



学院 荣誉 责任



嵌入式系统原理

Principle of Embedded System



合肥工业大学 · 计算机与信息学院

2023年6月

嵌入式系统的层次划分



学院 荣誉 责任



应用层

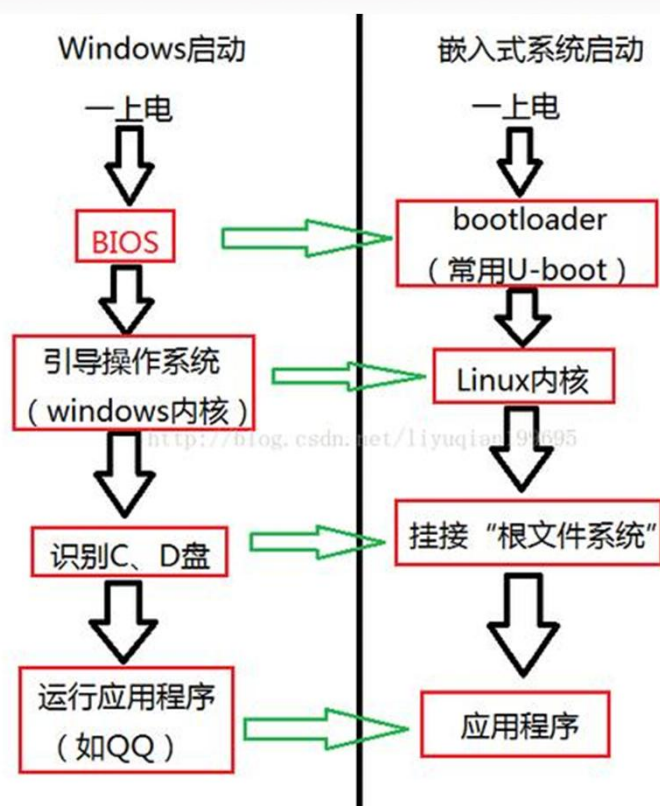
系统软件层

硬件层

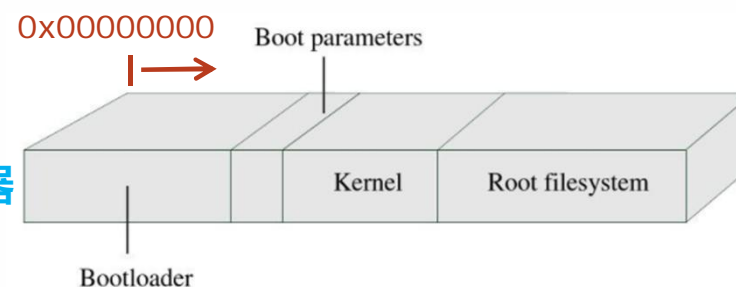
嵌入式系统的启动流程



学院 荣誉 责任



固态存储器





- 6.1 Bootloader过程
- 6.2 嵌入式操作系统简介
- 6.3 Linux终端命令
- 6.4 Shell编程
- 6.5 Linux编程基础

- Bootloader(**引导装载程序**)是嵌入式系统上电之后、**操作系统内核**运行之前运行的第一段程序。
- **用途**：对系统的硬件设备进行初始化，将操作系统映像(或固化的应用程序)装载到内存中，并建立内存空间的映射图，从而**为最终调用操作系统内核准备好正确的软硬件环境。**

应用程序

根文件系统

中间件

内核
驱动

Bootloader

硬件

■ Bootloader的实现严重依赖于**系统的硬件**，包括微处理器、嵌入式板级设备等。

- 依赖于微处理器架构：ARM、MIPS、RISC-V、x86等。
- 依赖于具体的板级配置：不同厂家的芯片、不同的内存空间。

应用程序

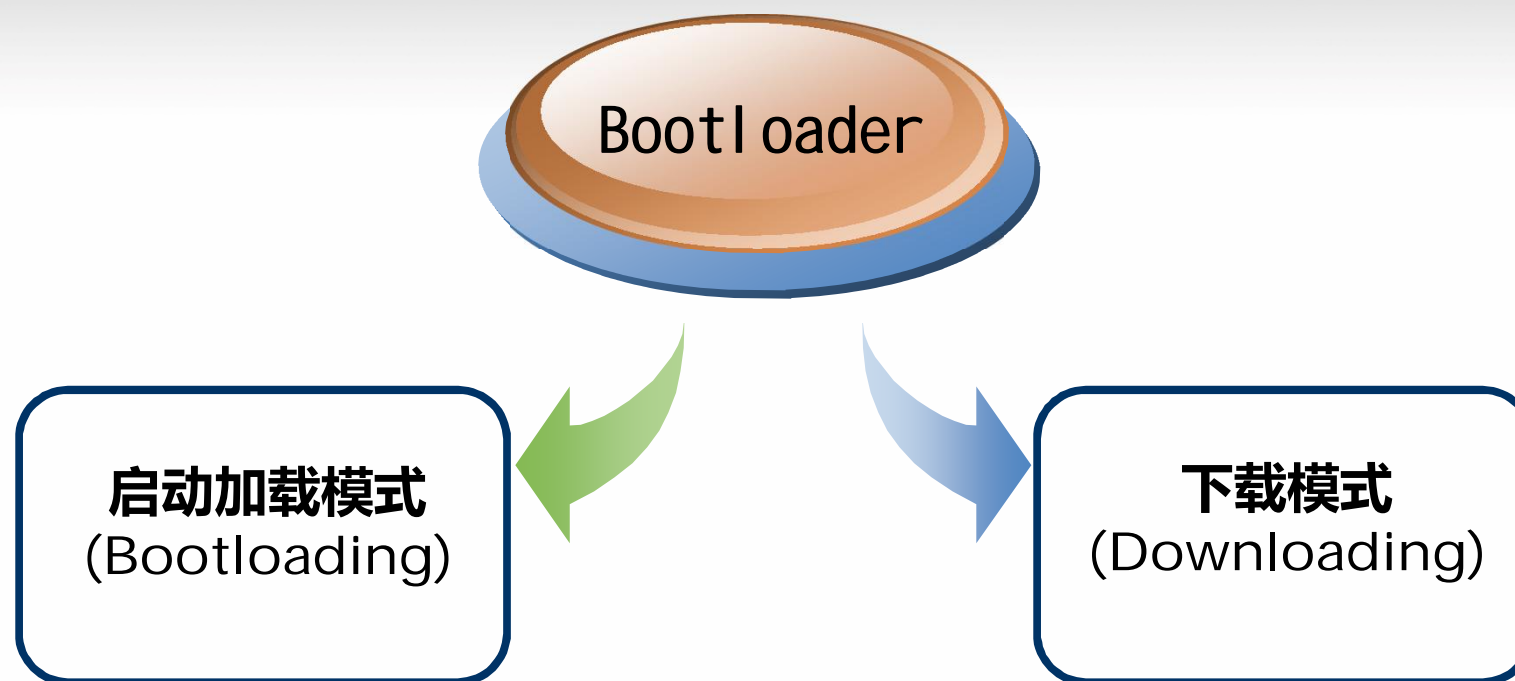
根文件系统

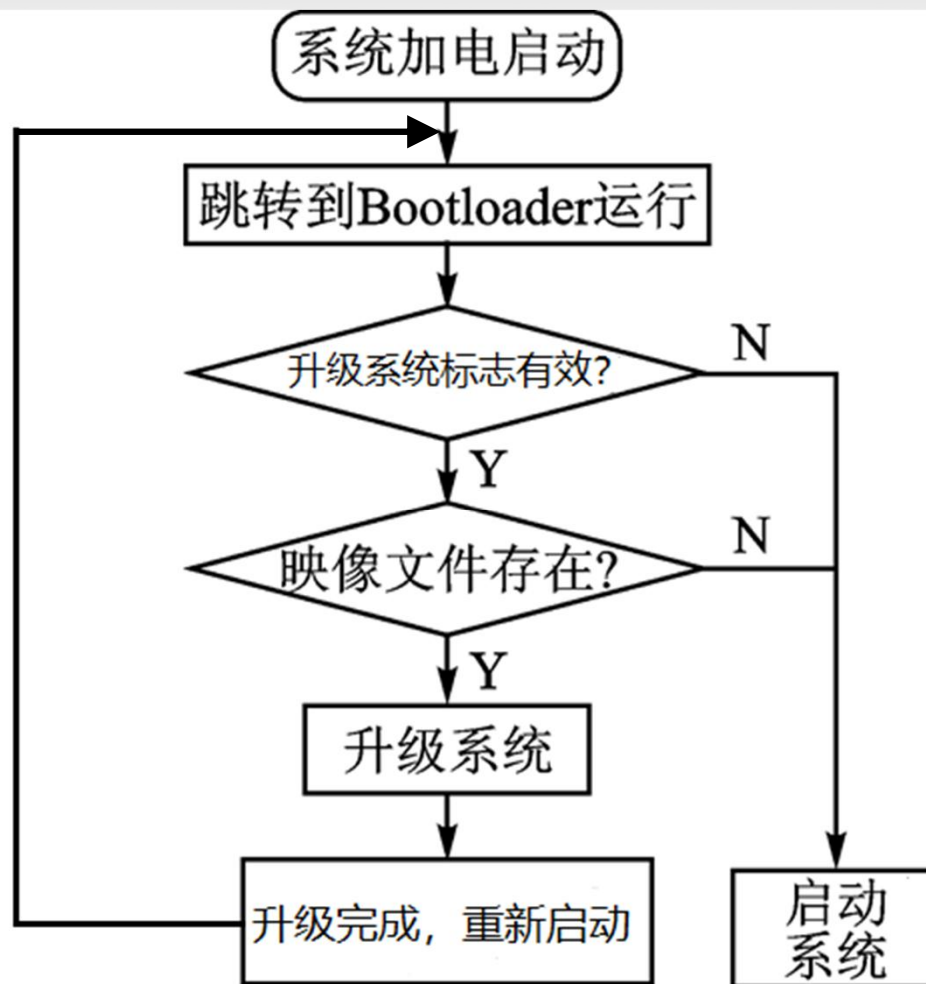
中间件

内核
驱动

Bootloader

硬件







(1) 启动加载模式(Bootloading)

- 也称自主(Autonomous)模式，是Bootloader的正常工作模式。
- 工作流程：
 - 准备好操作系统内核运行所需的环境和参数；
 - 从目标机自身的某个固态存储设备上将操作系统内核加载到 RAM 中；
 - 在RAM中运行操作系统内核。



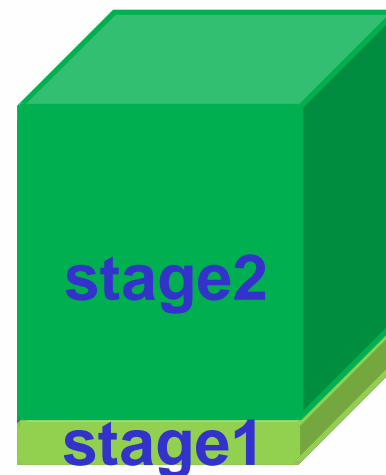
(2) 下载模式(Downloading)

- 用于系统调试或版本修改/升级：目标机(嵌入式计算机)上的Bootloader通过串口或网络等通信手段，从主机(通用计算机)下载文件到目标机的RAM中，然后写入目标机上的Flash类固态存储设备中。
- 通信设备与协议
 - 串口：xmodem / ymodem / zmodem协议中的一种。
 - 以太网：TFTP协议。
- 此模式下通常会向终端用户提供一个简单的命令行接口。

■ Bootl oader通常分为stage1和stage2两大部分。

- stage1——依赖于CPU体系结构的代码，例如设备初始化代码等。通常用汇编语言来实现，以达到短小精悍的目的。
- stage2——通常用C语言来实现，可以实现更复杂的功能，而且代码会具有更好的可读性和可移植性。

分级载入机制





■ stage1通常包括以下步骤(按执行的先后顺序):

- 硬件设备初始化。
- 为加载Bootloader的stage2准备RAM空间。
- 拷贝Bootloader的stage2到RAM空间中。
- 设置堆栈。
- 跳转到stage2的C程序入口点。

■ stage2通常包括以下步骤(按执行的先后顺序):

- 初始化本阶段要使用到的硬件设备。
- 检测系统内存映射(memory map)。
- 将内核(kernel)映像和根文件系统映像从Flash存储器读到RAM空间中。
- 为内核设置启动参数。
- 调用内核。

stage1的具体过程

1. **基本的硬件初始化**：为stage2的执行以及随后内核的执行准备基本的硬件环境。

(1) 屏蔽所有的中断

- 在 Bootloader执行的全过程中，可以不必响应任何中断；
- 中断屏蔽可通过设置微处理器内部**状态寄存器**的中断允许位来实现。

(2) 设置微处理器的速度和时钟频率

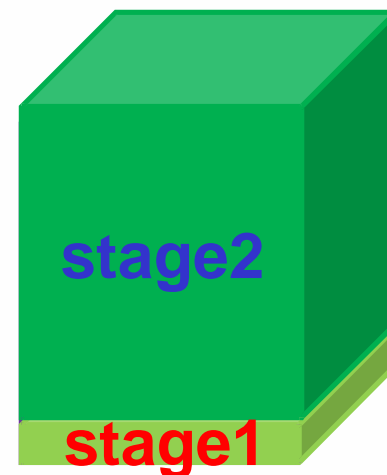
(3) RAM初始化

- 设置内存控制器的相关功能寄存器及控制寄存器等。

(4) 初始化LED

- 表明系统的状态（OK还是Error）。

(5) 关闭微处理器内部指令/数据Cache

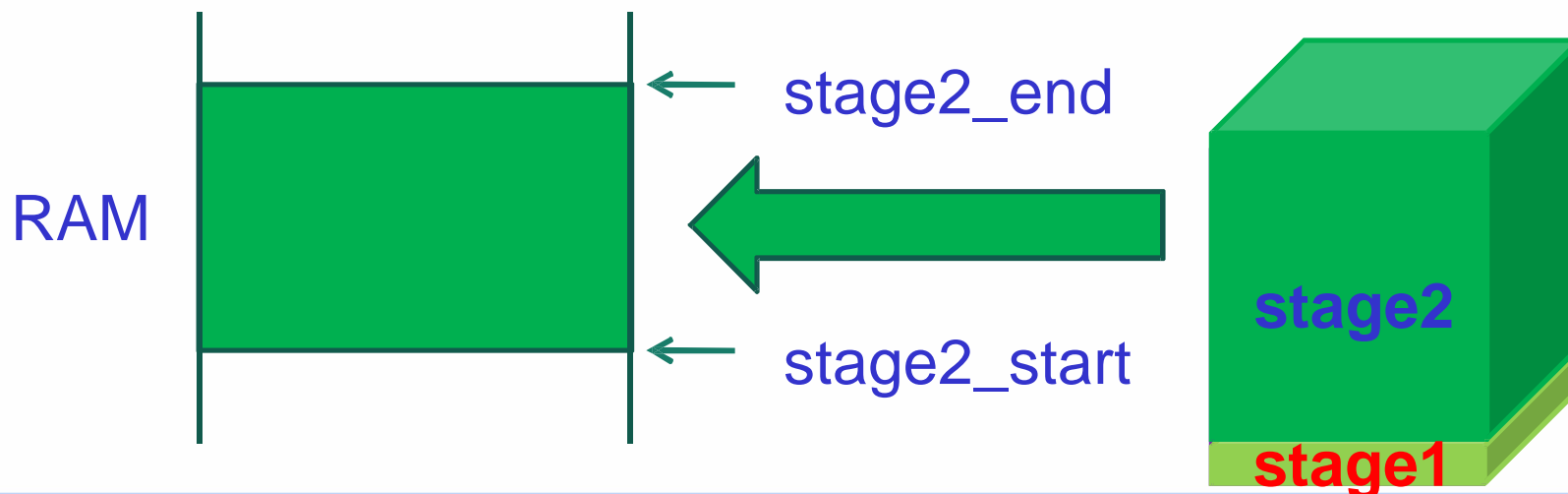


stage1的具体过程(续)

