目录

一、 Linux 文件系统

目录	概述
/	Linux 文件系统根目录
/bin	它是重要的二进制应用程序，包含二进制文件，系统的所有用户使用的命令都在这里。
/boot	启动包含引导加载程序的相关文件。
/dev	包含设备文件，终端文件，USB 或者连接到系统的任何设备。
/etc	配置文件，启动脚本等，包含所有程序所需要的配置文件，也包含了启动/停止单个应用程序的启动和关闭 shell 脚本。
/home	本地主要路径，所有用户用 home 目录存储个人信息。
/lib	系统库文件，包含支持位于 /bin 和 /sbin 下的二进制库文件。
/lost+found	在根目录下提供一个遗失+查找系统，必须在 root 用户下才能查看当前目录下的内容。
/media	挂载可移动介质。
/mnt	挂载文件系统。
/opt	提供一个可选的应用程序安装目录，可将软件安装至该目录，对软件进行测试。
/proc	特殊的动态目录，维护系统信息和状态，包括当前运行中进程信息。
/root	root 用户的主要目录文件夹。
/sbin	存放有可执行二进制文件，用于存放系统管理和系统维护的关键性命令，常见的 Linux 命令都位于/bin 目录下，系统管理员使用的 Linux 命令位于/sbin 目录下。
/tmp	系统和用户创建的临时文件，系统重启时，这个目录下的文件都会被删除。
/var	经常变化的文件，诸如日志文件或数据库等。

绝对路径：指文件在文件系统中的准确位置。通常在本地主机上，以根目录为起点。例如“/usr/games/gnect”就是绝对路径。

相对路径：指相对于用户当前位置的一个文件或目录的位置。例如，用户处在 usr 目录中时，只需要“games/gnect”就可确定这个文件。

二、 Linux 用户管理与文件权限

用户：用户是能够获取系统资源的权限的集合；每个用户都会分配一个特有的 id 号-uid。Linux 用户包括管理员（root）、系统用户、普通用户。

超级用户：根用户也就是 root 用户，它的 ID 是 0，也被称为超级用户，root 账户拥有对系统的完全控制权：可以修改、删除任何文件，运行任何命令。所以 root 用户也是系统里面最具危险性的用户，root 用户甚至可以在系统正常运行时删除所有文件系统，造成无法挽回的灾难。所以一般情况下，使用 root 用户登录系统时需要十分小心。

root 可以超越任何用户和用户组来对文件或目录进行读取、修改或删除（在系统正常的许可范围内）；对可执行程序的执行、终止；对硬件设备的添加、创建和移除等；也可以对文件和目录进行属主和权限进行修改，以适合系统管理的需要（因为 root 是系统中权限最高的特权用户）。

普通用户：也称为一般用户，它的 UID 为 1000-60000 之间，普通用户可以对自己目录下的文件进行访问和修改，也可以对经过授权的文件进行访问；在添加普通用户时，系统默认用户 ID 从 1000 开始编号。

虚拟用户：也称为系统用户，它的 UID 为 1-999 之间，虚拟用户最大的特点是不提供密码登录系统，它们的存在主要是为了方便系统的管理。

UID 指的是用户的 ID（User ID），一个用户 UID 标示一个给定用户，UID 是用户的唯一标示符，通过 UID 可以区分不同用户的类别（用户在登录系统时是通过 UID 来区分用户，而不是通过用户名来区分）：

用户组是具有相同特征用户的逻辑集合，有时我们需要让多个用户具有相同的权限，比如查看、修改某一个文件的权限，一种方法是分别对多个用户进行文件访问授权，如果有 10 个用户的话，就需要授权 10 次，显然这种方法不太合理；另一种方法是建立一个组，让这个组具有查看、修改此文件的权限，然后将所有需要访问此文件的用户放入这个组中，那么所有用户就具有了和组一样的权限。这就是用户组，将用户分组是 Linux 系统中对用户进行管理及控制访问权限的一种手段，通过定义用户组，在很大程度上简化了管理工作。用户组使用 GID 作为唯一标识符。

1. **用户和用户组的关系**

一对一：一个用户可以存在一个用户组中，作为组中的唯一成员；

一对多：一个用户可以存在多个用户组中，该用户具有多个组的共同权限；

多对一：多个用户可以存在一个用户组中，这些用户具有和组相同的权限；

多对多：多个用户可以存在多个用户组中，其实就是以上三种关系的扩展。

2. **重要文件：**

/etc/passwd　　存放 UID

/etc/shadow　　存放密码

/etc/group　　存放 GID

3. **修改组**-groupmod：

语法：groupmod [options] group_name

其中的命令选项说明如下：

-g 修改为要使用的 GID

-h 显示此帮助信息并退出

-n 修改为要使用的组名称

-o 允许使用重复的 GID

-p 更改密码(加密过的)

