev\_region(&dev, 0, 1, DEVICE\_NAME) < 0)  
 {  
 printk(KERN\_ALERT "Failed to allocate character device region\n");  
 return -1;  
 }  
   
 cdev\_init(&cdev, &fops);  
 if (cdev\_add(&cdev, dev, 1) < 0)  
 {  
 unregister\_chrdev\_region(dev, 1);  
 printk(KERN\_ALERT "Failed to add character device\n");  
 return -1;  
 }  
   
 printk(KERN\_INFO "Character device registered: %s\n", DEVICE\_NAME);  
 return 0;  
}  
   
static void \_\_exit chardev\_exit(void)  
{  
 // 模块退出函数  
 cdev\_del(&cdev);  
 unregister\_chrdev\_region(dev, 1);  
 printk(KERN\_INFO "Character device unregistered\n");  
}  
   
module\_init(chardev\_init);  
module\_exit(chardev\_exit);  
   
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("Your Name");  
MODULE\_DESCRIPTION("Character Device Driver");  
  
 这个模板定义了一个名为 'mydevice' 的字符设备驱动。驱动程序会将用户空间的数据写入到 'buffer' 中，并从 'buffer' 中读取数据返回给用户空间。其中，'device\_open' 和 'device\_release' 函数在设备打开和关闭时被调用，'device\_read' 和 'device\_write' 函数用于读取和写入设备数据。  
  
 该模板使用 'alloc\_chrdev\_region' 函数为字符设备分配主次设备号，并使用 'cdev\_init' 和 'cdev\_add' 函数将设备添加到系统中。在模块退出时，使用 'cdev\_del' 和 'unregister\_chrdev\_region' 函数来注销设备。  
  
 请注意，这只是一个简单的字符驱动框架模板，仅用于演示目的。实际的字符驱动可能需要更多的功能和错误处理。在开发字符驱动程序时，请仔细参考Linux内核文档和示例代码，以确保正确性和稳定性。

四、字符设备驱动开发总结

字符驱动是一种在操作系统内核中实现的设备驱动程序，用于与字符设备进行交互。字符设备是一种以字节为单位进行输入和输出的设备，例如终端、串口、打印机等。字符驱动框架提供了一组函数和数据结构，使得开发者可以编写自定义的字符设备驱动程序。

在Linux内核中，字符驱动的实现基于以下几个核心组件：

1.设备号：每个字符设备驱动在注册时都会被分配一个唯一的设备号。设备号包括主设备号和次设备号。主设备号标识设备驱动程序，次设备号标识具体的设备实例。

2.file\_operations 结构体：这是一个函数指针结构体，定义了设备驱动程序对外提供的操作接口。常见的函数包括 'open'、'release'、'read'、'write'、'ioctl' 等。开发者需要实现这些函数来处理设备的打开、关闭、读取和写入等操作。

3.cdev 结构体：'cdev' 是字符设备驱动的核心结构体，它代表一个字符设备实例。它包含了对应的 file\_operations 结构体指针以及设备号等信息。通过使用 'cdev' 结构体，开发者可以注册和管理字符设备。

4.字符设备注册和注销：在字符驱动的初始化阶段，需要使用 'alloc\_chrdev\_region' 函数为驱动程序分配设备号。然后使用 'cdev\_init' 初始化 'cdev' 结构体，并使用 'cdev\_add' 将设备添加到系统中。在驱动程序退出时，需要使用 'cdev\_del' 和 'unregister\_chrdev\_region' 函数来注销设备。

5.用户空间交互：字符驱动允许用户空间程序通过系统调用来访问设备。用户程序可以打开设备、读取和写入设备数据，并通过 'ioctl' 等方式发送控制命令。  
————————————————  
版权声明：本文为CSDN博主「Sunshine-Linux」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/Wang_XB_3434/article/details/131739858>

## 4. Linux 驱动中的信号量与自旋锁

文章目录

并发解决方法二（自旋锁）  
spin\_lock  
spin\_lock的变体  
自旋锁的使用  
读写自旋锁

设计自旋锁的最初目的是在多处理器系统中提供对共享数据的保护，其背后的核心思想是：设置一个在多处理器之间共享的全局变量锁v,并定义当V=I时为上锁状态，V=0为解锁状态·如果处理器A上的代码要进入临界区．它要先读取V的值，判断其是否为0，如果V不等于0表明有其他处理器上的代码正在对共享数据进行访问，此时处理器A进入忙等待即自旋状态，如果V=O表明当前没有其他处理器上的代码进入临界区，此时处理器A可以访问该资源，它先把V置1（自旋锁的上锁状态).然后进入临界区，访问完毕离开临界区时将V置0（自旋锁的解锁状态）。

上述自旋锁的设计思想在用具体代码实现时的关键之处在于，必须确保处理器A“读取v，判断v的值与更新”这一操作序列是个原子操作(atomic operation)•所谓原子操作，简单地说就是执行这个操作的指令序列在处理器上执行时等同于单条指令，也即该指令序列在执行时是不可分割的。  
spin\_lock

不同的处理器上有不同的指令用以实现上述的原子操作，所以spin\_lock的相关代码在不同体系架构上有不同的实现，为了帮助读者对spin\_lock这一机制建立具体的印象·下面以ARM处理器上的实现为例，仔细考察spin\_lock的幕后行为·下面的讨论先以多处理器为主·然后再讨论spin\_lock及其变体在单处理器上的演进。

在给出实际源码细节之前，先做个简短的说明，为了让读者更清楚地理解这里的代码，下面会对代码进行轻微调整，使之外在的表现形式更加紧凑而又不影响其内涵，同时也不会关注一些调试相关的数据成员，所以在摘录的代码中己将其移除。

下面是Linux源码中提供给设备驱动程序等内核模块使用的spin\_lock接口函数的定义：

static inline void spin\_lock(spinlock\_t \*lock)  
{  
 raw\_spin\_lock(&lock->rlock);  
}

1  
2  
3  
4

代码中的数据结构spinlock\_t，就是前面提到的在多处理器之间共享的自旋锁在现实源码中的具体表现，透过层层的定义，会发现实际上它就是个volatile unsigned int型变量：

typedef struct spinlock {  
 union {  
 struct raw\_spinlock rlock;

#ifdef CONFIG\_DEBUG\_LOCK\_ALLOC

# define LOCK\_PADSIZE (offsetof(struct raw\_spinlock, dep\_map))

struct {  
 u8 \_\_padding[LOCK\_PADSIZE];  
 struct lockdep\_map dep\_map;  
 };

#endif  
 };  
} spinlock\_t;

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13

typedef struct raw\_spinlock {  
 arch\_spinlock\_t raw\_lock;  
#ifdef CONFIG\_GENERIC\_LOCKBREAK  
 unsigned int break\_lock;  
#endif  
#ifdef CONFIG\_DEBUG\_SPINLOCK  
 unsigned int magic, owner\_cpu;  
 void \*owner;  
#endif  
#ifdef CONFIG\_DEBUG\_LOCK\_ALLOC  
 struct lockdep\_map dep\_map;  
#endif  
} raw\_spinlock\_t;

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13

typedef struct {  
 volatile unsigned int slock;  
} arch\_spinlock\_t;

1  
2  
3

spin\_lock函数中调用的raw\_spin\_lock是个宏·其实现是处理器相关的，对于ARM处理器而言，最终展开为

static inline void *\_raw\_spin\_lock(raw\_spinlock\_t \*lock)*  
*{*  
 *preempt\_disable();*  
 *spin\_acquire(&lock->dep\_map, 0, 0, \_RET\_IP*);  
 LOCK\_CONTENDED(lock, do\_raw\_spin\_trylock, do\_raw\_spin\_lock);  
}

1  
2  
3  
4  
5  
6

函数首先调用preempt\_disable宏，后者在定义了CONFIG\_PREEMPT,也即在支持内核可抢占的调度系统中时，将关闭调度器的可抢占特性。在没有定义CONFIG\_PREEMPT时，preempt\_disable是个空定义，什么也不做。

真正的上锁操作发生在后面的do\_raw\_spin\_lock函数中，不过在讨论该函数的实现细节前，先来看看为什么raw\_spin\_lock要先调用preempt\_disable来关闭系统的可抢占性·在一个打开了CONFIG\_PREEMPI•特性的Linux系统中，一个在内核态执行的路径也有可能被切换出处理器，典型地，比如当前进程正在内核态执行某一系统调用时，发生了一个外部中断·当中断处理函数返回时，因为内核的可抢占性，此时将会出现一个调度点，如果CPU的运行队列中出现了一个比当前被中断进程优先级更高的进程，那么被中断的进程将会被换出处理器，即便此时它正运行在内核态·单处理器上的这种因为内核的可抢占性所导致的两个不同进程并发执行的情形，非常类似于SMP系统上运行在不同处理器上的进程之间的并发，因此为了保护共享的资源不会受到破坏·必須在进入临界区前关闭内核的可抢占性·因为Linux内核源码试图统一自旋锁的接口代码，即不论是单处理器还是多处理器，不论内核是否配置了可抢占特性，提供给外部模块使用的相关自旋锁代码都只有一份，所以可以看到在上述的raw\_spin\_lock函数中加入了内核可抢占性相关的代码，即便是在没有配置内核可抢占的系统上．外部模块也都统一使用相同的spin\_lock和spin\_unlock接口函数。

函数接着调用do\_raw\_spin\_lock开始真正的上锁操作

void do\_raw\_spin\_lock(raw\_spinlock\_t \*lock)  
{  
 debug\_spin\_lock\_before(lock);  
 if (unlikely(!arch\_spin\_trylock(&lock->raw\_lock)))  
 \_\_spin\_lock\_debug(lock);  
 debug\_spin\_lock\_after(lock);  
}

1  
2  
3  
4  
5  
6  
7

与spin\_lock相对的是spin\_unlock，这是一个应该在离开临界区时调用的函数，用来释放此前获得的自旋锁·其外部接口定义如下

static inline void spin\_unlock(spinlock\_t \*lock)  
{  
 raw\_spin\_unlock(&lock->rlock);  
}

1  
2  
3  
4  
5

最终调用

#define **UNLOCK(lock) \**  
 **do { preempt\_enable(); \_**UNLOCK(lock); } while (0)

1  
2

函数先调用面raw\_spin\_unlock做实际的解锁操作，然后调用preempt\_enable函数打开内核可抢占性，对于没有定义CONFIG\_PREEMPT的系统，该宏是个空定义·do\_raw\_spin\_unlock函数在ARM处理器上的代码如下：

void do\_raw\_spin\_unlock(raw\_spinlock\_t \*lock)  
{  
 debug\_spin\_unlock(lock);  
 arch\_spin\_unlock(&lock->raw\_lock);  
}

1  
2  
3  
4  
5  
6

解锁操作比获得锁的操作要相对简单，只需史新锁变量为0即可，在ARM平台上利用单条指令嘶就可以完成该任务，所以代码非常简单，熹接用指令将自旋锁的状态更新为0，即解锁状态·针对spin\_lock应该调用spin\_unlock而不是其他形式的释放锁函数，驱动程序员必须确保这种获得锁和释放锁函数调用的一致性．  
spin\_lock的变体

