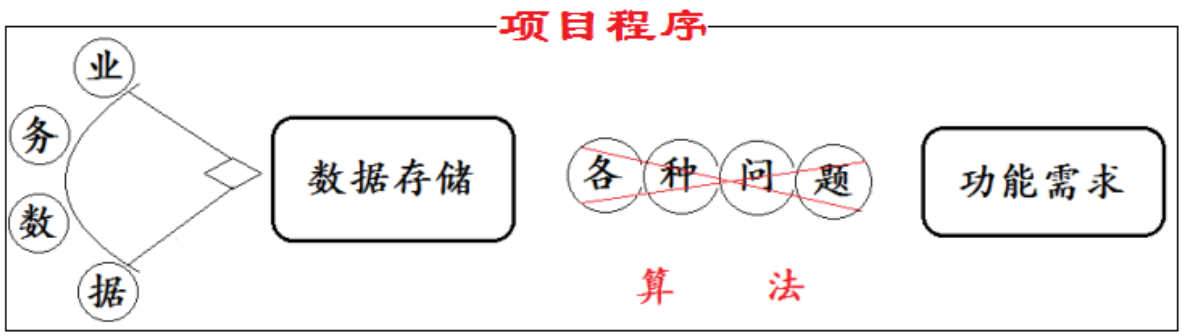
1 Shell基础

1.1 编程简介

1.1.1 编程语言解读

基础知识

程序



外在关系：

业务数据：用户访问业务时候，产生的信息内容

数据结构：静态的描述了数据元素之间的关系

算法：解决各种实际问题的方法和思路

数据结构 + 算法 = 程序

内在关系：

算法其实就是数据的表示和处理，而数据的处理受到数据的各种存储形式的约束，所以算法的效率和样式受到数据结构的严重约束。

数据结构存储下来的数据为为算法服务的，而算法存在的意义就是为了数据结构中的内容而存在的。

所以说：数据结构和算法，是你中有我，我中有你的合二为一的关系

理解：

我们一般说的数据不是干巴巴的字母数字，而是在某种场景下来对这些数据的含义进行分析等操作，数据一旦有了场景意义：

"一" 在不同场景的声音和含义。

纯粹的数据加上场景，他们就有了新的名称：ADT

ADT

举例一：

数据类型-人　　多个人，陈浩南、山鸡、大天二、大飞...

数据运算-关系　　彼此间的团队联系

抽象数据类型=类型+运算=人+关系　　洪兴

举例二：

游戏按钮“空格”：人物A(数据类型) + 打子弹(动作)

