bash编程

基础知识

语言简介

学习目标：了解 bash是干什么的，怎么使用。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

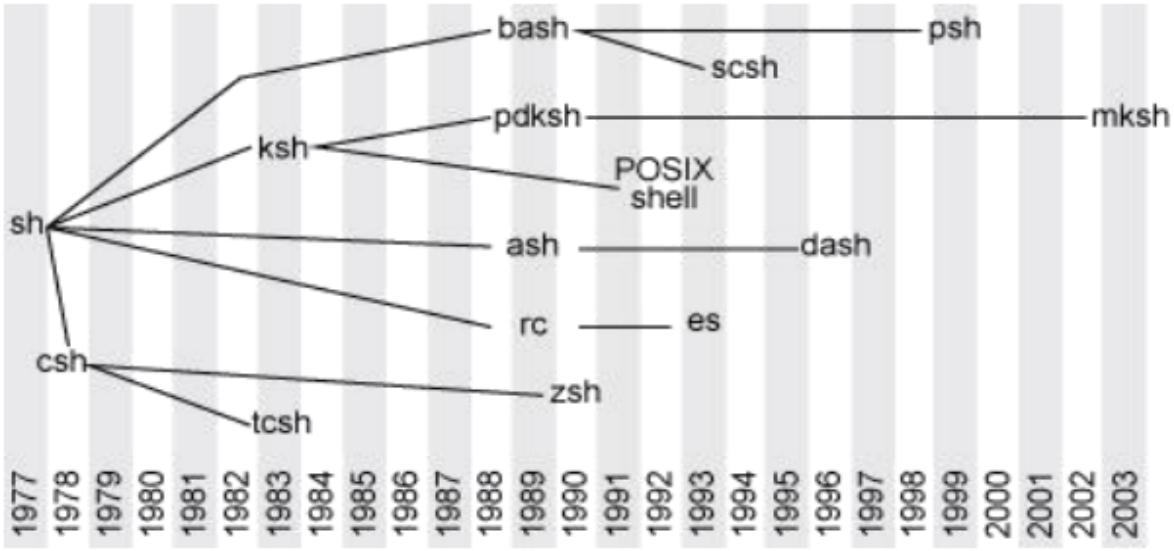
基础知识

简介

> 在计算机科学中，Shell就是一个命令解释器。 shell是位于操作系统和应用程序之间，是他们二者最主要的接口，shell负责把应用程序的输入命令信息解释给操作系统，将操作系统指令处理后的结果解释给应用程序。 一句话，shell就是在操作系统和应用程序之间的一个命令翻译工具。

分类

类型	说明
图形界面shell	图形界面shell就是我们常说的桌面
命令行式shell	windows系统:cmd.exe 命令提示字符 linux系统:sh / csh / ksh / bash / ...



我们常说的shell是命令行式的shell,在工作中常用的是linux系统下的bash。

简单实践

查看当前系统的shell类型

```
echo $SHELL
```

查看当前系统环境支持的shell

```
[root@linux-node1 ~]# cat /etc/shells
/usr/bin/sh
/usr/bin/bash
/usr/sbin/nologin
```

更改默认的shell

```
chsh <用户名> -s <新shell>
```

使用方式

方式	作用	特点
手工方式	手工敲击键盘,在shell的命令行输入命令,按Enter后,执行通过键盘输入的命令,然后shell返回并显示命令执行的结果.	逐行输入命令、逐行进行确认执行
脚本方式	就是说我们把手工执行的命令a，写到一个脚本文件b中，然后通过执行脚本b，达到执行命令a的效果. 当可执行的Linux命令或语句不在命令行状态下执行，而是通过一个文件执行时，我们称文件为shell脚本。	执行文件达到批量执行命令的效果

