1.1 shell 概述 1
1.1.1 几种常见的 Shell 2
1.1.2 Shell 脚本语言与编译型语言的差异 2
1.1.3 Shell 3
1.1.4 第一个 Shell 脚本 3
1.1.5 Shell 变量：Shell 变量的定义、删除变量、只读变量、变量类型 4
1.1.6 Shell 特殊变量：Shell $0, $#, $*, $@, $?, $$和命令行参数 6
1.1.7 Shell 运算符 7
1.1.8 文件测试运算符 14
1.1.9 Shell 注释 16
1.1.10 Shell 字符串 17
1.1.11 Shell 数组：shell 数组的定义、数组长度 18
1.1.12 Shell echo 命令 19
1.1.13 shell printf 命令：格式化输出语句 21
1.1.14 Shell if else 语句 22
1.1.15 Shell case esac 语句 25
1.1.16 Shell for 循环 26
1.1.17 Shell while 循环 28
1.1.18 Shell until 循环 29
1.1.19 Shell break 和 continue 命令 30
1.1.20 Shell 函数：Shell 函数返回值、删除函数、在终端调用函数 32
1.1.21 Shell 函数参数 35
1.1.22 Shell 输入输出重定向：Shell Here Document，/dev/null 文件 36
1.1.23 重定向深入讲解 37
1.1.24 Shell 文件包含 39

内容参考网络整理

1.1 shell 概述

Linux 命令在线查看网站：http://man.linuxde.net/

Shell 本身是一个用 C 语言编写的程序，它是用户使用 Unix/Linux 的桥梁，用户的大部分工作都是通过 Shell 完成的。Shell 既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。

它虽然不是 Unix/Linux 系统内核的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，shell 是最重要的实

用程序，深入了解和熟练掌握 shell 的特性极其使用方法，是用好 Unix/Linux 系统的关键。

可以说，shell 使用的熟练程度反映了用户对 Unix/Linux 使用的熟练程度。

注意：单独地学习 Shell 是没有意义的，请先参考 Unix/Linux 入门教程，了解 Unix/Linux 基础。

Shell 有两种执行命令的方式：
交互式（Interactive）：解释执行用户的命令，用户输入一条命令，Shell 就解释执行一条。
批处理（Batch）：用户事先写一个 Shell 脚本(Script)，其中有很多条命令，让 Shell 一次把这些命令执行完，而不必一条一条地敲命令。

Shell 脚本和编程语言很相似，也有变量和流程控制语句，但 Shell 脚本是解释执行的，不需要编译，Shell 程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到 Shell 提示符下执行。

Shell 初学者请注意，在平常应用中，建议不要用 root 帐号运行 Shell 。作为普通用户，不管您有意还是无意，都无法破坏系统；但如果是 root，那就不同了，只要敲几个字母，就可能导致灾难性后果。

1.1.1 几种常见的 Shell

上面提到过，Shell 是一种脚本语言，那么，就必须有解释器来执行这些脚本。

查看当前系统默认的 shell:

```
[xiao@localhost file_2]$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 05-25 19:24 /bin/sh -> bash
```

Unix/Linux 上常见的 Shell 脚本解释器有 bash、sh、csh、ksh 等，习惯上把它们称作一种 Shell。我们常说有多少种 Shell，其实说的是 Shell 脚本解释器。
bash
bash 是 Linux 标准默认的 shell，教程也基于 bash 讲解。

1.1.2 Shell 脚本语言与编译型语言的差异

大体上，可以将程序设计语言可以分为两类：编译型语言和解释型语言。
编译型语言
很多传统的程序设计语言，例如 Fortran、Ada、Pascal、C、C++和 Java，都是编译型语言。这类语言需要预先将我们写好的源代码(source code)转换成目标代码(object code)，这个过程被称作“编译”。

运行程序时，直接读取目标代码(object code)。由于编译后的目标代码(object code)非常接近计算机底层，因此执行效率很高，这是编译型语言的优点。

但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。例如，在 C++里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

解释型语言

解释型语言也被称作“脚本语言”。执行这类程序时，解释器(interpreter)需要读取我们编写的源代码(source code)，并将其转换成目标代码(object code)，再由计算机运行。因为每次执行程序都多了编译的过程，因此效率有所下降。