在前面讨论spin\_lock函数时，spin\_lock对多处理器系统中这种进程间真正的并发执行引起的竞态问題解决得很好，但是考虑图1所示这样一个场景：  
在这里插入图片描述  
处理器上的当前进程A因为要对某一全局性的链表g-list进行操作，所以在操作前通过调用spin\_lock来进入临界区（图中标号1所示),当它正处于临界区中时，进程A所在的处理器上发生了一个外部硬件中断，此时系统必须暫停当前进程A的执行转而去处理该中断（图中标号2所示假设该中断的处理例程中恰好也要操作g-list.因为这是一个共享的全局变量，所以在操作之前也要调用spinlock函数来对该共享变量进行保护（图中标号3所示），当中断处理例程中的spin-lock试图去获得自旋锁slock时，因为被它中断的进程A之前己经获得该锁，于是将导致中断处理例程进入自旋状态。**在中断处理例捍中出现一个自旋状态是非常致命的，因为中断处例程必须在尽可能短的时间内返回，而此时它却必须自旋。此时被它中断的进程A因中断处理函数不能返回而无行，也就不可能释放锁，所以将导致中断处理例程中的spin\_lock一直下去，导致死锁。**出现这种特定情况的本质原因在于对锁的竞争发生在不能真正并发执行的两条路径上，如果可以并发执行，那么在上面的案例中，被中断的进程依然可以继续执行继而释放锁·对这种问题的解决导致了spin\_lock函数其他变体的出现。

因处理外部的中断而引发spin\_lock缺陷的例子，使得必须在这种情况下对spin\_lock予以修正，于是出现了spin\_lock\_irq和spin\_lock\_irq\_save函数·spin\_lock\_irq函数接口定义如下：

static inline void spin\_lock\_irq(spinlock\_t \*lock)  
{  
 raw\_spin\_lock\_irq(&lock->rlock);  
}

1  
2  
3  
4

最终调用

#define **LOCK\_IRQ(lock) \**  
 **do { local\_irq\_disable();** LOCK(lock); } while (0)

1  
2

其中的raw\_spin\_lock\_irq函数的实现，相对于raw\_spin\_lock只是在调用preempt\_disable之前又调用了local\_irq\_disable,后者在本章前面部分己经讨论过，用来关闭本地处理器响应外部中断的能力，这样在获取一个锁时就可以确保不会发生中断，从而避免上面提到的死锁问題。local\_irq\_disable只能用来关闭本地处理器的中断，当一个通过调用spin\_lock\_irq拥有自旋锁V的进程在处理器A上执行时，虽然在处理器A上中断被关闭了，但是外部中断依然有机会发送到处理器B上，如果处理器B上的中断处理函数也试图去获得锁v,情况会怎样呢？因为此时处理器A上的进程可以继续执行，在它离开临界区时将释放锁，这样处理器B上的中断处理函数就可以结束此前的自旋状态·这从一个侧面说明通过自旋锁进入的临界区代码必须在尽可能短的时间内执行完毕，因为它执行的时间越长·别的处理器就越需要自旋以等待史长的时间（尤其是这种自旋发生在中断处理函数中).最糟糕的情况是进程在临界区中因为某种原因被换出处理器·所以作为使用自旋锁时一条确定的规则，任何拥有自旋锁的代码都必须是原子的，不能休眠。在实际的使用中，这条规则实践起来还远不像规则描述的那样直白，调用者需要仔细审视在拥有锁时的每个函数调用，因为睡眠有可能发生在这函数的内部，比如以GFP\_KERNEL作为分配掩码通过kmalloc函数来分配一块内存时．系统中仝闲的内存不足以满足本次分配的情形虽然非常少见，但是毕竟存在这种可能性，一旦这种可能性被确定，kmalloc会阻塞从而会被切换出处理器，如果kmalloc的调用者在此之前拥有某个自旋锁·那么这种情形下将对系统的稳定性造成极大的威胁。

如此，当知道一个自旋锁在中断处理的上下文中有可能会被使用到时·应该使用  
spin\_lock\_irq函数，而不是spin\_lock•后者只有在能确定中断上下文中不会使用到自旋锁  
的情形下才能使用。spin\_lock\_irq对应的释放锁函数为spin\_unlock\_irq，其接口定义为

static inline void spin\_unlock\_irq(spinlock\_t \*lock)  
{  
 raw\_spin\_unlock\_irq(&lock->rlock);  
}

1  
2  
3  
4  
5

#define **UNLOCK\_IRQ(lock) \**  
 **do { local\_irq\_enable();** UNLOCK(lock); } while (0)

1  
2

可见，在raw\_spin\_unlock\_irq函数中除了调用\_\_UNLOCK做实际的解锁操作外，还会打开本地处理器上的中断，以及开启内核的可抢占性。

与spin\_lock\_irq类似的还有一个spin\_lock\_save宏，它与spin\_lock\_irq函数最大的区别  
是，在关闭中断前会将处理器当前的FLAGS寄存器的值保存在一个变量中，当调用对应  
的spin\_unlock\_trqrestore来释放锁时·会将spin\_lock\_irq\_save中保存的FLAGS值重新写回  
到寄存器中·对于spin\_lock\_irq\_save和spin\_unlock\_irq\_restore的使用场合，可参考前面关于local\_irq\_save和local\_irq\_estore的讨论。  
自旋锁的使用

自旋锁（Spin Lock） 是一种典型的对临界资源进行互斥访问的手段， 其名称来源于它的工作方式。为了获得一个自旋锁， 在某CPU上运行的代码需先执行一个原子操作， 该操作测试并设置（Test-AndSet） 某个内存变量。 由于它是原子操作， 所以在该操作完成之前其他执行单元不可能访问这个内存变量。 如果测试结果表明锁已经空闲， 则程序获得这个自旋锁并继续执行； 如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作， 即进行所谓的“自旋”， 通俗地说就是“在原地打转”， 如图7.7所示。 当自旋锁的持有者通过重置该变量释放这个自旋锁后， 某个等待的“测试并设置”操作  
向其调用者报告锁已释放。  
在这里插入图片描述

理解自旋锁最简单的方法是把它作为一个变量看待， 该变量把一个临界区标记为“我当前在运行， 请稍等一会”或者标记为“我当前不在运行， 可以被使用”。 如果A执行单元首先进入例程， 它将持有自旋锁；当B执行单元试图进入同一个例程时， 将获知自旋锁已被持有， 需等到A执行单元释放后才能进入。

在ARM体系结构下， 自旋锁的实现借用了ldrex指令、 strex指令、 ARM处理器内存屏障指令dmb和dsb、 wfe指令和sev指令， 这类似于代码清单7.1的逻辑。 可以说既要保证排他性， 也要处理好内存屏障。

Linux中与自旋锁相关的操作主要有以下4种。

定义自旋锁

spinlock\_t lock;

1  
  
初始化自旋锁

spin\_lock\_init(lock);

1

该宏用于动态初始化自旋锁lock。

获得自旋锁

spin\_lock(lock);

1

该宏用于获得自旋锁lock， 如果能够立即获得锁， 它就马上返回， 否则， 它将在那里自旋， 直到该自旋锁的保持者释放。

spin\_trylock(lock);

1

该宏尝试获得自旋锁lock， 如果能立即获得锁， 它获得锁并返回true， 否则立即返回false， 实际上不再“在原地打转”。

释放自旋锁

spin\_unlock(lock);

1

该宏释放自旋锁lock， 它与spin\_trylock或spin\_lock配对使用。

自旋锁一般这样写

/\* 定义一个自旋锁*/*  
*spinlock\_t lock;*  
*spin\_lock\_init(&lock);*  
*spin\_lock (&lock) ; /* 获取自旋锁， 保护临界区 */*  
*. . ./* 临界区\*/  
spin\_unlock (&lock) ;

1  
2  
3  
4  
5  
6

自旋锁主要针对SMP或单CPU但内核可抢占的情况， 对于单CPU和内核不支持抢占的系统， 自旋锁退化为空操作。 在单CPU和内核可抢占的系统中， 自旋锁持有期间中内核的抢占将被禁止。 由于内核可抢占的单CPU系统的行为实际上很类似于SMP系统， 因此， 在这样的单CPU系统中使用自旋锁仍十分必要。 另外， 在多核SMP的情况下， 任何一个核拿到了自旋锁， 该核上的抢占调度也暂时禁止了， 但是没有禁止另外一个核的抢占调度。

尽管用了自旋锁可以保证临界区不受别的CPU和本CPU内的抢占进程打扰， 但是得到锁的代码路径在执行临界区的时候， 还可能受到中断和底半部（BH， 稍后的章节会介绍） 的影响。 为了防止这种影响，就需要用到自旋锁的衍生。 spin\_lock（） /spin\_unlock（） 是自旋锁机制的基础， 它们和关中断local\_irq\_disable（） /开中断local\_irq\_enable（） 、 关底半部local\_bh\_disable（） /开底半部local\_bh\_enable（） 、 关中断并保存状态字local\_irq\_save（） /开中断并恢复状态字local\_irq\_restore（） 结合就形成了整套自旋锁机制， 关系如下

spin\_lock\_irq() = spin\_lock() + local\_irq\_disable()  
spin\_unlock\_irq() = spin\_unlock() + local\_irq\_enable()  
spin\_lock\_irqsave() = spin\_lock() + local\_irq\_save()  
spin\_unlock\_irqrestore() = spin\_unlock() + local\_irq\_restore()  
spin\_lock\_bh() = spin\_lock() + local\_bh\_disable()spin\_unlock\_bh() = spin\_unlock() + local\_bh\_enable()

1  
2  
3  
4  
5

spin\_lock\_irq（ ） 、 spin\_lock\_irqsave（ ） 、 spin\_lock\_bh（ ） 类似函数会为自旋锁的使用系好“安全带”以避免突如其来的中断驶入对系统造成的伤害。

在多核编程的时候， 如果进程和中断可能访问同一片临界资源， 我们一般需要在进程上下文中调用spin\_lock\_irqsave（ ） /spin\_unlock\_irqrestore（ ） ， 在中断上下文中调用spin\_lock（ ） /spin\_unlock（ ） ， 如图7.8所示。 这样， 在CPU0上， 无论是进程上下文， 还是中断上下文获得了自旋锁， 此后， 如果CPU1无论是进程上下文， 还是中断上下文， 想获得同一自旋锁， 都必须忙等待， 这避免一切核间并发的可能性。同时， 由于每个核的进程上下文持有锁的时候用的是spin\_lock\_irqsave（ ） ， 所以该核上的中断是不可能进入的， 这避免了核内并发的可能性。

驱动工程师应谨慎使用自旋锁， 而且在使用中还要特别注意如下几个问题。

1） 自旋锁实际上是忙等锁， 当锁不可用时， CPU一直循环执行“测试并设置”该锁直到可用而取得该锁， CPU在等待自旋锁时不做任何有用的工作， 仅仅是等待。 因此， 只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。 当临界区很大， 或有共享设备的时候， 需要较长时间占用锁， 使用自旋锁会降低系统的性能。

