持续交付系列视频

编程知识体系学习

(Shell编程)

作者：传智播客 王老师

版本：V 1.0

文档编号：【20190328】

日期：2018年3月28日

目录

第 1 章 shell基础

1.1 运维&shell[了解]

学习目标：

了解 运维和自动化运维是什么及工作实现方式
记住 shell是什么，说出shell的两分类
知道 shell脚本的特点是什么

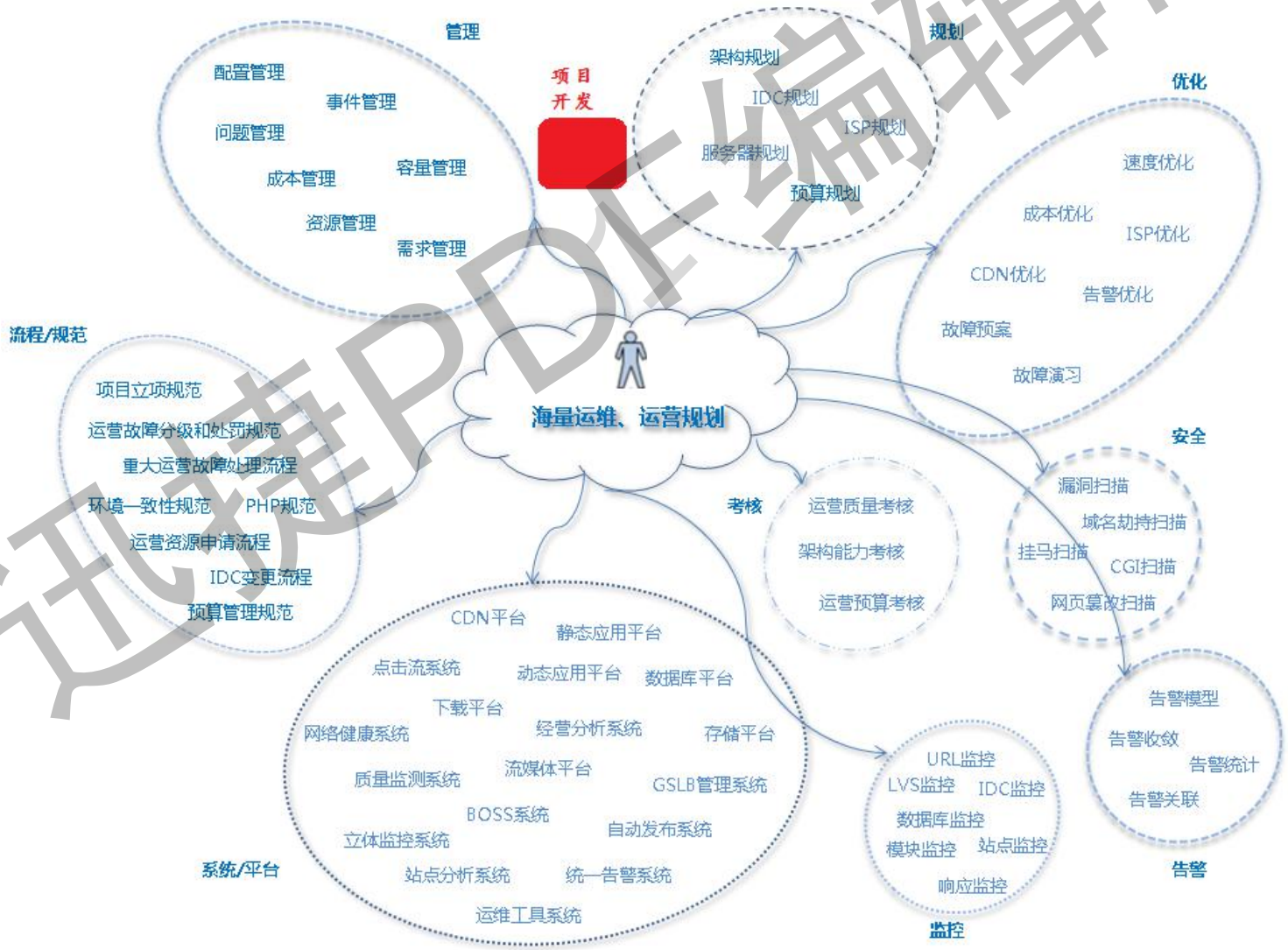
1.1.1 运维基础

这一节，我们从运维定位、工作范围、运维&shell 三个方面来学习。

运维定位

运维是什么，其实就是公司的一种技术岗位

工作范围



以美多商城、黑马头条项目为例：

规划：我们需要多少资源来支持项目的运行
管理：项目运行过程中的所有内容都管理起来
流程规范：所有操作都形成制度，提高工作效率
平台：大幅度提高工作效率

监控：实时查看项目运行状态指标
告警：状态指标异常，告知工作人员处理
安全：网站运营安全措施
优化：保证用户访问网站体验很好
考核：权责分配，保证利益

运维&shell

自动化运维：就是将图里面所有的工作都使用自动化的方式来实现。
实现自动化的方式很多，常见的方式：工具和脚本。
工作中常见的脚本有哪些呢？

shell脚本 和 其他开发语言脚本

注意：

shell脚本就是shell编程的一种具体实现

1.1.2 shell简介

这一节，我们从 shell定位、shell分类、使用方式 三个方面来学习。

shell定位

既然我们是来学shell，首先第一个问题：shell是什么？

shell的定义

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。

shell位置图

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。

shell的分类

shell分类

基本上shell分两大类：图形界面shell和命令行shell

图形界面shell

图形界面shell就是我们常说的桌面

命令行式shell

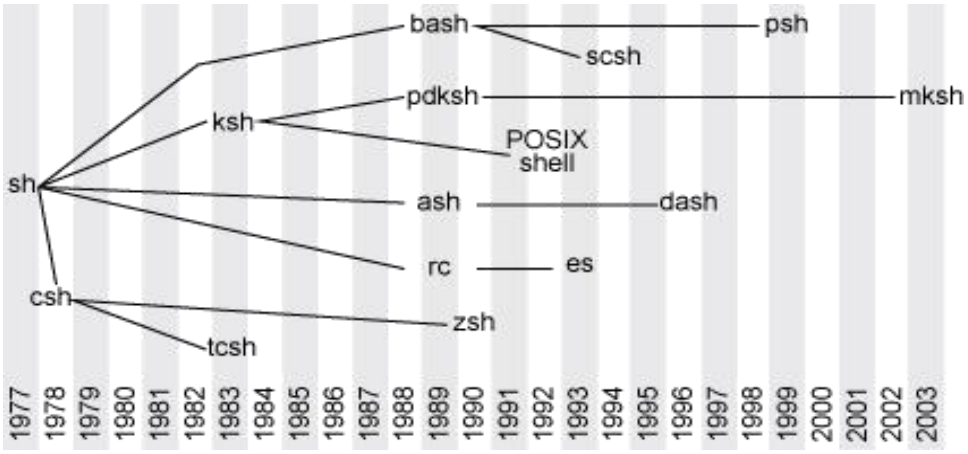
windows系统：

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。



查看系统shell信息

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
```

使用方式【记住】

shell使用方式主要有两种：手工和脚本

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.

重点：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。

重点：shell脚本就是linux命令的组合

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 'itcast'
```

脚本执行效果

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

1.2 shell脚本[记忆]

学习目标：

了解 shell脚本的内容小技巧

掌握 shell脚本的标准执行方法

掌握 shell脚本的开发规范和开发小技巧

1.2.1 创建脚本

这一节，我们从 创建方式、脚本命名、脚本内容、注释 四个方面来学习。

创建方式

创建脚本的常见编辑器是 vi/vim.

脚本命名

shell脚本的命名简单来说就是要**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

注释内容：

单行注释：

除了首行的#不是注释外，其他所有行内容，只要首个字符是#,那么就表示该行是注释

```
#!/bin/bash
echo '1'
# echo '2'          # 这一行就表示注释
echo '3'
```

多行注释：

多行注释有两种方法：:<<! ... ! 和 :<<字符 ... 字符

```
#!/bin/bash
echo '1'
:<<! echo '2'