使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象；缺点是它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用 C 或 C++来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有 awk、Perl、Python、Ruby 与 Shell。

1.1.3 Shell

因为 Shell 似乎是各 UNIX 系统之间通用的功能，并且经过了 POSIX 的标准化。因此，Shell 脚本只要“用心写”一次，即可应用到很多系统上。因此，之所以要使用 Shell 脚本是基于：
简单性：Shell 是一个高级语言；通过它，你可以简洁地表达复杂的操作。
可移植性：使用 POSIX 所定义的功能，可以做到脚本无须修改就可在不同的系统上执行。
开发容易：可以在短时间内完成一个功能强大又好用的脚本。

1.1.4 第一个 Shell 脚本

打开文本编辑器，新建一个文件，扩展名为 sh（sh 代表 shell），扩展名并不影响脚本执行，见名知意就好。

输入一些代码：

```
#!/bin/bash
echo "Hello World !"
```

“#!”是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，使用哪一种 Shell。echo 命令用于向窗口输出文本。

运行 Shell 脚本有两种方法。
作为可执行程序
将上面的代码保存为 test.sh，并 cd 到相应目录：

```
chmod 777 test.sh  #使脚本具有执行权限
```

```
./test.sh  #执行脚本
```

注意，一定要写成./test.sh，而不是 test.sh。运行其它二进制的程序也一样，直接写 test.sh，linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有/bin, /sbin, /usr/bin, /usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用./test.sh 告诉系统说，就在当前目录找。

系统没有指定脚本的情况下，通过这种方式运行 bash 脚本，第一行一定要写对，好让系统查找到正确的解释器。
如果系统一开始就指定了默认的 shell 脚本，第一行指定解析器的代码就可以省掉。

```
[xiao@localhost file_2]$ ls -l /bin/sh      //查看系统默认的脚本解析器
```

当提示命令解析找不到的时候可以在运行时，加上解析器：

```
[xiao@localhost file_2]$ bash bash_shell.sh
```

1.1.5Shell 变量：Shell 变量的定义、删除变量、只读变量、变量类型

Shell 支持自定义变量。

定义变量

定义变量时，变量名不加美元符号（$），如：

```
ABC="123"
```

注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。同时，变量名的命名须遵循如下规则：
首个字符必须为字母（a-z，A-Z）。
中间不能有空格，可以使用下划线（_）。
不能使用标点符号。
不能使用 bash 里的关键字（可用 help 命令查看保留关键字）。

变量定义举例：

```
A="ABCD "
B=100
```

使用变量

使用一个定义过的变量，只要在变量名前面加美元符号（$）即可，如：

```
A ="123"
echo $A
echo ${A}
```

变量名外面的花括号是可选的，加不加都行，加花括号是为了帮助解释器识别变量的边界，

推荐给所有变量加上花括号，这是个好的编程习惯。

重新定义变量

已定义的变量，可以被重新定义，如：

```
A="123"
echo ${A}

A="456"
echo ${A}
```

这样写是合法的，但注意，第二次赋值的时候不能写成 $A= “123”
---使用变量的时候才加美元符（$）。

只读变量

使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。

下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash
A="123"
readonly A  #声明只读变量
A="345"
```

运行脚本，结果如下：
./bash_shell.sh: line 3: A: readonly variable

删除变量

使用 unset 命令可以删除变量。语法：
unset variable_name
变量被删除后不能再次使用；unset 命令不能删除只读变量。

举个例子：

```
#!/bin/sh
A="1234"
unset A
echo $A
```

上面的脚本没有任何输出。

变量类型

运行 shell 时，会同时存在三种变量：

1）局部变量

局部变量在脚本或命令中定义，仅在当前 shell 实例中有效，其他 shell 启动的程序不能访问局部变量。

2）环境变量

所有的程序，包括 shell 启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候 shell 脚本也可以定义环境变量。

3）shell 变量

shell 变量是由 shell 程序设置的特殊变量。shell 变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了 shell 的正常运行。

1.1.6 Shell 特殊变量：Shell $0, $#, $*, $@, $?, $$和命令行参数

前面已经讲到，变量名只能包含数字、字母和下划线，因为某些包含其他字符的变量有特殊含义，这样的变量被称为特殊变量。

例如，$ 表示当前 Shell 进程的 ID，即 pid，看下面的代码：