2. 为加载stage2 准备RAM空间

- 为了加快stage2的执行速度，通常需要把stage2加载到RAM空间中执行，因此必须为其准备好一段可用的RAM空间范围。

3. 拷贝stage2到RAM中



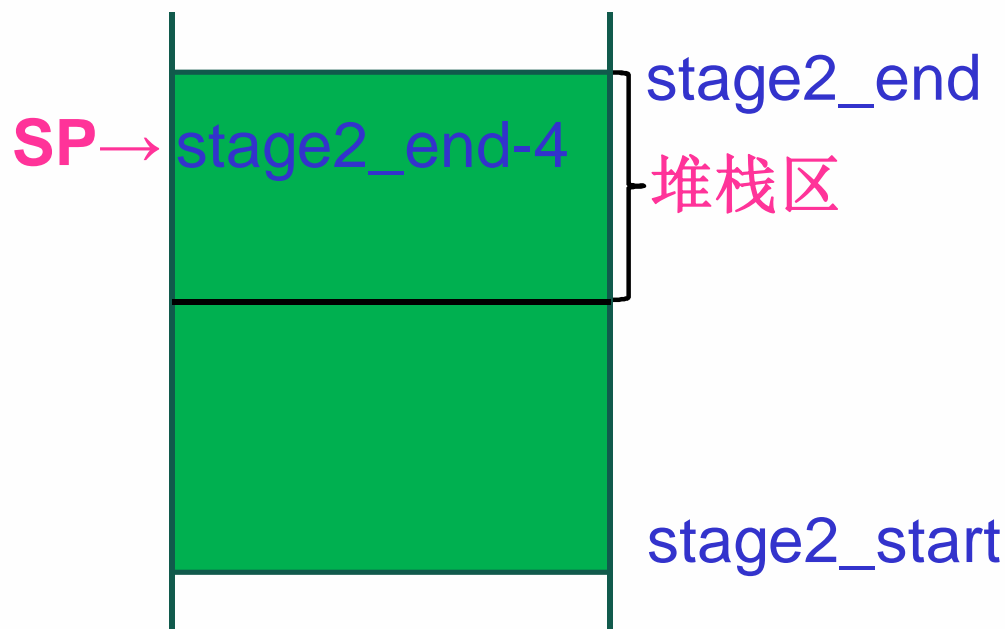
stage1的具体过程(续)

4. 设置堆栈指针SP，为执行C语言代码作好准备。

- SP指向stage2的RAM空间的最顶端；
- 向下生长型堆栈：SP指向stage2_end-4；
- ARM支持全部4种堆栈方式——满递增、空递增、满递减、空递减。

5. 跳转到stage2的C程序入口点。

- 对于ARM系统，可以通过修改PC寄存器为合适的地址来实现。





stage2的具体过程

1. 初始化本阶段要使用到的硬件设备

- 初始化至少一个串口，以便和终端用户进行I/O输出信息；
- 初始化定时器等外围部件(设备)；
- 设备初始化完成后，可以输出一些打印信息，例如：程序名称字符串、版本号等。

2. 检测系统的内存映射(Memory map)

- 在整个4GB物理地址空间中，有哪些地址范围被分配用来寻址系统的RAM单元。
- 不同微处理器有不同的内存映射，必须准确识别。



stage2的具体过程(续)

3. 加载内核映像(image文件)和根文件系统映像(image文件)

- 规划RAM的存储空间布局。
包括**内核映像**所占用的RAM范围；**根文件系统映像**所占用的RAM范围。
- 从Flash存储器上拷贝内核映像和根文件系统映像到RAM中。

4. 设置内核的启动参数

- 将内核映像和根文件系统映像拷贝到RAM空间中后，可以准备运行系统内核。但在调用内核之前，需要设置内核的启动参数。
- 方法：把启动参数封装在预定好的**数据结构**里，拷贝到RAM的某个地址，并通过通用寄存器R2向内核传递这个地址值。



stage2的具体过程(续)

5. 调用内核

- 直接跳转到内核的第一条指令处。
- 应满足跳转条件：

(1) 微处理器的寄存器的设置：

R0 = 0；

R1 = 机器类型ID；

R2 = 启动参数标记列表在RAM中起始基地址。

(2) 微处理器模式的设置：

必须禁止中断（IRQ和FIQ）；

必须处于SVC模式。 ← CPSR寄存器的控制位

(3) Cache和MMU的设置：

MMU必须关闭*； ← CP15的C1寄存器的[0]位=0

指令Cache可以打开也可以关闭；

数据Cache必须关闭。



- 常用的Bootloader有U-Boot、Vivi、Redboot、Lilo等。
 - U-Boot是德国DENX软件工程设计中心设计的Bootloader。
 - Vivi是韩国mizi公司专门为三星S3C2440芯片设计的Bootloader。



■ U-boot

- 支持NFS挂载、RAMDISK(压缩或非压缩)形式的根文件系统映像。
- 支持目标板环境参数多种存储方式，包括Flash存储器、NVRAM、EEPROM。
- 设备驱动：串口、SDRAM、FLASH、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI、RTC等驱动支持。
- 上电自检功能：
 - ✓ SDRAM、Flash存储器大小自动检测；
 - ✓ SDRAM故障检测；
 - ✓ 微处理器型号。



■ Vivi

➤ 阶段1 :

- ✓ 关WATCH DOG(disable watch dog timer)。
- ✓ 禁止所有中断(disable all interrupts)。
- ✓ 初始化系统时钟启动MPLL , FCLK=200MHz , HCLK=100MHz , PCLK=50MHz。
- ✓ 初始化内存控制寄存器 : S3C2440共有15个相关的寄存器 , 在此初始化其中13个寄存器。
- ✓ 检查是否从掉电模式唤醒(Check if this is a wake-up from sleep)。若是 , 则调用WakeupStart函数进行处理。
- ✓ 初始化UART0(set GPIO for UART & InitUART)。
- ✓ 跳到内存测试函数。
- ✓ 跳到bootloader的阶段2运行(get read to call C functions)。



■ Vivi(续)

➤ 阶段2 :

- ✓ 将内存清零。
- ✓ 对开发板进行初始化，包括初始化定时器和设置各GPIO引脚功能。
- ✓ 建立页表和启动MMU*。
- ✓ 申请一块内存区域。
- ✓ 封装驱动层提供的函数，为上层应用提供对存储设备(NOR Flash或者NAND Flash)操作的统一接口。



- 6.1 Bootloader过程
- 6.2 嵌入式操作系统简介
- 6.3 Linux终端命令
- 6.4 Shell编程
- 6.5 Linux编程基础



完全开源，软件资源
丰富，高性能，
稳定

VxWorks®



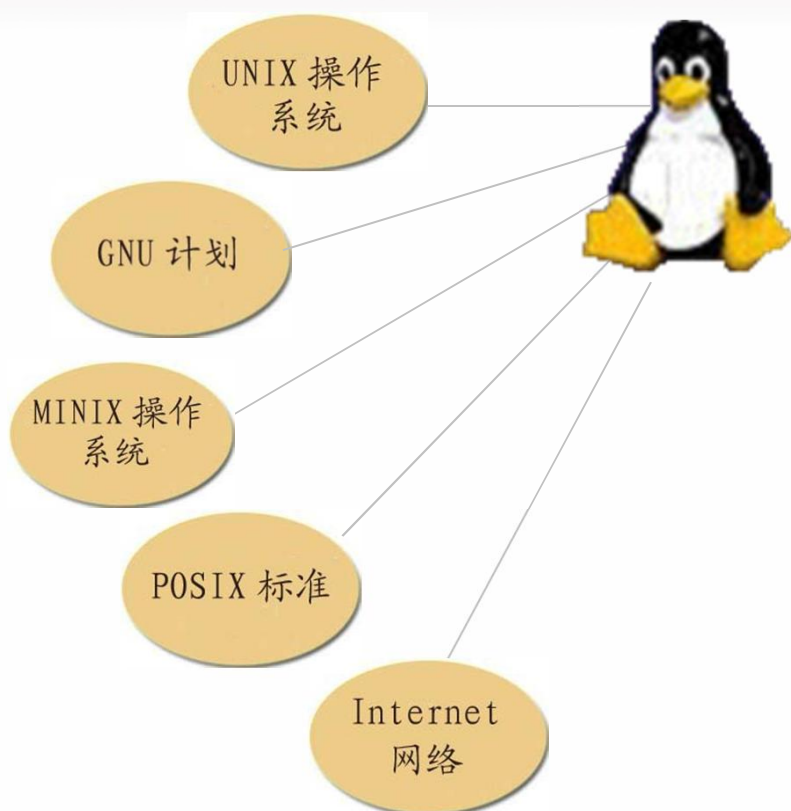
WIND RIVER

实时性强，价格高，
开发维护成本高



实时性、扩展性好，
对教育开源，仅包含
基本功能，无网络功
能，适合中小型应用

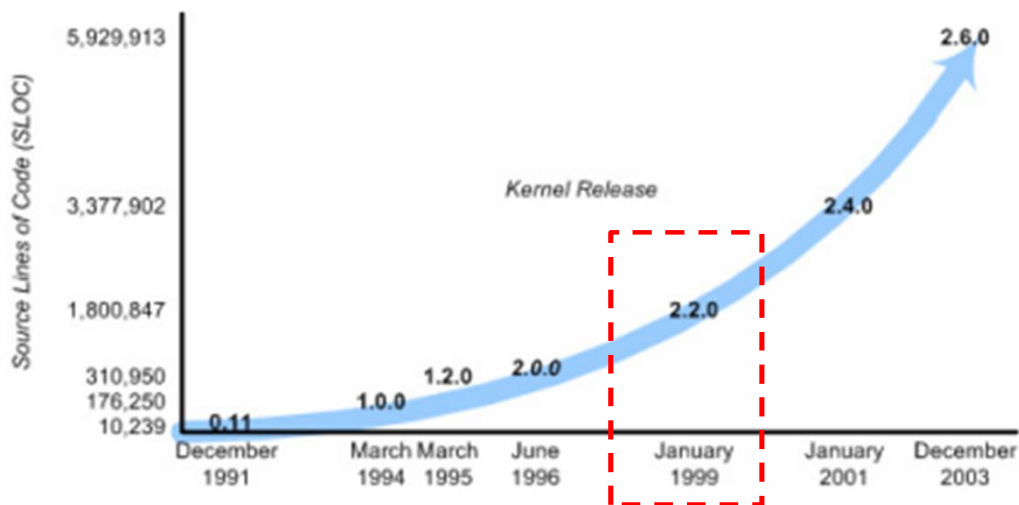




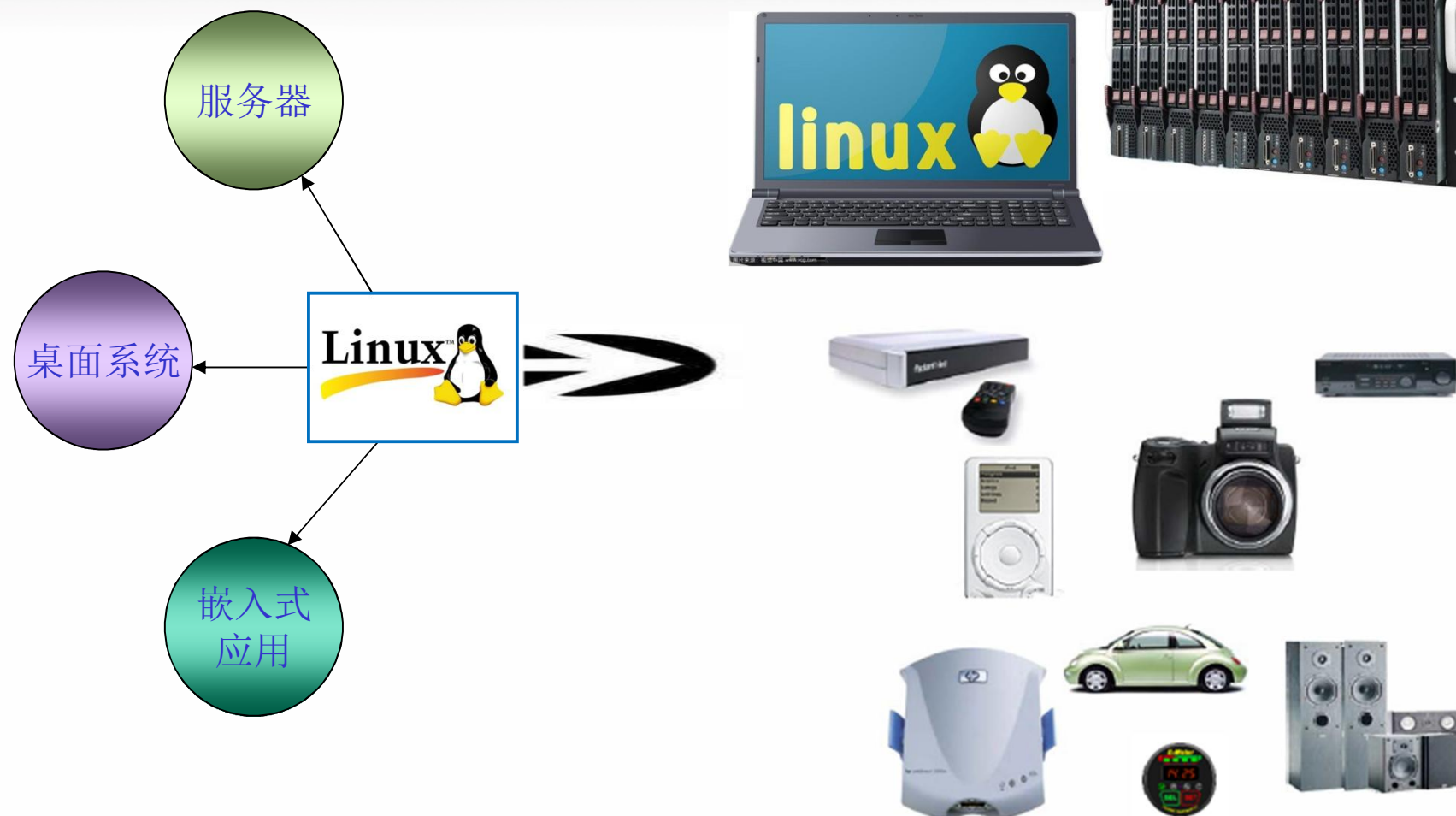
- UNIX操作系统是美国贝尔实验室于1969年夏在DEC PDP-7小型计算机上开发的一个分时操作系统。
- GNU 操作系统是由 Richard M. Stallman于1984年开发的，目标是创建一套完全自由的操作系统。——开源
- MINIX 操作系统是由 Andrew S. Tanenbaum于1987年开发的，主要用于学生学习操作系统原理。——开源
- Linux操作系统由Linus Torvalds于1991年开发，使Intel386或奔腾处理器的PC机上具有UNIX全部功能的操作系统。