echo '3'
echo '4'
!
echo '5'
```

1.2.2 脚本使用

这一节 我们从 执行方式、开发规范 两个方面来学习。

执行方式

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）
```

执行说明：

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

脚本开发规范

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)

1.3 变量[应用]

学习目标：

说出 变量的定义和目的

掌握 本地变量的查看格式和使用场景

掌握 全局变量的查看/定义格式和使用场景

掌握 普通变量的标准查看方法

说出 shell内置变量的特性和分类

掌握 shell内置变量的使用场景

1.3.1 什么是变量

这一节，我们从 变量定位、变量分类 两个方面来学习。

变量定位

变量包括两部分： 变量名=变量值

变量名 不变的

变量值 变化的

我们一般所说的变量指的是：变量名

变量分类

shell 中的变量分为三大类：本地变量、全局变量、内置变量

本地变量：手工方式定义的作用范围小的变量

全局变量：手工|默认方式定义作用范围大的变量

内置变量：bash命令中自带的一些参数变量

1.3.2 本地变量

这一节，我们从 本地变量、普通变量、命令变量 三个方面来学习。

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

普通变量的定义方式有如下三种，接下来我们就分别说一下这三种方式：

方式一：　　变量名=变量值

重点：

变量值必须是一个整体，中间没有特殊字符

方式二：　　变量名='变量值'

重点：

我看到的内容，我就输出什么内容

方式三：　　变量名="变量值"

重点：

如果变量值范围内，有可以解析的变量A，那么首先解析变量A，将A的结果和其他内容组合成一个整体，重新赋值给变量B

习惯：

数字不加引号，其他默认加双引号

命令变量

命令变量有两种定义方式，接下来我们就来介绍一下这两种方式

定义方式一：变量名=`命令`

注意：

` 是反引号

定义方式二：**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

1.3.3 全局变量

这一节，我们从 全局变量、查看命令、定义方式 三个方面来学习。

全局变量

全局变量就是：在当前系统的所有环境下都能生效的变量。

查看命令

可以通过命令查看环境变量

env　　**只显示全局变量**

定义方式

方法一：

变量=值

export 变量

方法二：（最常用）

export 变量=值

1.3.4 查看&删除

这一节，我们从 查看变量、删除变量 两个方面来学习。

查看变量：

方式一：　　$变量名

场景：

私下里，在命令行/脚本中使用

图省事

方式二：　　"$变量名"

场景：

私下里，在命令行/脚本中使用

图省事

方式三：　　${变量名}

场景：

echo " dsa dsafsa dsafsa ${变量名} f "

使用频率较高

方式四：　　**"${变量名}"**

场景：

标准使用方式

取消变量

unset 变量名

1.3.5 内置变量

这一节，我们从 内置变量、脚本文件、精确获取、默认值 四个方面来学习。

内置变量

我们之前学习的本地变量，全局变量都是需要通过定义，然后才能实现相应功能的，那么有没有一些变量我们可以直接拿过来使用实现某种具体的功能呢？有，这就是shell内置变量

脚本文件

符号	意义
$0	获取当前执行的shell脚本文件名
$n	获取当前执行的shell脚本的第n个参数值，n=1..9，当n为0时表示脚本的文件名，如果n大于9就要用大括号括起来${10}
$#	获取当前shell命令行中参数的总个数
$?	获取执行上一个指令的返回值（0为成功，非0为失败）

重点内置变量演示效果：

$0　　获取脚本的名称

示例：

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

$n　　获取当前脚本传入的第n个位置的参数

示例：