```
$echo $$
```

运行结果

29888

特殊变量列表

变量	含义
$0	当前脚本的文件名
$n	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。
$?	上个命令的退出状态，或函数的返回值。$? 也可以表示函数的返回值。
$$	当前 Shell 进程 ID。对于 Shell 脚本，就是这些脚本所在的进程 ID。

命令行参数

运行脚本时传递给脚本的参数称为命令行参数。命令行参数用 $n 表示，例如，$1 表示第一个参数，$2 表示第二个参数，依次类推。

退出状态

$? 可以获取上一个命令的退出状态。所谓退出状态，就是上一个命令执行后的返回结果。退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。

不过，也有一些命令返回其他值，表示不同类型的错误。

程序示例：

```
#!/bin/bash
echo "脚本文件名称：$0"
echo "第一个参数：$1"
echo "第二个参数：$2"
echo "传进来参数总数量：$# 个"
echo "传进来的所有参数：$*"
echo "传进来的所有参数：$@"
echo "命令执行的状态：$?"
echo "当前脚本的进程 ID 号：$$"
```

运行：

```
[root@localhost file_1]# ./bash_shell.sh 12 13 14
脚本文件名称：./bash_shell.sh
第一个参数：12
第二个参数：13
传进来参数总数量：3 个
传进来的所有参数：12 13 14
传进来的所有参数：12 13 14
命令执行的状态：0
当前脚本的进程 ID 号：19331
[root@localhost file_1]#
```

1.1.7 Shell 运算符

Bash 支持很多运算符，包括算数运算符、关系运算符、布尔运算符、字符串运算符和文件测试运算符。
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。

例如，两个数相加：

```
#!/bin/bash
A=`expr 2 + 2`
echo "sum = $A"
```

运行脚本输出：

```
SUM= 4
```

注意：

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。

算术运算符

先来看一个使用算术运算符的例子：

```
#!/bin/sh

a=10
b=20
val=`expr $a + $b`     #加法运算
echo "a + b : $val"

val=`expr $a - $b`    #减法运算
echo "a - b : $val"

val=`expr $a \* $b`   #乘法运算
echo "a * b : $val"

val=`expr $b / $a`    #除法运算
echo "b / a : $val"

val=`expr $b % $a`   #取余运算
echo "b % a : $val"

if [ $a == $b ]      #判断变量 a 和 b 是否相等
then
   echo "a is equal to b"  #相等
fi

if [ $a != $b ]     #判断变量 a 和 b 是否相等
then
   echo "a is not equal to b"  #不相等
fi
```

运行结果：

```
a + b : 30
a - b : -10
a * b : 200
```

```
b / a : 2
b % a : 0
a is not equal to b
```

注意：
乘号(*)前边必须加反斜杠(\)才能实现乘法运算；
if...then...fi 是条件语句，后续将会讲解。

算术运算符列表

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：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]。

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。

先来看一个关系运算符的例子：

```
#!/bin/sh

a=10
b=20
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi
```

```
if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

运行结果：

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

关系运算符列表

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true。（正确--1）	[$a -eq $b] 返回 true。
-ne	检测两个数是否相等，不相等返回 true。(1)	[$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 $b] 返回 false。(错误 0)
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	[$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。

布尔运算符

先来看一个布尔运算符的例子：

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
```

```
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

运行结果：

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

布尔运算符列表

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 $b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a $b -gt 100] 返回 false。

字符串运算符

先来看一个例子：

```
#!/bin/sh
a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi
```

```
if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

运行结果：

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

字符串运算符列表

运算符	说明	举例
=	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	检测字符串长度是否为 0，为 0 返回 true。	[-z $a] 返回 false。
-n	检测字符串长度是否为 0，不为 0 返回 true。	[-z $a] 返回 true。
str	检测字符串是否为空，不为空返回 true。	[$a] 返回 true。

1.1.8 文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。
例如，变量 file 表示文件“/var/www/tutorialspoint/unix/test.sh”，它的大小为 100 字节，具有 rwx 权限。下面的代码，将检测该文件的各种属性：

