Python开发之运维基础

讲师：王晓春

本章内容

马哥教育

IT人的高薪职业学院

- 编程基础
- 脚本基本格式
- 变量
- 运算
- 条件测试
- 流程控制
- 循环
- 函数

编程基础

- 程序：指令+数据
- 程序编程风格：
  - 过程式：以指令为中心，数据服务于指令
  - 对象式：以数据为中心，指令服务于数据
- shell程序：提供了编程能力，解释执行

编程基本概念

- 编程逻辑处理方式：
  - 顺序执行
  - 循环执行
  - 选择执行
- shell编程：过程式、解释执行
  - 编程语言的基本结构：
    - 各种系统命令的组合
    - 数据存储：变量、数组
    - 表达式: a + b
    - 语句:if

shell脚本基础

- shell脚本:

  包含一些命令或声明，并符合一定格式的文本文件

- 格式要求：首行shebang机制

  #!/bin/bash

  #!/usr/bin/python

  #!/usr/bin/perl

- shell脚本的用途有：
  - 自动化常用命令
  - 执行系统管理和故障排除
  - 创建简单的应用程序
  - 处理文本或文件

创建shell脚本

- 第一步：使用文本编辑器来创建文本文件
  - 第一行必须包括shell声明序列：#!

    #!/bin/bash
  - 添加注释

    注释以#开头
- 第二步：运行脚本
  - 给予执行权限，在命令行上指定脚本的绝对或相对路径
  - 直接运行解释器，将脚本作为解释器程序的参数运行

脚本规范

- 脚本代码开头约定
  1、第一行一般为调用使用的语言
  2、程序名，避免更改文件名为无法找到正确的文件
  3、版本号
  4、更改后的时间
  5、作者相关信息
  6、该程序的作用，及注意事项
  7、最后是各版本的更新简要说明

脚本的基本结构

- 脚本的基本结构

  #!SHEBANG

  CONFIGURATION_VARIABLES

  FUNCTION_DEFINITIONS

  MAIN_CODE

shell脚本示例


```
#!/bin/bash
# ----------------------------------------
# Filename:    hello.sh
# Revision:    1.1
# Date:        2017/06/01
# Author:      wang
# Email:       wang@gmail.com
# Website:     www.magedu.com
# Description: This is the first script
# ----------------------------------------
# Copyright:   2017 wang
# License:     GPL
 echo   “hello world”
```

脚本调试

- 检测脚本中的语法错误

  bash -n /path/to/some_script

- 调试执行

  bash -x /path/to/some_script

变量

- 变量：命名的内存空间
  - 数据存储方式：
    - 字符：
    - 数值：整型，浮点型
- 变量：变量类型
  - 作用：
    1. 数据存储格式
    2. 参与的运算
    3. 表示的数据范围
  - 类型：
    - 字符
    - 数值：整型、浮点型

变量

- 变量命名法则：
  1、不能使程序中的保留字：例如 if, for
  2、只能使用数字、字母及下划线，且不能以数字开头
  3、见名知义
  4、统一命名规则：驼峰命名法，建议大写

bash中变量的种类

- 根据变量的生效范围等标准划分下面变量类型：

局部变量：生效范围为当前shell进程；对当前shell之外的其它shell进程，包括当前shell的子shell进程均无效

环境（全局）变量：生效范围为当前shell进程及其子进程

本地变量：生效范围为当前shell进程中某代码片断，通常指函数

位置变量：$1, $2, ...来表示，用于让脚本在脚本代码中调用通过命令行传递给它的参数

特殊变量：$?, $0, $*, $@, $#,$$

局部变量

- 变量赋值：name= ‘value’
- 可以使用引用value:
  - (1) 可以是直接字串; name= “root"
  - (2) 变量引用：name="$USER"
  - (3) 命令引用：name=`COMMAND` name=$(COMMAND)
- 变量引用：${name} $name
  - "" ：弱引用，其中的变量引用会被替换为变量值
  - '' ：强引用，其中的变量引用不会被替换为变量值，而保持原字符串
- 显示已定义的所有变量：set
- 删除变量：unset name

