# Linux基本命令

## 1 Linux用户：

Linux用户分为超级用户（root）、普通用户。

超级用户：可以再 linux 系统下做任何事情，不受限制。

普通用户：在 linux 下做有限的事情。

超级用户的命令提示符是“#”，普通用户的命令提示符是“$”。

**命令：su [用户名]**

功能：切换用户。

例如，要从 root 用户切换到普通用户 user1，则使用 su user1su。要从普通用户 user 切换到 root 用户则使用 suproot（root可以省略），此时系统会提示输入 root 用户的口令。

超级用户到普通用户不需密码，普通用户和普通用户需要输入对应密码，，普通用户到超级用户需超级密码。超级用户只有一个拥有所有权限。用户名为root。只有root权限才能添加用户。

## 2 添加用户。

**命令：useradd –m 用户名 –s /bin/bash**

功能：添加一个普通用户。使用的shell脚本解析器是/bin/bash。

**命令：userdel –r 用户名**

功能：删除不在使用的普通用户。

例如，要想添加一个普通用户 user1 ，则可以使用useradd user1。使用该命令后，系统会在目录“/home”下建立一个名为 user1 的目录。加 -m 才会创建目录。输入命令后按TAB则联想命令帮助。当然也可以在其他目录下添加用户。但都会集成到/home中。添加用户时候，默认使用系统解析器，也可以指定 shell脚本解析器为 –s /bin/bash。

## 3 设置密码

命令：passwd 用户名

功能：设置或修改用户名的密码。

例如，我们要给刚才创建的 user1 用户设置一个密码123456，则使用 passwd user1，然后系统会提示你输入新密码。

## 4 查看文件或目录

**命令：ls [选项] [目录或文件]**

功能：对于目录，该命令列出该目录下的所有子目录与文件。对于文件，将列出文件名以及其他信息。Linux文件系统不是根据后缀名来执行文件，而是根据此文件是否有可执行权限。

常用的选项有：

-a 显示指定目录下所有子目录与文件。

例如 列出“/root/home”目录下的所有子目录及文件，则使用ls –a /root/home 。包括隐藏文件，以“.”开头。

-l 列出指定目录下所有目录及文件的详细信息。

例如 列出“/root/home”目录下的所有子目录及文件，则使用ls –l /root/home 。不显示隐藏文件。每行列出的详细信息依次是：

权限与文件类型 链接数 文件所有者 文件所属组 文件大小 最近修改时间 文件名字

drwxr-xr-x 2 liao20081228 liao20081228 4096 Jan 15 20:25 Music

-rw------- 1 liao20081228 liao20081228 82 Jan 17 04:37 .xsession

使用 ls –l 命令显示的信息中 ，开头是由 10 个字母构成的字符串，其中第一个字符表示文件类型，它可以是下列类型之一：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 符号 | 意义 | 符号 | 意义 | 符号 | 意义 |
| - | 普通文件 | l | 符号链接=快捷图标 | c | 字符设备，设备数据以单个字符为单位 |
| d | 目录 |  |  | p | 命名管道 |
| s | Socket文件 |  |  | b | 块设备文件，设备数据以块为单位 |

后面的 9 个字符表示文件的访问权限，分为 3 组，每组 3位。第一组表示文件创建者的权限，第二组表示同组用户的权限，第三组表示其他用户的权限。每一组的三个字符分别表示对文件的读、写、执行权限。各权限如下：r（4读）、w（2写）、x（1执行）、\_（没有设置权限0）。每一组可以用一个数字表示，例如 r\_x :5 , rw\_:6，R\_\_:4，那么这三组就可以用3个数字表示，例如rwxr\_xr\_x:755 ，rw\_r\_\_r\_\_:644。

ls 输出内容是有颜色的，比如：目录是蓝色，压缩文件是红色的显示，如果没有颜色，可以加上参数--color=never 表示输出没有彩色 ， 而 --color=auto 表示自动 ，--color=always 表示始终有颜色。如果需要更加详细的参数描述，可以通过如下方式获得 ls 的帮助：man ls。

-al 显示指定目录下所有子目录与文件详细信息，包括隐藏文件。

注意：Linux的选项如 -a，-l 可以组合使用 -al。其他命令的选项也一样。

## 5 改变工作目录。

**命令：cd 目录名**

功能：改变工作目录。将当前工作目录改变到指定的目录下，

例如要切换当前目录到“/home/user/0718”目录，则使用cd /home/user/0718。

Linux对于当前工作目录下的子目录或者文件的操作 可以直接使用目录名而不用加/。

常用的切换目录命令：

cd .. 到父目录，注意后面没有斜杠

cd / 到根目录，默认就是斜杠

cd ~ 到用户主目录下~ 与直接执行 cd 效果一样，~就等效于/home/用户名.注意后面没有斜杠

cd – 到上一次目录.后面不能加子目录

cd . 到当前目录，注意后面没有斜杠

## 6 显示当前工作目录。

**命令 ： pwd**

功能 ： 显示用户当前所在的目录。例如当我们使用命令

cd /home/user/0718 时，再使用命令 pwd 则命令行会显示/home/user/0718。

## 7 创建目录

**命令：mkdir [选项] dirname**

功能：在当前目录下创建一个名为 “dirname”的目录。

例如要在当前目录下创建一个名“07181”的目录，则使用命令：mkdir 07181。系统就会在当前目录下，创建一个 07181 的目录，此时可以使用 ls –l 查看。

## 8 删除空目录

**命令：rmdir [选项] dirname**

功能：在当前工作目录下删除目录名为“dirname”的空的子目录。

此时该子目录必须是个空目录。我们刚才创建了一个空目录07181，如果我们想把它删掉，则使用 rmdir 07181 。此时再使用 ls –l 列举一下，这时发现 07181 已经被删掉了。如果使用该命令删除一个非空的目录，则删除失败。

## 9 拷贝文件或目录

**命令： cp [选项] 源文件或目录 目标文件或目录**

功能：把指定的源文件复制到目标文件或把多个源文件复制到目标目录中。

常用参数：

-f若目标目录中存在与源文件同名的文件，则直接覆盖，不提示 。

例如将当前目录下的 main.c 文件拷贝到“/home/user/0718”下，并且若存在同名的则进行覆盖，使用：cp–f ./main.c /home/user/0718 。如果在拷贝的同时将源文件重命名，例如将当前目录下的main.c 文件拷贝到“/home/user/0718”目录下并命名为 main1.c，则使用：cp –f ./main.c /home/user/0718/main1.c。

-i当目标文件中存在于源文件同名的文件，copy时系统会提示是否进行覆盖。

如上例，若在拷贝过程中，目标文件中存在与源文件同名的文件，需要提示是否覆盖，则只需要将上例中的-f 改为 –I 即可。例如cp –i ./main.c /home/user/0718 。cp –i ./main.c /home/user/0718/main1.c。

-r如果要拷贝的是一个目录，此时将同时拷贝该目录下的子目录和文件。

此时目标文件必须为一个目录。例如 ，将 ”/home/user/0718” 目录下的所有文件及目录拷贝到“/home/user1”目录下，则使用cp –r /home/user/0718 /home/user1 。(重点)

## 10 移动文件或目录。

**命令：mv[选项] 源文件或目录 目标文件或目录**

功能：视 mv 命令中第二个参数类型的不同（是目标文件还是目标目录），mv 命令将文件重命名或将其移至一个新的目录中。当第二个参数类型是文件时，mv 命令完成文件重命名，此时，源文件只能有一个（也可以是源目录名），它将所给的源文件或目录重命名为给定的目标文件名。当第二个参数是已存在的目录名称时，源文件或目录参数可以有多个，mv 命令将各参数指定的源文件均移至目标目录中。

注意：在跨文件系统移动文件时，mv 先拷贝，再将原有文件删除，而链至该文件的链接也将丢失。

参数：

-i 如果在移动的过程中存在重名的，则进行提示是否覆盖。

-f若果在移动的过程中存在重名的，则直接进行覆盖，不会给出提示。

例如 要将 “/home/user/0718”下的 main.c 文件重命名为main.cpp，则使用

mv /home/user/0718/main.c /home/user/0718/main.cpp。要将“ /home/user/0718 ”下 的所有 内 容 移 动 到“/home/user/0719”,则使用mv –f /home/user/0718 /home/user/0719。

## 11 删除文件或目录

**命令：rm [选项] 文件或目录**

功能：在 linux 中创建文件很容易，系统中随时会有文件变得过时且毫无用处。用户可以用 rm 命令将其删除。该命令的功能为删除一个目录中的一个或多个文件或目录，它也可以将某个目录及其下的所有文件及子目录均删除。对于链接文件，只是删除了链接，原有文件均保持不变。如果删除时没有-r 选项则不会删除目录。

参数：

-f删除过程中不会给出提示。

-i 删除过程中会给出交互式提示。

-r 如果删除的是一个目录，则将该目录下的目录及子目录均删除掉。

例如要删除 “/home/user/0718”目录下的 main.cc main.exe则使用，rm -f /home/user/0718/main.cc /home/user/0718/main.exe若果要删除“/home/user/0718”这个目录，则使用rm –rf /home/user/0718。

## 12 查看文件内容(不是目录)

### 12.1 显示文件内容

**命令： cat[选项] [文件]**

功能： 查看目标文件的内容

参数：

-b 对非空输出行编号

-E 在每行结束处显示$

-n 对输出的所有行编号

-s 不输出多行空行。

例如 要查看当前目录下的 main.cc 的内容则使用，cat main.cc。

### 12.2 标准的输入输出与重定向：

文件描述符是一个整数,它代表一个打开的文件,标准的三个描述符号:

标准输入：一般指键盘,描述符为：0

标准输出：一般指屏幕输出,描述符为：1

错误输出：也是屏幕,描述符为：2

重定向符号:(相当于c语言的文件输出和输出，把结果输出到文件或从文件中读入)

< 文件名 从指定文件输入

> 文件名 输出到指定文件

>> 文件名 添加到指定文件

2> 文件名 错误信息输出到指定文件

&>文件名 错误和信息输出到指定文件

Cat>file1<file2或者Cat <file2>file1 从file2读入再输出到file1，会覆盖原有内容 。

./main text1.txt >text5.txt 2>&1。将错误信息输出到&1

cat 常常与重定向一起使用。其中>表示创建，>>表示追加,<<表示以什么结束

如果 cat 的命令行中没有参数，它就会从标准输入中读取数据，并将其送到标准输出。

linux 中创建空文件的四种方式：

方式 1： echo > a.txt（会有一个字节）

方式 2： touch b.txt

方式 3： cat> c.txt 按 ctrl+c 组合键退出；或 Ctrl+d

方式 4： vim d.txt 进入之后：wq 退出。

查看用户列表

cat /etc/passwd 可以看到当前所有的用户。

## 13 显示文件内容的前几行

**命令：head –n 行数值 文件名**

功能：显示目标文件的前几行。

例如 要显示 当前目录下 main.cc 的前 10 行，则使用Head –n 10 main.cc。

## 14 显示文件的后几行

**命令： tail –n 行数值 文件名**

功能：显示目标文件的最后几行。

例如 要显示“/home/user/0718/”目录下的 main.cc 文件的最后 10 行。则使用 tail –n 10 /home/user/0718/main.cc。

## 15 管道与命令替换

管道：是重定向的一种，就像一个导管一样，将一个程序或命令的输出作为另一个程序或命令的输入。

命令1| 命令2

命令替换：和重定向有点相似，但区别在于命令替换是将一个命令的输出作为另一个命令的参数。

命令1 `命令2`或 命令1 $(命令2)。

举例：首先列出当前的所有信息，并重定向到 aa 文件中：ls | cat >aa 或 ls >aa，然后，通过命令替换，列出 aa 文件中所有的文件信息：ls –l `cat aa` 或者用 ls –l $(cat aa)

## 16 文件或目录的创建掩码

**umask** 指文件或目录创建时要去掉的一些权限。先用umask设置掩码后再新建文件或文件目录

普通用户缺省时 umask 的值为 0002，超级用户为 0022。0002 表示创建目录时所有者的权限不去掉， 所属组权限不去掉，其他组权限写属性去掉。如果给出的mode已经去掉了掩码指出的权限，则不会再去除。

实际文件权限=给出的权限 & ~掩码

创建一文件以后，普通用户缺省的权限为0664超级用户：0644

创建一目录以后 ，普通用户缺省的权限为0775超级用户：0755

可以通过 umask 查看默认的缺省的掩码值。通过 umask 001修改掩码值。

## **17 改变目录或文件的权限**

只有文件所有者或者超级用户可以修改文件权限。

**命令：chmod**

功能：用于改变文件或目录的访问权限。用户用它控制文件或目录的访问权限。

语法：该命令有两种用法。一种是包含字母和操作符表达式的文字设定法；另一种是包含数字的数字设定法。

我们利用 ls –l 长格式列出文件或目录的基本信息如下：

文件类型与权限 链接数 文件所有者 文件属组 文件大小 最近修改的时间 名字

对于权限，有第一组表示文件所有者的权限，第二组表示同组用户的权限，第三组表示其他用户的权限。每一组的三个字符分别表示对文件的读、写和执行权限。可以通过 chmod 来修改权限。

### 17.1 文字设定法

**命令：chmod [who][+|-|=][mode] 文件名**

功能：修改指定文件名中 who的权限增加/去除/赋值为 mode

参数：

who 操作对象，可是下述字母中的任一个或者它们的组合：

u 表示“用户（user）”，即文件或目录的所有者。

g 表示“同组（group）用户”，即与文件所有者有相同组 ID 的所有用户。

o 表示“其他（others）用户”。

a 表示“所有（all）用户”。它是系统默认值。

操作符号可以是：

+ 添加某个权限。

- 取消某个权限。

= 赋予给定权限并取消其他所有权限（如果有的话）。

设置 mode 所表示的权限可用下述字母的任意组合(当组合的时候，who 不能少)：

r 可读

w 可写

x 可执行

文件名：以空格分开的要改变权限的文件列表，支持通配符。

在一个命令行中可给出多个权限方式，其间用逗号隔开。

例如：chmod g+r，o+r example 使同组和其他用户对文件 example 有读权限。

### 17.2 数字设定法(常用)

我们必须首先了解用数字表示的属性的含义：

0 表示没有权限，1 表示可执行权限，2 表示可写权限，4 表示可读权限，然后将其相加。所以数字属性的格式应为 3 个从 0 到 7 的八进制数，其顺序是（u）（g）（o）。

例如，如果想让某个文件的所有者有"读/写"二种权限，需要把 4（可读）+2（可写）＝6（读/写).

数字设定法的一般形式为：

**chmod [mode] 文件名**

例1：

$ chmod ug+w，o-x text

即设定文件 text 的属性为：

文件所有者（u） 增加写权限

与文件所有者同组用户（g） 增加写权限

其他用户（o） 删除执行权限

例 2：

$ chmod 644 mm.txt

文件所有者（u）inin 拥有读、写权限

与文件所有者同组人用户（g） 拥有读权限

其他人（o） 拥有读权限

## 18 查找文件

**命令：find 起始目录 查找条件 操作**

功能：在指定目录结构中搜索问价，并执行指定的操作。

该命令的查找条件可以是一个逻辑运算符 not、and、or 组成的复合条件。

-a：逻辑与，在命令中用-a 表示，表示只有当所给的条件都满足时，查找条件才满足。

例如在“/home/user”目录下查找名为 0718 类型是一个目录的文件。则使用find /home/user –name 0718 –a -type d

-o：逻辑或，在命令中用-o 表示，表示只要所给的条有一个满足，查找条件就满足。

例如在“/home/user”目录下查找名字为 main.cc 或名字为 main.c 的文件。则使用find /home/user -name main.cc -o –name main.c。

!：逻辑非，在命令中用！表示查找不满足所给条件的文件。

例如在“/home/user“下查找名字不是main.c 的文件，则使用find /home/user ! –name main.cc 。

常用的查找条件有：

### 18.1 根据名称和文件属性查找。

-name ’字串’ 查找文件名匹配所给字串的所有文件，字串内可用通配符\*、?、[ ]。

-gid n 查找属于 ID 号为 n 的用户组的所有文件。

-uid n 查找属于 ID 号为 n 的用户的所有文件。

-group ’字串’ 查找属于用户组名为所给字串的所有的文件。

-user ’字串’ 查找属于用户名为所给字串的所有的文件。

-empty 查找大小为 0 的目录或文件。

-perm 权限 查找具有指定权限的文件和目录，权限的表示可以如 711，644。

-size n[bckw] 查找指定文件大小的文件，n 后面的字符表示单位，缺省为 b，代表 512字节的块。

-type x 查找类型为 x 的文件，x 为下列字符之一：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 符号 | 意义 | 符号 | 意义 | 符号 | 意义 |
| - | 普通文件 | l | 符号链接=快捷图标 | c | 字符设备，设备数据以单个字符为单位 |
| d | 目录 |  |  | p | 命名管道 |
| s | Socket文件 |  |  | b | 块设备文件，设备数据以块为单位 |

### 18.2 根据时间查找

- amin n 查找 n 分钟以前被访问过的所有文件。（+表示 n分钟之前，-表示 n 分钟之内，+号和-号都不能省略）

- cmin n 查找 n 分钟以前文件状态被修改过的所有文件。

- mmin n 查找 n 分钟以前文件内容被修改过的所有文件。

- atime n 查找 n 天以前被访问过的所有文件。

- ctime n 查找 n 天以前文件状态被修改过的所有文件。

- mtime n 查找 n 天以前文件内容被修改过的所有文件。

### 18.3 可执行的操作。

- exec 命令名称 {}

对符合条件的文件执行所给的 Linux命令，而不询问用户是否需要执行该命令。{}表示命令的参数即为所找到的文件；命令的末尾必须以“ \;”结束。

例如，在“/home/user“目录下查找名为 main.c 文件并显示这些文件的详细信息，则使用

find /home/user –name main.c –exec ls –l {} \;或者

- ok 命令名称 {}

对符合条件的文件执行所给的 Linux命令，与 exec 不同的是，它会询问用户是否需要执行该命令。

## 19 查找文件内容

**命令：grep [选项][查找模式][文件名 1，文件名 2，…]**

功能：grep 过滤器查找指定字符模式的文件，并显示含有此模式的所有行。被寻找的模式称为正则表达式。

常用的一些正则表达式

^ :以什么开头 ，例如 ls –l | grep ^d 显示当前目录下的所有以 d开头的子目录的详细信息。

$ :以什么结尾 。例如 ls –l | grep c$ 显示当前目录下以 c 结尾的文件。

常用的参数：

-E 每个模式作为一个扩展的正则表达式对待。等价于 egrep，使用最新的正则表达式

-F 每个模式作为固定的字符串对待

-c 只显示匹配行的数量。

-i 比较式不区分大小写。

-n 在输出前加上匹配串所在的行号。

## 20 文档打包

**命令：tar[主选项+辅选项] 目标文档(可省略) 源文件或目录**

功能：把一大堆的文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于网络传输是非常有用的。tar不是压缩软件。