编程语言

编程

所谓的编程，就是将我们的功能思路用代码的方式实现出来。一般情况下，这些实现的思路主要有两种方式：

面向过程编程

将任务功能拆分成多个子部分，然后按照顺序依次执行下去。

- 问题规模小，可以步骤化，按部就班处理

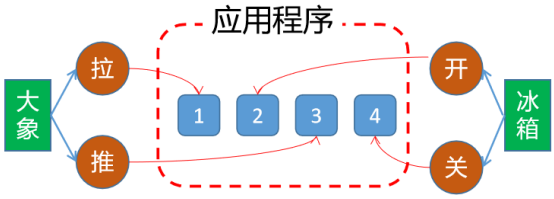
比较符合我们自己做一件事情的思路。

面向对象编程

将任务功能拆分成多个子部分，然后按照顺序依次找不同的人执行下去。

- 问题规模大，复杂系统

比较符合我们的领导指挥一个部门的人做事情的思路。



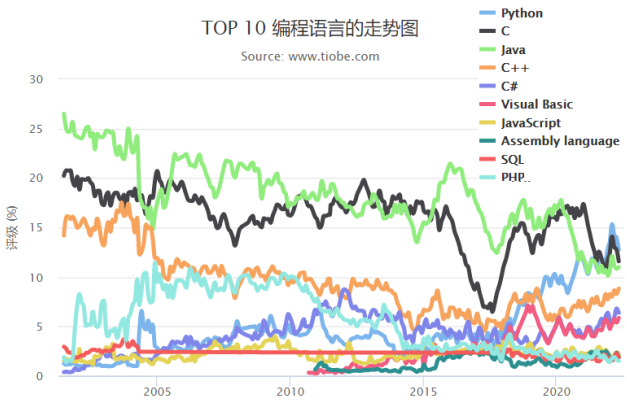
编程语言

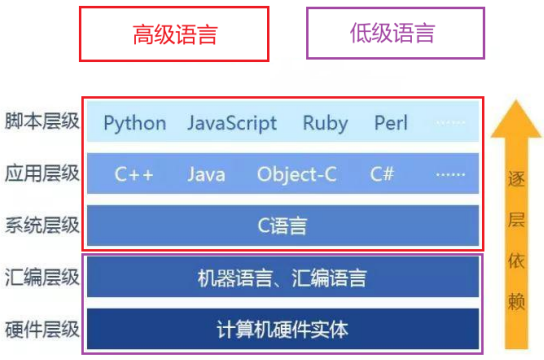
所谓编程语言，指的是以哪种风格代码方式把程序编写出来。每一种编程风格就是一系列的编程语言。

参考资料：

https://www.tiobe.com/tiobe-index/

https://hellogithub.com/report/tiobe/



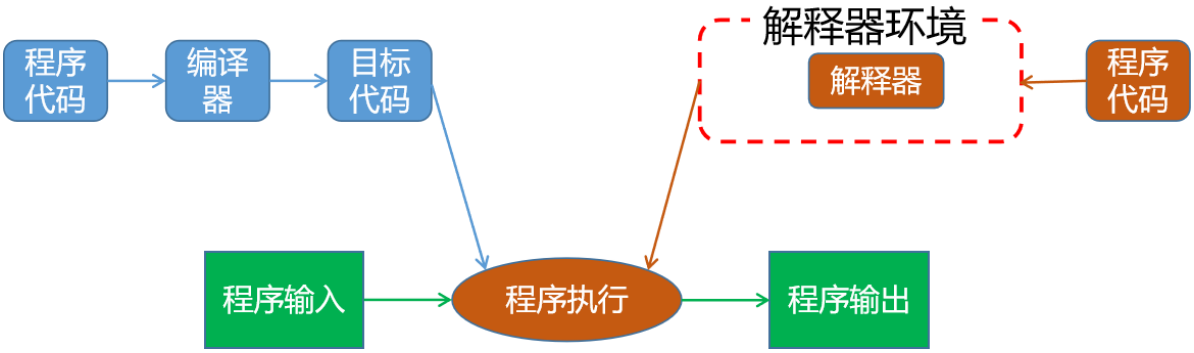


1.1.2 编程语言逻辑

语言分类

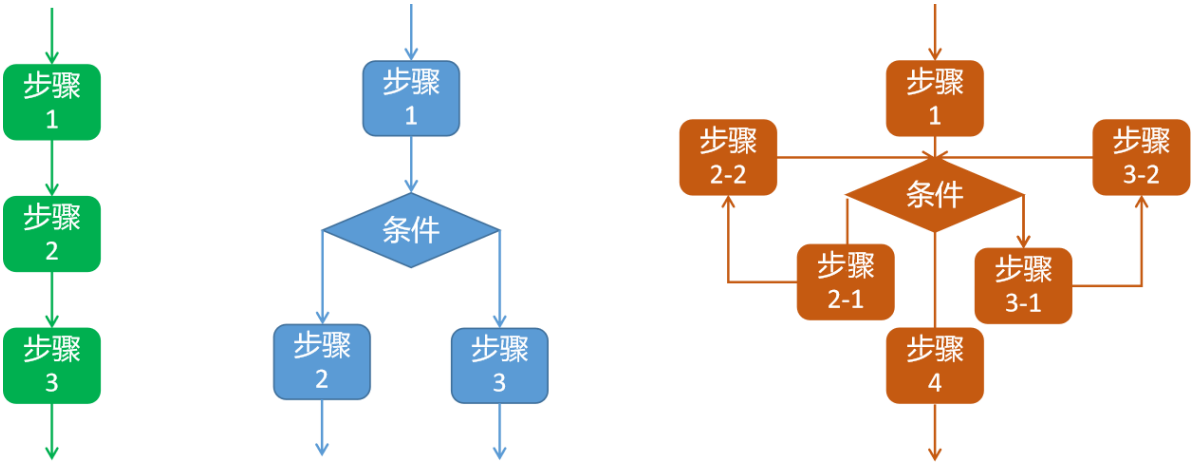
语言分类

低级编程语言：
    机器：
        - 二进制的0和1的序列，称为机器指令。
        - 一般人看不懂
    汇编：
        - 用一些助记符号替代机器指令，称为汇编语言。
        - 一般人看不懂，但是能够有感觉
高级编程语言：
    编译：
        - 借助于专属编译器将一些高级语言编译成机器代码文件，然后再交给程序去执行。
        - 如：C，C++等
    解释：
        - 将高级语言的代码逐行加载到解释器环境，逐行解释为机器代码，然后再交给程序去执行。
        - 如：shell，python，php，JavaScript等