shell脚本示例

现在我们来使用脚本的方式来执行以下

```
#!/bin/bash
# 这是临时shell脚本
echo 'nihao'
echo 'devops'
```

脚本执行效果

```
[root@linux-node1 ~]# /bin/bash test.sh
nihao
devops
```

小结

```
shell 是 命令解释器
shell 两分类图形+命令行(bash)
脚本 是可执行命令在一个文件中的组合
```

简单实践

学习目标：了解 bash是怎么操作什么的，怎么规范的使用。

这一节，我们从 通用操作、开发技巧、小结 三个方面来学习。

通用操作

脚本创建工具

```
常见编辑器是 vi/vim.
```

脚本命名

```
有意义，方便我们通过脚本名，来知道这个文件是干什么用的。
```

脚本内容

```
各种可以执行的命令
```

注释

```
单行注释(#),多行注释(:<<字符 ... 字符)
```

执行操作

```
Shell脚本的执行通常可以采用以下几种方式

bash /path/to/script-name
    或  /bin/bash /path/to/script-name     （**强烈推荐使用**）
/path/to/script-name
    或    ./script-name     （当前路径下执行脚本）
source script-name
    或    . script-name     （注意“.“点号）
```

说明

```
1、脚本文件本身没有可执行权限或者脚本首行没有命令解释器时使用的方法，我们推荐用bash执行。
使用频率：☆☆☆☆☆

2、脚本文件具有可执行权限时使用。
使用频率：☆☆☆☆

3、使用source或者.点号，加载shell脚本文件内容，使shell脚本内容环境和当前用户环境一致。
使用频率：☆☆☆
使用场景：环境一致性
```

开发技巧

开发技巧

```
1、脚本命名要有意义，文件后缀是.sh
2、脚本文件首行**是而且必须是**脚本解释器
    #!/bin/bash
3、脚本文件解释器后面要有脚本的基本信息等内容
    脚本文件中尽量不用中文注释;
    尽量用英文注释，防止本机或切换系统环境后中文乱码的困扰
    常见的注释信息：脚本名称、脚本功能描述、脚本版本、脚本作者、联系方式等
4、脚本文件常见执行方式：bash 脚本名
5、脚本内容执行：从上到下，依次执行
6、代码书写优秀习惯;
    1）成对内容的一次性写出来,**防止遗漏**。
        如：()、{}、[]、''、``、""
    2）[]中括号两端要有空格,书写时即可留出空格[    ],然后再退格书写内容。
    3）流程控制语句一次性书写完，再添加内容
7、通过缩进让代码易读;(即该有空格的地方就要有空格)
```

小结

```
基本操作：
    脚本名称 一定要有意义
    脚本内容 命令的合理罗列
    脚本注释 单行注释"#",多行注释":<<字符 ... 字符"
    shell脚本标准执行方法： /bin/bash /path/to/script-name
开发技巧：
    shell脚本开发规范重点：2-4-5
    shell脚本开发小技巧：3-6-7
```

变量基础

学习目标：了解 bash是干什么的，怎么使用。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简单介绍

```
变量包括两部分：
    变量名(不变的) + 变量值(变化的)

表现样式：
    变量名=变量值

功能定位
    通过变量名的调用，达到快速执行批量的效果
```

注意事项：

> 变量的全称应该成为变量赋值，简称变量，在工作中，我们一般只xx是变量，其实是是将这两者作为一个整体来描述了。准确来说，我们一般所说的变量其实指的是：变量名。

shell 中的变量分为三大类：

分类	作用
本地变量	变量名仅仅在当前终端有效
全局变量	变量名在当前操作系统的所有终端都有效
shell内置变量	shell解析器内部的一些功能参数变量

简单实践

小结

```
```

变量详解

学习目标：了解 bash的三大变量怎么使用。

这一节，我们从 本地变量、全局变量、基本操作、内置变量、小结 五个方面来学习。

本地变量

本地变量

> 本地变量就是：在当前系统的某个环境下才能生效的变量，作用范围小。
> 本地变量包含两种：普通变量和命令变量

普通变量的定义方式有如下三种

类型	样式	特点
无引号	变量名=变量值	变量值必须是一个整体，中间没有特殊字符 "=" 前后不能有空格
单引号	变量名='变量值'	原字符输出
双引号	变量名="变量值"	变量值先解析，后整合