```
#!/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"
echo "第三个位置的参数是： $3"
echo "第四个位置的参数是： $4"
```

$#　　获取当前脚本传入参数的数量

示例：

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

$?　　获取文件执行或者命令执行的返回状态值

示例：

```
# bash nihao
bash: nihao: No such file or directory
# echo $?
127

# ls
file1.sh  num.sh  test.sh  weizhi.sh
# echo $?
0
```

精确截取

格式：${变量名:起始位置:截取长度}

示例：

```
${file:0:5}　　从第1个字符开始，截取5个字符
${file:5:5}　　从第6个字符开始，截取5个字符
${file:0-6:3}　从倒数第6个字符开始，截取之后的3个字符
```

默认值

场景一：

变量a如果有内容，那么就输出a的变量值

变量a如果没有内容，那么就输出默认的内容

格式：

```
${变量名:-默认值}
```

套餐示例：

如果我输入的参数为空，那么输出内容是 "您选择的套餐是： 套餐 1"

如果我输入的参数为n，那么输出内容是 "您选择的套餐是： 套餐 n"

```
#!/bin/bash
# 套餐选择演示
a="$1"
echo "您选择的套餐是： 套餐 ${a:-1}"
```

场景二：

无论变量a是否有内容，都输出默认值

格式：

${变量名+默认值}

场景示例：

不管我说国家法定结婚年龄是 多少岁，都输出 国家法定结婚年龄(男性)是 22 岁

```
#!/bin/bash
# 默认值演示示例二
a="$1"
echo "国家法定结婚年龄(男性)是 ${a+22} 岁"
```

第 2 章 核心知识[应用]

2.1 表达式

学习目标：

了解 常用的两种测试语句
掌握 标准测试语句的格式和特点
说出 常用条件表达式种类
掌握 11种表达式的特点和使用场景
应用 2种计算表达式
应用 数组使用方式

2.1.1 测试语句

这一节，我们从 应用场景、语句格式 两个方面来学习。

应用场景

Shell环境根据命令执行后的返回状态值($?)来判断是否执行成功,当返回值为0,表示成功,值为其他时,表示失败。使用专门的测试工具---test命令，可以对特定条件进行测试，并根据返回值来判断条件是否成立（返回值0为成立）

语句格式

A: test 条件表达式

B: [条件表达式]

格式注意：

以上两种方法的作用完全一样，后者为常用。

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。

test跟 [] 的意思一样

条件成立，状态返回值是0

条件不成立，状态返回值是1

操作注意：

[]两侧为什么要有空格

```
root@ubuntu:~# [-x /bin/bash ]
[-x: command not found
```

可以看到：

两侧没有空格，就会报错，为什么呢？因为你不合规范

2.1.2 条件表达式

这一节，我们从 逻辑表达式、文件表达式、数值表达式、字符串表达式 四个方面来学习。

逻辑表达式

逻辑表达式一般用于判断多个条件之间的依赖关系。

常见的逻辑表达式有： && 和 ||

&&

命令1 && 命令2

如果命令1执行成功，那么我才执行命令2 -- 夫唱妇随

如果命令1执行失败，那么命令2也不执行

示例：

```
# [ 1 = 1 ] && echo "条件成立"
条件成立
# [ 1 = 2 ] && echo "条件成立"
#
```

||

命令1 || 命令2

如果命令1执行成功，那么命令2不执行 -- 对着干

如果命令1执行失败，那么命令2执行

示例：

```
# [ 1 = 2 ] || echo "条件不成立"
条件不成立
# [ 1 = 1 ] || echo "条件不成立"
#
```

文件表达式

-f 判断输入内容是否是一个文件

示例：

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

-d 判断输入内容是否是一个目录

示例：

```
# [ -d weizhi.sddh ] || echo "不是一个目录"
不是一个目录
# mkdir nihao
# [ -d nihao ] && echo "是一个目录"
是一个目录
```

-x 判断输入内容是否可执行

示例：

```
# [ -x age.sh ] || echo "文件没有执行权限"
文件没有执行权限
# [ -x test.sh ] && echo "文件有执行权限"
文件有执行权限
```

数值操作符

主要根据给定的两个值，判断第一个与第二个数的关系，如是否大于、小于、等于第二个数。常见选项如下：

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

字符串比较

```
str1 == str2          str1和str2字符串内容一致
str1 != str2          str1和str2字符串内容不一致，!表示相反的意思
```

实践

判断字符是否内容一致