常用参数：

-c：创建新的档案文件,后缀名为.tar

-r: 要把存档的文件追加到档案文件的末尾。tar -rf \*.tar test

-x：从档案文件中释放文件。

-f：使用档案文件或设备。这个参数必须位于参数的最后一个

-v：在归档过程中显示处理的文件。

-z：用gzip来压缩/解压缩文件，后缀名为.gz，加上该选项后可以将文件进行压缩。只能作用于文件

-j：用bzip2来压缩/解压缩文件，后缀名为.bz2，加上该选项后可以将文件进行压缩。只能作用于文件

例如，把“/home/user/0718”下的所有后缀为.c 的打包到source.tar，则使用 tar -cvf source.tar /home/user/0718/\*.c

若果在归档的过程中还要进行压缩，则使用tar –cvzf source.tar.gz /home/user/0718/\*.c。

如果要将归档的文件 source.tar 释放掉，则使用tar -xvf source.tar

若果将归档后的压缩文件释放掉，则使用tar -xvzf source.tar.gz。

## 21 文件压缩/解压

**命令：gzip [选项] 目标文档 源文件或目录**

功能：gzip 用来将文件压缩成后缀为.gz 的压缩文件，或者将后缀为.gz 的文件进行解压。只能作用于文件不能作用于文件夹，与tar结合才能压缩文件夹。

常用参数：

-d: 将压缩文件进行解压。Gzip –v man.c.gz main.c

-v： 在压缩或解压过程中显示解压或压缩的文件。Gzip –dv main.c main.c.gz

**命令：bzip2 [选项] 目标文档 源文件或目录**

功能： bzip2 用来将文件压缩成后缀名为.bz2 的压缩文件，或者将后缀为.bz2 的压缩文件解压。但会删除源文件。只能作用于文件不能作用于文件夹. 与tar结合才能压缩文件夹。

常用参数：

-z：将压缩文件进行压缩。bzip2 –zv main.c

-d: 将压缩文件进行解压。bzip2 –dv main.c.bz2

-v： 在压缩或解压过程中显示解压或压缩的文件。

**命令：zip/unzip [选项] 目标文档 源文件或目录**

功能：这是一对命令，zip进行压缩，unzip进行解压,压缩完成扩展名为.zip. 可以是文件也可以是文件夹.

常用参数

-r ：递归处理子文件或子目录. zip -vrtest.zip test/

-v ：显示指令执行过程或显示版本信息. unzip test.zip

各种压缩文件对应解压办法

|  |  |  |  |
| --- | --- | --- | --- |
| 文件格式 | 解压命令 | 文件格式 | 解压命令 |
| \*.tar | tar -xvf | \*.tar.bz2 | tar -xvjf |
| \*.gz | gzip -d | \*.Z | uncompress |
| \*.tar.gz和\*.tgz | tar -xvzf | \*.tar.Z | tar -xvZf |
| \*.bz2 | bzip2 -d | \*.rar | unrar e |
| \*.zip | unzip |  |  |

## 22 scp 远程 copy 文件命令

**命令：scp [可选参数] 源文件 目标文件**

**scp local\_filename username@ip:remote\_ filename**

将本地文件复制到远程主机目录或者**重命名**

**scp username@ip: remote\_ filename local\_filename**

将远程主机上的文件复制到本地

可选参数

–r：如果复制的是文件夹

local\_filename:本地文件名称,包含路径

username:远程主机的用户名

ip:远程主机 IP，也可以是域名

remote\_ filename：远程文件名，包含路径

例子：

scp file3 king@192.168.4.52:~/ 从本机 copy 到其他机器

scp king@192.168.4.52:~/file3 . 从其他机器 copy 到本机

## 23 ubuntu 设置固定 IP

首先点击右上角的《上下箭头》，点 Edit Connections,点 edit，点击ipv4 setting

## 24 其他命令

export PATH=$PATH //设置环境变量为全局变量

which 命令名 //找到命令所在路径

echo 文字或变量 //在显示器上显示一段文字，一般起到一个提示的作用。

## 25 chown 命令

**命令格式： chown [选项] [所有者] [:组] 文件**

功能：更改文件的所有者或组。通过chown改变文件的拥有者和群组。在更改文件的所有者或所属群组时，可以使用用户名称和用户识别码设置。普通用户不能将自己的文件改变成其他的拥有者。其操作权限一般为管理员。

必要参数:

-c 显示更改的部分的信息

-f 忽略错误信息

-h 修复符号链接，只对符号链接的文件做修改，而不改变其他相关文件

-R 递归式地改变指定目录及其下的所有子目录和文件的拥有者。

-v 显示详细的处理信息

-deference 作用于符号链接的指向，而不是链接文件本身，与-h相反

选择参数:

--reference=<目录或文件> 把指定的目录/文件作为参考，把操作的文件/目录设置成参考文件/目录相同拥有者和群组

--from=<当前用户：当前群组> 只有当前用户和群组跟指定的用户和群组相同时才进行改变

--help 显示帮助信息

--version 显示版本信息使用示例：

例子：

chown jim program.c

更改文件的所有者：文件 program.c 的所有者更改为 jim。作为所有者，jim 可以使用 chmod 命令允许或拒绝其他用户访问 program.c。

chown -R john:build /tmp/src

更改目录的所有者：将目录 /tmp/src 中所有文件的所有者和组更改为用户 john 和组 build

chown –R root:root rootfs

将rootfs文件夹及其子录的权限和组均改为root

chown –R liufan:liufan-desktop rootfs

将目录rootfs文件夹及子目录的所有者和组更改为用户liufan和组liufan-desktop ）

## 26 查看系统所有进程

ps –elf

存放于~/proc文件夹中

## 27 文件名

linux文件名中不能使用/符号。

文件名=dirname/basename

basename 文件名，获得文件名的基名

dirname 文件名，获得文件名的目录名

# Vim编辑器

### 1 vim状态

vim 分为两种状态，即命令状态和编辑状态，在命令状态下，所键入的字符系统均作命令来处理，如:q! 代表不保存退出，:wq保存并退出，而编辑状态则是用来编辑文本资料的。当你进入 vim 时，会首先进入命令状态。在命令状态下，按”i”(插入)或”a”(添加)可以进入编辑状态，在编辑状态，按 ESC 键进入命令状态。

### 2 添加

a 从光标后面开始添加文本

A 从光标所在行的末尾开始添加文本

### 3 插入

i 从光标前面开始插入文本

I 从光标所在行的开始处插入文本

### 4 删除

x 删除光标处的字符

dd 剪切光标所在的整行

[n]dd 删除光标所在行以及下面的n-1行

D 删除光标到行尾的文本，常用语删除注释语句 (d$)

### 5 光标移动：

^ 或者home光标移动到行首

$或者end 光标移动到行尾

Ctrl+u 向上翻半页

Ctrl+d 向下翻半页

Ctrl+b 向上翻一页

Ctrl+f 向下翻一页

Ctrl+e 编辑窗口中的文件内容整体上移一行。

Ctrl+y 编辑窗口中的文件内容整体下移一行

z + 回车 // 把当前行移动到屏幕顶部

100z + 回车 // 将移动第100行到屏幕的顶部，记得键入100z之后要再回车

z. // 把当前行移动到屏幕中央，不用回车，切记

z- // 把当前行移动到屏幕底部，同上，不用回车

gg 光标定位到文档头

G 光标定位

nG // 移动到第n行，注意此处G是大写。到文档尾

H 光标定位到当前页的第一行的行首

L 光标定位到当前页的最后一行的行首

M光标定位到当前页的中间一行的行首

w /W w：将光标右移一个字。光标停留在下一个字的字首位置；W：将光标右移一个字。光标停留在下一个字的字首位置（即使两个字之间存在标点符号）。

b /B b:将光标左移一个字。光标停留在下一个字的字首位置；B：将光标左移一个字。光标停留在下一个字的字首位置（即使两个字之间存在标点符号）。

e,E e：把光标移至当前所在字（或下一个字）的最后一个字符位置；E: 同e，只是以空格字符作为字的分隔符。fx //往右移动到 x 字符上

Fx 往左移动到 x 字符上

tx /往右移动到 x 字符前

Tx 往左移动到 x 字符后 （注意：以上四个命令中，其中x是键入的字符）

; //分号，配合 f 和 t 使用，重复一次

, //逗号，配合 f 和 t 使用，反方向重复一次

+ 下一行行首：//shift键切换到上档 '+'

- 上一行行首：

0 当前行行首

[n]+ 光标向后移动 n 行,[n]表示一个整数 10+

[n]- 光标向前移动 n 行,[n]表示一个整数 10

[n]G或:[n] 光标定位到第 n 行行首,[n]表示一个整数.

### 6 查找：

/pattern // 向前搜索模式，pattern代指要搜索的内容

?pattern // 向后搜索模式

n // 下一个搜索结果

N // 上一个搜索结果

### 7 替换：

替换光标所在字符

r 字符

替换当前行(只能替换光标之所在的行)

:s/[src]/[dst] /igc, i忽略大小写,g全部匹配（不加，只替换第一个）,c依次询问是够替换 例 :s/hello/world/ig

替换第n行

:[n] s/[src]/[dst]/ig，（n行中找） 例 :3,6s/hello/world

替换n1-n2行

:[n1],[n2]s/[src]/[dst]/ig，（n1-n2行中找） 例 :3,6s/hello/world

全部替换

:%s/[src]/[dst]/g ，将文档中所有 src 的字符串替换为 dst 字符串，:%s/^ //g 将文档每一行的行首的空格去掉

### 8 块操作：

v 可视化块选择状态，选中块之后，可以对块进行删除(d), 复制（y）剪切（x）。

ctrl+v，d选中删除，ctrl+v 用光标选中多行的开头，ctrl+i 选中后添加内容

### 9 剪切与复制

x剪切光标所在字符

yy 复制光标所在的整行

[n]yy 从光标开始往下复制 n 行,[n]表示一个整数

p 将复制后的文本粘贴到光标之前，

shift+p 将复制后的文本粘贴到光标之后

粘贴时最好进粘贴模式，避免自动对齐set paste，粘贴完后 set nopaste。

### 10 撤销与恢复

u 撤销上次操作

ctrl+r恢复u撤销的内容

### 11 保持与退出：

:q 在未修改文档的情况下退出

:q! 放弃文档的修改，强行退出

:w 文档存盘

:wq 文档存盘退出

:x 存盘退出

### 12 其他：

:help 命令 查看该命令的帮助提示

:%!xxd十六进制模式

:%!xxd –r 返回文本模式 中间有一个空格的

q：历史命令

win+l 锁定windows电脑

如果在编辑过程中不小心按了 Ctrl+s,vim 会处于僵死状态，按Ctrl+q 可以恢复。

执行 vim +3 main.c //表示定位到 main.c 的第 3 行

执行 vim +/printf main.c //表示定位到第一个 printf 处

在命令模式下输入:new 2.c //表示再打开一个 vim,是横向的 用:vnew 2.c 表示纵向

也可以通过:split vsplit sp vsp，把本文件自己分成两个窗口。

两个窗口之间进行切换的方式：Ctrl+w,w

在命令模式中输入 gg=G 可以自动对齐

$ sed: 管道查找替换程序（替换批量文件中的字符）单引号对引号内的内容默认使用一次转义，双引号则不用转义，要用\进行转型,linux命令

$ cat a.txt| sed's/aa/bb/' >b.txt //将 a.txt 中的 aa 替换成 bb 并重定向输出到 b.txt 中。linux命令

## 13 同时移动多行

缩进单行代码是两个大于号'>>'，回缩就是两个小于号'<<'。

如果想要缩进很多行代码的话就按照下面做：

第一种方式：visual模式下选中内容后，按一次大于号'>'缩进一次，按'6>'缩进六次，按'<'回缩。

第二种方式：command 模式下， m,n>回车,从m,n行左移一次，m,n>>回车，从m,n行左移两次.

>n,n行右移一次，>>n，n行右移两次

第三种方式：normal模式下，光标在需要处理的起始行，然后：行数>>，左移，行数<<右移

第四种方式： nomal模式下，移动到需要处理的首行ma移动到需要处理的末行<'a

# GitHub

1. $ sudo apt install git 安装git

2. git init 或git init dirname 将当前文件夹或指定文件夹初始为仓库

5. git add filename 将文件提交到暂存区

6. git commit –m “提交说明” 将暂存区文件提交到版本库,

7. git status 查看当前仓库状态，工作区文件是否add，暂存区文件是否commit

git status -s 查看当前仓库简述状态，工作区文件是否add，暂存区文件是否commit

8. git log 查看commit历史，可以看到各版本的id，但不包括已经删除的commit

9. git reflog 查看所有分支的所有操作记录（包括commit和reset的操作），包括已删除的commit记录

10. git diff [filename] 查看工作区的文件与上一个版本的不同之处

git diff HEAD [-- filename] 可以查看工作区和版本库里面最新版本的区别：

11. git checkout -- filename 撤销工作区文件的修改到最近一次git add后的状态（没有提交到暂存区）

12. git reset HEAD filename 撤销已add提交到暂存区的文件到工作区，再用git checkout就能丢弃修改

13. git reset --hard 版本回退，撤销已经commit到版本库，但还没有推送到远程仓库的修改

git reset --hard HEAD^ 版本回退一次

git reset --hard HEAD^^版本回退两次

git reset --hard HEAD~n 版本回退n次

git reset --hard 版本号 回退到指定版本号

git checkout id 文件名 将某个文件恢复到某个版本时的状态

14. rm filename 只删除工作区文件

git rm filename 删除暂存区文件

git commit –m “注释”

就可以从版本库中删除文件，如果不想删除 git checkout -- filename撤销删除

git rm -f filename 删除工作区和暂存区文件

git rm --cached 文件名 只删除暂存区文件

15. git checkout -b 分支名 创建并切换，相当于以下两条命令：

git branch 分支名 创建分支

git checkout 分支名 切换到分支

git branch 列出所有分支，当前分支前面会标一个\*号

git show-branch 查看分支信息

git merge分支名 用于合并指定分支到当前分支。在之前应切换到主分支。默认使用fast forward

git merge --no-ff -m "merge with no-ff" 分支名，禁用fast forward，避免删除分支时丢掉分支信息。

git branch -d 分支名 删除已经合并的某分支

git branch -D 分支名 强制删除未合并的分支

16. git stash 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 list 查看快照

git stash pop，恢复现场并删除快照，相当于以下两个命令

git stash apply 恢复现场但并不删除快照

git stash drop 删除快照；

17. git remote 查看远程库信息

git remote -v 查看远程库的详细信息

git remote show主机名，可以查看该主机的详细信息。

git remote add 主机名 地址 添加远程主机。 默认主机为origin

git remote rm 主机名 用于删除远程主机。

git remote rename 原主机名 <新主机名> 用于远程主机改名

18. git fetch 远程主机名 [分支名] 将远程主机的[某分支]更新，全部取回本地。对本地代码没影响。

git branch -r 查看远程分支， 远程分支用 远程主机/分支名 来访问

git branch -a 查看所有分支

git checkout -b newBrach 远程主机/分支名

git merge 远程主机/分支名 将远程分支合并到当前分支

19. git clone <版本库的网址> <本地目录名> 克隆远程仓库到目标目录中

支持多种协议：

git clone http[s]://example.com/path/to/repo.git/

git clone ssh://example.com/path/to/repo.git/

git clone git://example.com/path/to/repo.git/

git clone /opt/git/project.git

git clone file:///opt/git/project.git

git clone ftp[s]://example.com/path/to/repo.git/

git clone rsync://example.com/path/to/repo.git

20. git push <远程主机名> <本地分支名>:[远程分支名] 将本地分支的更新，推送到远程主机,

如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），

如果该远程分支不存在，则会被新建。

如果省略本地分支名，则表示删除指定远程分支，因为这等同于推送一个空的本地分支到远程分支。

如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

如果当前分支只有一个追踪分支，那么主机名都可以省略。

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机后面就可不加任何参数使用git push.

git push --all origin 将所有本地分支都推送到origin主机。

git push --force origin 强制推送，覆盖远程分支。

21. git pull [远程主机名] [远程分支名]:[本地分支名] 取回远程主机某分支，与本地的指定分支合并。

如果远程分支是与当前分支合并，则冒号后面的部分可以省略。等效于：

git fetch 远程主机 分支名

git merge 远程主机/分支名

如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。

如果当前分支只有一个追踪分支，连远程主机名都可以省略。

git branch --set-upstream 本地分支 远程主机/分支名 指定本地分支追踪到远程分支

git pull --rebase <远程主机名> <远程分支名>:<本地分支名> 合并需要采用rebase模式。

git pull -p 在本地删除远程已经删除的分支。等同于下面的命令

git fetch --prune origin

git fetch -p 在本地删除远程已经删除的分支。

22. HEAD指针指向当前分支，master指针指向主分支。

23. git tag tagname 为当前分支创建标签

git tag 或git tag -l查看所有标签

git tag tagname commitid 为指定commitid打上标签

git show tagname 查看标签信息

git tag tagname commitid -a -m "标签说明" 为指定commitid打上带说明信息的标签

git tag tagname commitid -s -m "标签说明" 用私钥签名一个标签

git tag -d tagname 删除本地标签

git push origin :refs/tags/tagname 删除远程标签，如果标签已推送本地和远程都需删除

git push origin tagname 将指定标签推送到远程

git push origin --tags 将本地所有标签推送到远程

24. 创建SSH Key。

在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key命令：

ssh-keygen -t rsa -C "youremail@example.com" 生成密钥用于创建远程仓库

登录github 账户——>设置 添加SSH key.pub的内容，新建仓库。

25. 开启两步验证后推送失败的解决方法

首先在github设置中启用Personal Access Token

再输入密码时输入访问令牌而不是账户密码

如果不想每次都输密码则可以使用 git config --global credential.helper store

26. git 配置，加上--global是针对所有用户起作用的，如果不加，那只针对当前的仓库起作用。每个仓库的Git配置文件是.git/config。

$git config --global color.ui true 让Git显示颜色，会让命令输出看起来更醒目。

在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

git config --global alias.别名 git命令名 为git命令配置别名中。

$ git config --global user.name "Your Name" 配置全局用户

$ git config --global user.email "email@example.com"