4. **删除组**-groupdel

groupdel 命令用于从系统中删除组，需要注意的是，若是在组中仍然包括某些用户，此时需要先删除这些用户后，才能删除组。

功能说明：用于删除指定的用户组，此命令不能删除用户归属的主用户组。

语法：groupdel [options] group_name

其中的命令选项说明如下：

-f 即便是用户的主组也继续删除

-h 显示此帮助信息并退出

5. **关联用户和组**-gpasswd

语法：gpasswd [option] group_name。

其中的命令选项说明如下：

-a **向组** GROUP **中添加用户** USER

-d **从组** GROUP **中添加或删除用户**

-M **设置组** GROUP **的成员列表**

-A **设置组的管理员列表**

-r **移除组** GROUP **的密码**

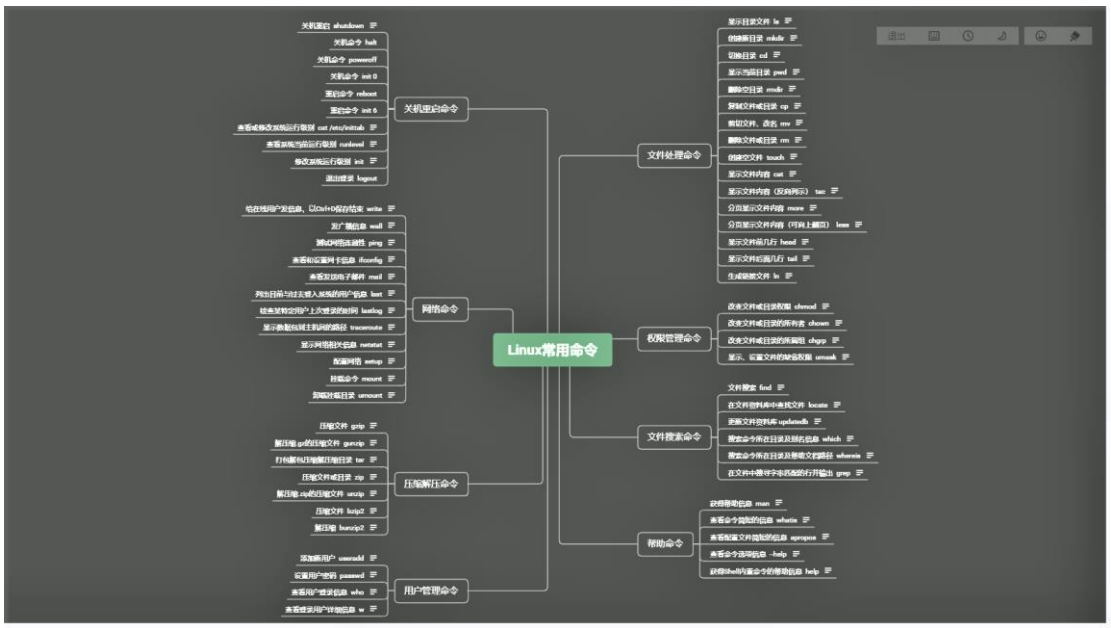
-R **向其成员限制访问组** GROUP

-Q **要** chroot **进的目录**

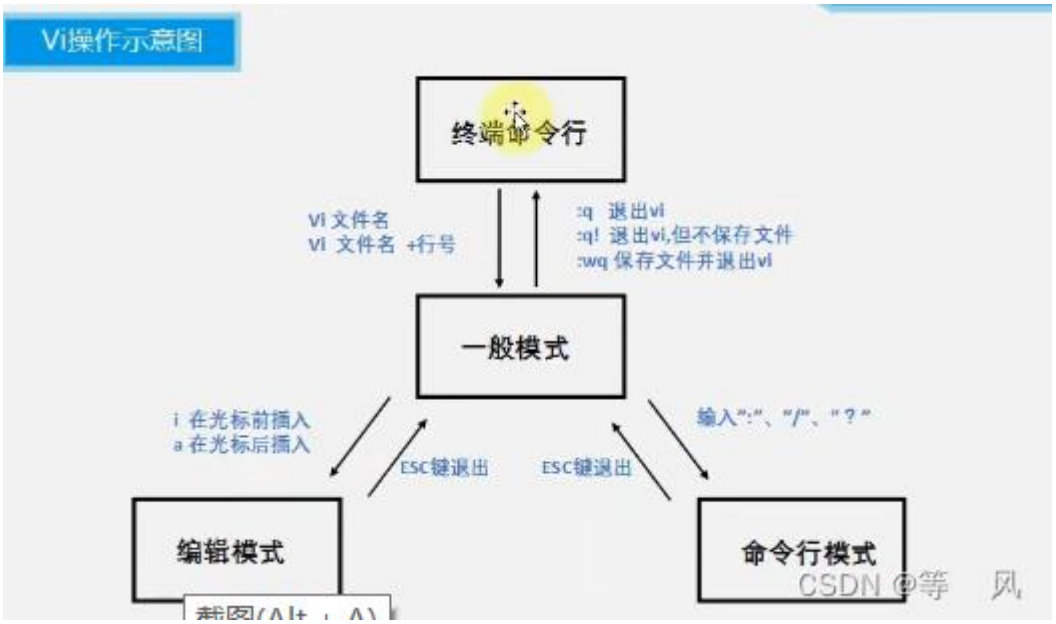
6. **文件权限（使用** ls -l **命令查看）**

文件类型	用户权限	用户组权限	其他用户权限
第0位	第1、2、3位	第4、5、6位	第7、8、9位
d:目录	r、w、x	r、w、x	r、w、x
-:文件	读、写、执行	读、写、执行	读、写、执行
…			

三、 Linux 常用命令


Linux常用命令
关机重启命令
关机重启 shutdown
关机命令 halt
关机命令 poweroff
关机命令 init 0
重启命令 reboot
重启命令 init 6
查看或修改系统运行级别 cat /etc/inittab
查看系统当前运行级别 runlevel
修改系统运行级别 init
退出登录 logout
网络命令
给在线用户发信息，以Ctrl+D保存结束 write
发广播信息 wall
测试网络连通性 ping
查看和配置网卡信息 ifconfig
查看发送电子邮件 mail
列出目前与过去登入系统的用户信息 last
检查某特定用户上次登录的时间 lastlog
显示数据包到主机间的路径 traceroute
显示网络相关信息 netstat
配置网络 setup
挂载命令 mount
卸载光驱目录 umount
压缩解压命令
压缩文件 gzip
解压缩.gz的压缩文件 gunzip
打包解包压缩解压缩目录 tar
压缩文件或目录 zip
解压缩.zip的压缩文件 unzip
压缩文件 bzip2
解压缩 bunzip2
用户管理命令
添加新用户 useradd
设置用户密码 passwd
查看用户登录信息 who
查看登录用户详细信息 w
文件处理命令
显示目录文件 ls
创建新目录 mkdir
切换目录 cd
显示当前目录 pwd
删除空目录 rmdir
复制文件或目录 cp
剪切文件，改名 mv
删除文件或目录 rm
创建空文件 touch
显示文件内容 cat
显示文件内容（反向列示） tac
分页显示文件内容 more
分页显示文件内容（可向上翻页） less
显示文件前几行 head
显示文件后面几行 tail
生成链接文件 ln
权限管理命令
改变文件或目录的权限 chmod
改变文件或目录的所有者 chown
改变文件或目录的所属组 chgrp
显示、设置文件的缺省权限 umask
文件搜索命令
文件搜索 find
在文件资料库中查找文件 locate
更新文件资料库 updatedb
搜索命令所在目录及别名信息 which
搜索命令所在目录及帮助文档路径 whereis
在文件中搜寻字串匹配的行并输出 grep
帮助命令
获得帮助信息 man
查看命令简短的信息 whatis
查看配置文件简短的信息 apropos
查看命令选项信息 --help
获得shell内置命令的帮助信息 help

四、 VI 编辑器的使用



模式	命令	描述
命令模式	i	在当前光标位置进入输入模式
命令模式	a	在当前光标位置 之后 进入输入模式
命令模式	I	在当前行的开头，进入输入模式
命令模式	A	在当前行的结尾，进入输入模式
命令模式	o	在当前光标下一行进入输入模式
命令模式	O	在当前光标上一行进入输入模式
输入模式	esc	任何情况下输入esc都能回到命令模式

命令模式	键盘上、键盘k	向上移动光标
命令模式	键盘下、键盘j	向下移动光标
命令模式	键盘左、键盘h	向左移动光标
命令模式	键盘右、键盘l	向后移动光标
命令模式	0	移动光标到当前行的开头
命令模式	$	移动光标到当前行的结尾
命令模式	pageup(PgUp)	向上翻页
命令模式	pangdown(PgDn)	向下翻页
命令模式	/	进入搜索模式
命令模式	n	向下继续搜索
命令模式	N	向上继续搜索

命令模式	dd	删除光标所在行的内容
命令模式	ndd	n是数字，表示删除当前光标向下n行
命令模式	yy	复制当前行
命令模式	nyy	n是数字，复制当前行和下面的n行
命令模式	p	粘贴复制的内容
命令模式	u	撤销修改
命令模式	ctrl + r	反向撤销修改
命令模式	gg	跳到首行
命令模式	G	跳到行尾
命令模式	dG	从当前行开始，向下全部删除
命令模式	dgg	从当前行开始，向上全部删除
命令模式	d$	从当前光标开始，删除到本行的结尾
命令模式	d0	从当前光标开始，删除到本行的开头

保存和退出

`:w` => 保存（只是对当前文件进行保存操作，并没有退出这个文件）

`:q` => 退出，注意文件必须先保存，然后才能退出

`:wq` => 保存并退出

`:q!` => 强制退出（此时文件未保存）

五、 Shell 脚本编程

1. 什么是 shell

shell 本质上是 linux 命令,一条一条命令组合在一起,实现某一个目的,就变成了 shell 脚本。它从一定程度上 减轻了工作量, 提高了工作效率。

2. 官方化的 shell 介绍

Shell 通过提示您输入,向操作系统解释该输入,然后处理来自操作系统的任何结果输出, 简单来说 Shell 就是一个用户跟操作系统之间的一个命令解释器。

- 常见的 shell 有哪些 (cat /etc/shells)

  Bourne Shell (/usr/bin/sh 或/bin/sh)

  Bourne Again Shell (/bin/bash)

  C Shell (/usr/bin/csh)

  K Shell (/usr/bin/ksh)

  Shell for Root (/sbin/sh)

  最常用的 shell 是 Bash, 也就是 Bourne Again Shell。Bash 由于易用和免费, 在日常工作中被广泛使用, 也是大多数 Linux 操作系统默认的 Shell 环境。

3. Shell 编程步骤

(1) 编辑:

sudo vi name.sh 开头:#!/bin/bash,#!用来声明脚本由什么 shell 解释,否则使用默认 shell

(2) 保存

(3) 修改权限:

sudo chmod 777 name.sh

(4) 运行:

./name.sh

/bin/bash name.sh(指定解析器)

source name.sh

. name.sh

4. Shell 语法

1. **定义变量**

   variable = value

   variable = 'value'　　**所见即所得，不会对其它变量的引用解引用**

   variable = "value"　　**会对引用变量解引用**

2. **变量使用**

   $variable

   ${variable}　　**界定范围**

3. **命令的结果赋值给变量**

   variable = `command`(**反引号**)

   variable = $(command)

