Linux基础之Shell编程进阶

讲师：王树森

1 shell编程进阶

1.1 数组实践

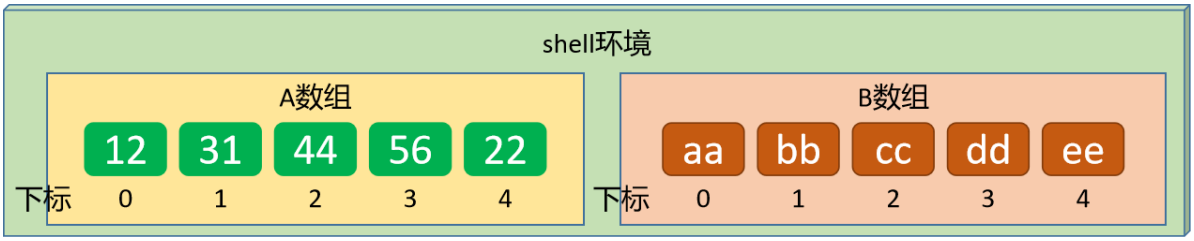
1.1.1 数组基础

基础知识

简介

> 数组（Array）是有序的元素序列，它是数据结构在shell当中非常常见的一种数据存储方式，它将有限个类型相同的数据放到一个集合中，这个集合就是数组。

> 为了操作方便，我们为数组定制一个名称变量，数组内的每一个数据都是数组元素，这些数组元素在集合中有顺序的观念，顺序的位置值我们称为下标。



数组分类

数组样式-从数据结构的本身出发，它主要有多种数组样式

一维数组

一维数组是最简单的数组，按照顺序将一系列数据排列下来即可，数组本身没有场景含义。

数据的表现样式：数组[下标]

适用场景：编程语言中基于数据的查询、聚合等操作

二维数组

二维数组是业务场景中常见的数组表现样式，即在一维数组的前提下，扩充了数据元素在场景中的含义。

数据的表现样式：数组[行下标][列下标]

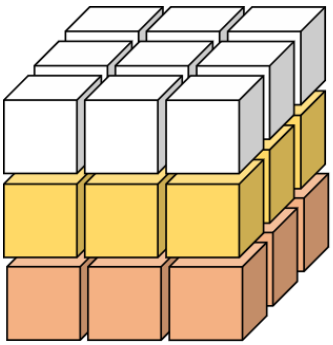
适用场景：数据库场景中基于数据的查询、聚合等操作

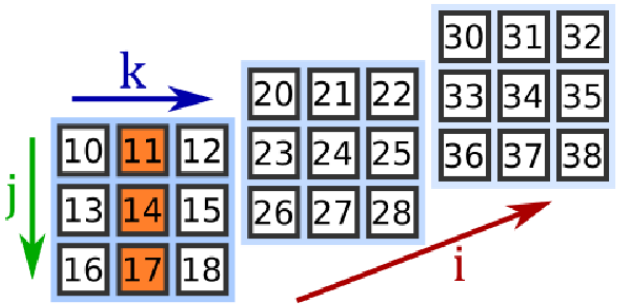
三维数组

三维数组是大型业务场景中通用的一种数组表现样式，它是在二维数据的前提下，扩充了数据空间的含义。

数据的表现样式：数组[行下标][列下标][页下标]

适用场景：数据分析场景中基于数据的查询、聚合等操作





注意：

1 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。

2 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

3 bash的数组支持稀疏格式（索引名称可以不连续）

1.1.2 数组定义

基础知识

数组创建

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、命令定义等多种样式

语法解读

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/))
```

简单实践

实践1-单行定义

```
定制数据数组
[root@rocky9 ~]# num_list=(123,234,345,456,567)
[root@rocky9 ~]# echo ${num_list[0]}
123,234,345,456,567

数据元素之间使用空格隔开
[root@rocky9 ~]# num_list=(123 234 345 456 567)
[root@rocky9 ~]# echo ${num_list[0]}
123
[root@rocky9 ~]# echo ${num_list[@]}
123 234 345 456 567
```

实践2-多行定义

```
定制数组
[root@rocky9 ~]# class_one=(
> zhangsan
> lisi
> wangwu
> zhaoliu
> )

查看数组元素
[root@rocky9 ~]# echo ${class_one[0]}
zhangsan
[root@rocky9 ~]# echo ${class_one[@]}
zhangsan lisi wangwu zhaoliu
```

实践3-单元素定义

```
定制数组
[root@rocky9 ~]# mix_list[0]=nihao
[root@rocky9 ~]# mix_list[2]=345
[root@rocky9 ~]# mix_list[4]="1.23,4.56"

查看数组元素
[root@rocky9 ~]# echo ${mix_list[1]}
[root@rocky9 ~]# echo ${mix_list[@]}
nihao 345 1.23,4.56
```

```
批量多元素定义
[root@rocky9 ~]# string_list=([0]="value-1" [3]="value-2")
[root@rocky9 ~]# echo ${string_list[@]}
value-1 value-2
[root@rocky9 ~]# echo ${!string_list[@]}
0 3
```

实践4-命令定义

```
定制数组元素
[root@rocky9 ~]# file_array=$(ls *.sh)

查看数组元素
[root@rocky9 ~]# echo ${file_array[0]}
count_head_feet.sh host_network_test.sh memory_info.sh simple_jumpserver.sh
simple_login.sh simple_login_string.sh site_healthcheck.sh test_argnum.sh
[root@rocky9 ~]# echo ${file_array[1]}
[root@rocky9 ~]# echo ${file_array[2]}
[root@rocky9 ~]# echo ${file_array[@]}
count_head_feet.sh host_network_test.sh memory_info.sh simple_jumpserver.sh
simple_login.sh simple_login_string.sh site_healthcheck.sh test_argnum.sh

注意：
    对于命令的数组创建来说，它只有一个元素
```

1.1.3 数组取值

基础知识

简介

```
    对于shell的数组数据来说，获取制定的数组元素主要有两种方法，一种是获取内容，一种是获取其他信息。
```

语法解读

```
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]}
        ${array_name[@]:起始位置:获取数量}
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"*"
```

```
获取数组索引
    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：
    获取所有的元素位置，下标位置写"@"或者"*"
```

```
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"*"
```

```
从系统中获取所有的数组
    declare -a
```

简单实践

实践1-基于索引找内容

```
设定数组内容
[root@rocky9 ~]# num_list=(123 234 345 456 567)

获取指定位置元素
[root@rocky9 ~]# echo ${num_list[0]}
123
[root@rocky9 ~]# echo ${num_list[1]}
234

获取所有位置元素
[root@rocky9 ~]# echo ${num_list[*]}
123 234 345 456 567
[root@rocky9 ~]# echo ${num_list[@]}
123 234 345 456 567

获取末尾位置元素
[root@rocky9 ~]# echo ${num_list[-1]}
567
[root@rocky9 ~]# echo ${num_list[-2]}
456

获取指定范围元素
[root@rocky9 ~]# echo ${num_list[@]:1:1}
234
[root@rocky9 ~]# echo ${num_list[@]:1:3}
234 345 456
```

实践2-基于内容获取元素

```
[root@rocky9 ~]# echo ${!num_list[@]}
0 1 2 3 4
[root@rocky9 ~]# echo ${!num_list[@]}
0 1 2 3 4
```

实践3-获取数组长度

```
获取数组的元素数量
[root@rocky9 ~]# echo ${#num_list[@]}
5
[root@rocky9 ~]# echo ${#num_list[*]}
5

获取数据元素的长度
[root@rocky9 ~]# echo ${#num_list[3]}
3
```

实践4-获取系统所有数组

```
设定数组
[root@rocky9 ~]# num_list=(123 234 345 456 567)

查看所有数组
[root@rocky9 ~]# declare -a
declare -a BASH_ARGC='()'
declare -a BASH_ARGV='()'
declare -a BASH_LINENO='()'
declare -a BASH_SOURCE='()'
declare -ar BASH_VERSINFO='([0]="4" [1]="2" [2]="46" [3]="2" [4]="release"
[5]="x86_64-redhat-linux-gnu")'
declare -a DIRSTACK='()'
declare -a FUNCNAME='()'
declare -a GROUPS='()'
declare -a PIPESTATUS='([0]="0")'
declare -a num_list='([0]="123" [1]="234" [2]="345" [3]="456" [4]="567")'
```

小结

1.1.4 数组变动

元素修改

简介

```
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
```

简单实践

```
修改指定位置的值
[root@rocky9 ~]# num_list[2]=aaa
[root@rocky9 ~]# echo ${num_list[@]}
123 234 aaa 456 567

替换元素值的特定内容
[root@rocky9 ~]# echo ${num_list[2]/aa/lualu-}
lualu-a
[root@rocky9 ~]# num_list[2]=${num_list[2]/aa/lualu-}
[root@rocky9 ~]# echo ${num_list[@]}
123 234 lualu-a 456 567
```

元素删除

简介

```
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。格式如下：
```

```
删除单元素
    unset array_name[index]
删除整个数组
    unset array_name
```

简单实践

```
删除指定的元素
[root@rocky9 ~]# echo ${num_list[@]}
123 234 lualu-a 456 567
[root@rocky9 ~]# unset num_list[2]
[root@rocky9 ~]# echo ${num_list[@]}
123 234 456 567
[root@rocky9 ~]# unset num_list[2]
[root@rocky9 ~]# echo ${num_list[@]}
123 234 456 567
[root@rocky9 ~]# unset num_list[1]
[root@rocky9 ~]# echo ${num_list[@]}
123 456 567
[root@rocky9 ~]# echo ${!num_list[@]}
0 3 4
```

```
替换元素值的特定内容
[root@rocky9 ~]# unset num_list
[root@rocky9 ~]# echo ${!num_list[@]}

[root@rocky9 ~]#
```

1.1.5 数组关联

基础知识

简介

```
    上一节，我们学习了shell环境下的数组定制的简写方式。数组的定制主要有如下两种：
定制索引数组 - 数组的索引是普通的数字
    declare -a array_name
    - 普通数组可以不事先声明,直接使用

定制关联数组 - 数组的索引是自定义的字母
    declare -A array_name
    - 关联数组必须先声明,再使用
```

简单实践

实践1-定制索引数组

```
定制一个空内容的数组
[root@rocky9 ~]# declare -a course
[root@rocky9 ~]# declare -a | grep course
declare -a course='()'

定制一个包含元素的数组
[root@rocky9 ~]# course=(yuwen shuxue yingyu)
[root@rocky9 ~]# declare -a | grep course
declare -a course='([0]="yuwen" [1]="shuxue" [2]="yingyu")'
```

实践2-定制关联数组

```
定制关联数组
[root@rocky9 ~]# declare -A score
[root@rocky9 ~]# declare -a | grep score
[root@rocky9 ~]# declare -A | grep score
declare -A score='()'

关联数组和数字索引数组不能通用
[root@rocky9 ~]# declare -a score
-bash: declare: score: 无法将关联数组转化为索引数组
```

```
关联数组的操作
[root@rocky9 ~]# score=([yingyu]="32" [yuwen]="67" [shuxue]="65")
[root@rocky9 ~]# declare -A | grep score
declare -A score='([yingyu]="32" [yuwen]="67" [shuxue]="65" )'
[root@rocky9 ~]# echo ${!score[@]}
yingyu yuwen shuxue
[root@rocky9 ~]# echo ${score[@]}
32 67 65
```

1.1.6 数组案例

信息统计

需求

```
分别打印CPU 1min 5min 15min load负载值
命令提示：
    uptime

信息显示：
    CPU 1 min平均负载为: 0.00
    CPU 5 min平均负载为: 0.01
    CPU 15 min平均负载为: 0.05
```

编写脚本

```
查看脚本内容
[root@rocky9 ~]# cat cpu_load.sh
#!/bin/bash
# 功能：采集系统cpu负载信息
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 获取CPU负载情况
cpu_load=($(uptime | tr -s " " | cut -d " " -f 9-11 | tr "," " "))

# 信息输出
echo -e "\e[31m\t系统cpu负载信息\e[0m"
echo -e "\e[32m================================="
echo "CPU 1 min平均负载为: ${cpu_load[0]}"
echo "CPU 5 min平均负载为: ${cpu_load[1]}"
echo "CPU 15 min平均负载为: ${cpu_load[2]}"
echo -e "=================================\e[0m"
```

```
脚本执行后效果
[root@rocky9 ~]# /bin/bash cpu_load.sh
        系统cpu负载信息
=================================
CPU 1 min平均负载为: 0.00
CPU 5 min平均负载为: 0.01
CPU 15 min平均负载为: 0.05
=================================
```

服务管理

需求

```
服务的管理动作有：
    "启动" "关闭" "重启" "重载" "状态"
服务的管理命令有：
    "start" "stop" "restart" "reload" "status"
选择不同的动作，输出不同的服务执行命令，格式如下：
    systemctl xxx service_name
```

编写脚本

```
[root@rocky9 ~]# cat service_manager.sh
#!/bin/bash
# 功能：定制服务管理的功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通数组
oper_array=(启动 关闭 重启 重载 状态)
# 定制关联数组
declare -A cmd_array
cmd_array=([启动]=start [关闭]=stop [重启]=restart [重载]=reload [状态]=status)

# 服务的操作提示
echo -e "\e[31m---------------服务的操作动作---------------
 1: 启动  2: 关闭  3: 重启  4: 重载  5: 状态
----------------------------------------------"'\033[0m'

# 选择服务操作类型
read -p "> 请输入服务的操作动作: " oper_num
echo
echo -e "\e[31m您选择的服务操作动作是:  ${oper_array[$oper_num-1]}\e[0m"
echo -e "\e[32m===============服务的执行动作===============
您即将对服务执行如下命令:
\tsystemctl ${cmd_array[${oper_array[$oper_num-1]}]} service_name
==============================================="'\033[0m'
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash service_manager.sh
---------------服务的操作动作---------------
 1: 启动  2: 关闭  3: 重启  4: 重载  5: 状态
----------------------------------------------
> 请输入服务的操作动作: 3

您选择的服务操作动作是:  重启
===============服务的执行动作===============
您即将对服务执行如下命令:
        systemctl restart service_name
==============================================
```

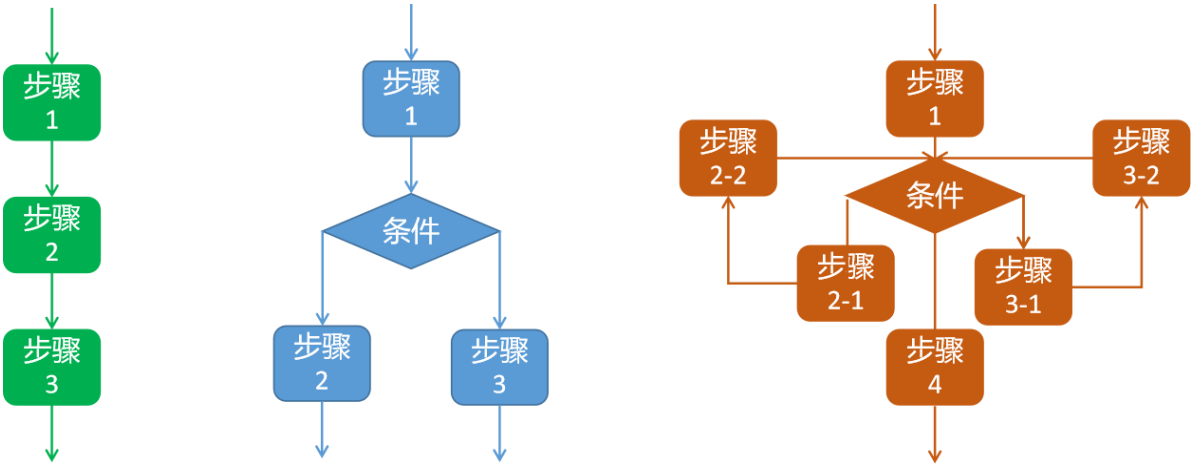
1.2 分支逻辑

1.2.1 流程基础

基础知识

编程逻辑

> 编程语言的目的是通过风格化的编程思路将代码写出来后，实现项目功能的。为了实现功能，我们通过在代码层面通过一些代码逻辑来实现：
>
> 顺序执行 - 程序按从上到下顺序执行
>
> 选择执行 - 程序执行过程中，根据条件选择不同的顺序执行
>
> 循环执行 - 程序执行过程中，根据条件重复执行代码



shell逻辑

结构化命令

> 结构化命令允许脚本程序根据条件或者相关命令的结果进行判断，执行一些功能命令块，在另外一些条件下，执行跳过这些命令块。
>
> 在shell编程中，这些结构化的命令主要有：

```
条件逻辑 - 多分支执行命令块
    - if控制语句
    - case控制语句
    - select控制语句
循环逻辑 - 多循环执行命令块
    - for控制语句
    - while控制语句
    - until控制语句
逻辑控制 - 命令块执行过程中，精细化控制
    - continue控制
    - break控制
    - exit控制
    - shift控制
```

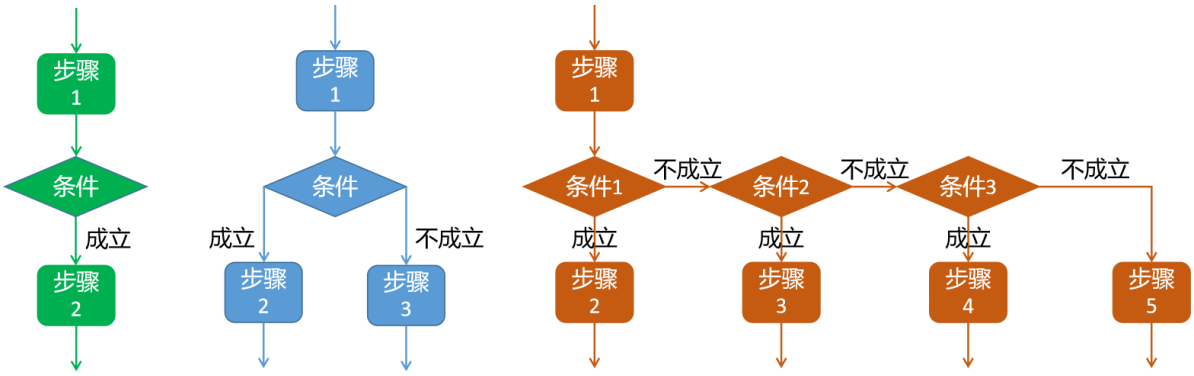
1.2.2 if语法解读

基础知识

简介

条件结构能够根据某个特定的条件，结合内置的测试表达式功能，结合测试的结果状态值对于条件进行判断，然后选择执行合适的任务。在bash中，if命令是条件结构最简单的形式。

shell中的if语句支持多种条件的决策形式：



单路决策 - 单分支if语句

样式：

```
if [ 条件 ]
then
    指令
fi
```

特点：

单一条件，只有一个输出

双路决策 - 双分支if语句

样式：

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

特点：

单一条件，两个输出

多路决策 - 多分支if语句

样式：

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

特点：

n个条件，n+1个输出

单行命令写法

```
if [ 条件1 ]; then 指令1; elif [ 条件2 ]; then 指令2; ... ; else 指令n; fi
```

```
关键点解读：
    1 if 和 then 配套使用
    2 if 和末尾的 fi 顺序反写
```

内嵌测试语句

```
    shell的if语句中关于条件判断这块内嵌了如下几种测试表达式语句：
        [ 表达式 ]      - 针对通用的判断场景
        [[ 表达式 ]]    - 针对扩展的判断场景
        (( 命令 ))      - (())代替let命令来测试数值表达式
```

简单实践

实践1-单if实践

```
[root@rocky9 ~]# cat single_branch_if.sh
#!/bin/bash
# 单分支if语句的使用场景

# 定制普通变量
gender="$1"

# 条件判断逻辑
if [ "${gender}" == "nan" ]
then
   echo "您的性别是 男"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash single_branch_if.sh nv
[root@rocky9 ~]# /bin/bash single_branch_if.sh nan
您的性别是 男
```

实践2-双if实践

```
[root@rocky9 ~]# cat double_branch_if.sh
#!/bin/bash
# 双分支if语句的使用场景

# 定制普通变量
gender="$1"

# 条件判断逻辑
if [ "${gender}" == "nan" ]
then
   echo "您的性别是 男"
else
   echo "您的性别是 女"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash double_branch_if.sh
您的性别是 女
[root@rocky9 ~]# /bin/bash double_branch_if.sh nan
您的性别是 男
[root@rocky9 ~]# /bin/bash double_branch_if.sh xxx
您的性别是 女
```

实践3-多if实践

```
[root@rocky9 ~]# cat multi_branch_if.sh
#!/bin/bash
# 多分支if语句的使用场景

# 定制普通变量
gender="$1"

# 条件判断逻辑
if [ "${gender}" == "nan" ]
then
   echo "您的性别是 男"
elif [ "${gender}" == "nv" ]
then
   echo "您的性别是 女"
else
   echo "您的性别，我不知道"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash multi_branch_if.sh
您的性别，我不知道
[root@rocky9 ~]# /bin/bash multi_branch_if.sh nan
您的性别是 男
[root@rocky9 ~]# /bin/bash multi_branch_if.sh nv
您的性别是 女
[root@rocky9 ~]# /bin/bash multi_branch_if.sh xxx
您的性别，我不知道
```

1.2.3 if 案例实践

服务管理

案例需求

