Bash Shell编程

**Shell是什么**

shell本身是一个命令解释器，介于操作系统的内核态和用户态之间，可以执行系统调用及系统命令等，让用户以此来与操作系统实现互动。同时，他也用来指一种计算机程序语言（类似于python等）。一个shell程序一般被认为一个脚本。

**Shell语言的流派**

目前shell有两大流派：

1. sh：

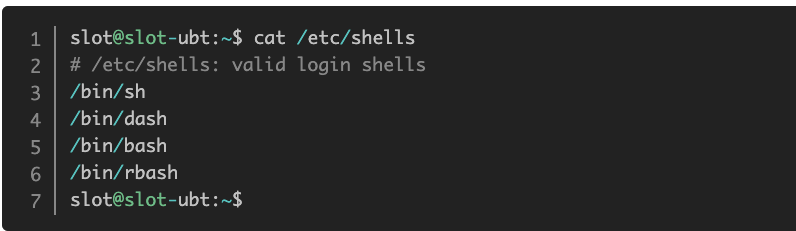
* burne shell (sh)
* burne again shell (bash)

1. csh:

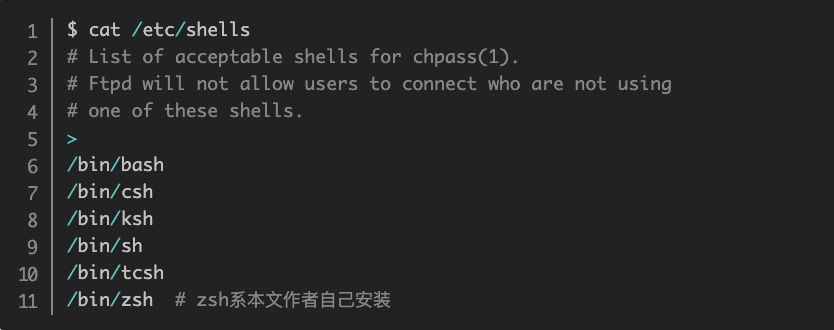
* c shell (csh)
* tc shell (tcsh)
* korn shell (ksh)

目前大部分的Linux系统预设的shell都是bash。

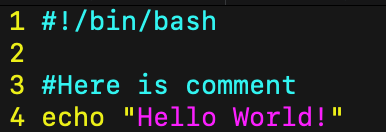
Ubuntu 16.04提供的shell环境(登陆成功后默认使用bash)



Mac OS提供的shell环境：



**一个极简的bash demo：hello\_world.sh**



**执行**

方法1：直接使用bash解释器来解释执行：



或者：



方法2：先将文件属性改为可执行状态：



或者：



然后再直接执行：

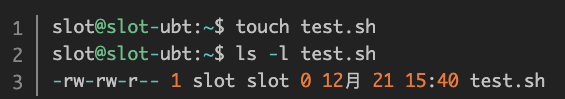


**解释**

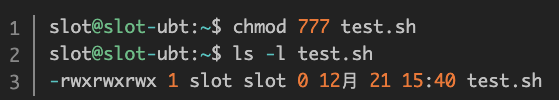
* **#！用来指定执行该脚本的解释器，后面的/bin/bash表明指定/bin目录下的bash程序来解释执行该脚本文件**
* **#开头的是注释行(#!除外)，shell中只有单行注释**

**补充知识**

通过ls -l 命令可以查看文件的属性，例如查看新建文件test.sh的属性。

****

可以看到，一般新建文件的默认属性是-rw-rw-r--，即644，不具有可执行属性x，可使用chmod命令来改变文件属性，例如将文件test.sh的属性修改为可读可写可执行(rwx：4+2+1=7)：