习惯：数字不加引号，其他默认加双引号

命令变量的定义方式有如下两种

类型	样式	特点
反引号	变量名=`命令`	反引号
小括号	**变量名=$(命令)**	$()

执行流程：

```
1、执行`或者$()范围内的命令
2、将命令执行后的结果，赋值给新的变量名A
```

全局变量

全局变量

```
在当前系统的所有环境下都能生效的变量，可以基于env命令查看
```

表现样式

```
export 变量=值
```

基本操作

基本操作

```
查看
    $变量名 、"$变量名"、${变量名}、"${变量名}"

取消
    unset 变量名
```

内置变量

属性含义

符号	意义	符号	意义
$0	获取当前脚本文件名	$n	获取脚本的第n个参数值，样式：$1,${10}
$#	获取脚本参数的总个数	$?	获取上一个指令的状态返回值（0为成功，非0为失败）
$*	获取所有参数，存放到一个字符串中	$@	获取所有参数，存放到一个列表里面
$$	获取当前程序进程号	$!	获取上一个进程的id

演示效果：

$0 获取脚本的名称

```
#!/bin/bash
# 获取脚本的名称
echo "脚本的名称是：  file.sh"
echo "脚本的名称是：$0"
```

$n 获取位置参数值

```
#!/bin/bash
# 获取指定位置的参数
echo "第一个位置的参数是： $1"
echo "第二个位置的参数是： $2"
```

$# 获取参数总数量

```
#!/bin/bash
# 获取当前脚本传入的参数数量
echo "当前脚本传入参数数量是： $#"
```

$? 获取文件执行或者命令执行的返回状态值

```
# bash nihao
bash: nihao: No such file or directory
# echo $?
127
# ls
file1.sh  num.sh  test.sh  weizhi.sh
# echo $?
0
```

$* vs $@

```
#!/bin/bash

# $*获取到的参数内容放到一个字符串
for arg in "$*"; do
    echo '$*获取到的参数内容：' $arg
done

# 查看$@获取的元素内容
for arg in "$@"; do
    echo '$@获取到的参数内容：' $arg
done
```

`$$` 效果

```
#!/bin/bash

# 获取当前程序执行时候的进程号
echo '$$ 获取当前程序执行时候的进程号'
```

`$!` 获取上一条命令的进程号

```
执行一条命令
# sleep 15 &
[1] 8080

# 获取上一条命令的进程号
# echo $!
8080
```

小结

```

```

条件表达式

学习目标：了解 bash基础的条件表达式。

这一节，我们从 条件判断、表达式、小结 三个方面来学习。

基础知识

语法格式

条件的结果无非就是成立或者不成立，而我们之前所学的$?就可以表示，判断条件是否成立的过程我们称为条件判断，他一般有两种表现形式：

```
A: test 条件表达式
B: [ 条件表达式 ]

注意：
    但后者需要注意方括号[、]与条件表达式之间至少有一个空格。
    条件成立，状态返回值是0；条件不成立，状态返回值是1
```

简单实践

逻辑表达式

```
符号            样式                作用
&&             命令1 && 命令2       只有1成功，2才执行
||            命令1 ||  命令2       1和2只能执行一个
```

示例：

```
# [ 1 = 1 ] && echo "条件成立"
条件成立

# [ 1 = 2 ] || echo "条件不成立"
条件不成立
```

```
# [ 1 = 2 ] && echo "条件成立"
#

 # [ 1 = 1 ] || echo "条件不成立"
 #
```

文件表达式

```
符号              样式                作用
-f|d|s          -f file_name     判断文件格式(普通文件|目录|链接文件)
-r|w|x          -x file_name     判断文件权限(读写执行)
```

示例

```
# [ -f weizhi.sh ] && echo "是一个文件"          # [ -f weizhi.sddh ] || echo "不是一个文件"
是一个文件                                      不是一个文件

# [ -d weizhi.sddh ] || echo "不是一个目录"     # [ -x age.sh ] || echo "文件没有执行权限"
不是一个目录                                    文件没有执行权限
```