```
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|stop|restart ]
```

脚本内容

```
[root@rocky9 ~]# cat service_manager_if.sh
#!/bin/bash
# 功能：定制服务管理的功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
service_ops="$1"

# 脚本基本判断
if [ $# -ne 1 ]
then
    echo -e "\e[31m$0 脚本的使用方式： $0 [ start | stop | restart ]\e[0m"
    exit
fi

# 脚本内容的判断
if [ "${service_ops}" == "start" ]
then
   echo -e "\e[31m服务启动中...\e[0m"
elif [ "${service_ops}" == "stop" ]
then
   echo -e "\e[31m服务关闭中...\e[0m"
elif [ "${service_ops}" == "restart" ]
then
   echo -e "\e[31m服务重启中...\e[0m"
else
   echo -e "\e[31m$0 脚本的使用方式： $0 [ start | stop | restart ]\e[0m"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash service_manager_if.sh
service_manager_if.sh 脚本的使用方式： service_manager_if.sh [ start | stop |
restart ]
[root@rocky9 ~]# /bin/bash service_manager_if.sh start
服务启动中...
[root@rocky9 ~]# /bin/bash service_manager_if.sh stop
服务关闭中...
[root@rocky9 ~]# /bin/bash service_manager_if.sh restart
服务重启中...
[root@rocky9 ~]# /bin/bash service_manager_if.sh xxx
service_manager_if.sh 脚本的使用方式： service_manager_if.sh [ start | stop |
restart ]
```

堡垒机登录

需求

```
    在之前的堡垒机功能基础上，扩充条件判断效果
```

脚本内容

```
[root@rocky9 ~]# cat simple_jumpserver_if.sh
#!/bin/bash
# 功能：定制堡垒机的展示页面
```

```
# 版本：v0.4
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
login_user='root'
login_pass='123456'

# 堡垒机的信息提示
echo -e "\e[31m \t\t 欢迎使用堡垒机"
echo -e "\e[32m-----------请选择你要登录的远程主机-----------
 1: 10.0.0.14 (nginx)
 2: 10.0.0.15 (tomcat)
 3: 10.0.0.19 (apache)
 q: 使用本地主机
-----------------------------------------------\033[0m"

# 由于暂时没有学习条件判断，所以暂时选择 q
read -p "请输入您要选择的远程主机编号: " host_index
read -p "请输入登录本地主机的用户名: " user
read -s -p "请输入登录本地主机的密码: " password
echo
# 远程连接主机
if [[ ${user} == ${login_user} && ${password} == ${login_pass} ]]
then
    echo -e "\e[31m主机登录验证成功\e[0m"
else
    echo -e "\e[31m您输入的用户名或密码有误\e[0m"
fi
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash simple_jumpserver_if.sh
                  欢迎使用堡垒机
-----------请选择你要登录的远程主机-----------
 1: 10.0.0.14 (nginx)
 2: 10.0.0.15 (tomcat)
 3: 10.0.0.19 (apache)
 q: 使用本地主机
-----------------------------------------------
请输入您要选择的远程主机编号: q
请输入登录本地主机的用户名: root
请输入登录本地主机的密码:
主机登录验证成功
[root@rocky9 ~]# /bin/bash simple_jumpserver_if.sh
                  欢迎使用堡垒机
-----------请选择你要登录的远程主机-----------
 1: 10.0.0.14 (nginx)
 2: 10.0.0.15 (tomcat)
 3: 10.0.0.19 (apache)
 q: 使用本地主机
-----------------------------------------------
请输入您要选择的远程主机编号: q
请输入登录本地主机的用户名: python
请输入登录本地主机的密码:
您输入的用户名或密码有误
```

1.2.4 case解读

基础知识

简介

case命令是一个多路分支的命令，它可以来代替 if/elif相关的命令，在case语句中，它通过引入一个变量接收用户输入的数据，然后依次与相关的值进行匹配判断，一旦找到对应的匹配值后，就执行相关的语句。

语法格式

```
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

案例需求

```
改造多分支if语句对脚本进行升级
     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 ]
```

脚本效果

```
查看脚本内容
[root@rocky9 ~]# cat service_manager_case.sh
#!/bin/bash
# 功能：定制服务管理的功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
service_ops="$1"

# 脚本内容的判断
```

```
case "${service_ops}" in
    "start")
        echo -e "\e[31m服务启动中...\e[0m";;
    "stop")
        echo -e "\e[31m服务关闭中...\e[0m";;
    "restart")
        echo -e "\e[31m服务重启中...\e[0m";;
    "start")
        echo -e "\e[31m服务启动中...\e[0m";;
    *)
        echo -e "\e[31m$0 脚本的使用方式： $0 [ start | stop | restart ]\e[0m";;
esac
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash service_manager_case.sh
service_manager_case.sh 脚本的使用方式： service_manager_case.sh [ start | stop |
restart ]
[root@rocky9 ~]# /bin/bash service_manager_case.sh start
服务启动中...
[root@rocky9 ~]# /bin/bash service_manager_case.sh stop
服务关闭中...
[root@rocky9 ~]# /bin/bash service_manager_case.sh restart
服务重启中...
[root@rocky9 ~]# /bin/bash service_manager_case.sh xxx
service_manager_case.sh 脚本的使用方式： service_manager_case.sh [ start | stop |
restart ]
```

1.2.5 case实践

环境标准化

案例需求

```
    由于项目环境复杂，虽然我们创建了大量的脚本，但是管理起来不方便，需要我们做一个简单的脚本执行管理平台，输入不同的环境关键字，罗列该环境下的相关脚本。
```

```
信息提示
        欢迎使用脚本管理平台
-----------请选择功能场景-----------
 1: 系统环境下脚本
 2: web环境下脚本
 3: 数据库环境下脚本
 4: 存储环境下脚本
 5: 其他环境下脚本
----------------------------------
```

准备工作

```
mkdir os web sql storage other
touch {os/os-{1..3}.sh,web/web-{1..4}.sh,sql/sql-{1..5}.sh,storage/st-
{1..2}.sh,other/other.sh}
```

脚本内容

查看脚本内容

```
[root@rocky9 ~]# cat scripts_manager_case.sh
#!/bin/bash
# 功能：定制服务管理的功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制数组变量
env_array=(os web sql storage other)

# 监控平台的信息提示
echo -e "\e[31m         欢迎使用脚本管理平台"
echo -e "\e[32m-----------请选择功能场景-----------
 1: 系统环境下脚本
 2: web环境下脚本
 3: 数据库环境下脚本
 4: 存储环境下脚本
 5: 其他环境下脚本
----------------------------------\033[0m"
# 定制业务逻辑
read -p "请输入功能标识: " env_id

# 脚本内容的判断
case "${env_array[$env_id-1]}" in
    "os")
        echo -e "\e[31m系统环境下脚本文件有:\e[0m"
        ls os;;
    "web")
        echo -e "\e[31mweb环境下脚本文件有:\e[0m"
        ls web;;
    "sql")
        echo -e "\e[31m数据库环境下脚本文件有:\e[0m"
        ls sql;;
    "storage")
        echo -e "\e[31m存储环境下脚本文件有:\e[0m"
        ls storage;;
    "other")
        echo -e "\e[31m其他环境下脚本文件有:\e[0m"
        ls other;;
    *)
        echo -e "\e[31m请输入有效的功能场景标识\e[0m";;
esac
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash scripts_manager_case.sh
         欢迎使用脚本管理平台
-----------请选择功能场景-----------
 1: 系统环境下脚本
 2: web环境下脚本
 3: 数据库环境下脚本
 4: 存储环境下脚本
 5: 其他环境下脚本
----------------------------------
请输入功能标识: 6
请输入有效的功能场景标识
```

```
[root@rocky9 ~]# /bin/bash scripts_manager_case.sh 3
        欢迎使用脚本管理平台
-----------请选择功能场景-----------
 1: 系统环境下脚本
 2: web环境下脚本
 3: 数据库环境下脚本
 4: 存储环境下脚本
 5: 其他环境下脚本
----------------------------------
请输入功能标识: 3
数据库环境下脚本文件有:
sql-1.sh  sql-2.sh  sql-3.sh  sql-4.sh  sql-5.sh
```

k8s部署

案例需求

```
    由于k8s项目环境复杂，它依赖于很多功能步骤操作，我们需要按照合理的阶段部署整体环境。相关的操作步骤信息如下。
```

```
基础环境部署
    跨主机免密码操作
    时间同步操作
    内核配置操作
    容器私有仓库操作
高可用环境部署
    高可用环境部署
    负载均衡环境部署
kubernetes基础环境部署
    证书管理
    etcd环境部署
    集群证书配置
主角色环境部署
    apiserver环境部署
    scheduler环境部署
    controller环境部署
    认证配置
    容器环境部署
    kubelet环境部署
    kube-proxy环境部署
从角色环境部署
    容器环境部署
    kubelet环境部署
    kube-proxy环境部署
```

脚本内容

```
查看脚本内容
[root@rocky9 ~]# cat kubernetes_manager_case.sh
#!/bin/bash
# 功能：定制kubernetes环境部署管理的功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制数组变量
```

```
env_array=(base ha k8s_base master slave)

# 监控平台的信息提示
echo -e "\e[31m      欢迎使用kubernetes部署平台"
echo -e "\e[32m-----------请选择部署阶段-----------
 1: 基础环境部署
 2: 高可用环境部署
 3: kubernetes基础环境部署
 4: 主角色环境部署
 5: 从角色环境部署
----------------------------------\033[0m"
# 定制业务逻辑
read -p "请输入功能标识: " env_id

# 脚本内容的判断
case "${env_array[$env_id-1]}" in
    "base")
        echo -e "\e[31m开始基础环境部署..."
        echo "1 执行跨主机免密码操作"
        echo "2 执行时间同步操作"
        echo "3 执行内核配置操作"
        echo -e "4 执行容器私有仓库部署操作\e[0m";;
    "ha")
        echo -e "\e[31高可用环境部署..."
        echo "1 执行高可用环境部署操作"
        echo -e "2 执行负载均衡环境部署操作\e[0m";;
    "k8s_base")
        echo -e "\e[31mkubernetes基础环境部署..."
        echo "1 执行证书管理操作"
        echo "2 执行etcd环境部署操作"
        echo -e "3 执行集群证书配置操作\e[0m";;
    "master")
        echo -e "\e[31m主角色环境部署..."
        echo "1 执行apiserver环境部署操作"
        echo "2 执行scheduler环境部署操作"
        echo "3 执行controller环境部署操作"
        echo "4 执行认证配置操作"
        echo "5 执行容器环境部署操作"
        echo "6 执行kubelet环境部署操作"
        echo -e "7 执行kube-proxy环境部署\e[0m";;
    "slave")
        echo -e "\e[31m主角色环境部署..."
        echo "1 执行容器环境部署操作"
        echo "2 执行kubelet环境部署操作"
        echo -e "3 执行kube-proxy环境部署\e[0m";;
    *)
        echo -e "\e[31m请输入有效的功能场景标识\e[0m";;
esac
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash kubernetes_manager_case.sh
      欢迎使用kubernetes部署平台
-----------请选择部署阶段-----------
 1: 基础环境部署
 2: 高可用环境部署
 3: kubernetes基础环境部署
 4: 主角色环境部署
```

```
 5: 从角色环境部署
---------------------------------
请输入功能标识: 7
请输入有效的功能场景标识
[root@rocky9 ~]# /bin/bash kubernetes_manager_case.sh
      欢迎使用kubernetes部署平台
-----------请选择部署阶段-----------
 1: 基础环境部署
 2: 高可用环境部署
 3: kubernetes基础环境部署
 4: 主角色环境部署
 5: 从角色环境部署
---------------------------------
请输入功能标识: 5
主角色环境部署...
1 执行容器环境部署操作
2 执行kubelet环境部署操作
3 执行kube-proxy环境部署
```

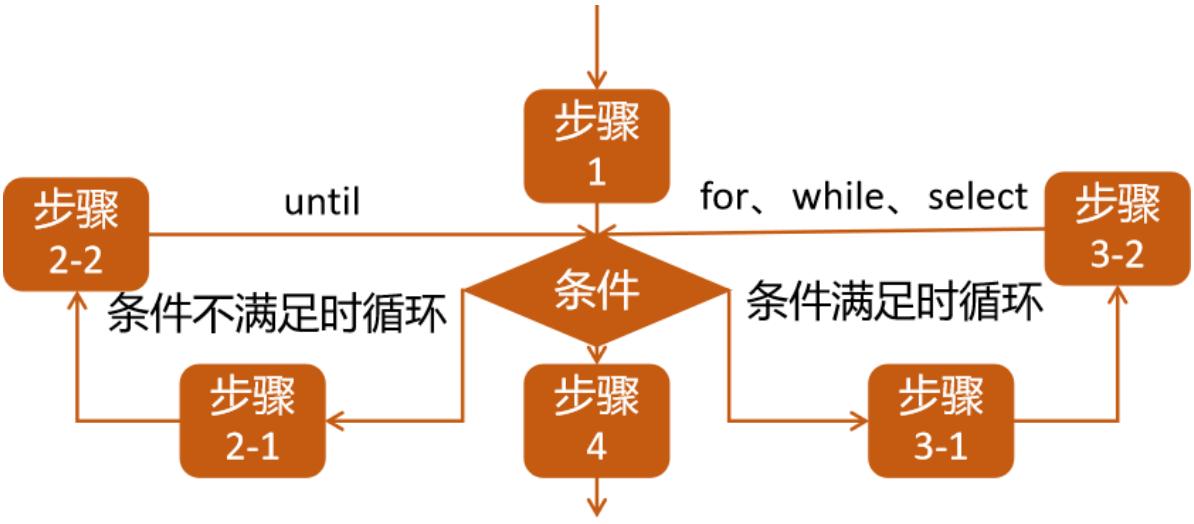
1.3 循环逻辑

1.3.1 for基础

基础知识

简介

> 生产工作中，我们有可能会遇到一种场景，需要重复性的执行相同的动作，我们在shell编程的过程中，我们可以借助于循环逻辑的方法来进行处理。



```
循环逻辑语法解析：
    关键字 [ 条件 ]
    do
        执行语句
    done

注意：
    这里的关键字主要有四种：
        for     - 循环遍历一个元素列表
        while   - 满足条件情况下一直循环下去
        until   - 不满足条件情况下一直循环下去
```

简单实践

for语法解析

```
场景：遍历列表
    for 值 in 列表
    do
       执行语句
    done
```

```
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
```

列表生成

```
样式1：手工列表
    - 1 2 3 4 5 6 7
样式2：定制列表
    - {1..7}
样式3：命令生成
    - $(seq 1 7)
样式4：脚本参数
    - $@ $*
```

实践1-手工列表

```
查看脚本内容
[root@rocky9 ~]# cat for_hand_list.sh
#!/bin/bash
# 功能：手工列表 for循环
for i in yuwen shuxue lishi
do
  echo "列表元素: ${i}"
done

脚本执行后效果
[root@rocky9 ~]# /bin/bash for_hand_list.sh
列表元素: yuwen
列表元素: shuxue
列表元素: lishi
```

实践2-定制列表

```
查看脚本内容
[root@rocky9 ~]# cat for_define_list.sh
#!/bin/bash
# 功能：定制列表 for循环
for i in {1..3}
do
  echo "列表元素: ${i}"
done

脚本执行后效果
[root@rocky9 ~]# /bin/bash for_define_list.sh
列表元素: 1
列表元素: 2
列表元素: 3
```

实践3-命令生成

```
查看脚本内容
[root@rocky9 ~]# cat for_cmd_list.sh
#!/bin/bash
# 功能：命令生成列表 for循环
for i in $(seq 1 3)
do
  echo "列表元素: ${i}"
done

脚本执行后效果
[root@rocky9 ~]# /bin/bash for_cmd_list.sh
列表元素: 1
列表元素: 2
列表元素: 3
```

实践4-脚本参数

```
查看脚本内容
[root@rocky9 ~]# cat for_arg_list.sh
#!/bin/bash
# 功能：脚本参数列表 for循环
for i in $@
do
  echo "列表元素: ${i}"
done

脚本执行后效果
[root@rocky9 ~]# /bin/bash for_arg_list.sh 1 2 3
列表元素: 1
列表元素: 2
列表元素: 3
```

1.3.2 for案例

普通循环

简介

所谓的普通循环，仅仅是在特定的范围循环中，对相关命令进行重复性的操作

批量实践1- 批量创建多个使用随机密码的用户

查看脚本内容

```
[root@rocky9 ~]# cat for_add_user.sh
#!/bin/bash
# 功能：for批量创建用户
# 提示：
# /dev/random和/dev/urandom设备文件会生成随机数，第一个依赖系统中断

# 定制普通变量
user_file='/tmp/user.txt'

# 保证文件可用
[ -f ${user_file} ] && > ${user_file}

# 定制批量创建用户的业务逻辑
for i in {1..5}
do
    # 创建用户
    useradd user-$i
    # 生成密码
    password=$(head /dev/urandom | tr -dc '[:alnum:]' | head -c 8)
    # 为用户添加密码
    echo ${password} | passwd --stdin user-$i > /dev/null 2>&1
    # 信息输出
    echo "用户: user-$i, 密码: ${password}" >> ${user_file}
    echo -e "\e[31m用户 user-$i 创建成功\e[0m"
done
```

脚本执行后效果

```
[root@rocky9 ~]# /bin/bash for_add_user.sh
用户 user-1 创建成功
用户 user-2 创建成功
用户 user-3 创建成功
用户 user-4 创建成功
用户 user-5 创建成功
[root@rocky9 ~]# cat /tmp/user.txt
用户: user-1, 密码: 6eiNoy6i
用户: user-2, 密码: rWZOVoIl
用户: user-3, 密码: i2MSnjls
用户: user-4, 密码: 9wzLjU7z
用户: user-5, 密码: U8a3Cj3R
```

清理环境

```
[root@rocky9 ~]# for i in {1..5};do userdel -r user-$i;done
[root@rocky9 ~]# grep user- /etc/passwd
```

批量实践2- 批量对特定网段的主机进行扫描

```
查看脚本内容
[root@rocky9 ~]# cat for_host_check.sh
#!/bin/bash
# 功能：for批量检测网段主机存活情况

# 定制普通变量
netsub='10.0.0'
net_file='/tmp/host.txt'

# 保证文件可用
[ -f ${net_file} ] && > ${net_file}

# 定制批量检测网段主机状态逻辑
for ip in {1..254}
do
    # 测试主机连通性
    host_status=$(ping -c1 -w0.01 $netsub.$ip >/dev/null 2>&1 && echo "UP" ||
echo "DOWN")
    echo "$netsub.$ip 主机状态: $host_status" >> ${net_file}
done
# 信息输出
live_num=$(grep UP ${net_file} | wc -l)
unlive_num=$(grep DOWN ${net_file} | wc -l)
echo -e "\e[31m${netsub}.0 网段主机存活情况\e[0m"
echo "-------------------------------------"
echo -e "\e[32m${netsub}.0 网段存活主机数量: ${live_num}\e[0m"
echo -e "\e[32m${netsub}.0 网段异常主机数量: ${unlive_num}\e[0m"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash for_host_check.sh
10.0.0.0 网段主机存活情况
-------------------------------------
10.0.0.0 网段存活主机数量: 2
10.0.0.0 网段异常主机数量: 252

[root@rocky9 ~]# grep UP /tmp/host.txt
10.0.0.2 主机状态: UP
10.0.0.12 主机状态: UP
```

赋值循环

简介

```
所谓的赋值循环，指的是在for循环过程中进行数据的统计等相关计算操作
```

统计实践1- 计算1+2+...+100 的结果

```
查看脚本内容
[root@rocky9 ~]# cat for_odd_num.sh
#!/bin/bash
# 功能：for统计数据之和
```

```
# 定制普通变量
all_sum=0
odd_sum=0

# 定制所有数据求和逻辑
for i in {1..100}
do
  let all_sum+=i
done

# 定制所有奇数求和逻辑
for i in {1..100..2}
do
  let odd_sum+=i
done

# 信息输出
echo -e "\e[31m所有数据之和: ${all_sum}\e[0m"
echo -e "\e[31m所有奇数之和: ${odd_sum}\e[0m"

脚本执行后效果
[root@rocky9 ~]# /bin/bash for_odd_num.sh
所有数据之和: 5050
所有奇数之和: 2500
```

1.3.3 while基础

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

简介

```
    while命令有点像 if/then 和 for循环之间的结合，while走循环之前会对输入的值进行条件判断，
如果满足条件的话，才会进入到循环体中执行对应的语句，否则的话就退出循环。
```

while语法解析

```
场景：只要条件满足，就一直循环下去
    while [ 条件判断 ]
    do
       执行语句
    done
```

```
注意：
    条件支持的样式 命令、[[ 字符串表达式 ]]、(( 数字表达式 ))
    true是一个特殊的条件，代表条件永远成立
```