27. 搭建git服务器

sudo apt install git 安装git

sudo adduser git 创建git用户

创建证书登录：收集所有需要登录的用户的公钥，就是他们自己的id\_rsa.pub文件，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

初始化Git仓库：先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入$ sudo git init --bare sample.git，Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把owner改为git：$ sudo chown -R git:git sample.git

禁用shell登录：出于安全考虑，第二步创建的git用户不允许登录shell，这可以通过编辑/etc/passwd文件完成。找到类似下面的一行：

git:x:1001:1001:,,,:/home/git:/bin/bash

改为：

git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

克隆远程仓库：现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行。

28. 为项目添加成员，仓库-->设置-->member-->权限

为分支设置权限， 仓库-->设置-->branches --> Protected branches -->权限

# Shell 编程

如果我们有一系列经常使用的 Linux 命令，我们可以把它们存储在一 个文件中。Shell 可以读取这个文件并执行其中的命令。这样的文件被称 为脚本文件。 执行 shell 脚本 要创建一个 shell 脚本，我们要使用任何编辑器比如 vi 在文本文件中编写 它，保存的文件最好是.sh 后缀的。 如：vi aa.sh chmod +x aa.sh 然后 ./aa.sh 或 bashaa.sh 或 shaa.sh 。

shell 脚本的编写语法如下：

## 1 程序以下面的行开始

#!/bin/bash（由于是在 redhat 下，所以系统默认 的 shell 是bashshell。）

## 2 注释 #

## 3 shell变量

### 3.1 变量

shell的变量没有数据类型，都是字符串，即使数值也是字符串。

**（1）创建变量：**变量名称=值。如果值有空格则必须用双引号或者单引号引用起来。 Eg: a=“hello” （=号两边不能有空格）

**（2）引用变量：一般情况引用变量**$变量名 或 ${变量名} ，字符串中引用变量“${变量名}”，注意’ ’与” ”的区别，单引号消除所有字符的特殊意义；双引号消除除$ 、” ”、’ ’三种以外其它字符的特殊意义）

**（3）输出变量：**echo 输出对象 **或**printf ”c语言格式控制”变量1变量2 ……格式化输出

**（4）删除变量**：unset 变量名

**（5）设为只读变量** readonly变量名

**（6）从键盘输入**，实现程序交互：read 变量名，Eg：read a

**（7）使用转义符：**使用$、”、’、空格、\都必须用转移符\，Eg:a=What\'s\ your\ \"topic\"\?

**（8）命令替换：** `命令`或$(命令) ， Eg:echo `pwd` 等效于echo $(pwd)

**（9）表达式计算:** shell不能直接进行算术计算只能借助 `expr 表达式` 或者 $(expr表达式) 或者$((表达式)) 或者$[表达式]

举例：

a=4;b=5

`expr 4+5` == $(expr 4+5) ==$(($a+$b))==$[$a+$b]

举例：

1>: #echo $a 结果为hello，等同于#echo ${a} 和 #echo“${a}”

2>:#echo “hello b$aa”结果为hellob,因为此时把 aa 作为一个整体变量，而且没有定义，所以只输出前面的字符串

3>: #echo“hello b${a}a”结果为hellobhelloa

4>: #echo ”${a}a”结果为helloa

5>: #echo‘${a}a’ 结果为${a}a,因为’’会消除所有特殊字符的意义。

6>: #echo‘\${a}a’ 结果为\${a}a.

7>unset a

8>readonly a

举例：

#!/bin/bash //开始行

#this is my firstshellproject //注释

read dir //读入 1 个目录名

mkdir ${dir} //在当前目录下创建该目录

cp -rf /etc/\*.conf ${dir} //复制 etc 下的 conf 文件到该目录

ls –l /etc/\* | grep^d | wc -l> etcdir.txt //统计etc下所有目录的数目到 etcdir.txt 中

Linux shell中，定义的函数变量，默认是全局global的，即使是在函数中定义的变量，也是全局的。所以，递归函数中的直接定义的变量，由于是全局变量变量，导致：下一层调用中修改了某个变量后，返回上一层时，变量的值不会恢复。所以导致运行结果不对。

解决办法就是，把（递归函数中的）所有的变量之前加上local定义，表示局部变量，这样运行的结果，和函数执行逻辑，就和C等其他语言类似，结果也就都正确了。

### 3.2环境变量

系统预定义的变量，一般在/etc/profile或者/etc/bashrc或者~/.bashrc或~/.profile中进行定义 :

.bash\_profile：该文件定义了用户的个人化路径与环境变量的文件名称。每个用户都可使用该文件输入专属于自己的shell信息，当前用户登入时，该文件仅仅执行一次。

.bashrc：该文件包含专属于自己的shell的bash信息，当登入或每次打开新的shell时，该文件被读取。例如你可以将用户自定义的alais或者自定义变量写到这个文件中。

.bash\_history：该文件用于记录命令历史。

.bash\_logout：当退出shell时，会执行该文件。你可以将一些清理的工作放到这个文件中。

BASH\_SUBSHELL 记录当前子shell的层次。BASH\_SUBSHELL是从0开始计数的整数。

BASH\_VERSINFO 是一个数组包含六个元素，这六个元素显示bash的版本信息。

BASH\_VERSION 显示shell版本的信息

DIRSTACK 记录了栈顶的目录值，初值为空

EUID 有效用户id

GLOBLGNORE 是由冒号分割的模式列表，表示通配时忽略的文件名集合。

GROUPS 记录当前用户所属的组

HISTSIZE　 历史记录数

HOME 用户主目录

HOSTNAME 主机名

HOSTTYPE 记录系统的硬件架构。

IFS 用于设置指定shell域分隔符，默认情况下为空格。

LANG 　　 语言相关的环境变量，多语言可以修改此环境变量

LC\_ADDRESS 地址书写方式

LC\_COLLATE 比较和排序习惯

LC\_CTYPE 用户所使用的语言符号及其分类

LC\_IDENTIFICATION locale对自身包含信息的概述

LC\_MEASUREMENT 度量衡表达方式()

LC\_MESSAGES 信息主要是提示信息,错误信息, 状态信息, 标题, 标签, 按钮和菜单等

LC\_MONETARY 货币单位

LC\_NAME 姓名书写方式

LC\_NUMERIC 数字

LC\_PAPER 默认纸张尺寸大小 ()

LC\_TELEPHONE 电话号码书写方式

LC\_TIME 时间显示格式()，

LOGNAME 登录名

LESSCLOSE

LS\_COLORS 颜色配置

MACHTYPE 记录硬件架构和操作系统类型

MAIL　　　　　　 当前用户的邮件存放目录

OLDPWD 用户上一次工作目录

OSTYPE 记录操作系统类型。

PATH 文件搜索路径，决定了shell将到哪些目录中寻找命令或程序

PPID 父进程ID

PS1　　　　　　 基本提示符，对于root用户是#，对于普通用户是$

PS2　　　　　　 附属提示符，默认是“>”

PWD 用户当前工作目录

QT\_QPA\_PLATFORMTHEME qt平台主题

SECONDS 记录脚本从开始到结束耗费的时间。

SHELL 　　　　 前用户Shell类型

SHELLOPTS 记录了处于“开”状态的shell选项列表，它只是一个只读变量。

SHLVL 记录了bash嵌套的层次，第一个Shell。 $SHLVL=1。如果在这个Shell中执

行脚本，脚本中的$SHLVL=2 。

SSH\_TTY

SSH\_CLIENT ssh协议客户端ip和端口

SSH\_CONNECTION ssh协议客户端和服务器端的ip与端口

TERM 颜色模式，16级或256级

UID 用户ID

USER 用户名

XDG\_SESSION\_ID 终端会话id

XDG\_RUNTIME\_DIR系统用户独立的运行目录

用 **env** 看环境所有变量；

用 **env| grep** "变量名" 查找环境变量 ；

用**export变量名=值** 进行设定或更改为环境变量，可以不赋值；

用 **unset 变量名** 取消变量的定义 。

举例:

name="RedHatLinux" //定义本地变量

export name //把name变为全局变量

sh //进入子shell

echo ${name} //全局变量可以作用于子进程，而本地变量不可以。

bash //退出子 shell,进入父 shell

export name="RedHat Linux" //直接输出name

**设置环境变量**：

举例：

把/etc/apache/bin 目录添加到 PATH 中：

#PATH=$PATH:/etc/apache/bin

vim /etc/profile 在里面添加 PATH=$PATH:/etc/apache/bin

vim ~/.bash\_profile 在里面修改 PATH 行，把/etc/apache/bin 加进去，此种方法针对当前用户有效。

注意：环境变量不是全局量，而是系统提供的变量。

### 3.3 特殊变量或者函数的传入参数

$1,$2…$n 传入的参数，相当于argv[i],注意i不能用变量。不算程序本身。当n>=10，要使用${n}

$0 表示shell程序名称，每一项相当于 main 函数中 argv[0]

$# 传递到脚本的参数列表,或表示参数个数，不包含$0

$\* 把脚本全部参数为一个字符串，以”$1 $2 … $n”的形式输出所有参数。

$@ 传入脚本的全部参数，argv[1] 到argv[n-1] ,每个参数作为一个字符串。

$? 前个命令执行情况或者函数返回值。

$$ 脚本运行的当前进程号

$! 后台运行的最后一个进程的ID号

$- 显示Shell使用的当前选项，与set命令功能相同。

举例：

vim 1.sh //建立一个名为1的sh文件

#!/bin/bash //起始行

echo $1 //传入的参数1

echo $2 //传入的参数2

echo $3 //传入的参数3

echo $# //传入的参数个数或列表

echo $@ //传入的全部参数

echo $\* //脚本中的全部参数

echo $$ //脚本运行的当前进程号

exit 3 //./1.sh 1 2 hello "hello world"

echo $? //上一个命令是否成功e（0 表示成功1 表示失败），或者函数返回值。

### 3.4 Shell字符串

字符串是shell编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。

**（1）单引号**

str='this is a string'

单引号字符串的限制：

单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；

单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

**（2）双引号**

str="Hello, I know your are \"$your\_name\"! \n"

双引号的优点：

双引号里可以有变量

双引号里可以出现转义字符

**（3）拼接字符串**

your\_name="qinjx"

greeting="hello, "$your\_name" !"

greeting\_1="hello, ${your\_name} !"

echo $greeting $greeting\_1

**（4）获取字符串长度或者数组长度用“#”**

string="abcd"

echo ${#string} #输出 4

**（5）提取子字符串**

以下实例从字符串第2个字符开始截取 4 个字符：

string="runoob is a great site"

echo ${string:1:4} #输出 unoo

**（6）查找子字符串**

查找字符 "i 或 s" 的位置：

string="runoob is a great company"

echo `expr index "$string" is` # 输出 8

## 4 Shell echo命令

Shell 的 echo 指令与 PHP 的 echo 指令类似，都是用于字符串的输出。

**命令格式：echo string**

可以使用echo实现更复杂的输出格式控制。

**（1）显示普通字符串:**

echo "It is a test"

这里的双引号完全可以省略，以下命令与上面实例效果一致：

echo It is a test

**（2）显示转义字符**

echo "\"It is a test\""

结果将是:

"It is a test"

同样，双引号也可以省略

**（3）输出变量**

read 命令从标准输入中读取一行,并把输入行的每个字段的值指定给 shell 变量

#!/bin/sh

read name

echo "$name It is a test"

以上代码保存为 test.sh，name 接收标准输入的变量，结果将是:

# sh test.sh

OK #标准输入

OK It is a test #输出

**（4）显示换行**

echo -e "OK! \n" # -e 开启转义

echo "It it a test"

输出结果：

OK!

It it a test

**（5）显示不换行**

#!/bin/sh

echo -e "OK! \c" # -e 开启转义 \c 不换行

echo "It is a test"

输出结果：

OK! It is a test

**（6）显示结果定向至文件**

echo "It is a test" > myfile

**（7）原样输出字符串，不进行转义或取变量(用单引号)**

echo '$name\"'

输出结果：

$name\"

**（8）显示命令执行结果**

echo `date`

结果将显示当前日期

Thu Jul 24 10:08:46 CST 2014

## 5 Shell printf 命令

printf 命令模仿 C 程序库（library）里的 printf() 程序。标准所定义，因此使用printf的脚本比使用echo移植性好。printf 使用引用文本或空格分隔的参数，外面可以在printf中使用格式化字符串，还可以制定字符串的宽度、左右对齐方式等。默认printf不会像 echo 自动添加换行符，我们可以手动添加 \n。

printf 命令的语法：

**printf format-string [arguments...]**

参数说明**：**

format-string: 为格式控制字符串，可以带双引号或单引号或不带引号。格式控制符和C语言一样。%s %c %d %f都是格式替代符。

%-10s 指一个宽度为10个字符（-表示左对齐，没有则表示右对齐），任何字符都会被显示在10个字符宽的字符内，如果不足则自动以空格填充，超过也会将内容全部显示出来。

%-4.2f 指格式化为小数，其中.2指保留2位小数。

格式控制只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用如果没有 arguments，那么 %s 用NULL代替，%d 用 0 代替

arguments: 为参数列表。

实例如下：

$ echo "Hello, Shell"

Hello, Shell

$ printf "Hello, Shell\n"

Hello, Shell

接下来,我来用一个脚本来体现printf的强大功能：

#!/bin/bash

printf "%-10s %-8s %-4s\n" 姓名 性别 体重kg

printf "%-10s %-8s %-4.2f\n" 郭靖 男 66.1234

printf "%-10s %-8s %-4.2f\n" 杨过 男 48.6543

printf "%-10s %-8s %-4.2f\n" 郭芙 女 47.9876

执行脚本，输出结果如下所示：

姓名 性别 体重kg

郭靖 男 66.12

杨过 男 48.65

|  |  |
| --- | --- |
| 序列 | 说明 |
| \a | 警告字符，通常为ASCII的BEL字符 |
| \b | 后退 |
| \c | 抑制（不显示）输出结果中任何结尾的换行字符（只在%b格式指示符控制下的参数字符串中有效），而且，任何留在参数里的字符、任何接下来的参数以及任何留在格式字符串中的字符，都被忽略 |
| \f | 换页（formfeed） |
| \n | 换行 |
| \r | 回车（Carriage return） |
| \t | 水平制表符 |
| \v | 垂直制表符 |
| \\ | 一个字面上的反斜杠字符 |
| \ddd | 表示1到3位数八进制值的字符。仅在格式字符串中有效 |
| \0ddd | 表示1到3位的八进制值字符 |

## 4 运算符与表达式

### 4.1 算术运算符

下表列出了常用的算术运算符，假定变量 a 为 10，变量 b 为 20：

|  |  |  |
| --- | --- | --- |
| 运算符 | 说明 | 举例 |
| + | 加法 | `expr $a + $b` 结果为 30。 |
| - | 减法 | `expr $a - $b` 结果为 -10。 |
| \* | 乘法 | `expr $a \\* $b` 结果为  200。 |
| / | 除法 | `expr $b / $a` 结果为 2。 |
| % | 取余 | `expr $b % $a` 结果为 0。 |
| = | 赋值 | a=$b 将把变量 b 的值赋给 a。 |
| == | 相等。用于比较两个数字，相同则返回 true。 | [ $a == $b ] 返回 false。 |
| != | 不相等。用于比较两个数字，不相同则返回 true。 | [ $a != $b ] 返回 true。 |

**注意：**

（1）条件表达式要放在方括号之间，并且要有空格，例如: **[$a==$b]** 是错误的，必须写成 **[ $a == $b ]**。

（2）乘号\*前边必须加反斜杠(\)才能实现乘法运算；

（3）$((表达式))，此处表达式中的 "\*" 不需要转义符号 "\"。

### 4.2 关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

下表列出了常用的关系运算符，假定变量 a 为 10，变量 b 为 20：

|  |  |  |
| --- | --- | --- |
| 运算符 | 说明 | 举例 |
| -eq | 检测两个数是否相等，相等返回 true。 | [ $a -eq $b ] 返回 false。 |
| -ne | 检测两个数是否相等，不相等返回 true。 | [ $a -ne $b ] 返回 true。 |
| -gt | 检测左边的数是否大于右边的，如果是，则返回 true。 | [ $a -gt $b ] 返回 false。 |
| -lt | 检测左边的数是否小于右边的，如果是，则返回 true。 | [ $a -lt $b ] 返回 true。 |
| -ge | 检测左边的数是否大于等于右边的，如果是，则返回 true。 | [ $a -ge $b ] 返回 false。 |
| -le | 检测左边的数是否小于等于右边的，如果是，则返回 true。 | [ $a -le $b ] 返回 true。 |

### 4.3 逻辑运算符

下表列出了常用的布尔运算符，假定变量 a 为 10，变量 b 为 20：

|  |  |  |
| --- | --- | --- |
| 运算符 | 说明 | 举例 |
| ! | 非运算，表达式为 true 则返回 false，否则返回 true。 | [ ! false ] 返回 true。 |
| -o或|| | 或运算，有一个表达式为 true 则返回 true。 | [ $a -lt 20 -o $b -gt 100 ] 返回 true。 |
| -a或&& | 与运算，两个表达式都为 true 才返回 true。 | [ $a -lt 20 -a $b -gt 100 ] 返回 false。 |