```
#!/bin/sh

file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]
then
   echo "File has read access"
else
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

if [ -f $file ]
then
   echo "File is an ordinary file"
else
```

```
   echo "This is sepcial file"
fi

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi

if [ -s $file ]
then
   echo "File size is zero"
else
   echo "File size is not zero"
fi

if [ -e $file ]
then
   echo "File exists"
else
   echo "File does not exist"
fi
```

运行结果：

```
File has read access
File has write permission
File has execute permission
File is an ordinary file
This is not a directory
File size is zero
File exists
```

文件测试运算符列表

操作符	说明	举例
-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	检测文件是否是字符设备文件，如果是，则返回 true。	[-b $file] 返

file		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。
-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。	[-f $file] 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	检测文件是否是具名管道，如果是，则返回 true。	[-p $file] 返回 false。
-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。
-r file	检测文件是否可读，如果是，则返回 true。	[-r $file] 返回 true。
-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。
-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。
-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。

1.1.9 Shell 注释

以“#”开头的行就是注释，会被解释器忽略。

sh 里没有多行注释，只能每一行加一个#号

如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？每一行加个#符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

1.1.10 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。

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）。

双引号

```
your_name='大侠'
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

拼接字符串

```
your_name="大侠"
greeting="hello, "$your_name" !"
greeting_1="hello, ${your_name} !"

echo $greeting $greeting_1
```

获取字符串长度

```
string="abcd"
echo ${#string}   #输出 4
```

提取子字符串

```
string="123456789"
echo ${string:1:4} #输出 2345
```

查找子字符串

```
string="123ABC567 "
echo `expr index "$string" ABC`  #输出 1
ABC 是要查找的字符串，$string：是存放字符串的变量
```

1.1.11 Shell 数组：shell 数组的定义、数组长度

Shell 在编程方面比 Windows 批处理强大很多，无论是在循环、运算。

bash 支持一维数组（不支持多维数组），并且没有限定数组的大小。类似与 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于 0。

定义数组

在 Shell 中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

```
array_name=(value1 ... valuen)
```

例如：

```
array_name=(value0 value1 value2 value3)
```

或者

```
array_name=(
value0
value1
value2
value3
)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

可以不使用连续的下标，而且下标的范围没有限制。

读取数组

读取数组元素值的一般格式是：

```
${array_name[index]}
```

例如：

```
valuen=${array_name[2]}
```

举个例子：

```
#!/bin/sh

NAME[0]="12"
NAME[1]="13"
NAME[2]="14"
NAME[3]="15"
NAME[4]="16"
```

```
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

运行脚本，输出：

```
$./test.sh
First Index: 12
Second Index: 13
```

使用@ 或 * 可以获取数组中的所有元素，例如：

```
${array_name[*]}
${array_name[@]}
```

举个例子：

```
#!/bin/sh
BUFF[0]="1"
BUFF[1]="2"
BUFF[2]="3"
BUFF[3]="4"
BUFF[4]="5"
echo "One: ${BUFF[*]}"
echo "Two: ${BUFF[@]}"
```

运行脚本 输出：

```
[root@localhost file_1]# ./bash_shell.sh
One: 1 2 3 4 5
Two: 1 2 3 4 5
```

获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

```
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

1.1.12 Shell echo 命令

echo 是 Shell 的一个内部指令，用于在屏幕上打印出指定的字符串。命令格式：

```
echo arg
```

您可以使用 echo 实现更复杂的输出格式控制。

显示转义字符

```
echo "\"It is a test\""
结果将是：
"It is a test"
```

双引号也可以省略。

显示变量

```
name="OK"
echo "$name It is a test"