简单实践

实践1-输出制定的范围数字

```
[root@rocky9 ~]# cat while_num_list.sh
#!/bin/bash
# 功能：while的输出5范围以内的数字

# 定制初始变量值
a=1

# 定制内容输出逻辑
while [ "${a}" -le 5 ]
do
   echo -n "${a} "
   # 每输出一次数据，数据值+1
   a=$((a+1))
done
echo
```

```
脚本执行后效果
[root@rocky9 ~]# /bin/bash while_num_list.sh
1 2 3 4 5
```

1.3.4 while案例

学习目标

这一节，我们从 案例实践、read实践、小结 三个方面来学习。

案例实践

统计实践1- 计算1+2+...+100 的结果

```
查看脚本内容
[root@rocky9 ~]# cat while_odd_num.sh
#!/bin/bash
# 功能：while统计数据之和

# 定制普通变量
all_sum=0
odd_sum=0

# 定制所有数据求和逻辑
i=1
while ((i<=100))
do
  let all_sum+=i
  let i++
done

# 定制所有奇数求和逻辑
i=1
while ((i<=100))
do
  let odd_sum+=i
  let i+=2
done
```

```
# 信息输出
echo -e "\e[31m所有数据之和: ${all_sum}\e[0m"
echo -e "\e[31m所有奇数之和: ${odd_sum}\e[0m"

脚本执行后效果
[root@rocky9 ~]# /bin/bash while_odd_num.sh
所有数据之和: 5050
所有奇数之和: 2500
```

实践2-持续检测网站存活

```
查看文件内容
[root@rocky9 ~]# cat while_site_healthcheck.sh
#!/bin/bash
# 功能：定制站点的检测功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
read -p "> 请输入待测试站点域名: " site_addr

# 持久检测站点状态
while true
do
  wget --spider -T5 -q -t2 ${site_addr} && echo "${site_addr} 站点正常" || echo
"${site_addr} 站点异常"
  sleep 1
done

注意：参数解析
    --spider：检查一个 URL 是否可用，而不是下载文件。
    -T5：这个选项设置网络连接的超时时间为 5 秒。
    -q：这个选项使 wget 在“安静”或“静默”模式下运行，即不显示任何进度或错误信息。
    -t2：这个选项设置重试次数为 2。
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash while_site_healthcheck.sh
> 请输入待测试站点域名: www.baidu.com
www.baidu.com 站点正常
www.baidu.com 站点正常
...
```

1.3.5 until基础

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

简介

```
    until命令本质上与while循环一致，区别在于until走循环之前会对输入的值进行条件判断，如果不满足条件的话，才会进入到循环体中执行对应的语句，否则的话就退出循环。
```

until语法解析

```
场景：只要条件不满足，就一直循环下去
    until [ 条件判断 ]
    do
       执行语句
    done
```

```
注意：
    条件支持的样式 命令、[[ 字符串表达式 ]]、(( 数字表达式 ))
    false是一个特殊的条件，代表条件永远不成立
```

简单实践

实践1-输出制定的范围数字

```
[root@rocky9 ~]# cat until_num_list.sh
#!/bin/bash
# 功能：until的输出5范围以内的数字

# 定制初始变量值
a=1

# 定制内容输出逻辑
until [ "${a}" -gt 5 ]
do
   echo -n "${a} "
   # 每输出一次数据，数据值+1
   a=$((a+1))
done
echo
```

```
脚本执行后效果
[root@rocky9 ~]# /bin/bash until_num_list.sh
1 2 3 4 5
```

实践2- 计算1+2+...+100 的结果

```
查看脚本内容
[root@rocky9 ~]# cat until_odd_num.sh
#!/bin/bash
# 功能：until统计数据之和

# 定制普通变量
all_sum=0
odd_sum=0

# 定制所有数据求和逻辑
i=1
until ((i>100))
do
```

```
  let all_sum+=i
  let i++
done

# 定制所有奇数求和逻辑
i=1
until ((i>100))
do
  let odd_sum+=i
  let i+=2
done

# 信息输出
echo -e "\e[31m所有数据之和: ${all_sum}\e[0m"
echo -e "\e[31m所有奇数之和: ${odd_sum}\e[0m"

脚本执行后效果
[root@rocky9 ~]# /bin/bash until_odd_num.sh
所有数据之和: 5050
所有奇数之和: 2500
```

1.4 循环控制

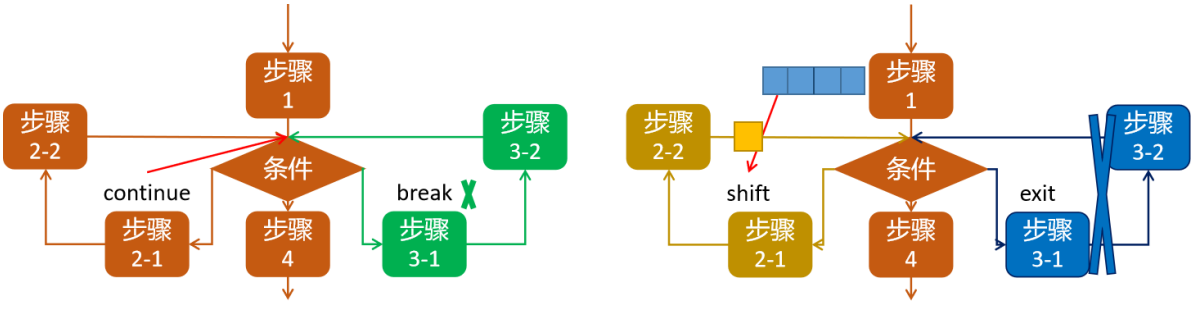
1.4.1 控制解析

基础知识

简介

```
    所谓的流程控制，主要针对的是，当我们处于流程步骤执行的过程中，因为某些特殊的原因，不得不停止既定的操作进行步骤的调整，常见的临时调整场景如下：
```

```
continue控制
    - 满足条件的情况下，临时停止当前的循环，直接进入到下一循环
break控制
    - 满足条件的情况下，提前退出当前的循环
exit控制
    - 直接退出当前循环的程序
shift控制
    - 依次从循环列表中读取读取内容，并将读取的内容从列表中剔除
```



简单实践

exit简介

exit在shell中是一个特殊的程序退出信号，不仅仅可以直接退出当前程序，还可以设定退出后的状态返回值，使用方式如下：

```
exit num
```

注意：

1 在脚本中遇到exit命令，脚本立即终止；终止退出状态取决于exit命令后面的数字

2 如果exit后面无数字,终止退出状态取决于exit命令前面命令执行结果

实践1- 设定退出状态值

```
网段内主机地址的存活性探测
[root@192 ~]# ping -c1 -W1 10.0.0.12 &> /dev/null && echo '10.0.0.12 is up' ||
(echo '10.0.0.12 is unreachable'; exit 1)
10.0.0.12 is up
[root@192 ~]# ping -c1 -W1 10.0.0.13 &> /dev/null && echo '10.0.0.13 is up' ||
(echo '10.0.0.13 is unreachable'; exit 6)
10.0.0.13 is unreachable
[root@192 ~]# echo $?
6

服务器网址探测
[root@192 ~]# curl -s -o /dev/null baidu.com &> /dev/null && echo 'baidu.com is
up' || (echo 'baidu.com is unreachable'; exit 7)
baidu.com is up
[root@192 ~]# curl -s -o /dev/null baidu.com1 &> /dev/null && echo 'baidu.com1
is up' || (echo 'baidu.com1 is unreachable'; exit 7)
baidu.com1 is unreachable
[root@192 ~]# echo $?
7
```

实践2-嵌套循环中exit退出程序

```
查看脚本内容
[root@rocky9 ~]# cat exit_multi_for.sh
#!/bin/bash
# 功能：exit退出脚本程序

# 外层循环遍历1-5
for var1 in {1..5}
do
   # 内层循环遍历a-d
   for var2 in {a..d}
   do
      # 判断退出条件，var1是2或者var2是c就退出内层循环
      if [ $var1 -eq 2 -o "$var2" == "c" ]
      then
         exit 111
      else
         echo "$var1 $var2"
      fi
   done
done
```

```
脚本执行效果
[root@192 ~]# /bin/bash exit_multi_for.sh
1 a
1 b
[root@192 ~]# echo $?
111
结果显示：
    一旦匹配到任何一个信息，就直接退出程序，而且状态码还是我们定制的。
```

1.4.2 break实践

基础知识

简介

```
    break命令是在处理过程中终止循环的一种简单方法。可以使用break命令退出任何类型的循环，包括for、while、until等。
```

```
break主要有两种场景的表现样式：
    单循环场景下，break是终止循环
        - 仅有一层 while 、for、until等
    嵌套循环场景下，break是可以终止内层循环和外层循环。
        - 存在多层while、for、until嵌套等
```

语法格式

```
break语法格式：
    for 循环列表
    do
        ...
        break num
    done
    注意：
        单循环下，break就代表退出循环
        多循环下，break的num大于嵌套的层数，就代表退出循环
```

简单实践

实践1-break终止单层循环

```
查看脚本内容
[root@rocky9 ~]# cat break_single_while.sh
#!/bin/bash
# 功能：break退出单层循环
while true
do
    read -p "输入你的数字，最好在 1 ~ 5: " 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
```

```
[root@rocky9 ~]# /bin/bash break_single_while.sh
输入你的数字，最好在 1 ~ 5: 2
你的数字是 2!
输入你的数字，最好在 1 ~ 5: 5
你的数字是 5!
输入你的数字，最好在 1 ~ 5: 6
你选择的数字没在 1 ~ 5, 退出!

结果显示:
    一旦出发break，当前循环就终止了
```

实践2-多层循环下break退出内层循环

```
[root@rocky9 ~]# cat break_multi_in_while.sh
#!/bin/bash
# 功能：break退出内层循环

# 外层循环遍历1-5
for var1 in {1..5}
do
   # 内层循环遍历a-d
   for var2 in {a..d}
   do
      # 判断退出条件，var1是2或者var2是c就退出内层循环
      if [ $var1 -eq 2 -o "$var2" == "c" ]
      then
         break
      else
         echo "$var1 $var2"
      fi
   done
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash break_multi_in_while.sh
1 a
1 b
3 a
3 b
4 a
4 b
5 a
5 b
结果显示:
    一旦出发break，则匹配内容及其后面的信息就不再输出了
```

实践3-多层循环下break退出外层循环

```
[root@rocky9 ~]# cat break_multi_out_while.sh
#!/bin/bash
# 功能：break退出外层循环

# 外层循环遍历1-5
for var1 in {1..5}
do
   # 内层循环遍历a-d
   for var2 in {a..d}
   do
      # 判断退出条件，var1是2或者var2是c就退出内层循环
      if [ $var1 -eq 2 -o "$var2" == "c" ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash break_multi_out_while.sh
1 a
1 b
结果显示：
    一旦匹配内层，则直接终止外层的循环
```

1.4.3 continue实践

基础知识

简介

```
    continue命令是在处理过程中跳出循环的一种简单方法。可以使用continue命令跳出当前的循环直接进入到下一个循环，包括for、while、until等。
```

```
continue主要有两种场景的表现样式：
    单循环场景下，continue是跳出当前循环
        - 仅有一层 while 、for、until等
    嵌套循环场景下，continue是可以跳出内层循环和外层循环。
        - 存在多层while、for、until嵌套等
```

语法格式

```
continue语法格式：
    for 循环列表
    do
        ...
        continue num
    done
    注意：
        单循环下，continue就代表跳出当前循环
        多循环下，continue的num就代表要继续的循环级别
```

简单实践

实践1-continue跳过当前单层循环

```
查看脚本内容
[root@rocky9 ~]# cat continue_single_while.sh
#!/bin/bash
# 功能：continue退出单层循环
while true
do
    read -p "输入你的数字，最好在 1 ~ 5: " 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
```

```
[root@rocky9 ~]# /bin/bash continue_single_while.sh
输入你的数字，最好在 1 ~ 5: 2
你的数字是 2!
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: 1
你的数字是 1!
输入你的数字，最好在 1 ~ 5: ^C

结果显示：
    即使输出的数据不是我们想要的，也不会退出循环逻辑
```

实践2-多层循环下continue跳过内层循环

```
[root@rocky9 ~]# cat continue_multi_in_for.sh
#!/bin/bash
# 功能：continue退出内层循环

# 外层循环遍历1-5
for var1 in {1..5}
do
   # 内层循环遍历a-d
   for var2 in {a..d}
   do
      # 判断退出条件，var1是2或者var2是c就退出内层循环
      if [ $var1 -eq 2 -o "$var2" == "c" ]
      then
         continue
      else
         echo "$var1 $var2"
      fi
   done
done
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash break_multi_in_for.sh
1 a
1 b
1 d
3 a
3 b
3 d
4 a
4 b
4 d
5 a
5 b
5 d
```

结果显示：

满足条件的信息，都直接跳到下一循环，继续执行，可以看到内层的d信息输出了

实践3-多层循环下continue退出外层循环

```
[root@rocky9 ~]# cat continue_multi_out_for.sh
#!/bin/bash
# 功能：continue退出外层循环

# 外层循环遍历1-5
for var1 in {1..5}
do
   # 内层循环遍历a-d
   for var2 in {a..d}
   do
      # 判断退出条件，var1是2或者var2是c就退出内层循环
      if [ $var1 -eq 2 -o "$var2" == "c" ]
      then
         continue 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash continue_multi_out_for.sh
1 a
1 b
3 a
3 b
4 a
4 b
5 a
5 b
```

结果显示：

满足条件的信息，直接将外层循环跳到下一循环，继续执行，可以看到外层的3-5信息输出了

1.5 函数

1.5.1 基础知识

基础知识

场景需求

在shell脚本的编写过程中，我们经常会遇到一些功能代码场景：多条命令组合在一起，实现一个特定的功能场景逻辑、一些命令在脚本内部的多个位置频繁出现。在这些场景的代码量往往不多，但是频繁使用的话，会导致脚本的整体逻辑脉络比较松散和框架散乱。

所以我们需要一种脚本逻辑，不仅仅能够满足松散代码的功能目的，还能精简重复的代码。函数就是来满足这种场景的解决方案 -- 而函数，也是所谓的面向对象编程的一种表现样式。

函数

所谓的函数，本质上就是一段能够满足特定功能的代码块。一旦定义好函数代码后，我们就可以在脚本的很多位置随意的使用。

定义功能代码块的动作叫 函数定义，使用函数代码的动作叫 函数调用。

函数的优势：

1. 代码模块化，调用方便，节省内存
2. 代码模块化，代码量少，排错简单
3. 代码模块化，可以改变代码的执行顺序

基本语法

定义函数：

样式1：标准格式

```
function 函数名{
    函数体
}
```

样式2：简约格式

```
函数名() {
    函数体
}
```

注意：

function 的作用和 () 的作用是一样的，都是定义一个函数。

函数的名称是自定义的，而且在脚本范围内必须唯一。

函数体内是普通的能够正常执行的命令，命令的执行流程符合顺序逻辑。

调用函数：

```
函数名
```

注意：

函数名出现在任何位置，就代表在该位置调用函数内代码块的执行。

函数名一般在函数定义后调用，否则的话会发生报错。

简单实践

实践1-标准函数的实践

```
[root@rocky9 ~]# cat function_simple_test.sh
```

```
#!/bin/bash
# 功能：简单函数的定义和调用

# 定制一个函数,提示脚本的使用方式
function Usage {
    echo -e "\e[31m脚本的使用帮助信息: xxx\e[0m"
}

# 定制脚本使用逻辑
if [ $# -eq 1 ]
then
    echo "您输入的脚本参数是1个"
else
    Usage
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_simple_test.sh
脚本的使用帮助信息: xxx
[root@rocky9 ~]# /bin/bash function_simple_test.sh aa
您输入的脚本参数是1个
[root@rocky9 ~]# /bin/bash function_simple_test.sh aa bb
脚本的使用帮助信息: xxx
```

实践2-变种函数的实践

```
[root@rocky9 ~]# cat function_simple_test2.sh
#!/bin/bash
# 功能：简单函数的定义和调用

# 定制一个函数,提示脚本的使用方式
Usage() {
    echo -e "\e[31m脚本的使用帮助信息: xxx\e[0m"
}

# 定制脚本使用逻辑
if [ $# -eq 1 ]
then
    echo "您输入的脚本参数是1个"
else
    Usage
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_simple_test2.sh
脚本的使用帮助信息: xxx
[root@rocky9 ~]# /bin/bash function_simple_test2.sh aa
您输入的脚本参数是1个
[root@rocky9 ~]# /bin/bash function_simple_test2.sh aa bb
脚本的使用帮助信息: xxx
```

实践3-函数的调用顺序和名称唯一 实践

```
[root@rocky9 ~]# cat function_simple_test3.sh
#!/bin/bash
```

```
# 功能：简单函数的定义和调用

# 定制一个函数,提示脚本的使用方式
Usage() {
    echo -e "\e[31m脚本的使用帮助信息: xxx\e[0m"
}
echo "第一次调用效果: "
Usage

# 定制同名的函数,提示脚本的使用方式
Usage() {
    echo -e "\e[31m脚本的使用帮助信息-------: xxx\e[0m"
}
# 定制脚本使用逻辑
if [ $# -eq 1 ]
then
    # 调用一个后面才会生成的函数
    func
else
    Usage
fi

# 定制一个函数
func() {
    echo "您输入的脚本参数是1个"
}
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_simple_test3.sh
第一次调用效果:
脚本的使用帮助信息: xxx
脚本的使用帮助信息-------: xxx
[root@rocky9 ~]# /bin/bash function_simple_test3.sh a
第一次调用效果:
脚本的使用帮助信息: xxx
function_simple_test3.sh:行18: func: 未找到命令

结果显示:
    函数名称重复的话，会导致同名函数被覆盖
    函数在没有定义前调用的话，会导致异常报错
```

1.5.2 函数退出

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

```
    我们可以将函数代码块，看成shell脚本内部的小型脚本，所以说函数代码块也会有执行状态返回值。对于函数来说，它通常支持两种种状态返回值的样式。
```

```
样式1-默认的退出状态
    默认情况下，函数的退出状态是函数体内的最后一条命令的退出状态，可以通过 $? 来获取
样式2-return定制状态返回值
    在函数体内部，通过return定制状态返回值的内容
    注意：
        return的状态返回值必须尽快使用，否则会被其他return的值覆盖
        return的状态返回值必须在 0-255，否则失效
```

简单实践

实践1-默认退出状态

```
[root@rocky9 ~]# cat function_exit_status1.sh
#!/bin/bash
# 功能：函数默认状态返回值

# 定制成功运行的函数
ok_func() {
    echo -e "\e[31m脚本的使用帮助信息: xxx\e[0m"
}
# 定制一个运行失败的函数
err_func() {
    666666
}
# 定制脚本使用逻辑
if [ $# -eq 1 ]
then
    err_func
    echo "错误函数的执行状态返回值: " $?
else
    ok_func
    echo "成功函数的执行状态返回值: " $?
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_exit_status1.sh
脚本的使用帮助信息: xxx
成功函数的执行状态返回值:  0
[root@rocky9 ~]# /bin/bash function_exit_status1.sh aa
function_exit_status1.sh:行10: 666666: 未找到命令
错误函数的执行状态返回值:  127
[root@rocky9 ~]# lll; echo $?
bash: lll: 未找到命令
127
结果显示：
    对于异常的函数来说，默认的状态返回值有安全隐患
```

实践2-return定制函数的返回值实践

```
[root@rocky9 ~]# cat function_exit_status2.sh
#!/bin/bash
# 功能：return定制函数状态返回值

# 定制成功运行的函数
ok_func() {
```

```
    echo -e "\e[31m脚本的使用帮助信息: xxx\e[0m"
    # 定制超范围的状态返回值
    return 666
}
# 定制一个运行失败的函数
err_func() {
    666666
    # 定制状态返回值
    return 222
}
# 定制脚本使用逻辑
if [ $# -eq 1 ]
then
    err_func
    echo "错误函数的执行状态返回值: " $?
else
    ok_func
    echo "成功函数的执行状态返回值: " $?
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_exit_status2.sh
脚本的使用帮助信息: xxx
成功函数的执行状态返回值:  154
[root@rocky9 ~]# /bin/bash function_exit_status2.sh aa
function_exit_status2.sh:行12: 666666: 未找到命令
错误函数的执行状态返回值:  222
结果显示:
    return的状态返回值范围必须满足要求
```

1.5.3 进阶实践

这一节，我们从 传参函数、脚本传参、小结 三个方面来学习。

传参函数

简介

```
    简单的函数定义和调用的实践，我们只能实现固定内容的输出，不具有灵活性。其实函数作为shell脚本内部的小脚本也支持脚本传参的一系列能力。基本语法效果如下
```

```
定义函数:
    函数名() {
        函数体 ${变量名}
    }
注意:
    函数体内通过 ${变量名} 来实现函数体的功能通用性。
```

```
调用函数:
    函数名 参数
注意:
    函数在调用的时候，接收一些参数并传输到函数体内部。
```

实践1-传参函数实践

```
查看脚本内容
[root@rocky9 ~]# cat function_arg_input.sh
#!/bin/bash
# 功能：传参函数定义和调用

# 定制数据运算的函数
add_func() {
    echo $(( $1 + $2 ))
}
sub_func() {
    echo $(( $1 - $2 ))
}
mul_func() {
    echo $(( $1 * $2 ))
}
div_func() {
    echo $(( $1 / $2 ))
}

echo -n "4+3="; add_func 4 3
echo -n "4-3="; sub_func 4 3
echo -n "4*3="; mul_func 4 3
echo -n "4/3="; div_func 4 3
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_arg_input.sh
4+3=7
4-3=1
4*3=12
4/3=1
```