### 4.4 字符串运算符

下表列出了常用的字符串运算符，假定变量 a 为 "abc"，变量 b 为 "efg"：

|  |  |  |
| --- | --- | --- |
| 运算符 | 说明 | 举例 |
| = 或== | 检测两个字符串是否相等，相等返回 true。 | [ $a = $b ] 返回 false。 |
| != | 检测两个字符串是否相等，不相等返回 true。 | [ $a != $b ] 返回 true。 |
| -z STRING | 检测字符串长度是否为0，为0返回 true。 | [ -z $a ] 返回 false。 |
| -n STRING | 检测字符串长度是否为0，不为0返回 true。 | [ -n $a ] 返回 true。 |
| STRING | 检测字符串是否为空，不为空返回 true。 | [ $a ] 返回 true。 |

### 4.5 文件测试运算符

文件测试运算符用于检测文件的各种属性。

|  |  |  |
| --- | --- | --- |
| 操作符 | 说明 | 举例 |
| -a file | 检测文件是否存在，如果是，则返回 true。 |  |
| -b file | 检测文件是否是块设备文件，如果是，则返回 true。 | [ -b $file ] 返回 false。 |
| -c file | 检测文件是否是字符设备文件，如果是，则返回 true。 | [ -c $file ] 返回 false。 |
| -d file | 检测文件是否是目录，如果是，则返回 true。 | [ -d $file ] 返回 false。 |
| -e file | 检测文件（包括目录）是否存在，如果是，则返回 true。 | [ -e $file ] 返回 true。 |
| -f file | 检测文件是否是普通文件，如果是，则返回 true。 | [ -f $file ] 返回 true。 |
| -h file | 检测文件是否是符号链接，如果是，则返回true |  |
| -g file | 检测文件是否设置了 SGID 位，如果是，则返回 true。 | [ -g $file ] 返回 false。 |
| -k file | 检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。 | [ -k $file ] 返回 false。 |
| -p file | 检测文件是否是有名管道，如果是，则返回 true。 | [ -p $file ] 返回 false。 |
| -r file | 检测文件是否可读，如果是，则返回 true。 | [ -r $file ] 返回 true。 |
| -s file | 检测文件是否为空（文件大小是否大于0），不为空返回 true。 | [ -s $file ] 返回 true。 |
| -t fd | 如果文件描述符 FD 打开且指向一个终端则为真。 |  |
| -u file | 检测文件是否设置了 SUID 位，如果是，则返回 true。 | [ -u $file ] 返回 false。 |
| -w file | 检测文件是否可写，如果是，则返回 true。 | [ -w $file ] 返回 true。 |
| -x file | 检测文件是否可执行，如果是，则返回 true。 | [ -x $file ] 返回 true。 |
| -O FILE | 如果 FILE 存在且属有效用户ID则为真。 |  |
| -G FILE | 如果 FILE 存在且属有效用户组则为真。 |  |
| -L FILE | 如果 FILE 存在且是一个符号连接则为真。 |  |
| -N FILE | 如果 FILE存在and has been mod如果ied since it was last read则为真。 |  |
| -S FILE | 如果 FILE 存在且是一个套接字则为真。 |  |
| FILE1 -nt FILE2 | 如果 FILE1 比 FILE2 要新, 或者 FILE1存在且 FILE2 不存在则为真。 |  |
| FILE1 -ot FILE2 | 如果 FILE1 比 FILE2 要老, 或者 FILE2 存在且 FILE1 不存在则为真。 |  |
| FILE1 -ef FILE2 | 如果 FILE1 和 FILE2 指向相同的设备和节点号则为真。 |  |
| -o OPTION | 如果 shell选项 “OPTIONNAME” 开启则为真。 |  |

关系运算符（-eq、-ne、-gt、-lt、-ge、-le）

逻辑运算符(&&、||、!)

赋值运算符(=、+=、-=、\*=、/=、%=、&=、^=、|=、<<=、>>=)

计算表达式有四种：$(())、$[]、 let var= 、expr

echo $[$v1< $v2] 计算逻辑表达式(用 1 表示 true,用 0 表示 false)

echo $[($v1<$v2)&&($v1>$v2)] 计算逻辑表达式

v3=2

let v3\*=$(($v1+$v2))

echo $v3 或 echo ${v3}

举例：

写 2.sh 要求输入 2 个数 计算 2 个数的和

#!/bin/bash

#this is my secondshell project

echo "please input the first number:"

read a

echo "please input the second number:"

read b

c=$(($a + $b))

echo "The result of $a +$b is $c"

## 5 Test 命令

1）判断表达式

test 表达式 1 –a 表达式 2 与运算,两个表达式都为真

test 表达式 1 –o 表达式 2 或运算 两个表达式有一个为真

test ！表达式 1 表达式真则返回false，否则返回true

举例：

测试是否是闰年：

test $(($iYear%400)) –eq 0 –o $(($iYear% 4)) -eq 0 –a $(($iYear %100)) -ne 0

2）判断字符串

test -n 字符串 字符串的长度非零返回true

test -z 字符串 字符串长度为零返回true

test ＝＝ 字符串 字符串相等返回true

test !＝ 字符串 字符串不等返回true

test str 字符串 字符串不为空返回true

举例：

a=”abc”

test $a ==”abc”

echo $?(0)

test $a ==”afd”

echo $?(1)

3）判断整数

test 整数 1 -eq 整数2 检测两个数是否相等，相等返回 true

test 整数 1 -ge 整数 2 检测左边的数是否大等于右边的，如果是，则返回 true

test 整数 1 -gt 整数 2 检测左边的数是否大于右边的，如果是，则返回 true

test 整数 1 -le 整数 2 检测左边的数是否小于等于右边的，如果是，则返回 true

test 整数 1 -lt 整数 2 检测左边的数是否小于右边的，如果是，则返回 true

test 整数 1 -ne 整数 2 检测两个数是否相等，不相等返回 true

4）判断文件

test File1 –ef File2 两个文件具有同样的设备号和 i 结点号

test File1 –nt File2 文件 1 比文件 2 新

test File1 –ot File2 文件 1 比文件 2 旧

test –b File 检测文件是否是块设备文件，如果是，则返回 true。

test –c File 检测文件是否是字符设备文件，如果是，则返回 true

test –d File 检测文件是否是目录，如果是，则返回 true

test –e File 文件存在检测文件（包括目录）是否存在，如果是，则返回 true

test –f File 检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。

test –g File 检测文件是否设置了 SGID 位，如果是，则返回 true

test –k File 检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true

test –p File 检测文件是否是具名管道，如果是，则返回 true

test –r File 检测文件是否可读，如果是，则返回 true

test –s File 检测文件是否为空（文件大小是否大于0），不为空返回 true

test –u File 检测文件是否设置了 SUID 位，如果是，则返回 true。

test –w File 检测文件是否可写，如果是，则返回 true

test –x File 检测文件是否可执行，如果是，则返回 true

举例：

a=2

test $a–ge 3

echo $?

## 6 Shell 数组

数组中可以存放多个值。Bash Shell **只支持一维数组**（不支持多维数组），初始化时不需要定义数组大小与大部分编程语言类似，数组元素的下标由0开始。

**数组定义：**

定义 1：a=(1 2 3 4 5) 下标从 0 开始， 各个数据之间用空格隔开

定义 2：a[0]=1;a[1]=2;a[2]=3

定义 3：a=([1]=1 [2]=2)

**数组元素引用** ：${a[1]}

**求数组长度：**${#a[@]}或${#a[\*]}

**截取部分数组元素：**${a[@]:1:2}从下标 1 开始后面显示 2 个

${a[@]:2} 截取下标从 2 到最后

**数组的所有元素：**${a[@]}或${a[\*]}

## 7 shell 选择

### 7.1 If 语句

单分支：

if[condition]

then

action

fi

双分支：

if[condition]

then

action

else

fi

多分支：

if[condition]

then

action

elif [condition2 ]

then

action2

elif [condition3]

then

……

else

action

fi

注意：

只有当 condition 为真时，该语句才执行操作，否则不执行操作，并继续执行 “fi” 之后的任何行。在使用时，将 “if” 和 “then” 放在不同行，如同行放置，则 if 语句必须要以分号；结束

举例:

用参数传 1 个文件名，该文件如果是文件并且可读可写就显示该文件，如果是目录就进入该目录，并判断 ls.sh 存在否，如果不存在就建立1个ls.sh 的文件并运行该文件。 该文件的内容是

ls -li /etc >etc.list

#!/bin/bash

if [-f $1 -a –r $1 -a –w $1]

//判断是普通文件并可读可写，也可以写成if test–f $1 –a –r $1 –a –w $1

then

cat $1 //显示文件内容

elif [-d $1] //否则如果是目录

then

cd $1 //进入目录

if [-e ls.sh] //如果 ls.sh 该文件存在

then

chmod +x ls.sh //赋予可执行的权限

./ls.sh //执行

else

touch ls.sh //如果不存在则创建 ls.sh

echo "#!/bin/bash" >>ls.sh //将程序写入 ls.sh 中保存

echo "ls –li /etc >etc.list" >>ls.sh //将要执行的命令写入 ls.sh 中保存 chmod+xls.sh //赋予可执行的权限

./ls.sh //执行

fi

fi

### 7.2 Case 语句

case 常用的语法形式如下：

case 变量值 in

模式1)

action

;;

模式2)

action

;;

\*)

action

;;

esac

case工作方式如上所示。取值后面必须为单词in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 \* 捕获该值，再执行后面的命令。

case的语法和C family语言差别很大，它需要一个esac（就是case反过来）作为结束标记，每个case分支用右圆括号，用两个分号表示break。

例 1:

echo "Is it morning? Please answer yes or no."

read YES\_OR\_NO

case "$YES\_OR\_NO" in

yes|y|Yes|YES)

echo "Good Morning!"

;;

[nN]\*) /\* 表示 n 或 N 开头的任意字段 \*

echo "Good Afternoon!"

;;

\*)

echo "Sorry, $YES\_OR\_NO not recognized. Enter yes or no."

exit 1

;;

esac

例2：编写一个加减乘除取模计算器

echo "please input thefirstnumber:"

read a

echo "pleaseinput the secondnumber:"

read b

echo "pleaseinput your operator:"

read c

case $c in

"+")

echo "the result of$a+ $bis$(($a +$b))"

;;

"-")

echo "the result of$a-$bis $(($a -$b))"

;;

"\*")

echo "the result of$a\*$bis $(($a \*$b))"

;;

"/")

echo "the result of$a/$bis $(($a /$b))"

;;

\*)

echo "notrue operator!"

;;

esac

## 8 shell 循环

同c一样shell循环可用break跳出循环，用continue继续下一循环

### 8.1 for 循环

for 变量 in 取值列表 //或者 for（（循环变量初值;循环条件；循环增量））

do

循环体

done

当变量值在列表里，for循环即执行一次所有命令，使用变量名获取列表中的当前取值。命令可为任何有效的shell命令和语句。in列表可以包含命令替换、字符串、文件名和目录。in列表是可选的，如果不用它，for循环使用命令行的位置参数。注意如果是目录则应该使用for 循环变量 in dir/\* 或者 for 循环变量 in `ls dirname`

Linux shell中，定义的函数变量，默认是全局global的，即使是在函数中定义的变量，也是全局的。所以，递归函数中的直接定义的变量，由于是全局变量变量，导致：下一层调用中修改了某个变量后，返回上一层时，变量的值不会恢复。所以导致运行结果不对。

解决办法就是，把（递归函数中的）所有的变量之前加上local定义，表示局部变量，这样运行的结果，和函数执行逻辑，就和C等其他语言类似，结果也就都正确了。

例1：

for x in one two three four

do

echo number $x

done

例 2：

for x in /etc/????.???? /var/lo\*/home/\*${PATH} //列举

do

echo $x

done

例3：/etc/r\*中的文件和目录

for myfile in /etc/r\*

do

if[ -d "$myfile" ]

then

echo"$myfile(dir)"

else

echo"$myfile"

fi

done

例 4：

for x in /var/log/\*

do

echo `basename $x` is a file living in/var/log

done

例 5：//冒泡排序

#!/bin/bash

a=(3 10 6 5 9 2 8 1 4 7)

for ((i=1;i<10; i++ ))

do

for ((j=0;j<10-i; j++ ))

do

if [ ${a[j]} –gt ${a[j+1]} ]

then

temp=${a[j]}

a[j]=${a[j+1]} //或者 a[j]=${a[$(($j+1))]}

a[j+1]=$temp

fi

done

done

for ((i=0;i<10; i++ ))

do

echo ${a[i]}

done

### 8.2 While 语句

循环变量=初值

while[循环条件] //或者while（（C语言写法））

do

循环体

done

例1：

myvar=0

while [ $myvar -ne 10 ]

do

echo $myvar

myvar=$(($myvar+1)) // myvar=$[ $myvar+1 ] 或者 myvar=$(expr $myvar+1)

done

例2：

#!/bin/bash

#this is my firstshell project

loopcount=0

result=0

while [$loopcount -lt100]

do

loopcount=$(($loopcount +1))

result=$(($loopcount+ $result))

done

echo "The resultof\'1+2+3+...+100\' is $result"

### 8.3 until 语句

循环变量=初值

until[循环条件] //或者 unitl（（c语言写法））

do

循环体

done

例1：

myvar=0

until [$myvar -eq10]

do

echo$myvar

myvar=$(($myvar+1))

done

## 9 Shell 函数

[function] 函数名() //funciton可省略

{

......

}

例1：

#declare a function named hello

function hello()

{

echo "Hello,$1 todayis `date`"

return 11

}

echo "now going tothe functionhello"

hello“ILOVE CHINA”

echo $?

echo "backfromthe function“

例 2：实现两个数相加

#!/bin/bash

function add()

{

return $(($1+$2))

}

a=10

b=20

add a b

echo $?

## 10 Shell 文件包含

和其他语言一样，Shell 也可以包含外部脚本。这样可以很方便的封装一些公用的代码作为一个独立的文件。

**Shell 文件包含的语法格式如下：**

. filename # 注意点号(.)和文件名中间有一空格

或

source filename

举例：

创建两个 shell 脚本文件。

test1.sh 代码如下：

#!/bin/bash

url="http://www.runoob.com"

test2.sh 代码如下：

#!/bin/bash

#引用test1.sh 文件

. ./test1.sh #或者source ./test1.sh

echo "菜鸟教程官网地址：$url"

接下来，我们为 test2.sh 添加可执行权限并执行：

$ chmod +x test2.sh

$ ./test2.sh

输出结果：

菜鸟教程官网地址：http://www.runoob.com

## 11 Shell 输入/输出重定向

大多数 UNIX 系统命令从你的终端接受输入并将所产生的输出发送回​​到您的终端。一个命令通常从一个叫标准输入的地方读取输入，默认情况下，这恰好是你的终端。同样，一个命令通常将其输出写入到标准输出，默认情况下，这也是你的终端。

|  |  |
| --- | --- |
| 命令 | 说明 |
| command > file | 将输出重定向到 file。 |
| command < file | 将输入重定向到 file。 |
| command >> file | 将输出以追加的方式重定向到 file。 |
| n > file | 将文件描述符为 n 的文件重定向到 file。 |
| n >> file | 将文件描述符为 n 的文件以追加的方式重定向到 file。 |
| n >& m | 将输出文件 m 和 n 合并。 |
| n <& m | 将输入文件 m 和 n 合并。 |
| << tag | 将开始标记 tag 和结束标记 tag 之间的内容作为输入。 |

### 11.1 重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

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。

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。

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。

默认情况下，command > file 将 stdout 重定向到 file，command < file 将stdin 重定向到 file。

如果希望 stderr 重定向到 file，可以这样写：

$ command 2 > file

如果希望 stderr 追加到 file 文件末尾，可以这样写：

$ command 2 >> file //2 表示标准错误文件(stderr)。

如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：

$ command > file 2>&1或者 $ command >> file 2>&1

如果希望对 stdin 和 stdout 都重定向，可以这样写：

$ command < file1 >file2

command 命令将 stdin 重定向到 file1，将 stdout 重定向到 file2。

### 11.2 Here Document

Here Document 是 Shell 中的一种特殊的重定向方式，用来将输入重定向到一个交互式 Shell 脚本或程序。

它的基本的形式如下：

command << delimiter

document

delimiter

它的作用是将两个 delimiter 之间的内容(document) 作为输入传递给 command。

注意：

结尾的delimiter 一定要顶格写，前面不能有任何字符，后面也不能有任何字符，包括空格和 tab 缩进。开始的delimiter前后的空格会被忽略掉。

举例：

在命令行中通过 wc -l 命令计算 Here Document 的行数：

$ wc -l << EOF

欢迎来到

菜鸟教程

www.runoob.com

EOF

3 # 输出结果为 3 行

我们也可以将 Here Document 用在脚本中，例如：

#!/bin/bash

cat << EOF

欢迎来到

菜鸟教程

www.runoob.com

EOF

执行以上脚本，输出结果：

欢迎来到

菜鸟教程

www.runoob.com

### 11.3 /dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 /dev/null：

$ command > /dev/null

/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到"禁止输出"的效果。

如果希望屏蔽 stdout 和 stderr，可以这样写：

$ command > /dev/null 2>&1

注意：0 是标准输入（STDIN），1 是标准输出（STDOUT），2 是标准错误输出（STDERR）。

# gcc/g++编译器

对于.c 格式的 C 文件，可以采用 gcc 或 g++编译

对于 .cc、.cpp 格式的 C++文件，应该采用 g++进行编译

常用的选项：

-c 表示编译源文件

-o 表示输出目标文件

-g 表示在目标文件中产生调试信息，用于 gdb 调试

-D <宏定义> 编译时将宏定义传入进去，一般是非常重要的宏定义才用-D传

-Wall 打开所有类型的警告。

-E

## 1 gcc 编译过程

预编译—>编译-->汇编-->链接

当我们进行编译的时候,要使用一系列的工具,我们称之为工具链.