**Bash中的变量**

**变量的定义与赋值**

不像C、java等静态语言需要先声明然后才能使用，而是和python等动态语言类似，Bash变量在使用时直接定义，例如：



注意：

* **=两边不能有空格！否则就是语法错误**
* Bash变量铭铭只能使用字母，下划线和数字，并且不能以数字开头。

**变量的引用**

使用已定义的变量时，只要在变量名前面加$符号即可：



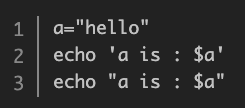
或者使用${var\_name}的形式，{}是可选的，主要是帮助解释器更好地识别变量的边界（推荐）：



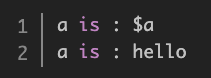
**注意‘’和“”的区别：**

* ‘’:单引号里的任何字符都会原样输出，单引号中对变量的引用是无效的，且单引号中不能出现单引号（对单引号使用转义符也不行）。
* “”：双引号里可以引用变量，可以出现转义字符。

**实例：**

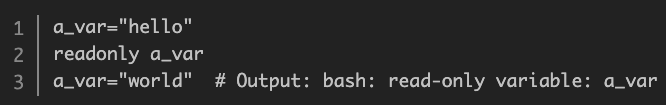


**输出：**

****

**只读变量**

使用**readonly**命令可以将变量限定为只读变量，这与C语言中的const常量类型的情况相同。

****

**删除变量**

使用unset命令可以删除变量，但是不能删除只读变量。

变量被删除后不能再次使用。

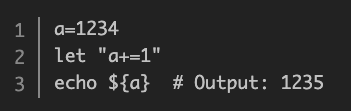


**变量的类型**

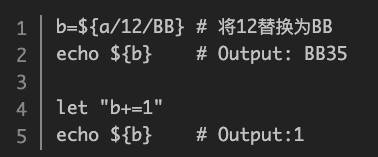
诸如C、java、python等这些高级语言中的变量是有类型的，例如数字类型（整形、浮点型等）、字符串类型、布尔类型，面向对象语言中还有引用类型等。但是，在Bash中，并不对变量区分类型。

本质上来说，Bash变量都是字符串，但是依赖于上下文，Bash也允许比较操作和算术操作。决定这些的关键因素是**变量中的值是否只有数字，**只有当变量是纯数字时，该变量类型才是“数字类型”的，否则就是字符串类型的。

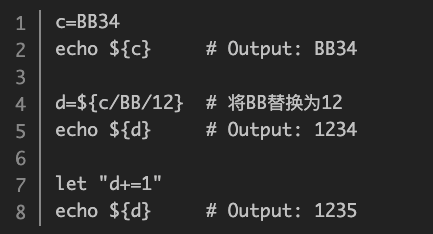
**纯数字变量就是“数字变量”：**

****

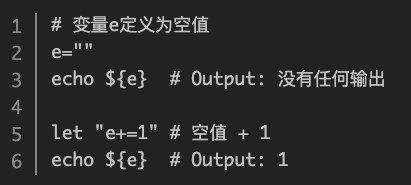
**数字+字符串：字符串变量，字符串变量不能进行数学运算**

****

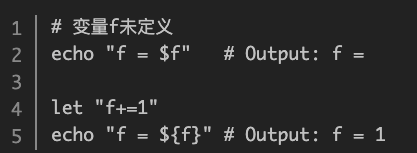
**将变量中的非数字字符替换成数字，得到数字变量**

****

**空变量+数字：数字变量**

****

**未定义的变量+数字：数字变量**

****

**变量的作用域**

* 局部变量：这种变量只有在变量所在的代码块或者函数中才可见，需要使用local声明。
* 全局变量：Bash中用户自定义的普通变量默认是全局变量，可以在本文件中的其他位置引用。
* 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。如果一个shell脚本设置了环境变量，需要使用export命令来通知脚本的环境。

**Bash中的运算符**

* 数学运算符
* 关系运算符
* 布尔运算符
* 逻辑运算符
* 字符串运算符
* 文件测试运算符

**数学运算符**

|  |  |
| --- | --- |
| **数学运算符** | **说明** |
| + | 加法 |
| - | 减法 |
| \* | 乘法 |
| / | 除法 |
| % | 取余 |
| = | 赋值 |
| == | 相等测试，相等则返回true |
| != | 不相等测试，不相等则返回true |