2） 自旋锁可能导致系统死锁。 引发这个问题最常见的情况是递归使用一个自旋锁， 即如果一个已经拥有某个自旋锁的CPU想第二次获得这个自旋锁， 则该CPU将死锁。图7.8 自  
在这里插入图片描述

3） 在自旋锁锁定期间不能调用可能引起进程调度的函数。 如果进程获得自旋锁之后再阻塞， 如调用copy\_from\_user（ ） 、 copy\_to\_user（ ） 、 kmalloc（ ） 和msleep（ ） 等函数， 则可能导致内核的崩溃。

4） 在单核情况下编程的时候， 也应该认为自己的CPU是多核的， 驱动特别强调跨平台的概念。 比如， 在单CPU的情况下， 若中断和进程可能访问同一临界区， 进程里调用spin\_lock\_irqsave（ ） 是安全的， 在中断里其实不调用spin\_lock（ ） 也没有问题， 因为spin\_lock\_irqsave（ ） 可以保证这个CPU的中断服务程序不可能执行。 但是， 若CPU变成多核， spin\_lock\_irqsave（） 不能屏蔽另外一个核的中断， 所以另外一个核就可能造成并发问题。 因此， 无论如何， 我们在中断服务程序里也应该调用spin\_lock（） 。

使用自旋锁使设备只能被一个进程打开  
读写自旋锁

自旋锁不关心锁定的临界区究竟在进行什么操作， 不管是读还是写， 它都一视同仁。 即便多个执行单元同时读取临界资源也会被锁住。 实际上， 对共享资源并发访问时， 多个执行单元同时读取它是不会有问题的， 自旋锁的衍生锁读写自旋锁（rwlock） 可允许读的并发。 读写自旋锁是一种比自旋锁粒度更小的锁机制， 它保留了“自旋”的概念， 但是在写操作方面， 只能最多有1个写进程， 在读操作方面， 同时可以有多个读执行单元。 当然， 读和写也不能同时进行。

定义和初始化读写自旋锁

rwlock\_t my\_rwlock;  
rwlock\_init(&my\_rwlock); /\* 动态初始化 \*/

1  
2  
  
读锁定

void read\_lock(rwlock\_t \*lock);  
void read\_lock\_irqsave(rwlock\_t \*lock, unsigned long flags);  
void read\_lock\_irq(rwlock\_t \*lock);  
void read\_lock\_bh(rwlock\_t \*lock);

1  
2  
3  
4  
  
读解锁

void read\_unlock(rwlock\_t \*lock);  
void read\_unlock\_irqrestore(rwlock\_t \*lock, unsigned long flags);  
void read\_unlock\_irq(rwlock\_t \*lock);  
void read\_unlock\_bh(rwlock\_t \*lock);

1  
2  
3  
4

在对共享资源进行读取之前， 应该先调用读锁定函数， 完成之后应调用读解锁函数。

read\_lock\_irqsave（） 、 read\_lock\_irq（） 和read\_lock\_bh（） 也分别是read\_lock（） 分别与local\_irq\_save（） 、 local\_irq\_disable（） 和local\_bh\_disable（） 的组合， 读解锁函数read\_unlock\_irqrestore（） 、 read\_unlock\_irq（） 、 read\_unlock\_bh（） 的情况与此类似

写锁定

void write\_lock(rwlock\_t \*lock);  
void write\_lock\_irqsave(rwlock\_t \*lock, unsigned long flags);  
void write\_lock\_irq(rwlock\_t \*lock);  
void write\_lock\_bh(rwlock\_t \*lock);  
int write\_trylock(rwlock\_t \*lock);

1  
2  
3  
4  
5  
  
写解锁

void write\_unlock(rwlock\_t \*lock);  
void write\_unlock\_irqrestore(rwlock\_t \*lock, unsigned long flags);void write\_unlock\_irq(rwlock\_t \*lock);  
void write\_unlock\_bh(rwlock\_t \*lock);

1  
2  
3

write\_lock\_irqsave（ ） 、 write\_lock\_irq（ ） 、 write\_lock\_bh（ ） 分别是write\_lock（ ） 与local\_irq\_save（ ） 、 local\_irq\_disable（ ） 和local\_bh\_disable（ ） 的组合， 写解锁函数write\_unlock\_irqrestore（ ） 、 write\_unlock\_irq（ ） 、 write\_unlock\_bh（ ） 的情况与此类似。

在对共享资源进行写之前， 应该先调用写锁定函数， 完成之后应调用写解锁函数。 和spin\_trylock（ ）一样， write\_trylock（ ） 也只是尝试获取读写自旋锁， 不管成功失败， 都会立即返回。

读写自旋锁一般这样被使用

rwlock\_t lock; /\* 定义rwlock */*  
*rwlock\_init(&lock); /* 初始化rwlock */*  
*/* 读时获取锁*/*  
*read\_lock(&lock);*  
*... /* 临界资源 */*  
*read\_unlock(&lock);*  
*/* 写时获取锁*/*  
*write\_lock\_irqsave(&lock, flags);*  
*... /* 临界资源 \*/  
write\_unlock\_irqrestore(&lock, flags);

————————————————  
版权声明：本文为CSDN博主「Paranoid-up」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/weixin_52849254/article/details/128935253>

## 5. Linux 驱动中断的常用函数及作用

中断函数  
中断号

中断号用来区分不同的中断，Linux内核使用一个int变量表示中断号。中断号也称为中断线。  
中断API函数  
request\_irq

Linux内核使用中断需要申请，request\_irq用于申请中断。request\_irq会导致睡眠，不能再中断上下文或禁止睡眠的代码中使用。request\_irq会使能中断。

int request\_irq(unsigned int irq,  
 irq\_handler\_t handler,  
 unsigned long flags,  
 const char \*name,  
 void \*dev);  
  
irq：申请中断的中断号。  
handler：中断处理函数，中断发生后执行中断处理函数。  
flags：中断标志。  
name：中断名，设置完成后在/proc/interrupts中查看中断名。  
dev：设备结构体，如果将flags设置为IRQF\_SHARED共享中断，dev用于区分不同的中断，dev设备结构体传递给中断处理函数的第二个参数。  
返回值：0，中断申请成功；负值，中断申请失败，-EBUSY表示中断已经被申请。

中断标志flags 标志 描述  
IRQF\_SHARED 多个设备共享一个中断号，共享的所有中断都必须指定IRQF\_SHARED标志  
IRQF\_ONESHOT 中断执行一次就结束  
IRQF\_TRIGGER\_NONE 无触发  
IRQF\_TRIGGER\_RISING 上升沿触发  
IRQF\_TRIGGER\_FALLING 下降沿触发  
IRQF\_TRIGGER\_HIGH 高电平触发  
IRQF\_TRIGGER\_LOW 低电平触发  
free\_irq

中断使用完后释放相应的中断，如果中断不是共享中断，free\_irq会删除中断处理函数并禁止中断。

void free\_irq(unsigned int irq,  
 void \*dev);  
  
irq：释放的中断号。  
dev：区分不同的共享中断，共享中断只有在释放最后中断处理函数时才会被禁止。  
返回值：无。

中断处理函数

第一个参数：中断号。

第二个参数：指向void的指针，与request\_irq函数的dev参数保持一直，用于区分共享中断的不同设备。

返回值：irqreturn\_t类型，共有三种返回值，一般使用IRQ\_HANDLED。

/\* 中断处理函数 \*/  
irqreturn\_t(\*irq\_handler\_t)(int, void \*)  
   
/\* 返回值irqreturn\_t类型定义 \*/  
enum irqreturn {  
 IRQ\_NONE = (0 << 0);  
 IRQ\_HANDLED = (1 << 0);  
 IRQ\_WAKE\_THREAD = (1 << 1);  
};  
   
typedef enum irqreturn irqreturn\_t

中断使能/禁止函数

disable\_irq函数需要等到当前正在执行的中断处理函数执行完才返回，需要保证不会产生新的中断，并且所有已经开始执行的中断程序已经全部退出。

/\* 中断使能 \*/  
void enable\_irq(unsigned int irq);  
/\* 中断禁止 \*/  
void disable\_irq(unsigned int irq);  
  
irq：中断号。

disable\_irq\_nosync函数调用后立即返回，不会等待当前中断处理程序执行完毕。

void disable\_irq\_nosync(unsigned int irq);

使能全局中断和禁止全局中断

local\_irq\_enable();  
local\_irq\_disable();

local\_irq\_save禁止中断，把中断状态保存在flags；local\_irq\_restore恢复中断，恢复至flags状态。

local\_irq\_save(flags);  
local\_irq\_restore(flags);

中断上半部和下半部

上半部：中断处理函数，对时间敏感、处理速度较快、不会占用很长时间的操作放在上半部进行。

下半部：中断处理过程中比较耗时的程序放在下半部进行，Linux内核提供多种下半部机制。

Linux内核将中断分为上半部和下半部的主要目的：实现中断处理函数的快进快出。

上半部和下半部举例：上半部将数据拷贝到内存，下半部对数据进行处理。

需要根据实际情况判断哪些代码属于上半部还是下半部，可参考如下情况进行判断：

要处理的内容不希望被其他中断打断，放在上半部。  
要处理的任务对时间敏感，放在上半部。  
要处理的任务和硬件有关，放在上半部。  
其余的任务，放在下半部。

中断下半部常用处理方法  
软中断（不推荐使用）

Linux内核定义softirq\_action结构体表示软中断。

struct softirq\_action {  
 void (\*action)(struct softirq\_action \*);  
?;

在kernel/softirq.c中定义了10个软中断。softirq\_action结构体中的action成员变量是软中断的服务函数。

static struct softirq\_action softirq\_vec[NR\_SOFTRQS];  
   
enum {  
 HI\_SOFTIRQ=0, /\* 高优先级软中断 \*/  
 TIMER\_SOFTIRQ, /\* 定时器软中断 \*/  
 NET\_TX\_SOFTIRQ, /\* 网络数据发送软中断 \*/  
 NET\_RX\_SOFTIRQ, /\* 网络数据接收软中断 \*/  
 BLOCK\_SOFTIRQ,  
 BLOCK\_IOPOLL\_SOFTIRQ,  
 TASKLET\_SOFTIRQ, /\* tasklet 软中断 \*/  
 SCHED\_SOFTIRQ, /\* 调度软中断 \*/  
 HRTIMER\_SOFTIRQ, /\* 高精度定时器软中断 \*/  
 RCU\_SOFTIRQ, /\* RCU 软中断 \*/  
 NR\_SOFTIRQS  
};