一般表达式

数字	样式	特点	字符串	样式	特点
eq	数字1 eq 数字2	相等eq，不等ne	== !=	str1 == str2	字符串内容是否一致
gt	数字1 gt 数字2	gt大于，小于lt	-z -n	-z str1	z空，n不空

注意：在字符串表达式中，"==" 可以简写为 "=", 但是我们不推荐。

数字示例：

```
n1 -eq n2    相等          n1 -ne n2    不等于          n1 -ge n2    大于等于
n1 -gt n2    大于          n1 -lt n2    小于          n1 -le n2    小于等于
```

字符示例：

```
[ a == a ]          [ a != a ]
echo $?          echo $?

[ -z daf ]    [ ! -z daf ]    [ -n daf ]    [ ! -n daf ]
```

小结

```

```

逻辑组合

学习目标：了解 进阶的bash逻辑控制语句。

这一节，我们从 条件组合、逻辑进阶、小结 三个方面来学习。

条件组合

条件组合

所谓的条件组合，主要说的是，多条件场景下的 或与非 三种情况，这些内容，我们一般基于下列方式来进行解析：

```
    !   非运算，表达式结果取反。示例 [ ! false ] 返回 true。
    -o  或运算，多个表达式只要有一个为true，整体为true。示例 [ $a -lt 20 -o $b -gt 100 ]。
    -a  与运算，多个表达式都为true，整体为true。示例[ $a -lt 20 -a $b -gt 100 ] 返回 false。
```

非运算实践

```
# a=5
# [ $a -gt 5 ]
# echo $?
1
# [ ! $a -gt 5 ]
# echo $?
0
```

或运算实践

```
# a=5
# b=7
# [ $a -gt 3 -o $b -lt 7 ]
# echo $?
0
# [ $a -lt 3 -o $b -lt 7 ]
# echo $?
1
```

与运算实践

```
# a=5
# b=7
# [ $a -gt 3 -a $b -lt 7 ]
# echo $?
1
# [ $a -gt 3 -a $b -eq 7 ]
# echo $?
0
```

逻辑进阶

简介

所谓的逻辑进阶，说的还是 && 和 || ，只不过说的是完整写法，结果是整体的状态值

```
    [[ 条件1 && 条件2 ]]  只有所有条件为真，整体结果为真。
    [[ 条件1 || 条件2 ]]  只要有一个条件为真，整体结果为真。
```

其实，它也可以理解为 -a 或者 -o 的另外一种写法。

或运算实践

```
# [[ $a -gt 3 || $b -lt 7 ]]
# echo $?
0
# [[ $a -lt 3 || $b -lt 7 ]]
# echo $?
1
```

与运算实践

```
# [[ $a -gt 3 && $b -lt 7 ]]
# echo $?
1
# [[ $a -gt 3 && $b -eq 7 ]]
# echo $?
0
```

小结

```
```

计算表达式

学习目标：了解 bash常见的计算表达式。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

需求

```
我们需要在 shell 内部对数据进行普通的数据计算操作
```

基本语法

```
方法1：数据运算
    $((数据计算))
    注意：数据计算表达式不受空格限制
方法2：本身计算
    let 变量名=数据计算
    注意：数据计算表达式必须在一起，不允许有空格
方法3：数据运算
    expr 数据计算
    注意：计算的结果不会交给一个值，也不支持变量
```

简单实践

方法1实践

```
# echo $((5+1))
6

# a=5
# echo $((a+=1))
6
```

方法2实践

```
# let a=4+3
# echo $a
7
```

方法3计算

```
# expr 3 + 4
7

# expr $a + 1
10

# a=10
# b=20
# val=`expr $a + $b`
# echo $val
30
```

小结

```
```

字符串基本操作

学习目标：了解 bash的字符串的常见操作。

这一节，我们从 长度计算、提取内容、转换操作、小结 四个方面来学习。

长度计算

需求

```
获取字符串的内容长度
```

方法

```
方法1：  ${#String}
方法2：  expr length "$String"
方法3：  echo $String |awk '{print length}'
```