**注意：乘号\*前边必须加反斜杠\才能实现乘法运算**

**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

|  |  |
| --- | --- |
| **关系运算符** | **说明** |
| -eq | (equal)检测两个数是否相等，相等则返回true |
| -ne | (not equal)检测两个数是否相等，不相等则返回true |
| -gt | (greater than)检测左边的数是否大于右边的，如果是，则返回true |
| -lt | (lower than)检测左边的数是否小于右边的，如果是，则返回true |
| -ge | (greater than)检测左边的数是否大于等于右边的，如果是，则返回true |
| -le | (lower than)检测左边的数是否小于等于右边的，如果是，则返回true |

**布尔运算符**

|  |  |
| --- | --- |
| 布尔运算符 | 说明 |
| -a | 与运算，两个表达式都为true才返回true |
| -o | 或运算，有一个表达式为true则返回true |
| ! | 非运算，表达式为true则返回false，否则返回true |

**逻辑运算符**

|  |  |
| --- | --- |
| 逻辑运算符 | 说明 |
| && | 逻辑与 |
| || | 逻辑或 |

**字符串运算符**

|  |  |  |
| --- | --- | --- |
| 字符串运算符 | 说明 | 举例(a=“abc” b=“def”) |
| = | 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，不相等则返回true | [$a!=$b]，返回true |
| -z | 检测字符串长度是否为0，为0返回true | [-z$a]，返回false |
| -n | 检测字符串长度是否为0，不为0则返回true | [-n$a]，返回true |
| str\_name | 检测字符串是否为空，不为空返回true | [$a]返回true |

**文件运算符**

文件测试运算符用于检测Unix文件的各种属性

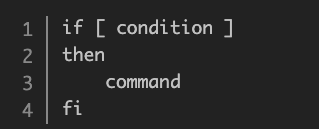
|  |  |
| --- | --- |
| 文件测试运算符 | 说明 |
| -b file | 检测文件是否是块设备文件，如果是，则返回true |
| -c file | 检测文件是否是字符设备文件，如果是，则返回true |
| -d file | 检测文件是否是目录，如果是，则返回true |
| -f file | 检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回true |
| -r file | 检测文件是否可读，如果是，则返回true |
| -w file | 检测文件是否可写，如果是，则返回true |
| -x file | 检测文件是否可执行，如果是，则返回true |
| -s file | 检测文件是否为空（文件大小是否为0），不为空则返回true |
| -e file | 检测文件（包括目录）是否存在，如果是，则返回true |

**Bash的控制流**

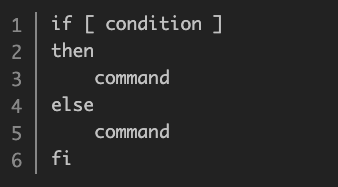
**条件语句**

**if语句**

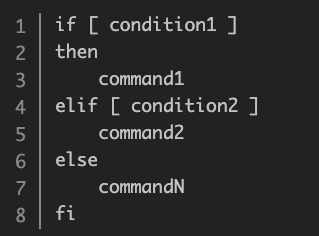
* **if**

****

* **if-else**

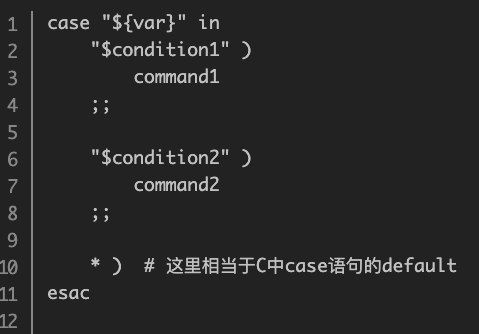
****

* **if-elif-else**

****

**注意：勿忘最后的fi（if的反向拼写）**

**case语句**

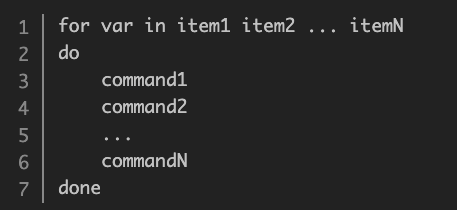
****

**注意：**

* + 对变量使用“”并不是强制的，因为不会发生单词分离
  + 每句测试行，都以右小括号）结尾；
  + 每个条件块都以两个分号；；结尾（作用类似于C中的break）；
  + case块的结束以esac(case的反向拼写)结尾。