- 从诞生开始，Linux内核就从来没有停止过升级。
- 从0.02版本到1999年具有里程碑意义的2.2版本，一直到今天看到的x.x.xx版本。

从未停止过升级！



- Linux内核版本有两种：
 - - 稳定版和开发版
- Linux内核的命名机制：
`num.num.num`.
 - - 第一个数字是主版本号
 - - 第二个数字是次版本号
 - - 第三个数字是修订版本号



- Linux的发展离不开GNU(GNU is Not Unix)计划(又称**革奴计划**)，是由Richard Stallman在1983年9月27日公开发起的，它的**目标是创建一套完全自由的操作系统**。
- GNU计划开发出许多高质量的免费软件，如GCC、GDB、Bash Shell等，**这些软件为Linux的开发创造了基本的环境，是Linux发展的重要基础。因此，严格来说，Linux应该称为GNU/ Linux。**

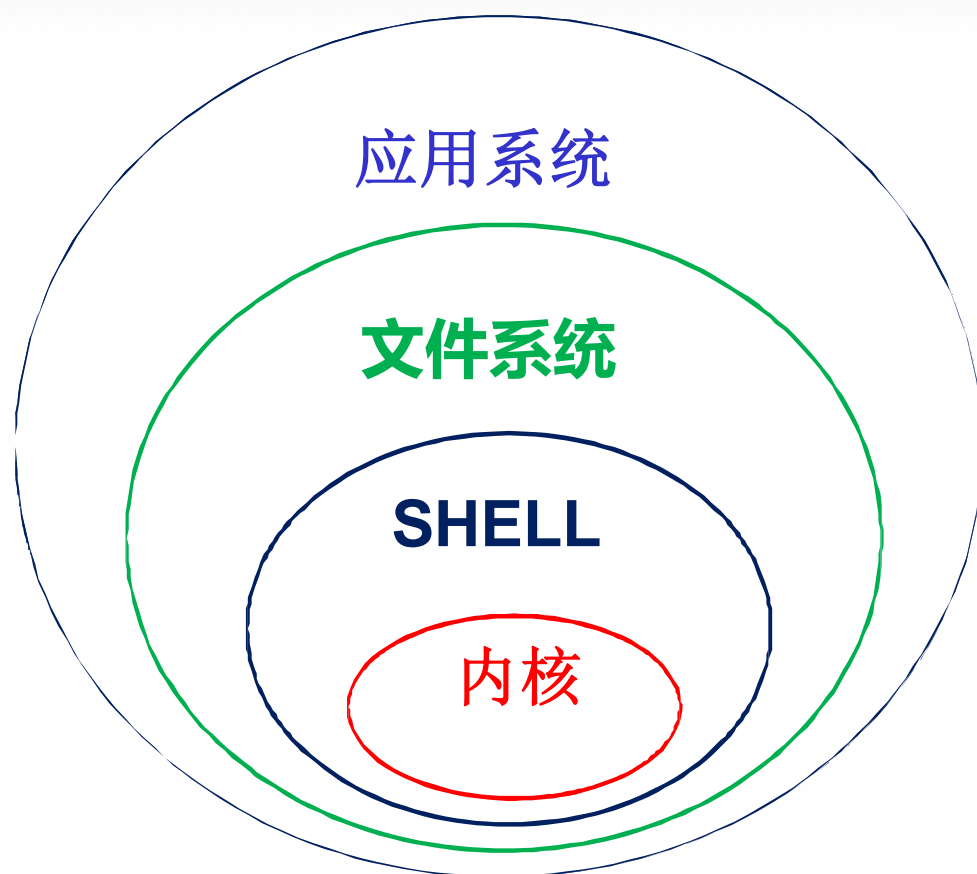
■ 一个典型的Linux发行版包括：

- Linux内核
- 一些GNU程序库和工具
- 命令行shell
- 图形界面和桌面环境，如KDE/GNOME
- 数千种从办公套件、文本编辑器到科学工具的应用软件

- Debian
- 红帽(Redhat)
- Ubuntu
- Suse
- Fedora
- CentOS

.....





应用程序的程序集，包括文本编辑器、编程语言、X Window、办公套件、Internet工具等

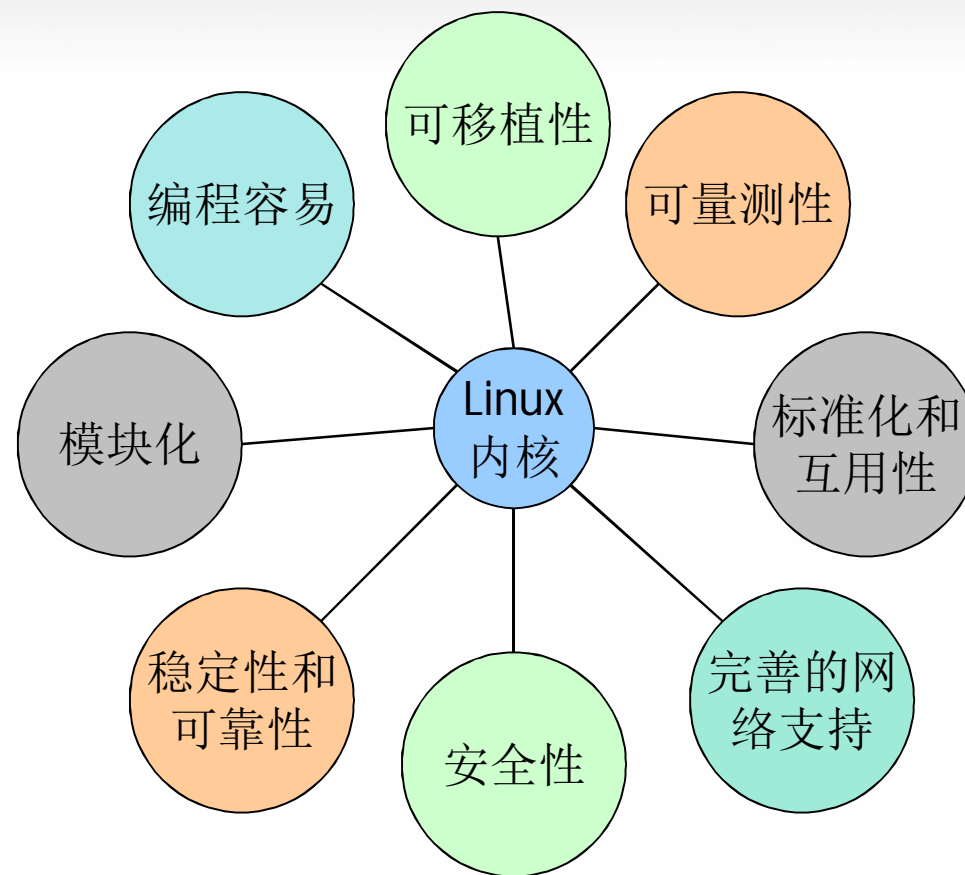
文件存放在磁盘等存储设备上的组织方法。支持ext4、NFS、ISO9660等

用户与内核交互的一种接口，可输入命令等

运行程序和管理硬件设备的核心，用于管理内存、CPU和其它相关组件。

■ 内核的任务

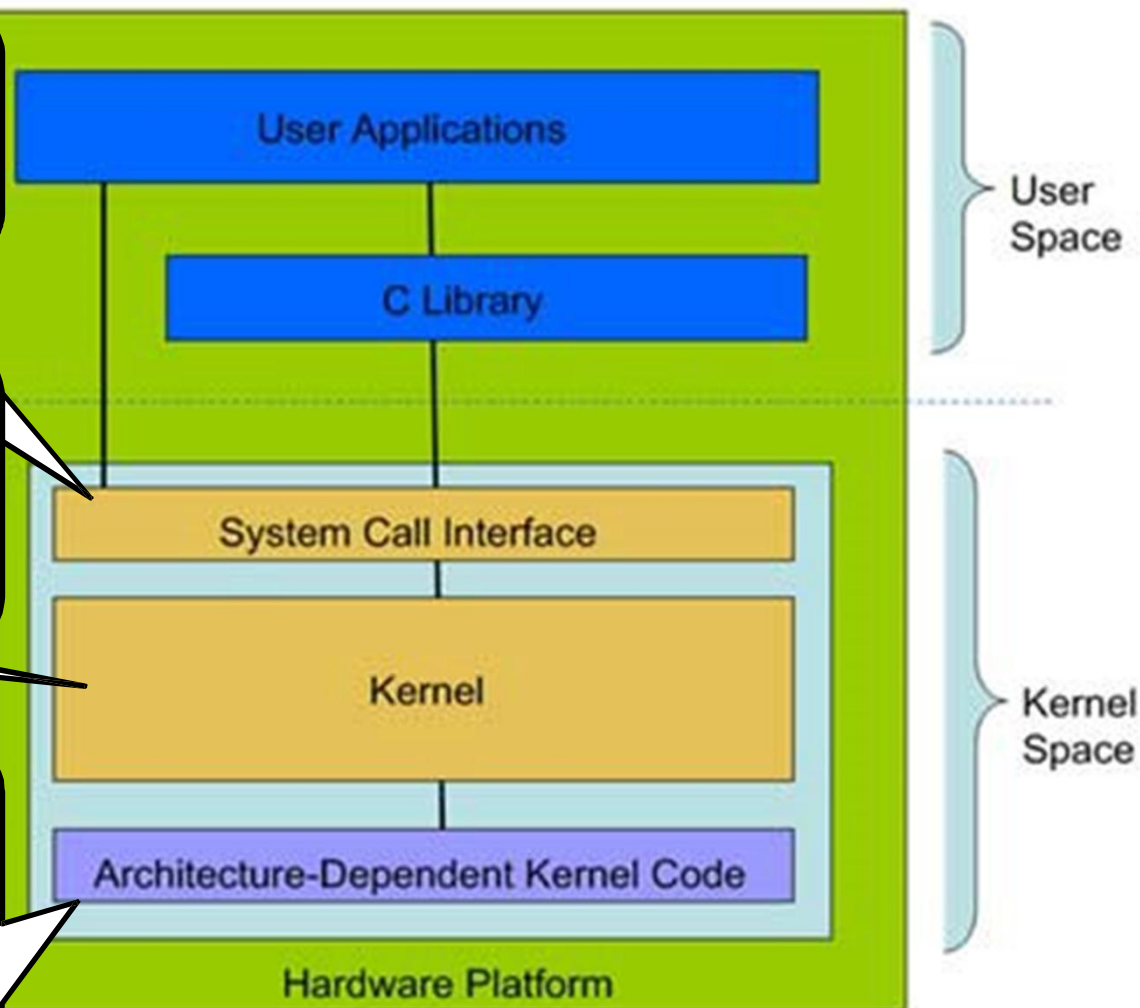
- 用于应用程序执行的流程管理。
- 内存和I / O管理。
- 系统调用控制——**内核的核心行为**。
- 借助设备驱动程序进行设备管理。



系统调用接口，它实现了一些基本的功能，例如read和write。

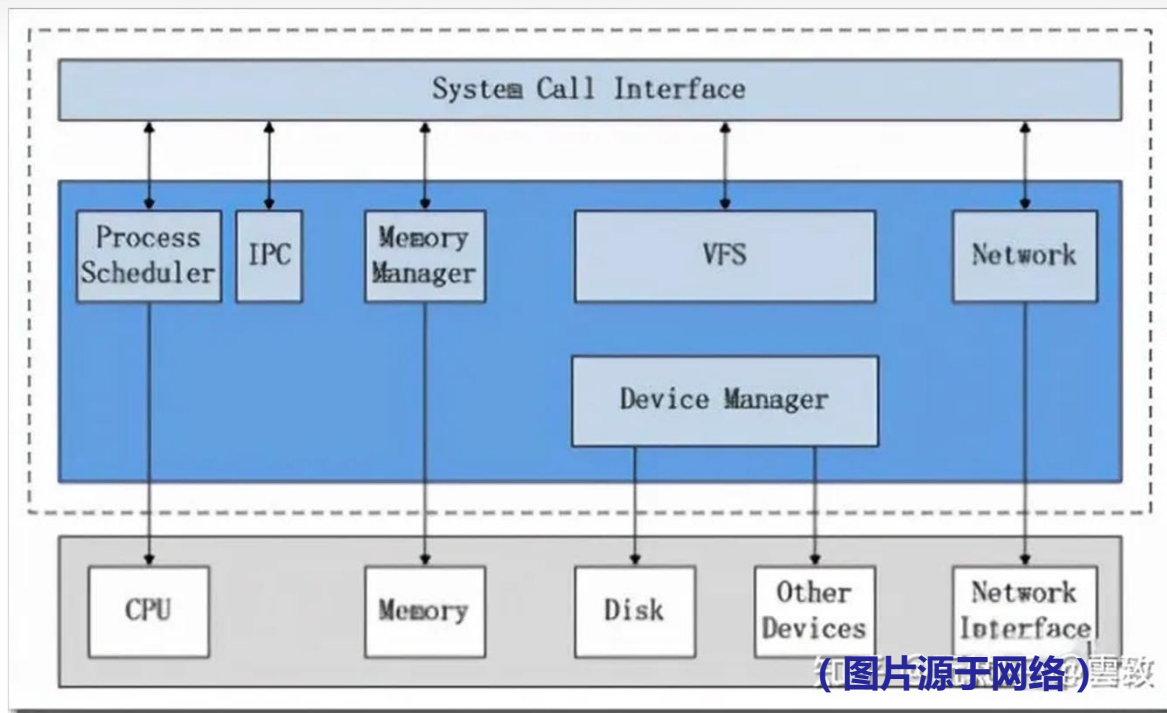
独立于体系结构的内核代码，是Linux所支持的所有处理器体系结构所通用的。

依赖于体系结构的代码，构成了通常称为BSP(Board Support Package)的部分。



■ 五个主要子系统

- 进程调度
- 进程间通信
- 内存管理
- 虚拟文件系统
- 网络接口



■ 其他部分

- 各子系统需要对应的设备驱动程序
- 依赖体系结构的代码

- (1) 下载内核及其关于ARM平台的补丁（如：linux- 3.4.6.tar.gz 和 patch-3.4.6.gz ）；
- (2) 给内核打补丁：zcat ../ patch-3.4.6.gz | patch ；
- (3) 准备交叉编译环境；
- (4) 修改相关的配置文件：如修改Makefile文件中关于交叉编译工具相关的内容，之后可以使用其进行编译；
- (5) 修改Linux内核源码：主要是修改与CPU相关的部分；
- (6) 内核裁剪：根据项目的需要裁剪内核模块；
- (7) 内核编译：将裁剪好的内核进行编译，生成二进制映像文件；
- (8) 将生成的二进制映像文件，烧写到目标平台。