```
root@ubuntu:~# [ a == a ]
root@ubuntu:~# echo $?
0
root@ubuntu:~# [ a != a ]
root@ubuntu:~# echo $?
1
```

2.1.3 计算表达式

这一节，我们从 使用场景、计算格式、计算实践 三个方面来学习。

使用场景

计算表达式，简单来说就是对具体的内容进行算数计算

计算格式

方式一：

$(())　　$((计算表达式))

方式二：

let　　let 计算表达式

注意：

$(())中只能用+-*/和()运算符，并且只能做整数运算

计算实践

$(())演示效果

格式：a=$((变量名a+1))

注意：

表达式范围内，空格不限制

计算示例

```
root@ubuntu:~# echo $((100/5))
20
```

let演示效果

格式：let　变量名a=变量名a+1

注意：

表达式必须是一个整体，中间不能出现空格等特殊字符

let示例

```
root@ubuntu:~# i=1
root@ubuntu:~# let i=i+7
root@ubuntu:~# echo $i
```

```
8
```

2.1.4 数组操作

这一节，我们从 数组简介、定义格式、信息查看、增删改 四个方面来学习。

数组简介

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

定义格式

在Shell中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的语法格式：

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

注意：

基于元素的格式，主要有单行定义、多行定义、单元素定义、命令定义等

示例：

单行定义

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

多行定义

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

单元素定义

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

注意：

单元素定义的时候，可以不使用连续的下标，而且下标的范围没有限制。

命令定义就是value的值以命令方式来获取

```
file_array=($(ls /tmp/))
```

信息查看

基于索引找内容

读取数组元素值可以根据元素的下标值来获取，语法格式如下：

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

注意：

获取具体的元素内容，指定其下标值，从0开始

获取所有的元素内容，下标位置写"@"或者"*"

代码示例：

```
echo ${array_name[1]}
echo ${array_name[@]}
echo ${array_name[*]}
```

基于内容找索引

在找内容的时候，有时候不知道数组的索引都有哪些，我们可以基于如下方式来获取，数组的所有索引：

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

注意：

获取所有的元素内容，下标位置写"@"或者"*"

代码示例

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

获取长度

获取数组长度的方法与获取字符串长度的方法相同，格式如下：

${#array_name[index]}

注意：

获取具体的元素长度，指定其下标值，从0开始

获取所有的元素个数，下标位置写"@"或者"*"

代码示例：

```
echo ${#array_name[1]}
echo ${#array_name[@]}
echo ${#array_name[*]}
```

删改查

获取元素

数组元素的获取，有单元素获取，和元素部分内容获取。

单元素获取格式：

${array_name[index]}

元素部分内容的获取类似于字符串截取，格式如下：

${array_name[index]:pos:length}

代码示例

```
echo ${array_name[1]}
echo ${array_name[2]:0:2}
```

更改元素

数组元素的改其实就是定义数组时候的单元素定义，主要包含两种，元素替换，元素部分内容替换，格式如下

元素内容替换：

array_name[index]=值

注意：

在修改元素的时候，index的值一定要保持准确

元素部分内容替换，可以参考字符串替换格式：

${array_name[index]/原内容/新内容}

注意：

默认是演示效果，原数组未被修改，如果真要更改需要结合单元素内容替换

代码示例

```
array_name[1]=444
echo ${array_name[2]/va/hahhah}
```

删除数组

将shell中的数组删除，可以使用unset来实现，主要有两种情况：删除单元素，删除整个数组。

删除单元素

unset array_name[index]

删除整个数组

unset array_name

代码示例

```
unset array_name[1]
unset array_name
```

2.2 linux常见符号

学习目标：

掌握 两种重定向符号的格式和使用场景
掌握 管道符的特点和使用
掌握 后台展示符号的意义和作用
掌握 信息符号的作用和使用场景

2.2.1 重定向

这一节，我们从 重定向、>符号、>>符号 三个方面来学习。

重定向

在shell脚本中有两种常见的重定向符号 > 和 >>

> 符号

作用：

> 表示将符号左侧的内容，以覆盖的方式输入到右侧文件中

演示：

查看文件内容

```
root@centos ~]# cat file.txt
nihao
```

使用重定向符号给文件中增加内容

```
root@centos ~]# echo "file1.txt" > file.txt
```

再次查看文件内容

```
root@centos ~]# cat file.txt
file1.txt
```

>> 符号

作用：

>> 表示将符号左侧的内容，以追加的方式输入到右侧文件的末尾行中

演示：

查看文件内容

```
root@centos ~]# cat file.txt
file1.txt
```

使用重定向符号给文件中增加内容

```
root@centos ~]# echo "file2.txt" >> file.txt
```

再次查看文件内容

```
root@centos ~]# cat file.txt
file1.txt
file2.txt
```

2.2.2 管道符 |

这一节，我们从 定义、使用格式、命令演示 三个方面来学习。

定义：

| 这个就是管道符，传递信息使用的

使用格式：

命令1 | 命令2

管道符左侧命令1 执行后的结果，**传递**给管道符右侧的命令2使用

命令演示：

查看当前系统中的全局变量SHELL

```
root@centos ~]# env | grep SHELL
SHELL=/bin/bash
```

2.2.3 其他符号

这一节，我们从 后台展示、信息符号、特殊设备 三个方面来学习。

后台展示

定义：

& 就是将一个命令从前台转到后台执行

使用格式：

命令 &

命令演示：

