# shell执行方式

1. 作为可执行程序

chmod +x ./test.sh

./test.sh

1. 作为解释器参数

/bin/bash test.sh

这种运行方式是，直接运行解释器，其参数就是shell脚本的文件名。

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

# shell 变量

1. 数组操作与字符串的操作很重要。
2. shell的数据类型其实就两种，字符串与数字。

# shell语句的分隔符与分界符

1. 回车符、“;”作为作为命令语句的分隔符。
2. 空格则为一个语句的分隔符。

# inux后台运行和关闭、查看后台任务

1. **& 命令**

加在一个命令的最后，可以把这个命令放到后台执行。

如：**watch -n 10 sh test.sh & #每10s在后台执行一次test.sh脚本**

1. **ctrl + z 命令**

可以将一个正在前台执行的命令放到后台，并且处于暂停状态。

1. **jobs 命令**

查看当前有多少在后台运行的命令，jobs -l选项可显示所有任务的PID，jobs的状态可以是running, stopped, Terminated。但是如果任务被终止了（kill），shell 从当前的shell环境已知的列表中删除任务的进程标识。

1. **fg 命令**

将后台中的命令调至**前台**继续运行。如果后台中有多个命令，可以用fg %jobnumber（是命令编号，不是进程号）将选中的命令调出。

http://images.cnitblog.com/blog/408927/201409/182233000819052.png

1. **bg 命令**

将一个在后台暂停的命令，变成在**后台**继续执行。如果后台中有多个命令，可以用bg %jobnumber将选中的命令调出。

1. **kill 命令**

法子1：通过jobs命令查看job号（假设为num），然后执行kill %num

法子2：通过ps命令查看job的进程号（PID，假设为pid），然后执行kill pid前台进程的终止：Ctrl+c

1. **nohup 命令**

如果让程序始终在后台执行，即使关闭当前的终端也执行（之前的&做不到）。这时候需要nohup，该命令可以在你退出帐户/关闭终端之后继续运行相应的进程。关闭终端后，在另一个终端jobs已经无法看到后台跑得程序了，此时利用ps（进程查看命令）

**ps -aux | grep "test.sh" # a:显示所有程序 u:以用户为主的格式来显示 x:显示所有程序，不以终端机来区分**

# shell加深理解

原来ls、cd、| 、> 等都是shell命令

# shell运算符

说到shell运算符，确实有点多了，分的也比较细致，要求也挺多，主要原因在于shell本身就是解释性语言，需要分隔符与分界符，解析程序的时候需要。

运算符分数字类型与字符串类型：

**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 ]。**

1. 数字运算符：只能运用在数字类型上，除非字符串也是数字类型，否则不可以用在字符串上。
   1. 算数运算符：

两点注意：

#!/bin/bash

val=`expr 2 + 2`

echo "两数之和为 : $val"

原生bash不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。expr 是一款表达式计算工具，使用它能完成表达式的求值操作。

表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样。完整的表达式要被 ` ` 包含，注意这个字符不是常用的单引号，在 Esc 键下边。

* 1. 关系运算符：
  2. 布尔运算符：

1. 字符串类型：
2. 文件测试运算符：

# 流程控制

在sh/bash里可不能这么写，如果else分支没有语句执行，就不要写这个else。

if语句语法格式：

if condition

then

command1

command2

……

commandn

fi

# test命令

test命令与条件表达式的[ \*\*\*\* ] 性质一样，在用流程控制语句上时。

如：实例1与实例2一样。

实例1：

if [ $a != $b ]

then

echo "a 不等于 b"

fi

实例2：

if test $a != $b ]

then

echo "a 不等于 b"

fi

# shell函数

linux shell 可以用户定义函数，然后在shell脚本中可以随便调用。

shell中函数的定义格式如下：

[ function ] funname [()]

{

action;

[return int;]

}

说明：

1. 可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。
2. 参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。 return后跟数值n(0-255)。
3. 函数返回值在调用该函数后通过 $? 来获得。
4. 注意：所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至shell解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可。
5. 函数参数，在Shell中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1表示第一个参数，$2表示第二个参数...
6. 注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当n>=10时，需要使用${n}来获取参数。
7. 另外，还有几个特殊字符用来处理参数：



# shell 输入输出重定向

1. 输入重定向：
2. 输出重定向：
3. 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：

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

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

例子：

**[root@redhat box]# ls a.txt b.txt 1>file.out 2>file.err**

执行后，有任何返回值，原因是，返回值都重定向到相应的文件中了，而不再前端显示。

[root@redhat box]# cat file.out

a.txt

[root@redhat box]# cat file.err

ls: b.txt: No such file or directory

一般来说，"1>" 通常可以省略成 ">"。

**即可以把如上命令写成： ls a.txt b.txt >file.out 2>file.err。**

有了这些认识才能理解 "1>&2" 和 "2>&1"。

1>&2 正确返回值传递给2输出通道 &2表示2输出通道。

如果此处错写成 1>2, 就表示把1输出重定向到文件2中。

2>&1 错误返回值传递给1输出通道, 同样&1表示1输出通道。

& 是一个描述符，如果1或2前不加&，会被当成一个普通文件。

1>&2 意思是把标准输出重定向到标准错误

2>&1 意思是把标准错误输出重定向到标准输出。

**&>filename 意思是把标准输出和标准错误输出都重定向到文件filename中**

# /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 #省略了1

其实等价于：

command 1 > /dev/null 2>&1

# 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**

**注：**被包含的文件 filename 不需要可执行权限。