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



1.2 shell基础

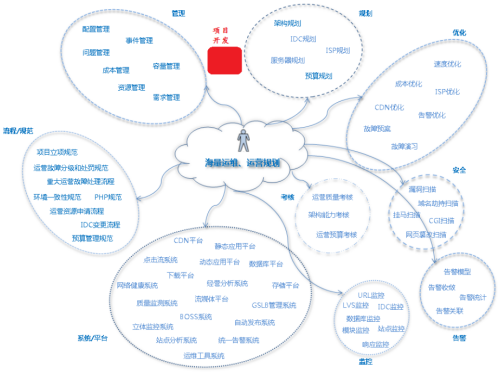
1.2.1 shell简介

运维

简介

运维是什么？

所谓的运维，其实就是公司的内部项目当中的一个技术岗位而已，它主要做的是项目的维护性工作。它所涉及的内容范围非常多。



以xx项目为例：

规划：我们需要多少资源来支持项目的运行

管理：项目运行过程中的所有内容都管理起来

流程规范：所有操作都形成制度，提高工作效率

平台：大幅度提高工作效率

监控：实时查看项目运行状态指标

告警：状态指标异常，告知工作人员处理

安全：网站运营安全措施

优化：保证用户访问网站体验很好

考核：权责分配，保证利益

自动化运维：就是将图里面所有的工作都使用自动化的方式来实现。

实现自动化的方式很多，常见的方式：工具和脚本。

工作中常见的脚本有哪些呢？

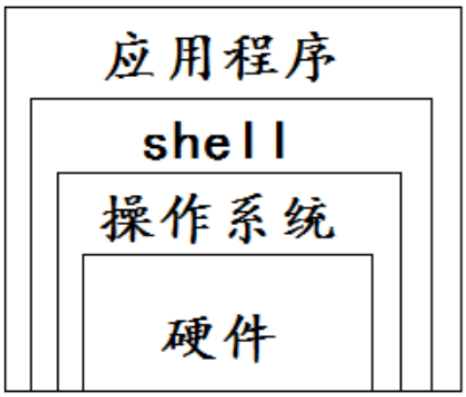
shell脚本 和 其他开发语言脚本

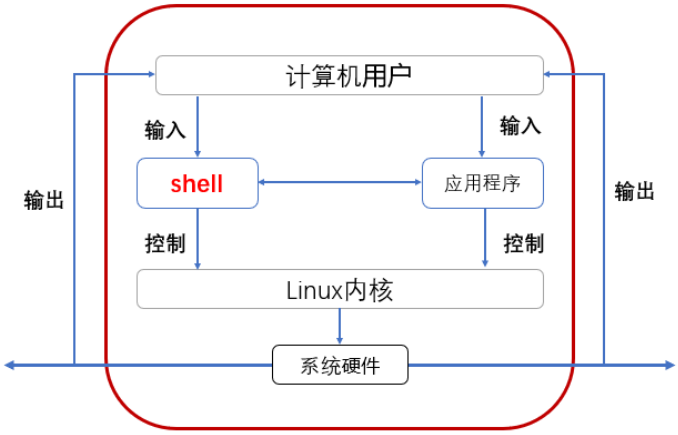
shell语言

shell定义

在计算机科学中，Shell就是一个命令解释器。

shell是位于操作系统和应用程序之间，是他们二者最主要的接口，shell负责把应用程序的输入命令信息解释给操作系统，将操作系统指令处理后的结果解释给应用程序。





一句话，shell就是在操作系统和应用程序之间的一个命令翻译工具。

1.2.2 shell脚本实践

脚本基础

shell脚本

当可执行的Linux命令或语句不在命令行状态下执行，而是通过一个文件执行时，我们称文件为shell脚本。

应用场景

重复化、复杂化的工作，通过把工作的命令写成脚本，以后仅仅需要执行脚本就能完成这些工作。
    自动化分析处理
    自动化备份
    自动化批量部署安装
    等等...