4. **删除变量**:unset

5. **特殊变量**

变量	含义
$0	当前脚本的文件名。
$n (n≥1)	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是 $1，第二个参数是 $2。
$#	传递给脚本或函数的参数个数。
$*	传递给脚本或函数的所有参数。
$@	传递给脚本或函数的所有参数。当被双引号" "包含时，$@ 与 $* 稍有不同.
$?	上个命令的退出状态或者获取函数返回值。
$$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

6. **从键盘读取数据**:read (read -p "input a:" a)
7. **退出当前进程**:exit
8. **对整数进行数学运算**: (())
9. **逻辑与/或**: &&/||([$a eq $b] || echo "a = b")

10. 判断某个条件是否成立：test expression 或[expression]

选项	作用
-eq	判断数值是否相等
-ne	判断数值是否不相等
-gt	判断数值是否大于
-lt	判断数值是否小于
-ge	判断数值是否大于等于
-le	判断数值是否小于到等于
-z str	判断字符串 str 是否为空
-n str	判断字符串str是否为非空
=和==	判断字符串str是否相等
-d filename	判断文件是否存在，并且是否为目录文件。
-f filename	判断文件是否存在，并且是否为普通文件。

11. 管道：command1|command2 命令拼接，command1 的输入作为 command2 的输入。
12. If 语句

    if 条件

    then

        statement

    fi
13. If else 语句

    if 条件 1

    then

        statement

    else

        statement

    fi

14. If elif else 语句

```
if 条件1
then
    Statement1
elif 条件2
then
    Statement2
……
else
    Statement
fi
```

15. case in 语句

```
case 表达式 in
    pattem1)
        Statement1
        …
    Pattem2)
        Statement2
        …
    *)      //其它情况
        Statement
esac
```

16. for in 语句

```
for variable in value_list
do
    statements
done
```

其中 value_list 可以为：

具体的值、取值范围{}、使用命令的执行结果、shell 通配符、特殊变量

17. While 循环

```
while 条件
do
    Statements;
done
```

18. 函数

```
    function names(){
        statemens
        [return value]
    }
```

六、 Linux 环境变量

1. 环境变量的含义：

环境变量一般是指操作系统中指定运行环境的一些参数，即系统预定义的参数。它相当于一个指针，想要查看变量的值，需要加上“$"符号。环境变量就是一个变量，他的值随用户的不同而变化。

2. 环境变量的分类：

按作用范围分：

环境变量：相当于全局变量，存在于所有的 Shell 中，具有继承性。

本地变量：相当于局部变量，只存在于当前 Shell 中，本地变量包含环境变量，但非环境变量不具有继承性。

按生存周期分：

永久性环境变量：需要修改配置文件，变量永久生效。

暂时性环境变量：使用 export 定义，关闭 Shell 后失效。

3. 常见的环境变量

PATH：表示在当前目录下执行的每一条指令的搜索路径，每个目录以冒号隔开。当执行一条指令时，系统就会从系统文件中去寻找，找到了就执行；否则不执行。

HOME：该变量指定用户的主工作目录，即用户登录到 Linux 系统时，默认的目录。

HISTSIZE：Linux 系统保存历史命令的数目。

HOSTNAME：该变量显示当前主机名称。

SHELL：该变量显示用户当前使用的解析器。

4. 相关命令

echo $变量名：显示某个环境变量的值。

env：显示所有的环境变量。

set：显示本地定义的 shell 变量和环境变量。

export:设置一个新的环境变量，使用 export 定义的环境变量是暂时性环境变量，只在当前 shell 有效，关闭 Shell 后失效。

unset:清除指定的环境变量。

readonly:设置只读的环境变量。

5. 需要修改的文件

Linux 加载环境变量的顺序：

/etc/environment

/etc/profile

/etc/bash.bashrc

/etc/profile.d/test.sh

~/.profile

~/.bashrc

用户级别环境变量定义文件：~/.bashrc、~/.profile（部分系统为：~/.bash_profile)

系统级别环境变量定义文件：/etc/bashrc、/etc/profile(部分系统为：/etc/bash_profile)、/etc/environment

/.profile 文件只在用户登录的时候读取一次

/.bashrc 会在每次运行 Shell 脚本的时候读取一次

全部用户、全部进程共享/etc/bash.bashrc

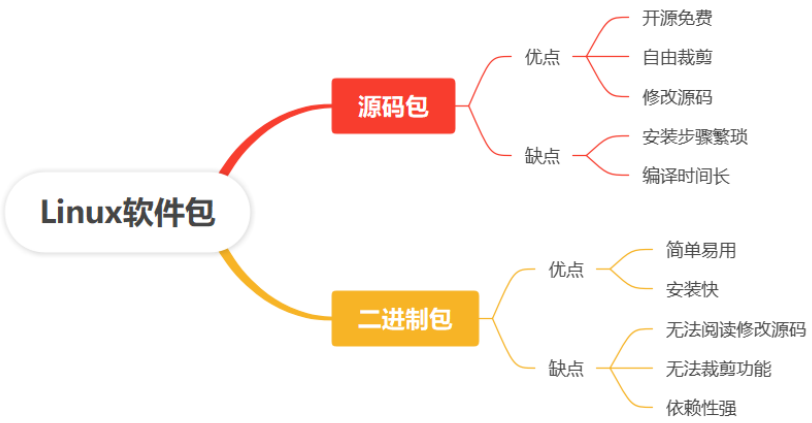
一个用户、全部进程共享~/bashrc

七、 软件包

1. 组成

文件类型	保存目录
普通程序	/usr/bin
Root 权限程序	/usr/sbin
程序配置文件	/etc
日志文件	/var/log
文档文件	/usr/share/doc

2. 分类



3. dpkg 工具

dpkg 是底层包管理工具，主要用于对已下载已安装的 deb 包进行管理。

安装软件：dpkg -i xxxx.deb

查看安装目录：dpkg -L xxxx

显示版本：dpkg -l xxxx

详细信息：dpkg -s xxxx

罗列内容：dpkg -c xxxx

卸载软件：dpkg -r xxxx

4. deb 包文件结构

DEBIAN 目录:

control 文件 (必要):

Section: 申明软件的类别

Priority: 申明软件对于系统的重要程度

Essential: 申明是否是系统最基本的软件包 (选项为 yes/no), 如果是的话, 这就表明该软件是维持系统稳定和正常运行的软件包, 不允许任何形式的卸载 (除非进行强制性的卸载)

Architecture: 支持的硬件包平台

Depends:软件所依赖的其他软件包和库文件。如果是依赖多个软件包和库文件, 彼此之间采用逗号隔开;

Pre-Depends:软件安装前必须安装、配置依赖性的软件包和库文件, 它常常用于必须的预运行脚本需求;

Package:软件名称

Version:版本

Maintainer:软件包的维护者

Description:对软件的描述

Preinst:解压前执行的脚本, 为正在被升级的包停止相关服务

Postinst: 软件安装完后,执行该 Shell 脚本,一般用来配置软件执行环境, 必须以"#!/bin/sh"为首行, 然后给该脚本赋予可执行权限。

Prerm:卸载之前执行的脚本

Postrm:卸载后执行的脚本

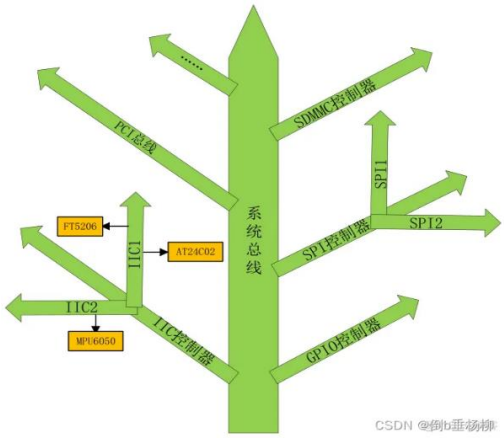
copyright:版本申明

changlog:修改记录

八、 设备树

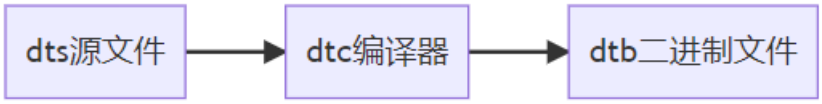
1. 作用

设备树是一个描述硬件的数据结构，甚至你可以将其看成一个大结构体（这个结构体就是平台，成员就是具体的设备），需要注意的是设备树并不能解决所有的硬件配置问题（例如：机器识别），它只是提供一种语言，将硬件的配置从 linux 内核的源码中提取出来。