其中包括:预处理器,编译,汇编器 as,连接器. 一个编译过程包括下面

几个阶段：

(1)预处理：预处理器将对源文件中的宏进行展开。

(2)编译：gcc 将 c 文件编译成 汇编文件。

(3)汇编：as 将汇编文件编译成机器码。

(4)链接：将目标文件和外部符号进行连接，得到一个可执行二进制文件。

下面以一个很简单的 test.c 来探讨这个过程。

#include <stdio.h>

#defineNUMBER (1+2)

int main()

{

int x =NUMBER;

return 0;

}

(1)预处理：

gcc –E test.c -o test.i

我们用 cat 查看 test.i 的内容如下：

int main()

int x=(1+2);

return 0;

我们可以看到，文件中宏定义NUMBER 出现的位置被(1+2)替换掉了，其它的内容保持不变。与typedef区别就在于typedef只是为对象去取一个别名，使用别名和原名效果一样，但不会替换。

(2)编译：gcc –S test.i –o test.s 通过 cat test.s 查看 test.s 的内容为代码。

(3)汇编：as test.s -o test.o 利用 as 将汇编文件编译成机器码。得到输出文件为 test.o。

test.o 中为目标机器上的二进制文件. 用 nm 查看文件中的符号： nm test.o 输出如下：00000000 T main。有的编译器上会显示：00000000 b .bss 00000000 d.data 00000000 t .text U\_\_\_main U\_\_alloca 00000000 T \_main 。

既然已经是二进制目标文件了，能不能执行呢？试一下./test.o,提示cannotexecutebinary file.原来\_\_\_main 前面的 U 表示这个符号的地址还没有定下来，T 表示这个符号属于代码。

(4)链接：gcc –o test test.o,将所有的.o 文件链接起来生产可执行程序。

## 2 gcc 所支持后缀名：

|  |  |  |  |
| --- | --- | --- | --- |
| 后缀名 | 所对应的语言 | 后缀名 | 所对应的语言 |
| .c | C原始程序 | .s/.S | 汇编语言原始程序 |
| .C/.cc/.cxx | C++原始程序 | .h | 预处理文件（头文件） |
| .m | Objectic原始程序 | .o | 目标文件 |
| .i | 已经过预处理的C原始程序 | .a/.so | 编译后的库文件 |
| .ii | 已经过预处理的C++原始程序 |  |  |

## 3 gcc 常用选项：

|  |  |
| --- | --- |
| 选项与命令格式 | 含义 |
| -E file.c –o file.i或者–o file.i -E file.c | 只进行预编译，不做其处理，由\*.c文件生成\*.i文件。缺省输出到屏幕 |
| -S file.i [–o file.s]或者–o file.s -S file.i | 只编译不汇编，生成汇编代码，将\*.i文件转换为\*.s文件，缺省输出同名.s文件 |
| -c file.c [–o file.o]或者 –o file.o -c file.c | 只编译不链接，生成目标文件“.o”。缺省生成同名.o文件。 |
| -o file2 file1或file1 -o file2 | 链接，指定file2.exe文件作为file1.\*文件的输出文件，不可省略或者写成 |
| -g，位置任意 | 在可执行程序中包含标准调试信息 |
| -v，位置任意 | 打印出编译器内部编译各过程的命令行信息和编译器的版本 |
| -I dir，位置任意 | 在头文件的搜索列表中添加dir 目录 |

**预处理阶段**：对包含的头文件（#include）和宏定义（#define、#ifdef等）进行处理

gcc –E hello.c –o hello.i //-o表示输出为指定文件，类型 -E将源文件（\*.c）转换为（\*.i）

**编译阶段**：检查代码规范性、语法错误等，在检查无误后把代码翻译成汇编语言

gcc –S hello.i -o hello.s //-S 将已预处理的 C 原始程序（\*.i）转换为（\*.s）

**汇编链接阶段**：将.s 的文件以及库文件整合起来链接为可执行程序

gcc hello.s –o hello.exe //最后将汇编语言原始程序(\*.s)和一些库函数整合成（\*.exe）

例1:

#include <stdio.h>

#define MAX100

#define max(a,b) ((a)>(b)?(a):(b)) //宏定义，执行-E 之后被替换

main()

{

printf("MAX=%d\n",MAX);

printf("max(3,4)=%d\n",max(3,4));

}

//法一：（理解原理用，四步:.c-->.i-->.s->.o->.exe）

gcc –E project1.c –o project1.i //预编译，生成已预编译过的 C 原始程序\*.i

gcc –S project1.i–o project1.s //编译，生成汇编语言原始程序\*.s

as project1.s –o project1.o //汇编

gcc project1.o –o project1.exe //链接，生成可执行程序

//法二：（理解原理用，三步: .c-->.i-->.s-->.exe）

gcc –E project1.c –o project1.i //预编译，生成已预编译过的 C 原始程序\*.i

gcc –S project1.i–o project1.s //编译，生成汇编语言原始程序\*.s

gcc project1.s –o project1.exe //汇编链接，生成可执行程序

//法三：（常用,两步.c-->.o -->.exe）

gcc –c project1.c –o project1.o //编译

gcc project1.o –o project1.exe //链接

//法四：（常用，一步：.c-->.exe）

gcc project1.c–o project1.exe //编译并链接

//法五：（常用，一步：.c-->.exe）

gcc project1.c //缺省生成a.out可执行文件

例2:gcc 传递宏定义选项 –D的作用，就是定义宏的另一种方式

#include <stdio.h>

main()

{

#ifdef cjy //表示如果定义了 cjy，即命令行参数传了cjy，就执行下面的输出

printf("cjy is defined!\n");

#else

printf("cjy is not defined!\n");

#endif

printf("main exit\n");

}

gcc–E project2.c –o project2.i–D cjy //条件编译，用-D 传递，如果没有传cjy则执行#else

gcc–S project2.i –o project2.s

gcc–o project2.exe project2.s

或：

gcc–o project2 project2.c –D cjy//条件编译，用-D 传递，如果没有传cjy则执行#else

## 4 gcc 库选项

|  |  |
| --- | --- |
| 选项 | 含义 |
| -static | 链接静态库，禁止使用动态库.用于最后链接阶段 |
| -shared | 1.可以生成动态库文件  2.进行动态编译，尽可能地链接动态库，只有没有动态库时才会连接同名静态库（默认选项，可省略） |
| -L dir | 在库文件的搜索路径列表中添加dir目录 |
| -lname | 链接称为libname.a（静态库）或者libname.so（动态库）的库文件若两个库都存在则根据编译方式时-static还是-shared来进行链接 |
| -fPIC或-fpic | 生成使用相对地址的与位置无关的目标代码。行通常使用gcc的-shared选项从该PIC目标文件生成动态库文件 |

**-fPIC 功能**

-fPIC 作用于编译阶段，告诉编译器产生与位置无关代码(Position-Independent Code)，则产生的代码中，没有绝对地址，全部使用相对地址，故而代码可以被加载器加载到内存的任意位置，都可以正确的执行。这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的。

gcc -shared -fPIC -o 1.so 1.c

这里有一个-fPIC参数，PIC就是position independent code。PIC使.so文件的代码段变为真正意义上的共享，如果不加-fPIC,则加载.so文件的代码段时,代码段引用的数据对象需要重定位, 重定位会修改代码段的内容,这就造成每个使用这个.so文件代码段的进程在内核里都会生成这个.so文件代码段的副本.每个副本都不一样,取决于这个.so文件代码段和数据段内存映射的位置.不加fPIC编译出来的so,是要再加载时根据加载到的位置再次重定位的.(因为它里面的代码并不是位置无关代码)，如果被多个应用程序共同使用,那么它们必须每个程序维护一份so的代码副本了.(因为so被每个程序加载的位置都不同,显然这些重定位后的代码也不同,当然不能共享)，我们总是用fPIC来生成动态库文件,从来不用fPIC来生成静态库文件.fPIC与动态链接可以说基本没有关系,libc.so一样可以不用fPIC编译,只是这样的so必须要在加载到用户程序的地址空间时重定向所有表目.因此,不用fPIC编译so并不总是不好.如果你满足以下4个需求/条件:

（1）该库可能需要经常更新

（2）该库需要非常高的效率(尤其是有很多全局量的使用时)

（3）该库并不很大.

（4）该库基本不需要被多个应用程序共享

就可以不适用-fpic来编译so文件。

如果用没有加这个参数的编译后的共享库，也可以使用的话，可能是两个原因：

（1）gcc默认开启-fPIC选项

（2）loader使你的代码位置无关

从GCC来看，shared应该是包含fPIC选项的，但似 乎不是所以系统都支持，所以最好显式加上fPIC选项。

**函数库分为静态库和动态库。**

静态库是目标文件.a 的归档文件（格式为 libname.a）。如果在编译某个程序时链接静态库，则链接器将会搜索静态库并直接拷贝到该程序的可执行二进制文件到当前文件中；

动态库（格式为 libname.so[. 主版本号.次版本号.发行号]）。在程序编译时并不会被链接到目标代码中，而是在程序运行时才被加载器载入。跳转到动态库所在的地址。

**第一种建立和使用动态库函数的方法：**

（1）创建静态库

$ gcc -c add.c //编译 add.c 源文件默认生成 add.o 目标文件

$ ar rcsv libadd.a add.o //对目标文件\*.o 进行归档，生成 lib\*.a,此处 lib 要写

$ gcc -o mian main.c –L ./ –ladd –I ./ //不要忘记-L 后面的那个. （即在库文件的搜索 路径中添加当前路径， -ladd 表示链接库文件libadd.a/.so， -I./表示包含在当前目录中的头文件）

$./main //因为是静态编译，生成的执行文件可以独立于.a文件运行。

**ar程序**：

格式：ar rcsv libxxx.a xx1.o xx2.o

参数r：在库中插入模块(替换)。当插入的模块名已经在库中存在，则替换同名的模块。如果若干模块中有一个模块在库中不存在，ar显示一个错误消息，并不替换其他同名模块。默认的情况下，新的成员增加在库的结尾处，可以使用其他任选项来改变增加的位置。【1】

参数c：创建一个库。不管库是否存在，都将创建。

参数s：创建目标文件索引，这在创建较大的库时能加快时间。（补充：如果不需要创建索引，可改成大写S参数；如果.a文件缺少索引，可以使用ranlib命令添加）

参数v：显示生成结果

格式：ar t libxxx.a

显示库文件中有哪些目标文件，只显示名称。

格式：ar tv libxxx.a

显示库文件中有哪些目标文件，显示文件名、时间、大小等详细信息。

格式：nm -s libxxx.a

显示库文件中的索引表。

格式：ranlib libxxx.a

为库文件创建索引表。

（2）创建动态库

分三步：

$ gcc -fPIC -Wall -c add.c //编译 add.c 源文件生成 add.o 目标文件

$ gcc -shared -o libadd.so add.o //对目标文件\*.o 进行归档，生成 lib\*.so,此lib 要写

$ gcc -o main main.c –L . –ladd –I .//.和./等效

分两步：

gcc -fPIC -Wall -shared add.c -o libadd.so//生成动态库文件

gcc -o main main.c –L ./ –ladd//编译时链接动态库

创建动态链接库之后，以后就可以使用该动态链接库了。例如在 test.c 里面调用了原来库中的函数，则编译时执行 gcc–o test test.c –L ./ –ladd 就可以了。

在运行 main 前，需要注册动态库的路径。

方法有 3 种：

①修改/etc/ld.so.conf

②修改LD\_LIBRARY\_PATH 环境变量,LD\_LIBRARY\_PATH=. ./main

③将库文件拷贝到/lib 或者/usr/lib 下（系统默认搜索库路径）。

# cp libadd.so /lib //通常采用的方法， cp lib\*.so /lib或者cp libadd.so /usr/lib

# ./main //运行main

如果不拷贝，生成.so 之后还有两种方法：

gcc –o main main.c –L .–Wl,-rpath,${PWD} –ladd //不明白就算了

**第二种建立和使用动态库函数的方法(不复制)**

采用dlopen,dlclose,dlerror,dlsym加载动态链接库

为了使程序方便扩展，具备通用性，可以采用插件形式。采用异步事件驱动模型，保证主程序逻辑不变，将各个业务已动态链接库的形式加载进来，这就是所谓的插件。linux提供了加载和处理动态链接库的系统调用，非常方便。

**dlopen、dlsym函数介绍**

在linux上man dlopen可以看到使用说明，函数声明如下：

#include <dlfcn.h>

void \*dlopen(const char \*filename, int flag);

char \*dlerror(void);

void \*dlsym(void \*handle, const char \*symbol);

int dlclose(void \*handle);

dlopen以指定模式打开指定的动态连接库文件，并返回一个句柄（指针）-给调用进程，

dlerror返回出现的错误，

dlsym通过句柄和连接符名称获取函数名或者变量名，

dlclose来卸载打开的库。

dlopen打开模式如下：

RTLD\_LAZY  暂缓决定，等有需要时再解出符号

RTLD\_NOW  立即决定，返回前解除所有未决定的符号。

举例：

例如将如下程序编译为动态链接库libcaculate.so，程序如下：

int add(int a,int b)

{

return (a + b);

}

int sub(int a, int b)

{

return (a - b);

}

int mul(int a, int b)

{

return (a \* b);

}

int div(int a, int b)

{

return (a / b);

}

编译如下： gcc -fPIC -s

hared caculate.c -o libcaculate.so

采用上面生成的libcaculate.so，写个测试程序如下：

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

//动态链接库路径

#define LIB\_CACULATE\_PATH "./libcaculate.so"

//函数指针

**typedef** int **(\***CAC\_FUNC**)(**int**,** int**);**

int main**()**

**{**

void **\***handle**;**

char **\***error**;**

CAC\_FUNC cac\_func **=** **NULL;**

//打开动态链接库

handle **=** dlopen**(**LIB\_CACULATE\_PATH**,** RTLD\_LAZY**);**

**if** **(!**handle**)**

**{**

fprintf**(**stderr**,** "%s\n"**,** dlerror**());**

exit**(**EXIT\_FAILURE**);**

**}**

//清除之前存在的错误

dlerror**();**

//获取一个函数

**\*(**void **\*\*)** **(&**cac\_func**)** **=** dlsym**(**handle**,** "add"**);**

**if** **((**error **=** dlerror**())** **!=** **NULL)**

**{**

fprintf**(**stderr**,** "%s\n"**,** error**);**

exit**(**EXIT\_FAILURE**);**

**}**

printf**(**"add: %d\n"**,** **(\***cac\_func**)(**2**,**7**));**

cac\_func **=** **(**CAC\_FUNC**)**dlsym**(**handle**,** "sub"**);**

printf**(**"sub: %d\n"**,** cac\_func**(**9**,**2**));**

cac\_func **=** **(**CAC\_FUNC**)**dlsym**(**handle**,** "mul"**);**

printf**(**"mul: %d\n"**,** cac\_func**(**3**,**2**));**

cac\_func **=** **(**CAC\_FUNC**)**dlsym**(**handle**,** "div"**);**

printf**(**"div: %d\n"**,** cac\_func**(**8**,**2**));**

//关闭动态链接库

dlclose**(**handle**);**

exit**(**EXIT\_SUCCESS**);**

**}**

编译选项如下：**gcc -rdynamic -o main main.c –ldl**

静态库与动态库的比较：

动态库只在执行时才被链接使用，不是直接编译为可执行文件，并且一个动态库可以被多个程序使用故可称为共享库。

静态库将会整合到程序中，在程序执行时不用加载静态库。 因此，静态库会使你的程序臃肿并且难以升级，但比较容易部署。而动态库会使你的程序轻便易于升级但难以部署。

举例:

写一个求两个数 +， --， \*， / 的函数 func.c(func.h)，在 main.c中调用执行相应的算术操作，但是不直接针对 main.c 编译链接，而是在 function.sh 中对 func 函数创建静态库和动态库，并分别自动执行 main 函数。（注：当然可以直接针对 main 函数）

编写 function.sh

#!/bin/bash

echo "==========================="

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

echo " 1.createstatic lib " //静态库创建

echo " 2.createshared lib " //动态库创建

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

echo "==========================="

echo "pleaseinput youroperator:"

read op

case $op in

"1") //以静态库的方式

gcc -c ${1}.c //${1}接收第一个传进来的参数 func,并编译它

ar rcsv lib${1}.a ${1}.o //将其打包为静态库

gcc -o ${2} ${2}.c -L. -l${1} //${2}接收第二个传进来的 main

./${2} //运行 main 程序输出结果

;;

"2") //以动态库的方式

gcc -fpic –c ${1}.c

gcc -shared –o lib${1}.so ${1}.o

gcc -o ${2} ${2}.c -L. -l${1}

sudo cp lib${1}.so /lib/ //切换到 root 用户下

./${2}

;;

\*)

exit 3

;;

esac

编写 func.h

#ifndef \_\_FUNC\_H

#define\_\_FUNC\_H

extern int add**(**int**,**int**);**

extern int sub**(**int**,**int**);**

extern int mul**(**int**,**int**);**

extern int div**(**int**,**int**);**

#endif

编写 func.c

int add**(**int a**,**int b**)**

**{**

**return** a **+**b**;**

**}**

int sub**(**int a**,**int b**)**

**{**

**return** a**-** b**;**

**}**

int mul**(**int a**,**int b**)**

**{**

**return** a**\***b**;**

**}**

int div**(**int a**,**int b**)**

**{**

**return** a **/**b**;**

**}**

编写 main.c

#include <stdio.h>

#include "func.h"

int main**()**

**{**

printf**(**"add(3,4)=%d\n"**,**add**(**3**,**4**));**

printf**(**"sub(4,1)=%d\n"**,**sub**(**4**,**1**));**

printf**(**"mul(3,2)=%d\n"**,**mul**(**3**,**2**));**

printf**(**"div(6,2)=%d\n"**,**div**(**6**,**2**));**

**}**

#sh function.sh func main //将 func,main 作为参数传递进去

当然也可以不用写 function.sh，可以直接针对 main 操作，分别采用静态库和动态库的方式将函数 func.c 打包

（1） 将 func.o 打包为静态库函数 libfunc.a，并执行程序：

#gcc–c func.c //1. 将 func.c 编译为 func.o

#ar rcsv libfunc.a func.o //2. 用 ar rcsv将func.o打包为静态库,libfunc.a(前面的lib要写)