脚本创建

脚本创建工具：
    创建脚本的常见编辑器是 vi/vim.

脚本命名
    shell脚本的命名简单来说就是要有意义，方便我们通过脚本名，来知道这个文件是干什么用的。

脚本内容：
    各种可以执行的命令

脚本注释

单行注释：
    除了首行的#不是注释外，其他所有行内容，只要首个字符是#,那么就表示该行是注释
多行注释：
    多行注释有两种方法：:<<! ... ! 和 :<<字符 ... 字符

脚本实践

脚本示例1

脚本内容

```
[root@rocky9 ~]# cat get_netinfo.sh
#!/bin/bash
# 功能：获取当前主机的网卡设备信息
# 作者：wangshuji
# 版本：V0.1
# 联系：www.sswang.com

# 获取ip地址信息
ifconfig eth0 | grep -w inet | awk '{print $2}' | xargs echo "IP: "

# 获取掩码地址信息
ifconfig eth0 | grep -w inet | awk '{print $4}' | xargs echo "NetMask: "

# 获取广播地址信息
ifconfig eth0 | grep -w inet | awk '{print $6}' | xargs echo "Broadcast: "

# 获取MAC地址信息
ifconfig eth0 | grep ether | awk '{print $2}' |xargs echo "MAC Address: "
```

```
执行脚本
[root@rocky9 ~]# /bin/bash get_netinfo.sh
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
```

脚本注释

```
更改脚本内容
[root@rocky9 ~]# cat get_netinfo.sh
#!/bin/bash
:<<!
功能：获取当前主机的网卡设备信息
作者：wangshuji
版本：V0.1
联系：www.sswang.com
!
... ...
```

1.2.3 脚本执行

脚本执行

脚本执行方法

```
方法1：
    bash /path/to/script-name    或    /bin/bash /path/to/script-name   （强烈推荐使用）

方法2：
    /path/to/script-name         或    ./script-name    （当前路径下执行脚本）

方法3：
    source script-name           或    . script-name    （注意“.“点号）

方法1变种：
    cat /path/to/script-name | bash
    bash /path/to/script-name
```

```
脚本执行说明
1、脚本文件本身没有可执行权限或者脚本首行没有命令解释器时使用的方法，我们推荐用bash执行。
    使用频率：☆☆☆☆☆
2、脚本文件具有可执行权限时使用。
    使用频率：☆☆☆☆
3、使用source或者.点号，加载shell脚本文件内容，使shell脚本内容环境和当前用户环境一致。
    使用频率：☆☆☆
    使用场景：环境一致性
```

执行示例

```
方法1：
[root@rocky9 ~]# /bin/bash get_netinfo.sh
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
```

```
方法2：
[root@rocky9 ~]# ./get_netinfo.sh
bash: ./get_netinfo.sh: 权限不够
[root@rocky9 ~]# ll get_netinfo.sh
-rw-r--r-- 1 root root 521 6月   7 20:41 get_netinfo.sh
[root@rocky9 ~]# chmod +x get_netinfo.sh
[root@rocky9 ~]# ./get_netinfo.sh
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
```

```
方法3：
[root@rocky9 ~]# source get_netinfo.sh
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
[root@rocky9 ~]# chmod -x get_netinfo.sh
[root@rocky9 ~]# ll get_netinfo.sh
-rw-r--r-- 1 root root 521 6月   7 20:41 get_netinfo.sh
[root@rocky9 ~]# source get_netinfo.sh
```

```
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
```

```
方法1变种：
[root@rocky9 ~]# cat get_netinfo.sh | bash
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
[root@rocky9 ~]# bash < get_netinfo.sh
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
MAC Address:  00:0c:29:23:23:8c
```

1.2.4 脚本调试

脚本调试

需求

```
    我们在编写脚本的时候，往往会受到各种因素的限制，导致脚本功能非常大或者内容有误，如果直接执行脚本的时候，因为脚本内容有误，导致脚本执行失败。
    所以我们需要在脚本执行的时候，保证脚本没问题，我们可以借助于多种脚本调试方式来验证脚本。
```

调试方式

```
-n      检查脚本中的语法错误
-v      先显示脚本所有内容，然后执行脚本，结果输出，如果执行遇到错误，将错误输出。
-x      将执行的每一条命令和执行结果都打印出来
```