```
admin-1@ubuntu:~# sleep 4
界面卡住 4 秒后消失
admin-1@ubuntu:~# sleep 10 &
[1] 4198
admin-1@ubuntu:~# ps aux | grep sleep
root       4198  0.0  0.0   9032   808 pts/17   S    21:58   0:00 sleep 10
root       4200  0.0  0.0  15964   944 pts/17   S+   21:58   0:00 grep --color=auto sleep
```

信息符号

符号详解：

1 表示正确输出的信息

2 表示错误输出的信息

2>&1 代表所有输出的信息

符号示例

标准正确输出示例

```
cat nihao.txt 1>> zhengque
```

标准错误输出示例

```
dsfadsfadsfa 2>> errfile
```

综合演练示例

脚本内容

```
#!/bin/bash
echo '下一条错误命令'
dsfsafsafdsa
```

脚本执行效果

```
admin-1@ubuntu:~# bash ceshi.sh
下一条错误命令
ceshi.sh: line 3: dsfsafsafdsa: command not found
```

1 和 2 综合演练

```
admin-1@ubuntu:~# bash ceshi.sh 1>> ceshi-ok 2>> ceshi-err
admin-1@ubuntu:~# cat ceshi-ok
下一条错误命令
```

```
admin-1@ubuntu:~# cat ceshi-err
ceshi.sh: line 3: dsfsafsafdsa: command not found
```

全部信息演练

```
admin-1@ubuntu:~# bash ceshi.sh >> ceshi-all 2>&1
admin-1@ubuntu:~# cat ceshi-all
下一条错误命令
ceshi.sh: line 3: dsfsafsafdsa: command not found
```

特殊设备

/dev/null 是linux下的一个设备文件，

这个文件类似于一个垃圾桶，特点是：容量无限大

2.3 简单流程控制

学习目标

了解 3种if语句和case的格式

掌握 双分支if语句和case语句的应用

掌握 3种循环语句的格式和应用

掌握 4种循环退出语句

2.3.1 if语句

这一节，我们从 单分支、双分支、多分支、多分支应用 四个方面来学习。

单分支

语法格式

```
if [ 条件 ]
then
    指令
fi
```

场景：

单一条件，只有一个输出

单分支if语句示例

```
#!/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

语法格式

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

场景：

一个条件，两种结果

双分支if语句示例

```
#!/bin/bash
# 单 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

语法格式

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

场景：

n个条件，n+1个结果

多分支if语句示例

```
#!/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
```

多分支应用

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|stop|restart]

脚本内容

```
admin-1@ubuntu:/data/scripts/python-n# cat if.sh
#!/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
```

2.3.2 case语句

这一节，我们从 语句格式、语句示例 两个方面来学习。

语句格式

```
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;

语句示例

场景：在多if语句的基础上对脚本进行升级

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|stop|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 | stop | restart ]"
        ;;
esac
```

2.3.3 循环语句

这一节，我们从 for语句、while语句、until语句 三个方面来学习.

for语句

语法格式

语法格式

```
for 值 in 列表
do
   执行语句
done
```

场景：

遍历列表

注意：

”for” 循环总是接收 “in” 语句之后的某种类型的字列表

执行次数和list列表中常数或字符串的个数相同，当循环的数量足够了，就自动退出

示例：遍历文件

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

while语句

语法格式

```
while 条件
do
   执行语句
done
```

注意：

条件的类型：

命令、[[字符串表达式]]、((数字表达式))

场景：

只要条件满足，就一直循环下去

语句示例

脚本内容

```
#!/bin/bash
# while 的示例
a=1
while [ "${a}" -lt 5 ]
do
   echo "${a}"
   a=$((a+1))
done
```

until语句

语法格式

```
until 条件
do
   执行语句
done
```

注意：

条件的类型：

命令、[[字符串表达式]]、((数字表达式))

场景：

只要条件不满足，就一直循环下去

until语句示例

脚本内容

```
#!/bin/bash
# until 的示例
a=1
until [ "${a}" -eq 5 ]
do
   echo "${a}"
   a=$((a+1))
done
```

2.3.4 循环退出

这一节，我们从 退出简介、break示例、break n示例、continue示例、exit示例 五个方面来学习。

退出简介

对于某些循环情况，我们需要在特定的条件下退出，主要有以下指令来实现：break、break n、continue

指令详解：

break　　跳出所有循环

break n　　跳出第n个循环(由内向外)

continue 跳出当前循环
exit 退出程序

break示例

需求：

脚本进入死循环直至用户输入数字大于5

代码示例：

```
#!/bin/bash
while :
do
    echo -n "输入你的数字，最好在 1 ~ 5: "
    read aNum
    case $aNum in
        1|2|3|4|5)
          echo "你的数字是 $aNum!"
        ;;
        *)
          echo "你选择的数字没在 1 ~ 5，退出!"
          break
        ;;
    esac
done
```

效果示例：

```
~]# /bin/bash while.sh
输入你的数字，最好在 1 ~ 5: 1
你的数字是 1!
输入你的数字，最好在 1 ~ 5: 6
你选择的数字没在 1 ~ 5，退出!
```

break n

需求：

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

如果 var1 等于 2，并且 var2 等于 0，就跳出循环

代码示例：

```
#!/bin/bash
for var1 in {1..5}
do
   for var2 in {0..4}
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

continue命令

需求：

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。

代码示例：

```
#!/bin/bash
while :
do
    echo -n "输入你的数字，最好在 1 ~ 5: "
    read aNum
    case $aNum in
        1|2|3|4|5)
          echo "你的数字是 $aNum!"
        ;;
        *)
          echo "你选择的数字没在 1 ~ 5，退出!"
          continue
        ;;
    esac
done
```