注册软中断处理函数

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*));

nr：要开起的软中断，从10个软中断中选择。  
action：软中断对应的处理函数。  
返回值：无。

触发中断

void raise\_softirq(unsigned int nr);

nr：要开起的软中断，从10个软中断中选择。  
返回值：无。

软中断必须哎编译的时候静态注册。  
tasklet（建议使用）

Linux内核定义tasklet\_struct结构体表示tasklet。

struct tasklet\_struct {  
 struct tasklet\_struct \*next; /\* 下一个 tasklet \*/  
 unsigned long state; /\* tasklet 状态 \*/  
 atomic\_t count; /\* 计数器，记录对 tasklet 的引用数 \*/  
 void (\*func)(unsigned long); /\* tasklet 执行的函数 \*/  
 unsigned long data; /\* 函数 func 的参数 \*/  
};

先定义一个tasklet，然后使用tasklet\_init初始化。

void tasklet\_init(struct tasklet\_struct \*t,  
 void (\*func)(unsigned long),   
 unsigned long data);  
  
 t：要初始化的tasklet。  
func：tasklet 的处理函数。  
data：要传递给func函数的参数  
返回值：无。

使用宏DECLARE\_TASKLET一次性完成tasklet的定义和初始化。

DECLARE\_TASKLET(name, func, data);

name：tasklet的名字。  
func：tasklet的处理函数。  
data：传递给func的参数。

tasklet也要用到上半部，只是上半部的中断处理函数重点是调度tasklet\_schedule使tasklet在合适的时间运行。

void tasklet\_schedule(struct tasklet\_struct \*t);

t：要调用的tasklet，也是宏DECLARE\_TASKLET里面的name。  
返回值：无。

tasklet例程

// 定义tasklet  
struct tasklet\_struct test\_tasklet;  
   
// tasklet处理函数  
void test\_tasklet\_func(unsigned long data) {  
 // tasklet的具体处理代码  
}  
   
// 中断处理函数  
irqreturn\_t test\_handler(int irq, void \*dev\_id) {  
   
 //调度tasklet  
 tasklet\_schedule(&test\_tasklet);  
   
}  
   
// 驱动入口函数  
static int \_\_init xxx\_init(void) {  
 // 初始化tasklet  
 tasklet\_init(&test\_tasklet, test\_tasklet\_func, data);  
 // 注册中断处理函数  
 request\_irq(xxx\_irq, test\_handler, 0, "xxx", &xxx\_dev);  
}

工作队列（需要睡眠的工作中使用）

工作队列在进程上下文执行，将要推后的工作交给一个内核线程去执行，允许睡眠或重新调度。

Linux内核定义work\_struct结构体表示一个工作。

struct work\_struct {  
 atomic\_long\_t data;  
 struct list\_head entry;  
 work\_func\_t func; // 工作队列处理函数  
};

许多工作组成工作队列，定义workqueue\_struct结构体表示工作队列。

struct workqueue\_struct {  
 struct list\_head pwqs;  
 struct list\_head list;  
 struct mutex mutex;  
 int work\_color;  
 int flush\_color;  
 atomic\_t nr\_pwqs\_to\_flush;  
 struct wq\_flusher \*first\_flusher;  
 struct list\_head flusher\_queue;  
 struct list\_head flusher\_overflow;  
 struct list\_head maydays;  
 struct worker \*rescuer;  
 int nr\_drainers;  
 int saved\_max\_active;  
 struct workqueue\_attrs \*unbound\_attrs;  
 struct pool\_workqueue \*dfl\_pwq;  
 char name[WQ\_NAME\_LEN];  
 struct rcu\_head rcu;  
 unsigned int flags \_\_\_\_cacheline\_aligned;  
 struct pool\_workqueue \_\_percpu \*cpu\_pwqs;  
 struct pool\_workqueue \_\_rcu \*numa\_pwq\_tbl[];  
};

Linux内核使用工作者线程worker thread处理工作队列中的各个工作，Linux内核定义worker结构体表示工作者线程。

struct worker {  
 union {  
 struct list\_head entry;  
 struct hlist\_node hentry;  
 };  
 struct work\_struct \*current\_work;  
 work\_func\_t current\_func;  
 struct pool\_workqueue \*current\_pwq;  
 bool desc\_valid;  
 struct list\_head scheduled;  
 struct task\_struct \*task;  
 struct worker\_pool \*pool;  
 struct list\_head node;  
 unsigned long last\_active;  
 unsigned int flags;  
 int id;  
 char desc[WORKER\_DESC\_LEN];  
 struct workqueue\_struct \*rescue\_wq;  
};

Linux驱动开发，只需要定义工作work\_struct即可。定义work\_struct结构体变量，然后INIT\_WORK初始化。

#define INIT\_WORK(\_work, \_func)

\_work：要初始化的工作。  
\_func：工作处理函数 。

也可以使用宏DECLARE\_WORK一次性完成工作的创建和初始化。

#define DECLARE\_WORK(n, f)

n：定义的工作。  
f：工作处理函数。

工作的调度函数schedule\_work

bool schedule\_work(struct work\_struct \*work);

work：要调度的工作。  
返回值：0，成功；其他值，失败。

工作队列例程

// 定义工作  
struct work\_struct test\_work;  
   
// 工作处理函数  
void test\_work\_func\_t(struct work\_struct \*work) {  
   
}  
   
// 中断处理函数  
irqreturn\_t test\_handler(int irq, void \*dev\_id) {  
   
 // 调度工作work  
 schedule\_work(&test\_work);  
   
}  
   
// 驱动入口函数  
static int \_\_init xxx\_init(void) {  
 // 初始化工作  
 INIT\_WORK(&test\_work, test\_work\_func\_t);  
 // 注册中断处理函数  
 request\_irq(xxx\_irq, test\_handler, 0, "xxx", &xxx\_dev);  
}

设备树设置中断信息

interrupt-controller，表示当前节点为中断控制器。  
#interrupt-cells，指定interrupts属性的cells大小。  
interrupts，指定中断号、触发方式。  
interrupt-parent，指定父中断，即中断控制器。

Linux内核通过读取设备树中的中断属性信息配置中断。imx6ull.dtsi文件中intc节点表示如下：

intc: interrupt-controller@00a01000 {  
 compatible = "arm, cortex-a7-gic";  
 #interrupt-cells = <3>;  
 interrupt-controller;  
 reg = <0x00a01000 0x1000>,  
 <0x00a02000 0x100>;  
};

#interrupt-cells描述interrupt属性的cells大小，对于ARM处理器的GIC中断控制器，共有3个cells。

第一个cells表示中断类型，0，SPI中断；1，PPI中断。  
第二个cells表示中断号，SPI中断的中断号范围0-987，PPI中断的中断号范围0-15。  
第三个cells表示标志，bit[3:0]表示中端触发类型，1，上升沿触发；2，下降沿触发；4，高电平触发；8，低电平触发。bit[15:8]表示PPI中断的CPU掩码。

interrupt-controller节点为空表示当前节点是中断控制器。

gpio节点也可作为中断控制器，imx6ull.dtsi文件中gpio5节点表示如下：

gpio5: gpio@0x0ac000 {  
 compatible = "fsl, imx6ul-gpio", "fsl, imx35-gpio";  
 reg = <0x020ac000 0x4000>;  
 interrupts = <GIC\_SPI 74 IRQ\_TYPE\_LEVEL\_HIGH>,  
 <GIC\_SPI 75 IRQ\_TYPE\_LEVEL\_HIGH>;  
 gpio-controller;  
 #gpio-cells = <2>;  
 interrupt-controller;  
 #interrupt-cells = <2>;  
};

interrupts描述中断源信息，中断类型都是SPI，触发电平是高电平，查看芯片手册gpio5的中断号一个是74，对应GPIO5\_IO00GPIO5\_IO15低16位IO，一个是75，对应GPIO5\_IO16GPIO5\_IO31高16位IO。

Linux内核include/linux/irq.h文件中定义了中断号（线）的状态：

/\*  
 \* IRQ line status.  
 \*  
 \* Bits 0-7 are the same as the IRQF\_\* bits in linux/interrupt.h  
 \*  
 \* IRQ\_TYPE\_NONE - default, unspecified type  
 \* IRQ\_TYPE\_EDGE\_RISING - rising edge triggered  
 \* IRQ\_TYPE\_EDGE\_FALLING - falling edge triggered  
 \* IRQ\_TYPE\_EDGE\_BOTH - rising and falling edge triggered  
 \* IRQ\_TYPE\_LEVEL\_HIGH - high level triggered  
 \* IRQ\_TYPE\_LEVEL\_LOW - low level triggered  
 \* IRQ\_TYPE\_LEVEL\_MASK - Mask to filter out the level bits  
 \* IRQ\_TYPE\_SENSE\_MASK - Mask for all the above bits  
 \* IRQ\_TYPE\_DEFAULT - For use by some PICs to ask irq\_set\_type  
 \* to setup the HW to a sane default (used  
 \* by irqdomain map() callbacks to synchronize  
 \* the HW state and SW flags for a newly  
 \* allocated descriptor).  
 \*  
 \* IRQ\_TYPE\_PROBE - Special flag for probing in progress  
 \*  
 \* Bits which can be modified via irq\_set/clear/modify\_status\_flags()  
 \* IRQ\_LEVEL - Interrupt is level type. Will be also  
 \* updated in the code when the above trigger  
 \* bits are modified via irq\_set\_irq\_type()  
 \* IRQ\_PER\_CPU - Mark an interrupt PER\_CPU. Will protect  
 \* it from affinity setting  
 \* IRQ\_NOPROBE - Interrupt cannot be probed by autoprobing  
 \* IRQ\_NOREQUEST - Interrupt cannot be requested via  
 \* request\_irq()  
 \* IRQ\_NOTHREAD - Interrupt cannot be threaded  
 \* IRQ\_NOAUTOEN - Interrupt is not automatically enabled in  
 \* request/setup\_irq()  
 \* IRQ\_NO\_BALANCING - Interrupt cannot be balanced (affinity set)  
 \* IRQ\_MOVE\_PCNTXT - Interrupt can be migrated from process context  
 \* IRQ\_NESTED\_TRHEAD - Interrupt nests into another thread  
 \* IRQ\_PER\_CPU\_DEVID - Dev\_id is a per-cpu variable  
 \* IRQ\_IS\_POLLED - Always polled by another interrupt. Exclude  
 \* it from the spurious interrupt detection  
 \* mechanism and from core side polling.  
 \*/  
enum {  
 IRQ\_TYPE\_NONE = 0x00000000,  
 IRQ\_TYPE\_EDGE\_RISING = 0x00000001,  
 IRQ\_TYPE\_EDGE\_FALLING = 0x00000002,  
 IRQ\_TYPE\_EDGE\_BOTH = (IRQ\_TYPE\_EDGE\_FALLING | IRQ\_TYPE\_EDGE\_RISING),  
 IRQ\_TYPE\_LEVEL\_HIGH = 0x00000004,  
 IRQ\_TYPE\_LEVEL\_LOW = 0x00000008,  
 IRQ\_TYPE\_LEVEL\_MASK = (IRQ\_TYPE\_LEVEL\_LOW | IRQ\_TYPE\_LEVEL\_HIGH),  
 IRQ\_TYPE\_SENSE\_MASK = 0x0000000f,  
 IRQ\_TYPE\_DEFAULT = IRQ\_TYPE\_SENSE\_MASK,  
   