简单实践

准备工作

```
准备备份文件
cp get_netinfo.sh get_netinfo-error.sh

设置错误文件
[root@rocky9 ~]# cat get_netinfo-error.sh
...
# 将最后一行末尾的"取消
ifconfig eth0 | grep ether | awk '{print $2}' |xargs echo "MAC Address: "
```

```
错误脚本执行效果
[root@rocky9 ~]# /bin/bash get_netinfo-error.sh
IP:  10.0.0.12
NetMask:  255.255.255.0
Broadcast:  10.0.0.255
get_netinfo-error.sh:行19: 寻找匹配的 `"' 是遇到了未预期的文件结束符
get_netinfo-error.sh:行20: 语法错误: 未预期的文件结尾
```

检查语法实践

检查语法实践

```
[root@rocky9 ~]# /bin/bash -n get_netinfo-error.sh
get_netinfo-error.sh:行19: 寻找匹配的 `"' 是遇到了未预期的文件结束符
get_netinfo-error.sh:行20: 语法错误: 未预期的文件结尾
```

检查语法调试

```
[root@rocky9 ~]# /bin/bash -v get_netinfo-error.sh
#!/bin/bash
:<<!
功能：获取当前主机的网卡设备信息
作者：wangshuji
版本：v0.1
联系：www.sswang.com
!

# 获取ip地址信息
ifconfig eth0 | grep -w inet | awk '{print $2}' | xargs echo "IP: "
IP:  10.0.0.12

# 获取掩码地址信息
ifconfig eth0 | grep -w inet | awk '{print $4}' | xargs echo "NetMask: "
NetMask:  255.255.255.0

# 获取广播地址信息
ifconfig eth0 | grep -w inet | awk '{print $6}' | xargs echo "Broadcast: "
Broadcast:  10.0.0.255

# 获取MAC地址信息
ifconfig eth0 | grep ether | awk '{print $2}' |xargs echo "MAC Address:
get_netinfo-error.sh:行19: 寻找匹配的 `"' 是遇到了未预期的文件结束符
get_netinfo-error.sh:行20: 语法错误: 未预期的文件结尾
```

脚本跟踪实践

```
失败演示
[root@rocky9 ~]# /bin/bash -x get_netinfo-error.sh
+ :
+ ifconfig eth0
+ awk '{print $2}'
+ grep -w inet
+ xargs echo 'IP: '
IP:  10.0.0.12
+ ifconfig eth0
+ grep -w inet
+ awk '{print $4}'
+ xargs echo 'NetMask: '
NetMask:  255.255.255.0
+ ifconfig eth0
+ grep -w inet
+ xargs echo 'Broadcast: '
+ awk '{print $6}'
Broadcast:  10.0.0.255
```

```
get_netinfo-error.sh:行19: 寻找匹配的 `"' 是遇到了未预期的文件结束符
get_netinfo-error.sh:行20: 语法错误: 未预期的文件结尾
```

```
成功演示
[root@rocky9 ~]# /bin/bash -x get_netinfo.sh
+ :
+ ifconfig eth0
+ awk '{print $2}'
+ grep -w inet
+ xargs echo 'IP: '
IP:  10.0.0.12
+ ifconfig eth0
+ grep -w inet
+ awk '{print $4}'
+ xargs echo 'NetMask: '
NetMask:  255.255.255.0
+ ifconfig eth0
+ grep -w inet
+ awk '{print $6}'
+ xargs echo 'Broadcast: '
Broadcast:  10.0.0.255
+ ifconfig eth0
+ grep ether
+ awk '{print $2}'
+ xargs echo 'MAC Address: '
MAC Address:  00:0c:29:23:23:8c
```

1.2.5 脚本开发规范

开发规范

脚本规范

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

规范解析

编写梳理

```
shell脚本开发规范重点：2-4-5
shell脚本开发小技巧：3-6-7
```

其他技巧

```
1 尽可能记忆更多的命令
2 掌握脚本的标准的格式
3 多看、多模仿、多思考
```

2 shell变量

2.1 变量基础

2.1.1 变量场景

数据存储

数据存储

所谓的数据存储，我们从三方面来理解这句话：

1、数据保存到哪里 -- 各种媒介，CPU、内存、磁盘、磁带、网盘...