#gcc–o main.exe main.c –L . –lfunc //3. 链接库函数和执行main.c生成可执行程序main.exe

#./main.exe //4. 执行./main.exe

（2）将 func.o 打包为动态库函数 libfunc.so，并执行程序：

#gcc–fpic –c func.c //1. 用动态库的方式将 func.c 编译为 func.o

# gcc –shared–o libfunc.so func.o //2. 用gcc-shared将func.o打包为动态库 libfunc.so

#gcc–o main.exemain.c –L.–lfunc //3. 链接库函数和执行 main.c 生成可执行程序 main.exe

# sudo cp libfunc.so /lib //4. 非超级用户要用 sudo 将动态库libfunc.so 拷贝到/lib 目录下

# ./main.exe //5. 执行./main.exe

## 5 gcc--- 警告选项

对于如下程序：

#include <stdio.h>

void main()

{

long long temp =1;

printf(“This is a bad code!\n”);

return 0;

}

-ansi : 生成标准语法（ANSI C 标准）所要求的警告信息（并不列出所有警告）

$ gcc–ansi warning.c –o warning

warning.c: 在函数“main”中：

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:4 警告：“main”的返回类型不是“int”

可以看出，该选项并没有发现“long long”这个无效数据类型的错误

-pedantic : 列出 ANSI C 标准的全部警告信息。

$ gcc–pedantic warning.c –o warning

warning.c: 在函数“main”中：

warning.c:5 警告：ISOC89 不支持“long long”

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:4 警告：“main”的返回类型不是“int”

-Wall : 列出所有的警告信息（常用）

$ gcc–Wall warning.c –o warning

warning.c:4 警告：“main”的返回类型不是“int”

warning.c: 在函数“main”中：

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:5 警告：未使用的变量“tmp”

–Werror ：要求gcc将所有的警告都当成错误来处理

$gcc –Werror warning.c –o warming

通常用的是-Wall 显示所有有用的报警信息。

## 6 gcc--- 优化选项

gcc 对代码进行优化通过选项“-On”来控制优化级别（不是零时大写的o，n 是整数）。不同的优化级别对应不同的优化处理工作。如使用优化选项：

“-O1”主要进行线程跳转和延迟退栈两种优化。

“-O2”除了完成所有“-O1”级别的优化之外，还要进行一些额外调整工作，如处理其指令调度等。

“-O3”则还包括循环展开或其他一些与处理器特性相关的优化工作。

虽然优化选项可以加速代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句也有可能因为循环展开而变得到处都有，所有这些对调试来讲都是不好的。所以在调试的时候最好不要使用任何的优化选项，只有当程序在最终发行的时候才考虑对其进行优化。

通常用的是-O2

-D <宏定义> //编译时将宏定义传入进去

eg: # gcc –o hello –Wall –O2 hello.c

举例：

有两个文件 main.cpp,func.cpp其中

main.cpp 内容为：

#include <stdio.h>

int MyFunc**();**

int main**()**

**{**

#ifdef \_DEBUG

printf**(**"Debug MyFunc is:%d\n"**,**MyFunc**());**

#else

printf**(**"NDEBUG MyFunc is:%d\n"**,**MyFunc**());**

#endif

**}**

func**.**cpp 内容为：

int MyFunc**()**

**{**

**return** 123**;**

**}**

法1：分别编译再链接，每个源文件必须分开编译

编译：

g++ -c func.cpp //C++文件，一定要用g++编译,生成同名但扩展名为.o的二进制目标文件func.o

g++ -c main.cpp //将编译 main.cpp,生成同名的但扩展名为.o 的二进制目标文件main.o

g++ -c func.cpp -o func.o//编译 func.cpp,指定输出目标文件为func.o

g++ -c main.cpp –o main.o//编译 main.cpp,指定输出目标文件为main.o

链接：

g++ main.o func.o //链接,默认情况下生成的是 a.out 可执行文件

g++ -o a.out main.o func.o //链接,指定生成a.out 可执行文件

g++ -o a.out \*.o //链接所有.o文件,生成a.out 可执行文件

链接目标文件 main.o 和 func.o 最后形成可执行文件 a.out。对于第一种，如果没有显式指定可执行文件名，g++默认为 a.out。也可以将编译和链接的过程合为一块处理：

法2；整体编译再链接，可以同时编译多个文件

g++ \*.cpp //编译并链接所有cpp文件 ，生成可执行文件 a.out

g++ func.cpp main.cpp //编译并链接指定的cpp文件，默认生成 a.out可执行文件

g++ -o a.out func.cpp main.cpp //编译并链接指定的cpp文件，指定生成 a.out可执行文件

但都将先编译指定的源文件，如果成功的话，再链接成可执行文件 a.out

如果希望在编译时传入宏定义，可使用-D 参数,例如

g++ \*.cpp -D \_DEBUG

# make 工程管理器

可以试想一下，有一个上百个文件的代码构成的项目，如果其中只有一个活少数几个文件进行了修改，如果再从头到尾将每一个文件都重新编译是个比较繁琐的过程。为此，引入了 Make 工程管理器的概念，工程管理器指管理较多的文件，它是自动管理器，能根据文件时间自动发现更新过的文件而减少编译的工作量，同时通过读入 Makefile 文件来执行大量的编译工作。（本目录下）

## 1 Makefile 格式

target: dependency\_files //目标项:依赖项

< TAB > command //必须以 Tab 开头，command 编译命令，**支持直接用shell脚本**

注意：在写 command 命令行的时候，必须要在前面按 TAB 键,，其实就是shell 程序

例如，有 Makefile 文件，内容如下：

main.exe:main.o func.o //有头文件时要加入头文件

g++ -o main.exe main.o func.o

main.o:main.cpp

g++ -c main.cpp

func.o:func.cpp

g++ -c func.cpp

使用 make 编译：

（1）对于该 Makefile 文件,程序 make 处理过程如下:

make程序首先读到第 1 行的目标文件 main.exe 和它的两个依赖文件main.o和func.o;然后比较文件main.exe和main.o/func.o的产生时间，如果 main.exe 比 main.o或func.o 旧的话，则执行第2行命令，以产生目标文件main.exe。

（2）在执行第2行的命令前,它首先会查看 makefile 中的其他定义，看有没有以第1行 main.o 和 func.o 为目标文件的依赖文件，如果有的话，继续按照(1)、(2)的方式匹配下去。

（3）根据(2)的匹配过程，make 程序发现第3行有目标文件 main.o 依赖于main.cpp，则比较目 main.o 与它的依赖文件 main.cpp 的文件新旧,如果main.o比main.cpp旧,则执行第4行的命令以产生目标文件main.o.

（4）在执行第 4 条命令时,main.cpp 在文件 makefile不再有依赖文件的定义,make程序不再继续往下匹配,而是执行第 4 条命令,产生目标文件main.o

目标文件 func.o 按照上面的同样方式判断产生.

执行(3)、(4)产生完 main.o 和 func.o 以后，则第 2 行的命令可以顺利地执行了，最终产生了第 1 行的目标文件 main.exe。

**依赖关系中最好加入头文件，**原因在于：

在 C++ 中因为头文件包含了一些实现的内容（比如类成员的定义），而这些内容由于面向对象的封装原则，客户代码是不去理会的，然而编译客户代码时却是需要的。那么当这些内容发生改变的时候，实现文件（.cpp）往往也发生改变，这导致相应 .o 的重新编译生成。然而如果不把该头文件加入到客户代码的依赖关系中，因为客户代码没有改变，时间戳未发生变化，客户代码不会被重编译，而这种情况在连接时也未必会被发现，最后只能导致难以调试的运行时错误。

## 2 特殊处理与伪目标

.PHONY 是 Makefile 文件的关键字，表示它后面列表中的目标均为伪目标。如果为all则代表之后的所有目标。伪目标通常用在清理文件、强制重新编译等情况下。伪目标可以有依赖项

例1:main.c 函数，func.c 函数为前面计算+,-,\*,/运算的程序

#vim Makefile //系统默认的文件名为 Makefile

main.exe:main.o func.o //表示要想生成 main.exe 文件，要依赖于main.o 和 func.o 文件

gcc -o main.exe main.o func.o//如果 main.o,func.o 已经存在了，就链接成 main.exe

main.o:main.c //表示 main.o 文件依赖于 main.c 文件

gcc -c main.c //编译 main.c，默认生成 main.o。可写为：gcc –cmain.c –o main.o

func.o:func.c //表示 func.o 文件依赖于 func.c 文件

gcc -c func.c //如果 func.c 存在，则编译 func.c,生成 func.o

.PHONY:rebuild clean //表示后面的是伪目标，通常用在清理文件、强制重新编译等情况下

rebuild: clean main.exe //先执行清理，在执行重新编译

clean:

rm –rf main.o func.o main.exe //最后删除.o 和.exe 的文件

按 ESC 键之后，:wq 保存退出。再执行下面的命令：

#make //直接 make,即从默认文件名（Makefile）的第一行开始执行

#make clean //表示执行 clean: 开始的命令段

#make func.o //表示执行 func.o: 开始的命令段

#make rebuild //则先执行清除，再重新编译连接

如果不用系统默认的文件名 Makefile，而是用户随便起的一个名字，则 make 后面必须要加上-f，如：

#vi Makefile11

#make –f Makefile11 clean //表示执行 clean: 开始的命令段

#make –f Makefile11 main.exe //表示执行 main.exe: 开始的命令段

这里要说明一点的是，clean不是一个文件，它只不过是一个动作名字，有点像C语言中的标签一样，其冒号后什么也没有，那么，make就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在make命令后明显得指出这个lable的名字。这样的方法非常有用，我们可以在一个makefile中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

## 3 特殊字符、变量、函数与规则

随着软件项目的变大、变复杂，源文件也越来越多，如果采用前面的方式写 makefile 文件，将会使 makefile 也变得复杂而难于维护。通过make支持的变量定义、规则和内置函数，可以写出通用性较强的makefile 文件，使得同一个 makefile 文件能够适应不能的项目。

### 3.1 特殊字符

|  |  |  |  |
| --- | --- | --- | --- |
| \ | 换行符或者转义符号 | \* | 匹配0个或多个字符 |
| # | 注释符,一定要另起一行不然容易出错 | ? | 匹配一个字符 |
| @ | 不显示当前执行的命令行 | % | 匹配1个或多个字符 |
| ~ | 用户主目录或者宿主目录 | $$ | $字符 |
| $ | 取变量的值 |  |  |

### 3.2 变量

#### 3.2.1 变量定义和引用

变量：用来代替一个文本字符串

定义变量的2种方法：

变量名=变量值 递规变量展开(几个变量共享一个值) //a=b；b=c；则执行时a就会与c相同

变量名:=变量值 简单变量展开(类似于 C++的赋值) //通常采用这种形式

使用变量的一般方法：

$(变量名)=??? // 给变量赋值

???=$(变量名) //引用某个变量

举例：将以前的那个可以写为：

OBJS:=main.o func.o //相当于 main.o func.o

EXE:=main.exe

$(EXE):$(OBJS)

g++ -o $(EXE) $(OBJS)

main.o:main.cpp

g++ -c main.cpp –o main.o

func.o:func.cpp

g++ -c func.cpp –o func.o

.PHONY:rebuild clean

rebuild:clean main.exe

clean:

rm –rf $(EXE) $(OBJS)

#### 3.2.2 变量分类

**变量**分为：用户自定义变量，预定义变量（CFLAGS），自动变量，环境变量，模式变量，目标变量，参数变量。

目标变量：位于冒号左边的变量。目标项：依赖项

参数变量：override 变量名，表变量值由命令行传入

模式变量：模式表达式:变量名，如%.o：main.c

**自动变量**：指在使用的时候，自动用特定的值替换。常用的有：

|  |  |
| --- | --- |
| 变量 | 说明 |
| $@ | 当前规则的目标文件(重点) |
| $< | 当前规则的第一个依赖文件 |
| $^ | 当前规则的所有依赖文件，以空格分隔(重点) |
| $? | 规则中日期新于目标文件的所有相关文件列表，逗号分隔 |
| $(@D) | 目标文件的目录名部分 |
| $(@F) | 目标文件的文件名部分 |

**预定义变量**：内部事先定义好的变量，但是它的值是固定的，并且有些的值是为空的。

|  |  |  |  |
| --- | --- | --- | --- |
| AR | 库文件打包程序默认为 ar | ARFLAGS | 库选项，无默认 |
| AS | 汇编程序，默认为 as | ASFLAGS | 汇编选项，无默认 |
| CC | c 编译器默认为 gcc | CFLAGS | c 编译器选项，无默认 |
| CPP | c 预编译器，默认为$(CC) –E | CPPFLAGS | c 预编译器选项，无默认 |
| CXX | c++编译器，默认为 g++ | CXXFLAGS | c++编译器选项 |
| RM | 删除，默认为 rm –f |  |  |

**环境变量**：

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉make，让make在自动去找。

Makefile文件中的环境变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

VPATH = src:../headers

上面的的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）：

举例：用预定义变量：

CFLAGS:=-Wall -O2 –fpic

根据内部变量，可以将 makefile 改写为：

OBJS:=main.o func.o

CC:=g++

main.exe:$(OBJS)

$(CC) $^ -o $@

main.o:main.cpp

$(CC) -c $^ -o $@

func.o:func.cpp

$(CC) -o -c $^ $@

#### 3.3.3 变量的高级用法

##### 3.3.3.1 变量值的替换

我们可以替换变量中的共有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”。

还是看一个示例吧：

foo := a.o b.o c.o

bar := $(foo:.o=.c)

这个示例中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中所有以“.o”字串“结尾”全部替换成“.c”，所以我们的“$(bar)”的值就是“a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

foo := a.o b.o c.o

bar := $(foo:%.o=%.c)

这依赖于被替换字串中的有相同的模式，模式中必须包含一个“%”字符，这个例子同样让$(bar)变量的值为“a.c b.c c.c”。

##### 3.3.3.2 把变量的值再当成变量

先看一个例子：

x := y

y := z

a := $($(x))

##### 3.3.3.3 变量追加

我们可以使用“+=”操作符给变量追加值，如：

objects = main.o foo.o bar.o utils.o

objects += another.o //

于是，我们的$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o被追加进去了）

！注意 有空格分隔

使用“+=”操作符，可以模拟为下面的这种例子：

objects = main.o foo.o bar.o utils.o

objects := $(objects) another.o//

所不同的是，用“+=”更为简洁。

如果变量之前没有定义过，那么，“+=”会自动变成“=”，如果前面有变量定义，那么“+=”会继承于前次操作的赋值符。如果前一次的是“:=”，那么“+=”会以“:=”作为其赋值符，如：

variable := value

variable += more

等价于：

variable := value

variable := $(variable) more

但如果是这种情况：

variable = value

variable += more

由于前次的赋值符是“=”，所以“+=”也会以“=”来做为赋值，那么岂不会发生变量的递补归定义，这是很不好的，所以make会自动为我们解决这个问题，我们不必担心这个问题。

### 3.3.规则

**规则**分为：普通规则，隐含规则，模式规则

#### 3.3.1 隐含规则

\*.o 文件自动依赖\*.c 或\*.cc \*.cpp 文件，所以可以省略main.o:main.cpp ，但同时也不可调试

makefile 可写为：

OBJS :=main.o fun.o

CFLAGS:=-Wall –O2 -g

main.exe:$(OBJS)

gcc $^ -o $@

#### 3.3.2 模式规则

通过匹配模式找字符串，就是shell通配符，

（1）%：按规则匹配1或多个任意字符串，%可存放匹配的字符以用于下一个%。

（2）\*匹配0个或多个任意字符，只能匹配不能存放匹配的字符。

（3）？匹配一个字符。

（4）[list]匹配任意单一字符，[!list]匹配除list以外的任意一字符，

（5）[c1-c2]匹配c1-c2中的任意单一字符，

（6）{string1，string2,…}匹配多个字符串中的一个。

如：

%.o: %.cpp //任何目标文件的依赖文件是与目标文件同名的并且扩展名为.cpp 的文件

makefile 可写为：

OBJS :=main.o fun.o

CFLAGS:=-Wall –O2 –g

main.exe: $(OBJS)

gcc $^-o $@

%.o: %.cpp //模式通配

gcc –o $@ -c $^

另外还可以指定将\*.o、\*.exe、\*.a、\*.so 等编译到指定的目录中：

makefile 文件可写成：

DIR:=./Debug/ //DIR自定义变量

EXE:=main.exe //EXE自定义变量

OBJS:=main.o //OBJS自定义变量

LIBFUNCSO:=libfunc.so //OBJS自定义变量

CFLAGS:=-fpic //预定义变量

$(DIR)$(EXE):$(DIR) $(OBJS) $(DIR) $(LIBFUNCSO)//目标文件:依赖文件

gcc -o $@ $< -L ./ -lfunc

$(DIR) $(LIBFUNCSO):$(DIR) func.o

gcc -shared -o $@ $^

$(DIR) main.o:main.c

gcc -o $@ -c $^

$(DIR) func.o:func.c

gcc $(CFLAGS) -c$^ -o $@

.PHONY:rebuild clean

rebuild:clean $(DIR)$(EXE)

clean:

rm -rf $(DIR)\*.o $(DIR)\*.exe $(DIR)\*.so

注意：当 OBJS 里面有多项的时候，此时$(DIR)$(OBJS)只能影响到 OBJS中第一个，后面的全部无效，因此需要全部列出来。

### 3.4 函数

#### 3.4.1 字符串处函数

**$(wildcard filename)**

搜索当前目录下的文件名，展开所有符合其参数描述的文件名成一列，文件间以空格间隔。

**$(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 ee,EE,feet on the street)，把“feet on the street”中的“ee”替换成“EE”，返回结果是“fEEt on the strEEt”。

**$(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，<replacement>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,x.c.c bar.c)，把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.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。