脚本传参

简介

```
    传参函数定义和调用的实践，实现了函数层面的灵活性，但是它受到函数调用本身的参数限制。往往这些参数我们需要在脚本执行的时候传递进去，从而实现脚本功能的灵活性。
    基本语法效果如下
```

```
定义函数：
    函数名() {
        函数体 ${函数参数}
    }

调用函数：
    函数名 ${脚本参数}

脚本执行：
    /bin/bash /path/to/scripts.sh arg

注意：
    由于脚本内部调用脚本参数和函数体内调用函数参数都遵循位置变量的使用
    所以，一般情况下，我们会借助于临时变量的方式接收各自的参数，从而避免引起误会
```

实践1-脚本传参函数实践

查看脚本内容

```
[root@rocky9 ~]# cat function_arg_scripts.sh
#!/bin/bash
# 功能：脚本传参函数调用

# 定制数据运算的函数
add_func() {
    echo $(( $1 + $2 ))
}
sub_func() {
    echo $(( $1 - $2 ))
}
mul_func() {
    echo $(( $1 * $2 ))
}
div_func() {
    echo $(( $1 / $2 ))
}

[ $# -ne 2  ] && echo "必须传递两个数字参数" && exit
echo -n "$1+$2="; add_func $1 $2
echo -n "$1-$2="; sub_func $1 $2
echo -n "$1*$2="; mul_func $1 $2
echo -n "$1/$2="; div_func $1 $2
```

注意：

这种简单的脚本传参函数调用，导致大量的位置参数，容易引起混乱，需要改造

脚本执行效果

```
[root@rocky9 ~]# /bin/bash function_arg_scripts.sh
必须传递两个数字参数
[root@rocky9 ~]# /bin/bash function_arg_scripts.sh 5 4
5+4=9
5-4=1
5*4=20
5/4=1
```

实践2-脚本传参函数进阶实践

查看脚本内容

```
[root@rocky9 ~]# cat function_arg_scripts2.sh
#!/bin/bash
# 功能：传参函数定义和调用

# 接收脚本传参
arg1=$1
arg2=$2
# 定制数据运算的函数
add_func() {
    num1=$1
    num2=$2
    echo $(( ${num1} + ${num2} ))
}
sub_func() {
```

```
    num1=$1
    num2=$2
    echo $(( ${num1} - ${num2} ))
}
mul_func() {
    num1=$1
    num2=$2
    echo $(( ${num1} * ${num2} ))
}
div_func() {
    num1=$1
    num2=$2
    echo $(( ${num1} / ${num2} ))
}

[ $# -ne 2  ] && echo "必须传递两个数字参数" && exit
echo -n "${arg1}+${arg2}="; add_func ${arg1} ${arg2}
echo -n "${arg1}-${arg2}="; sub_func ${arg1} ${arg2}
echo -n "${arg1}*${arg2}="; mul_func ${arg1} ${arg2}
echo -n "${arg1}/${arg2}="; div_func ${arg1} ${arg2}
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_arg_scripts2.sh
必须传递两个数字参数
[root@rocky9 ~]# /bin/bash function_arg_scripts2.sh 7 5
7+5=12
7-5=2
7*5=35
7/5=1
```

1.5.4 综合案例

这一节，我们从 信息采集、环境部署、小结 三个方面来学习。

信息采集

脚本实践-采集系统负载信息

```
查看脚本内容
[root@rocky9 ~]# cat function_systemctl_load.sh
#!/bin/bash
# 功能：采集系统负载信息
# 版本：v0.3
# 作者：王树森
# 联系：sswang.magedu.com

# 定制资源类型
resource_type=(CPU MEM)


# 定制cpu信息输出函数
cpu_info() {
    cpu_attribute=(1 5 15)
    cpu_load=($(uptime | tr -s " " | cut -d " " -f 9-11 | tr "," " "))
    echo -e "\e[31m\t系统CPU负载信息\e[0m"
```

```
    echo -e "\e[32m================================="
    for index in ${!cpu_attribute[@]}
    do
        echo "CPU ${cpu_attribute[$index]} min平均负载为: ${cpu_load[$index]}"
    done
    echo -e "=================================\e[0m"
}
# 获取内存相关属性信息
mem_info() {
    free_attribute=(总量 使用 空闲)
    free_info=($(free -m | grep Mem | tr -s " " | cut -d " " -f 2-4))
    echo -e "\e[31m\t系统内存负载信息\e[0m"
    echo -e "\e[32m================================="
    for index in ${!free_attribute[@]}
    do
        echo "内存 ${free_attribute[$index]} 信息为: ${free_info[$index]} M"
    done
    echo -e "=================================\e[0m"
}

# 服务的操作提示
echo -e "\e[31m---------------查看资源操作动作---------------
 1: CPU  2: MEM
-------------------------------------------"'\033[0m'
# 选择服务操作类型
while true
do
    read -p "> 请输入要查看的资源信息类型: " resource_id
    echo
    case ${resource_type[$resource_id-1]} in
       "CPU")
            cpu_info;;
        "MEM")
            mem_info;;
        *)
            echo -e "\e[31m\t请输入有效的信息类型\e[0m";;
    esac
done
```

脚本使用效果

```
[root@rocky9 ~]# /bin/bash function_systemctl_load.sh
---------------查看资源操作动作---------------
 1: CPU  2: MEM
-------------------------------------------
> 请输入要查看的资源信息类型: 1

        系统CPU负载信息
=================================
CPU 1 min平均负载为: 0.00
CPU 5 min平均负载为: 0.01
CPU 15 min平均负载为: 0.05
=================================
> 请输入要查看的资源信息类型: 2

        系统内存负载信息
=================================
内存 总量 信息为: 3770 M
```

```
内存 使用 信息为: 237 M
内存 空闲 信息为: 3290 M
==================================
> 请输入要查看的资源信息类型: 3

        请输入有效的信息类型
> 请输入要查看的资源信息类型: ^C
[root@rocky9 ~]#
```

环境部署

需求

```
定制kubernetes环境部署管理的功能脚本改造
    1 功能函数实现
    2 扩充while循环执行功能
    3 增加q退出环境功能
```

脚本内容

```
查看脚本内容
[root@rocky9 ~]# cat function_kubernetes_manager.sh
#!/bin/bash
# 功能：定制kubernetes环境部署管理的功能
# 版本：v0.2
# 作者：王树森
# 联系：sswang.magedu.com

# 定制数组变量
env_array=(base ha k8s_base master slave)

# 监控平台的信息提示
menu(){
    echo -e "\e[31m      欢迎使用kubernetes部署平台"
    echo -e "\e[32m-----------请选择部署阶段-----------"
    echo -e " 1: 基础环境部署"
    echo -e " 2: 高可用环境部署"
    echo -e " 3: kubernetes基础环境部署"
    echo -e " 4: 主角色环境部署"
    echo -e " 5: 从角色环境部署"
    echo -e " q: 退出"
    echo -e "----------------------------------\033[0m"
}
# 定制基础环境
os_base_func(){
    echo -e "\e[31m开始基础环境部署..."
    echo "1 执行跨主机免密码操作"
    echo "2 执行时间同步操作"
    echo "3 执行内核配置操作"
    echo -e "4 执行容器私有仓库部署操作\e[0m"
}

# 定制高可用环境
ha_func(){
    echo -e "\e[31高可用环境部署..."
    echo "1 执行高可用环境部署操作"
    echo -e "2 执行负载均衡环境部署操作\e[0m"
```

```
}

# 定制k8s基础环境
k8s_base_func(){
    echo -e "\e[31mkubernetes基础环境部署..."
    echo "1 执行证书管理操作"
    echo "2 执行etcd环境部署操作"
    echo -e "3 执行集群证书配置操作\e[0m"
}

# 定制主角色环境
master_func(){
    echo -e "\e[31m主角色环境部署..."
    echo "1 执行apiserver环境部署操作"
    echo "2 执行scheduler环境部署操作"
    echo "3 执行controller环境部署操作"
    echo "4 执行认证配置操作"
    echo "5 执行容器环境部署操作"
    echo "6 执行kubelet环境部署操作"
    echo -e "7 执行kube-proxy环境部署\e[0m"
}
# 定制从角色环境
slave_func(){
    echo -e "\e[31m主角色环境部署..."
    echo "1 执行容器环境部署操作"
    echo "2 执行kubelet环境部署操作"
    echo -e "3 执行kube-proxy环境部署\e[0m"
}
# 定制错误提示信息
usage_func(){
    echo -e "\e[31m请输入有效的功能场景标识\e[0m"
}


# 脚本内容的判断
while true
do
    # 定制业务逻辑
    menu
    read -p "请输入功能标识: " env_id
    if [ ${env_id} == "q" ];then
        exit
    else
        # 执行配套业务逻辑
        case "${env_array[$env_id-1]}" in
            "base")
                os_base_func;;
            "ha")
                ha_func;;
            "k8s_base")
                k8s_base_func;;
            "master")
                master_func;;
            "slave")
                slave_func;;
            *)
                usage_func;;
        esac
```

```
    fi
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_kubernetes_manager.sh
      欢迎使用kubernetes部署平台
-----------请选择部署阶段-----------
 1: 基础环境部署
 2: 高可用环境部署
 3: kubernetes基础环境部署
 4: 主角色环境部署
 5: 从角色环境部署
 q: 退出
----------------------------------
请输入功能标识: 6
请输入有效的功能场景标识
      欢迎使用kubernetes部署平台
-----------请选择部署阶段-----------
 1: 基础环境部署
 2: 高可用环境部署
 3: kubernetes基础环境部署
 4: 主角色环境部署
 5: 从角色环境部署
 q: 退出
----------------------------------
请输入功能标识: 5
主角色环境部署...
1 执行容器环境部署操作
2 执行kubelet环境部署操作
3 执行kube-proxy环境部署
      欢迎使用kubernetes部署平台
-----------请选择部署阶段-----------
 1: 基础环境部署
 2: 高可用环境部署
 3: kubernetes基础环境部署
 4: 主角色环境部署
 5: 从角色环境部署
 q: 退出
----------------------------------
请输入功能标识: q
```

1.5.5 函数变量

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

> 在函数作用范围中，我们可以通过大量的变量来实现特定数据的临时存储，不仅仅可以实现脚本层面的变量可视化，还可以借助于变量本身的全局和本地的特点实现更加强大的功能场景。

变量类型

```
全局变量：
    默认情况下，脚本中的普通变量就是全局变量，作用范围是shell脚本的所有地方，在函数内部也可以正常使用
    而且函数内可以可以修改脚本级别的全局变量
局部变量：
    我们可以通过local语法，将变量的作用范围限制在一段代码块范围中。
    注意：脚本内无法使用local语法，仅限于函数体内
```

简单实践

实践1-全局变量实践

```
查看脚本内容
[root@rocky9 ~]# cat function_env_test.sh
#!/bin/bash
# 功能：全局变量实践

# 定制普通的全局变量
message="helloworld"

# 定制一个函数,提示脚本的使用方式
function Usage {
    echo "直接调用脚本的message: ${message}"
    message="function-message"
    echo "函数体重置后的message: ${message}"
}

# 定制脚本使用逻辑
while true
do
    read -p "查看变量的方式[ 1-脚本内 | 2-函数内 ]: " type
    if [ ${type} == "1" ];then
        # 直接在脚本环境使用全局变量
        echo ${message}
    elif [ ${type} == "2" ];then
        # 函数内部使用全局变量
        Usage
    fi
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_env_test.sh
查看变量的方式[ 1-脚本内 | 2-函数内 ]: 1
helloworld
查看变量的方式[ 1-脚本内 | 2-函数内 ]: 2
直接调用脚本的message: helloworld
函数体重置后的message: function-message
查看变量的方式[ 1-脚本内 | 2-函数内 ]: 1
function-message                        # 结果显示，函数体内的变量生效了
查看变量的方式[ 1-脚本内 | 2-函数内 ]: ^C
[root@rocky9 ~]#
结果显示：
    对于脚本的普通变量，函数内外都可以正常使用，而且函数内可以直接修改脚本级别的普通变量
```

实践2-本地变量实践

查看脚本内容

```
[root@rocky9 ~]# cat function_env_test2.sh
#!/bin/bash
# 功能：local定制函数级别的局部变量实践

# 定制普通变量
message="helloworld"
local local_env="local"

# 定制一个函数,提示脚本的使用方式
function Usage {
    echo "直接调用脚本的变量: ${message}-${local_env}"
    local message="function-message"
    echo "函数体重置后的变量: ${message}-${local_env}"
}

# 定制脚本使用逻辑
while true
do
    read -p "查看变量的方式[ 1-脚本内 | 2-函数内 ]: " type
    if [ ${type} == "1" ];then
        # 直接在脚本环境使用普通变量
        echo ${message}
    elif [ ${type} == "2" ];then
        # 函数内部使用普通变量
        Usage
    fi
done
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash function_env_test2.sh
function_env_test2.sh: 第 6 行:local: 只能在函数中使用
查看变量的方式[ 1-脚本内 | 2-函数内 ]: 1
helloworld
查看变量的方式[ 1-脚本内 | 2-函数内 ]: 2
直接调用脚本的变量: helloworld-
函数体重置后的变量: function-message-
查看变量的方式[ 1-脚本内 | 2-函数内 ]: 1
helloworld
查看变量的方式[ 1-脚本内 | 2-函数内 ]: ^C
[root@rocky9 ~]#
```

结果显示：

local仅限于函数中使用，函数外部无法使用

经过local限制后的环境变量，无法被函数体外进行使用

1.6 脚本自动化

1.6.1 信号基础

基础知识

简介

> 当我们在构建一些更高级的脚本的时候，就会涉及到如何在linux系统上来更好的运行和控制它们，到目前为止，我们运行脚本的方式都是以实时的模式，在命令行来运行它。但是这并不是脚本唯一的运行方式，我们可以在linux系统中以更丰富的方式来运行它们，甚至在脚本遇到不可查的异常中止时候，以关闭linux终端界面的方式终止脚本。
>
> 这些能力都是基于信号的机制来实现了

信号

> linux使用信号与系统上运行的进程进行通信，想要对shell的脚本控制，只需要传递相关信号给shell脚本即可。

信号	值	描述	信号	值	描述
1	SIGHUP	挂起进程	15	SIGTERM	优雅的终止进程
2	SIGINT	终止进程	17	SIGSTOP	无条件停止进程，不终止进程
3	SIGQUIT	停止进程	18	SIGTSTP	停止或暂停进程，不终止进程
9	SIGKILL	无条件终止进程	19	SIGCONT	继续运行停止的进程

> 默认情况下，bash shell会忽略收到的任何SIGQUIT(3)和SIGTERM(15)信号（正因为这样交互式shell才不会被意外终止）。但是bash shell会处理收到的SIGHUP(1)和SIGINT(2)信号。
>
> 如果bash shell收到SIGHUP信号，它会退出。但在退出之前，它会将信号传给shell启动的所有进程（比如shell脚本）。通过SIGINT信号，可以中断shell，Linux内核停止将CPU的处理时间分配给shell，当这种情况发生时，shell会将SIGINT信号传给shell启动的所有进程。

生成信号

```
终止进程：
    ctrl+c,
暂停进程：
    ctrl+z,停止的进程继续保留在内存中，并能从停止的位置继续运行
恢复进程：
    jobs查看运行任务，fg num 重新执行
杀死进程：
    kill -9 pid
```

简单实践

实践1-终止进程

```
[root@rocky9 ~]# sleep 1000
^C
[root@rocky9 ~]#
```

实践2-挂起进程

```
[root@rocky9 ~]# sleep 1000
^Z
[1]+  已停止               sleep 1000
[root@rocky9 ~]# ps aux  | grep sleep
root       39067  0.0  0.0 108052   360 pts/0    T    17:28   0:00 sleep 1000
```

实践3-恢复进程

```
查看所有挂起进程
[root@rocky9 ~]# jobs
[1]+  已停止               sleep 1000

恢复挂起进程的id
[root@rocky9 ~]# fg 1
sleep 1000
^C
[root@rocky9 ~]#
```

实践4-杀死进程

```
后台执行命令
[root@rocky9 ~]# sleep 1000 &
[1] 39074
[root@rocky9 ~]# ps aux  | grep sleep | grep -v grep
root       39074  0.0  0.0 108052   360 pts/0    S    17:30   0:00 sleep 1000

强制杀死进程
[root@rocky9 ~]# kill -9 39074
[root@rocky9 ~]#
[1]+  已杀死               sleep 1000
[root@rocky9 ~]# jobs
```

1.6.2 信号捕捉

--- 拓展资料 ---

基础知识

简介

```
    shell编程提供了一种方式，让我们可以随意的控制脚本的运行状态，这就需要涉及到信号的捕获操作。在shell编程中，我们可以借助于 trap命令实现指定shell脚本要watch哪些linux信号并从shell中拦截。如果脚本收到了trap命令中列出的信号，它会阻止它被shell处理，而在本地处理。
```

trap命令格式

```
命令格式
    trap commands signals

命令示例:
    # 收到指定信号后，执行自定义指令，而不会执行原操作
    trap '触发指令' 信号
```

```
    # 忽略信号的操作
    trap '' 信号

    # 恢复原信号的操作
    trap '-' 信号

    # 列出自定义信号操作
    trap -p

    # 当脚本退出时，执行finish函数
    trap finish EXIT
```

简单实践

实践1-捕获终止信号

```
查看脚本内容
[root@rocky9 ~]# cat signal_trap_test1.sh
#!/bin/bash
# 功能：脚本信号捕捉

# 捕获关闭信号
trap "你敢关我，就不关，气死你" SIGINT SIGTERM
trap "走了，不送" EXIT

# 检测逻辑效果
while true
do
    read -p "请输入一个数据：" value
    echo "您输入的数据是：${value}"
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash signal_trap_test1.sh
请输入一个数据：4
您输入的数据是：4
请输入一个数据：^Csignal_trap_test1.sh:行1：你敢关我，就不关，气死你：未找到命令

您输入的数据是：
请输入一个数据：^Z
[1]+  已停止               /bin/bash signal_trap_test1.sh
[root@rocky9 ~]#
[root@rocky9 ~]# jobs
[1]+  已停止               /bin/bash signal_trap_test1.sh
[root@rocky9 ~]# fg 1
/bin/bash signal_trap_test1.sh

您输入的数据是：
请输入一个数据：3
您输入的数据是：3
```

```
另开一个终端，直接kill进程
[root@rocky9 ~]# ps aux | grep sign
root       39142  0.0  0.0 113288  1460 pts/0    S+   17:43   0:00 /bin/bash
signal_trap_test1.sh
[root@rocky9 ~]# kill -9 39142

回到之前的终端查看效果
[root@rocky9 ~]# fg 1
/bin/bash signal_trap_test1.sh

您输入的数据是:
请输入一个数据：3
您输入的数据是: 3
请输入一个数据：已杀死
```

实践2-捕获正常退出

```
查看脚本内容
[root@rocky9 ~]# cat signal_trap_test2.sh
#!/bin/bash
# 功能：脚本信号捕捉

# 捕获关闭信号
trap "echo '走了.不送'" EXIT

value="0"
# 检测逻辑效果
while true
do
    read -p "请输入一个数据：" value
    if [ ${value} == "9" ]
    then
        exit
    else
        echo "您输入的数据是: ${value}"
    fi
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash signal_trap_test2.sh
请输入一个数据：3
您输入的数据是: 3
请输入一个数据：9
走了.不送
```

实践3-移除捕获

```
查看脚本内容
[root@rocky9 ~]# cat signal_trap_test3.sh
#!/bin/bash
# 功能：移除脚本信号捕捉

# 捕获关闭信号
trap "echo '走了.不送'" EXIT
```

```
i=1
# 检测逻辑效果
while [ $i -le 3 ]
do
    read -p "请输入一个数据：" value
    if [ ${value} == "9" ]
    then
        exit
    else
        echo "您输入的数据是: ${value}"
    fi
    let i+=1
done

# 移除捕获信号
trap - EXIT
echo "移除了捕获信号"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash signal_trap_test3.sh
请输入一个数据：9
走了.不送
[root@rocky9 ~]# /bin/bash signal_trap_test3.sh
请输入一个数据：1
您输入的数据是: 1
请输入一个数据：2
您输入的数据是: 2
请输入一个数据：3
您输入的数据是: 3
移除了捕获信号

结果显示：
    在没有走到信号捕获移除的时候，捕获仍然生效
```

1.6.3 expect基础

基础知识

场景需求

```
在日常工作中，经常会遇到各种重复性的"手工交互"操作，虽然没有什么技术含量，但是相当的重要。在实际的工作场景中，这种重复性的手工操作动作，非常的繁多，但是对于量大的工作来说，效率就非常低效了。所以我们就需要有一种工具，能够简化我们重复的手工操作。
```

expect简介

```
expect是一个免费的编程工具，由DonLibes制作，作为Tcl脚本语言的一个扩展,它可以根据程序的提示，模拟标准输入提供给程序，从而实现自动的交互式任务，而无需人为干预,可以用作Unix系统中进行应用程序的自动化控制和测试的软件工具。

说白了，expect就是一套用来实现自动交互功能的软件。它主要应用于执行命令和程序时，系统以交互形式要求输入指定字符串，实现交互通信。在使用的过程中，主要是以脚本文件的样式来存在

官方网站：
    https://www.nist.gov/services-resources/software/expect
工具手册：
    man expect
```