内核烧写

■ 通过串口

➤ Minicom

■ 通过MicroUSB口

➤ MiniTools

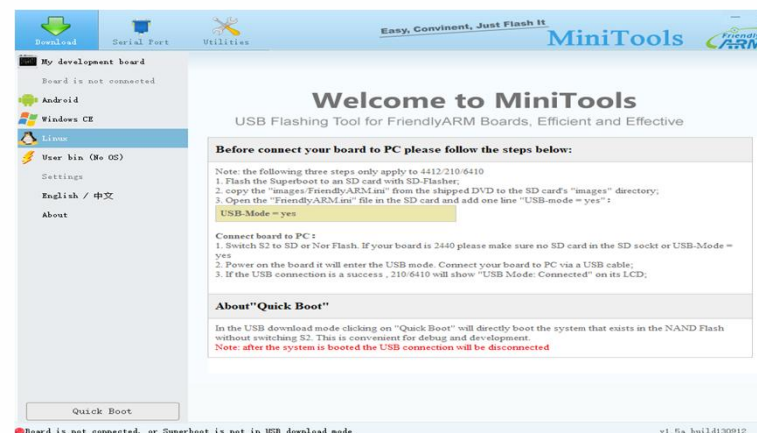
■ 通过SD存储卡

➤ Bootloader

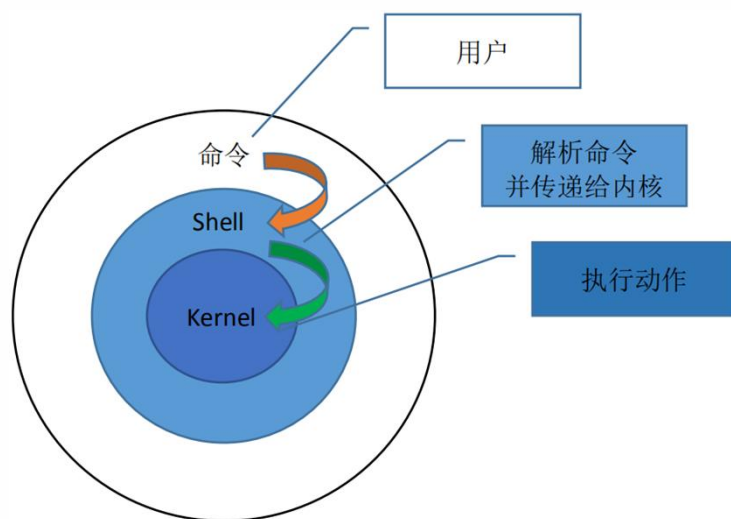
■ 通过网络

➤ Uboot下使用tftp烧写

■ 通过JTAG接口



- Shell是Linux系统的用户界面，提供了用户与内核进行交互操作的一种接口(命令解释器)。
- Shell接收用户输入的命令，并把它送入内核去执行。
- Shell起着协调用户与系统的一致性和在用户与系统之间进行交互的作用。

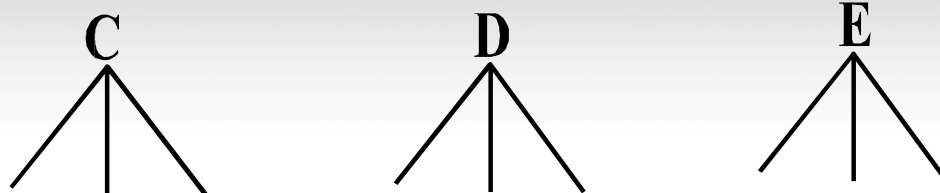


```
[root@lcl ~]# vim a.py
[root@lcl ~]# ./a.py
./a.py: line 3: print: command not found
[root@lcl ~]# mv ./a.py /usr/bin/a
[root@lcl ~]# a
/usr/bin/a: line 3: print: command not found
[root@lcl ~]# vim /usr/bin/a
[root@lcl ~]# cat /usr/bin/a
#!/usr/bin/python
print '2666'
[root@lcl ~]# a
2666
[root@lcl ~]# chmod a-x /usr/bin/a
[root@lcl ~]# a
-bash: /usr/bin/a: Permission denied
[root@lcl ~]#
```



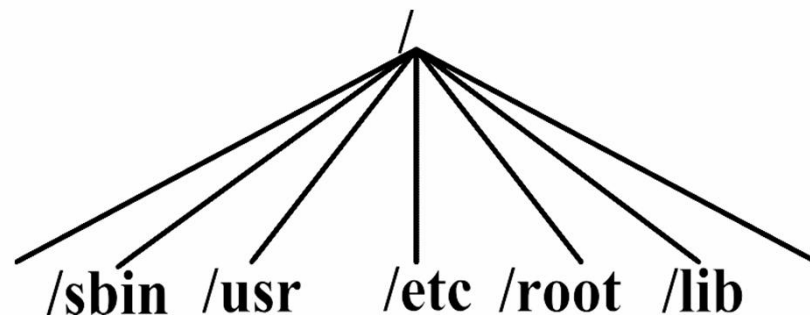
- **即文件系统**，是Linux操作系统中负责管理和存储文件信息的组件。
- **由三部分组成**：与文件管理有关的软件、 被管理的文件以及实施文件管理所需的数据结构。
- **Linux支持的文件系统**
 - FAT16、FAT32；
 - NTFS；
 - ext2、ext3、ext4
 - swap；
 - NFS；
 - ISO9660。

Windows



目录树结构：以每个分区为树根，有几个分区就有几个树型结构。

Linux

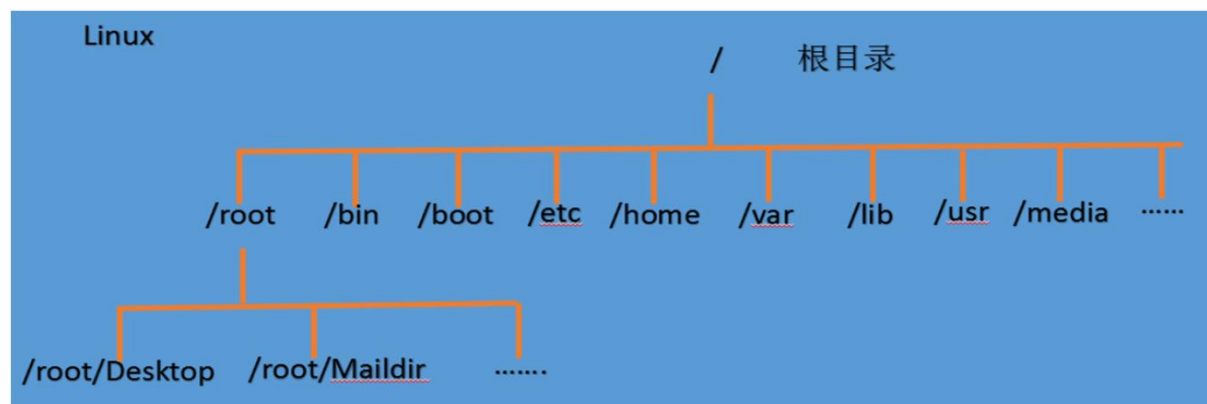


单一目录树结构，最上层是根目录，
所有目录都是从根目录出发而生成的。

**本质不同：Windows文件系统只负责文件存储；
Linux文件系统管理所有软硬件资源。**

■ Linux系统以**目录**的方式来组织和管理系统中的所有文件。

- 通过目录将系统中所有的文件**分级、分层组织**在一起，形成了Linux文件系统的**树型层次结构**。
- 以**根目录“ / ”**为起点，所有其他的目录都由根目录派生而来。
- 特殊目录：“ . ” 代表该目录自己，“ .. ” 代表该目录的父目录。对于根目录，“ . ” 和“ .. ” 都代表其自己。





- **工作目录**：用户登录到Linux系统后，始终处于某个目录之中，此目录被称为“工作目录”或“当前目录”。
- **用户主目录**(Home Directory)：是系统管理员在增加用户时为该用户建立的目录。每个用户都有自己的主目录，使用符号~表示。

```
[root@localhost ~]# cat /etc/hosts  
[root@localhost ~]# cat /etc/sysconfig/network  
[root@localhost ~]# cat /etc/redhat-release /proc/version
```



■ Linux系统中三种基本的文件类型

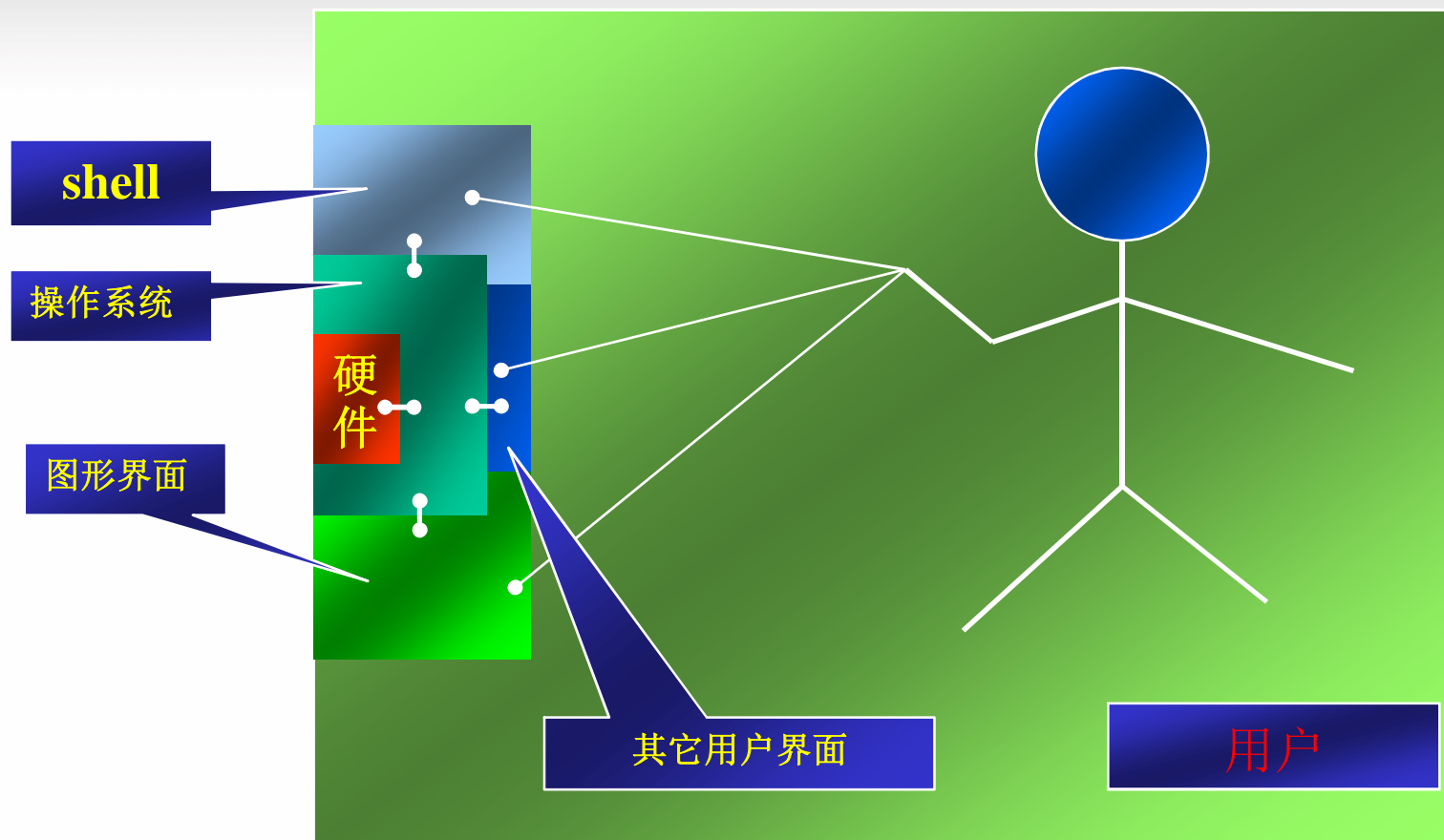
- **普通文件**：又分为文本文件和二进制文件；
- **目录文件**：目录文件存储了一组相关文件的位置、大小等与文件有关的信息；
- **设备文件**：Linux系统把每一个I/O设备都看成一个文件，与普通文件一样处理，这样可以使文件与设备的操作尽可能统一。