环境变量

- 变量声明、赋值：

  export name=VALUE

  declare -x name=VALUE

- 变量引用：$name, ${name}

- 显示所有环境变量：

  env

  printenv

  export

  declare -x

- 删除变量：

  unset name

环境变量

- bash内建的环境变量：
  - PATH
  - SHELL
  - USER
  - UID
  - HOME
  - PWD
  - SHLVL
  - LANG
  - MAIL
  - HOSTNAME
  - HISTSIZE
  - …

只读和位置变量

- 只读变量：只能声明，但不能修改和删除
  - 声明只读变量：

    readonly name

    declare -r name
  - 查看只读变量：

    readonly –p
- 位置变量：在脚本代码中调用通过命令行传递给脚本的参数

  $1, $2, ...：对应第1、第2等参数，shift [n]换位置

  $0: 命令本身

  $*: 传递给脚本的所有参数，全部参数合为一个字符串

  $@: 传递给脚本的所有参数，每个参数为独立字符串

  $#: 传递给脚本的参数的个数

  $@ $* 只在被双引号包起来的时候才会有差异

  set -- 清空所有位置变量

退出状态

- 进程使用退出状态来报告成功或失败
  - 0 代表成功，1 - 255代表失败
  - $? 变量保存最近的命令退出状态
- 例如：

```
ping -c1 -W1 hostdown &> /dev/null
echo $?
```

退出状态码

- bash自定义退出状态码

  exit [n]：自定义退出状态码

  注意：脚本中一旦遇到exit命令，脚本会立即终止；终止退出状态取决于exit命令后面的数字

  注意：如果未给脚本指定退出状态码，整个脚本的退出状态码取决于脚本中执行的最后一条命令的状态码

算术运算

- bash中的算术运算:help let

  +, -, *, /, %取模（取余）, **（乘方）

  实现算术运算：

  (1) let var=算术表达式

  (2) var=$[算术表达式]

  (3) var=$((算术表达式))

  (4) var=$(expr arg1 arg2 arg3 ...)

  (5) declare –i var = 数值

  (6) echo ‘算术表达式’ | bc

- 乘法符号有些场景中需要转义，如*
- bash有内建的随机数生成器：RANDOM（0-32767）

  echo $[$RANDOM%50] ：0-49之间随机数

赋值

- 增强型赋值：

  +=, -=, *=, /=, %=

- let varOPERvalue

  例如:let count+=3

  自加3后自赋值

- 自增，自减：

  let var+=1

  let var++

  let var-=1

  let var--

逻辑运算

- true, false
  1, 0
- 与 :
  1 与 1 = 1
  1 与 0 = 0
  0 与 1 = 0
  0 与 0 = 0
- 或:
  1 或 1 = 1
  1 或 0 = 1
  0 或 1 = 1
  0 或 0 = 0

逻辑运算

- 非：！

  ！1 = 0

  ！0 = 1

- 短路运算

  短路与

  第一个为0，结果必定为0

  第一个为1，第二个必须要参与运算

  短路或

  第一个为1，结果必定为1

  第一个为0，第二个必须要参与运算

- 异或：^

  异或的两个值,相同为假，不同为真

条件测试

- 判断某需求是否满足，需要由测试机制来实现

  专用的测试表达式需要由测试命令辅助完成测试过程

- 评估布尔声明，以便用在条件性执行中
  - 若真，则返回0
  - 若假，则返回1
- 测试命令：
  - test EXPRESSION
  - [EXPRESSION]
  - [[EXPRESSION]]

  注意：EXPRESSION前后必须有空白字符

bash的数值测试

- -v VAR

  变量VAR是否设置

- 数值测试：

  -gt 是否大于

  -ge 是否大于等于

  -eq 是否等于

  -ne 是否不等于

  -lt 是否小于

  -le 是否小于等于

bash的字符串测试

- 字符串测试：

  == 是否等于

  > ascii码是否大于ascii码

  < 是否小于

  != 是否不等于

  =~ 左侧字符串是否能够被右侧的PATTERN所匹配

  注意: 此表达式一般用于[[ ]]中；扩展的正则表达式

  -z "STRING “ 字符串是否为空，空为真，不空为假

  -n "STRING “ 字符串是否不空，不空为真，空为假

- 注意：用于字符串比较时的用到的操作数都应该使用引号

Bash的文件测试

- 存在性测试
  - -a FILE：同-e
  - -e FILE: 文件存在性测试，存在为真，否则为假
- 存在性及类别测试
  - -b FILE：是否存在且为块设备文件
  - -c FILE：是否存在且为字符设备文件
  - -d FILE：是否存在且为目录文件
  - -f FILE：是否存在且为普通文件
  - -h FILE 或 -L FILE：存在且为符号链接文件
  - -p FILE：是否存在且为命名管道文件
  - -S FILE：是否存在且为套接字文件