简单实践

```
# a=charsetstring

# echo ${#a}
13

# expr length "$a"
13

# echo "$a" | awk '{print length}'
13
```

提取内容

需求

```
我们需要字符串内部的部分数据
```

方法

```
${string:偏移量:截取长度}
注意：
    偏移量  指的是列表的索引值，即默认第一个字符的索引值是0
    截取长度  表示要截取的字符串长度，如果为负值，表示反向截取
```

简单实践

```
# echo $a
charsetstring

# echo ${a:3:5}
rsets

# echo ${a:1:-1}          获取2~最后位置的元素
harsetstrin

# echo ${a:2}             获取2~最后位置的元素
arsetstring

# echo ${a::-2}           从首位到倒数第二位
charsetstri
```

转换操作

需求

```
对字符串内容进行简单的大小写更改
```

方法

```
${string^^}   全部转换成大写
${string,,}   全部转换成小写
```

简单实践

```
# c=SetCharsetString

# echo ${c^^}
SETCHARSETSTRING

# echo ${c,,}
setcharsetstring
```

字符串进阶

学习目标：了解 bash的字符串的进阶操作。

这一节，我们从 删除操作、替换操作、变量赋值、小结 四个方面来学习。

删除操作

需求

```
根据需求，删除或者替换指定的信息
```

常见语法

```
${string#字符串}          删除string内部头部存在匹配的字符串的话，删除头部的字符
${string##字符串正则}     贪婪模式，删除string内部所有匹配到的信息

${string%字符串}          删除string内部尾部存在匹配的字符串的话，删除头部的字符
${string%%字符串正则}     贪婪模式，删除string内部所有匹配到的信息

${string/字符串正则}      删除首次匹配到的内容
${string//字符串正则}     删除全部匹配到的内容

${string/#字符串正则}     删除匹配到的头部内容，等同于 ${string#字符串}
${string/%字符串正则}     删除匹配到的尾部内容，等同于 ${string%字符串}
```

删除头部信息

```
# echo $b
setcharsetstring

# echo ${b#set}
charsetstring

# echo ${b#char}
setcharsetstring

# echo ${b##s*t}
ring
```

删除尾部信息

```
# echo ${b%ring}
setcharsetst

# echo ${b%%r*g}
setcha
```

删除操作

```
# echo ${b/set}
charsetstring

# echo ${b//set}
charstring

# echo ${b/#set}
charsetstring

# echo ${b/%ing}
setcharsetstr
```

替换操作

需求

```
修改字符串内部的部分信息，将其替换为指定的内容
```

语法

```
${string/正则表达式/替换内容}     将正则表达式 第一次 匹配到的信息替换成指定的信息
${string//正则表达式/替换内容}    将正则表达式 全部 匹配到的信息替换成指定的信息
```

替换实践

```
# echo $b
setcharsetstring

# echo ${b/set/SET}
SETcharsetstring

# echo ${b//set/SET}
SETcharSETstring
```

赋值操作

需求

```
在编写脚本的时候，我们需要进行对某些变量进行 临时性 或 即时性 的赋值操作，从而满足对应的需求。
```

语法

```
${string:-value}：  如果变量string为空或者未设置，则返回value的值；否则返回string自己的值
${string:=value}：  如果变量string为空或者未设置，则将value的值交给string；否则返回string自己的值

${string:+value}：  如果变量string有内容，则返回value的值
${string:?value}：  如果变量string为空或者未设置，则返回value的值到标准错误输出 strout
```

:- 实践

```
# echo $a
charsetstring

# echo ${a:-666}
charsetstring

# a=""
# echo ${a:-666}
666

# echo ${a}
```

:= 实践

```
# echo ${aa}

# echo ${aa:=666}
666

# echo ${aa}
666

# echo ${aa:=777}
666
```

:+ 实践

```
# echo ${aa}
666

# echo ${aa:+888}
888

# aa=""
# echo ${aa:+888}
```

:? 实践

```
# aa=""
# echo ${aa:+888}

# echo ${aa:?'error'}
-bash: aa: error

# aa="aaa"
# echo ${aa:?'error'}
aaa
```