2. 使用

在 linux 中，设备树文件的类型有.dts .dtc 和.dtb。.dtb 文件是.dts 被 DTC 编译后的二进制格式的设备树文件，DTC 是将.dts 编译为.dtb 的工具，相当于 gcc，三者关系如下图：



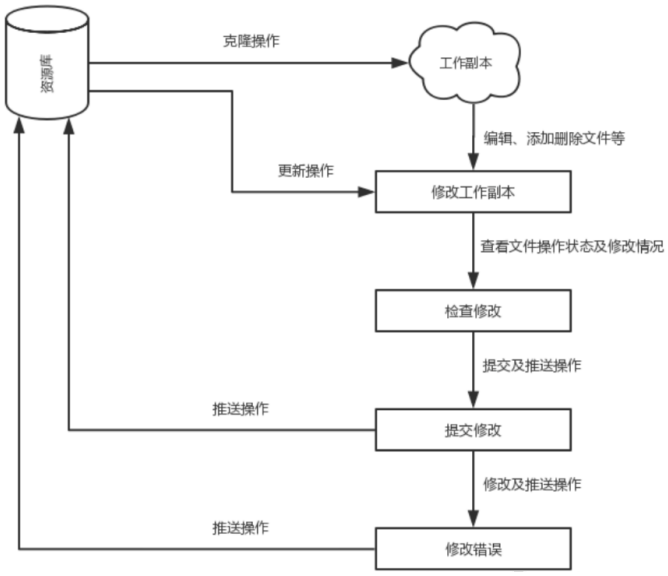
.dts 是设备数源文件，可以包含其他.dtsi 文件。由于一个 SoC 可能对应多个设备，一个 SOC 可以做很多不同的板子，这些不同的板子肯定是有共同的信息，将这些共同的信息提取出来作为一个通用.dsti 文件，其他的.dts 文件直接引用这个通用文件即可，类似于 C 语言中的头文件。一般.dts 描述板级信息（也就是开发板上有哪些 IIC 设备、SPI 设备等），.dtsi 描述 SOC 级信息（也就是 SOC 有几个 CPU、主频是多少、各个外设控制器信息等）。

九、 Git 的使用

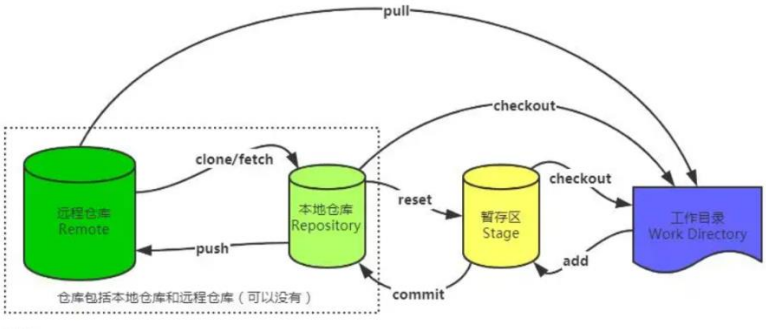
1. 集中式与分布式

分布式是 git 区别在于集中式的版本控制系统每次在写代码时都需要从服务器中拉取一份下来，并且如果服务器丢失了，那么所有的就都丢失了，你本机客户端仅保存当前的版本信息，换句话说，集中式就是把代码放在一个服务器上集中管理，你的所有回滚等操作都需要服务器的支持。分布式的区别在于，每个人的电脑都是服务器，当你从主仓库拉取一份代码下来后，你的电脑就是服务器，无需担心主仓库被删或者找不到的情况，你可以自由在本地回滚，提交，当你想把自己的代码提交到主仓库时，只需要合并推送到主仓库就可以了，同时你可以把自己的代码新建一份仓库分享给其它人。

2. 工作流程



3. Git 工作区、暂存区和版本库



工作区:就是你在电脑里能看到的目录，平时存放项目代码的地方。

暂存区:暂存区有时也叫作索引(index)，一般存放在 .git 目录下的 index 文件 (.git/index) 中，事实上它只是一个文件，保存即将提交到文件列表信息

版本库/仓库区:工作区有一个隐藏目录 .git，这个不算工作区，而是 Git 的版本库。就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中 HEAD 指向最新放入仓库的版本

远程仓库:远程仓库，托管代码的服务器，可以简单的认为是你项目组中的一台电脑用于远程数据交换。

4. Git 版本管理

(1). 创建版本

创建目录然后使用 git init 命令将目录变成 Git 仓库。多了一个.git 文件夹，这个目录是 Git 来跟踪管理版本库的

(2). 添加文件到版本库

一、创建文件，然后使用 git add filename 命令将文件添加到 Git 仓库。

二、用 git commit -m "text"命令将文件提交到 Git 仓库

commit 可以一次提交很多文件，所以你可以多次 add 不同的文件。git add 将工作区文件提交到暂存区，git commit 将暂存区的内容提交到本地 git 仓库。

可以把 Git 中的 commit 理解成"快照"，每当觉得文件修改到一定程度的时候，就可以保存一个快照，当你把文件改乱或者误删，还可以从最近的一个 commit 恢复。

如果 git add 添加错文件，想执行撤销动作，可以执行 git reset HEAD

如果 git commit 后，想撤回 commit (此时尚未执行 git push 操作)，可以执行 git reset HEAD^

(3). 版本管理

运行 git status 命令可以查看状态。

用 git diff 这个命令查看上一次对文件做了什么修改。

（4）. 版本回退

可以使用 git log 查看所有提交版本信息。

可以使用 git reset HEAD^命令，把当前版本回退到上一个版本。上一个版本就是 HEAD^，上上一个版本就是 HEAD^^，当然往上 100 个版本写 100 个^比较容易数不过来，所以写成 HEAD~100。

如果命令窗口没有关掉使用命令 git reset –hard 〈版本号〉即可回到未来版本，其中版本号写前几位即可。

关掉命令窗口后找不到版本号，使用 git reflog 可以查看记录。

Git 的版本回退速度之所以快，是因为 Git 在内部有个指向当前版本的 HEAD 指针，当回退版本的时候，Git 仅仅是把 HEAD 指向改变，顺便把工作区的文件更新了。

5. 连接远程仓库

（1）. 新建仓库

在 GitHub 或 Gitee 中创建。

（2）. 创建 SSH 公钥

为什么要使用 SSH Key 呢？因为无论是 Github 还是 Gitee，需要识别出每次推送提交的人员确定是你提交的或者确定是你团队开发小组人员提交的，而不是别人冒充的，如果没有公钥，那无论任何人只要知道了你的仓库地址都可以向你的仓库推送提交，也可以随意修改了，这是非常可怕的。

而 Gitee 支持 HTTPS 协议，所以只要 Gitee 知道你本地的公钥和远程仓库中的公钥是一致的，就可以确定你可以向远程仓库提交推送。

当然 Gitee 允许你添加多个公钥 Key，假定你有若干台电脑，你一会在公司提交推送，一会儿在家提交推送，只要每台电脑的公钥 Key 都添加到 Gitee 远程仓库中，就可以在每台电脑上往远程仓库推送了。

（3）. 从本地 Git 仓库连接远程 Git 仓库，并做推送

Git 全局设置:

```
git config --global user.name "Tiam"
git config --global user.email "2131277807@qq.com"
```

创建 git 仓库:

```
mkdir 21334545678
cd 21334545678
git init
touch README.md
git add README.md
git commit -m "first commit"
git remote add origin https://gitee.com/l5201314q/21334545678.git
git push -u origin "master"
```

已有仓库?

```
cd existing_git_repo
git remote add origin https://gitee.com/l5201314q/21334545678.git
git push -u origin "master"
```