 IRQ\_TYPE\_PROBE = 0x00000010,  
   
 IRQ\_LEVEL = (1 << 8),  
 IRQ\_PER\_CPU = (1 << 9),  
 IRQ\_NOPROBE = (1 << 10),  
 IRQ\_NOREQUEST = (1 << 11),  
 IRQ\_NOAUTOEN = (1 << 12),  
 IRQ\_NO\_BALANCING = (1 << 13),  
 IRQ\_MOVE\_PCNTXT = (1 << 14),  
 IRQ\_NESTED\_THREAD = (1 << 15),  
 IRQ\_NOTHREAD = (1 << 16),  
 IRQ\_PER\_CPU\_DEVID = (1 << 17),  
 IRQ\_IS\_POLLED = (1 << 18),  
};

具体开发板中fxls8471磁力计的中断属性信息示例如下：

fxls8471@1e {  
 compatible = "fsl, fxls8471";  
 reg = <0x1e>;  
 position = <0>;  
   
 interrupt-parent = <&gpio5>;  
 interrupts = <0 8>;  
};

具体开发板的设备树一般只需要添加两行：

interrupt-parent属性设置gpio5为中断控制器 。  
interrupts设置中断信息，0，表示GPIO5\_IO00，8表示低电平触发。

获取中断号  
 irq\_of\_parse\_and\_map

unsigned int irq\_of\_parse\_and\_map(struct device\_node \*dev,  
 int index);  
  
dev：设备节点。  
index：interrupts属性可能包含多条中断信息，index指定要获取的信息。  
返回值：中断号。

gpio\_to\_irq

获取gpio对应的中断号

inT gpio\_to\_irq(unsigned int gpio);

gpio:GPIO编号。  
返回值：GPIO对应的中断号。

————————————————  
版权声明：本文为CSDN博主「奶油芝士汉堡包」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/weixin_41276397/article/details/124284739>

## 6. Linux 驱动 platform 虚拟总线

1、为什么需要platform总线

举一个例子，对于同一个主机来说，他可以支持很多I2C设备，对于同一个I2C设备来说，他也可以给很多主机来用，如果每个主机对应每个设备都需要一段驱动代码的话，会非常的冗余，根据高内聚低耦合的原则，这样是非常不好的。所以就需要这么一个统一的接口，将二者分离开来，设备端只负责设备，驱动端只负责驱动。于是提出platform这个虚拟总线，相应的就有 platform\_driver 和 platform\_device。当设备或者驱动加载时，就会去对面查看是否有匹配的内容。

2、设备端：platform\_device  
2.1 platform\_device结构体

struct platform\_device {  
 const char \*name; //用于匹配的名字  
 int id; //总线号 PLATFORM\_DEVID\_AUTO  
 //bool id\_auto; //TRUE  
 struct device dev; //父类  
 u32 num\_resources; //资源的个数  
 struct resource \*resource; //设备信息结构体  
 char \*driver\_override;   
}  
struct device{  
 void (\*release)(struct device \*dev); //释放资源的函数  
};  
   
struct resource { //设备信息结构体  
 resource\_size\_t start; //资源的起始值   
 resource\_size\_t end; //资源的结束值   
 unsigned long flags; //资源的类型  
 IORESOURCE\_IO //GPIO类型的资源  
 IORESOURCE\_MEM //内存类型的资源  
 IORESOURCE\_IRQ //中断类型的资源  
 IORESOURCE\_DMA //DMA类型的资源  
};

2.2 注册

int platform\_device\_register(struct platform\_device \*);

2.3 注销

void platform\_device\_unregister(struct platform\_device \*);

3、驱动端：platform\_driver  
3.1 platform\_driver结构体

struct platform\_driver {  
 int (\*probe)(struct platform\_device \*); //匹配成功执行的函数  
 int (\*remove)(struct platform\_device \*); //分离的时候执行的函数  
 struct device\_driver driver; //父类  
 const struct platform\_device\_id \*id\_table;   
};  
   
struct device\_driver {  
 const char \*name;   
 const struct of\_device\_id \*of\_match\_table;   
}；

3.2 注册

int platform\_driver\_register (struct platform\_driver \*);

3.3 注销

void platform\_driver\_unregister(struct platform\_driver \*);

4、总线  
4.1 bus\_type

Linux 内核用 bus\_type 结构体来表示总线，我们所用的 I2C、SPI、USB 都是用这个结构体来定义的。该结构体如下：

struct bus\_type {  
 const char \*name;  
 const char \*dev\_name;  
 struct device \*dev\_root;  
 struct device\_attribute \*dev\_attrs; /\* use dev\_groups instead \*/  
 const struct attribute\_group \*\*bus\_groups;  
 const struct attribute\_group \*\*dev\_groups;  
 const struct attribute\_group \*\*drv\_groups;  
   
 int (\*match)(struct device \*dev, struct device\_driver \*drv);  
 int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);  
 int (\*probe)(struct device \*dev);  
 int (\*remove)(struct device \*dev);  
 void (\*shutdown)(struct device \*dev);  
   
 int (\*online)(struct device \*dev);  
 int (\*offline)(struct device \*dev);  
   
 int (\*suspend)(struct device \*dev, pm\_message\_t state);  
 int (\*resume)(struct device \*dev);  
   
 const struct dev\_pm\_ops \*pm;  
   
 const struct iommu\_ops \*iommu\_ops;  
   
 struct subsys\_private \*p;  
 struct lock\_class\_key lock\_key;  
};

4.2 platform\_bus\_type

platform总线是 bus\_type的一个具体实例，定义如下：

struct bus\_type platform\_bus\_type = {  
 .name = "platform",  
 .dev\_groups = platform\_dev\_groups,  
 .match = platform\_match,  
 .uevent = platform\_uevent,  
 .pm = &platform\_dev\_pm\_ops,  
};

5、匹配  
5.1 匹配规则，platform\_match

在platform\_bus\_type中，match函数就是用来匹配的，platform\_match函数实现如下：

static int platform\_match(struct device \*dev, struct device\_driver \*drv)  
{  
 struct platform\_device \*pdev = to\_platform\_device(dev);  
 struct platform\_driver \*pdrv = to\_platform\_driver(drv);  
   
 /\* When driver\_override is set, only bind to the matching driver \*/  
 if (pdev->driver\_override)  
 return !strcmp(pdev->driver\_override, drv->name);  
   
 /\* Attempt an OF style match first \*/  
 if (of\_driver\_match\_device(dev, drv))  
 return 1;  
   
 /\* Then try ACPI style match \*/  
 if (acpi\_driver\_match\_device(dev, drv))  
 return 1;  
   
 /\* Then try to match against the id table \*/  
 if (pdrv->id\_table)  
 return platform\_match\_id(pdrv->id\_table, pdev) != NULL;  
   
 /\* fall-back to driver name match \*/  
 return (strcmp(pdev->name, drv->name) == 0);  
}  
   
static inline int of\_driver\_match\_device(struct device \*dev,  
 const struct device\_driver \*drv)  
{  
 return of\_match\_device(drv->of\_match\_table, dev) != NULL;  
}  
  
1、platform\_device.driver\_override 和 platform\_driver.driver.name  
  
2、设备树中的compatible 和 platform\_driver.driver.of\_match\_table 的 compatible  
  
3、platform\_device.name 和 platform\_driver.id\_table[i].name  
  
4、platform\_device.name 和 platform\_driver.driver.name

5.2 platform\_device匹配流程

platform\_device\_register(&pdev){  
 return platform\_device\_add(pdev)  
}  
->  
pdev->dev.bus = &platform\_bus\_type  
device\_add(&pdev->dev)  
->  
bus\_add\_device(dev) //放入链表  
bus\_probe\_device(dev)  
->  
device\_initial\_probe(dev)  
->  
\_\_device\_attach(dev, true)  
->  
bus\_for\_each\_drv(dev->bus, NULL, &data, \_\_device\_attach\_driver)  
->  
\_\_device\_attach\_driver  
->  
driver\_match\_device(drv, dev) //是否匹配  
return driver\_probe\_device(drv, dev) //调用 probe 函数

5.3 platform\_driver匹配流程

#define platform\_driver\_register(drv)  
->  
\_\_platform\_driver\_register(drv, THIS\_MODULE)  
->  
drv->driver.bus = &platform\_bus\_type; //指定为platform bus  
driver\_register(&drv->driver)  
->  
bus\_add\_driver(drv) //放入链表  
->  
driver\_attach(drv)  
->  
bus\_for\_each\_dev(drv->bus, NULL, drv, \_\_driver\_attach)  
->  
\_\_driver\_attach  
->  
driver\_match\_device(drv, dev)  
->  
drv->bus->match(dev, drv) //是否匹配

6、在没有设备树时，使用name进行匹配  
6.1 设备端程序

struct resource res[] = {  
 [0] = {  
 .start = 0x12345678,  
 .end = 0x12345678+49,  
 .flags = IORESOURCE\_MEM,   
 },  
 [1] = {  
 .start = 71,  
 .end = 71,  
 .flags = IORESOURCE\_IRQ,  
 }  
};  
   
void pdev\_release(struct device \*dev)  
{  
   
}  
   
struct platform\_device pdev = {  
 .name = "aabbccdd",  
 .id = PLATFORM\_DEVID\_AUTO, //自动分配  
 .dev = {  
 .release = pdev\_release,  
 },  
 .resource = res,  
 .num\_resources = ARRAY\_SIZE(res),  
};  
   
static int \_\_init pdev\_init(void)  
{  
 return platform\_device\_register(&pdev);  
}  
   
static void \_\_exit pdev\_exit(void)   
{  
 platform\_device\_unregister(&pdev);  
}

6.2 驱动端程序

int pdrv\_probe(struct platform\_device\*pdev)  
{  
 return 0;  
}  
   
int pdrv\_remove(struct platform\_device\*pdev)  
{  
 return 0;  
}  
   
struct platform\_driver pdrv = {  
 .probe = pdrv\_probe,  
 .remove = pdrv\_remove,  
 .driver = {  
 .name = "aabbccdd",  
 },  
};  
   
static int \_\_init pdrv\_init(void)  
{  
 return platform\_driver\_register(&pdrv);  
}  
   
static void \_\_exit pdrv\_exit(void)  
{  
 platform\_driver\_unregister(&pdrv);  
}

7、在没有设备树时，使用 idtable 进行匹配  
7.1 设备端程序

与 6.1 设备端不一样的地方

struct platform\_device pdev = {  
 .name = "hello1",  
 .id = PLATFORM\_DEVID\_AUTO, //自动分配  
 .dev = {  
 .release = pdev\_release,  
 },  
 .resource = res,  
 .num\_resources = ARRAY\_SIZE(res),  
};

7.2 驱动端程序

与 6.2 驱动端不一样的地方

struct platform\_device\_id idtable[] = {  
 {"hello1",0},  
 {"hello2",1},  
 {"hello3",2},  
 {/\*end\*/}  
};  
   
struct platform\_driver pdrv = {  
 .probe = pdrv\_probe,  
 .remove = pdrv\_remove,  
 .driver = {  
 .name = "aabbccdd", //这个name一定要填，因为要以这个名字创建文件夹   
 },  
 .id\_table = idtable,  
};