2、数据保存的效果 -- 完整、安全、有效

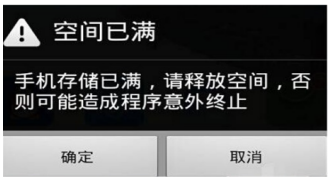
3、数据保存的单元 -- 存储空间

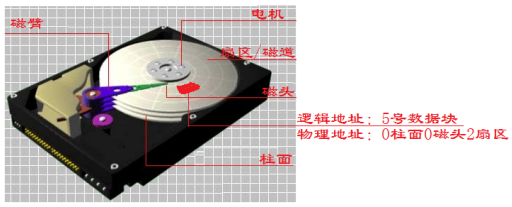
数据的存储空间默认不是一个整体，而是由一个个的存储单元组成，每一个存储单元都有一个唯一的整数编号，我们称这个编号为：地址

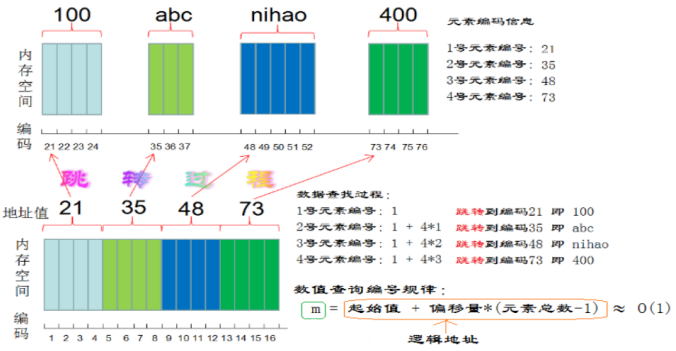
存储单元的作用：存储数据+读写数据

存储空间大小：1字节(B) = 8bit == 00000000

地址特点：十六进制，例如：0x20000000







对于数据的存储来说，主要有两种样式：物理地址和逻辑地址。

物理地址：内存或硬盘中真正存储数据的位置，也就是说通过磁盘设备查找的位置

逻辑地址：用于查找物理地址的存储块地址叫逻辑地址。程序中用的地址一般都是逻辑地址

逻辑地址包括两部分：起始值(十六进制)+偏移量(十六进制)

数据表的描述主要是逻辑地址，因为程序一般存储的是逻辑地址。

```
数据一旦存储下来就不再发生变化了，而程序中可能会在很多场景中用到同一个数据，就会出现两个问题：
    1 物理地址人听得懂，机器看不懂
        - 所以用逻辑地址找物理地址
    2 软件可以通过逻辑地址找到数据地址，但是软件不懂场景
        - 所以人用逻辑地址的别名来代指向同一个xx地址

应用程序中为了 多场景应用这个逻辑地址的别名，本质上就是“变量”。
```

变量场景

变量的本质

```
变量的本质其实就是 通过一个名称帮助程序快速找到内存中具体数据的地址。
    - 变量说白了就是指向xx值。
```

编程语言

```
编程语言在数据调用层面分类的话，可以分为两类：
    静态编译语言：
        使用变量前，先声明变量类型，之后类型不能改变，在编译时检查。
        如：java，c
    动态编译语言：
        不用事先声明，可随时改变类型。
        如：bash，Python
```

```
根据编程语言在使用变量的程度上，可以划分为强类型、弱类型语言：
    强类型语言：
        不同类型数据操作，必须经过强制转换才同一类型才能运算。
            如java ，  c# ，python
        示例：
            print('shuji' + 10) 提示出错，不会自动转换类型
            print('shuji' + str(10)) 结果为magedu10，需要显示转换类型
    弱类型语言：
        语言的运行时会隐式做数据类型转换。无须指定类型，默认均为字符型；
        参与运算会自动进行隐式类型转换；变量无须事先定义可直接调用。
        如：bash ，php，javascript
        示例：
            echo 'aaa'+222
```

2.1.2 变量定义

变量定义

变量定义

```
变量包括三部分：
    变量名 - 不变的
    变量值 - 变化的
    赋值动作 - 变量名指向变量值
表现样式：
    变量名=变量值
```