- **设备**是指计算机中的外围硬件装置，即除了CPU和内存以外的所有设备。
- 在Linux系统中，文件和设备都遵从**按名访问**的原则，用户可以通过使用文件的方法来使用设备。
- 设备名以文件系统中的设备文件的形式存在，所有的设备文件存放在`/dev`目录下。

```
[root@pinyoyougou-docker dev]# ls
agpgart      fd          nvram       stderr      tty23      tty41      tty6        vcs2
autofs       full        oldmem      stdin       tty24      tty42      tty60       vcs3
block        fuse        parport0    stdout      tty25      tty43      tty61       vcs4
bsg          hidraw0     port        tty         tty26      tty44      tty62       vcs5
btrfs-control hpet        ppp         tty0        tty27      tty45      tty63       vcs6
bus          hugepages  ptmx        tty1        tty28      tty46      tty7        vcsa
cdrom        initctl     pts         tty10       tty29      tty47      tty8        vcsa1
char         input       random      tty11       tty3        tty48      tty9        vcsa2
cl           kmsg        raw         tty12       tty30      tty49      ttyS0       vcsa3
console      log         rtc         tty13       tty31      tty5        ttyS1       vcsa4
core         loop-control sda         tty14       tty32      ttyS0      ttyS2       vcsa5
cpu          mapper      sda1        tty15       tty33      ttyS1      ttyS3       vcsa6
cpu_dma_latency mcelog     sda2        tty16       tty34      ttyS2      uhid        vfio
crash        mem         sg0         tty17       tty35      ttyS3      uinput      vga_arbiter
disk         midi        sgl         tty18       tty36      ttyS4      urandom     vhci
dm-0         mqueue     snapshot    tty19       tty37      ttyS5      usbmon0     vhost-net
dm-1         net         snd         tty2        tty38      ttyS6      usbmon1     vmci
dmideid      network_latency sr0         tty20       tty39      ttyS7      usbmon2     vsock
dri          network_throughput tty21       tty4        ttyS8      vcs        zero
fb0          null        sr0         tty22       tty40      ttyS9      vcs1
```



- 6.1 Bootloader过程
- 6.2 嵌入式操作系统简介
- 6.3 Linux终端命令
- 6.4 Shell编程
- 6.5 Linux编程基础





■ Shell的种类

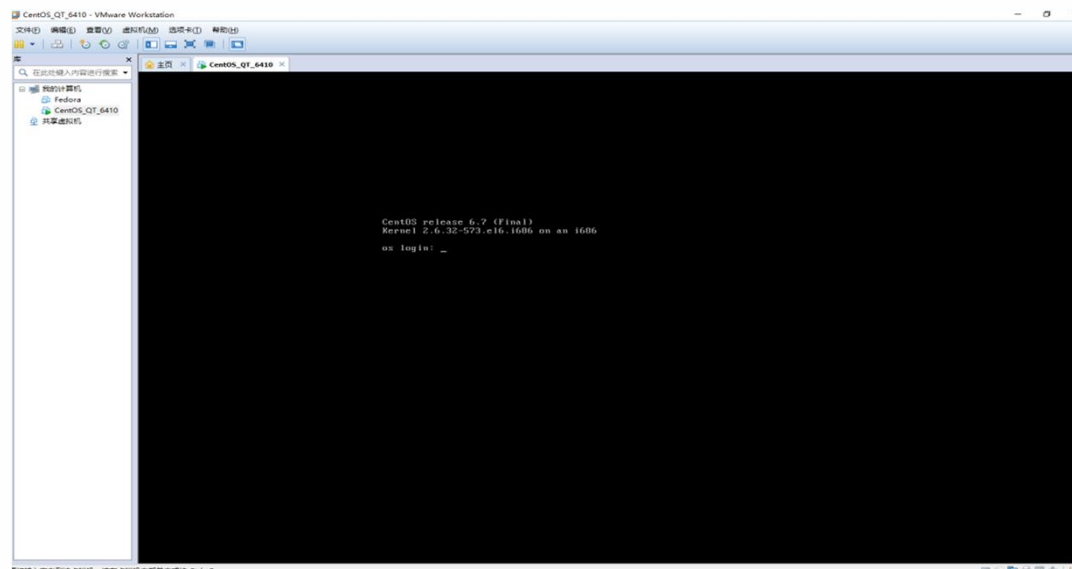
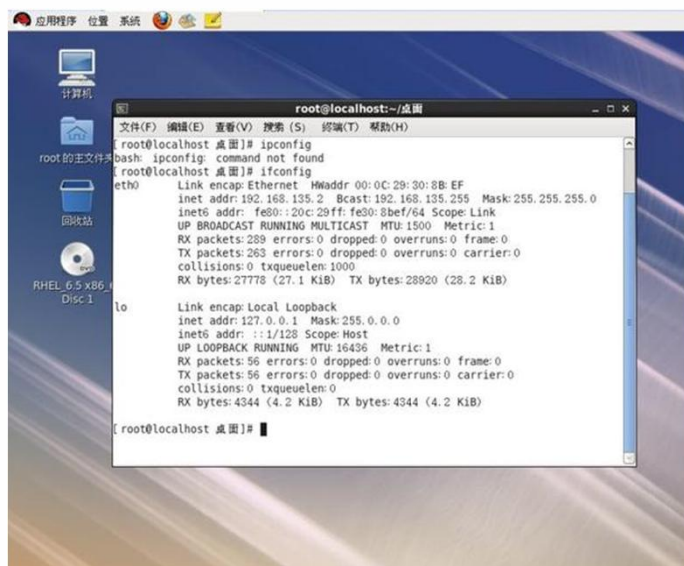
- ash : 贝尔实验室开发的shell。
- bash : GNU的Bourne Again shell , 是GNU默认的shell。
- tcsh : Berkeley UNIX C shell。

■ Shell的主要功能

- 命令解释器、命令通配符、命令补全、别名机制、命令历史。

■ Linux中执行Shell命令的方式

- 虚拟终端：图形界面的终端模拟器(terminal)。
- 进入与切换：Ctrl+Alt+(F1~F6)，返回：Ctrl+Alt+F7





■ 命令行技巧

- Tab键可补全命令名、命令参数选项、文件名；
- 上下键（UP↑、DOWN↓）调用命令历史记录。

■ 命令的终止

- 大部分命令，可用ctrl+c终止执行；
- 部分命令，例如man，可用“q”退出。

登录

■ 进入Linux系统，必须要进入用户的帐号。在系统安装过程中可以创建以下帐号：

- **root-超级用户帐号**：供系统管理员使用，可以在系统中执行包括管理在内的任何操作。
- **普通用户账号**：供普通用户使用，可以进行有限的操作。

■ 用户登录分两步

- 第一步：输入用户名；
- 第二步：输入用户的口令。用户正确地输入用户名和口令后，就能合法地进入系统，并执行各种操作。此时屏幕显示：

✓ [root@localhost /root]#

✓ 其中：超级用户的提示符是“ # ”，其他用户的提示符是“ \$ ”。

[root@localhost /root]#



Linux终端(Shell)命令格式

■ 命令名 [选项] [操作对象]

- 单字符选项前使用一个减号(-)，单词选项前使用两个减号(--).
- 多个单字符选项前可以只使用一个减号。
- 最简单的Shell命令只有命令名，复杂的Shell命令可以有多个选项。
- 操作对象可以是文件也可以是目录。有些命令必须使用多个操作对象，如文件复制命令cp必须指定源操作对象和目标操作对象。
- 命令名、选项和操作对象都作为Shell命令执行时的输入，它们之间用空格分隔开。

增加用户

■ useradd

➤ 格式：useradd [选项] 用户名

➤ 范例

✓ useradd hfut

✓ 采用默认值，添加名字为hfut的普通用户

选项	含义
-u UID	手工指定用户的 UID，注意 UID 的范围（不要小于 500）。
-d 主目录	手工指定用户的主目录。主目录必须写绝对路径，而且如果需要手工指定主目录，则一定要注意权限；
-c 用户说明	手工指定/etc/passwd文件中各用户信息中第 5 个字段的描述性内容，可随意配置；
-g 组名	手工指定用户的初始组。一般以和用户名相同的组作为用户的初始组，在创建用户时会默认建立初始组。一旦手动指定，则系统将不会在创建此默认的初始组目录。
-G 组名	指定用户的附加组。我们把用户加入其他组，一般都使用附加组；
-s shell	手工指定用户的登录 Shell，默认是 /bin/bash；
-e 日期	指定用户的失效日期，格式为 "YYYY-MM-DD"。也就是 /etc/shadow 文件的第八个字段；
-o	允许创建的用户的 UID 相同。例如，执行 "useradd -u 0 -o usertest" 命令建立用户 usertest，它的 UID 和 root 用户的 UID 相同，都是 0；
-m	建立用户时强制建立用户的家目录。在建立系统用户时，该选项是默认的；
-r	创建系统用户，也就是 UID 在 1~499 之间，供系统程序使用的用户。由于系统用户主要用于运行系统所需服务的权限配置，因此系统用户的创建默认不会创建主目录。

切换用户

■ su

➤ 格式：su [选项] [用户名]

➤ 范例

✓ su - root

✓ 切换到root用户，并将root的环境变量同时带入。

✓ su - root -c " useradd hfut"

✓ 以root用户执行添加hfut用户命令，但不切换到root用户。

选项	说明
-	选项只使用“-”代表连带用户的环境变量一起切换
-c 命令	仅执行一次命令，而不切换用户身份



关机

■ shutdown

- 格式 : shutdown [-t seconds] [-rkhncfF] time [message]
- 范例
 - ✓ shutdown now
 - ✓ 立即关机

清除屏幕

■ clear

- 格式 : clear

文件复制(拷贝)

■ cp

➤ 格式：cp [选项] 源文件或目录 目标文件或目录

➤ 范例

✓ cp /home/test /tmp/

✓ 将/home目录下test文件copy到/tmp目录下

✓ cp -r /home/lky /tmp/

✓ 将/home目录下的lky目录copy到/tmp目录下

- a: 归档模式，保留源文件或目录的所有属性，包括权限、所有者、时间戳等。
- f: 强制模式，不提示确认即覆盖目标文件或目录。
- i: 交互模式，覆盖目标文件或目录前询问是否确认。
- r: 递归模式，复制目录及其下所有文件和子目录。
- v: 详细模式，显示复制的每个文件或目录名称。



文件移动或更名

■ mv

➤ 格式：mv [选项] 源文件或目录 目标文件或目录

➤ 范例

✓ mv /home/test /home/test1

✓ 将/home目录下test文件更名为/test1

✓ mv /home/lky /tmp/

✓ 将/home目录下的lky目录移动(剪切)到/tmp目录下



文件删除

■ rm

- 格式：rm [选项] 源文件或目录
- 范例
 - ✓ rm /home/test
 - ✓ 将/home目录下test文件删除
 - ✓ rm -r /home/lky （等价于rm --recursive /home/lky ）
 - ✓ 将/home目录下lky目录中的所有文件和子目录（递归地）删除



创建目录

■ mkdir

- 格式：mkdir [选项] 目录名
- 范例
 - ✓ mkdir /home/test
 - ✓ 在/home目录下创建test目录

 - ✓ mkdir -p /home/lky/tmp/
 - ✓ 创建/home/lky/tmp目录，如果lky不存在，先创建lky



改变工作目录

■ cd

➤ 格式：cd 目录名

➤ 范例

✓ cd /home 或者 cd home

✓ 进入/home目录

✓ cd ..

✓ 返回上级目录

✓ cd /

✓ 进入系统根目录

初始位于根目录/

```
[Knight@localhost ~]$ cd home  
[Knight@localhost home]$ cd ..  
[Knight@localhost ~]$ cd /home  
[Knight@localhost home]$ cd /  
[Knight@localhost ~]$
```



查看当前路径

■ pwd

➤ 格式：pwd

➤ 范例

✓ pwd

✓ 显示当前工作目录的绝对路径

```
[root@qsync httpd]# pwd  
/var/log/httpd
```


查看目录

■ ls

➤ 格式：ls [选项] [目录或文件]

➤ 范例

✓ ls /home

✓ 显示/home目录下文件与目录(不包含隐藏文件)

✓ ls -a /home

✓ 显示/home目录下所有文件与目录(包含隐藏文件)

✓ ls -l /home

✓ 以列表的方式列出home目录下的文件与目录

||指令的别名

```
[root@os hello]# ls -l
总用量 20
-rwxr-xr-x. 1 root root 7816 6月 15 17:00 hello
-rwxrw-rw-. 1 root root 74 3月 4 2016 hello.c
-rw-r--r--. 1 root root 1032 5月 29 15:22 hello.o
-rwxrw-rw-. 1 root root 160 3月 4 2016 Makefile
```

打包与压缩

■ tar

➤ 格式：tar [选项] 目录或文件

➤ 范例

✓ tar cvf tmp.tar /home/tmp

✓ 将/home/tmp目录下所有文件与目录打包成一个tmp.tar文件

✓ tar xvf tmp.tar

✓ 将打包文件tmp.tar在当前目录下解开

✓ tar cvzf tmp.tar.gz /home/tmp

✓ 将/home/tmp目录下所有文件与目录打包压缩成一个tmp.tar.gz文件

✓ tar xvzf cvf tmp.tar.gz

✓ 将打包压缩文件tmp.tar.gz在当前目录下解开

访问权限

- 系统中的**每个文件和目录**都有访问许可权限，用来确定**谁**可以通过**何种方式**对文件和目录进行访问。文件或目录的访问权限分为**可读(r表示)**、**可写(w表示)**和**可执行(x表示)**三种。

对于目录：

- 可读：表示可以列出目录中有什么文件；
- 可写：表示可以在目录中删除和增加文件；
- 可执行：表示可以列出目录下文件的信息。

- 对文件或目录进行访问的三种用户类型：**文件所有者**、**与所有者同组的用户**、**其他用户**。所有者一般是文件的创建者。

访问权限(续)