8、获取设备信息  
8.1 获取设备信息的API  
8.1.1 platform\_get\_resource

struct resource \*platform\_get\_resource(struct platform\_device \*dev,  
 unsigned int type, unsigned int index)  
/\*  
功能：在驱动中获取设备信息  
参数：  
 @dev :platform\_device的结构体指针  
 @type:资源的类型  
 @index:同类型资源的索引号  
返回值：成功返回resource的结构体指针，失败返回NULL  
\*/

8.1.2 platform\_get\_irq

int platform\_get\_irq(struct platform\_device \*dev, unsigned int index)  
/\*  
功能：获取中断类型的资源  
参数：  
 @dev :platform\_device的结构体指针  
 @index:中断类型资源的索引号   
返回值：成功返回中断号，失败返回错误码  
\*/

8.1.3 根据device\_node获取设备信息

Linux驱动开发：设备树节点与属性\_凛冬将至\_\_的博客-CSDN博客的7与8两节

8.2 驱动程序

完整的驱动程序就不再重写了，在 6.2 驱动程序中 probe 函数中得到设备信息

struct resource \*res;  
int pdrv\_probe(struct platform\_device\*pdev)  
{  
 res = platform\_get\_resource(pdev,IORESOURCE\_MEM,0);  
   
 irqno = platform\_get\_irq(pdev,0);  
   
 printk("addr = %#llx,irqno = %d\n",res->start,irqno);  
   
 return 0;  
}

9、module\_platform\_driver：一键注册platform

//在linux/platform\_device.h中  
#define module\_platform\_driver(\_\_platform\_driver)   
 module\_driver(\_\_platform\_driver, platform\_driver\_register,   
 platform\_driver\_unregister)  
   
//##代表字符串的拼接  
#define module\_driver(\_\_driver, \_\_register, \_\_unregister, ...)   
static int \_\_init \_\_driver##\_init(void)   
{   
 return \_\_register(&(\_\_driver) , ##\_\_VA\_ARGS\_\_);   
}   
module\_init(\_\_driver##\_init);   
static void \_\_exit \_\_driver##\_exit(void)   
{   
 \_\_unregister(&(\_\_driver) , ##\_\_VA\_ARGS\_\_);   
}   
module\_exit(\_\_driver##\_exit);

使用该宏： module\_platform\_driver(pdrv)，即被转化为：

#define module\_platform\_driver(pdrv)   
 module\_driver(pdrv, platform\_driver\_register, platform\_driver\_unregister)  
   
#define module\_driver(pdrv, platform\_driver\_register, platform\_driver\_unregister)   
   
static int \_\_init pdrv\_init(void)   
{   
 return platform\_driver\_register(&pdrv);   
}   
   
static void \_\_exit pdrv\_exit(void)   
{   
 platform\_driver\_unregister(&pdrv);   
}   
module\_init(pdrv\_init);   
module\_exit(pdrv\_exit);

10、MODULE\_DEVICE\_TABLE：实现热插拔  
10.1 定义以及使用方法

//定义在linux/module.h中  
#define MODULE\_DEVICE\_TABLE(type, name)   
extern const typeof(name) \_\_mod\_##type##\_\_##name##\_device\_table   
 \_\_attribute\_\_ ((unused, alias(\_\_stringify(name))))

使用时，参数如下：

MODULE\_DEVICE\_TABLE（of，match\_table）  
of：总线类型  
match\_table：idtable数组首地址

10.2 如何实现热插拔的功能

1.先将 pdev.ko 和 pdrv.ko 放到下面的目录中  
/lib/modules/5.4.0-148-generic/kernel/drivers/platform  
2.执行depmod -a命令，让内核重新检索文件的位置  
3.安装 pdev.ko , pdrv.ko 会被自动安装

11、 platform设备树匹配  
11.1 修改设备树以及驱动程序的compatible属性  
11.1.1 驱动端

在 5.1 中我们已经看过了匹配的流程，其中第二种方式就是用设备树匹配：设备树中的compatible 和 platform\_driver.driver.of\_match\_table 的 compatible进行匹配

struct of\_device\_id oftable[] = {  
 {.compatible = "aaa,aaa",},  
 {.compatible = "bbb,bbb",},  
 {.compatible = "ccc,ccc",},  
 {/\*end\*/} //一定要有一个空的在  
};  
   
struct platform\_driver {  
 .driver = {  
 .of\_match\_table = oftable,  
 },  
};  
   
struct device\_driver driver {  
 struct device\_driver driver;  
}  
   
struct device\_driver {  
 const struct of\_device\_id \*of\_match\_table;  
}  
   
struct of\_device\_id {  
 char name[32];  
 char type[32];  
 char compatible[128]; //通过本选项和设备树完成匹配  
 const void \*data;  
};

11.1.2 设备树

节点下需要有个 compatible 属性，并且该属性需要与 oftable 中的 compatible 名字相同，例如：

myplatform{  
 compatible = "aaa,aaa";  
};

11.2 驱动程序：获取设备树中的中断以及GPIO资源

有关GPIO部分请看:  
  
Linux驱动开发：gpio子系统\_凛冬将至\_\_的博客-CSDN博客  
  
有关中断部分请看：  
  
Linux驱动开发：中断子系统\_凛冬将至\_\_的博客-CSDN博客  
  
有关阻塞部分请看：  
  
Linux驱动开发 IO模型：阻塞IO\_linux阻塞io\_凛冬将至\_\_的博客-CSDN博客

11.2.1 修改设备树

在根节点下添加自己的节点

myplatform{  
 compatible = "aaa,aaa";  
 interrupt-parent = <&gpiof>;  
 interrupts = <9 0>;  
 reg = <0x12345678 0x40>;  
 led1 = <&gpioe 10 0>;  
};

11.2.2 驱动程序

#define IRQNAME "key\_irq"  
int irqno, major;  
struct gpio\_desc\* desc;  
struct class\* cls;  
struct device\* dev;  
wait\_queue\_head\_t wq;  
int condition=0;  
int status=0;  
irqreturn\_t key\_irq\_handle(int irq, void\* dev)  
{  
 //1.设置status和led1  
 status = gpiod\_get\_value(desc);  
 status = !status;  
 gpiod\_set\_value(desc,status);  
   
 //2唤醒  
 condition=1;  
 wake\_up\_interruptible(&wq);  
   
 return IRQ\_HANDLED;  
}  
int pdrv\_open(struct inode\* inode, struct file\* file)  
{  
 return 0;  
}  
   
ssize\_t pdrv\_read(struct file\*file,  
 char \_\_user\*ubuf, size\_t size, loff\_t\*offs)  
{  
 int ret;  
 if(file->f\_flags & O\_NONBLOCK){  
 return -EINVAL;  
 }else{  
 ret = wait\_event\_interruptible(wq,condition);  
 }  
   
 ret = copy\_to\_user(ubuf,&status,size);  
   
 condition = 0;  
   
 return size;  
}  
int pdrv\_close(struct inode\* inode, struct file\* file)  
{  
 return 0;  
}  
struct file\_operations fops = {  
 .open = pdrv\_open,  
 .read = pdrv\_read,  
 .release = pdrv\_close,  
};  
int pdrv\_probe(struct platform\_device\* pdev)  
{  
 int ret;  
 // 1.获取设备树中的设备信息  
 irqno = platform\_get\_irq(pdev, 0);  
 desc = gpiod\_get\_from\_of\_node(pdev->dev.of\_node, "led1", 0, GPIOD\_OUT\_LOW, NULL);  
   
 // 2.注册中断  
 ret = request\_irq(irqno, key\_irq\_handle, IRQF\_TRIGGER\_FALLING, IRQNAME, NULL);  
   
 // 3.注册字符设备驱动  
 major = register\_chrdev(0, IRQNAME, &fops);  
 cls = class\_create(THIS\_MODULE, IRQNAME);  
 dev = device\_create(cls, NULL, MKDEV(major, 0), NULL, IRQNAME);  
   
 //4.初始化等待队列头  
 init\_waitqueue\_head(&wq);  
 return 0;  
}  
int pdrv\_remove(struct platform\_device\* pdev)  
{  
 device\_destroy(cls, MKDEV(major, 0));  
 class\_destroy(cls);  
 unregister\_chrdev(major, IRQNAME);  
 free\_irq(irqno, NULL);  
 gpiod\_put(desc);  
 return 0;  
}  
const struct of\_device\_id oftable[] = {  
 {  
 .compatible = "aaa,aaa",  
 },  
 { /\*end\*/ }  
};  
struct platform\_driver pdrv = {  
 .probe = pdrv\_probe,  
 .remove = pdrv\_remove,  
 .driver = {  
 .name = "bbb", //虽然用不到，但是一定要写  
 .of\_match\_table = oftable,  
 },  
   
};  
//一键注册  
module\_platform\_driver(pdrv);

11.3 应用程序

int main(int argc,const char \* argv[])  
{  
 int fd,status;  
 if((fd = open("/dev/key\_irq",O\_RDWR))==-1)  
 PRINT\_ERR("open error");  
   
 while(1){  
 read(fd,&status,sizeof(status));  
 printf("status = %d\n",status);  
 }  
   
 close(fd);  
 return 0;  
}

————————————————  
版权声明：本文为CSDN博主「凛冬将至**」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。**  
**原文链接：<https://blog.csdn.net/W>**winter/article/details/130517484

## 7. IPCC 通信框架

由于MP157 是一款多核异构的芯片，其中既包含的高性能的A7 核及实时性强的M4 内核，那么这两种处理器在工作时，怎么互相协调配合呢？这就涉及到了核间通信的概念了。

IPCC (inter-processor communication controller) 用于处理器间的数据交换的通知。它提供了一种非阻塞的信号机制，并提供原子的方式进行信号发布和信息检索。注意，核间通信的共享内存缓冲区是在MCU 的SRAM 中分配的，它不是IPCC 外设的一部分。  
外设简述

IPCC 外设提供了硬件支持，来管理两个处理器实例之间的处理器间通信。每个处理器拥有特定的寄存器区域和中断。有点像硬件信号量的功能。

IPCC 提供了六个双向通道信号。每个通道分为两个子通道，每个子通道提供从“发送方”处理器到“接收方”处理器的单向信号:

P1\_TO\_P2 子通道（从P1 发到P2）  
P2\_TO\_P1 子通道（从P2 发到P1）

子通道中包括如下功能：

一个标志位，用于标识通道正在被占用和空闲的两种状态，这个标志被“发送方”处理器设置为被占用，并被“接收方”处理器清除。  
两个相关的中断（所有通道都共享）:  
– RXO: RX 通道被占用，连接到“接收器”处理器。  
– TXF: TX 通道空闲，连接到“发送”处理器。  
带多路复用的中断掩码功能。