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

命名规范

```
1 名称有意义
2 名称细节
    命名只能使用英文字母，数字和下划线，首个字符不能以数字开头。
    中间不能有空格，可以使用下划线（_）。
    不能使用标点符号。
    不能使用bash里的关键字（可用help命令查看保留关键字）。
3 命名样式
    大驼峰HelloWorld,每个单词的首字母是大写
    小驼峰helloWorld,第一个单词的首字母小写，后续每个单词的首字母是大写
    下划线：Hello_World
    大小写字母： helloworld, HELLOWORLD
```

```
注意：
    对于开发人员来说，他们对于变量名的规范比较多，什么类、函数、对象、属性、命名空间等都有要求
    对于运维人员来说，记住一个词 -- 有意义。
```

变量分类

```
shell 中的变量分为三大类：
    本地变量          变量名仅仅在当前终端有效
    全局变量          变量名在当前操作系统的所有终端都有效
    shell内置变量     shell解析器内部的一些功能参数变量
```

```
注意：
    这里的变量分类的特点仅仅是从字面上来理解的，因为在实际的操作的时候，还会涉及到环境优先级的问题
    所以生产中对于这三者的划分没有特别大的强制。
```

2.1.3 基本操作

变量查看

语法解析

```
基本格式
    $变量名
```

示例

```
查看默认的shell类型
[root@rocky9 ~]# echo $SHELL
/bin/bash
```

变量定义

普通语法解析

```
基本格式
    变量名=变量值

注意：
    = 两侧不允许有空格
```

示例

```
查看一个空值变量名
[root@rocky9 ~]#  echo $myname

定制变量实践
[root@rocky9 ~]# myname=shuji
[root@rocky9 ~]#  echo $myname
shuji
```

```
错误的定制变量命令
[root@rocky9 ~]#  echo $myage

[root@rocky9 ~]# myage = 18
bash: myage: 未找到命令
[root@rocky9 ~]#  echo $myage
```

类型变量定义

```
命令语法
    declare 参数 变量名=变量值
参数解析：
    -i 将变量看成整数
    -r 使变量只读  readonly,==**该变量的值无法改变，并且不能为unset**==
    -x 标记变量为全局变量，类似于export
    -a  指定为索引数组（普通数组）；查看普通数组
    -A 指定为关联数组；查看关联数组
注意：
    在生产场景中，这种方法比较鸡肋，使用频率 0-20次/3年
```

```
设定制定类型的变量值
[root@rocky9 ~]# declare -i mynum='shuzi'
[root@rocky9 ~]# echo $mynum
0
[root@rocky9 ~]# declare -i mynum='123456'
[root@rocky9 ~]# echo $mynum
123456
```

```
设定只读类型变量
[root@rocky9 ~]# declare -r myread1="aaa"
[root@rocky9 ~]# myread2=myread
[root@rocky9 ~]# readonly myread2
```

```
查看只读变量
[root@rocky9 ~]# declare -r | grep myread
declare -r myread1="aaa"
declare -r myread2="myread"
[root@rocky9 ~]# readonly -p | grep myread
declare -r myread1="aaa"
declare -r myread2="myread"

无法使用unset删除只读变量
[root@rocky9 ~]# unset myread1 myread2
bash: unset: myread1: 无法反设定: 只读 variable
bash: unset: myread2: 无法反设定: 只读 variable

借助于exit方式删除只读变量
[root@rocky9 ~]# exit
...... 重新登录后再次查看
[root@rocky9 ~]# declare -r | grep myread
[root@rocky9 ~]#
```

变量移除

语法解析

```
基本格式
    unset 变量名
```

示例

```
查看刚才定制的变量名
[root@rocky9 ~]#  echo $myname
shuji

移除变量名
[root@rocky9 ~]# unset myname
[root@rocky9 ~]#  echo $myname

[root@rocky9 ~]#
```

2.2 本地变量

2.2.1 本地变量分类

基础知识

本地变量

```
所谓的本地变量就是：在当前系统的某个环境下才能生效的变量，作用范围小。
```

变量分类

```
本地变量按照变量值的生成方式包含两种：
    普通变量：
        自定义变量名和变量值
    命令变量：
        自定义变量名，而变量值是通过一条命令获取的
```

2.2.2 普通变量

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

变量分类

```
所谓的本地变量就是：在当前系统的某个环境下才能生效的变量，作用范围小。本地变量按照变量值的生成方式包含两种：
    普通变量：
        自定义变量名和变量值
    命令变量：
        自定义变量名，而变量值是通过一条命令获取的
```