结果将是：
OK It is a test
```

同样双引号也可以省略。

如果变量与其它字符相连的话，需要使用大括号（{ }）：

```
mouth=8
echo "${mouth}-1-2009"
结果将是：
8-1-2009
```

显示换行

```
echo "OK!\n"
echo "It is a test"
输出：
OK!
It is a test
```

显示不换行

```
echo "OK!\c"
echo "It is a test"
输出：
OK!It si a test
```

显示结果重定向至文件

```
echo "It is a test" > myfile    # 如果 myfile 这个文件没有，会自动创建，每次写入会覆盖之前的内容。
```

1.1.13 shell printf 命令：格式化输出语句

printf 命令用于格式化输出， 是 echo 命令的增强版。它是 C 语言 printf()库函数的一个有限的变形，并且在语法上有些不同。

注意：printf 由 POSIX 标准所定义，移植性要比 echo 好。
如同 echo 命令，printf 命令也可以输出简单的字符串：

```
$printf "Hello, Shell\n"
Hello, Shell
```

printf 不像 echo 那样会自动换行，必须显式添加换行符(\n)。

printf 命令的语法：
printf format-string [arguments...]
format-string 为格式控制字符串，arguments 为参数列表。
示例：printf "%s\n" "1234"

printf()在 C 语言入门教程中已经讲到，功能和用法与 printf 命令类似，请查看：C 语言格式输出函数 printf()详解

这里仅说明与 C 语言 printf()函数的不同：

- printf 命令不用加括号
- format-string 可以没有引号，但最好加上，单引号双引号均可。
- 参数多于格式控制符(%)时，format-string 可以重用，可以将所有参数都转换。
- arguments 使用空格分隔，不用逗号。

请看下面的例子：

```
# format-string 为双引号
$ printf "%d %s\n" 1 "abc"
1 abc
# 单引号与双引号效果一样
$ printf '%d %s\n' 1 "abc"
1 abc
# 没有引号也可以输出
$ printf %s abcdef
abcdef
# 格式只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用
```

```
$ printf %s abc def
abcdef
$ printf "%s\n" abc def
abc
def
$ printf "%s %s %s\n" a b c d e f g h i j
a b c
d e f
g h i
j
# 如果没有 arguments，那么 %s 用 NULL 代替，%d 用 0 代替
$ printf "%s and %d \n"
and 0
# 如果以 %d 的格式来显示字符串，那么会有警告，提示无效的数字，此时默认置为 0
$ printf "The first program always prints'%s,%d\n'" Hello Shell
-bash: printf: Shell: invalid number
The first program always prints 'Hello,0'
$
```

注意，根据 POSIX 标准，浮点格式%e、%E、%f、%g 与%G 是“不需要被支持”。这是因为 awk 支持浮点预算，且有它自己的 printf 语句。这样 Shell 程序中需要将浮点数值进行格式化的打印时，可使用小型的 awk 程序实现。然而，内建于 bash、ksh93 和 zsh 中的 printf 命令都支持浮点格式。

1.1.14 Shell if else 语句

if 语句通过关系运算符判断表达式的真假来决定执行哪个分支。Shell 有三种 if ... else 语句：
if ... fi 语句；
if ... else ... fi 语句；
if ... elif ... else ... fi 语句。

1) if ... else 语句

```
if ... else 语句的语法：
if [ 表达式 ]
then
   表达式为真，就执行语句。
fi
```

如果 表达式 返回 true，then 后边的语句将会被执行；如果返回 false，不会执行任何语句。

最后必须以 fi 来结尾闭合 if，fi 就是 if 倒过来拼写，后面也会遇见。
注意：表达式和方括号([ ])之间必须有空格，否则会有语法错误。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "相等"
fi

if [ $a != $b ]
then
   echo "不相等"
fi
```

运行结果：
不相等

2) if ... else ... fi 语句

if ... else ... fi 语句的语法：

```
if [ 表达式 ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

如果表达式返回 true，那么 then 后边的语句将会被执行；否则，执行 else 后边的语句。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
```

```
    echo "相等"
else
    echo "不相等"
fi
```

执行结果：

```
不相等
```

3) if ... elif ... fi 语句

if ... elif ... fi 语句可以对多个条件进行判断，语法为：

```
if [ 表达式 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ 表达式 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ 表达式 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

哪一个 表达式 的值为 true，就执行哪个 表达式 后面的语句；如果都为 false，那么不执行任何语句。

举个例子：

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
```

```
fi
```