IPCC 支持以下信道的操作模式：

* 单工通信方式：  
  – 仅使用一个子信道。  
  – 单向消息：“发送者”处理器将通信数据发布到内存中后，它将通道状态标志设置为已占用。当消息被处理时，“接收者”处理器清除该标志。
* 半双工通讯方式：  
   – 仅使用一个子信道。  
   – 双向消息：“发送者”处理器将通信数据发布到内存中后，它将通道状态标志设置为已占用。当消息被处理并且响应在共享内存中可用时，“接收器”处理器将清除该标志。  
   全双工通讯方式：  
   – 子通道用于异步模式。  
   – 通过将子通道状态标志设置为占用，任何处理器都可以异步发布消息。当消息被处理时，“接收者”处理器清除该标志。可以将这种模式视为给定通道上两个单工模式的组合。

核间通信的模型如下：

在这里插入图片描述  
框架简述

IPCC 作为核间通信的桥梁，它仅承担着通知的角色，负责消息的分发、中断的处理等。

实际上，IPCC 外设这个角色只是多核异构核间通信中的一块，在我们使用多核异构核间通信时，往往不仅希望使用到核间的消息通知，还希望能在不同的核心中进行数据的交互（比如M4 核进行实时的AD 数据采集处理，完成后，M4 核可通过异构的框架将数据呈递给A7 核，A7 核再进行更复杂的应用）。那么在这个需求的驱动下，就出现了一些框架相互配合使用的情况了，下面我们就给大家介绍这些内核框架。  
RemoteProc framework

远程处理器框架（RPROC、RemoteProc）允许不同的平台/体系结构控制（打开电源，加载固件，关闭电源）远程处理器，同时抽象出硬件差异。此外，它还提供监视和调试远程协处理器的服务。

以MP157 为例，其RemoteProc 可分为两块，分别是A7 核端、M4 核端：

remoteproc：这是远程处理器框架的通用部分（在MP157 中为A7 核端）。

它的作用是：

－ 将ELF 固件加载到远程处理器内存中。  
－ 解析固件资源表以设置关联的资源（例如IPC，内存分割和跟踪）。  
－ 控制远程处理器的执行（启动，停止⋯）。  
－ 提供监视和调试远程固件的服务。

stm32\_rproc：这是远程处理器平台驱动程序（在MP157 中为M4 核端）。

它的作用是：

－ 将stm32 特定的功能（回调）注册到RPROC 框架。  
－ 处理与远程处理器关联的平台资源（例如寄存器，看门狗，复位，时钟和存储器）。  
－ 通过邮箱框架将通知（通知）转发到远程处理器。

ST 官方参考资料：  
<https://wiki.stmicroelectronics.cn/stm32mpu/wiki/Linux_RPMsg_framework_overview>  
RPMsg framework

此小节为大家简述有关Linux RPMsg 框架的内容。RPMsg 框架是一个基于virtio 的消息总线，它允许本地处理器与系统上可用的远程处理器通信。

此框架在多核异构中承担的角色如下图：

在这里插入图片描述

Linux RPMsg 框架是在virtio 框架顶层上实现的消息传送框架，其用于主机和远程处理器进行通信。它基于virtio vring，可通过共享内存向远程CPU 发送消息或从远程CPU 接收消息。

这些vring 是单向的，一个vring 专用于发送到远程处理器的消息，另一个vring 用于从远程处理器接收的消息。此外，共享缓冲区需要在两个处理器都可见的内存空间中创建。

当新消息在共享缓冲区中等待时，会使用到另一个框架Linux Mailbox framework ，该框架将用于通知对应的Core。

依靠这些框架，RPMsg 框架实现了基于不同通道的通信。通道可被文本名称标识，并有一个本地(“源”) 的RPMsg 地址和一个远程(“目的”) 的RPMsg 地址。

在远程处理器端（MP157 则为M4 核），也必须使用RPMSG 框架。RPMSG 框架的实现存在几种解决方案，ST 建议使用OpenAMP 方案，并在SDK 中给出了示例。

Github OpenAMP 框架.

简单来说，MP157 的A7 核与M4 核，通过一个标准的RPMsg 框架来建立起联系，完成数据传递。

在这里插入图片描述

具体原理可以参考：

RPMsg-Messaging-Protocol .  
RPMsg-Communication-Flow .

Linux 内核源码目录给出的rpmsg client 的示例代码位置如下：

samples/rpmsg/rpmsg\_client\_sample.c

rpmsg 框架Linux 内核驱动源码位于：

drivers/rpmsg

ST 官方参考资料：

<https://wiki.stmicroelectronics.cn/stm32mpu/wiki/Linux_remoteproc_framework_overview>  
Mailbox framework

此小节为大家简述有关Linux 邮箱框架的内容。邮箱框架涉及异构多核系统的处理器间通信。

此框架的结构如下图：

在这里插入图片描述

邮箱框架被用于内核间进行消息或信号的交换，常用于主机和协处理器间。邮箱由以下模块组成：

一个邮箱控制器(mailbox controller)，依赖于硬件平台实现，比如MP157 的IPCC 外设：  
– 它负责配置和处理来自IPCC 外围设备的IRQ。  
– 它为邮箱客户端提供了通用API。  
  
一个邮箱客户端(mailbox client)，负责发送或接收消息。

关于此框架的权威描述，在内核文档中的如下目录：

Documentation/mailbox.txt

一般而言mailbox controller 和client 都由芯片厂商来负责实现，因为这依赖于外设。我们更常关注的，则是mailbox client 的创建和使用。

ST 实现的mailbox client 代码位置如下：

drivers/remoteproc/stm32\_rproc.c

在内核中还给出了一份mailbox client 的示例驱动代码，代码通过debugfs 子系统，将mailbox 的操作暴露给了用户空间，用户可以直接通过debugfs 来使用mailbox，进行消息在不同内核中的传递。

mailbox 框架的设备树描述可参考内核源码文档：

Documentation/devicetree/bindings/mailbox/mailbox.txt

一个简单的mailbox client 设备树节点，可以参考内核源码目录：

Documentation/devicetree/bindings/mailbox/sti-mailbox.txt

内核源码目录给出的mailbox client 的示例代码位置如下：

drivers/mailbox/mailbox-test.c

ST 官方参考资料：

<https://wiki.stmicroelectronics.cn/stm32mpu/wiki/Linux_Mailbox_framework_overview>  
框架小结

前面介绍了三个框架，它们并不是独立工作的，而是相互协调的，彼此关联。我们可以通过两张图来查看它们之间的关系。

以RemoteProc 框架为主视角出发：

在这里插入图片描述

可以理清三个框架的关系，RemoteProc 可以说是骨架，关联到了RPMsg 框架、Mailbox 框架。  
设备树插件描述  
IPCC 设备树节点

设备树节点位于arch/arm/boot/dts/stm32mp157c.dtsi

列表1: IPCC 设备树节点

ipcc: mailbox@4c001000 {  
 compatible = "st,stm32mp1-ipcc";  
 #mbox-cells = <1>;  
 reg = <0x4c001000 0x400>;  
 st,proc-id = <0>;  
 interrupts-extended =  
 <&intc GIC\_SPI 100 IRQ\_TYPE\_LEVEL\_HIGH>,  
 <&intc GIC\_SPI 101 IRQ\_TYPE\_LEVEL\_HIGH>,  
 <&exti 61 1>;  
 interrupt-names = "rx", "tx", "wakeup";  
 clocks = <&rcc IPCC>;  
 wakeup-source;  
 power-domains = <&pd\_core>;  
 status = "disabled";  
};

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15

使用节点位于arch/arm/boot/dts/stm32mp157a-basic.dts

列表2: 使用IPCC 设备树节点

&ipcc {  
 status = "okay";  
};

1  
2  
3

设备树中的compatible =“st,stm32mp1-ipcc”属性，会匹配到drivers/mailbox/stm32-ipcc.c 驱动程序，驱动程序中会创建一个mbox controller。  
A7<–>M4 rproc 设备树节点

设备树节点位于arch/arm/boot/dts/stm32mp157c.dtsi

列表3: rproc 设备树节点

m4\_rproc: m4@0 {  
 compatible = "st,stm32mp1-rproc";  
 #address-cells = <1>;  
 #size-cells = <1>;

ranges = <0x00000000 0x38000000 0x10000>,  
 <0x30000000 0x30000000 0x60000>,  
 <0x10000000 0x10000000 0x60000>;  
resets = <&rcc MCU\_R>;  
reset-names = "mcu\_rst";  
st,syscfg-pdds = <&pwr 0x014 0x1>;  
st,syscfg-holdboot = <&rcc 0x10C 0x1>;  
st,syscfg-tz = <&rcc 0x000 0x1>;  
st,syscfg-rsc-tbl = <&tamp 0x144 0xFFFFFFFF>;  
status = "disabled";  
  
m4\_system\_resources {  
 compatible = "rproc-srm-core";  
 status = "disabled";  
};

};

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21

使用节点位于arch/arm/boot/dts/stm32mp157a-basic.dts

列表4: 使用rproc 设备树节点

&m4\_rproc {  
 memory-region = <&retram>, <&mcuram>, <&mcuram2>, <&vdev0vring0>,  
 <&vdev0vring1>, <&vdev0buffer>;  
 mboxes = <&ipcc 0>, <&ipcc 1>, <&ipcc 2>;  
 mbox-names = "vq0", "vq1", "shutdown";  
 interrupt-parent = <&exti>;  
 interrupts = <68 1>;  
 interrupt-names = "wdg";  
 wakeup-source;  
 recovery;  
 status = "okay";  
};

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12

设备树中的compatible = “st,stm32mp1-rproc” 属性， 会匹配到drivers/remoteproc/stm32\_rproc.c 驱动程序，驱动程序中会创建一个mbox client，并基于RemoteProc、RPMsg框架与mbox client 进行关联。  
实验代码简述

这里我们就简单讲解一下M4 核端的代码和一些概念，更详细的内容则需大家自己研究了。

rpmsg 框架下有通信端点的概念，数据在两个端点间传输。端点间的数据传输是rpmsg 框架下数据传输最原始的形式，我们可以在原始的数据传输形式上再做一层封装，抽象出一些特定类型的设备。

每个端点注册的底层实现，就是一个内核设备的注册（使用的是平台总线模型），故注册的端点设备，可以利用到驱动的Probe 功能（具体实现详见drivers/rpmsg/rpmsg\_core.c 300 行后内容）。

在M4 端，通过调用openamp 库中的OPENAMP\_create\_endpoint 函数，并在调用时指定参数name（即为设备名称），即可在内核中注册一个对应的rpmsg 框架平台设备，该设备最终可以通过name（设备名称）来匹配到相应的A7 端内核驱动：

在这里插入图片描述

所以Linux rpmsg 框架下使用平台总线模型与端点通讯的方式结合，给一些需要有特殊操作的自定设备，提供了支持的可能。比如异构间的通讯，可以封装成串口通讯模型。