■ 每个文件或目录的访问权限使用**三组(每组3位)**的方式表示：

- 第一组为文件所有者的读、写和执行权限；
- 第二组为与所有者同组的用户的读、写和执行权限；
- 第三组为系统中其他用户的读、写和执行权限。

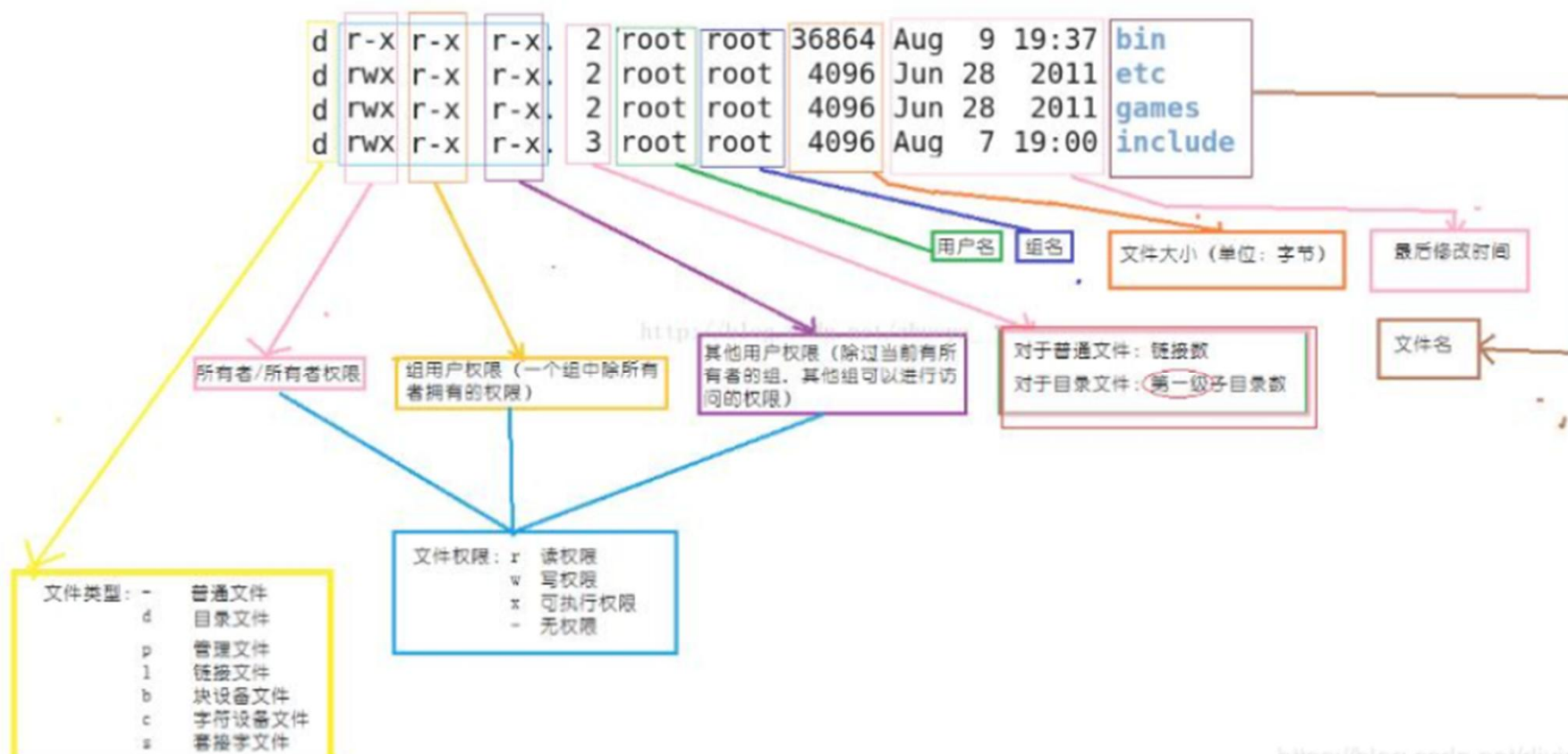

hello.c

所有者权限 **rw-** 所属组权限 **r--** 其他人权限 **r--**

-表示无权限

访问权限(续)

- 当用ls -l命令显示目录的详细信息时，最左边的一列为文件的访问权限。



改变访问权限

■ chmod

➤ 格式：chmod [who] [mode] [权限] 文件名

➤ 参数

✓ who

- u 表示文件的所有者
- g 表示与文件的所有者同组的用户
- o 表示其他用户
- a 表示所有用户——系统默认值

✓ mode :

- + 添加权限
- - 取消权限
- = 赋予权限

➤ 范例：

✓ chmod g + w hello.c

✓ 添加与hello.c文件所有者同组用户对该文件的写权限。

改变访问权限(续)

■ 权限可使用8进制数表示

➤ $r=4$, $w=2$, $x=1$, 分别以数字之和表示权限

➤ 范例:

✓ `chmod 761 hello.c`

✓ 文件hello.c的权限为：

①文件所有者对hello.c文件可读、可写、可执行；

②与所有者同组的用户对hello.c文件可读、可写、不可执行；

③系统中其他用户对hello.c文件不可读、不可写、可执行。

✓ `chmod 777 hello`

✓ 目录hello的权限为：所有用户都是可读、可写、可执行。

显示文字

■ echo

- 格式：echo [选项] <字符串>
- 选项：
 - ✓ -n：取消输出后行末的换行符号
 - ✓ -e：支持反斜线控制的字符转换
- 范例：
 - ✓ echo "hello world"
 - ✓ echo -n "hello world"
 - ✓ echo -e "\\hello world"

表 1 控制字符

控制字符	作用
\\	输出\本身
\a	输出警告音
\b	退格键，也就是向左删除键
\c	取消输出行末的换行符。和“-n”选项一致
\e	Esc键
\f	换页符
\n	换行符
\r	回车键
\t	制表符，也就是Tab键
\v	垂直制表符
\Onnn	按照八进制 ASCII 码表输出字符。其中 0 为数字 0，nnn 是三位八进制数
\xhh	按照十六进制 ASCH 码表输出字符。其中 hh 是两位十六进制数

```
Knight@localhost:~  
File Edit View Terminal Tabs Help  
[Knight@localhost ~]$ echo "hello world"  
hello world  
[Knight@localhost ~]$ echo -n "hello world"  
hello world[Knight@localhost ~]$ echo -e "\\hello world"  
\hello world  
[Knight@localhost ~]$
```




显示/设定日期时间

■ date

- 格式：date [选项] [+格式]
- 选项：
 - ✓ -d：按指定格式**显示**日期时间——缺省参数
 - ✓ -s：按指定格式**设置**日期时间
- 范例：
 - ✓ date +%F %T
 - ✓ 以YYYY-MM-DD hh:mm:ss的格式显示日期时间

 - ✓ date -s " 00:00:00 20230101"
 - ✓ 设置当前时间为2023年1月1日0点0分0秒



显示日历

■ cal

- 格式：cal [选项] [[月] 年]
- 选项：
 - ✓ 不添加选项，显示本月的日历
 - ✓ -y：显示指定年份的日历。若不指定，则显示当前年份的日历
 - ✓ -n：显示与本月相邻n个月的日历
- 范例：
 - ✓ cal
 - ✓ cal -y 2022 (=cal 2022)

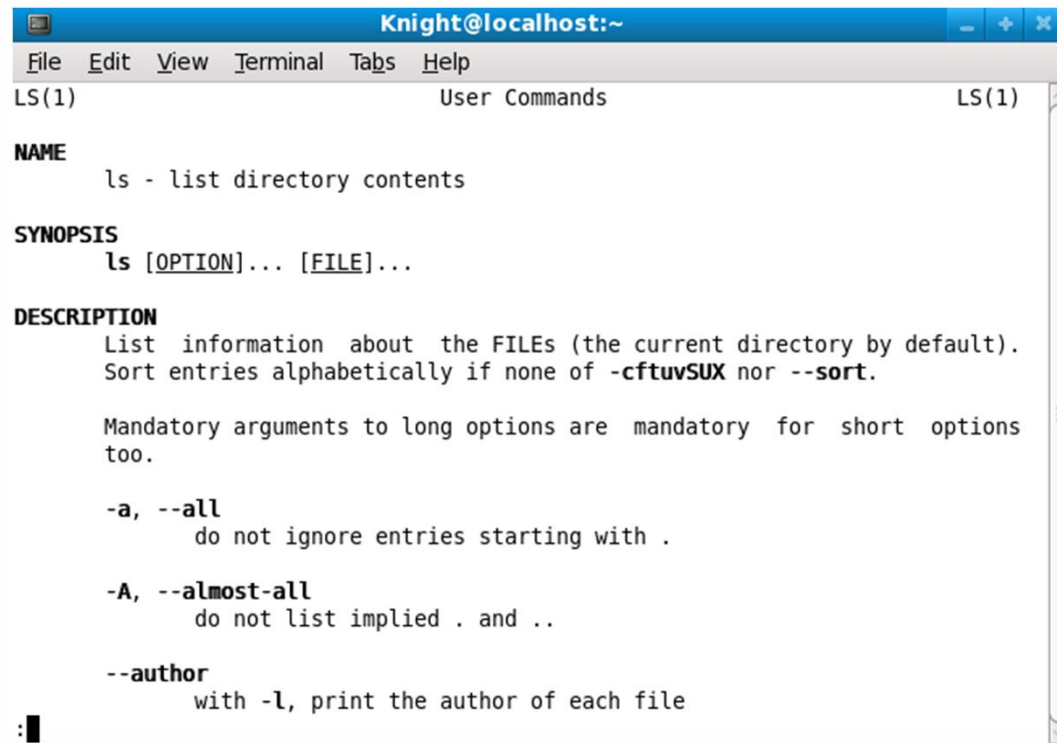
帮助

■ man

➤ 格式：man 命令件名

➤ 范例：

✓ man ls



```
Knight@localhost:~  
File Edit View Terminal Tabs Help  
LS(1) User Commands LS(1)  
  
NAME  
    ls - list directory contents  
  
SYNOPSIS  
    ls [OPTION]... [FILE]...  
  
DESCRIPTION  
    List information about the FILES (the current directory by default).  
    Sort entries alphabetically if none of -cftuvSUX nor --sort.  
  
    Mandatory arguments to long options are mandatory for short options  
    too.  
  
    -a, --all  
        do not ignore entries starting with .  
  
    -A, --almost-all  
        do not list implied . and ..  
  
    --author  
        with -l, print the author of each file  
  
:
```



- 6.1 Bootloader过程
- 6.2 嵌入式操作系统简介
- 6.3 Linux终端命令
- 6.4 Shell编程
- 6.5 Linux编程基础

■ Shell程序的特点及用途

- 可以认为是将Shell命令按控制结构组织到一个文本文件中，批量地交给Shell去执行。
- 解释执行，不生成可以执行的二进制文件。
- 不同的Shell解释器使用不同的Shell命令语法。
- 可以帮助用户完成特定的任务，提高使用、维护系统的效率，例如更好地配置和使用Linux 系统。

■ 一般结构

➤ Shell类型

➤ 函数

➤ 主过程

```
#!/bin/bash  
function fun1(){  
}  
.....  
function funn(){  
}  
.....
```

Shell 类型

函数定义

主过程



greeting.sh

1	<code>#!/bin/bash</code>
2	<code>#a Simple shell Script Example</code>
3	<code>#a Function</code>
4	<code>function say_hello()</code>
5	<code>{</code>
6	<code>echo "Enter Your Name,Please. :"</code>
7	<code>read name</code>
8	<code>echo "Hello \$name"</code>
9	<code>}</code>
10	<code>echo "Programme Starts Here....."</code>
11	<code>say_hello</code>
12	<code>echo "Programme Ends."</code>

解释

以 **#!** 开始，其后为当前使用的Shell类型

以 **#** 开始，其后为程序注释

同上

以 **functin** 开始，定义函数

函数开始

echo命令输出字符串

读入用户的输入到变量**name**

输出

函数结束

程序开始的第一条命令，输出提示信息

调用函数

输出提示，提示程序结束



■ 程序编译和运行过程

➤ 一般步骤

- ✓ 编辑文件
- ✓ 保存文件
- ✓ 将文件赋予可以执行的权限
- ✓ 运行及排错

➤ 常用到的命令

- ✓ vi : 编辑、保存文件
- ✓ ls -l : 查看文件权限
- ✓ chmod : 改变程序执行权限
- ✓ 直接键入文件名运行文件

Shell程序操作



学院 荣誉 责任



```
[tom@localhost ~]$ ll
```

查看权限

```
总用量 20
```

查看权限，初始状态无执行（x）权限

```
-rw-r--r--  1 root root    0
```

```
drwxr-xr-x  2 tom  tom  4096
```

增加可执行（x）的权限

```
-rw-rw-r--  1 tom  tom   210  6月 29 22:58 greeting.sh
```

```
[tom@localhost ~]$ chmod +x greeting.sh
```

```
[tom@localhost ~]$ ll
```

```
总用量 20
```

```
-rw-r--r--  1 root root    0  6月 20 19:32 abc.txt
```

```
drwxr-xr-x  2 tom  tom  4096  6月 19 01:23 Desktop
```

```
-rwxrwxr-x  1 tom  tom   210  6月 29 22:58 greeting.sh
```

```
[tom@localhost ~]$ ./greeting.sh
```

```
Programme Starts Here.....
```

```
Enter Your Name,Please.  :
```

```
tom
```

```
Hello tom
```

```
Programme Ends.
```

查看权限，已经具备执行（x）权限

运行程序

```
[tom@localhost ~]$
```

程序运行过程输出

- Shell编程中，使用变量无需事先声明
- 变量的赋值与引用：
 - 赋值：变量名=变量值（注意：不能留空格）
 - 引用：\$var #引用var变量
- Shell变量有几种类型
 - 用户自定义变量
 - 环境变量
 - 位置参数变量
 - 专用参数变量

- 由用户自己定义、修改和使用。默认赋值是字符串赋值。

```
var=1  
var=$var+1  
echo $var
```

#打印的结果是什么呢？

- 为了达到想要的效果(1+1=2)，有以下几种表达方式：

- let "var += 1"
- var=\${var+1}
- var=`expr \$var + 1` #注意加号两边的空格

■ 变量的引用

➤ 格式:

- ✓ 方式一：\$变量名；
- ✓ 方式二：\${变量名}

➤ 范例：

- ✓ a=1
- ✓ abc="hello"
- ✓ echo \$a
- ✓ echo \${abc}

- ✓ aa=Hello
- ✓ echo \$aa
- ✓ 输出：Hello
- ✓ aa=" Yes Sir "
- ✓ echo \$aa
- ✓ 输出：Yes Sir

- ✓ aa=7+5
- ✓ echo \$aa
- ✓ 输出：7+5

注：等号两边不能有空格。如果字符串两边有空格，必须加用引号。



■ echo

- 格式：echo arg
- 在屏幕上显示由arg指定的字符串。

■ export

- 格式：export 变量[=变量值]
- Shell可以用export把它的变量向下带入子Shell，从而让子进程继承父进程中的环境变量。
- 不带任何变量名的export语句将显示出当前所有的export变量。



■ read

- 格式：`read [-p "信息"] [var1 var2 ...]`
- 从键盘输入内容为变量赋值。
- 若省略变量名，则将输入的内容存入REPLY变量。

■ readonly

- 格式：`readonly 变量`
- 不能被清除或重新赋值的变量。



■ 双引号

- 双引号内的字符，除\$、` (倒引号)和\仍保留其特殊功能外，其余字符均作为普通字符对待。

■ 单引号

- 由单引号括起来的字符都作为普通字符出现，即使是\$、`和\。

■ 特殊符号

- **\$**表示变量替换或表达式计算。
- **` (倒引号)**表示命令替换。倒引号括起来的字符串被Shell解释为命令行并会先执行该命令行，并以它的标准输出结果取代整个倒引号部分。
- ****为转义字符。