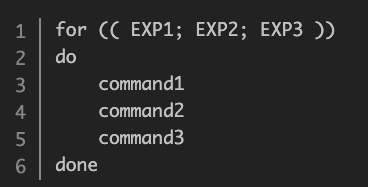
**循环语句**

**for语句**

****

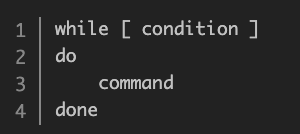
注意：在循环的每次执行中，var将顺序的存取list(item1…itemN)中列出的变量。

**C风格的for循环**

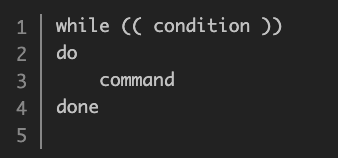


**注意：(())中对变量的引用可以不加$**

**While语句**

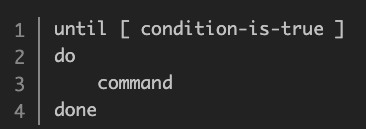


**C风格的while循环**

****

**注意：(())中对变量的引用同样可以不加$**

**until语句**

****

**注意：util结构在循环的顶部判断条件，并且如果条件一直为false，那就一直循环下去，直到条件为真才结束循环（与while相反）。**

**Bash支持的编程模型**

Bash只支持过程式的编程模型，不支持面向对象和函数式等高级编程模型，更不支持对高级设计模式的实现。

**Bash的传参机制**

shell通过位置参数(positional parameters)来给脚本文件传递参数，就是从命令行中传进来的参数，$0，$1，$2，$3…其中：

* + $0是该脚本文件的名字
  + $1是第一个参数，$2是第二个参数

注意：

* + $9以后就需要大括号了，如${10}、${11}、${12}

另外，还有几个特殊字符用来处理参数：

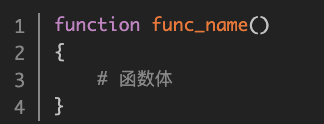
|  |  |
| --- | --- |
| 参数处理 | 说明 |
| $# | 传递到脚本的参数个数 |
| $\* | 以一个单字符串显示所有向脚本传递的参数 |
| $@ | 与$\*相同，但是使用时加引号，并在引号中返回每个参数 |
| $$ | 脚本运行的当前进程ID号 |
| $! | 后台运行的最后一个进程的ID号 |
| $- | 显示shell使用的当前状态，与set命令功能相同 |
| $? | 显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误 |

$\*和$@的区别：

* + 相同点：都表示引用所有的位置参数
  + 不同点：只有在双引号中体现出来。假设在脚本运行时写了三个参数1、2、3，则“\*”等价于“1 2 3”(传递了一个参数)，而“@”等价于“1”“2”“3”（传递了三个参数）

**Bash函数**

**函数的定义**



注意：

* + 关键字function是可选的
  + 函数定义必须在第一次调用前完成，shell没有像C中的函数声明
  + 在一个函数内嵌套另一个函数也是可以的，但是不常用