**$(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。

**$(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。

**$(filter-out <pattern...>,<text> )**

名称：反过滤函数——filter-out。

功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。

**$(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。

示例：$(sort foo bar lose)返回“bar foo lose” 。

备注：sort函数会去掉<list>中相同的单词。

**$(word <n>,<text> )**

名称：取单词函数——word。

功能：取字符串<text>中第<n>个单词。（从一开始）

返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字符串。

示例：$(word 2, foo bar baz)返回值是“bar”。

**$(wordlist <s>,<e>,<text> )**

名称：取单词串函数——wordlist。

功能：从字符串<text>中取从<s>开始到<e>的单词串。<s>和<e>是一个数字。

返回：返回字符串<text>中从<s>到<e>的单词字串。如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。

示例： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

**$(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)返回值是“3”。

备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words<text>),<text> )。

**$(firstword <text> )**

名称：首单词函数——firstword。

功能：取字符串<text>中的第一个单词。

返回：返回字符串<text>的第一个单词。

示例：$(firstword foo bar)返回值是“foo”。

备注：这个函数可以用word函数来实现：$(word 1,<text> )。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。这里举一个现实中应用的例子。我们知道，make使用“VPATH”变量来指定“依赖文件”的搜索路径。于是，我们可以利用这个搜索路径来指定编译器对头文件的搜索路径参数CFLAGS，如： override CFLAGS += $(patsubst %,-I%,$(subst:, ,$(VPATH)))，如果我们的“$(VPATH)”值是“src:../headers”，那么“$(patsubst %,-I%,$(subst:, ,$(VPATH)))”将返回“-Isrc -I../headers”，这正是cc或gcc搜索头文件路径的参数。

**$(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)返回值是“aaa111 bbb222 333”。

#### 3.4.2 文件名操作函数

下面我们要介绍的函数主要是处理文件名的。每个函数的参数字符串都会被当做一个或是一系列的文件名来对待。

**$(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)返回值是“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)返回值是“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.c hacks)返回值是“.c .c”。

**$(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,包括目录。

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)返回值是“src/foo src-1.0/bar”。

**$(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)返回值是“foo.c bar.c”。

**$(addprefix <prefix>,<names...> )**

名称：加前缀函数——addprefix。

功能：把前缀<prefix>加到<names>中的每个单词后面。

返回：返回加过前缀的文件名序列。

示例：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

#### 3.4.3 foreach 函数

foreach 函数和别的函数非常的不一样。因为这个函数是用来做循环用的，Makefile中的foreach函数几乎是仿照于Unix标准Shell（/bin /sh）中的for语句，或是C-Shell（/bin/csh）中的foreach语句而构建的。它的语法是：

**$(foreach <var>,<list>,<text> )**

这个函数的意思是，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。

所以，<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。举个例子：

names := a b c d

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

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach中的<var>参数是一个临时的局部变量，foreach函数执行完后，参数<var>的变量将不在作用，其作用域只在foreach函数当中。

#### 3.4.4 if 函数

if函数很像GNU的make所支持的条件语句——ifeq（参见后前面所述的章节），if函数的语法是：

**$(if <condition>,<then-part> )**

或是

**$(if <condition>,<then-part>,<else-part> )**

可见，if函数可以包含“else”部分，或是不含。即if函数的参数可以是两个，也可以是三个。<condition>参数是if的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else-part> 会被计算。

而if函数的返回值是，如果<condition>为真（非空字符串），那个<then- part>会是整个函数的返回值，如果<condition>为假（空字符串），那么<else-part>会是整个函数的返回值，此时如果<else-part>没有被定义，那么，整个函数返回空字串。

所以，<then-part>和<else-part>只会有一个被计算。

#### 3.4.5 call函数

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。其语法是：

$(call <expression>,<parm1>,<parm2>,<parm3>...)

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。例如：

reverse = $(1) $(2)

foo = $(call reverse,a,b)

那么，foo的值就是“a b”。当然，参数的次序是可以自定义的，不一定是顺序的，如：

reverse = $(2) $(1)

foo = $(call reverse,a,b)

此时的foo的值就是“b a”。

#### 3.4.6 origin函数

origin函数不像其它的函数，他并不操作变量的值，他只是告诉你的这个变量是哪里来的？其语法是：

**$(origin <variable> )**

注意，<variable>是变量的名字，不应该是引用。所以你最好不要在<variable>中使用“$”字符。Origin函数会以其返回值来告诉你这个变量的“出生情况”，下面，是origin函数的返回值:

“undefined”,如果<variable>从来没有定义过，origin函数返回这个值“undefined”。

“default”,如果<variable>是一个默认的定义，比如“CC”这个变量，这种变量我们将在后面讲述。

“environment”,如果<variable>是一个环境变量，并且当Makefile被执行时，“-e”参数没有被打开。

“file”,如果<variable>这个变量被定义在Makefile中。

“command line”,如果<variable>这个变量是被命令行定义的。

“override”,如果<variable>是被override指示符重新定义的。

“automatic”,如果<variable>是一个命令运行中的自动化变量。关于自动化变量已经在前面讲述。

这些信息对于我们编写Makefile是非常有用的，例如，假设我们有一个Makefile其包了一个定义文件Make.def，在Make.def中定义了一个变量“bletch”，而我们的环境中也有一个环境变量“bletch”，此时，我们想判断一下，如果变量来源于环境，那么我们就把之重定义了，如果来源于Make.def或是命令行等非环境的，那么我们就不重新定义它。于是，在我们的Makefile中，我们可以这样写：

ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif

当然，你也许会说，使用override关键字不就可以重新定义环境中的变量了吗？为什么需要使用这样的步骤？是的，我们用override是可以达到这样的效果，可是override过于粗暴，它同时会把从命令行定义的变量也覆盖了，而我们只想重新定义环境传来的，而不想重新定义命令行传来的。

#### 3.4.7 shell函数

shell 函数也不像其它的函数。顾名思义，它的参数应该就是操作系统Shell的命令。它和反引号“`”是相同的功能。这就是说，shell函数把执行操作系统命令后的输出作为函数返回。于是，我们可以用操作系统命令以及字符串处理命令awk，sed等等命令来生成一个变量，如：

contents := $(shell cat foo)

files := $(shell echo \*.c)

注意，这个函数会新生成一个Shell程序来执行命令，所以你要注意其运行性能，如果你的Makefile中有一些比较复杂的规则，并大量使用了这个函数，那么对于你的系统性能是有害的。特别是Makefile的隐晦的规则可能会让你的shell函数执行的次数比你想像的多得多。

#### 3.4.8 控制make的函数

make提供了一些函数来控制make的运行。通常，你需要检测一些运行Makefile时的运行时信息，并且根据这些信息来决定，你是让make继续执行，还是停止。

**$(error <text ...> )**

产生一个致命的错误，<text ...>是错误信息。注意，error函数不会在一被使用就会产生错误信息，所以如果你把其定义在某个变量中，并在后续的脚本中使用这个变量，那么也是可以的。例如：

示例一：

ifdef ERROR\_001

$(error error is $(ERROR\_001))

endif

示例二：

ERR = $(error found an error!)

.PHONY: err

err:$(ERR)

示例一会在变量ERROR\_001定义了后执行时产生error调用，而示例二则在目录err被执行时才发生error调用。

**$(warning <text ...> )**

这个函数很像error函数，只是它并不会让make退出，只是输出一段警告信息，而make继续执行。

### 3.5 条件表达式

条件表达式的语法为：

key <conditional >

<text-if-true>

endif

以及：

key <conditional >

<text-if-true>

else

<text-if-false>

endif

其中key表示条件关键字，如“ifeq”。这个关键字有四个。ifeq，ifneq，ifdef，ifndef。

注意：关键字后有空格

#### 3.5.1 ifeq

语法是：

ifeq (<arg1>, <arg2> )

ifeq '<arg1>' '<arg2>'

ifeq "<arg1>" "<arg2>"

ifeq "<arg1>" '<arg2>'

ifeq '<arg1>' "<arg2>"

比较参数“arg1”和“arg2”的值是否相同。当然，参数中我们还可以使用make的函数。如：

ifeq ($(strip $(foo))，)

<text-if-empty>

endif

这个示例中使用了“strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

#### **3.5.2 ifneq**

语法是：

ifneq (<arg1>, <arg2> )

ifneq '<arg1>' '<arg2>'

ifneq "<arg1>" "<arg2>"

ifneq "<arg1>" '<arg2>'

ifneq '<arg1>' "<arg2>"

其比较参数“arg1”和“arg2”的值是否相同，如果不同，则为真。和“ifeq”类似。

#### 3.5.3 ifdef

语法是：

ifdef <variable-name>

如果变量<variable-name>的值非空，那表达式为真。否则，表达式为假。当然，<variable-name>同样可以是一个函数的返回值。注意，ifdef只是测试一个变量是否有值，其并不会把变量扩展到当前位置。还是来看两个例子：

示例一：

bar =

foo = $(bar)

ifdef foo

frobozz = yes

else

frobozz = no

endif

示例二：

foo =

ifdef foo

frobozz = yes

else

frobozz = no

endif

第一个例子中，“$(frobozz)”值是“yes”，第二个则是“no”。

#### 3.5.4 ifndef

语法是：

ifndef <variable-name>

这个我就不多说了，和“ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“#”同样也是安全的。“else”和“endif”也一样，只要不是以[Tab]键开始就行了。

特别注意的是，make是在读取Makefile时就计算条件表达式的值，并根据条件表达式的值来选择语句，所以，你最好不要把自动化变量（如“$@”等）放入条件表达式中，因为自动化变量是在运行时才有的。而且，为了避免混乱，make不允许把整个条件语句分成两部分放在不同的文件中。

## 4 引用其他Makefile文件

在Makefile使用include关键字可以把别的Makefile包含进来，这很像C语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：

include<filename>，filename可以是当前操作系统Shell的文件模式（可以保含路径和通配符）

在include前面可以有一些空字符，但是绝不能是[Tab]键开始。

## 5 综合实例

下面是一个较为通用的 makefile:流程控制语句类似shell脚本

DIR := ./debug //DIR自定义变量

EXE := $(DIR)/Main.exe //EXE自定义变量

CC := g++ //CC预定义便令

LIBS := //LIBS自定义变量

SRCS := $(wildcard \*.cpp) $(wildcard \*.c) $(wildcard \*.cc)//查找当前目录下所有的c和c++原始程序

OCPP := $(patsubst %.cpp, $(DIR)/%.o, $(wildcard \*.cpp))//把当前目录下搜索的的.cpp文件中的.cpp文件替换为同名的.o文件

OC := $(patsubst %.c, $(DIR)/%.co, $(wildcard \*.c)) //把当前目录下搜索的的.c文件中的.c文件替换为同名的.co文件

OCC := $(patsubst %.cc, $(DIR)/%.cco, $(wildcard \*.cc)) //把当前目录下搜索的的.cc文件中的.cc文件替换为同名的.cco文件

OBJS := $(OC) $(OCC) $(OCPP) // OBJS自定义变量，依赖对象

RM := rm –rf //RM预定义变量

CXXFLAGS := -Wall –g // CXXFLAGS预定义便令， c++编译器选项

start : mkdebug $(EXE)

mkdebug :

@if [ ! -d $(DIR) ]then mkdir $(DIR); fi //使用shell脚本或命令要加@

$(EXE) : $(OBJS)

$(CC) -o $@ $(OBJS) $(addprefix -l,$(LIBS))

$(DIR)/%.o : %.cpp

$(CC) -c $(CXXFLAGS) $< -o $@

$(DIR)/%.co : %.c

$(CC) -c $(CXXFLAGS) $< -o $@

$(DIR)/%.cco : %.cc

$(CC) -c $(CXXFLAGS) $< -o $@

.PHONY : clean rebuild

clean :

@$(RM) $(DIR)/\*.exe $(DIR)/\*.o $(DIR)/\*.co $(DIR)/\*.cco

rebuild: clean start

（注意 gcc 和 g++的区别：当 main.c 中调用了其他源文件的程序时，gcc -o main.o -c main.c 或 gcc -o Debug/main.o -c main.c 都没有问题，而 g++ -o main.o -c main.c 没问题，但 g++ -o Debug/main.o -c main.c 有问题，如果想 解决这个问题，需要在 main.c 中添加对于函数调用的函数原型声明即可）

**make 的命令行选项**

|  |  |
| --- | --- |
| 命令格式 | 含义 |
| -C dir | 读入指定目录下的makefile |
| -f file | 读入当前目录下的file文件作为makefile |
| -i | 忽略所有的命令执行错误 |
| -I dir | 指定被包含的makefile所在目录 |
| -n | 只打印要执行的命令，而不执行这些命令 |
| -P | 显示make变量数据库和隐含规则 |
| -s | 在执行命令时不显示命令 |
| -w | 如果make在执行过程中改变目录，则打印当前目录名 |

采用 for 循环编译多个目标文件的 Makefile 写法：

SRCS =open1.c open2.c

OBJECTS =$(SRCS:%.c=%.o)

TARGETS =$(SRCS:%.c=%)

all : $(TARGETS)

@for i in $(TARGETS)

do

gcc -o $${i} $${i}.c

done

.PHONY:clean

clean:

rm $(TARGETS)

# 程序调试gdb

Linux 包含了一个叫 gdb 的调试程序。gdb 可以用来调试 C 和 C++ 程序。在程序编译时用 -g 选项可打开调试选项.单必须保证源文件存在。关于 GDB 常用命令，请参考《GDB 常用命令.txt》

常见的调试程序的步骤如下：

gcc –o filename –Wall filename.c –g //进入调试用 gcc -o func.o func.c -g

gdb filename //进入调试

l //显示代码 (list)

b 4 //在第四行设置断点 相当于 Windows 的 F9(break)

r //运行 相当于 Windows 的 F5 (run)

n //下一步不进入函数 相当于 Windows 的 F10(next)

s //表示单步进入函数， 相当于 Windows 的 F11 (step)

p I //打印变量 I 相当于 Windows 的 Watch 窗口 (print)

c //运行到最后(continue)

q //退出 相当于 Windows 的 Shift+F5(quit)

gdb 的命令行选项及详细解释(红色代表简写)

|  |  |
| --- | --- |
| 命令格式 | 含义 |
| set args 运行时的参数 | 指定运行时的参数 ,如set args 2 |
| show args | 查看设置好的运行参数 |
| path dir | 设定程序的运行路径 |
| show path | 查看程序的运行路径 |
| set environment var [=value] | 设置环境变量 |
| show environment [var] | 查看环境变量 |
| cd dir | 进入dir目录，相当于cd命令 |
| pwd | 显示当前工作目录 |
| shell command | 运行shell的command命令 |
| info b | 查看所设断点 |
| break[文件名:]行号或函数名 <条件表达式> | 设置断点 |
| tbreak[文件名:]行号或函数名 <条件表达式> | 设置临时断点，到达后被自动删除 |
| delete[断点号] | 删除指定断点，断点号为 info b 的第一栏，缺省则删除所有断点 |
| disable[断点号] | 停止指定断点， 但用info b仍能看见该断点，缺省则删除所有断点 |
| enable[断点号] | 激活被disable停止的断点 |
| condition[断点号]<条件表达式> | 修改対应断点的条件 |
| ignore[断点号]<num> | 在程序执行时忽略对应断点num次 |
| step | 单步恢复程序运行，进入函数调用，相当于vs的逐语句F11 |
| next | 单步恢复程序运行，不进入函数调用，相当于vs的逐过程F10 |
| finish | 运行程序直到当前函数完成返回 |
| continue | 继续执行函数直到遇到新的断点或者函数结束 |
| list[函数名]:[行号] | 查看指定位置代码 |
| file[文件名] | 加载指定文件 |
| forward-search 正则表达式 | 源代码向前搜索 |
| reverse--search 正则表达式 | 源代码向后搜索 |
| dir dir | 停止路径名 |
| show directories | 显示定义了的源文件搜索路径 |
| info line | 显示加载到gdb内存中搞得代码 |
| print 表示式|变量 | 查看程序运行时对应表达式和变量的值，相当于vs的监视窗口 |
| x<n/f/u> | 查看内存变量内容，n为整数，表示显示内存的长度，f表示显示的格式，u表示从当前地址往后请求显示的字节数 |
| display | 设定在单步运行或其他情况中，自动显示的对应表达式的内容 |
| backtrace | 查看当前栈的情况，即可以查到调用哪些函数尚未返回 |
| quit | 退出 ，相当于 vs 的 Shift+F5 |
| run | 运行 ，相当于 vs 的 F5 |

\*启动gdb：

$gdb

启动gdb进行交互。

\*启动gdb，并且分屏显示源代码：

$gdb -tui

这样,使用了'-tui'选项，启动可以直接将屏幕分成两个部分，上面显示源代码，比用list方便多了。这时候使用上下方向键可以查看源代码,想要命令行使用上下键就用[Ctrl]n和[Ctrl]p.