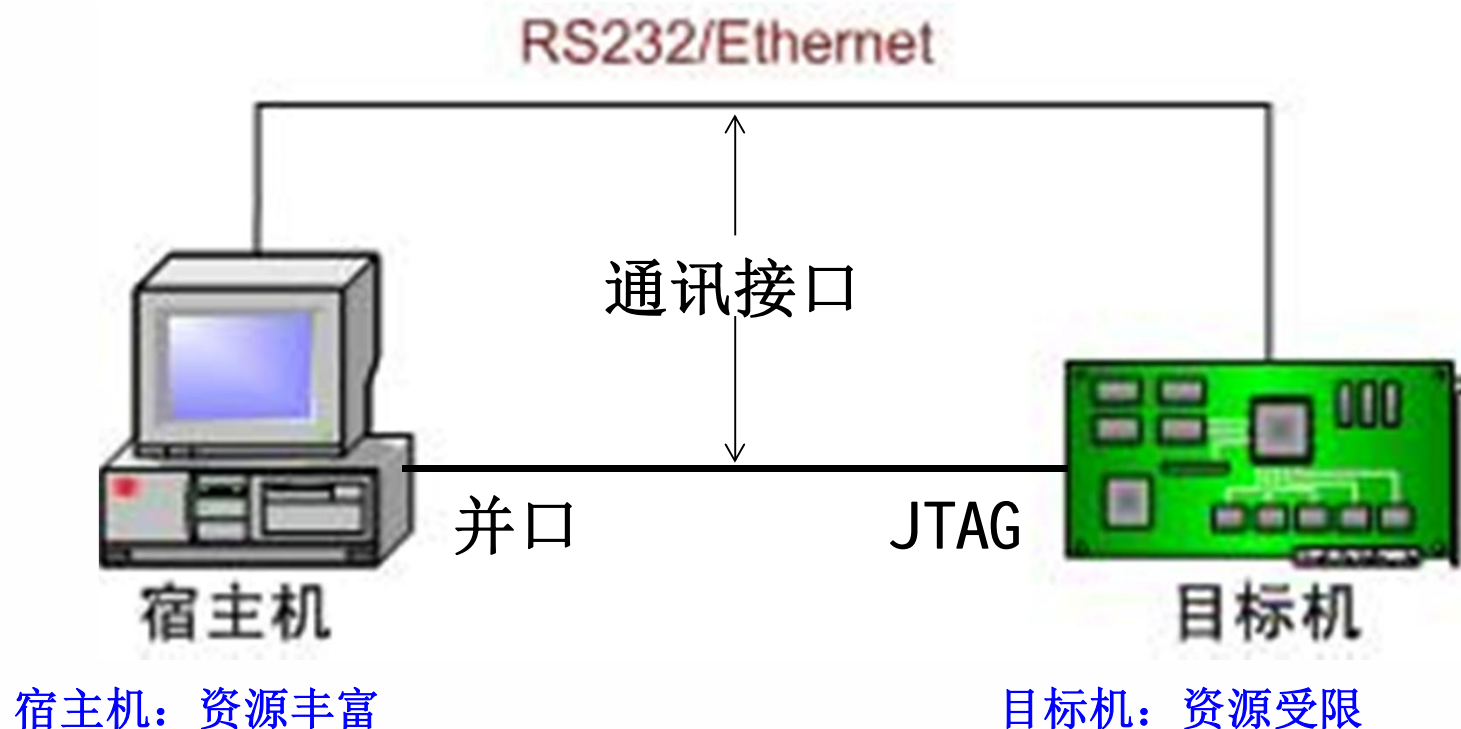
■ 范例

- names="Zhangsan Lisi Wangwu"
- echo" Dir is `pwd` and logname is \$LOGNAME"
- echo 'The time is `date` , the file is \$HOME/abc'
 - ✓ 输出 : The time is `date` , the file is \$HOME/abc
- echo" \${2+3},\${HOME}"
 - ✓ 输出 : 5,/root
- echo \${2<<3},\${8>>1}
 - ✓ 输出 : 16,4
- echo \${2>3},\${3>2}
 - ✓ 输出 : 0,1 (表达式为false时输出0 , 为true时输出1)



- 6.1 Bootloader过程
- 6.2 嵌入式操作系统简介
- 6.3 Linux终端命令
- 6.4 Shell编程
- 6.5 Linux编程基础

- 嵌入式系统采用**双机开发模式**：宿主机 - 目标机开发模式，利用资源丰富的PC机来开发嵌入式软件。



■ 什么是交叉编译？

- 在一种平台上编译出能在另一种平台(体系结构不同)上运行的程序；
- 在PC平台(X86)上编译出能运行在ARM平台上的程序，即编译得到的程序在X86平台上不能运行，必须放到ARM平台上才能运行；
- 用来编译这种程序的编译器就叫交叉编译器；
- 为了不与本地编译器混淆，交叉编译器的名字一般都有前缀，例如：arm-linux-gcc。



1、程序编辑

vi hello.c

```
1 #include <stdio.h>
2 int func(int n)
3 {
4     int sum = 0, i;
5     for(i=0; i<n; i++)
6     {
7         sum += i;
8     }
9     return sum;
10 }
11
12 main()
13 {
14     int i;
15     long result=0;
16     for(i=1; i<=100; i++)
17     {
18         result+=i;
19     }
20     printf("result[1-100]=%d \n", result);
21     printf("result[1-250]=%d \n", func(250));
22 }
```

2、程序编译

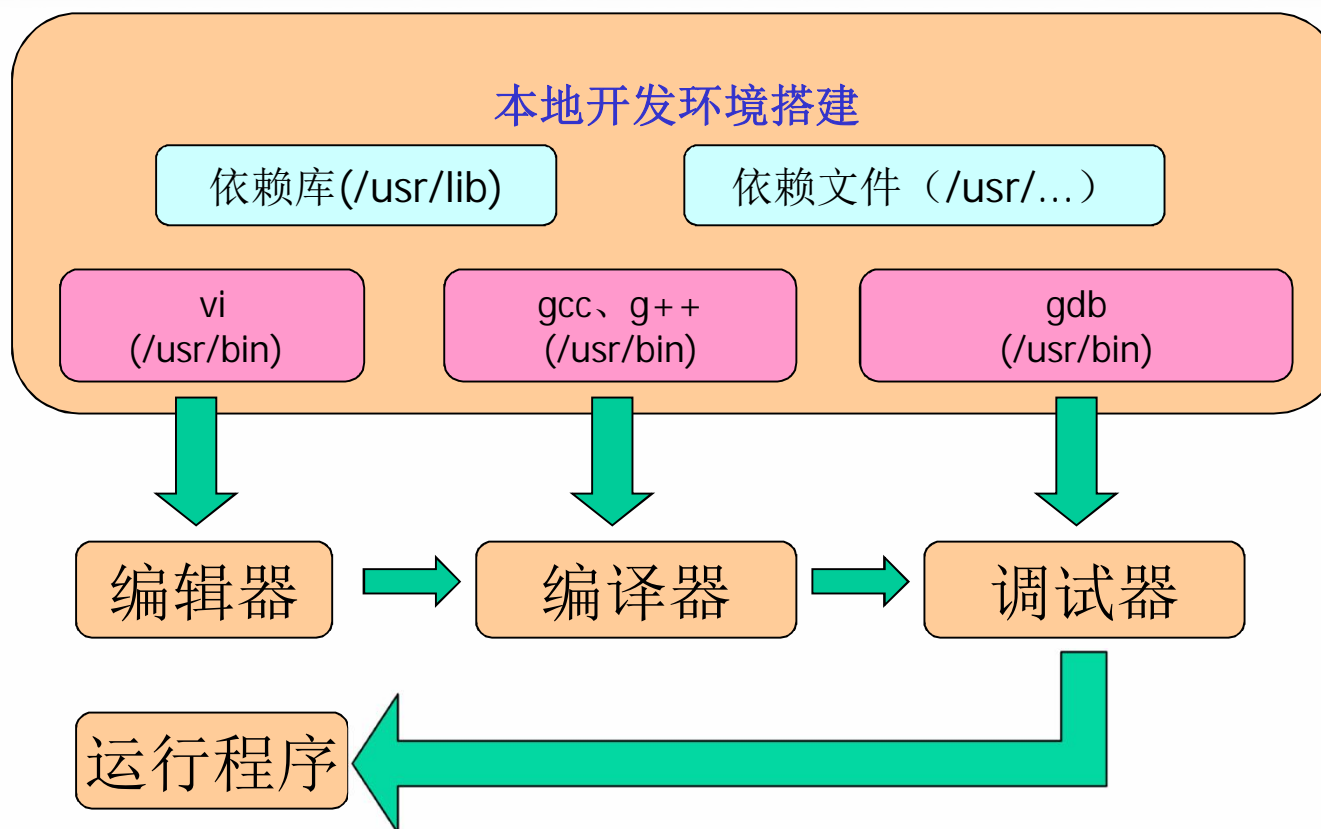
gcc -o hello hello.c

3、程序调试

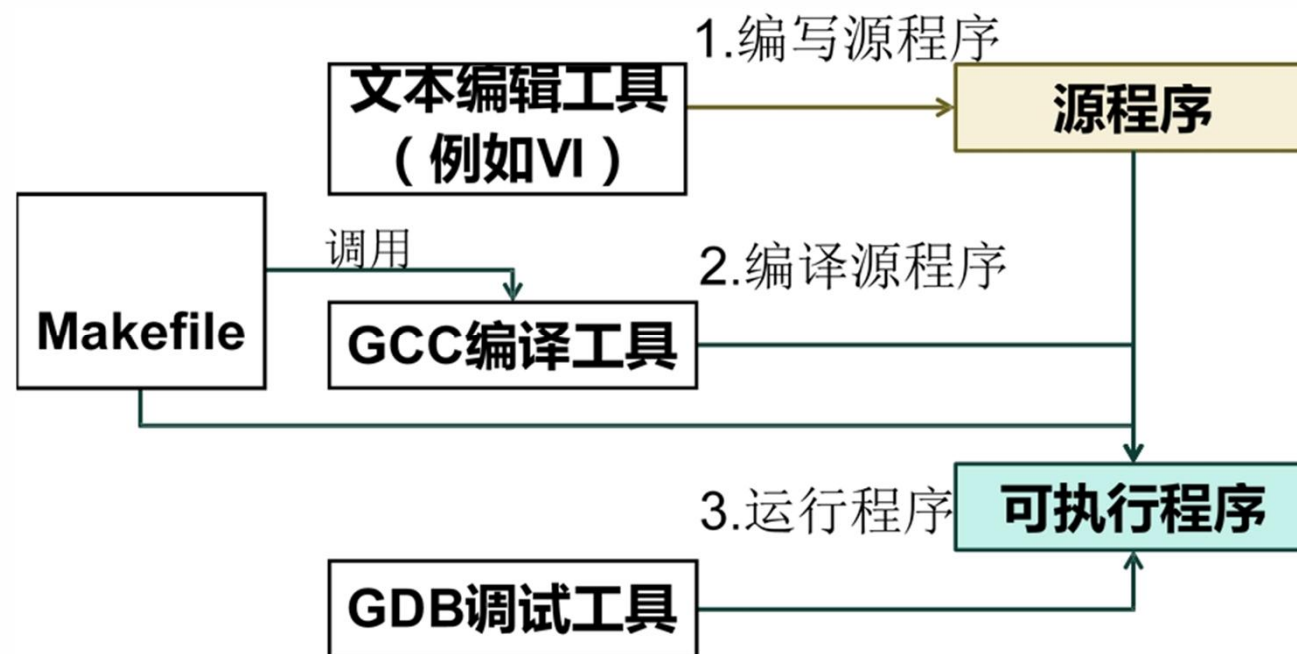
gdb hello

4、程序运行

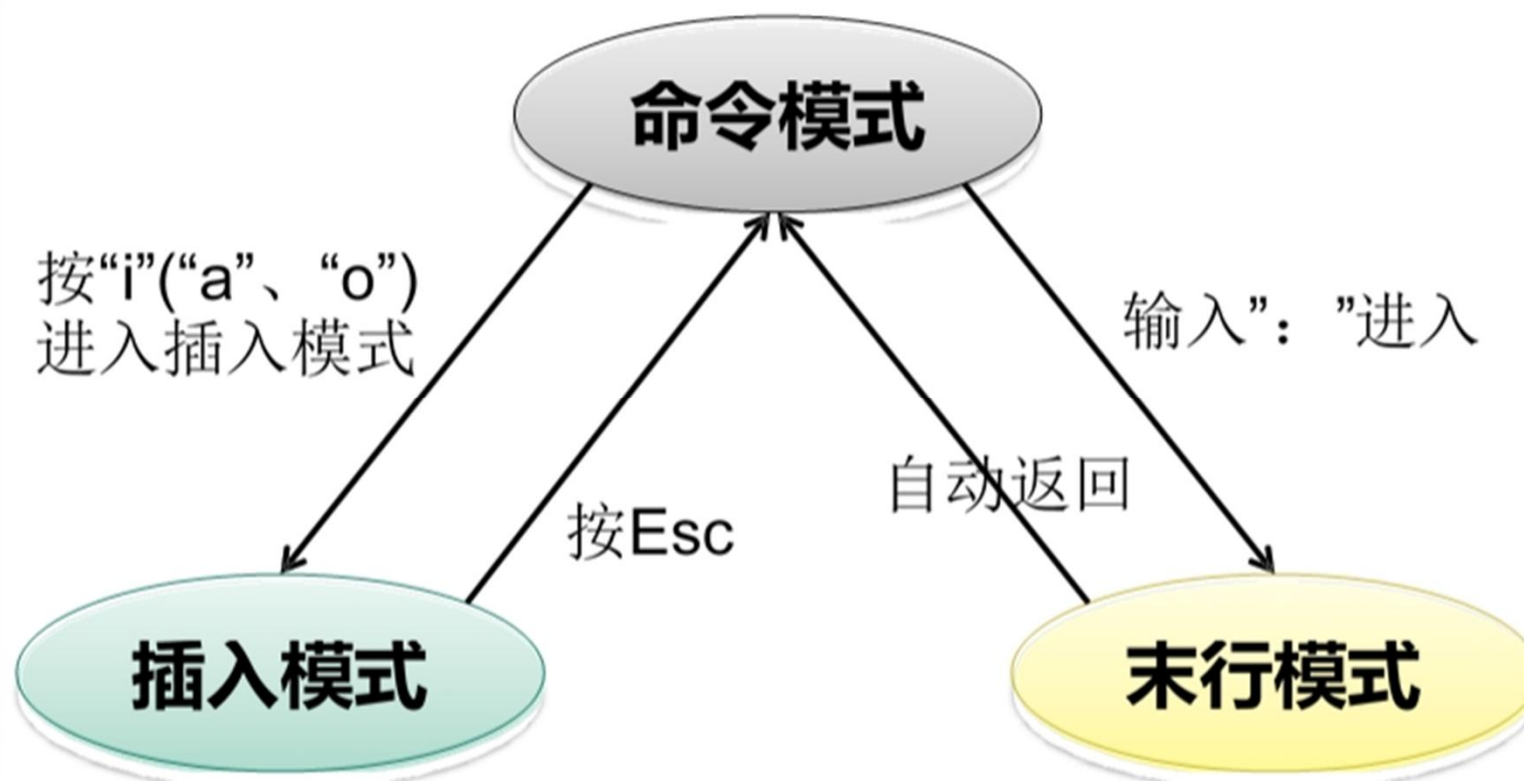
./hello



- VI编辑器
- GCC编译器
- Make工程管理器
- GDB调试器



■ VI编辑器的工作模式





■ VI编辑器的末(底)行模式

➤ 在**命令模式**下输入冒号“:”，可以进入**末行模式**。

➤ 范例：(文件的保存和退出)