软件部署

```
安装软件
[root@rocky9 ~]# yum install expect -y

查看效果
[root@rocky9 ~]# expect -v
expect version 5.45
```

```
进入专用的命令交互界面
[root@rocky9 ~]# expect
expect1.1>  ls
anaconda-ks.cfg
expect1.2> exit
```

常见符号

```
{ }：
    作用2：代码块儿，但是两个 {} 边界必须在一起。
        正确样式：
        if {代码块1 } {
            代码块2
            }
        错误示例：
        if {$count < 0}
        {
           break;
        }
    注意：
        无论什么时候，{}边界符号与其他内容都最好有空格隔开，尤其是边界外的内容
注意：
    在expect 中，没有小括号的概念和应用
```

常用命令

```
set         设定环境变量
                格式：set 变量名 变量值
                样式：set host "192.168.8.12"


spawn       启动新的进程，模拟手工在命令行启动服务
                格式：spawn 手工执行命令
```

```
                样式：spawn ssh python@$host

expect      接收一个新进程的反馈信息，我们根据进程的反馈，再发送对应的交互命令
                格式：expect "交互界面用户输入处的关键字"
                样式：expect "*password*"

send        接收一个字符串参数，并将该参数发送到新进程。
                格式：send "用户输入的信息"
                样式：send "$password\r"

interact    退出自动化交互界面，进入用户交互状态，如果需要用户交互的话，这条命令必须在最后一行
                格式：interact
                样式：interact

其他命令
    expect eof  expect执行内容的结束标识符，退出当前脚本，与interact只能存在一个
    exp_continue    expect获取期望后，还会有另外的期望，那么我们就把多个期望连续执行
```

简单实践

实践1-简单的登录交互脚本

```
查看脚本内容
[root@rocky9 ~]# cat login_test.expect
#!/usr/bin/expect

# 1 设定环境变量
set username sswang

# 2 发起远程登录请求
spawn ssh $username@10.0.0.12

# 3 识别用户输入的位置关键字
expect "yes/no"

# 4 发送正确的信息
send "yes\r"

# 5 识别密码关键字，并传递密码信息
send "\r"
expect "password:"
send "123456\r"

# 6 切换回用户交互界面
interact

注意:
    由于password前面会涉及到一次Enter操作，所以在password匹配前，输入一次 \r
```

```
清理历史记录
[root@rocky9 ~]# rm -f .ssh/know_hosts

执行脚本内容
[root@rocky9 ~]# expect login_test.expect
spawn ssh sswang@10.0.0.12
```

```
The authenticity of host '10.0.0.12 (10.0.0.12)' can't be established.
ECDSA key fingerprint is SHA256:XUJsgk4cTORxdcswxIKBGFgrrqFQzpHmKnRRV6ABMk4.
ECDSA key fingerprint is MD5:71:74:46:50:3f:40:4e:af:ad:d3:0c:de:2c:fc:30:c0.
Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added '10.0.0.12' (ECDSA) to the list of known hosts.
sswang@10.0.0.12's password:
[sswang@localhost ~]$ id
uid=1000(sswang) gid=1000(sswang) 组=1000(sswang)
```

实践2-脚本结合

expect 除了使用专用的expect脚本来实现特定功能之外，它还可以与其他脚本嵌套在一起进行使用。最常用的结合方式就是 shell结合。

在于shell结合使用的时候，无非就是将expect的执行命令使用 <<-EOF 。。。 EOF 包装在一起即可。样式：

```
/usr/bin/expect<<-EOF
spawn ...
...
expect eof
EOF
```

注意：

由于expect在shell中是作为一个子部分而存在的，所以，一般情况下，expect结束的时候，使用eof命令表示expect的内容到此结束

查看脚本内容

```
[root@rocky9 ~]# cat expect_auto_login.sh
#!/bin/bash
# 功能：shell自动登录测试
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
host="$1"
username="$2"
password="$3"

/usr/bin/expect <<-EOF
# 发出连接进程
spawn ssh ${username}@${host}

# - 正常登陆
expect {
    "yes/no*" {  send "yes\n"; exp_continue  }
    "password:" {send "${password}\n";}
}
puts "测试完毕!!!"
expect eof
EOF
```

```
脚本测试效果
[root@rocky9 ~]# /bin/bash expect_auto_login.sh 10.0.0.12 sswang 123456
spawn ssh sswang@10.0.0.12
sswang@10.0.0.12's password: 测试完毕!!!

[sswang@localhost ~]$ exit
[root@rocky9 ~]#
```

1.6.4 综合案例

自动分区

简介

```
    当系统配置完毕后，我们可以采用fdisk命令对额外的磁盘进行磁盘分区。而expect可以实现这个效果。
```

```
一定要注意：脚本中的关键字，一定要是实际显示的信息
```

手工演示

```
[root@rocky9 ~]# fdisk /dev/sdc
欢迎使用 fdisk (util-linux 2.23.2)。

更改将停留在内存中，直到您决定将更改写入磁盘。
使用写入命令前请三思。

Device does not contain a recognized partition table
使用磁盘标识符 0x17fc4c8a 创建新的 DOS 磁盘标签。

命令(输入 m 获取帮助)：n                  # 输入n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p                  # 输入p
分区号 (1-4，默认 1)：                  # 输入 Enter
起始 扇区 (2048-41943039，默认为 2048)：# 输入 Enter
将使用默认值 2048
Last 扇区，+扇区 or +size{K,M,G} (2048-41943039，默认为 41943039)：# 输入Enter
将使用默认值 41943039
分区 1 已设置为 Linux 类型，大小设为 20 GiB

命令(输入 m 获取帮助)：wq                 # 输入wq
The partition table has been altered!

Calling ioctl() to re-read partition table.
正在同步磁盘。
[root@rocky9 ~]# mkfs -t ext4 /dev/sdc
mke2fs 1.42.9 (28-Dec-2013)
/dev/sdc is entire device, not just one partition!
无论如何也要继续? (y,n) y
文件系统标签=
OS type: Linux
块大小=4096 (log=2)
分块大小=4096 (log=2)
```

```
Stride=0 blocks, Stripe width=0 blocks
1310720 inodes, 5242880 blocks
262144 blocks (5.00%) reserved for the super user
第一个数据块=0
Maximum filesystem blocks=2153775104
160 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000

Allocating group tables: 完成
正在写入inode表: 完成
Creating journal (32768 blocks): 完成
Writing superblocks and filesystem accounting information: 完成

[root@rocky9 ~]# mkdir /haha
[root@rocky9 ~]# mount /dev/sdc /haha
[root@rocky9 ~]# echo nihao > /haha/h.txt
[root@rocky9 ~]# ls /haha
h.txt  lost+found
```

脚本实践

```
查看脚本内容
[root@rocky9 ~]# cat expect_auto_partition.sh
#!/bin/bash
# 功能：shell自动磁盘分区格式化测试
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
mount_dir='/disk_check'

# 检测基本环境
[ -f /usr/bin/expect ] && echo "expect 环境正常" || ("expect 环境异常" && exit)

# 查看磁盘列表
fdisk -l | grep "磁盘 /dev/s"

# 定制磁盘分区操作
read -p "请输入需要挂载得硬盘路径: " disk_name

# expect自动分区操作
/usr/bin/expect << EOF
set timeout 30
spawn bash -c "fdisk ${disk_name}"
expect "命令*" {send "n\r"}
expect "*default p*" {send "p\r"}
expect "*默认 1*" {send "\r"}
expect "起始 扇区*" {send "\r"}
expect "Last 扇区*" {send "\r"}
expect "命令*" {send "wq\r"}
expect eof
interact
```

```
EOF

# expect自动格式化操作
read -p "请输入硬盘的类型: " disk_type
/usr/bin/expect << EOF
set timeout 30
spawn bash -c "mkfs -t ${disk_type} ${disk_name}"
expect "*y,n*" {send "y\r"}
expect eof
interact
EOF

# 磁盘挂载测试
[ -d ${mount_dir} ] && rm -rf ${mount_dir}
mkdir ${mount_dir}
mount ${disk_name} ${mount_dir}
echo disk_check > ${mount_dir}/check.txt
[ -f ${mount_dir}/check.txt ] && echo "${mount_dir} 挂载成功" || (echo
"${mount_dir} 挂载失败" && exit)
umount ${mount_dir}
[ ! -f ${mount_dir}/check.txt ] && echo "${mount_dir} 卸载成功" || (echo
"${mount_dir} 卸载失败" && exit)
rm -rf ${mount_dir}
```

创建用户实践

案例需求

> 借助于expect实现在特定的主机上批量创建用户

脚本实践

```
查看脚本内容
[root@rocky9 ~]# cat expect_auto_register.sh
#!/bin/bash
# 功能：shell自动远程主机创建用户测试
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制初始变量
login_user='root'
login_pass='123456'
host_file='ip.txt'
new_user='sswang-1'
new_pass='123456'

# 批量创建用户
cat ${host_file} | while read ip
do
    expect <<-EOF
        set timeout 30
        spawn ssh $login_user@$ip
        expect {
            "yes/no" { send "yes\n";exp_continue }
            "password" { send "${login_pass}\n" }
        }
```

```
        expect "]#" { send "useradd ${new_user}\n" }
        expect "]#" { send "echo ${new_pass} |passwd --stdin ${new_user}\n" }
        expect "]#" { send "who\n" }
        expect "]#" { send "exit\n" }
        expect eof
    EOF
done
```

```
注意：
    关于 expect的匹配关键字，一定要在 目标主机上存在，否则就会出现异常。
    比如 ubuntu系统的关键字和rocky的关键字就不一样。
```

```
脚本执行后效果
[root@rocky9 ~]# echo 10.0.0.12 > ip.txt
[root@rocky9 ~]# /bin/bash expect_auto_register.sh
spawn ssh root@10.0.0.12
root@10.0.0.12's password:
[root@rocky9 ~]# useradd shuji-1
[root@rocky9 ~]# echo 123456 |passwd --stdin shuji-1
更改用户 shuji-1 的密码 。
passwd：所有的身份验证令牌已经成功更新。
[root@rocky9 ~]# who
root     pts/1        2022-06-26 08:47 (10.0.0.12)
root     pts/2        2022-06-26 07:15 (10.0.0.1)
[root@rocky9 ~]# exit
登出
Connection to 10.0.0.12 closed.
```

```
校验用户创建
[root@rocky9 ~]# id shuji-1
uid=1001(shuji-1) gid=1001(shuji-1) 组=1001(shuji-1)
```

1.6.5 高级赋值

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

简介

```
    所谓的高级赋值，是另外的一种变量值获取方法，这里涉及到更多我们学习之外的一些shell内置变量格式,其实这部分的内容主要还是在字符串的基础上，如何更精细的获取特定的信息内容：主要涉及到的内容样式如下：
```

```
字符串截取按分隔符截取: # 右  % 左
    ${file#/}       删除匹配结果，保留第一个/右边的字符串
    ${file##/}      删除匹配结果，保留最后一个/右边的字符串
    ${file%/}       删除匹配结果，保留第一个/左边的字符串
    ${file%%/}      删除匹配结果，保留最后一个/左边的字符串
    注意：
        匹配内容的正则表达式，尽量不要出现特殊边界字符
```

```
字符串替换
    ${file/dir/path}    把第一个dir替换成path：/path1/dir2/dir3/n
    ${file//dir/path}   把所有dir替换成path：/path1/path2/path3/n
    ${file/#dir/path}   将从左侧能匹配到的dir，则替换成 path 然后返回；否则直接返回
${var}。
    ${file/%dir/path}   将从右侧能匹配到的dir，则替换成 path 然后返回；否则直接返回
${var}。
    注意：
        如果匹配内容使用的是正则符号，应该注意正则符号的写法
```

```
字符串转换
    ${file^^}       把file中的所有小写字母转换为大写
    ${file,,}       把file中的所有大写字母转换为小写
```

简单实践

实践1-字符串截取

```
字符串截取示例
[root@rocky9 ~]# string=abc12342341
[root@rocky9 ~]# echo ${string#a*3}
42341
[root@rocky9 ~]# echo ${string#c*3}
abc12342341
[root@rocky9 ~]# echo ${string#*c1*3}
42341
[root@rocky9 ~]# echo ${string##a*3}
41
[root@rocky9 ~]# echo ${string%3*1}
abc12342
[root@rocky9 ~]# echo ${string%%3*1}
abc12
```

```
字符串截取赋值
[root@rocky9 ~]# file=/var/log/nginx/access.log
[root@rocky9 ~]# filename=${file##*/}
[root@rocky9 ~]# echo $filename
access.log
[root@rocky9 ~]# filedir=${file%/*}
[root@rocky9 ~]# echo $filedir
/var/log/nginx
```

实践2-字符串替换

```
字符串替换示例
[root@rocky9 ~]# str="apple, tree, apple tree, apple"
[root@rocky9 ~]# echo ${str/apple/APPLE}
APPLE, tree, apple tree, apple
[root@rocky9 ~]# echo ${str//apple/APPLE}
APPLE, tree, APPLE tree, APPLE
[root@rocky9 ~]# echo ${str/#apple/APPLE}
APPLE, tree, apple tree, apple
[root@rocky9 ~]# echo ${str/%apple/APPLE}
apple, tree, apple tree, APPLE
```

```
使用正则的情况下，代表尽可能多的匹配
[root@rocky9 ~]# file=dir1@dir2@dir3@n.txt
[root@rocky9 ~]# echo ${file/#d*r/DIR}
DIR3@n.txt
[root@rocky9 ~]# echo ${file/%3*/DIR}
dir1@dir2@dirDIR
```

实践3-字符串转换

```
[root@rocky9 ~]# str="apple, tree, apple tree, apple"
[root@rocky9 ~]# upper_str=${str^^}
[root@rocky9 ~]# echo ${upper_str}
APPLE, TREE, APPLE TREE, APPLE
[root@rocky9 ~]# lower_str=${upper_str,,}
[root@rocky9 ~]# echo ${lower_str}
apple, tree, apple tree, apple
```

1.6.6 嵌套变量

基础知识

场景现象

```
场景1：我们知道，命令变量的的表现样式：
    ver=$(命令)
    -- 执行原理是，当`` 或者 $() 范围中存在能够正常解析的命令的话，会先执行命令，然后将命令执行的结果交个一个变量名。
场景2：它还有另外一种样式 -- 普通变量的第三种样式双引号
    ming=shuji; name="wang-$ming"
    -- 解析原理：双引号会首先查看变量值范围内是否有可以解析的变量名，如果有的话，将解析后的结果放到变量值范围内，组合成一个新的变量值，然后交给变量名。
```

```
    上面的两种场景的特点就在于，一个命令行中，借助于$() 或者 "" 发起一次隐藏的命令执行，但是有些场景下，表面上的一个命令需要发起更深一层的命令执行才可以实现指定的功能。在这种场景下，无论是$() 还是 ""都无法满足要求了
```

场景示例

```
循环遍历演示
[root@rocky9 ~]# for i in {1..10}; do  echo "$i "; done
1
2
3
4
5
6
7
8
9
10
示例解读：这里出现一层隐藏命令执行
    1 {1..10} 会自动进行命令解析
        [root@rocky9 ~]# echo {1..10}
```

```
        1 2 3 4 5 6 7 8 9 10
    然后执行for命令
        for i in 1 2 3 4 5 6 7 8 9 10
```

```
双层隐藏命令解读
[root@rocky9 ~]# n=10
[root@rocky9 ~]# for i in {1..$n}; do  echo "$i "; done
{1..10}

示例解读：
    在for语句中，其实我们的目的与上面的演示一样，但是区别在于这里有两层隐藏的命令执行
    1 $n 需要解析成 10
    2 {1..10} 需要解析成 1 2 3 4 5 6 7 8 9 10
    最后执行 for命令 for 1 2 3 4 5 6 7 8 9 10
```

```
问题：
    linux命令行，默认情况下是无法执行两层隐藏命令的执行的，在有些场景中，我们可以通过$() 来实现多层命令的解读，示例如下：
    [root@rocky9 ~]# cmd=who
    [root@rocky9 ~]# echo $(${cmd}ami)
    root
    但是，这里我们无法实现，因为 {1..10} 不是命令。
    [root@rocky9 ~]# for i in $({1..$n}); do  echo "$i "; done
    -bash: {1..10}: 未找到命令
```

解决方法

```
    在shell中，它提供了一个专属的命令，可以实现多层隐藏命令的解析，不仅仅能够解析，还能够将相关环境的属性重现，从而实现多层隐藏命令的顺利执行。这个命令就是 eval。
```

```
eval原理
    1 eval命令将会首先扫描命令行整体
    2 发现解析则解析，发现执行则预先执行，实现所有隐藏命令的成功执行
    3 将隐藏命令执行的最终结果进行置换
    4 最后执行命令行表面的命令。
```

简单实践

实践1-eval简单实践

```
for循环演示
[root@rocky9 ~]# n=10
[root@rocky9 ~]# for i in $(eval echo {1..$n}); do  echo "$i "; done
1
2
3
4
5
6
7
8
9
10
```

```
示例解读
    1 命令改造$(eval echo {1..$n})
        1-1 $n先解析为10，命令替换为 {1..10}
        1-2 通过 eval 带入 echo 命令环境
        1-3 $() 执行 echo {1..10} 输出为 1 2 3 4 5 6 7 8 9 10
    2 整体置换命令结果
        for i in 1 2 3 4 5 6 7 8 9 10
```

实践2-eval的命令扩展演示

```
查看文件内容
[root@rocky9 ~]# echo 'hello-in-world' > infile.txt
[root@rocky9 ~]# cat infile.txt
hello-in-world

脚本内容演示
[root@rocky9 ~]# echo 'hello-in-world' > infile.txt
[root@rocky9 ~]# cat infile.txt
hello-in-world
[root@rocky9 ~]# cmd="cat infile.txt"
[root@rocky9 ~]# echo $(${cmd})
hello-in-world
[root@rocky9 ~]# echo ${cmd}
cat infile.txt

不是我们想要的，我们可以借助于eval 和 $() 方式来实现隐藏命令的解读
[root@rocky9 ~]# eval ${cmd}
hello-in-world
[root@rocky9 ~]# echo $(${cmd})
hello-in-world
```

实践3-eval变量名的预制解析

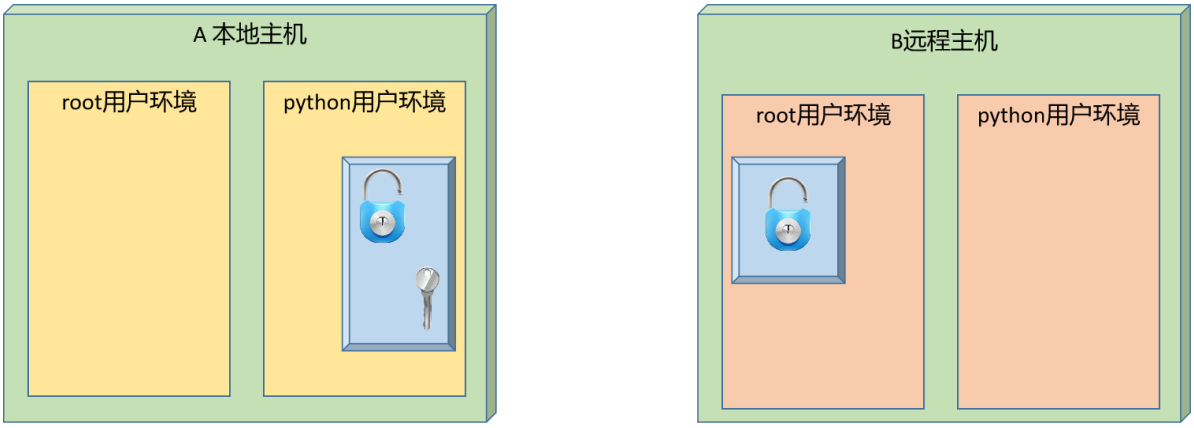
```
定制嵌套的环境变量
[root@rocky9 ~]# str=a
[root@rocky9 ~]# num=1
[root@rocky9 ~]# $str$num=hello
-bash: a1=hello: 未找到命令

借助于eval命令来实现
[root@rocky9 ~]# eval $str$num=hello
[root@rocky9 ~]# echo $a1
hello

借助于eval实现变量名的嵌套
[root@rocky9 ~]# eval $str=$a1
[root@rocky9 ~]# echo $a
hello
解读：
    $str 就是 a，$a1就是hello，
    eval执行的命令就是  a=hello
```