使用命令 git remote add origin "远程仓库的地址"，这个远程仓库地址就是我们上述刚建好远程仓库时的 HTTPS 协议的链接地址，连接远程 Git 仓库。添加后，远程库的名字就是 origin，这是 Git 默认的叫法，也可以改成别的，但是 origin 这个名字一看就知道是远程库，所以还是尽量不要改这个了。

然后我们就直接将本地的内容推送到远程的 learngit 仓库中，使用命令 git push -u origin master

由于推送前远程的 learngit 仓库是空的，我们第一次推送 master 主分支时，加上了-u 参数，Git 不但会把我们本地的 master 主分支推送到远程新的 master 主分支中，还会把本地的 master 分支和远程的 master 分支联系起来了，在以后的推送 push 或拉取 pull 时就可以简化命令了。

（4）. 解除本地 Git 仓库与远程 Git 仓库的关联，非“删除”

如果添加的时候地址写错了，或者就是想删除远程库，可以用 git remote rm <name>命令。使用前，建议先用 git remote -v 查看远程库信息：

然后，可以根据名字删除，比如要删除 origin:

```
git remote rm origin     #删除远程的 origin
```

上述的删除操作，只是解除了本地与远程的绑定关联关系，并不是物理上的删除远程库，远程库本身并不会做任何改变。要想真正删除远程库，需要登录 Gitee，在后台页面找到远程库，才能真正删除。

（5）. 从远程仓库拉取最新内容，更新本地仓库内容（非克隆）

- git fetch

git fetch 相当于是从远程获取最新版本到远程的 master 主分支中，然后将远程的 master 注分支带着最新版的内容拉取到本地仓库，但是不会自动合并本地仓库内，也就是需要自己再次合并：

```
git fetch origin master      #从远程仓库下载所有最新版本的内容
```

然后通过命令查看，下载的最新版的内容与本地仓库的内容有哪些区别：

```
git log -p master..origin/master      #查看日志对比内容
```

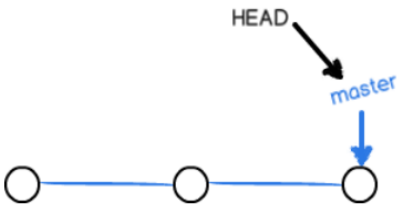
- git pull

git pull 指令相当于是 git fetch 和 git merge 两者的结合体，可以从远程仓库拉取最新版本的内容，并直接合并本地的 master 主分支的内容。

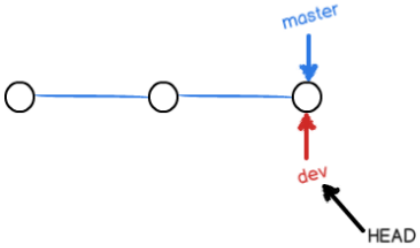
6. 分支管理

什么是分支？有了分支，你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，既安全，又不影响别人工作。合并后再删掉分支，这和直接在 master 分支上工作效果是一样的，但过程更安全。

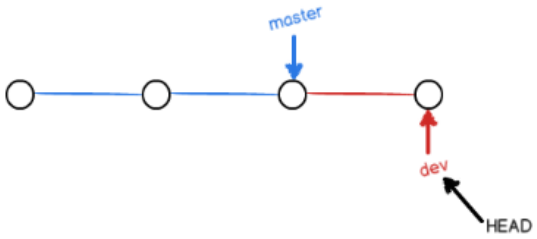
Git 会把每次提交串成一条时间线，这条时间线就是一个分支。git init 时 Git 会自动创建一个 master 分支，即主分支。HEAD 严格来说不是指向提交，而是指向 master，master 才是指向提交的，所以，HEAD 指向的就是当前分支。



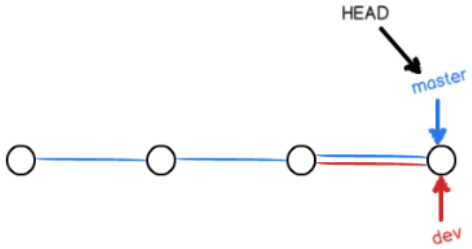
当创建新的 dev 分支时，Git 新建了一个指针叫 dev，指向 master 相同的提交，再把 HEAD 指向 dev，就表示当前分支在 dev 上。



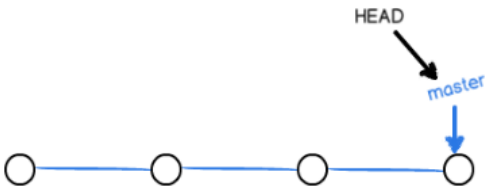
从现在开始，对工作区的修改和提交就是针对 dev 分支了，比如新提交一次后，dev 指针往前移动一步，而 master 指针不变：



假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上。其实就是直接把 master 指向 dev 的当前提交：



合并完分支后也可以删除 dev 分支。删除 dev 分支就是把 dev 指针给删掉，删掉后就剩下一条 master 分支：



(1). 创建分支

首先创建一个 dev 分支，用 git checkout -b 命令创建并切换分支，相当于创建 git branch dev + 切换 git checkout dev。新版本提供 git switch -c <dev>创建并切换到新的分支，git switch <master> 直接切换到已有分支。

可以使用 git branch 命令查看当前分支，该命令会列出所有分支，当前分支前面会标一个*号。

(2). 合并分支

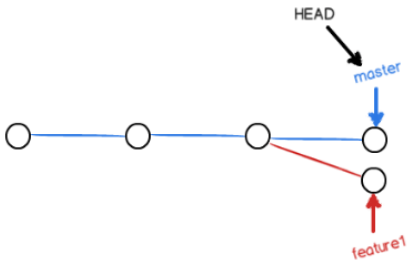
在 dev 分支完成工作后，git checkout 到 master 分支，可以发现此时文件并没有发生任何改变，此时使用 git merge dev 命令可以将 dev 分支合并到 master 分支上，这是文件内容和修改后一样。

（3）. 删除分支

Git 中使用 git branch -d <dev>命令删除分支：如果 commit 提交了，但是还没有合并，这时删除分支的话 Git 会报错提示 test 分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的-D 参数。

（4）. 冲突分支

master 分支和要合并的分支各自都分别有新的提交时：



这种情况下，Git 无法执行“快速合并”，只能试图把各自的修改合并起来。必须手动解决冲突后再提交。git status 可以告诉我们冲突的文件，cat 这个文件 Git 用<<<<<<<, =======, >>>>>>>标记出不同分支的内容，我们需要手动修改后再进行提交。

（5）. 分支管理策略

通常合并分支时，Git 会用 Fast forward 模式，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用 Fast forward 模式，Git 就会在 merge 时生成一个新的 commit，这样，从分支历史上就可以看出分支信息。

合并分支时，加上--no-ff 参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而 fast forward 合并就看不出来曾经做过合并。git merge --no-ff -m "txt" <dev>因为本次合并要创建一个新的 commit，所以加上-m 参数，把 commit 描述写进去。合并后用 git log 看看分支历史。

（6）. Bug 分支

当你接到一个修复 bug 的任务时，当你准备创建一个分支 issue-101 来修复它时，突然想起来，当前正在 dev 上进行的工作还没有提交，Git 提供了一个 stash 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作。

git stash 保护现场

到 bug 分支创建分支修改 bug，bug 修改后合并删除分支，回到之前的分支。

刚才的工作现场存到哪去了？用 git stash list 命令查看：

工作现场还在，Git 把 stash 内容存在某个地方了，有两个办法可以恢复：

用 git stash apply 恢复，但是恢复后，stash 内容并不删除，需要用 git stash drop 删除。

用 git stash pop，恢复的同时把 stash 内容也删了。再用 git stash list 查看，就看不到任何 stash 内容了。可以多次 stash，恢复的时候，先用 git stash list 查看，然后恢复指定的 stash，用命令：git stash apply stash@{0}