效果显示：

```
~]# /bin/bash while.sh
输入你的数字，最好在 1 ~ 5: 6
你选择的数字没在 1 ~ 5，退出!
输入你的数字，最好在 1 ~ 5:
```

exit示例

需求

在嵌套循环中，exit 命令表示退出当前程序

如果 var1 等于 2，并且 var2 等于 0，就跳出循环

代码实践

```
#!/bin/bash
for var1 in {1..5}
do
   for var2 in {0..4}
   do
     if [ $var1 -eq 2 -a $var2 -eq 0 ]
     then
       exit
     else
        echo "$var1 $var2"
     fi
   done
done
```

2.4 复杂流程控制

学习目标：

掌握 简单函数定义和调用格式及场景

掌握 传参函数定义和调用格式及场景

掌握 函数4层次的使用场景

2.4.1 函数基础知识

这一节，我们从 函数定义、函数样式 两个方面来学习。

函数定义

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。

函数样式

简单函数格式：

定义函数：

```
函数名(){
    函数体
}
```

调用函数：

```
函数名
```

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
```

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

2.4.2 函数实践

这一节，我们从 四种函数样式 来进行相应的学习。

简单函数定义和调用示例

```
#!/bin/bash
# 函数使用场景一：执行频繁的命令
dayin(){
  echo "wo de mingzi shi  111"
}
dayin
```

函数传参和函数体内调用参数示例

```
#!/bin/bash
# 函数的使用场景二
dayin(){
  echo "wo de mingzi shi $1"
}
dayin 111
```

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}"
```

2.5 常见命令详解

本节学习目标：

掌握 grep 命令的格式和常用参数意义和使用场景
掌握 sed 命令的格式和常用参数意义和使用场景
掌握 awk 命令的格式和常用参数意义和使用场景
掌握 find 命令的格式和常用参数意义和使用场景

2.5.1 grep命令详解

这一节，我们从 命令格式、命令实践 两个方面来学习。

命令格式

grep命令是我们常用的一个强大的文本搜索命令。
grep [参数] [关键字] <文件名>
注意：
我们在查看某个文件的内容的时候，是需要有<文件名>
grep命令在结合|(管道符)使用的情况下，后面的<文件名>是没有的
可以通过 grep --help 查看grep的帮助信息

参数详解

-c：只输出匹配行的计数。
-n：显示匹配行及行号。
-v：显示不包含匹配文本的所有行。

命令实践

模板文件

```
root@centos ~]# cat find.txt
nihao aaa
nihao AAA
NiHao bbb
nihao CCC
```

-c： 输出匹配到aaa的个数

```
root@centos ~]# grep -c aaa find.txt
1
```

-n： 输出匹配内容，同时显示行号

```
root@centos ~]# grep -n CCC find.txt
4:nihao CCC
```

-v: 匹配到的内容部输出，输出不匹配的内容

```
root@centos ~]# grep -v ni find.txt
NiHao bbb
```

小技巧：

精确定位错误代码

```
grep -nr [错误关键字] *
```

2.5.2 sed命令详解

这一节，我们从 格式详解、查看实践、替换实践、增加实践、删除实践 五个方面来学习。

格式详解

sed 行文件编辑工具。因为它编辑文件是以行为单位的。

命令格式：

```
sed [参数] '<匹配条件> [动作]' [文件名]
```

注意：

可以通过 sed --help 查看帮助信息

参数详解：

参数为空	表示sed的操作效果，实际上不对文件进行编辑
-n	取消静默输出
-i	表示对文件进行编辑
	注意：mac版本的bash中使用 -i参数，必须在后面单独加个东西： -i ''

匹配条件：

匹配条件分为两种：数字行号或者关键字匹配

关键字匹配格式：

```
'/关键字/'
```

注意：

隔离符号 / 可以更换成 @、#、！等符号

根据情况使用，如果关键字和隔离符号有冲突，就更换成其他的符号即可。

动作详解

a	在匹配到的内容下一行增加内容
i	在匹配到的内容当前行增加内容
d	删除匹配到的内容
s	替换匹配到的内容
p	查看指定内容

注意：

上面的动作应该在参数为-i的时候使用，不然的话不会有效果

查看实践

模板文件内容

```
root@centos ~]# cat sed.txt
nihao sed1 sed2 sed3
nihao sed4 sed5 sed6
nihao sed7 sed8 sed9
```

查看第2行的内容

```
root@centos ~]# sed -n '3p' sed.txt
```

```
nihao sed7 sed8 sed9
```

打印2-3行的内容

```
root@centos ~]# sed -n '2,3p' sed.txt
nihao sed7 sed8 sed9
```

替换实践

关于替换，我们从三个方面来学习：

行号、列号、全体

命令格式：

sed -i [替换格式] [文件名]

注意：替换命令的写法

's###' ---> 's#原内容##' ---> 's#原内容#替换后内容#'

样式一：

sed -i '行号s#原内容#替换后内容#列号' [文件名]

样式二：

sed -i 's#原内容#替换后内容#g' [文件名]

常见替换格式：

替换每行首个匹配内容：

sed -i 's#原内容#替换后内容#' 文件名

示例：替换首每行的第1个sed为SED

```
root@centos ~]# sed -i 's#sed#SED#' sed.txt
root@centos ~]# cat sed.txt
nihao SED sed sed
nihao SED sed sed
nihao SED sed sed
```