1.6.7 综合案例

免密认证

案例需求

> A 以主机免密码认证 连接到 远程主机B



> 我们要做主机间免密码认证需要做三个动作
>
> 1、本机生成密钥对
>
> 2、对端机器使用公钥文件认证
>
> 3、验证

手工演示

本地主机生成秘钥对

```
[root@rocky9 ~]# ssh-keygen -t rsa -P "" -f ~/.ssh/id_rsa
Generating public/private rsa key pair.
Created directory '/root/.ssh'.
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:Ncra/fPpaVs+M18l9Kn7CQq33zmWQSoJ/ujuugCkNjM root@localhost
The key's randomart image is:
+---[RSA 2048]----+
|                 |
|                 |
|   .      o   .  |
|  o    . + . . o.|
| E .    S . . +.o|
|. + .  o o o ..o.|
|     .. ..+..o  =|
|      .  .oo+ =%+|
|       o*+ ooBO*O|
+----[SHA256]-----+
```

将公钥信息传递给远程主机的指定用户

```
[root@rocky9 ~]# ssh-copy-id -i /root/.ssh/id_rsa.pub root@10.0.0.12
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/root/.ssh/id_rsa.pub"
The authenticity of host '10.0.0.12 (10.0.0.12)' can't be established.
ECDSA key fingerprint is SHA256:XUJsgk4cTORxdcswxIKBGFgrrqFQzpHmKnRRV6ABMk4.
ECDSA key fingerprint is MD5:71:74:46:50:3f:40:4e:af:ad:d3:0c:de:2c:fc:30:c0.
```

```
Are you sure you want to continue connecting (yes/no)? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter
out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
prompted now it is to install the new keys
root@10.0.0.12's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'root@10.0.0.12'"
and check to make sure that only the key(s) you wanted were added.
```

```
本地主机测试验证效果
[root@rocky9 ~]# ssh root@10.0.0.12 "ifconfig eth0 | grep netmas"
        inet 10.0.0.12  netmask 255.255.255.0  broadcast 10.0.0.255
```

简单实践

```
定制脚本文件 remotehost_sshkey_auth.sh
#!/bin/bash
# 功能：设置ssh跨主机免密码认证
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
user_dir="/root"
login_uesr='root'
login_pass='123456'

# 定制数组变量
target_type=(部署 免密 退出)

# 定制安装软件的函数
expect_install(){
    yum install expect -y >> /dev/null
    echo "软件安装完毕"
}

# 定制ssh秘钥对的生成
sshkey_create(){
    # 清理历史秘钥
    [ -d ${user_dir}/.ssh ] && rm -rf ${user_dir}/.ssh
    # 生成新的秘钥
    ssh-keygen -t rsa -P "" -f ${user_dir}/.ssh/id_rsa >> /dev/null
    echo "秘钥生成完毕"
}

# 定制expect的认证逻辑
expect_process(){
    # 注意：这里不要乱用$1,可以参考函数和脚本间的数组传参
    command="$@"
    expect -c "
        spawn ${command}
        expect {
            \"*yes/no*\" {send \"yes\r\"; exp_continue}
```

```
            \"*password*\" {send \"${login_pass}\r\"; exp_continue}
            \"*Password*\" {send \"${login_pass}\r\";}
        }"
}

# 跨主机密码认证
sshkey_auth(){
    local host_list="$1"
    for i in ${host_list}
    do
        command="/usr/bin/ssh-copy-id -i /root/.ssh/id_rsa.pub"
        remote="${login_uesr}@$i"
        expect_process ${command} ${remote}
    done
}

# 定制服务的操作提示功能函数
menu(){
    echo -e "\e[31m---------------管理平台操作界面---------------"
    echo -e " 1: 秘钥准备  2: 免密认证  3: 退出操作"
    echo -e "-----------------------------------------\033[0m"
}

# 定制脚本帮助信息
Usage(){
    echo "请输入有效的操作标识!!!"
}

# 定制业务逻辑
while true
do
    menu
    read -p "> 请输入要操作的目标类型: " target_id
    if [ ${target_type[$target_id-1]} == "部署" ];then
        echo "开始部署秘钥环境..."
        expect_install
        sshkey_create
    elif [ ${target_type[$target_id-1]} == "免密" ];then
        read -p "> 请输入免密10.0.0网段主机的范围，示例{12..19}: " num_list
        # eval的隐藏命令解析，仅接收{12..13} 格式，不接受 12 13 风格
        ip_list=$(eval echo 10.0.0.${num_list})
        sshkey_auth ${ip_list}
    elif [ ${target_type[$target_id-1]} == "退出" ];then
        echo "准备退出管理操作界面..."
        exit
    else
        Usage
    fi
done
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash remotehost_sshkey_auth.sh
---------------管理平台操作界面---------------
 1: 秘钥准备  2: 免密认证  3: 退出操作
-----------------------------------------
> 请输入要操作的目标类型: 1
开始部署秘钥环境...
```

```
软件安装完毕
秘钥生成完毕
---------------管理平台操作界面---------------
 1: 秘钥准备   2: 免密认证   3: 退出操作
----------------------------------------------
> 请输入要操作的目标类型: 2
> 请输入免密10.0.0网段主机的范围，示例{12..19}: {12..13}
spawn /usr/bin/ssh-copy-id -i /root/.ssh/id_rsa.pub root@10.0.0.12
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/root/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter
out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
prompted now it is to install the new keys
root@10.0.0.12's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'root@10.0.0.12'"
and check to make sure that only the key(s) you wanted were added.

---------------管理平台操作界面---------------
 1: 秘钥准备   2: 免密认证   3: 退出操作
----------------------------------------------
> 请输入要操作的目标类型: 3
准备退出管理操作界面...
```

1.7 编程进阶(拓展)

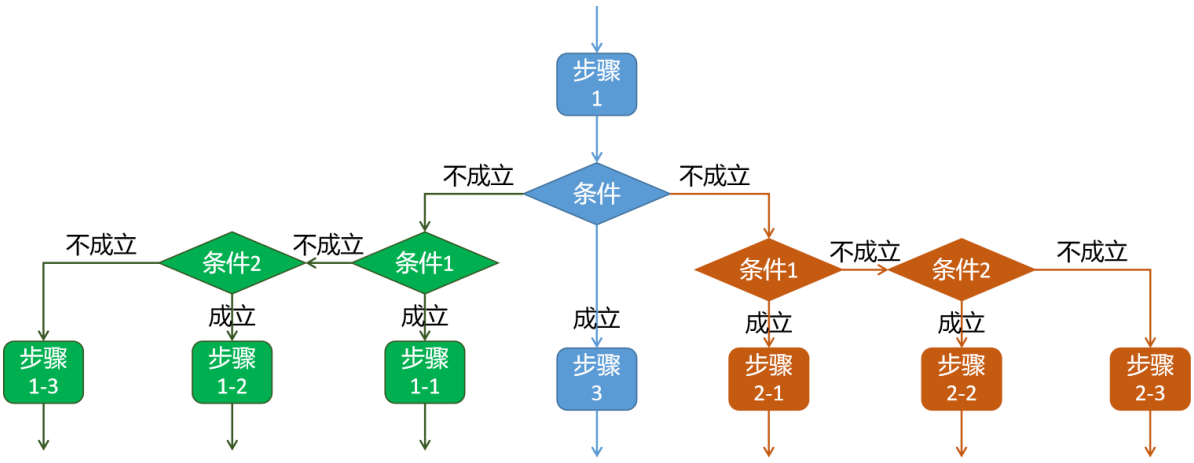
这里面是shell编程大量的应用案例技巧，大家可以根据自己的实际学习能力进行选择。

1.7.1 if嵌套

基础知识

简介

> 一个if语句仅仅能够针对一个场景的多种情况。当我们面对多场景的条件判断的时候，一个if结构语句无法满足需求，这个时候，我们可以借助于嵌套if的结构语句来实现相应的效果。



注意：
    如果是多个独立的、彼此没有关联的业务场景，可以使用不同的if语句即可
    嵌套的if语句只能适用于彼此关联的业务场景

简单实践

案例需求

运维管理人员，通过监控平台获取站点的运行状态数据信息，当发现问题的时候，根据情况进行后续判断：
    状况1： 问题类型
        研发标识 - 交给开发团队
        测试标识 - 交给测试团队
        运维标识 - 交给运维团队
    状况2： 问题级别
        红灯 - 紧急故障
        黄灯 - 严重故障
        绿灯 - 一般故障
        灰灯 - 未知故障，后续操作

脚本内容

```
[root@rocky9 ~]# cat monitor_operator_if.sh
#!/bin/bash
# 功能：定制监控异常的处理措施
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
monitor_type=(研发 测试 运维)
error_level=(红灯 黄灯 绿灯 灰灯)

# 监控平台的信息提示
echo -e "\e[31m \t\t 欢迎使用监控处理平台"
echo -e "\e[32m-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
---------------------------------\033[0m"
# 定制业务逻辑
read -p "请输入问题标识: " monitor_id
# 判断问题类型是否有效
if [ ${#monitor_type[@]} -lt ${monitor_id} ]
then
    echo -e "\e[31m无效标识，请输入正确的问题标识\e[0m"
else
    # 定制问题类型识别逻辑
    if [ ${monitor_type[$monitor_id-1]} == "研发" ]
    then
        echo -e "\e[31m转交研发团队处理\e[0m"
    elif [ ${monitor_type[$monitor_id-1]} == "测试" ]
    then
        echo -e "\e[31m转交测试团队处理\e[0m"
    elif [ ${monitor_type[$monitor_id-1]} == "运维" ]
    then
        echo -e "\e[32m-----------请选择故障级别-----------"
        echo " 1: 红灯 2: 黄灯 3: 绿灯 4: 灰灯"
        echo -e "---------------------------------\033[0m"
```

```
        read -p "请输入故障级别: " level_id
        # 判断故障级别是否有效
        if [ ${#error_level[@]} -lt ${level_id} ]
        then
            echo -e "\e[31m无效标识，请输入正确的故障级别\e[0m"
        else
            # 定制故障级别逻辑
            if [ ${error_level[$level_id-1]} == "红灯" ]
            then
                echo -e "\e[31m请按照 紧急故障 性质进行处理\e[0m"
            elif [ ${error_level[$level_id-1]} == "黄灯" ]
            then
                echo -e "\e[31m请按照 严重故障 性质进行处理\e[0m"
            elif [ ${error_level[$level_id-1]} == "绿灯" ]
            then
                echo -e "\e[31m请按照 一般故障 性质进行处理\e[0m"
            elif [ ${error_level[$level_id-1]} == "灰灯" ]
            then
                echo -e "\e[31m请按照 未知故障 性质进行处理\e[0m"
            fi
        fi
    fi
fi
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash monitor_operator_if.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 4
无效标识，请输入正确的问题标识
[root@rocky9 ~]# /bin/bash monitor_operator_if.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 1
转交研发团队处理
[root@rocky9 ~]# /bin/bash monitor_operator_if.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 3
-----------请选择故障级别-----------
 1: 红灯 2: 黄灯 3: 绿灯 4: 灰灯
----------------------------------
请输入故障级别: 1
请按照 紧急故障 性质进行处理
[root@rocky9 ~]# /bin/bash monitor_operator_if.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 3
-----------请选择故障级别-----------
```

```
 1: 红灯 2: 黄灯 3: 绿灯 4: 灰灯
---------------------------------
请输入故障级别: 6
无效标识，请输入正确的故障级别
```

1.7.2 if条件进阶

特殊条件

简介

```
    if条件控制语句支持(())来代替let命令测试表达式的执行效果，支持[[]]实现正则级别的内容匹配。
```

```
表达式的样式如下：
if (( 表达式 ))  或 [[ 内容匹配 ]]
then
    指令
fi
```

实践1-(()) 计算条件匹配

```
查看脚本内容
[root@rocky9 ~]# cat odd_even_if.sh
#!/bin/bash
# (()) 在if分支中的应用

# 接收一个数字
read -p "请输入一个数字: " num

# 定制数字奇数和偶数判断
if (( ${num} % 2 == 0 ))
then
    echo -e "\e[31m ${num} 是一个偶数\e[0m"
else
    echo -e "\e[31m ${num} 是一个奇数\e[0m"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash odd_even_if.sh
请输入一个数字: 2
 2 是一个偶数
[root@rocky9 ~]# /bin/bash odd_even_if.sh
请输入一个数字: 1
 1 是一个奇数
```

实践2-[[]]扩展匹配条件

```
查看脚本内容
[root@rocky9 ~]# cat condition_extend_if.sh
#!/bin/bash
# [[]] 在if分支中的应用

# 接收一个数字
read -p "请输入一个单词: " string
```

```
# 判断内容是否是我们想要的
if [[ ${string} == v* ]]
then
    echo -e "\e[31m ${num} 是满足条件的单词\e[0m"
else
    echo -e "\e[31m ${num} 不满足条件\e[0m"
fi
```

```
执行脚本效果
[root@rocky9 ~]# /bin/bash condition_extend_if.sh
请输入一个单词: very
  是满足条件的单词
[root@rocky9 ~]# /bin/bash condition_extend_if.sh
请输入一个单词: hello
  不满足条件
```

实践3-[[]]扩展实践

```
#!/bin/bash
# [[]] 在if分支中的应用

# 接收一个数字
read -p "请输入你的身高(m为单位): " height

# 身高判断逻辑
if [[ ! ${height} =~ ^[0-2](\.[0-9]{,2})?$ ]]
then
    echo -e "\e[31m你确定自己的身高是 ${height} ?\e[0m"
else
    echo -e "\e[31m我说嘛，${height} 才是你的身高!\e[0m"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash person_height_if.sh
请输入你的身高(m为单位): 3
你确定自己的身高是 3 ?
[root@rocky9 ~]# /bin/bash person_height_if.sh
请输入你的身高(m为单位): 1.2
我说嘛，1.2 才是你的身高!
```

单行命令

简介

```
所谓的单行命令，其实指的是对于简单的if条件判断，我们可以直接一行显示，减少代码的行数
```

简单实践

```
示例1-命令行的if语句
[root@rocky9 ~]# num1=3 num2=1
[root@rocky9 ~]# if [ $num1 -gt $num2 ]; then echo "$num1 数字大"; else echo
"$num2 数字大"; fi
3 数字大
[root@rocky9 ~]# num1=3 num2=9
[root@rocky9 ~]# if [ $num1 -gt $num2 ]; then echo "$num1 数字大"; else echo
"$num2 数字大"; fi
9 数字大
```

```
示例2-使用逻辑表达式来满足if效果
[root@rocky9 ~]# num1=3 num2=1
[root@rocky9 ~]# [ $num1 -gt $num2 ] && echo "$num1 数字大" || echo "$num2 数字大"
3 数字大
[root@rocky9 ~]# num1=3 num2=9
[root@rocky9 ~]# [ $num1 -gt $num2 ] && echo "$num1 数字大" || echo "$num2 数字大"
9 数字大
```

1.7.3 case嵌套1

基础知识

简介

```
这里的嵌套实践，与if语句的嵌套实践，基本一致，只不过组合的方式发生了一些变化。常见的组合样式如
下：
```

```
case嵌套if语句
    case "变量" in
        "值1")
            if [ 条件判断 ]
            ...
            ;;
        ...
    esac
```

```
case嵌套case语句
    case "变量" in
        "值1")
            case语句
        ...
    esac
```

```
if嵌套case语句
    if [ 条件判断 ]
    then
        case "变量名" in
           "值1")
              指令1;;
           ...
        esac
    else
        ...
    fi
```

简单实践

案例需求

```
场景简介：
    火车站安检
        安检失败
            - 携带违禁物品禁止进站
        安检成功后进行火车票检查
            - 允许登上上火车
            - 禁止登上火车
```

实践1-case嵌套if语句

```
查看脚本内容
[root@rocky9 ~]# cat case_if.sh
#!/bin/bash
# 功能：case嵌套if语句实践

# 定制业务逻辑
read -p "是否携带违禁物品： " safe_check
# 业务逻辑定制
case "${safe_check}" in
    "true")
        echo -e "\e[31m不允许进入火车站\e[0m";;
    "false")
        read -p "火车票是否过期： " ticket_check
        if [ ${ticket_check} == "true" ]
        then
            echo -e "\e[31m不允许登上火车站\e[0m"
        else
            echo -e "\e[31m允许登上火车站\e[0m"
        fi;;
    *)
        echo -e "\e[31m再检查一遍\e[0m";;
esac
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash case_if.sh
是否携带违禁物品： true
不允许进入火车站
[root@rocky9 ~]# /bin/bash case_if.sh
是否携带违禁物品： false
```

```
火车票是否过期: true
不允许登上火车站
[root@rocky9 ~]# /bin/bash case_if.sh
是否携带违禁物品: false
火车票是否过期: false
允许登上火车站
[root@rocky9 ~]# /bin/bash case_if.sh
是否携带违禁物品: hah
再检查一遍
```

实践2-case嵌套case语句

```
查看脚本内容
[root@rocky9 ~]# cat case_case.sh
#!/bin/bash
# 功能：case嵌套case语句实践

# 定制业务逻辑
read -p "是否携带违禁物品: " safe_check
# 业务逻辑定制
case "${safe_check}" in
    "true")
        echo -e "\e[31m不允许进入火车站\e[0m";;
    "false")
        read -p "火车票是否过期: " ticket_check
        case ${ticket_check} in
            "true")
                echo -e "\e[31m不允许登上火车站\e[0m";;
            "false")
                echo -e "\e[31m允许登上火车站\e[0m";;
            *)
                echo -e "\e[31m再检查一遍\e[0m";;
        esac;;
    *)
        echo -e "\e[31m再检查一遍\e[0m";;
esac
```

实践3-if嵌套case语句

```
查看脚本内容
[root@rocky9 ~]# cat if_case.sh
#!/bin/bash
# 功能：if嵌套case语句实践

# 定制业务逻辑
read -p "是否携带违禁物品: " safe_check
# 业务逻辑定制
if [ "${safe_check}" == "true" ]
then
    echo -e "\e[31m不允许进入火车站\e[0m"
elif [ "${safe_check}" == "false" ]
then
    read -p "火车票是否过期: " ticket_check
    case ${ticket_check} in
        "true")
            echo -e "\e[31m不允许登上火车站\e[0m";;
```

```
        "false")
            echo -e "\e[31m允许登上火车站\e[0m";;
        *)
            echo -e "\e[31m再检查一遍\e[0m";;
    esac
else
    echo -e "\e[31m再检查一遍\e[0m"
fi
```

1.7.4 case嵌套2

案例需求

```
运维管理人员，通过监控平台获取站点的运行状态数据信息，当发现问题的时候，根据情况进行后续判断：
    状况1： 问题类型
        研发标识 - 交给开发团队
        测试标识 - 交给测试团队
        运维标识 - 交给运维团队
    状况2： 问题级别
        红灯 - 紧急故障
        黄灯 - 严重故障
        绿灯 - 一般故障
        灰灯 - 未知故障，后续操作
```

简单实践

实践-改造嵌套if的监控管理脚本

```
查看脚本内容
[root@rocky9 ~]# cat monitor_operator_if_case.sh
# 功能：定制监控异常的处理措施
# 版本：v0.2
# 作者：王树森
# 联系：sswang.magedu.com

# 定制普通变量
monitor_type=(研发 测试 运维)
error_level=(红灯 黄灯 绿灯 灰灯)

# 监控平台的信息提示
echo -e "\e[31m \t\t 欢迎使用监控处理平台"
echo -e "\e[32m-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
---------------------------------\033[0m"
# 定制业务逻辑
read -p "请输入问题标识: " monitor_id
# 判断问题类型是否有效
if [ ${#monitor_type[@]} -lt ${monitor_id} ]
then
    echo -e "\e[31m无效标识，请输入正确的问题标识\e[0m"
else
    # 定制问题类型识别逻辑
    case ${monitor_type[$monitor_id-1]} in
        "研发")
            echo -e "\e[31m转交研发团队处理\e[0m";;
        "测试")
```

```
            echo -e "\e[31m转交测试团队处理\e[0m";;
        "运维")
            echo -e "\e[32m-----------请选择故障级别-----------"
            echo " 1: 红灯 2: 黄灯 3: 绿灯 4: 灰灯"
            echo -e "----------------------------------\033[0m"
            read -p "请输入故障级别: " level_id
            # 定制故障级别逻辑
            case ${error_level[$level_id-1]} in
                "红灯")
                    echo -e "\e[31m请按照 紧急故障 性质进行处理\e[0m";;
                "黄灯")
                    echo -e "\e[31m请按照 严重故障 性质进行处理\e[0m";;
                "绿灯")
                    echo -e "\e[31m请按照 一般故障 性质进行处理\e[0m";;
                "灰灯")
                    echo -e "\e[31m请按照 未知故障 性质进行处理\e[0m";;
                *)
                    echo -e "\e[31m无效标识，请输入正确的故障级别\e[0m";;
            esac
    esac
fi
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash monitor_operator_if_case.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 1
转交研发团队处理
[root@rocky9 ~]# /bin/bash monitor_operator_if_case.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 4
无效标识，请输入正确的问题标识

[root@rocky9 ~]# /bin/bash monitor_operator_if_case.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 3
-----------请选择故障级别-----------
 1: 红灯 2: 黄灯 3: 绿灯 4: 灰灯
----------------------------------
请输入故障级别: 3
请按照 一般故障 性质进行处理
[root@rocky9 ~]# /bin/bash monitor_operator_if_case.sh
                  欢迎使用监控处理平台
-----------请选择问题类型-----------
 1: 研发 2: 测试 3: 运维
----------------------------------
请输入问题标识: 3
-----------请选择故障级别-----------
 1: 红灯 2: 黄灯 3: 绿灯 4: 灰灯
```

```
----------------------------------
请输入故障级别: 6
无效标识，请输入正确的故障级别
```