运行结果：
a is less than b

if ... else 语句也可以写成一行，以命令的方式来运行，像这样：
```
if test $[2*3] -eq $[1+5]; then echo 'The two numbers are equal!'; fi;
```

if ... else 语句也经常与 test 命令结合使用，如下所示：

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
    echo 'The two numbers are equal!'
else
    echo 'The two numbers are not equal!'
fi
```

输出：
The two numbers are equal!
test 命令用于检查某个条件是否成立，与方括号([ ])类似。

1.1.15 Shell case esac 语句

case ... esac 与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构。

case 语句匹配一个值或一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：
```
case 值 in

模式 1)
    command1
    command2
    command3
    ;;
模式 2）
    command1
    command2
    command3
    ;;
*)
    command1
    command2
```

```
    command3
    ;;
esac
```

case 工作方式如上所示。取值后面必须为关键字 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 ;;。

;; 与其他语言中的 break 类似，意思是跳到整个 case 语句的最后。

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。

下面的脚本提示输入 1 到 4，与每一种模式进行匹配：

```
echo 'Input a number between 1 to 4'
echo 'Your number is:\c'
read aNum
case $aNum in
    1)  echo 'You select 1'
    ;;
    2)  echo 'You select 2'
    ;;
    3)  echo 'You select 3'
    ;;
    4)  echo 'You select 4'
    ;;
    *)  echo 'You do not select a number between 1 to 4'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
Input a number between 1 to 4
Your number is:3
You select 3
```

1.1.16 Shell for 循环

与其他编程语言类似，Shell 支持 for 循环。

for 循环一般格式为：

```
for 变量 in 列表
do
    command1
    command2
    ...
```

```
    commandN
done
```

列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量。

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。

例如，顺序输出当前列表中的数字：

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

运行结果：

```
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
```

顺序输出字符串中的字符：

```
for str in 'This is a string'
do
    echo $str
done
```

运行结果：

```
This is a string
```

输出数据与 C 语言 for 循环差不多

```
for((i=1;i<=10;i++))  或者  for i in $(seq 10)
do
    echo $i
done
```

运行结果：

```
[root@localhost file_1]# ./bash_shell.sh
1
2
3
4
5
```

6
7
8
9
10

显示主目录下以 .bash 开头的文件：

```
#!/bin/bash

for FILE in $HOME/.bash*
do
   echo $FILE
done
```

运行结果：
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc

1.1.17 Shell while 循环

while 循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。
其格式为：

```
while command
do
     Statement(s) to be executed if command is true
Done
```

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。
示例：
while 循环

```
i=0
while(($i<5))
do
echo $i
i=`expr $i + 1`    #每次 i 加 1
done
```

运行结果：

```
[root@localhost file_1]# ./bash_shell.sh
```

```
0
1
2
3
4
```

以下是一个基本的 while 循环，测试条件是：如果 COUNTER 小于 5，那么返回 true。COUNTER 从 0 开始，每次循环处理时，COUNTER 加 1。运行上述脚本，返回数字 1 到 5，然后终止。

```
COUNTER=0
while [ $COUNTER -lt 5 ]
do
    COUNTER='expr $COUNTER+1'
    echo $COUNTER
done
```

运行脚本，输出：

```
1
2
3
4
5
```

while 循环可用于读取键盘信息。下面的例子中，输入信息被设置为变量 FILM，按<Ctrl-D>结束循环。

```
echo 'type <CTRL-D> to terminate'
echo -n 'enter your most liked film: '
while read FILM
do
    echo "Yeah! great film the $FILM"
done
```

运行脚本，输出类似下面：

```
type <CTRL-D> to terminate
enter your most liked film: Sound of Music
Yeah! great film the Sound of Music
```

1.1.18 Shell until 循环

until 循环执行一系列命令直至条件为 true 时停止。until 循环与 while 循环在处理方式上刚好相反。一般 while 循环优于 until 循环，但在某些时候，也只是极少数情况下，until 循环更加有用。

until 循环格式为：

```
until command
do
```

```
    Statement(s) to be executed until command is true
done
```

command 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。

例如，使用 until 命令输出 0 ~ 9 的数字：