- ✓ :w 保存
- ✓ :q 退出
- ✓ :w! 强制保存
- ✓ :q! 强制退出
- ✓ :wq (或x) 保存退出
- ✓ :wq! (或x !) 强制保存退出

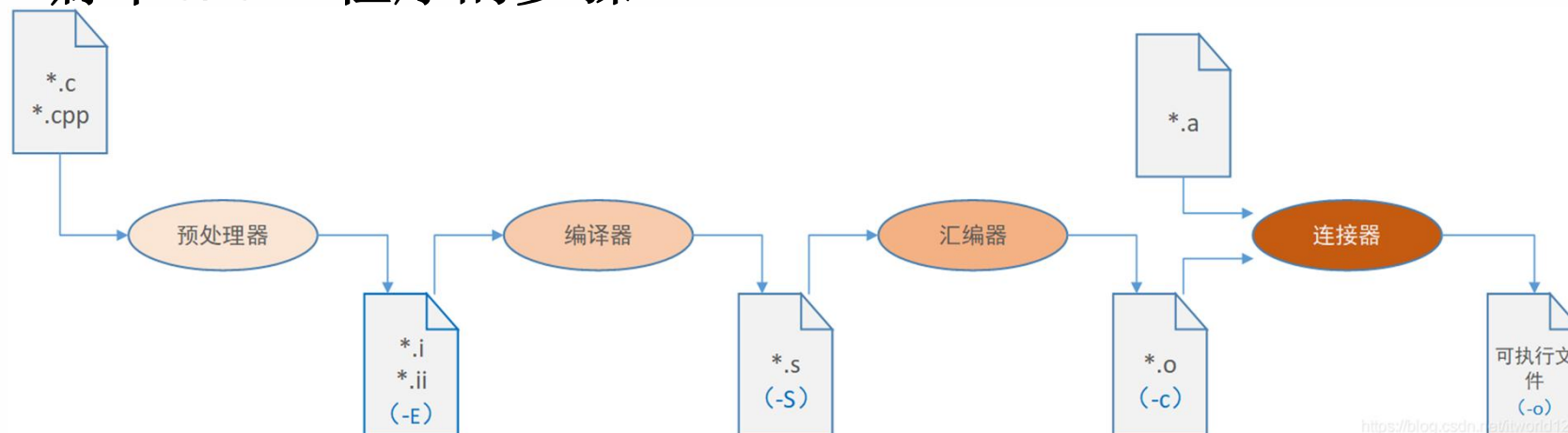
■ GCC（即gcc）的含义：

- GNU project C and C++ Compiler
- GNU Compiler Collection

■ 管理与维护

- 由GNU project（译为：项目、计划）完成

■ 编译C/C++程序的步骤



<https://blog.csdn.net/ailworld123>

■ 基本使用格式

➤ gcc [选项] <文件名>

注意：[选项]
区分大小写！

■ 常用选项及含义

gcc常用选项	
选项	含义
-o file	将经过gcc处理过的结果存为文件 <i>file</i> ，这个结果文件可能是预处理文件、汇编文件、目标文件或者最终的可执行文件。 假设被处理的源文件为 <i>source.suffix</i> ，如果这个选项被省略了，那么生成的可执行文件默认名称为 <i>a.out</i> ；目标文件默认名为 <i>source.o</i> ；汇编文件默认名为 <i>source.s</i> ；生成的预处理文件则会发送到标准输出设备。

■ 常用选项及含义(续)

gcc常用选项	
选项	含义
-E	仅对源文件进行预处理，不执行编译、汇编和链接操作。
-S	仅对源文件进行编译，不执行汇编和链接操作。
-c	仅对源文件进行汇编，不执行链接操作。 在对源文件进行查错时，或只需产生目标文件时可以使用该选项。

■ 常用选项及含义(续)

gcc常用选项	
选项	含义
-g[gdb]	在可执行文件中加入调试信息，方便进行程序的调试。 如果使用括号中的选项，表示加入gdb扩展的调试信息，方便使用gdb来进行调试。
-O[0、1、2、3]	对生成的代码使用优化。 中括号中的部分为优化级别，缺省的情况为2级优化，0为不进行优化。
-Dname[=definition]	将名为name的宏定义为definition。 如果中括号中的部分缺省，则宏被定义为1。

大写

■ 常用选项及含义(续)

gcc常用选项	
选项	含义
-I <i>dir</i>	在编译源程序时增加一个搜索头文件的额外目录—— <i>dir</i> ，即 include 增加一个搜索的额外目录。
-L <i>dir</i>	在编译源文件时增加一个搜索库文件的额外目录—— <i>dir</i>
-l <i>library</i>	在编译链接文件时增加一个额外的库，库名为 <i>library.a</i>
-W	禁止所有警告
-W <i>warning</i>	允许产生 <i>warning</i> 类型的警告， <i>warning</i> 可以是： main 、 unused 等很多取值，最常用是 -Wall ，表示产生所有警告。如果 <i>warning</i> 取值为 error ，其含义是将所有警告作为错误（ error ），即出现警告就停止编译。



gcc文件扩展名规范

扩展名	类型	可进行的操作方式
.c	c语言源程序	预处理、编译、汇编、链接
.C, .cc, .cp, .cpp, .c++, .cxx	c++语言源程序	预处理、编译、汇编、链接
.i	预处理后的c语言源程序	编译、汇编、链接
.ii	预处理后的c++语言源程序	编译、汇编、链接
.s	预处理后的汇编程序	汇编、链接
.S	未预处理的汇编程序	预处理、汇编、链接
.h	头文件	不进行任何操作
.o	目标文件	链接



■ 使用gcc编译代码

➤ 源代码

```
源程序——hello.c  
#include <stdio.h>  
int main(void)  
{  
    printf("hello gcc!\r\n");  
    return 0;  
}
```



■ 生成预处理文件

➤ 命令

✓ `gcc -E hello.c -o hello.i` #将hello.c预处理输出hello.i

预处理文件hello.i的部分内容

```
.....
extern void funlockfile (FILE *__stream) ;
# 679 "/usr/include/stdio.h" 3

# 2 "hello.c"

int main(void)
{
    printf("hello gcc!\n");
    return 0;
}
```




■ 生成汇编文件

➤ 命令

✓ gcc -S hello.i -o hello.s

#将预处理输出文件hello.i汇编成

#hello.s文件

文件hello.s的部分内容

```
.....  
main:  
    pushl    %ebp  
    movl     %esp, %ebp  
    .....  
    addl     $16, %esp  
    movl     $0, %eax  
    leave  
    ret  
    .....
```



■ 生成目标文件

➤ 命令

✓ gcc -c hello.s -o hello.o

#将汇编输出文件hello.s编译输出

#hello.o文件

■ 生成可执行文件

➤ 命令

✓ gcc hello.o -o hello

✓ 运行程序

● ./hello

hello gcc!

- 编译多个文件



greeting.h

```
#ifndef _GREETING_H
#define _GREETING_H
void greeting (char * name);
#endif
```

greeting.c

```
#include <stdio.h>
#include "greeting.h"
void greeting (char * name)
{
    printf("Hello %s!\r\n",name);
}
```

my_app.c

```
#include <stdio.h>
#include "greeting.h"
#define N 10
int main(void)
{
    char name[N];
    printf("Your Name,Please:");
    scanf("%s",name);
    greeting(name);
    return 0;
}
```

• 编译多个文件



学院 荣誉 责任



■ 目录结构(1)

➤ 编译命令



```
gcc my_app.c greeting.c -o my_app
```

■ 目录结构(2)

➤ 编译方式(1)



```
gcc my_app.c functions/greeting.c -o my_app -I functions  
# 将my_app.c和functions目录下的greeting.c文件编译为可执行文件my_app,  
# 同时在functions目录中搜索头文件。
```

• 编译多个文件



学院 荣誉 责任



■ 目录结构(2)

➤ 编译方式(2)

✓ 分步编译

✓ 思路：

- 编译每一个.c文件，得到.o的目标文件；
- 将每一个.o的目标文件链接成一个可执行的文件。

✓ 命令：

- 1、`gcc -c my_app.c -I functions`
- 2、`gcc -c functions/greeting.c`
- 3、`gcc my_app.o greeting.o -o my_app`





■ 使用make工具进行编译

➤ 基本格式：

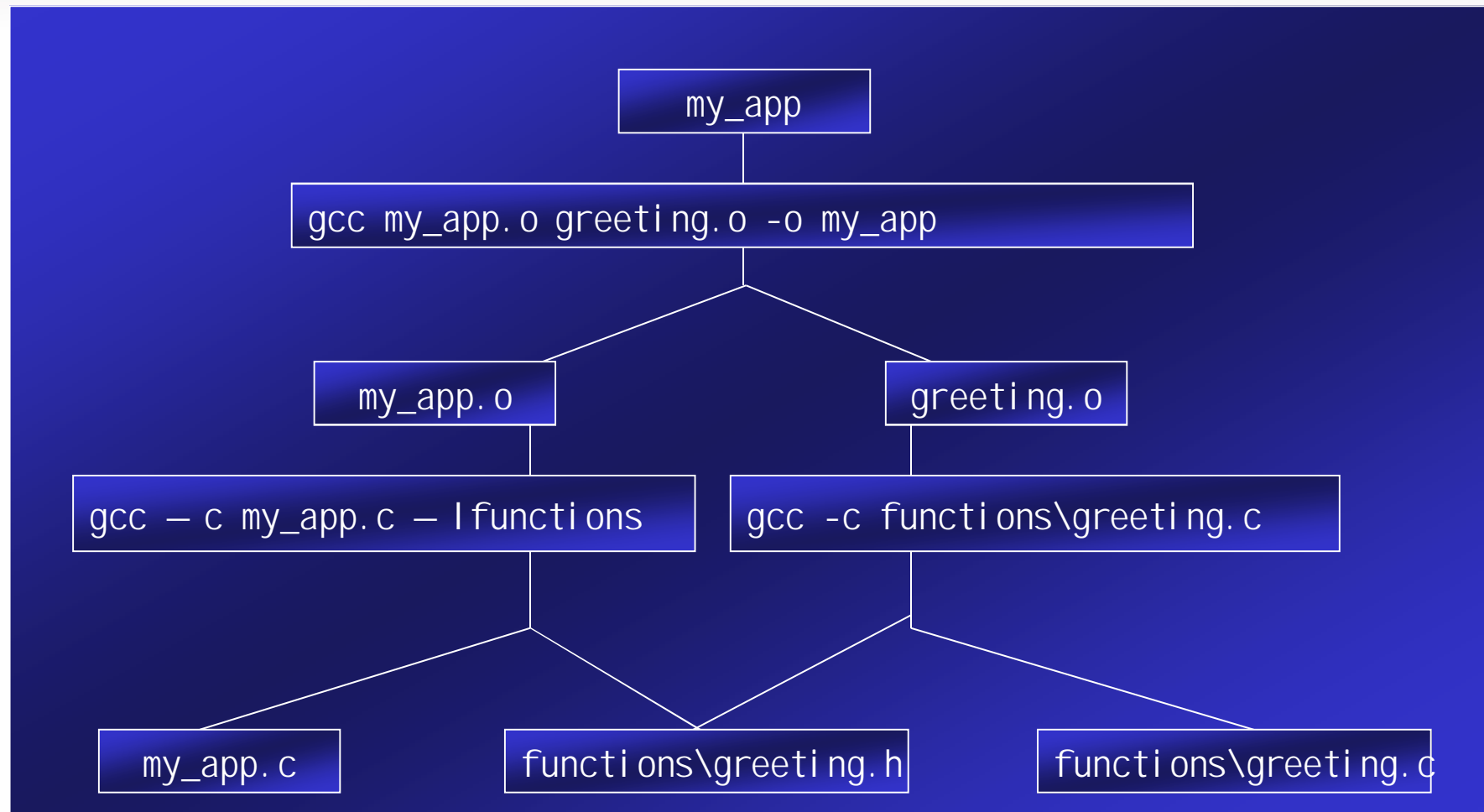
- ✓ make [[命令选项] [命令参数]]
- ✓ 使用make工具时，会寻找Makefile作为编译指导文件。



■ Makefile示例

Makefile文件	
1	my_app:greeting.o my_app.o
2	gcc my_app.o greeting.o -o my_app
3	greeting.o:functions\greeting.c functions\greeting.h
4	gcc -c functions\greeting.c
5	my_app.o:my_app.c functions\greeting.h
6	gcc -c my_app.c -I functions

■ 目标的依赖关系





■ 实用的Makefile

更实用的Makefile文件

- | | |
|---|---|
| 1 | <code>OBJS=greeting.o my_app.o</code> |
| 2 | <code>CC=gcc</code> |
| 3 | <code>CFLAGS=-Wall -O -g</code> |
| 4 | <code>my_app:\${OBJS}</code> |
| 5 | <code> \${CC} \${OBJS} -o my_app</code> |
| 6 | <code>greeting.o:functions\greeting.c functions\greeting.h</code> |
| 7 | <code> \${CC} \${CFLAGS} -c functions\greeting.c</code> |
| 8 | <code>my_app.o:my_app.c functions\greeting.h</code> |
| 9 | <code> \${CC} \${CFLAGS} -c my_app.c -Ifunctions</code> |



■ 实验中使用的Makefile

```
CC=arm-linux-gcc
EXEC=hello
OBS=hello.o
CFLAGS+=
LDFLAGS+=

all:$(EXEC)
$(EXEC):$(OBS)
    $(CC) $(LDFLAGS) -o $@ $(OBS)

clean:
    rm -f $(EXEC) *.elf *.gdb *.o
~
```

如果一个工程有3个头文件和8个c文件，则对应的Makefile文件如下：

edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o

– cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o

main.o : main.c defs.h

– cc -c main.c

kbd.o : kbd.c defs.h command.h

– cc -c kbd.c

command.o : command.c defs.h command.h

– cc -c command.c

display.o : display.c defs.h buffer.h

– cc -c display.c

insert.o : insert.c defs.h buffer.h

– cc -c insert.c

search.o : search.c defs.h buffer.h

– cc -c search.c

files.o : files.c defs.h buffer.h command.h

– cc -c files.c

utils.o : utils.c defs.h

– cc -c utils.c

clean :

– rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o



The End!