Bash的文件权限测试

- 文件权限测试：
  - -r FILE：是否存在且可读
  - -w FILE: 是否存在且可写
  - -x FILE: 是否存在且可执行
- 文件特殊权限测试：
  - -u FILE：是否存在且拥有suid权限
  - -g FILE：是否存在且拥有sgid权限
  - -k FILE：是否存在且拥有sticky权限

Bash的文件属性测试

- 文件大小测试：
  - -s FILE: 是否存在且非空
- 文件是否打开：
  - -t fd: fd 文件描述符是否在某终端已经打开
  - -N FILE：文件自从上一次被读取之后是否被修改过
  - -O FILE：当前有效用户是否为文件属主
  - -G FILE：当前有效用户是否为文件属组

Bash的文件属性测试

- 双目测试：

  FILE1 -ef FILE2: FILE1是否是FILE2的硬链接

  FILE1 -nt FILE2: FILE1是否新于FILE2（mtime）

  FILE1 -ot FILE2: FILE1是否旧于FILE2

Bash的组合测试条件

- 第一种方式：

  EXPRESSION1 -a EXPRESSION2 并且

  EXPRESSION1 -o EXPRESSION2 或者

  ! EXPRESSION

  必须使用测试命令进行

- 第二种方式：

  COMMAND1 && COMMAND2 并且

  COMMAND1 || COMMAND2 或者

  ! COMMAND 非

  如：[[-r FILE]] && [[-w FILE]]

逻辑组合

- 根据退出状态而定，命令可以有条件地运行
  - && 代表条件性的AND THEN
  - || 代表条件性的OR ELSE
- 示例

```
test "$A" == "$B" && echo "Strings are equal"
test “$A” -eq “$B” && echo "Integers are equal “
[ "$A" == "$B" ] && echo "Strings are equal"
[ "$A" -eq "$B" ] && echo "Integers are equal “
 [ -f /bin/cat -a -x /bin/cat ] && cat /etc/fstab
  [ -z “$HOSTNAME” -o $HOSTNAME "==\
  "localhost.localdomain" ] && hostname www.magedu.com
```

条件性的执行操作符

- 示例：

```
grep -q no_such_user /etc/passwd || echo 'No such user'
No such user
ping -c1 -W2 station1 &> /dev/null  \
>    && echo "station1 is up"          \
>    || (echo 'station1 is unreachable'; exit 1)
station1 is up
```

使用read命令来接受输入

- 使用read来把输入值分配给一个或多个shell变量

  -p　指定要显示的提示

  -s　静默输入，一般用于密码

  -n　N 指定输入的字符长度N

  -d　‘字符’　输入结束符

  -t　N　TIMEOUT为N秒

  read　从标准输入中读取值，给每个单词分配一个变量

  所有剩余单词都被分配给最后一个变量

  read -p “Enter a filename: “ FILE

流程控制

- 过程式编程语言：
  - 顺序执行
  - 选择执行
  - 循环执行

条件选择if语句

- 选择执行：
- 注意：if语句可嵌套
- 单分支

```
if 判断条件;then
        条件为真的分支代码
fi
```

- 双分支

```
if 判断条件; then
        条件为真的分支代码
else
        条件为假的分支代码
fi
```

if 语句

- 多分支

```
if 判断条件1; then
   条件为真的分支代码
elif 判断条件2; then
   条件为真的分支代码
elif 判断条件3; then
   条件为真的分支代码
else
   以上条件都为假的分支代码
fi
```

- 逐条件进行判断，第一次遇为“真”条件时，执行其分支，而后结束整个if语句

If示例

- 根据命令的退出状态来执行命令

```
if ping -c1 -W2 station1 &> /dev/null; then
        echo 'Station1 is UP'
elif grep "station1" ~/maintenance.txt &> /dev/null;then
        echo 'Station1 is undergoing maintenance ‘
else
        echo 'Station1 is unexpectedly DOWN!'
        exit 1
fi
```