替换全部匹配内容：

sed -i 's#原内容#替换后内容#g' 文件名

示例：替换全部sed为des

```
root@centos ~]# sed -i 's#sed#SED#g' sed.txt
root@centos ~]# cat sed.txt
nihao SED SED SED
nihao SED SED SED
nihao SED SED SED
```

指定行号替换首个匹配内容：

sed -i '行号s#原内容#替换后内容#' 文件名

示例：替换第2行的首个SED为sed

```
root@centos ~]# sed -i '2s#SED#sed#' sed.txt
root@centos ~]# cat sed.txt
nihao SED SED SED
nihao sed SED SED
nihao SED SED SED
```

首行指定列号替换匹配内容：

sed -i 's#原内容#替换后内容#列号' 文件名

示例：替换每行的第2个SED为sed

```
root@centos ~]# sed -i 's#SED#sed#2' sed.txt
root@centos ~]# cat sed.txt
nihao SED sed SED
nihao sed SED sed
nihao SED sed SED
```

指定行号列号匹配内容：

sed -i '行号s#原内容#替换后内容#列号' 文件名

示例：替换第3行的第2个SED为sed

```
root@centos ~]# sed -i '3s#SED#sed#2' sed.txt
root@centos ~]# cat sed.txt
nihao SED sed SED
nihao sed SED sed
nihao SED sed sed
```

增加实践

作用：

在指定行号的下一行增加内容

格式：

sed -i '行号a\增加的内容' 文件名

注意：

如果增加多行，可以在行号位置写个范围值，彼此间使用逗号隔开，例如

sed -i '1,3a\增加内容' 文件名

演示效果：

指定行号增加内容

```
root@centos ~]# sed -i '2a\zengjia-2' sed.txt
root@centos ~]# cat sed.txt
nihao SED sed SED
nihao sed SED sed
zengjia-2
nihao SED sed sed
```

指定1~3每行都增加内容

```
root@centos ~]# sed -i '1,3a\tongshi-2' sed.txt
root@centos ~]# cat sed.txt
nihao SED sed SED
tongshi-2
nihao sed SED sed
tongshi-2
zengjia-2
tongshi-2
nihao SED sed sed
```

作用：

在指定行号的当行增加内容

格式：

sed -i '行号i\增加的内容' 文件名

注意：

如果增加多行，可以在行号位置写个范围值，彼此间使用逗号隔开，例如

sed -i '1,3i\增加内容' 文件名

演示效果：

指定行号增加内容

```
root@centos ~]# sed -i '1i\insert-1' sed.txt
root@centos ~]# cat sed.txt
insert-1
nihao SED sed SED
tongshi-2
nihao sed SED sed
tongshi-2
zengjia-2
tongshi-2
nihao SED sed sed
```

删除实践

作用：

指定行号删除

格式：

sed -i '行号d' 文件名

注意：

如果删除多行，可以在行号位置多写几个行号，彼此间使用逗号隔开，例如

sed -i '1,3d' 文件名

删除演练

删除第4行内容

```
root@centos ~]# sed -i '4d' sed.txt
root@centos ~]# cat sed.txt
insert-1
nihao SED sed SED
tongshi-2
tongshi-2
zengjia-2
tongshi-2
nihao SED sed sed
```

删除多行(3-5行)内容

```
root@centos ~]# sed -i '3,5d' sed.txt
root@centos ~]# cat sed.txt
insert-1
nihao SED sed SED
tongshi-2
nihao SED sed sed
```

2.5.3 awk命令详解

这一节，我们从 格式详解、简单实践、进阶实践 三个方面来学习。

格式详解

awk是一个功能非常强大的文档编辑工具，它不仅能以行为单位还能以列为单位处理文件。

命令格式

```
awk [参数] '[ 动作]' [文件名]
```

常见参数：

```
-F      指定列的分隔符
-f      调用脚本
-v      定义变量
```

常见动作：

```
print  显示内容
    $0      显示当前行所有内容
    $n      显示当前行的第n列内容，如果存在多个$n，它们之间使用逗号(,)隔开
动作组成
    BEGIN{ 命令 }          初始代码块，主要和变量相关
    /pattern/{ 命令 }      匹配、执行代码块
    END{ 命令 }            结束代码块，主要和信息输出相关
```

内置变量

```
FILENAME   当前输入文件的文件名，该变量是只读的
NR         指定显示行的行号
NF         输出 最后一列的内容
OFS        输出格式的列分隔符，缺省是空格
FS         输入文件的列分融符，缺省是连续的空格和Tab
```

简单实践

模板文件内容

```
root@centos ~]# cat awk.txt
nihao awk1 awk2 awk3
nihao awk4 awk5 awk6
```

print动作

过滤信息

过滤包含nihao的内容

```
root@centos ~]# awk '/nihao/' awk.txt
nihao awk1 awk2 awk3
nihao awk4 awk5 awk6
```

打印指定列内容

打印第1列的内容

```
root@centos ~]# awk '{print $1}' awk.txt
nihao
nihao
```

打印行号

打印内容时候，打印行号

```
root@centos ~]# awk '{print NR,$1}' awk.txt
```

```
1 nihao
2 nihao
```

注意：

NR在动作内部表示行号

指定行打印内容

打印第一行第1和第3列内容

```
root@centos ~]# awk  'NR==1 {print $1,$3}' awk.txt
nihao awk
```