1.7.4 for (())

基础实践

简介

```
    在for循环的语法中，它还支持一种包含赋值+循环双功能的语法，也就是双小括号(())，这种语法的语法格式如下：
    样式1：  单元素样式
        for (( i=0; i<10; i++ ))
    样式2：  多元素样式
        for (( i=0,j=0; i<10; i++,j++ ))
```

```
功能解读
    1 第一部分定制一个包含初始值的变量名
    2 第二部分定制循环的结束条件
    3 第三部分定制循环过程中的变量变化效果
        - 为了让循环出现结束
```

```
注意事项：
    1 变量的复制可以包含空格
    2 条件中的变量不能用$符号
    3 第三部分的数据操作过程不用 expr格式
```

简单实践

实践1-(())简单使用

```
输出1-5的数字
[root@rocky9 ~]# for ((i=1;i<=5;i++));do echo $i;done
1
2
3
4
5

输出1-10中的所有奇数
[root@rocky9 ~]# for ((i=1;i<=10;i+=2));do echo $i;done
1
3
5
7
9

输出1-10中的所有偶数
[root@rocky9 ~]# for ((i=2;i<=10;i+=2));do echo $i;done
2
4
6
8
10
```

实践2-100个数字的求和

```
查看脚本内容
[root@rocky9 ~]# cat for_odd_num.sh
#!/bin/bash
# 功能：for统计数据之和

# 定制普通变量
all_sum=0
odd_sum=0

# 定制所有数据求和逻辑
for ((i=1;i<=100;i++))
do
  let all_sum+=i
done

# 定制所有奇数求和逻辑
for ((i=1;i<=100;i+=2))
do
  let odd_sum+=i
done

# 信息输出
echo -e "\e[31m所有数据之和: ${all_sum}\e[0m"
echo -e "\e[31m所有奇数之和: ${odd_sum}\e[0m"

脚本执行后效果
[root@rocky9 ~]# /bin/bash for_odd_sum.sh
所有数据之和: 5050
所有奇数之和: 2500
```

1.7.5 for(())案例

进度条案例

实践1-命令行进度条数字

```
[root@rocky9 ~]# for ((i = 0; i<=100; ++i)); do printf "\e[4D%3d%%" $i;sleep
0.1s; done
 27%
... ...
[root@rocky9 ~]# for ((i = 0; i<=100; ++i)); do printf "\e[4D%3d%%" $i;sleep
0.1s; done
 99%

命令解读：
    %3d%% 指的是 3个数字位置 + 1个%位置，共计4个位置
    防止信息输出的叠加，采用\e[4D，每次输出信息的时候，光标左移4个位置，信息不会出现叠加

\e[3D 的演示
[root@rocky9 ~]# for ((i = 0; i <= 100; ++i)); do printf "\e[2D%3d%%" $i;sleep
0.1s; done
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 56 5 5 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9 9100%[root@rocky9 ~]#
```

实践2-脚本生成进度条

查看文件内容

```
[root@rocky9 ~]# cat progress_bar_for.sh
#!/bin/bash
# 定制简单的进度条

# 定制进度条的进度符号
str="#"
# 定制进度转动提示符号,注意\\转义
arr=("|" "/" "-" "\\")

# 定制进度条循环控制
for ((i=0; i<=50; i++))
do
    # 设定数组信息的变化索引
    let index=i%4
    # 打印信息，格式：【%s进度符号】【%d进度数字】【%c进度进行中】
    # 注意：信息的显示宽度和进度的数字应该适配，否则终端显示不全
    printf "[%-50s][%d%%]%c\r" "$str" "$(($i*2))" "${arr[$index]}"
    # 进度的频率
    sleep 0.2
    # 进度符前进
    str+="#"
done
printf "\n"
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash progress_bar_for.sh
[##################################################][100%]-
```

实践3-生成10个随机数保存于数组中，并找出其最大值和最小值

查看脚本内容

```
[root@rocky9 ~]# cat compare_nums_for.sh
#!/bin/bash
# 设定随机数比大小

# 设定基本变量
declare -i min max
declare -a nums

# 设定大小比较
for ((i=0;i<10;i++))
do
    # 将随机数添加到数组中
    nums[$i]=$RANDOM
    # 设定初始值
    [ $i -eq 0 ] && min=${nums[0]} max=${nums[0]}
    # 设定最大值
    [ ${nums[$i]} > $max ] && max=${nums[$i]}
    # 设定最小值
    [ ${nums[$i]} < $min ] && min=${nums[$i]}
done
echo -e "\e[31m    随机数的统计信息\e[0m"
echo "------------------------------"
```

```
echo -e "\e[32m所有的随机数：${nums[@]}"
echo -e "最大的随机数：${max}"
echo -e "最小的随机数：${min}\e[0m"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash compare_num_for.sh.sh
    随机数的统计信息
------------------------------
所有的随机数：32506 31542 6495 11273 8532 5789 22139 25904 16252 7357
最大的随机数：7357
最小的随机数：32506
```

1.7.6 for嵌套

基础知识

简介

```
这里的嵌套实践，与选择语句的嵌套实践基本一致，只不过组合的方式发生了一些变化。常见的组合样式如下：
```

```
for嵌套for语句
    for 循环列表1
    do
        for 循环列表2
        do
            ...
        done
    done
```

```
for嵌套if|case语句
    for 循环列表
    do
        if 条件判断语句
        或
        case 条件判断语句
    done
```

```
if语句嵌套for语句
    if 条件判断
    then
        for 循环列表语句
    fi

case语句嵌套for语句
    case 条件判断
        for 循环列表语句
        ;;
    esac
```

简单实践

for嵌套for语句实践1-输出99乘法表

查看脚本内容

```
[root@rocky9 ~]# cat for_nine_table.sh
#!/bin/bash
# 功能：for打印99乘法表

# 定制打印99乘法表的业务逻辑
# 对第一个位置的数字进行循环
for num1 in {1..9}
do
    # 对第二个位置的数字进行循环
    for num2 in $(seq $num1)
    do
        # 信息输出,\t\c 的目的是删除后续信息，生成的内容是固定长度
        echo -e "\e[$[RANDOM%9+31]m${num1} x ${num2} = $[num1*num2]\e[0m\t\c"
    done
    echo # 一个子循环一行内容
done
```

脚本执行效果

```
[root@localhost ~]# /bin/bash for_nine_table.sh
1x1=1
2x1=2   2x2=4
3x1=3   3x2=6   3x3=9
4x1=4   4x2=8   4x3=12  4x4=16
5x1=5   5x2=10  5x3=15  5x4=20  5x5=25
6x1=6   6x2=12  6x3=18  6x4=24  6x5=30  6x6=36
7x1=7   7x2=14  7x3=21  7x4=28  7x5=35  7x6=42  7x7=49
8x1=8   8x2=16  8x3=24  8x4=32  8x5=40  8x6=48  8x7=56  8x8=64
9x1=9   9x2=18  9x3=27  9x4=36  9x5=45  9x6=54  9x7=63  9x8=72  9x9=81
[root@localhost ~]#
```

for嵌套if语句实践2-判断目录中的文件类型

准备工作

```
mkdir dir/{server,soft,scripts,logs} -p
touch dir/{{a..c}.sh,{1..3}.txt}
```

查看脚本内容

```
[root@rocky9 ~]# cat file_type_for_if.sh
#!/bin/bash
# 功能：for嵌套if查看目录下的文件类型

# 定制普通变量
dir_name='dir'

# 获取所有文件列表
for file in $(ls dir)
do
    # 判断文件类型
    if [ -d ${dir_name}/${file} ]
    then
        echo -e "\e[31m${dir_name}/${file} 是一个目录文件\e[0m"
    elif [ -f ${dir_name}/${file} ]
    then
```

```
        echo -e "\e[31m${dir_name}/${file} 是一个普通文件\e[0m"
    fi
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash file_type_for_if.sh
dir/1.txt 是一个普通文件
dir/2.txt 是一个普通文件
dir/3.txt 是一个普通文件
dir/a.sh 是一个普通文件
dir/b.sh 是一个普通文件
dir/c.sh 是一个普通文件
dir/logs 是一个目录文件
dir/scripts 是一个目录文件
dir/server 是一个目录文件
dir/soft 是一个目录文件
```

```
收尾动作
[root@rocky9 ~]# rm -rf dir
```

if嵌套for语句实践3-获取系统支持的shell类型

```
查看脚本内容
[root@rocky9 ~]# cat os_shell_if_for.sh
#!/bin/bash
# 功能：if嵌套for查看系统支持的shell类型

# 定制普通变量
shell_file='/etc/shells'

# 获取所有文件列表
if [ -f ${shell_file} ]
then
    for shell in $(grep sh /etc/shells)
    do
        echo -e "\e[31m当前系统支持的shell类型有：${shell}\e[0m"
    done
else
    echo -e "\e[31m没有 ${shell_file} 文件\e[0m"
fi
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash os_shell_if_for.sh
当前系统支持的shell类型有：/bin/sh
当前系统支持的shell类型有：/bin/bash
当前系统支持的shell类型有：/usr/bin/sh
当前系统支持的shell类型有：/usr/bin/bash
```

1.7.7 for案例

信息收集

案例需求

```
    根据提示信息，选择输出 cpu 或者 内存信息。
```

脚本实践-采集系统负载信息

```
查看脚本内容
[root@rocky9 ~]# cat systemctl_load.sh
#!/bin/bash
# 功能：采集系统负载信息
# 版本：v0.2
# 作者：王树森
# 联系：sswang.magedu.com

# 定制资源类型
resource_type=(CPU MEM)
cpu_attribute=(1 5 15)
free_attribute=(总量 使用 空闲)

# 获取相关的属性信息
cpu_load=($(uptime | tr -s " " | cut -d " " -f 11-13 | tr "," " "))
free_info=($(free -m | grep Mem | tr -s " " | cut -d " " -f 2-4))
# 服务的操作提示
echo -e "\e[31m---------------查看资源操作动作---------------
 1: CPU  2: MEM
-----------------------------------------"'\033[0m'
# 选择服务操作类型
read -p "> 请输入要查看的资源信息类型: " resource_id
echo
if [ ${resource_type[$resource_id-1]} == "CPU" ]
then
    echo -e "\e[31m\t系统CPU负载信息\e[0m"
    echo -e "\e[32m================================="
    for index in ${!cpu_attribute[@]}
    do
        echo "CPU ${cpu_attribute[$index]} min平均负载为: ${cpu_load[$index]}"
    done
    echo -e "=================================\e[0m"
elif [ ${resource_type[$resource_id-1]} == "MEM" ]
then
    echo -e "\e[31m\t系统内存负载信息\e[0m"
    echo -e "\e[32m================================="
    for index in ${!free_attribute[@]}
    do
        echo "内存 ${free_attribute[$index]} 信息为: ${free_info[$index]} M"
    done
    echo -e "=================================\e[0m"
fi
```

```
脚本使用效果
[root@rocky9 ~]# /bin/bash systemctl_load.sh
---------------查看资源操作动作---------------
 1: CPU  2: MEM
-----------------------------------------
> 请输入要查看的资源信息类型: 1

        系统CPU负载信息
=================================
```

```
CPU 1 min平均负载为: 0.00
CPU 5 min平均负载为: 0.01
CPU 15 min平均负载为: 0.05
==================================
[root@rocky9 ~]# /bin/bash systemctl_load.sh
---------------查看资源操作动作---------------
 1: CPU  2: MEM
-------------------------------------------
> 请输入要查看的资源信息类型: 2

        系统内存负载信息
==================================
内存 总量 信息为: 3770 M
内存 使用 信息为: 247 M
内存 空闲 信息为: 3302 M
==================================
```

其他实践

需求

```
    按照信息提示，分别打印 三角形 和 等腰梯形
        *                    *****
       * *                  *******
      * * *                *********
     * * * *              ***********
    * * * * *            *************
```

脚本内容

```
[root@rocky9 ~]# cat drawn_graph.sh
#!/bin/bash
# 功能：打印相关图形
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

graph_type=(三角形 梯形)
# 服务的操作提示
echo -e "\e[31m---------------查看可以绘制的图形---------------
 1: 三角形  2: 梯形
-------------------------------------------"'\033[0m'
# 选择服务操作类型
read -p "> 请输入要查看的资源信息类型: " graph_id
case ${graph_type[$graph_id-1]} in
    "三角形")
        read -p "> 请输入三角形绘制的层数: " layer_num
        # 定制打印n层的三角形
        for i in $(seq 1 ${layer_num});do
            # 定制打印三角形左侧的空格效果
            for m in $(seq $[${layer_num}-$i]);do
                echo -n " "
            done
            # 定制打印三角形核心部分
            for j in $(seq $i);do
                echo -n "* "
            done
```

```
            # 打印完每行就换行
            echo
        done;;
    "梯形")
        read -p "> 请输入梯形绘制的层数: " layer_num
        # 定制打印n层的梯形
        print_num=${layer_num}
        for i in $(seq 1 ${layer_num});do
            # 定制打印梯形左侧的空格效果
            for m in $(seq $[${layer_num}-$i]);do
                echo -n " "
            done
            # 定制打印梯形核心部分
            for j in $(seq $[$print_num]);do
                echo -n "*"
            done
            let print_num+=2
            echo
        done;;
    *)
        echo -e "\e[31m\t请输入正确的绘图类型id\e[0m";;
esac
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash drawn_graph.sh
---------------查看可以绘制的图形---------------
 1: 三角形  2: 梯形
------------------------------------------
> 请输入要查看的资源信息类型: 1
> 请输入三角形绘制的层数: 5
    *
   * *
  * * *
 * * * *
* * * * *
[root@rocky9 ~]# /bin/bash drawn_graph.sh
---------------查看可以绘制的图形---------------
 1: 三角形  2: 梯形
------------------------------------------
> 请输入要查看的资源信息类型: 2
> 请输入梯形绘制的层数: 5
    *****
   *******
  *********
 ***********
*************
```

1.7.8 while read

read实践

功能简介

```
    while中有一种特殊的语法，while read line 它可以从文本中逐行读取相关的内容，然后存储到一个临时变量line中，然后我们后续就可以逐行对文本内容进行操作
```

语法解读

```
样式1： cat提前读
cat a.log | while read line
do
    echo "File: ${line}"
done
```

```
样式2： exec提前读
exec 0< a.log
while read line
do
    echo "${line}"
done
```

```
样式3：结尾导入读
while read line
do
    echo "File: ${line}"
done < a.log
```

```
注意：
    方法1和3可以直接在命令行来实验，但是方法2必须在脚本中才能实验
```

实践1-命令行实践

```
方法1实践读取文件
[root@rocky9 ~]# cat /etc/hosts | while read line;do echo "File: ${line}";done
File: 127.0.0.1   localhost localhost.localdomain localhost4
localhost4.localdomain4
File: ::1         localhost localhost.localdomain localhost6
localhost6.localdomain6

方法3实践读取文件
[root@rocky9 ~]# while read line;do echo "File: ${line}";done < /etc/hosts
File: 127.0.0.1   localhost localhost.localdomain localhost4
localhost4.localdomain4
File: ::1         localhost localhost.localdomain localhost6
localhost6.localdomain6
```

实践2-脚本实践

```
方法2实践读取文件
[root@rocky9 ~]# cat while_read_file.sh
#!/bin/bash
# 功能：while的exec读取文件内容

# 定制普通变量
read -p "> 请输入待读取的文件路径: " file_path

# 持久检测站点状态
exec < ${file_path}
while read line
```

```
do
    echo "File: ${line}"
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash while_read_file.sh
> 请输入待读取的文件路径: /etc/hosts
File: 127.0.0.1   localhost localhost.localdomain localhost4
localhost4.localdomain4
File: ::1         localhost localhost.localdomain localhost6
localhost6.localdomain6
```

1.7.9 while嵌套

基础知识

简介

```
这里的嵌套实践，与选择语句的嵌套实践基本一致，只不过组合的方式发生了一些变化。常见的组合样式如下：
```

```
while嵌套while语句
    while 循环条件
    do
        while 循环条件语句
    done
```

```
while嵌套if语句
    while 循环条件
    do
        if 条件控制语句
    done
```

简单实践

while嵌套while语句实践1-输出99乘法表

```
查看脚本内容
[root@rocky9 ~]# cat while_nine_table.sh
#!/bin/bash
# 功能：while打印99乘法表

# 定制打印99乘法表的业务逻辑
# 对第一个位置的数字进行循环
num1=1
while [ ${num1} -le 9 ]
do
    # 对第二个位置的数字进行循环
    num2=1
    while [ ${num2} -le ${num1} ]
    do
```

```
        # 信息输出,\t\c 的目的是删除后续信息，生成的内容是固定长度
        echo -e "\e[$[RANDOM%9+31]m${num1}x${num2}=$[num1*num2]\e[0m\t\c"
        num2=$[$num2+1]
    done
    echo # 一个子循环一行内容
    num1=$[$num1+1]
done
```

```
脚本文件执行效果
[root@rocky9 ~]# /bin/bash while_nine_table.sh
1x1=1
2x1=2   2x2=4
3x1=3   3x2=6   3x3=9
4x1=4   4x2=8   4x3=12  4x4=16
5x1=5   5x2=10  5x3=15  5x4=20  5x5=25
6x1=6   6x2=12  6x3=18  6x4=24  6x5=30  6x6=36
7x1=7   7x2=14  7x3=21  7x4=28  7x5=35  7x6=42  7x7=49
8x1=8   8x2=16  8x3=24  8x4=32  8x5=40  8x6=48  8x7=56  8x8=64
9x1=9   9x2=18  9x3=27  9x4=36  9x5=45  9x6=54  9x7=63  9x8=72  9x9=81
```

while嵌套if语句实践2-手机发送短信1次/1毛，余额低于1毛提示无法发送请充值

```
[root@rocky9 ~]# cat while_mobile_bill.sh
#!/bin/bash
# 功能：while提示收集发短信

# 定制普通变量
read -p "> 请输入收集话费余额(元): " mobile_bill

# 定制普通变量
sms_num=0
bull_count=$[$mobile_bill * 10]
while [ $bull_count -ge 0 ]
do
    sms_num=$(($sms_num+1))
    if [ $bull_count -lt 1 ];then
        echo "剩余费用不足，请充话费!"
    else
        echo "截至目前，您已发送 ${sms_num} 条短信。"
    fi
    bull_count=$(($bull_count-1))
    sleep 0.1
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash while_mobile_bill.sh
> 请输入收集话费余额(元): 1
截至目前，您已发送 1 条短信。
截至目前，您已发送 2 条短信。
截至目前，您已发送 3 条短信。
截至目前，您已发送 4 条短信。
截至目前，您已发送 5 条短信。
截至目前，您已发送 6 条短信。
截至目前，您已发送 7 条短信。
截至目前，您已发送 8 条短信。
截至目前，您已发送 9 条短信。
```

```
截至目前，您已发送 10 条短信。
剩余费用不足，请充话费！
```

1.7.10 until嵌套

基础知识

简介

```
这里的嵌套实践，与while语句的嵌套实践基本一致，只不过组合的方式发生了一些变化。常见的组合样式如下：
```

```
until嵌套until语句
    until 循环条件
    do
        until 循环条件语句
    done
```

```
until嵌套if语句
    until 循环条件
    do
        if 条件控制语句
    done
```

简单实践

until嵌套until语句实践1-输出99乘法表

```
查看脚本内容
[root@rocky9 ~]# cat until_nine_table.sh
#!/bin/bash
# 功能：until打印倒序99乘法表

# 定制打印99乘法表的业务逻辑
# 对第一个位置的数字进行循环
num1=9
until [ ${num1} -eq 0 ]
do
    # 对第二个位置的数字进行循环
    num2=1
    until [ ${num2} -gt ${num1} ]
    do
        # 信息输出,\t\c 的目的是删除后续信息，生成的内容是固定长度
        echo -e "\e[$[RANDOM%9+31]m${num1}x${num2}=$[num1*num2]\e[0m\t\c"
        num2=$[$num2+1]
    done
    num1=$[$num1-1]
    echo # 一个子循环一行内容
done
```

脚本文件执行效果

```
[root@rocky9 ~]# /bin/bash until_nine_table.sh
9x1=9   9x2=18  9x3=27  9x4=36  9x5=45  9x6=54  9x7=63  9x8=72  9x9=81
8x1=8   8x2=16  8x3=24  8x4=32  8x5=40  8x6=48  8x7=56  8x8=64
7x1=7   7x2=14  7x3=21  7x4=28  7x5=35  7x6=42  7x7=49
6x1=6   6x2=12  6x3=18  6x4=24  6x5=30  6x6=36
5x1=5   5x2=10  5x3=15  5x4=20  5x5=25
4x1=4   4x2=8   4x3=12  4x4=16
3x1=3   3x2=6   3x3=9
2x1=2   2x2=4
1x1=1
```

until嵌套if语句实践2-手机发送短信1次/1毛，余额低于1毛提示无法发送请充值

```
[root@rocky9 ~]# cat until_mobile_bill.sh
#!/bin/bash
# 功能：until提示收集发短信

# 定制普通变量
read -p "> 请输入收集话费余额(元): " mobile_bill

# 定制普通变量
sms_num=0
bull_count=$[$mobile_bill * 10]
until [ $bull_count -lt 0 ]
do
    sms_num=$(($sms_num+1))
    if [ $bull_count -lt 1 ];then
        echo "剩余费用不足，请充话费!"
    else
        echo "截至目前，您已发送 ${sms_num} 条短信。"
    fi
    bull_count=$(($bull_count-1))
    sleep 0.1
done
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash until_mobile_bill.sh
> 请输入收集话费余额(元): 1
截至目前，您已发送 1 条短信。
截至目前，您已发送 2 条短信。
截至目前，您已发送 3 条短信。
截至目前，您已发送 4 条短信。
截至目前，您已发送 5 条短信。
截至目前，您已发送 6 条短信。
截至目前，您已发送 7 条短信。
截至目前，您已发送 8 条短信。
截至目前，您已发送 9 条短信。
截至目前，您已发送 10 条短信。
剩余费用不足，请充话费!
```