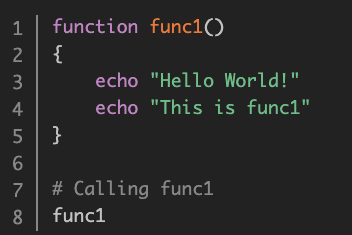
**函数的参数传递**

函数以位置来引用传递过来的参数（就好像它们是位置参数一样），例如$1，$2

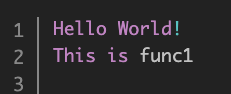
**函数的调用**

函数被调用或被触发，只需要简单的用函数名来调用，**有参数的话将参数依次置于函数名之后。**

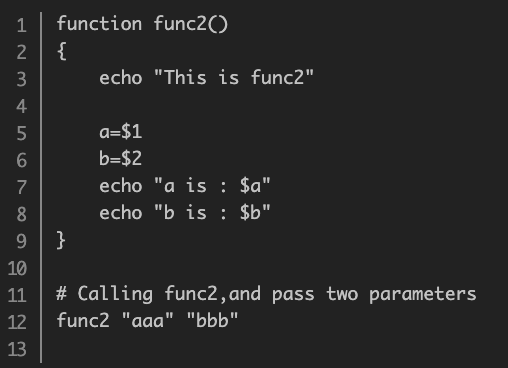
**无参函数的调用**



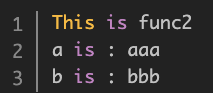
输出：



**有参函数的调用**



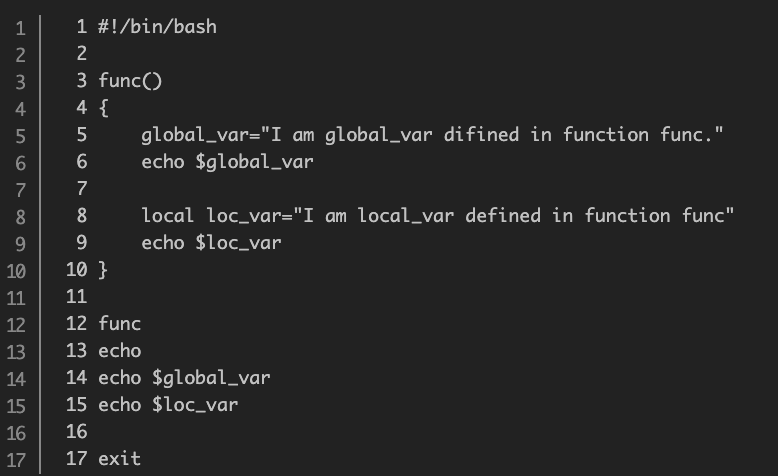
输出：



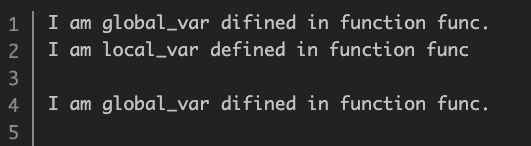
**函数中参数的作用域**

在函数调用之前，所有在函数内声明且没有明确声明为local的变量都可以在函数体外可见（默认为全局变量）。

如果变量用local来声明，那么它只能在该变量声明的代码块中可见，这个代码块就是局部范围。在一个函数内，局部变量意味着只能在函数代码块内它才有意义。



输出：



**Bash数组**

Bash只支持一维数组，用圆括号()来表示，数组元素之间用“空格”符号来分割。

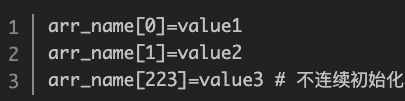
**数组的定义与初始化**

初始化时不需要指定数组的大小。和其他大部分语言一样，bash数组元素的下标从0开始。

**初始化方式1：直接初始化**



**初始化方式2：用下标初始化**



注意：

* + 数组成员不必一定要连续，空缺元素是允许的；
  + 数组的一部分成员允许不被初始化，没有被初始化的元素将打印空值-NULL

**访问数组元素**

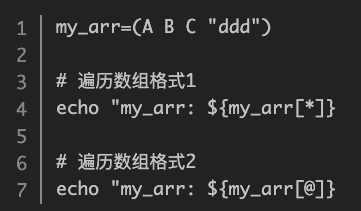
访问数组元素的一般格式：



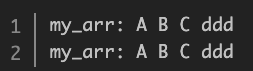
**遍历数组**

使用\*或者@可以获取数组中的所有元素

实例：



输出：



**获取数组长度**

获取数组长度，即数组中的元素个数，和遍历数组语法很相似，只是在数组名前加了#符号。

格式：



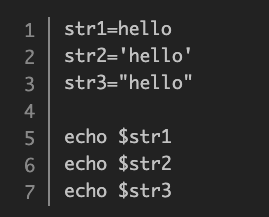
或者：



**Bash字符串操作**

在bash中，**字符串可以用单引号，也可以用双引号，也可以不用引号**。单双引号的区别如前文所述。

字符串实例：



输出：



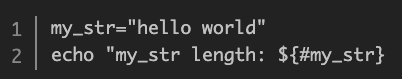
**获取字符串长度**

获取字符串长度和获取数组长度一样，也是在参数名前面加#。