小结

```

```

数组操作

学习目标：

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

数组

```
    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。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.

    在shell中，数组也称为 关联数组，因为它是将 变量名和值关联在一起了。
```

作用

```
将一个范围的数据放在一起，进行批量管理

分类：
    数组按照表现形式上，可以划分为：
        关联数组(普通数组) - 所有元素按照顺序依次排列
        稀疏数组 - 元素的不按照顺序排列，可以有间隔，即 1 3 5 等，中间找不到 2 4
```

常见操作

```
创建数组
    定制空数组                declare -a array_name
    定制数组                  array_name=(value1 ... valuen)
    定制数组(单值)             array_name[0]=value0
    定制稀疏数组               array_name=([index1]=v1 [index3]=v3 [index7]=v7)
查看数组
    获取具体内容               ${array_name[index]}
    获取长度                  ${#array_name[index]}
    获取内容的部分信息         ${array_name[index]:pos:length}
    获取所有索引               ${!array_name[@]}
更改数组
    更改具体值                 array_name[index]=值
```

```
    部分内容替换                ${array_name[index]/原内容/新内容}
删除数组
    删除具体值                 unset array_name[index]
    删除整个数组                unset array_name
```

简单实践

创建实践

```
标准创建
array_name=(value0 value1 value2 value3)

逐个增加
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2

稀疏数组
array_name=([0]=v1 [2]=v2 [4]=v4)

技巧 - 把指定目录下的所有文件动态添加到数组中
file_array=($(ls /tmp/))
```

查看实践

```
查看指定位置元素
echo ${array_name[1]}
    注意：索引的值是从 0 开始计算的

查看所有位置元素
echo ${array_name[@]}
echo ${array_name[*]}

获取所有位置的索引号
echo ${!array_name[*]}

获取元素数量
echo ${#array_name[1]}
echo ${#array_name[@]}
echo ${#array_name[*]}

获取元素的部分信息
echo ${array_name[2]:0:2}
```

更改实践

```
单值更改
array_name[1]=444
echo ${array_name[1]}

内容更改
echo ${array_name[2]}
echo ${array_name[2]/va/hahhah}
```

删除实践

```
删除单个元素
unset array_name[1]

删除整个数组
unset array_name
```

小结

配置文件

学习目标：了解 bash的多种配置文件。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

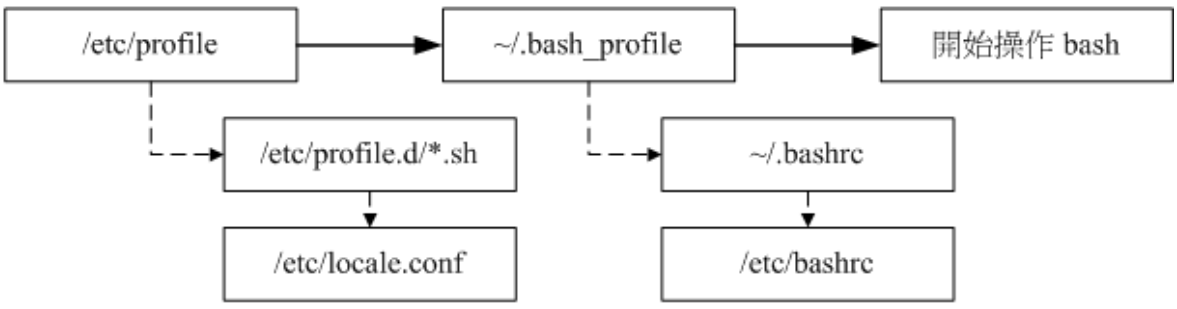
基础知识

需求

```
简单了解bash启动过程中涉及到的主要配置文件
```

bash涉及到的几个重要文件

```
系统级别：
    /etc/profile              操作系统级别
    /etc/profile.d/*.sh       系统软件级别
个人级别：    (优先级从上到下，一般只会有最后一个)
    ~/.bash_profile
    ~/.bash_login
    ~/.profile
bash级别：
    ~/.bashrc                 定制用户环境
```