$gdb app

启动gdb调试指定程序app，这样就在启动gdb之后直接载入了app可执行程序，需要注意的是，载入的app程序必须在编译的时候有gdb调试选项，例如'gcc -g app app.c',注意，如果修改了程序的源代码，但是没有编译，那么在gdb中显示的会是改动后的源代码，但是运行的是改动前的程序，这样会导致跟踪错乱的。

\*启动程序之后，再用gdb调试：

$gdb <program> <PID>

这里，<program>是程序的可执行文件名，<PID>是要调试程序的PID.如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程ID。gdb会自动attach上去，并调试他。program应该在PATH环境变量中搜索得到。

\*启动程序之后，再启动gdb调试：

$gdb <PID>

这里，程序是一个服务程序，那么你可以指定这个服务程序运行时的进程ID,<PID>是要调试程序的PID.这样gdb就附加到程序上了，但是现在还没法查看源代码,用file命令指明可执行文件就可以显示源代码了。

\*启动gdb之后的交互命令： （交互命令支持[Tab]补全。）

\*显示帮助信息：

(gdb) help

\*载入指定的程序：

(gdb) file app

这样在gdb中载入想要调试的可执行程序app。如果刚开始运行gdb而不是用gdb app启动的话可以这样载入app程序，当然编译app的时候要加入-g调试选项。

\*重新运行调试的程序：

(gdb) run

要想运行准备调试的程序，可使用run命令，在它后面可以跟随发给该程序的任何参数，包括标准输入和标准输出说明符(<和> )和shell通配符（\*、？、[、]）在内。

\*修改发送给程序的参数：

(gdb) set args no

这里，假设我使用"r yes"设置程序启动参数为yes，那么这里的set args会设置参数argv[1]为no。

\*显示缺省的参数列表：

(gdb) show args

\*列出指定区域(n1到n2之间)的代码：

(gdb) list n1,n2

这样,list可以简写为l,将会显示n1行和n2行之间的代码，如果使用-tui启动gdb，将会在相应的位置显示。如果没有n1和n2参数，那么就会默认显示当前行和之后的10行，再执行又下滚10行。另外，list还可以接函数名。

一般来说在list后面可以跟以下这们的参数：

<linenum> 行号。

<+offset> 当前行号的正偏移量。

<-offset> 当前行号的负偏移量。

<filename:linenum> 哪个文件的哪一行或者n1到n2行

<function> 函数名。

<filename:function> 哪个文件中的哪个函数。

<\*address> 程序运行时的语句在内存中的地址。

\*执行下一步：

(gdb) next

这样，执行一行代码，如果是函数也会跳过函数。这个命令可以简化为n.

\*执行N次下一步：

(gdb) next N

\*执行上次执行的命令：

(gdb) [Enter]

这里，直接输入回车就会执行上次的命令了。

\*单步进入：

(gdb) step

这样，也会执行一行代码，不过如果遇到函数的话就会进入函数的内部，再一行一行的执行。

\*执行完当前函数返回到调用它的函数：

(gdb) finish

这里，运行程序，直到当前函数运行完毕返回再停止。例如进入的单步执行如果已经进入了某函数，而想退出该函数返回到它的调用函数中，可使用命令finish.

\*指定程序直到退出当前循环体：

(gdb) until 或(gdb) u

这里，发现需要把光标停止在循环的头部，然后输入u这样就自动执行全部的循环了。

\*跳转执行程序到第5行：

(gdb) jump 5

这里，可以简写为"j 5"需要注意的是，跳转到第5行执行完毕之后，如果后面没有断点则继续执行，而并不是停在那里了。

另外，跳转不会改变当前的堆栈内容，所以跳到别的函数中就会有奇怪的现象，因此最好跳转在一个函数内部进行,跳转的参数也可以是程序代码行的地址,函数名等等类似list。

\*强制返回当前函数:

(gdb) return

这样，将会忽略当前函数还没有执行完毕的语句，强制返回。return后面可以接一个表达式，表达式的返回值就是函数的返回值。

\*强制调用函数：

(gdb) call <expr>

这里,<expr>可以是一个函数，这样就会返回函数的返回值，如果函数的返回类型是void那么就不会打印函数的返回值,但是实践发现，函数运行过程中的打印语句还是没有被打印出来。

\*强制调用函数2：

(gdb) print <expr>

这里，print和call的功能类似，不同的是，如果函数的返回值是void那么call不会打印返回值，但是print还是会打印出函数的返回值并且存放到历史记录中。

\*在当前的文件中某一行（假设为6）设定断点：

(gdb) break 6

\*设置条件断点：

(gdb) break 46 if testsize==100

这里，如果testsize==100就在46行处断点。

\*检测表达式变化则停住：

(gdb) watch 表达式

硬件观测断点，这里，表达式一旦变化，则停住。watch <expr> 为表达式（变量）expr设置一个观察点。一量表达式值有变化时，马上停住程序(也是一种断点)。

\*在当前的文件中为某一函数(假设为func)处设定断点：

(gdb) break func

\*给指定文件（fileName）的某个行（N）处设置断点：

(gdb) break fileName:N

这里，给某文件中的函数设置断点是同理的。

\*显示当前gdb断点信息：

(gdb) info breakpoints

这里，可以简写为info break.会显示当前所有的断点，断点号，断点位置等等。

\*删除N号断点：

(gdb) delete N

\*删除所有断点：

(gdb) delete

\*清除行N上面的所有断点：

(gdb) clear N

\*继续运行程序直接运行到下一个断点：

(gdb) continue

这里，如果没有断点就一直运行。

\*显示当前调用函数堆栈中的函数：

(gdb) backtrace

命令产生一张列表，包含着从最近的过程开始的所有有效过程和调用这些过程的参数。当然，这里也会显示出当前运行到了哪里(文件，行)。

\*查看当前调试程序的语言环境：

(gdb) show language

这里，如果gdb不能识别你所调试的程序，那么默认是c语言。

\*查看当前函数的程序语言：

(gdb) info frame

\*显示当前的调试源文件：

(gdb) info source

这样会显示当前所在的源代码文件信息,例如文件名称，程序语言等。

\*手动设置当前的程序语言为c++:

(gdb) set language c++

这里，如果gdb没有检测出你的程序语言，你可以这样设置。

\*查看可以设置的程序语言：

(gdb) set language

这里，使用没有参数的set language可以查看gdb中可以设置的程序语言。

\*终止一个正在调试的程序：

(gdb) kill

这里，输入kill就会终止正在调试的程序了。

\*print显示变量(var)值：

(gdb) print var

这里，print可以简写为p,print 是gdb的一个功能很强的命令，利用它可以显示被调试的语言中任何有效的表达式。表达式除了包含你程序中的变量外，还可以包含函数调用,复杂数据结构和历史等等。

\*用16进制显示(var)值：

(gdb) print /x var

这里可以知道，print可以指定显示的格式，这里用'/x'表示16进制的格式。

可以支持的变量显示格式有：

x 按十六进制格式显示变量。

d 按十进制格式显示变量。

u 按十六进制格式显示无符号整型。

o 按八进制格式显示变量。

t 按二进制格式显示变量。

a 按十六进制格式显示变量。

c 按字符格式显示变量。

f 按浮点数格式显示变量。

\*如果a是一个动态数组，10个元素，如果要显示则：

(gdb) print \*a@10

这样，会显示10个元素，无论a是double或者是int的都会正确地显示10个元素。 静态数组可以直接用print数组名，就可以显示数组中所有数据的内容了

\*修改运行时候的变量值：

(gdb) print x=4

这里，x=4是C/C++的语法，意为把变量x值改为4，如果你当前调试的语言是Pascal，那么你可以使用Pascal的语法：x:=4。

\*显示一个变量var的类型：

(gdb) whatis var

察看寄存器

要查看寄存器的值，很简单，可以使用如下命令：

info registers

查看寄存器的情况。（除了浮点寄存器）

info all-registers

查看所有寄存器的情况。（包括浮点寄存器）

info registers

查看所指定的寄存器的情况。

寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址（ip），程序的当前堆栈地址（sp）等等。你同样可以使用print命令来访问寄存器的情况，只需要在寄存器名字前加一个$符号就可以了。如：p $eip。

print 操作符

@是一个和数组有关的操作符，在后面会有更详细的说明。

:: 指定一个在文件或是一个函数中的变量。

{} 表示一个指向内存地址的类型为type的一个对象。

print 设置

set print address

set print address on

打开地址输出，当程序显示函数信息时，GDB会显出函数的参数地址。

set print arrayset print array on

打开数组显示，打开后当数组显示时，每个元素占一行，如果不打开的话，每个元素则以逗号分隔。

set print elements

这个选项主要是设置数组的，如果你的数组太大了，那么就可以指定一个来指定数据显示的最大长度，当到达这个长度时，GDB就不再往下显示了。如果设置为0，则表示不限制。

set print null-stop

如果打开了这个选项，那么当显示字符串时，遇到结束符则停止显示。这个选项默认为off。

set print pretty on

如果打开printf pretty这个选项，那么当GDB显示结构体时会比较漂亮。

set print union

设置显示结构体时，是否显式其内的联合体数据。

set print object

在C++中，如果一个对象指针指向其派生类，如果打开这个选项，GDB会自动按照虚方法调用的规则显示输出，如果关闭这个选项的话，GDB就不管虚函数表了。

\*以更详细的方式显示变量var的类型：

(gdb) ptype var

这里，会打印出var的结构定义。

按 Tab 键补齐命令,用光标键上下翻动历史命令. 用 help up 看帮助。

gdb 应用举例:

下面列出了将被调试的程序：它显示一个简单的问候, 再用反序将它列出

main**.**cpp**:**

void MyPrint**(**const char **\***pszSrc**);**

void MyPrint2**(**const char **\***pszSrc**);**

int main **()**

**{**

char szSrc**[]** **=** "hello there"**;**

MyPrint**(**szSrc**);**

MyPrint2**(**szSrc**);**

**}**

func**.**cpp

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

void MyPrint**(**const char **\***pszSrc**)**

**{**

printf**(**"The string is %s\n"**,** pszSrc**);**

**}**

void MyPrint2**(**const char **\***pszSrc**)**

**{**

char **\***pszRev**;**

int i**,**iLen**;**

iLen**=**strlen**(**pszSrc**);**

pszRev**=(**char **\*)**malloc**(**iLen**+**1**);**

**for(**i**=**0**;**i**<**iLen**;**i**++)**

pszRev**[**i**]=**pszSrc**[**iLen**-**i**];** //经过调试，发现此处应为：[iLen-i-1]

pszRev**[**iLen**]=**'\0'**;**

printf**(**"The revert string is:%s\n"**,**pszRev**);**

free**(**pszRev**);**

**}**

用下面的命令编译它(注意加上-g 的编译选项):

法1：

g++ -c main.cpp –g

g++ -c func.cpp –g

g++ -o main main.o func.o //或者g++ main.o func.o –o main

gdb main //如果不用gdb main，则可以输入gdb进入调试之后再输入(gdb) file main

(gdb) l[ist] func.cpp:1 //列出源代码: 技巧: 在 gdb 提示符下按回车健将重复上一个命令.

(gdb) break 17 //b

(gdb) run //r

(gdb) watch pszRev[i] //w

(gdb) next //n

(gdb) info b //[查看所有断点信息]

(gdb) continue //c，[继续全速运行]

(gdb) p i //查看i的值

(gdb) p iLen //打印iLen的值

**小知识：**

（1）在linux 中利用 system(“clear”);实现类似于 windows 里面的清屏函数 system(“cls”);

（2）LINUX 中可以通过下面的方式可以实现 system("pause");功能：

printf(“Press any key to continue…”);

getchar();

getchar(); //要用两个 getchar()函数

（3）limux 中利用 getchar()函数刷新输入缓冲区，利用fflush(stdout)刷新输出缓冲区

（4）命令 x 是用来检查内存情况，英文是 examine 含义，使用方法x /20xb 变量首地址，其中 20x 代表 16 进制的长度，b 代表字节的含义

（5）ulimint 用来显示当前的各种用户进程限制

针对段错误，可以通过 ulimit -c unlimited 设置 core file size 为不限制大小，设置完毕后，可以通过 ulimit -a 进行查看是否设置 ok，这时候再次运行程序，会产生 core 文件，通过 gdb 可执行程序 core 文件进行调试。直接通过 bt 可以看到程序段错误时的现场，通过 f 1 可以直接切换到程序现场。如：gdb ./test2 core

（6）调试正在运行的程序，通过 attach 进程 ID，调试正在运行的程序

# ulimit

## 1 ulimit介绍

ulimit 是一种 linux 系统的内键功能，它具有一套参数集，用于为由它生成的 shell 进程及其子进程的资源使用设置限制。ulimit 用于限制 shell 启动进程所占用的资源，支持以下各种类型的限制：所创建的内核文件的大小、进程数据块的大小、Shell 进程创建文件的大小、内存锁住的大小、常驻内存集的大小、打开文件描述符的数量、分配堆栈的最大大小、CPU 时间、单个用户的最大线程数、Shell 进程所能使用的最大虚拟内存。同时，它支持硬资源和软资源的限制。作为临时限制，ulimit 可以作用于通过使用其命令登录的 shell 会话，在会话终止时便结束限制，并不影响于其他 shell 会话。而对于长期的固定限制，ulimit 命令语句又可以被添加到由登录 shell 读取的文件中，作用于特定的 shell 用户。

## 2 ulimint的有效作用范围

ulimit 作为对资源使用限制的一种工作，是有其作用范围的。那么，它限制的对象是单个用户，单个进程，还是整个系统呢？事实上，ulimit 限制的是当前 shell 进程以及其派生的子进程。举例来说，如果用户同时运行了两个 shell 终端进程，只在其中一个环境中执行了 ulimit – s 100，则该 shell 进程里创建文件的大小收到相应的限制，而同时另一个 shell 终端包括其上运行的子程序都不会受其影响。

## 3 ulimint使用方式

ulimit 命令的格式为：

ulimit [-aHS] [options] [limit]

参数：

-a 显示目前资源限制的设定。

-c <core文件上限> 　设定core文件的最大值，单位为区块。

-d <数据节区大小> 　程序数据节区的最大值，单位为KB。

-f <文件大小> 　 shell所能建立的最大文件，单位为区块。

-H 　 设定资源的硬性限制，也就是管理员所设下的限制。

-m <内存大小> 　 指定可使用内存的上限，单位为KB。

-n <文件数目> 　 指定同一时间最多可开启的文件数。

-p <缓冲区大小> 　 指定管道缓冲区的大小，单位512字节。

-s <堆叠大小> 　 指定堆叠的上限，单位为KB。

-S 　 设定资源的弹性限制。

-t <CPU时间> 　 指定CPU使用时间的上限，单位为秒。

-u <程序数目> 　 用户最多可开启的程序数目。

-v <虚拟内存大小> 　指定可使用的虚拟内存上限，单位为KB。

查看限制情况：

ulimit -a

可以看到如下信息：

core file size (blocks, -c) 0

data seg size (kbytes, -d) unlimited

file size (blocks, -f) unlimited

pending signals (-i) 1024

max locked memory (kbytes, -l) 32

max memory size (kbytes, -m) unlimited

open files (-n) 1024

pipe size (512 bytes, -p) 8

POSIX message queues (bytes, -q) 819200

stack size (kbytes, -s) 10240

cpu time (seconds, -t) unlimited

max user processes (-u) 4096

virtual memory (kbytes, -v) unlimited

file locks (-x) unlimited

## 4 通过ulimit保存程序错误现场

当系统中的一些程序在遇到一些错误以及crash时，系统会自动产生core文件记录crash时刻系统信息，包括内存和寄存器信息，用以程序员日 后debug时可以使用。这些错误包括段错误、非法指令、总线错误或用户自己生成的退出信息等等，一般地，core文件在当前文件夹中存放。

core文件有时可能在你发生错误时，并没有出现在你当前的文件夹中，发生这种情况的原因有两个：一个是当前终端被设置为不能弹出core文件；另一种则是core文件被指定了路径。

对于前者，我们可以使用ulimit 命令对core文件的大小进行设定。一般默认情况下，core文件的大小被设置为0，这样系统就不dump出core文件了。这时，使用命令：ulimit -c unlimited 进行设置，就可以把core文件的大小设置为无限大，同时也可以使用数字来替代unlimited，对core文件的上限制做更精确的设定。

针对断错误，可以通过 ulimit -c unlimited 设置 core file size 为不限制大小，设置完毕后，可以通过 ulimit -a 进行查看是否设置 ok，这时候再次运行程序，会产生 core 文件，通过 gdb 可执行程序 core 文件进行调试。直接通过 bt 可以看到程序段错误时的现场，通过 f 1可以直接切换到程序现场。如：gdb ./test2 core

## 5 设置（内核转存core dump）文件目录和命名规则

在默认的情况下，很多系统的core文件是生成在你运行程序的目录下，或者你在程序中chdir后的那个目录，然后在core文件的后面加了一个 pid。在实际工作中，这样可能会造成很多目录下产生core文件，不便于管理，实际上除了可以设置core文件的大小之外，还可以对core文件的名称进行一些规定。这种设置是对/proc/sys/kernel/core\_pattern和/proc/sys/kernel/core\_uses\_pid这两个文件进行修改。

/proc/sys/kernel/core\_uses\_pid可以控制产生的core文件的文件名中是否添加pid作为扩展，如果添加则文件内容为1，否则为0

proc/sys/kernel/core\_pattern可以设置格式化的core文件保存位置或文件名。

改动这两个文件的方法如下：

echo <pattern> > /proc/sys/kernel/core\_pattern

echo <"0"/"1"> /proc/sys/kernel/core\_uses\_pid

并且注意，只有超级用户才可以修改这两个表。

core\_pattern接受的是core文件名称的pattern，它包含任何字符串，并且用%作为转移符号生成一些标示符，为core文件名称加入特殊含义。已定义的标示符有如下这些：

%%：相当于%

%p：相当于<pid>，添加pid

%u：相当于<uid>，添加当前uid

%g：相当于<gid>，添加当前gid

%s：相当于导致dump的信号的数字，添加导致产生core的信号

%t：相当于dump的时间，添加core文件生成时的unix时间

%e：相当于执行文件的名称，添加命令名

%h：相当于hostname，添加主机名

除以上这些标志位外，还规定：

1）末尾的单个%可以直接去除；

2）%加上除上述以外的任何字符，%和该字符都会被去除；

3）所有其他字符都作为一般字符加入名称中；

4）core文件的名称最大值为64个字节（包括'/0'）；

5）core\_pattern中默认的pattern为core；

6）为了保持兼容性，通过设置core\_uses\_pid，可以在core文件的末尾加上%p；

7）pattern中可以包含路径信息

比如原来文件内容是core-%e，可以这样修改:

echo "/tmp/core-%e-%p" > core\_pattern

将会控制所产生的core文件会存放到/corefile目录下，产生的文件名为core-命令名-pid-时间戳。

当然，你可以用下列方式来完成

sysctl -w kernel.core\_pattern=/tmp/core-%e-%p

这些操作一旦计算机重启，则会丢失，如果你想持久化这些操作，可以在 /etc/sysctl.conf文件中增加：

kernel.core\_pattern=/tmp/core%p

加好后，如果你想不重启看看效果的话，则用下面的命令：

sysctl -p /etc/sysctl.conf

# 