-F参数

指定隔离分隔符，查看内容

```
root@centos ~]# cat linshi.txt
root:x:0:0:root:/root:/bin/bash
root@centos ~]# awk -F ':' '{print $1,$7}' linshi.txt
root /bin/bash
```

-f参数

编写脚本，格式化输出

```
root@centos ~]# cat filename
/ni/{print "第" NR "行:"  "内容:" $0}
```

加载脚本，按格式输出信息

```
root@centos ~]# awk -f filename awk.txt
第 1 行:内容:nihao awk1 awk2 awk3
第 2 行:内容:nihao awk4 awk5 awk6
```

注意：

使用格式： awk -f 脚本文件 内容文件

-v参数

命令行方式

传入单值

```
root@centos ~]# echo | awk -v var=100 '{print var}'
100
```

传入多值

```
root@centos ~]# echo | awk '{print v1,v2}' v1=1000 v2=2000
1000 2000
```

注意：

-v的使用，需要结合|符号才可以

格式化输出

设置显示分隔符，显示内容

```
root@centos ~]# awk 'BEGIN{OFS=":"} {print NR,$0}' awk.txt
1:nihao awk awk awk
2:nihao awk awk awk
```

设置输出样式

```
root@centos ~]# awk -F: 'BEGIN {print "----开始了----"} {print NR,$0} END {print "-----结 束了
"--}' awk.txt
----开始了----
1 nihao awk1 awk2 awk3
2 nihao awk4 awk5 awk6
```

```
-----结束了---
```

进阶实践

if语句示例

列出当前目录中大于500字节的信息

```
root@centos ~]# ls -l | awk '{if (($5>=500)) print "\n" "文件: " $9 "\n" "大小: " $5 "B" "\n"}
'


文件: anaconda-ks.cfg
大小: 988B
```

列出当前目录中大于500字节的普通文件信息

```
root@centos ~]# ls -l | awk '{if (($5>=500  && /^-/ )) print "\n" "文件: " $9 "\n" "大小: " $5
"B" "\n"}'


文件: anaconda-ks.cfg
大小: 988B
```

for语句示例

按顺序输出所有的内容

```
root@centos ~]# echo "abcde" | awk -F '' '{for(i=1;i<=NF;i++) print $i}'
a
b
c
d
e
```

按输入的倒序输出所有内容

```
[root@lvs-router ~]# echo "abcde" | awk -F '' '{for(i=NF;i>=1;i--) print $i}'
e
d
c
b
a
```

2.5.4 find命令详解

这一节，我们从 格式详解、命令实践 两个方面来学习。

格式详解

命令格式:

```
find [路径] [参数] [关键字] [动作]
```

参数详解

-name 按照文件名查找文件。

-user 按照文件属主来查找文件。

-group 按照文件所属的组来查找文件。

-type 查找某一类型的文件，

诸如:

b - 块设备文件　　d - 目录　　c - 字符设备文件

p - 管道文件　　l - 符号链接文件　　f - 普通文件。

-size n(K|M|G): [c]　　查找文件长度为n块的文件，带有c时表示文件长度以字节计。
-perm (/|-)权限　按照文件权限来查找文件。　/表示有一个权限匹配即可，-表示相反权限匹配才可
-mtime (-|+)n　查找n天数(内|外)修改的文件
-ctime (-|+)n　查找n天数(内|外)改变的文件
-atime (-|+)n　查找n天数(内|外)访问的文件
-depth　在查找文件时，首先查找当前目录中的文件，然后再在其子目录中查找。
-mindepth n　在查找文件时，查找当前目录中的第n层目录的文件，然后再在其子目录中查找。
-path "子目录"　在指定目录的子目录下查找，一般与-prune使用
-prune　在查找文件时，忽略指定的内容，不能和-depth共用，否则会自动忽视prune
-newer　查找比指定文件新的文件
! ：表示取反

动作详解

动作就表示，我们对查找出来的文件做进一步的操作，主要都动作有三个

-print　默认选项，显示名称，-o -print 表示不仅仅显示目录名，还显示目录里面的文件名
-ls　显示文件属性
-exec 命令 {} \;　使用命令对查找结果处理，查找结果使用"{}"来表示

简单实践

基本实践

在当前系统中查找一个叫awk的文件

```
root@centos ~]# sudo find /home/admin-1/ -name "awk.txt"
/home/admin-1/awk.txt
```

在当前系统中查找文件类型为普通文件的文件

```
root@centos ~]# find /tmp -type f
/tmp/.X0-lock
/tmp/vgauthsvclog.txt.0
/tmp/unity_support_test.0
/tmp/config-err-4igbXW
```

根目录下查找5日以内更改的文件

```
find / -mtime -5
```

在/tmp/目录下查找3日以前更改的文件，可以用：

```
find /tmp/ -mtime +3
```

在目录下查找不包含backup子目录

```
find /data/scripts -path "/data/scripts/backup" -prune -o -print
```

忽略多个文件夹

```
find . \( -path "./backup" -o -path "./backup2" \) -prune -o -print
```

动作实践

以列表方式查看查找到的文件

```
find /etc -perm -640 -ls
```

对查找到的文件进行改名

```
find ./ -perm -002 -exec mv {} {}.old \;
```

查找到的文件删除

```
find . -name .svn | xargs rm -rf
```

查找磁盘中大于3M的文件

```
find . -size +3000k -exec ls -ld {} ;
```