简单实践

python虚拟环境

```
1 安装python
    sudo apt install python3 python3-pip -y
    注意：python3 默认已经安装好了
2 安装虚拟环境
    sudo apt install virtualenv virtualenvwrapper -y
3 定制虚拟环境
    # 3-1 定制配置
    python@python-auto:~$ tail -n3 ~/.bashrc
```

```
    # 定制虚拟环境的配置
    export WORKON_HOME=$HOME/.virtualenvs
    source /usr/share/virtualenvwrapper/virtualenvwrapper.sh

    # 3-2 配置生效
    source ~/.bashrc

4 创建虚拟环境安装模块
    创建命令
        makevirtualenv -p /usr/bin/python3 python_auto
```

小结

```

```

流程控制

if条件

学习目标：了解 bash if语句的基本使用。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

单分支if - 单条件，单结果

```
if [ 条件 ]
then
    指令1
fi
```

双分支if - 单条件，两结果

```
if [ 条件 ]
then
    指令1
else
    指令2
fi
```

多分支if - n条件，n+1结果

```
if [ 条件 ]
then
    指令1
elif [ 条件2 ]
then
    指令2
else
    指令3
fi
```

简单实践

单分支if示例

```
#!/bin/bash
# 单if语句的使用场景
if [ "$1" == "nan" ]
then
    echo "您的性别是 男"
fi
```

双分支if示例

```
#!/bin/bash
# 双if语句的使用场景
if [ "$1" == "nan" ]
then
    echo "您的性别是 男"
else
    echo "您的性别是 女"
fi
```

多分支if示例

```
#!/bin/bash
# 多if语句的使用场景
if [ "$1" == "nan" ]
then
    echo "您的性别是 男"
elif [ "$1" == "nv" ]
then
    echo "您的性别是 女"
else
    echo "您的性别，我不知道"
fi
```

综合实践

需求：

要求脚本执行需要有参数，通过传入参数来实现不同的功能。

参数和功能详情如下：

参数	执行效果
start	服务启动中...
stop	服务关闭中...
restart	服务重启中...
*	脚本 X.sh 使用方式 X.sh [start or stop or restart]

脚本内容

```
#!/bin/bash
# 多if语句的使用场景
if [ "$1" == "start" ]
then
   echo "服务启动中..."
elif [ "$1" == "stop" ]
then
   echo "服务关闭中..."
elif [ "$1" == "restart" ]
then
   echo "服务重启中..."
else
   echo "$0 脚本的使用方式：  $0 [ start | stop | restart ]"
fi
```

小结

```

```

case条件

学习目标：了解 bash的case条件控制。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

基本格式

```
case 变量名 in
   值1)
      指令1
         ;;
   ...
   值n)
       指令n
         ;;
esac
```

注意：

```
首行关键字是case，末行关键字esac
选择项后面都有 )
每个选择的执行语句结尾都有两个分号;
```

简单实践

需求：

```
要求脚本执行需要有参数，通过传入参数来实现不同的功能。
```

参数和功能详情如下：

参数	执行效果
start	服务启动中...
stop	服务关闭中...
restart	服务重启中...
*	脚本 X.sh 使用方式 /bin/bash X.sh [start or stop or restart]

脚本内容

```
# cat case.sh

#!/bin/bash
# case语句使用场景
case "$1" in
    "start")
        echo "服务启动中..."
        ;;
    "stop")
        echo "服务关闭中..."
        ;;
    "restart")
        echo "服务重启中..."
        ;;
    *)
        echo "$0 脚本的使用方式:  $0 [ start  or stop  or restart ]"
        ;;
esac
```