1.7.11 函数+数组

基础知识

简介

```
    我们知道在shell脚本中，数组可以帮助我们做很多灵活的事情，尤其是数据的归类存储。而数组在函数的使用过程中，还是受到了一定程度的限制。这些限制主要体现在以下几个方面；
```

```
限制1：
    以变量的方式传递数组给函数，函数内部无法正常使用
限制2：
    我们只能先解析所有数组元素，然后再传递给函数，接着在函数体内部重新组合
限制3：
    函数内部以echo方式输出数组所有元素，然后再函数外部重新组合为数组
```

简单实践

实践1-函数无法正常接收数据元素

```
查看脚本内容
[root@rocky9 ~]# cat function_array_input.sh
#!/bin/bash
# 功能: 函数接收数组元素

# 定制功能函数
func_array(){
    echo "函数内接收的参数: $@"
}

# 定制数组变量
myarray=(aa bb cc dd ee)
echo "myarray数组的所有元素有: ${myarray[@]}"

# 数组传递给函数
func_array $myarray
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_array_input.sh
myarray数组的所有元素有: aa bb cc dd ee
函数内接收的参数: aa

结果显示：
    虽然我们传递数组给函数，但是函数无法正常接收数组
```

实践2-函数接收数组

```
查看脚本内容
[root@rocky9 ~]# cat function_array_input2.sh
#!/bin/bash
# 功能: 函数接收数组元素

# 定制功能函数
func_array(){
```

```
    echo "函数内接收的参数: $@"
    func_arr=($(echo $@))
    echo "函数内func_arr的数组元素有: ${func_arr[@]}"
}

# 定制数组变量
myarray=(aa bb cc dd ee)
echo "myarray数组的所有元素有: ${myarray[@]}"

# 数组解析后，将所有元素传递给函数
func_array ${myarray[@]}
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_array_input2.sh
myarray数组的所有元素有: aa bb cc dd ee
函数内接收的参数: aa bb cc dd ee
函数内func_arr的数组元素有: aa bb cc dd ee
```

实践3-脚本接收函数内数组

```
查看脚本内容
[root@rocky9 ~]# cat function_array_output.sh
#!/bin/bash
# 功能: 脚本接收函数内数组元素

# 定制功能函数
func_array(){
    # 函数体内构造新数组
    func_arr=($(echo $@))
    # 生成新数组
    for (( i=0; i<${#func_arr[@]}; i++ ))
    do
        newarray[$i]=$[ ${func_arr[$i]} * 3 ]
    done
    # 逐个返回数组元素
    echo ${newarray[@]}
}

# 定制数组变量
myarray=(1 2 3 4 5)
echo "myarray数组的所有元素有: ${myarray[@]}"

# 接收函数体返回的数组内容
result=($(func_array ${myarray[@]}))
echo "函数返回的result数组元素: ${result[@]}"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_array_output.sh
myarray数组的所有元素有: 1 2 3 4 5
函数返回的result数组元素: 3 6 9 12 15
```

1.7.12 函数嵌套

基础知识

简介

```
    所谓的函数嵌套，主要是在函数间或者文件间相互使用的一种方式。它主要有三种样式：
```

```
样式1：函数间调用
    - 函数体内部调用其他的函数名
样式2：文件间调用
    - 函数体内部调用另外一个文件的函数名
    - 需要额外做一步文件source的加载动作
    注意：我们将专门提供函数的文件称为 -- 函数库
样式3：函数自调用
    - 函数体内部调用自己的函数名，将复杂的逻辑简单化
```

简单实践

函数间调用实践1-图形信息打印

```
    按照信息提示，分别打印 三角形 和 等腰梯形
        *                    *****
       * *                  *******
      * * *                *********
     * * * *              ***********
    * * * * *            *************
```

```
[root@rocky9 ~]# cat function_drawn_graph.sh
#!/bin/bash
# 功能：打印相关图形
# 版本：v0.2
# 作者：王树森
# 联系：sswang.magedu.com

graph_type=(三角形 梯形)
# 定制服务的操作提示功能函数
menu(){
    echo -e "\e[31m---------------查看可以绘制的图形---------------"
    echo -e " 1: 三角形  2: 梯形"
    echo -e "-----------------------------------------\033[0m"
}

# 定制打印左侧空格效果
left_bland_func(){
   layer_num="$1"
   sub_num="$2"
   for m in $(seq $[${layer_num}-${sub_num}]);do
        echo -n " "
   done
}

# 打印图形的核心内容部分
kernel_character_func(){
   char_num="$1"
```

```
    char_mark="$2"
    for j in $(seq ${char_num});do
        echo -n "${char_mark}"
    done
}
# 定制打印三角形的函数
triangle_func(){
    # 接收函数传参
    layer_num=$1
    # 定制打印n层的三角形
    for i in $(seq 1 ${layer_num});do
        # 定制打印三角形左侧的空格效果
        left_bland_func ${layer_num} $i
        # 定制打印三角形核心部分
        kernel_character_func $i "* "
        # 打印完每行就换行
        echo
    done
}
# 定制梯形的功能函数
trapezium_func(){
    print_num=${layer_num}
    for i in $(seq 1 ${layer_num});do
        # 定制打印梯形左侧的空格效果
        left_bland_func ${layer_num} $i
        # 定制打印梯形核心部分
        kernel_character_func $print_num "*"
        let print_num+=2
        echo
    done
}
# 选择服务操作类型
while true;do
    menu
    read -p "> 请输入要查看的资源信息类型: " graph_id
    case ${graph_type[$graph_id-1]} in
        "三角形")
            read -p "> 请输入三角形绘制的层数: " layer_num
            triangle_func ${layer_num}
            ;;
        "梯形")
            read -p "> 请输入梯形绘制的层数: " layer_num
            # 定制打印n层的梯形
            trapezium_func ${layer_num}
            ;;
        *)
            echo -e "\e[31m\t请输入正确的绘图类型id\e[0m";;
    esac
done
```

脚本执行后效果

```
[root@rocky9 ~]# /bin/bash function_drawn_graph.sh
---------------查看可以绘制的图形---------------
 1: 三角形  2: 梯形
-----------------------------------------
> 请输入要查看的资源信息类型: 1
> 请输入三角形绘制的层数: 5
```

```
    *
   * *
  * * *
 * * * *
* * * * *
---------------查看可以绘制的图形---------------
 1: 三角形   2: 梯形
-----------------------------------------
> 请输入要查看的资源信息类型: 2
> 请输入梯形绘制的层数: 5
    *****
   *******
  *********
 ***********
*************
---------------查看可以绘制的图形---------------
 1: 三角形   2: 梯形
-----------------------------------------
> 请输入要查看的资源信息类型: ^C
[root@rocky9 ~]#
```

文件间调用实践2-拆分function_drawn_graph.sh脚本

```
需求：拆分绘图脚本文件
    1 将脚本文件中的功能逻辑函数拆分出来以单独的文件存在
    2 脚本文件保留核心逻辑功能
```

```
创建功能函数库文件目录
[root@rocky9 ~]# mkdir lib

查看库文件内容
[root@rocky9 ~]# cat lib/drawn_func.sh
#!/bin/bash
# 功能：打印相关图形功能函数库
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制服务的操作提示功能函数
menu(){
    echo -e "\e[31m---------------查看可以绘制的图形---------------"
    echo -e " 1: 三角形   2: 梯形"
    echo -e "-----------------------------------------\033[0m"
}

# 定制打印三角形左侧空格效果
left_bland_func(){
   layer_num="$1"
   sub_num="$2"
   for m in $(seq $[${layer_num}-${sub_num}]);do
        echo -n " "
   done
}

# 打印图形的核心内容部分
kernel_character_func(){
```

```
    char_num="$1"
    char_mark="$2"
    for j in $(seq ${char_num});do
        echo -n "${char_mark}"
    done
}
# 定制打印三角形的函数
triangle_func(){
    # 接收函数传参
    layer_num=$1
    # 定制打印n层的三角形
    for i in $(seq 1 ${layer_num});do
        # 定制打印三角形左侧的空格效果
        left_bland_func ${layer_num} $i
        # 定制打印三角形核心部分
        kernel_character_func $i "* "
        # 打印完每行就换行
        echo
    done
}
# 定制梯形的功能函数
trapezium_func(){
    print_num=${layer_num}
    for i in $(seq 1 ${layer_num});do
        # 定制打印梯形左侧的空格效果
        left_bland_func ${layer_num} $i
        # 定制打印梯形核心部分
        kernel_character_func $print_num "*"
        let print_num+=2
        echo
    done
}
```

查看脚本框架文件

```
[root@rocky9 ~]# cat function_drawn_graph-lib.sh
#!/bin/bash
# 功能：打印相关图形
# 版本：v0.3
# 作者：王树森
# 联系：sswang.magedu.com

# 定制数组变量
graph_type=(三角形 梯形)

# 加载功能函数库文件
source ./lib/drawn_func.sh

# 选择服务操作类型
while true;do
    menu
    read -p "> 请输入要查看的资源信息类型： " graph_id
    case ${graph_type[$graph_id-1]} in
        "三角形")
            read -p "> 请输入三角形绘制的层数： " layer_num
            triangle_func ${layer_num}
            ;;
        "梯形")
```

```
            read -p "> 请输入梯形绘制的层数: " layer_num
            # 定制打印n层的梯形
            trapezium_func ${layer_num}
            ;;
        *)
            echo -e "\e[31m\t请输入正确的绘图类型id\e[0m";;
    esac
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_drawn_graph-lib.sh
---------------查看可以绘制的图形---------------
 1: 三角形  2: 梯形
-------------------------------------------
> 请输入要查看的资源信息类型: 1
> 请输入三角形绘制的层数: 5
    *
   * *
  * * *
 * * * *
* * * * *
---------------查看可以绘制的图形---------------
 1: 三角形  2: 梯形
-------------------------------------------
> 请输入要查看的资源信息类型: ^C
[root@rocky9 ~]#
```

1.7.13 函数自调用

简单实践

简介

```
    函数自调用也称函数递归，说白了就是 函数调用自身，实现数据递归能力的实现
```

实践-函数自调用

```
需求: 实现数学阶乘的实践
    示例：5的阶乘
        完整格式：5! = 1 * 2 * 3 * 4 * 5 = 120
        简写格式：5! = 5 * (1 * 2 * 3 * 4) = 5 * 4!
    公式: x! = x * (x-1)!
```

```
查看脚本内容
[root@rocky9 ~]# cat function_func_test1.sh
#!/bin/bash
# 功能：函数自调用实践

# 定制功能函数框架
self_func(){
    # 接收一个参数
    num=$1
    if [ ${num} -eq 1 ];then
```

```
        echo 1
    else
        # 定制一个临时本地变量，获取递减后的值
        local temp=$[ ${num} - 1 ]
        # 使用函数自调用方式获取内容
        local result=$(self_func $temp)
        # 格式化输出信息
        echo $[ $result * ${num} ]
    fi
}

# 检测逻辑效果
while true
do
    read -p "请输入一个您要查询的阶乘: " value
    result=$(self_func ${value})
    echo "${value}的阶乘是: ${result}"
done
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash function_func_test1.sh
请输入一个您要查询的阶乘: 5
5的阶乘是: 120
请输入一个您要查询的阶乘: 6
6的阶乘是: 720
请输入一个您要查询的阶乘: 7
7的阶乘是: 5040
请输入一个您要查询的阶乘: ^C
[root@rocky9 ~]#
```

案例实践

实践1-遍历制定目录下的所有文件

```
准备工作
[root@rocky9 ~]# mkdir -p dir/{softs/{nginx,tomcat},logs,server/{java,python}}
[root@rocky9 ~]# touch dir/softs/{nginx/nginx.conf,tomcat/server.xml}
[root@rocky9 ~]# touch dir/logs/user{1..3}.log
[root@rocky9 ~]# touch dir/server/{java/java.jar,python/python.py}
[root@rocky9 ~]# tree dir/
dir/
├── logs
│   ├── user1.log
│   ├── user2.log
│   └── user3.log
├── server
│   ├── java
│   │   └── java.jar
│   └── python
│       └── python.py
└── softs
    ├── nginx
    │   └── nginx.conf
    └── tomcat
        └── server.xml

7 directories, 7 files
```

```
[root@rocky9 ~]# cat function_scan_dir.sh
#!/bin/bash
# 功能：扫描目录下所有文件
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 定制功能函数框架
# 定制目录扫描功能函数
scan_dir() {
    # 定制临时局部功能变量
    # cur_dir 当前目录 workdir 工作目录
    local cur_dir workdir

    # 接收要检查的目录，进入到目录中
    workdir=$1
    cd ${workdir}

    # 对工作目录进行简单判断，根目录没有父目录
    if [ ${workdir} = "/" ]
    then
        cur_dir=""
    else
        cur_dir=$(pwd)
    fi

    # 查看当前目录下的文件列表
    for item in $(ls ${cur_dir})
    do
        # 如果文件是目录，则继续查看目录下文件
        if test -d ${item};then
            cd ${item}
            scandir ${cur_dir}/${item}
            cd ..
        # 如果文件是普通文件，则输出信息即可
        else
            echo ${cur_dir}/${item}
        fi
    done
}

# 检测逻辑效果
while true
do
    read -p "请输入一个您要查询的目录：" value
    if [ -d ${value} ]
    then
        scandir ${value}
    else
        echo "您输入的不是目录，请重新输入!"
    fi
done
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash function_scan_dir.sh
```

```
请输入一个您要查询的目录：dir
/root/dir/logs/user1.log
/root/dir/logs/user2.log
/root/dir/logs/user3.log
/root/dir/server/java/java.jar
/root/dir/server/python/python.py
/root/dir/softs/nginx/nginx.conf
/root/dir/softs/tomcat/server.xml
请输入一个您要查询的目录：^C
[root@rocky9 ~]#
结果显示：
    该脚本达到了我们需要的目录遍历效果
```

1.7.14 综合练习

学习目标

这一节，我们从 案例解读、脚本实践、小结 三个方面来学习。

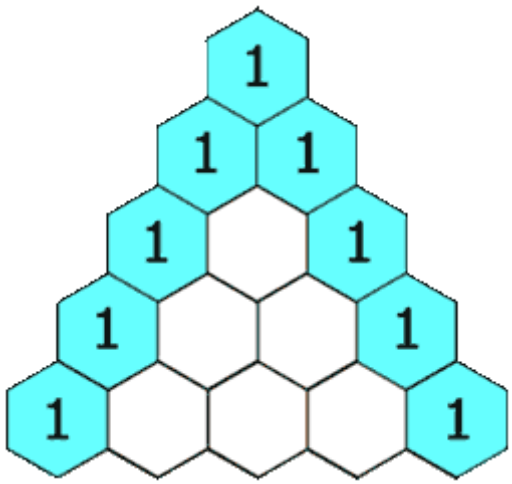
案例解读

案例需求

```
使用shell脚本绘制一个杨辉三角
```

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5   10  10  5   1
        1   6   15  20  15  6   1
      1   7   21  35  35  21  7   1
    1   8   28  56  70  56  28  8   1
  1   9   36  84  126 126 84  36  9   1
1   10  45  120 210 252 210 120 45  10  1
```

案例解读



```
1、每行数字左右对称，从1开始变大，然后变小为1。
2、第n行的数字个数为n个，所有数字和为 2^(n−1)。
3、每个数字等于上一行的左右临近两个数字之和。
4、第n行的数字依次为 1、1×(n-1)、1×(n-1)×(n-2)/2、1×(n-1)×(n-2)/2×(n-3)/3 ...
...
```

脚本实践

脚本实践

```
查看脚本内容
[root@rocky9 ~]# cat yanghui_triangle.sh
#!/bin/bash
# 功能：shell定制杨辉三角功能
# 版本：v0.1
# 作者：王树森
# 联系：sswang.magedu.com

# 判断输入是否为整数
check_int(){
    # 设定数据标识
    flag=true
    read -p "请输入一个数据值: " layer_num
    # 通过在循环内部进行数据操作判断是否是数据
    while $flag;do
      expr $layer_num + 0 > /dev/null 2>&1
      if [ $? -eq 0 ]
      then
          flag=false
      else
          read -p "请输入一个数据值: " layer_num
      fi
    done
}

# 定制一个数组
declare -a num_array

check_int
# 定制杨辉三角的行数变量 row
for(( row=1; row<=layer_num; row++ ))
do
   #打印杨辉三角的左侧空白
   for k in $(seq $[$layer_num - $row])
   do
       echo -n "    "
   done
   # 定制每行的数据获取
   for(( col=1; col<=row; col++ ))
   do
       # 第n行的第1个和第n行的第n个数字为1
       if [ $col -eq 1  -o $row -eq $col ]
       then
         # 设定每行的两个边界数字为1
         num_array[$row$col]=1
       else
         # 获取上一行的两个临近数据
```

```
            let row_up=row-1  # 获取上一行的数据
            let col_up=col-1  # 获取上一行的临近数据
            # 获取当前行的数据值为 上一行临近数据的数据和
            let
num_array[$row$col]=${num_array[$row_up$col_up]}+${num_array[${row_up}${col}]}
        fi
    done
    # 打印每行的数据
    for(( col=1; col<=row; col++ ))
    do
        printf "%-8s" ${num_array[$row$col]}
    done
    echo
done
```

```
脚本执行效果
[root@rocky9 ]# /bin/bash yanghui_triangle.sh
请输入一个数据值: 8
                        1
                    1       1
                1       2       1
            1       3       3       1
        1       4       6       4       1
    1       5       10      10      5       1
1       6       15      20      15      6       1
1       7       21      35      35      21      7       1
```

函数嵌套改造

```
脚本改造后内容
[root@rocky9 ~]# cat yanghui_triangle.sh
#!/bin/bash
# 功能：shell定制杨辉三角功能
# 版本：v0.2
# 作者：王树森
# 联系：sswang.magedu.com

# 定制一个数组
declare -a num_array

# 判断输入是否为整数
check_int(){
    # 设定数据标识
    flag=true
    read -p "请输入一个数据值(q退出): " layer_num
    [ $layer_num == "q" ] && exit
    # 通过在循环内部进行数据操作判断是否是数据
    while $flag;do
      expr $layer_num + 0 > /dev/null 2>&1
      if [ $? -eq 0 ]
      then
          flag=false
      else
          read -p "请输入一个数据值: " layer_num
      fi
    done
```

```
}

# 定制左侧空格打印逻辑
left_blank_func(){
   # 获取参数值
   layer_num=$1
   row=$2
   # 空格打印逻辑
   for k in $(seq $[$layer_num - $row])
   do
       echo -n "    "
   done
}

# 获取每行的数据值
col_num_count(){
   # 获取参数值
   row=$1
   # 数据获取逻辑
   for(( col=1; col<=row; col++ ))
   do
       # 第n行的第1个和第n行的第n个数字为1
       if [ $col -eq 1  -o $row -eq $col ]
       then
         # 设定每行的两个边界数字为1
         num_array[$row$col]=1
       else
         # 获取上一行的两个临近数据
         let row_up=row-1  # 获取上一行的数据
         let col_up=col-1  # 获取上一行的临近数据
         # 获取当前行的数据值为 上一行临近数据的数据和
         let
num_array[$row$col]=${num_array[$row_up$col_up]}+${num_array[${row_up}${col}]}
       fi
   done
}
# 每行数据打印逻辑
col_num_print(){
   # 获取参数值
   row=$1
   # 数据打印逻辑
   for(( col=1; col<=row; col++ ))
   do
       printf "%-8s" ${num_array[$row$col]}
   done
   echo
}
while true
do
  check_int
  # 定制杨辉三角的行数变量 raw
  for(( row=1; row<=layer_num; row++ ))
  do
     #打印杨辉三角的左侧空白
     left_blank_func $layer_num $row
     # 获取数据的值
     col_num_count $row
     # 打印每行的所有数据
```

```
    col_num_print $row
  done
done
```

```
[root@rocky9 ~]# /bin/bash yanghui_triangle.sh
请输入一个数据值(q退出): 5
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
请输入一个数据值(q退出): q
[root@rocky9 ~]#
```

1.7.15 shell大项目

项目简介

简介

```
使用shell编程语言，来完成一个大平台的功能管理。
```

参考资料

```
https://gitee.com/wshs1117/shell_k8s.git
```