条件判断：case语句


```
case 变量引用 in
PAT1)
        分支1
        ;;
PAT2)
        分支2
        ;;
...
*)
        默认分支
        ;;
esac
```

case支持glob风格的通配符：

*: 任意长度任意字符

?: 任意单个字符

[]：指定范围内的任意单个字符

a|b: a或b

循环

- 循环执行
  - 将某代码段重复运行多次
  - 重复运行多少次：
    - 循环次数事先已知
    - 循环次数事先未知
  - 有进入条件和退出条件
- for, while, until

for循环

- for 变量名 in 列表;do
  循环体
  done
- 执行机制：
  依次将列表中的元素赋值给“变量名”；每次赋值后即执行一次循环体; 直到列表中的元素耗尽，循环结束

for循环

- 列表生成方式：
  - (1) 直接给出列表
  - (2) 整数列表：
    - (a) {start..end}
    - (b) $(seq [start [step]] end)
  - (3) 返回列表的命令
    - $(COMMAND)
  - (4) 使用glob，如：*.sh
  - (5) 变量引用；
    - $@, $*

while循环

- while CONDITION; do
  循环体
  done
- CONDITION：循环控制条件；进入循环之前，先做一次判断；每一次循环之后会再次做判断；条件为“true”，则执行一次循环；直到条件测试状态为“false”终止循环
- 因此：CONDTION一般应该有循环控制变量；而此变量的值会在循环体不断地被修正
- 进入条件：CONDITION为true
- 退出条件：CONDITION为false

until循环

- until CONDITION; do
  循环体
- done

- 进入条件： CONDITION 为false
- 退出条件： CONDITION 为true

循环控制语句continue

- 用于循环体中
- continue [N]：提前结束第N层的本轮循环，而直接进入下一轮判断；最内层为第1层

```
while CONDTIITON1; do
        CMD1
        ...
        if CONDITION2; then
                continue
        fi
        CMDn
        ...
done
```

循环控制语句break

- 用于循环体中
- break [N]：提前结束第N层循环，最内层为第1层

```
while CONDTIITON1; do
        CMD1
        ...
        if CONDITION2; then
                break
        fi
        CMDn
        ...
done
```

特殊用法

- 双小括号方法，即((...))格式，也可以用于算术运算
- 双小括号方法也可以使bash Shell实现C语言风格的变量操作

```
I=10
((I++))
```

- for循环的特殊格式：

```
for ((控制变量初始化;条件判断表达式;控制变量的修正表达式))
do
        循环体
done
```

- 控制变量初始化：仅在运行到循环代码段时执行一次
- 控制变量的修正表达式：每轮循环结束会先进行控制变量修正运算，而后再做条件判断

函数介绍

- 函数function是由若干条shell命令组成的语句块，实现代码重用和模块化编程
- 它与shell程序形式上是相似的，不同的是它不是一个单独的进程，不能独立运行，而是shell程序的一部分
- 函数和shell程序比较相似，区别在于：
  - Shell程序在子Shell中运行
  - 而Shell函数在当前Shell中运行。因此在当前Shell中，函数可以对shell中变量进行修改

定义函数

- 函数由两部分组成：函数名和函数体
- help function
- 语法一：

```
f_name（ ）{
        ...函数体...
}
```

- 语法二：

```
function f_name {
        ...函数体...
}
```

- 语法三：

```
function f_name（ ）{
        ...函数体...
}
```

函数使用

- 函数的定义和使用：
  - 可在交互式环境下定义函数
  - 可将函数放在脚本文件中作为它的一部分
  - 可放在只包含函数的单独文件中
- 调用：函数只有被调用才会执行

  调用：给定函数名

  函数名出现的地方，会被自动替换为函数代码
- 函数的生命周期：被调用时创建，返回时终止

函数返回值

- 函数有两种返回值：
- 函数的执行结果返回值：
  (1) 使用echo等命令进行输出
  (2) 函数体中调用命令的输出结果
- 函数的退出状态码：
  (1) 默认取决于函数中执行的最后一条命令的退出状态码
  (2) 自定义退出状态码，其格式为：
  return 从函数中返回，用最后状态命令决定返回值
  return 0 无错误返回。
  return 1-255 有错误返回