```
#!/bin/bash
a=0
until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

运行结果：

```
0
1
2
3
4
5
6
7
8
9
```

1.1.19 Shell break 和 continue 命令

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，像大多数编程语言一样，Shell 也使用 break 和 continue 来跳出循环。

break 命令

break 命令允许跳出所有循环（终止执行后面的所有循环）。

下面的例子中，脚本进入死循环直至用户输入数字大于 5。要跳出这个循环，返回到 shell 提示符下，就要使用 break 命令。

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
```

```
        *) echo "You do not select a number between 1 to 5, game is over!"
            break
        ;;
    esac
done
```

在嵌套循环中，break 命令后面还可以跟一个整数，表示跳出第几层循环。例如：
break n
表示跳出第 n 层循环。

下面是一个嵌套循环的例子，如果 var1 等于 2，并且 var2 等于 0，就跳出循环：

```
#!/bin/bash

for var1 in 1 2 3
do
   for var2 in 0 5
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

如上，break 2 表示直接跳出外层循环。运行结果：
1 0
1 5

continue 命令

continue 命令与 break 命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环。

对上面的例子进行修改：

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "Your number is $aNum!"
        ;;
```

```
        *) echo "You do not select a number between 1 to 5!"
            continue
            echo "Game is over!"
        ;;
    esac
done
```

运行代码发现，当输入大于 5 的数字时，该例中的循环不会结束，语句
echo "Game is over!"
永远不会被执行。

同样，continue 后面也可以跟一个数字，表示跳出第几层循环。

再看一个 continue 的例子：

```
#!/bin/bash

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
   Q=`expr $NUM % 2`
   if [ $Q -eq 0 ]
   then
      echo "Number is an even number!!"
      continue
   fi
   echo "Found odd number"
done
```

运行结果：

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd numbers
```

1.1.20 Shell 函数：Shell 函数返回值、删除函数、在终端调用函数

函数可以让我们将一个复杂功能划分成若干模块，让程序结构更加清晰，代码重复利用率更高。像其他编程语言一样，Shell 也支持函数。Shell 函数必须先定义后使用。

Shell 函数的定义格式如下：

```
function_name () {
    list of commands
```

```
    [ return value ]
}
```

如果你愿意，也可以在函数名前加上关键字 function：

```
function function_name () {
    list of commands
    [ return value ]
}
```

函数返回值，可以显式增加 return 语句；如果不加，会将最后一条命令运行结果作为返回值。

Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0 表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

如果一定要让函数返回字符串，那么可以先定义一个变量，用来接收函数的计算结果，脚本在需要的时候访问这个变量来获得函数返回值。

先来看一个例子：

```
#!/bin/bash

# Define your function here
Hello () {
   echo "1234567890"
}

# Invoke your function
Hello
```

运行结果：

```
$./test.sh
1234567890$
```

调用函数只需要给出函数名，不需要加括号。

再来看一个带有 return 语句的函数：

```
#!/bin/bash
funWithReturn(){
    echo "The function is to get the sum of two numbers..."
    echo -n "Input first number: "
    read aNum
    echo -n "Input another number: "
    read anotherNum
```

```
    echo "The two numbers are $aNum and $anotherNum !"
    return $(($aNum+$anotherNum))
}
funWithReturn
# Capture value returnd by last command
ret=$?
echo "The sum of two numbers is $ret !"
```

运行结果：

```
The function is to get the sum of two numbers...
Input first number: 25
Input another number: 50
The two numbers are 25 and 50 !
The sum of two numbers is 75 !
```

函数返回值在调用该函数后通过 $? 来获得。

再来看一个函数嵌套的例子：

```
#!/bin/bash

# Calling one function from another
number_one () {
    echo "1234567890"
    number_two
}

number_two () {
    echo "abcdefg"
}

number_one
number_two
```

运行结果：

```
1234567890
number_one
```

像删除变量一样，删除函数也可以使用 unset 命令，不过要加上 .f 选项，如下所示：

```
$unset .f function_name
```

如果你希望直接从终端调用函数，可以将函数定义在主目录下的 .profile 文件，这样每次登录后，在命令提示符后面输入函数名字就可以立即调用。

1.1.21 Shell 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...

带参数的函数示例：

```
#!/bin/bash
funWithParam(){
    echo "The value of the first parameter is $1 !"
    echo "The value of the second parameter is $2 !"
    echo "The value of the tenth parameter is $10 !"
    echo "The value of the tenth parameter is ${10} !"
    echo "The value of the eleventh parameter is ${11} !"
    echo "The amount of the parameters is $# !" # 参数个数
    echo "The string of the parameters is $* !" # 传递给函数的所有参数
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

运行脚本：