在 master 分支上修复了 bug 后， dev 分支是早期从 master 分支分出来的，所以，这个 bug 其实在当前 dev 分支上也存在。那怎么在 dev 分支上修复同样的 bug？重复操作一次？但是 Git 提供了更简单的方法，同样的 bug，要在 dev 上修复，我们只需要把 4c805e2 fix bug 101 这个提交所做的修改“复制”到 dev 分支。注意：我们只想复制 4c805e2 fix bug 101 这个提交所做的修改，并不是把整个 master 分支 merge 过来。Git 提供了一个 cherry-pick <id>命令，让我们能复制一个特定的提交到当前分支：

Git 自动给 dev 分支做了一次提交，注意这次提交的 commit 是 1d4b803，它并不同于 master 的 4c805e2，因为这两个 commit 只是改动相同，但确实是两个不同的 commit。用 git cherry-pick，我们就不需要在 dev 分支上手动再把修 bug 的过程重复一遍。

7. github 问题：master 和 main 分支合并

github 创建仓库后默认分支是 main，而本地创建是 master。解决步骤如下：

先给本地分支 master 改名：git branch -M main

查看所有分支：git branch -a

删除远程分支 master：git push origin --delete master

确认删除情况：git branch -a

切换到当前分支 main，也就要保留下来的分支：git checkout main

合并分支：git merge remotes/origin/main

说明：拒绝合并，需要忽略这个限制，添加“--allow-unrelated-histories”：

```
git merge remotes/origin/main --allow-unrelated-histories
```

提交修改：git push origin main

再次查看分支情况：git branch -a

十、 交叉编译

1. gcc 编译过程

gcc 编译过程分为 4 个阶段：预处理、编译、汇编、链接。

预处理：头文件包含、宏替换、条件编译、删除注释

编译：主要进行词法、语法、语义分析等，检查无误后将预处理好的文件编译成汇编文件。

汇编：将汇编文件转换成二进制目标文件。

链接：将项目中的各个二进制文件+所需的库+启动代码链接成可执行文件。

2. 概念

交叉编译是在一个平台上生成另一个平台上的可执行代码。

比如我们在 ubuntu 上面编写树莓派的代码，并编译成可执行代码，如 a.out，是在树莓派上面运行，不是在 ubuntu linux 上面运行。

交叉编译是相对复杂的，必须考虑如下几个问题：CPU 架构；字节序：大端和小端；浮点数的支持；应用程序二进制接口。

3. 命名规则

交叉编译工具链的命名规则一般为：

$arch [-$vendor] -$os [-[gnu][eabi][hf]]-gcc

arch - 体系架构，如 arm，mips 等，不可省略

vendor - 工具链提供商，可省略

os - 目标操作系统，不可省略

eabi - 嵌入式应用二进制接口可选的参数包括：

abi：二进制应用接口。

eabi：嵌入式二进制应用接口，主要针对嵌入式平台。

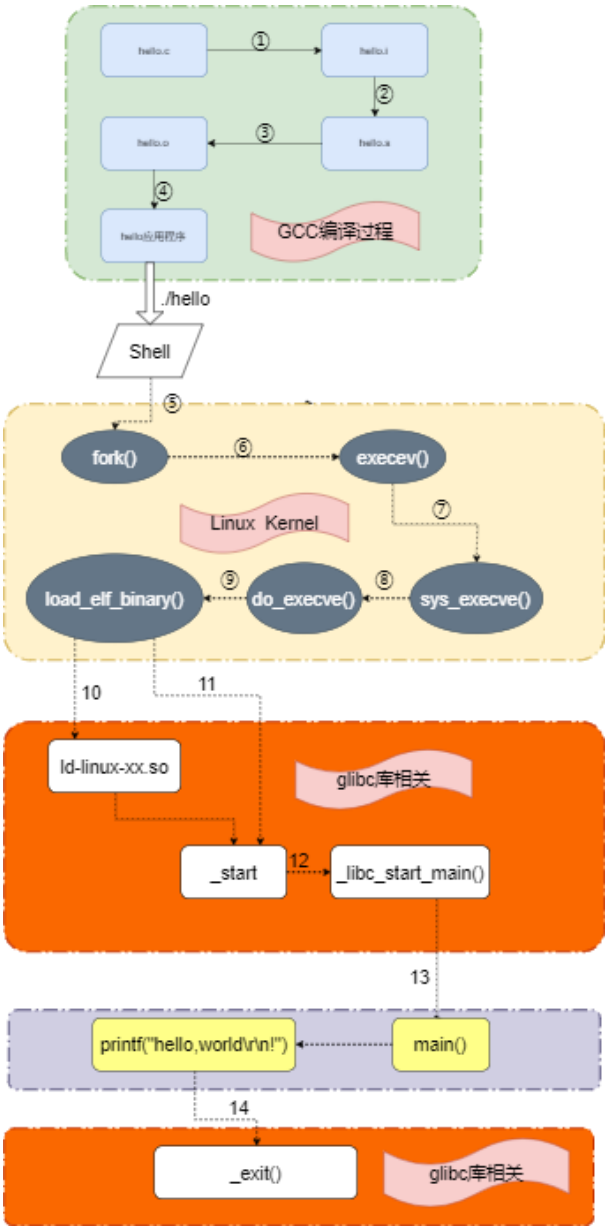
gnu：加 gnu 表示编译器使用的是 gnu glibc 的库

el：表示使用软浮点处理单元(softfp)。其实在 armel 中，关于浮点数计算的约定有三种。以 gcc 为例，对应的 -mfloat-abi 参数值有三个：soft,softfp,hard。soft 是指所有浮点运算全部在软件层实现，效率当然不高，会存在不必要的浮点到整数、整数到浮点的转换，只适合于早期没有浮点计算单

元的 ARM 处理器；softfp 是目前 armel 的默认设置，它将浮点计算交给 FPU 处理，但函数参数的传递使用通用的整型寄存器而不是 FPU 寄存器；hard 则使用 FPU 浮点寄存器将函数参数传递给 FPU 处理。需要注意的是，在兼容性上，soft 与后两者是兼容的，但 softfp 和 hard 两种模式不兼容，armel 使用 softfp，因此将 hard 模式的 armel 单独作为一个 abi，称之为 armhf

hf：表示使用硬件浮点处理单元（hard）

十一、 Linux 下 helloworld 执行过程



1. 预处理 hello.c，主要是处理程序里面的文件包含、处理宏定义、条件编译。

2. 把 c 文件编译成为汇编文件(.s)，其中进行了词法分析，语法分析，语义分析、生成中间代码、对代码进行优化等工作。

3. 把汇编文件(.s)编译成可重定位文件(.o)。

4. 把可重定位文件(.o)链接成为可执行文件，其中链接可分为静态链接和动态链接

静态链接：在编译阶段把所有用到的库打包到自己的可执行程序中，具有较好的兼容性，不依赖外部环境，但是生成的程序比较大。

动态链接：在应用程序运行时，链接器去加载外部的共享库，并完成共享库和动态编译程序之间的链接。不同的程序可以共用代码库，节省内存空间。

5. 控制台输入./hello 命令后，Shell 会调用 fork()函数创建一个新的进程来执行该程序。

6. exeve()函数可以理解为向新建的进程，填充可执行程序(hello)。

7. sys_execve()函数为 linux 系统调用，被 exeve()函数调用，这里的系统调用可以理解为是操作系统系统开放给用户的最底层接口。

8. do_exeve()函数是 sys_execve()函数的核心。load_elf_binary()函数会去文件系统中读取 hello 程序到内存，然后判断它是否是动态链接的可执行程序，如果不是，则进一步判断是否是静态链接的文件。

9. ld-linux-xx.so 是 glibc 库中的动态连接器。如果 hello 程序是动态链接程序，该动态链接器会去加载共享库，并完成共享库和程序的链接工作， 然后准备真正开始执行 hell 程序。

10. 相反，如果 hello 程序是静态编译的程序，则无需再加载链接共享库，直接开始准备执行 hello 程序。

11. 第 10 和 11 步分别执行之后.都会开始执行 hello 程序,_start 是程序的真正入口，而该符号在 glibc 中。也就是说程序的真正入口在 glibc。