交互式环境下定义和使用函数

- 示例：

```
dir() {
> ls -l
> }
```

- 定义该函数后，若在$后面键入dir，其显示结果同ls -l的作用相同

  dir

- 该dir函数将一直保留到用户从系统退出，或执行了如下所示的unset命令

  unset dir

在脚本中定义及使用函数

- 函数在使用前必须定义，因此应将函数定义放在脚本开始部分，直至shell首次发现它后才能使用
- 调用函数仅使用其函数名即可
- 示例：

```
cat func1
#!/bin/bash
# func1
hello()
{
  echo "Hello there today's date is `date +%F`"
}
echo "now going to the function hello"
hello
echo "back from the function"
```

使用函数文件

- 可以将经常使用的函数存入函数文件，然后将函数文件载入shell
- 文件名可任意选取，但最好与相关任务有某种联系。例如：functions.main
- 一旦函数文件载入shell，就可以在命令行或脚本中调用函数。可以使用set命令查看所有定义的函数，其输出列表包括已经载入shell的所有函数
- 若要改动函数，首先用unset命令从shell中删除函数。改动完毕后，再重新载入此文件

创建函数文件

- 函数文件示例：

```
cat  functions.main
#!/bin/bash
#functions.main
findit()
{
   if [ $# -lt 1 ] ;  then
      echo "Usage:findit file"
      return 1
   fi
   find / -name $1 –print
}
```

载入函数

- 函数文件已创建好后，要将它载入shell
- 定位函数文件并载入shell的格式：

  . filename 或 source filename
- 注意：此即<点> <空格> <文件名>

  这里的文件名要带正确路径
- 示例：

  上例中的函数，可使用如下命令：

  . functions.main

检查载入函数

- 使用set命令检查函数是否已载入。set命令将在shell中显示所有的载入函数
- 示例：

```
set
 findit=( )
{
   if [ $# -lt 1 ]; then
   echo "usage :findit file";
   return 1
   fi
   find / -name $1 -print
}
```

执行shell函数

- 要执行函数，简单地键入函数名即可
- 示例：

```
findit groups
/usr/bin/groups
/usr/local/backups/groups.bak
```

删除shell函数

- 现在对函数做一些改动后，需要先删除函数，使其对shell不可用。使用unset命令完成删除函数
- 命令格式为：

  unset function_name
- 示例：

  unset findit

  再键入set命令，函数将不再显示

函数参数

- 函数可以接受参数：

传递参数给函数：调用函数时，在函数名后面以空白分隔给定参数列表即可；例如“testfunc arg1 arg2 ...”

在函数体中当中，可使用$1, $2, ...调用这些参数；还可以使用$@, $*, $#等特殊变量

函数变量

- 变量作用域：

  环境变量：当前shell和子shell有效

  本地变量：只在当前shell进程有效，为执行脚本会启动专用子shell进程；因此，本地变量的作用范围是当前shell脚本程序文件，包括脚本中的函数

  局部变量：函数的生命周期；函数结束时变量被自动销毁

- 注意：如果函数中有局部变量，如果其名称同本地变量，使用局部变量
- 在函数中定义局部变量的方法

  local NAME=VALUE

函数递归示例

- 函数递归：
  - 函数直接或间接调用自身
  - 注意递归层数
- 递归实例：

阶乘是基斯顿·卡曼于 1808 年发明的运算符号，是数学术语

一个正整数的阶乘（factorial）是所有小于及等于该数的正整数的积，并且有0的阶乘为1，自然数n的阶乘写作n!

n!=1×2×3×...×n

阶乘亦可以递归方式定义：0!=1，n!=(n-1)!×n

n!=n(n-1)(n-2)...1

n(n-1)! = n(n-1)(n-2)!

函数递归示例

- 示例：fact.sh

```
#!/bin/bash
#
fact() {
    if [ $1 -eq 0 -o $1 -eq 1 ]; then
                echo 1
    else
                echo $[$1*$(fact $[$1-1])]
    fi
}
fact $1
```

fork炸弹

- fork炸弹是一种恶意程序，它的内部是一个不断在fork进程的无限循环，实质是一个简单的递归程序。由于程序是递归的，如果没有任何限制，这会导致这个简单的程序迅速耗尽系统里面的所有资源
- 函数实现

```
:(){ :|:& };:
bomb() { bomb | bomb & }; bomb
```

- 脚本实现

```
cat Bomb.sh
 #!/bin/bash
./$0|./$0&
```

祝大家学业有成

谢　谢