在我们提供的M4 内核固件的代码中，注册了两种Linux 内核自带的rpmsg 框架下，原生支持的设备模型，这两种设备类型是rpmsg-tty-channel、rpmsg-client-sample：

rpmsg-tty-channel：tty 终端设备，对应内核驱动源码drivers/rpmsg/rpmsg\_tty.c，此  
驱动模块默认被编译进内核。  
rpmsg-client-sample：框架原生的通讯方式测试设备（放在内核里作为演示该框架的Demo提供的），对应内核驱动源码samples/rpmsg/rpmsg\_client\_sample.c，此驱动默认被编译成模块，并放置在文件系统/lib/modules/4.19.94-stm-r1/kernel/samples/rpmsg/rpmsg\_client\_sample.ko 中，当设备与驱动发生匹配时，系统会  
自动insmod 该驱动模块。  
还有一种字符设备模型，rpmsg\_chrdev，源码位于drivers/rpmsg/rpmsg\_char.c ，我  
们的代码中未实验，可自行研究。

在M4 核的代码中，还初始化了usart3 作为M4 内核的Log 输出串口，我们可以通过串口模块接入开发板上的usart3，来查看M4 内核输出的Log。最终工程代码会被用于生成ELF 固件，ELF固件即为程序，会运行在MP157 的M4 内核上。

综上，通过原生的rpmsg 框架设备、/dev/ttyRPMSGx 节点以及M4 内核使用的usart3 资源，我们就可以进行简单的实验了。本实验的代码也比较简单，这里就讲解这么多。  
实验准备

由于多核异构的框架是与处理器的架构紧密联系在一起的，所以一般这些框架驱动会由芯片厂商为我们提供好。野火MP157 开发板默认开启了这些驱动支持，并且开启了对应的设备树，我们直接进行使用就可以了。

在前面我们提到了，M4 内核要与A7 内核通讯需要共用一个框架，那么M4 内核的运行的程序里，就需要有对应的框架代码，这个为大家提供的工程中已经包含。最终我们将代码生成的ELF固件，通过A7 内核的remoteproc 子系统加载到M4 内核上，即可做好前期的准备工作。

生成ELF 固件的工程代码位于\linux\_driver\framework\_ipcc\STM32Cube\_FW\_MP1\_V1.2.0\Projects\STM32MP157C-EV1\Applications\OpenAMP\OpenAMP\_raw 目录下，感兴趣可自行研究，工程可用MDK 或CubeIDE 打开（在工程目录中由对应文件夹）。

重要: 在M4 核的代码中，还初始化了usart3，实验前请务必将usart3 的设备树插件关闭。  
实验操作

M4 核的固件我们已经成功编译并放在了/linux\_driver/framework\_ipcc 目录下，我们将M4 核的固件OpenAMP\_raw\_CM4.elf 上传至Linux 文件系统的/lib/firmware/ 目录。此目录存放着Linux 系统中会使用到的各种固件。

执行如下命令指定M4 内核加载的固件，默认在root 用户下操作：

# 进入remoteproc 子系统目录

cd /sys/class/remoteproc/remoteproc0

# 导入M4 内核固件名称

echo OpenAMP\_raw\_CM4.elf > firmware

1  
2  
3  
4

在同一目录下，执行如下命令可启动停止M4 内核：

# 启动M4 内核

echo start > state

# 停止M4 内核

echo stop > state

1  
2  
3  
4

启动M4 内核后信息如下：

在这里插入图片描述

M4 内核加载固件并启动后，在串口终端中打印出了一些信息，我们通过串口模块接入usart3引脚，再打开串口调试助手设置波特率为115200，可以看到M4 固件初始化的usart3 作为串口printf 出来的信息，为[INFO ]M4 send to A7 : hello world! ，并且A7 端的驱动也打印出了rpmsg\_client\_sample virtio0.rpmsg-client-sample.-1.0: incomingmsg 1 (src: 0x0) 说明M4 核及A7 核驱动正常工作了。

此外， 输入lsmod， 我们还可以看到演示设备创建后， 对应被动态加载的驱动模块，rpmsg\_client\_sample ：

在这里插入图片描述

下面，我们进行第二个设备测试，通过前面现象中的LOG，我们可以看到被枚举出的tty 设备节点/dev/ttyRPMSG0 节点，我们就通过该节点测试tty 设备的功能，输入如下命令：

echo "hello M4 core , i'm A7!" > /dev/ttyRPMSG0

1

实验现象如下所示：

在这里插入图片描述

上图为A7 通过虚拟的tty 终端设备，转发到M4 内核的消息内容，最终通过M4 核固件的串口Log 功能打印出来对应信息。

自此，所有实验结束。  
————————————————  
版权声明：本文为CSDN博主「PicassoCao」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：<https://blog.csdn.net/picassocao/article/details/129097784>

## 8. OpenAMP

通常在AMP（非对称多处理）配置中，会采用在不同的处理核上运行不同的软件环境并执行各自的代码程序，各核心之间通力合作实现处理器性能的提升。在AMP系统中，所谓的主处理器通常是指最先启动且主要负责管理其他CPU以及这些CPU上软件环境的CPU处理器。而远程处理器简单的讲就是指被主核所支配的CPU。主软件环境通常需要进行远程核心的管理以及任务调度，在需要进行计算任务分配时，将选择性地驱动远程内核上的远程软件环境，并交代任务信息。

OpenAMP软件框架为开发AMP系统提供了必要的API函数。OpenAMP 是Mentor Graphics 与赛灵思公司为了使在AMP系统的设计中开发出的RTOS和裸机程序能够与开源Linux社区提供的接口进行互通讯，而共同通过的一个标准化的嵌入式多核框架。的全称是 ,即开源的非对称多处理框架。OpenAMP框架提供了非对称多处理系统软件开发所需的软件组件。OpenAMP框架是一种软件框架，这种软件框架能够为非对称多处理(AMP)系统开发人员提供三大重要组件：

l 该组件是RPMsg组件的实现基础，在使用Remoteproc和RPMsg进行核间通信时使用到了这个概念。在后面会讲解Virtio的实现原理。

l :该组件用于主机上，实现对远程处理器及其相关软件环境进行生命周期管理(LCM)，说白了就是能够对远程处理器实现启动和关闭、加载远程处理器运用程序到内存适合位置，并完成virtio和RPMSG设备的注册等。

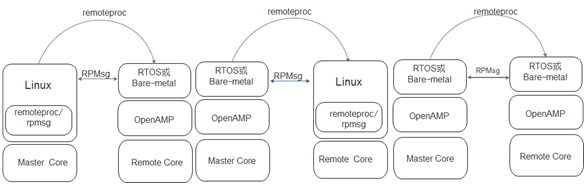
l ：实现多核处理器IPC通信的通道，基于virtio组件实现。

软件框架为开发人员提的标准化接口使得在AMP结构中设计和管理软硬件变得更为简单。和基础架构最初是由德州仪器设计开发的，并专门用在内核。使用该基础架构可以实现在主软件环境下对远程软件环境进行生命周期管理并进行核心间的通讯。但是，所提供的这些基础架构在一定程度上存在着局限性。首先，基础架构通常都是默认将作为其主操作系统，然而并不支持把作为非对称多处理结构配置中的远程软件环境。另外，和API只能从系统的内核空间中获得，由于缺乏相应的等效API或库，这些组件并没能用在其他的一些操作系统上，这造成了适用范围上的局限性。

软件框架很好的解决了以上的问题。该框架是和赛灵思共同通过地一个开源项目，其源代码可以在GitHub（<https://github.com/OpenAMP/open-amp>）上获得或从该文档相同目录下src文件夹中获得（open-amp-openamp\_zynq\_kernel\_3.17）。

框架能够被很好地使用在实时操作系统或者裸机软件环境中，使得在这些软件环境下能够使用与Linux系统具有相同功能的和，且之间在使用上具有很好的兼容性。框架中的和基础架构使用VirtIO作为传输抽象层。下图显示了框架支持的各种软件环境及配置，由图所示，该框架可以被用在、或者裸机环境下，通过和组件实现不同软件环境之间的通讯。

嵌入式多核框架同样可以作为独立库使用。允许主处理器RTOS或者裸机软件环境中的程序对远程处理器进行生命周期管理并且使用与远程处理器进行通信。



该系列主要基于Linux master/FreeRTOS remote配置，讲解如何使用OpenAMP框架实现多核处理器核间通信，包括开发环境的搭建、工程的建立和编译等，并在需要的时候介绍所涉及的Virtio、RPMsg组件的工作原理。在开始设计开发之前，首先需要搭建相应的开发环境。

<https://www.cnblogs.com/jiandahao/p/9310978.html>

# 符录

# 参考文献

# 索引

操作系统宏内核和微内核区别 <https://www.bilibili.com/read/cv23329414/>

4岁小女孩修改一个字符获网友狂点赞<https://www.linuxprobe.com/four-years-linux.html>

操作系统的四个基本特征是什么:<https://www.php.cn/faq/495187.html>

实时性操作系统和非实时性操作系统的概念及区别<https://blog.csdn.net/Dinh_ding/article/details/125780067>

车载实时操作系统与非实时操作系统<https://zhuanlan.zhihu.com/p/84148986>

Linux 中有哪些不同类型的 Shell？<https://zhuanlan.zhihu.com/p/614551538>

LINUX过时了 <https://www.cnblogs.com/coder-dreamer/archive/2013/01/11/2856642.html>

Linux——用户、用户组管理和文件权限（8）<https://zhuanlan.zhihu.com/p/424584330>

嵌入式保姆级教程-一文讲解出Linux的文件目录结构 <https://zhuanlan.zhihu.com/p/624152435>

Linux发展史 <https://www.cnblogs.com/haojinfei/articles/13652634.html>

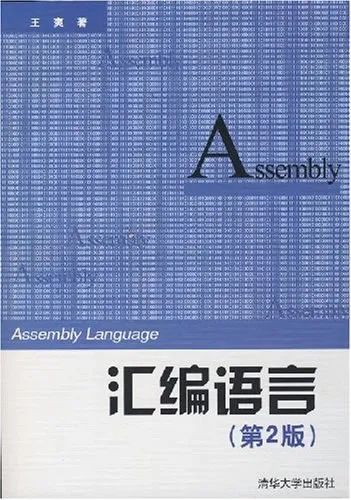
The UNIX Time-Sharing System <https://chsasank.com/classic_papers/unix-time-sharing-system.html>

GDB 简介 调试 使用实例 <https://zhuanlan.zhihu.com/p/265099138>

多线程编程指南:<https://docs.oracle.com/cd/E19253-01/819-7051/index.html>

有关与进程间通信的mmap方法以及进程间互斥锁的使用：<https://blog.csdn.net/wlqxyl/article/details/127426969>

Linux POSIX消息队列用于线程间通信：<https://zhuanlan.zhihu.com/p/638279470>

**王爽《汇编学习第二版》**

《嵌入式 Linux 应用开发完全手册.韦东山》

《MiniGUI 用户手册》

《ARM嵌入式MiniGUI初步于应用开发范例》

minigui基础篇——窗口和消息之窗口  
<https://blog.csdn.net/qq_43601873/article/details/127164896>

MiniGUI原理分析 <https://blog.csdn.net/lieye_leaves/article/details/8947165>

QT 主要模块 <https://zhuanlan.zhihu.com/p/656765687>