小结

```

```

循环控制

学习目标：了解 bash的多种配置文件。

这一节，我们从 for循环、while循环、until循环、小结 四个方面来学习。

for循环

语法格式

```
for 条件
do
    执行语句
done
```

特点：

```
for语句，循环的数量有列表中元素个数来决定
```

示例

```
#!/bin/bash
# for语句的使用示例
for i in $(ls /root)
do
  echo "${i}"
done
```

while循环

语法格式

```
while 条件
do
    执行语句
done
```

特点：

```
while语句，只有条件不满足的时候，才会终止循环
```

示例

```
#!/bin/bash
# while语句的使用示例
a=1

while [ $a -lt 10 ]
do
  echo "${a}"
  let a=a+1
done
```

until循环

语法格式

```
until 条件
do
    执行语句
done
```

特点：

```
until语句，只有条件满足的时候，才会终止循环
```

示例

```
#!/bin/bash
# until语句的使用示例
a=1

until [ $a -eq 10 ]
do
  echo "${a}"
  let a=a+1
done
```

小结

```
```

循环退出

学习目标：了解 bash的循环退出方式。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

需求

```
我们在进行循环控制的时候，有时候，需要根据实际情况，终止循环或者跳过当前循环，这就用到了循环退出的基本功能。
```

基本语法

```
break
    条件满足的时候，终止执行后面的所有循环
continue
    条件满足的时候，跳过当前的循环，进入到下一个循环
```

简单实践

break实践

```
#!/bin/bash
```

```
# while语句的使用示例
a=1

while [ $a -lt 10 ]
do
  if [ $a == 5 ]
  then
    echo "终止后续所有循环"
    break
  fi
    echo "${a}"
    let a=a+1
done
```

break效果

```
1
2
3
4
终止后续所有循环
```

continue实践

```
#!/bin/bash
# while语句的使用示例
a=1

while [ $a -lt 10 ]
do
  let a=a+1
  if [ $a == 5 ]
  then
    echo "跳过当前循环，执行下一循环"
    continue
  fi
  echo "${a}"
done
```

continue效果

```
2
3
4
跳过当前循环，执行下一循环
6
7
8
9
10
```

小结

```

```

函数基础

学习目标：了解 bash的函数功能。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

简介

```
函数就是将某些命令组合起来实现某一特殊功能的方式，是脚本编写中非常重要的一部分。
```

简单函数格式：

定义函数：

```
function 函数名(){
    函数体
}
```

调用函数：

```
函数名
```

注意：function 关键字可以省略。

传参函数格式:

定义格式：

```
函数名(){
    函数体 $n
}
```

调用函数：

```
函数名 参数
```

状态返回值

```
函数名(){
    函数体 $n
    return 1
}
```

简单实践

简单 函数调用

```
#!/bin/bash
# 函数传参演示

# 定义传参数函数
dayin(){
  echo "wode mignzi shi zhangsan"
}

# 函数传参
dayin
```

传参 函数调用

```
#!/bin/bash
# 函数传参演示

# 定义传参数函数
dayin(){
  echo "wode mignzi shi $1"
}

# 函数传参
dayin 张三
```

函数返回值

```
#!/bin/bash
# 函数传参演示

# 定义传参数函数
dayin(){
  echo "wode mignzi shi $1"
  return 3
}

# 函数传参
dayin 张三
echo "函数执行的状态返回值: $?"
```

小结

```

```

函数进阶

学习目标：了解 bash的多种配置文件。

这一节，我们从 基础知识、简单实践、小结 三个方面来学习。

基础知识

脚本传参 函数调用：

脚本传参数

```
/bin/bash 脚本名 参数
```

函数体调用参数：

```
函数名(){
    函数体 $1
}
函数名 $1
```

脚本传参 函数调用(生产用)

脚本传参数

```
/bin/bash 脚本名 参数
```

函数体调用参数：

```
本地变量名 = "$1"
函数名(){
    函数体 $1
}
函数名 "${本地变量名}"
```

注意：类似于shell内置变量中的位置参数

简单实践

脚本传参 函数调用

```
#!/bin/bash
# 函数传参演示

# 定义传参数函数
dayin(){
  echo "wode mignzi shi $1"
}

# 函数传参
dayin $1
```

脚本传参，函数调用(生产用)

```
#!/bin/bash
# 函数的使用场景二
canshu = "$1"
dayin(){
  echo "wo de mingzi shi $1"
}
dayin "${canshu}"
```

小结

```
```