12. __libc_start_main()也是 glibc 中的函数，用于在执行用户程序前进行一些初始化工作。

13. 调用用户程序中的 mian()函数，开始执行 printf 打印函数。

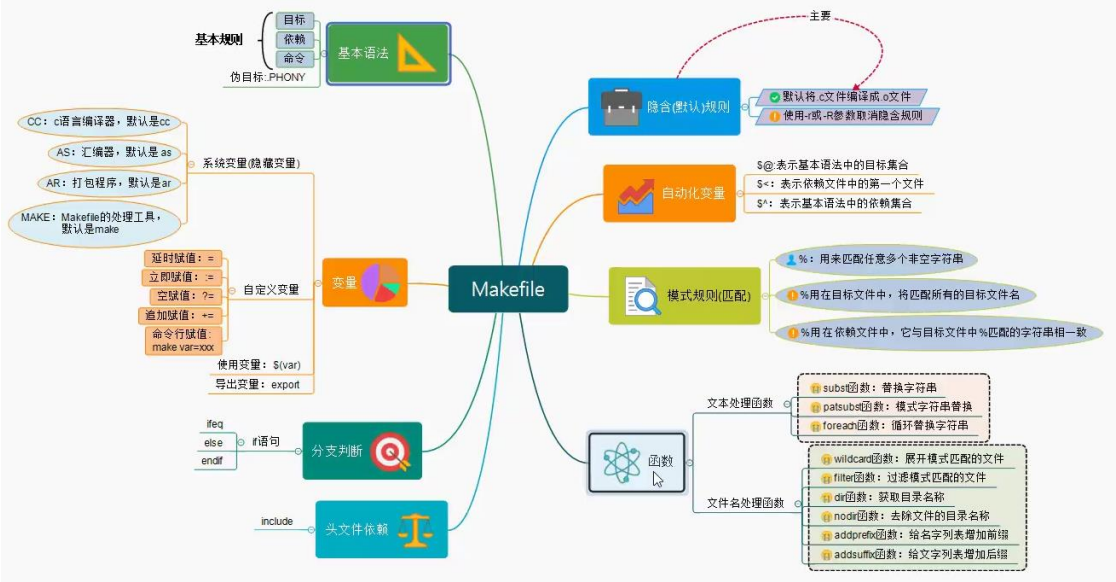
14. 程序执行完了之后，调用 glibc 库中的_exit()函数，来结束当前进程。

十二、 Makefile

1. 什么是 Makefile

一个工程中的源文件不计其数，按期类型、功能、模块分别放在若干个文件中，MakeFile 文件定义了一系列的规则来制定哪些文件需先要编译、哪些文件需要后编译、哪些文件需要重新编译、甚至于进行更加复杂的操作，因为 Makefile 文件就像是一个 shell 脚本一样，也可以执行操作系统的命令

Makefile 带来的好处就是 自动化编译，一旦写好，只需要一个 make 命令，整个工程完全自动化编译，极大提高了软件开发效率，make 是一个命令工具，是一个解释 Makefile 文件中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，例如 Delphi 的 make、Visual C++的 nmaek、Linux 下 Gun 的 make。



2. Makefile 三要素

目标、依赖、命令

```
.PHONY:targetb
targeta:targetb targetc
        echo "targeta"

targetb:
        echo "targetb"

targetc:
        echo "targetc"
```

依赖

命令

目标

伪目标 .PHONY：伪目标只是一个标签，targetb 是个伪目标没有依赖文件，只有用 make 来调用时才会执行，当目录下有与 make 命令同名的文件时执行 make 命令就会出现错误。解决办法就是使用伪目标

3. 工作原理

命令在执行之前，需要先检查规则中的依赖是否存在。

（1） 如果存在，执行命令。

（2） 如果不存在，向下检查其它的规则，检查有没有一个规则是用来生成这个依赖的，如果找到了，则执行该规则中的命令。

检测更新，在执行规则中的命令时，会比较目标和依赖文件的时间。

（1） 如果依赖的时间比目标的时间晚，需要重新生成目标。

（2） 如果依赖的时间比目标的时间早，目标不需要更新，对应规则中的命令不需要被执行。

即依赖文件发生改变则要重新执行。

4. 变量和模式匹配

（1） 系统变量：

CC：指代 C 编译器

C++:C++编译器的名称

AS：指代汇编器

MAKE:指代 make 工具

（2） 自定义变量

=，延迟赋值，只有值被引用时才会被赋值。

:=,立即赋值，假如某变量在前面已经定义赋值过，则将本次赋值作为最新的变量值，和 C 语言中的=类似。

？=，空赋值，当某变量前面已经定义赋值过，则不执行本次定义赋值，否则执行本次赋值，只有变量为空的时候才执行赋值。

+=，追加赋值，旧值保持不变，将新值黏贴到旧值后面。

（3） 模式匹配

%，匹配任意多个非空字符。

*、？、…,用法同 shell

(4) 自动化变量

$* 不包含扩展名的目标文件名称

$+ 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件

$< 第一个依赖文件的名称

$? 所有时间戳比目标文件晚的的依赖文件，并以空格分开

$@ 目标文件的完整名称

$^ 所有不重复的目标依赖文件，以空格分开

$% 如果目标是归档成员，则该变量表示目标的归档成员名称

5. 默认规则

.o 文件默认使用.c 文件来进行编译。

6. 条件分支

var1 和 var2 值相等则执行 f1 否则执行 f2

```
ifeq(var1,var2)
    f1
    …
else
    f2
    …
```

var1 和 var2 值不相等则执行 f1 否则执行 f2

```
ifneq(var1,var2)
    f1
    …
else
    f2
    …
```

7. 常用函数

(1) patsubst —— 按格式替换字符

patsubs 函数的作用是使用目标字符（格式）替换源字符（格式），函数返回替换以后的结果，常常搭配通配符 % 使用，% 表示任意长度的字串，如果 <src_pattern> 和 <dst_pattern> 都包含 % ，那么此时 % 表示的字符内容是一样的。

将 <text> 中的 源格式 <src_pattern> 替换成目标格式 <dst_pattern>

ret = $(patsubst <src_pattern>,<dst_pattern>,<text>)

示例：将所有满足后缀为 .cpp 格式的字符串替换为 后缀为 .o 格式

$(patsubst %.cpp,%.o,add.cpp bar.cpp)

(2) notdir —— 获取文件路径非目录部分

获取一个文件路径的非目录部分，也可以理解为获取文件名（含后缀），本质是获取最后一个反斜杠 '/' 之后的内容。如果没有反斜杠，直接返回本身。

示例：$(notdir src/foo.c sum.txt)

结果：foo.c sum.txt

(3) wildcard —— 获取指定格式的文件列表

原型（不同格式之间使用空格隔开）：$(wildcard <pattern...>)

示例：获取当前目录下所有的 .cpp 文件和 test 目录下所有的 .cpp 文件

$(wildcard *.cpp test/*.cpp)

结果：main.cpp test/sub.cpp

(4) foreach —— 循环函数

$(foreach <var>,<list>,< expression >)

将 <list> 中的参数逐一取出放到 <var> 变量中，然后再执行 <expression> 中的表达式。

循环执行中：每执行一次循环都会返回一个字符串，foreach 循环会将返回的字符串汇总，不同字符串通过空格分隔

循环执行结束：当整个循环结束的时候，返回汇总的字符串（不同字串以空格分隔）

示例：names := a b c d

files := $(foreach n,${names},$(n).o)