格式：



实例：



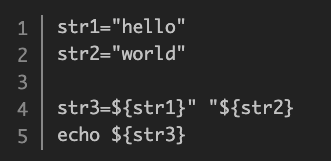
输出：



**字符串拼接**

**和java、python语言相似，bash允许直接将字符串拼接在一起来获得新的字符串。**

实例：



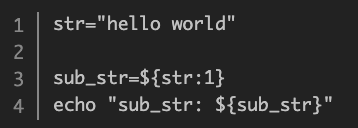
输出：



**提取子串**

|  |  |
| --- | --- |
| 表达式 | 含义 |
| ${string:position} | 在string中，从位置position开始提取子串 |
| ${string:position:length} | 在string中，从位置position开始提取长度为length的子串 |

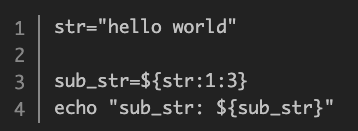
实例1：从位置1开始提取子串



输出：



实例2：从位置1开始提取长度为3的子串



输出：



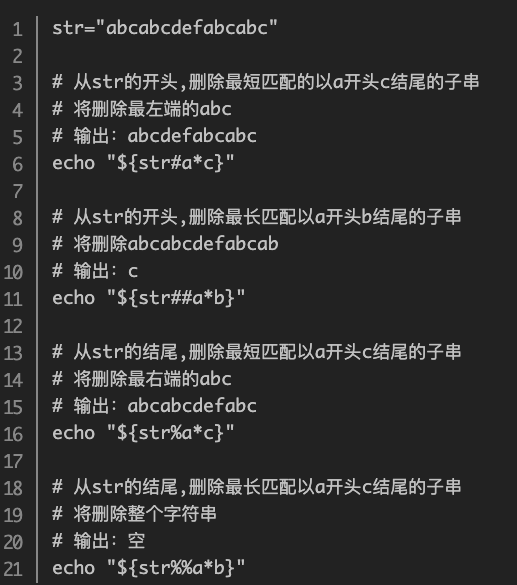
**删除子串**

|  |  |
| --- | --- |
| 表达式 | 含义 |
| ${string#substring} | 从string的开头，删除最短匹配substring的子串 |
| ${string##substring} | 从string的开头，删除最长匹配substring的子串 |
| ${string%substring} | 从string的结尾，删除最短匹配substring的子串 |
| ${string%%substring} | 从string的结尾，删除最长匹配substring的子串 |

注意：

* + #表示从头匹配，%表示从尾匹配
  + 一个符号(#或者%)表示最短匹配，两个符号(##或者%%)表示最长匹配
  + substring可以是正则表达式

实例：



**替换子串**

|  |  |
| --- | --- |
| 表达式 | 含义 |
| ${string/substring/replacement} | 使用$replacement来代替第一个匹配的$substring |
| ${string//substring/replacement} | 使用$replacement代替所有匹配的$substring |
| ${string/#substring/replacement} | 如果$string的前缀匹配$substring，那么就用$replacement来代替匹配到的$substring |
| ${string/%substring/replacement} | 如果$string的后缀匹配$substring，那么就用$replacement来代替匹配到的$substring |

实例：



bash遍历一个目录下的所有文本：