```
The value of the first parameter is 1 !
The value of the second parameter is 2 !
The value of the tenth parameter is 10 !
The value of the tenth parameter is 34 !
The value of the eleventh parameter is 73 !
The amount of the parameters is 12 !
The string of the parameters is 1 2 3 4 5 6 7 8 9 34 73 !"
```

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。

另外，还有几个特殊变量用来处理参数，前面已经提到：

特殊变量	说明
$#	传递给函数的参数个数。
$*	显示所有传递给函数的参数。
$@	与$*相同，但是略有区别，请查看 Shell 特殊变量。
$?	函数的返回值。

1.1.22 Shell 输入输出重定向：Shell Here Document，/dev/null 文件

Unix 命令默认从标准输入设备(stdin)获取输入，将结果输出到标准输出设备(stdout)显示。一般情况下，标准输入设备就是键盘，标准输出设备就是终端，即显示器。

输出重定向

命令的输出不仅可以是显示器，还可以很容易的转移向到文件，这被称为输出重定向。

命令输出重定向的语法为：

```
$ command > file
```

这样，输出到显示器的内容就可以被重定向到文件。

例如，下面的命令在显示器上不会看到任何输出：

```
$ who > users
```

打开 users 文件，可以看到下面的内容：

```
$ cat users
oko          tty01    Sep 12 07:30
ai           tty15    Sep 12 13:32
ruth         tty21    Sep 12 10:10
pat          tty24    Sep 12 13:07
steve        tty25    Sep 12 13:03
$
```

输出重定向会覆盖文件内容，请看下面的例子：

```
$ echo line 1 > users
$ cat users
line 1
$
```

如果不希望文件内容被覆盖，可以使用 >> 追加到文件末尾，例如：

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

输入重定向

和输出重定向一样，Unix 命令也可以从文件获取输入，语法为：

```
command < file
```

这样，本来需要从键盘获取输入的命令会转移到文件读取内容。

注意：输出重定向是大于号(>)，输入重定向是小于号(<)。

例如，计算 users 文件中的行数，可以使用下面的命令：

```
$ wc -l users
2 users
```

$
也可以将输入重定向到 users 文件：
$ wc -l < users
2
$
注意：上面两个例子的结果不同：第一个例子，会输出文件名；第二个不会，因为它仅仅知道从标准输入读取内容。

1.1.23 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
$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。

全部可用的重定向命令列表

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。

Here Document

Here Document 目前没有统一的翻译，这里暂译为”嵌入文档“。Here Document 是 Shell 中的一种特殊的重定向方式，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。

下面的例子，通过 wc -l 命令计算 document 的行数：

```
$wc -l << EOF
    This is a simple lookup program
    for good (and bad) restaurants
    in Cape Town.
EOF
3
$
```

也可以 将 Here Document 用在脚本中，例如：

```
#!/bin/bash

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

运行结果：

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

下面的脚本通过 vi 编辑器将 document 保存到 test.txt 文件：

```
#!/bin/sh
```

```
filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

运行脚本：

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

打开 test.txt，可以看到下面的内容：

```
$ cat test.txt
This file was created automatically from
a shell script
$
s
```

/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
```

1.1.24 Shell 文件包含

像其他语言一样，Shell 也可以包含外部脚本，将外部脚本的内容合并到当前脚本。

Shell 中包含脚本可以使用：

```
. filename
```

或

```
source filename
```

两种方式的效果相同，简单起见，一般使用点号(.)，但是注意点号(.)和文件名中间有一空格。

例如，创建两个脚本，一个是被调用脚本 subscript.sh，内容如下：

```
url="123456789"
```

一个是主文件 main.sh，内容如下：

```
#!/bin/bash
. ./subscript.sh
echo $url
```

执行脚本：

```
$chomd +x main.sh
./main.sh
123456789
```

注意：被包含脚本不需要有执行权限。