结果：a.o b.o c.o d.o

8. **通用 Makefile 文件示例**

```
####################选择编译器##################
ARCH ?= x86
ifeq ($(ARCH),x86)
    CC = gcc
else
    CC = arm-linux-gnueabihf-gcc
endif
######################目标######################
TARGET = mp3
#######生成成文件、源文件、包含文件名称#######
BUILD_DIR = build
SRC_DIR = module1 module2
INC_DIR = include

#########获取 SRC_DIR 文件下所有的.h 文件########
INCLUDES = $(foreach dir, $(INC_DIR), $(wildcard $(dir)/*.h))
###获取 SRC_DIR 文件下所有的.c 文件###
SOURCES = $(foreach dir, $(SRC_DIR), $(wildcard $(dir)/*.c))

#############将所有.c 后缀换成.o 后缀#############
OBJS = $(patsubst %.c, $(BUILD_DIR)/%.o, $(notdir $(SOURCES)))
###gcc 指定头文件路径的参数-I###
CFLAGS = $(patsubst %,-I%,$(INC_DIR))

###vpath 的关键字，它用于定义 make 的查找路径###
VPATH = $(SRC_DIR)

#############目标 gcc main.o -o mp3#############
$(BUILD_DIR)/$(TARGET):$(OBJS)
    $(CC) $^ -o $@

###依赖 gcc -c  main.c -o main.o -I ./include###
$(BUILD_DIR)/%.o:%.c $(INCLUDE) | create_build
    $(CC) -c $< -o $@ $(CFLAGS)

.PHONE:create_build
create_build:
    mkdir -p $(BUILD_DIR)

.PHONE:clean
clean:
    rm -r $(BUILD_DIR)
```

十三、 一切皆文件

1. 概念

“一切皆文件”，指的是，对所有文件（目录、字符设备、块设备、套接字、打印机等）操作，读写都可用 fopen()/fclose()/fwrite()/fread()等函数进行处理。屏蔽了硬件的区别，所有设备都抽象成文件，提供统一的接口给用户。虽然类型各不相同，但是对其提供的却是同一套操作界面。更进一步，对文件的操作也可以跨文件系统执行。

2. 虚拟文件系统 (Virtual File System，简称 VFS)

linux 支持多种文件系统（如 vfat,ext2,ext3 等），为了方便管理，在所有这些文件系统上面提供了一层抽象，即虚拟文件系统。虚拟文件系统为各类文件系统提供了统一的操作界面和应用编程接口，也就是说，不论是什么类型的文件系统，都必须提供符合 VFS 标准的接口。

3. 类型:

普通文件 # xxx.log

目录 # /usr/ /home/

字符设备文件 # /dev/tty 的属性是 crw-rw-rw- ，注意前面第一个字符是 c ，这表示字符设备文件,比如猫等串口设备

块设备文件 # /dev/hda1 的属性是 brw-r----- ，注意前面的第一个字符是 b，这表示块设备，比如硬盘，光驱等设备

套接字文件 # /var/lib/mysql/mysql.sock srwxrwxrwx

管道 # pipe

符号链接文件 # softlink...

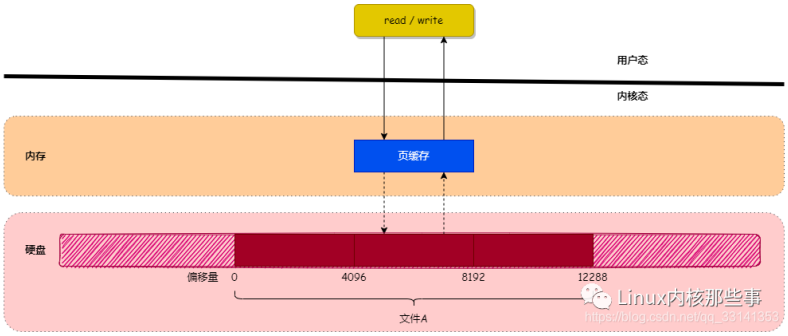
4. 文件描述符和打开模式

(1) 系统 IO 编程

打开文件：open　　写文件：write

读文件：read　　移动文件指针的位置:lsenk

关闭文件：close　　页缓存和回写：sync



当用户对文件进行读写时，实际上是对文件的页缓存进行读写。所以对文件进行读写操作时，会分为以下两种情况进行处理：

当从文件中读取数据时，如果要读取的数据所在的页缓存已经存在，那么就直接把页缓存的数据拷贝给用户即可。否则，内核首先会申请一个空闲的内存页（页缓存），然后从文件中读取数据到页缓存，并且把页缓存的数据拷贝给用户。

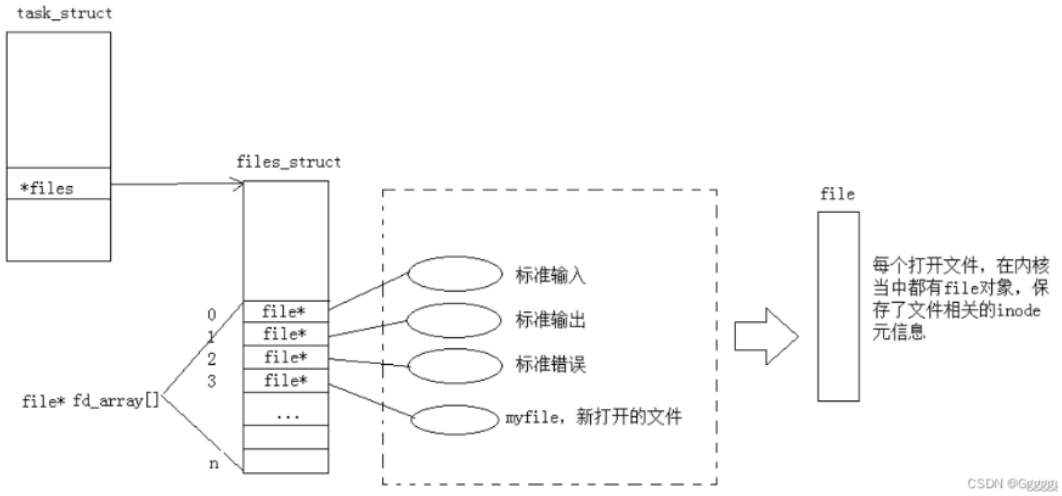
当向文件中写入数据时，如果要写入的数据所在的页缓存已经存在，那么直接把新数据写入到页缓存即可。否则，内核首先会申请一个空闲的内存页（页缓存），然后从文件中读取数据到页缓存，并且把新数据写入到页缓存中。对于被修改的页缓存，内核会定时把这些页缓存刷新到文件中。

sync 函数：强制把修改过的页缓存区数据写入磁盘

伪代码：

```
int fd;
fd = open(filename,flags,mode);
lsenk(fd,offset,whence);
write(fd,buf,write_len);
read(fd,buf,read_len);
close(fd);
```

(2) 文件描述符



在 Linux 内核中，PCB 为 task_struct。其中，task_struct 中就包含了一个文件结构体指针 files_struct* fs。该指针就是指向的文件结构体。重点是该文件结构体中有一个指针数组 file* fd_arry[]（文件映射表），该指针数组指向的就是我们所打开的文件。我们所新打开的文件 fd 的值是文件描述表中最小的为空的位置。

文件描述符 fd 是进程中 file_struct 结构体成员 fd_array 的数组下标。

(3) 打开模式

- 主模式（互斥）

  O_RDONLY：只读模式

  O_WRONLY：只写模式

  O_RDWR：读写模式

- 从模式

  O_CREATE：当文件不存在时，需要去创建文件

  O_APPEND：追加模式(将文件读写位置设置到文件末尾)

  O_DIRECT：直接 IO 模式（write/read 读写数据不经过页缓存区）

  O_SYNC：同步模式（不需要调用 sync 函数）

  O_NOBLOCK：非阻塞模式（文件 IO 五大模式之一）

  文件 IO 五大模式：阻塞模式（无法正常读取时处于休眠状态）

  非阻塞模式（不处于休眠状态）

  IO 多路复用

  异步 IO

  信号驱动 IO

(4) 标准 IO 函数

标准 C 库 I/O 函数在读写的时候，中间有一个缓冲区，而在 Linux 系统中，底层的系统调用并不直接使用标准 C 库的缓冲机制，而是通过一些系统级别的缓冲区来进行 I/O 操作。Linux 提供了一种称为"buffered I/O"（缓冲 I/O）的机制，通过在内核中使用缓冲区来优化读写操作。这使得在用户空间的应用程序调用系统调用时，实际的 I/O 操作可能并不是立即发生的。如果中间有缓冲区的话在进行读写操作的时候会先存到缓冲区，再刷新到磁盘，它比直接逐条读写到磁盘效率要高。

打开文件：fopen　　写文件：fwrite

读文件：fread　　移动文件指针的位置:fsenk

关闭文件：fclose

强制把 IO 缓存区的数据写入页缓存区：fflush