基本格式

序号	样式	要点
方式一	变量名=变量值	变量值必须是一个整体，中间没有特殊字符 "=" 前后不能有空格
方式二	变量名='变量值'	原字符输出，我看到的内容，我就输出什么内容，
方式三	变量名="变量值"	如果变量值范围内，有可以解析的变量A，那么首先解析变量A， 将A的结果和其他内容组合成一个整体，重新赋值给变量B

```
习惯：
    数字不加引号，其他默认加双引号
    因为bash属于弱类型语言，默认会将所有内容当成字符串
```

变量定义

```
查看默认的空值变量
[root@rocky9 ~]# echo $name

方法1设定变量
[root@rocky9 ~]# name=shuji
[root@rocky9 ~]# echo $name
shuji

方法2设定变量
[root@rocky9 ~]# name='shuji1'
[root@rocky9 ~]# echo $name
```

```
shuji1

方法3设定变量
[root@rocky9 ~]# name="shuji2"
[root@rocky9 ~]# echo $name
shuji2

清理变量
[root@rocky9 ~]# unset name
```

作用区别演示

```
查看默认的空值变量
[root@rocky9 ~]# echo $name2

方法1设定变量的要点，变量值必须是一个整体
[root@rocky9 ~]# name2=shuji haoshuai
bash: haoshuai: 未找到命令
[root@rocky9 ~]# echo $name2
原因解析：
    空格是一个特殊符号，表示两条命令的隔开
    它将shuji 和 haoshuai当成两条命令了，所以报错信息是命令找不到

方法2设定变量
[root@rocky9 ~]# name2='shuji haoshuai'
[root@rocky9 ~]# echo $name2
shuji haoshuai

方法3设定变量
[root@rocky9 ~]# name2="shuji haoweiwu"
[root@rocky9 ~]# echo $name2
haoweiwu

清理变量
[root@rocky9 ~]# unset name2
```

```
定制基础变量
[root@rocky9 ~]# name=shuji
[root@rocky9 ~]# echo $name
shuji

方法2设定变量
[root@rocky9 ~]# name2='dan-$name'
[root@rocky9 ~]# echo $name2
dan-$name

方法3设定变量
[root@rocky9 ~]# name2="shuang-$name"
[root@rocky9 ~]# echo $name2
shuang-shuji
```

2.2.3 命令变量

基础知识

基本格式

```
定义方式一：
    变量名=`命令`
    注意：
    ` 是反引号

定义方式二：
    变量名=$(命令)
```

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

简单实践

命令变量实践

```
查看默认的空值变量
[root@rocky9 ~]# echo $myuser

方法1设定变量名
[root@rocky9 ~]# myuser=`whoami`
[root@rocky9 ~]# echo $myuser
root
```

```
查看默认的空值变量
[root@rocky9 ~]# echo $mydir

方法2设定变量名
[root@rocky9 ~]# mydir=$(pwd)
[root@rocky9 ~]# echo $mydir
/root

清理变量
[root@rocky9 ~]# unset mydir myuser
```

其他常见的实践

```
自动生成一系列数字
[root@rocky9 ~]# NUM=`seq 10`
[root@rocky9 ~]# echo $NUM
1 2 3 4 5 6 7 8 9 10

文件备份添加时间戳
[root@rocky9 ~]# touch file-a
[root@rocky9 ~]# cp file-a file-a-$(date +%F)
[root@rocky9 ~]# ls file-a*
file-a  file-a-2022-06-08
```

简单小综合实践

```
[root@rocky9 ~]# cat get_netinfo_v2.sh
#!/bin/bash
# 功能：获取当前主机的网卡设备信息
# 作者：wangshuji
# 版本：V0.2
# 联系：www.sswang.com

# 定制基础变量
RED="\E[1;31m"
GREEN="echo -e \E[1;32m"
END="\E[0m"

# 获取ip地址信息
IPDDR=$(ifconfig eth0 | grep -w inet | awk '{print $2}')
# 获取掩码地址信息
NETMAST=$(ifconfig eth0 | grep -w inet | awk '{print $4}')
# 获取广播地址信息
BROADCAST=$(ifconfig eth0 | grep -w inet | awk '{print $6}')
# 获取MAC地址信息
MACADDR=$(ifconfig eth0 | grep ether | awk '{print $2}')

# 打印网络基本信息
$GREEN---------主机网卡基本信息---------$END
echo -e  "HOSTNAME:      $RED `hostname` $END"
echo -e  "IP:            $RED $IPDDR $END"
echo -e  "NetMask:       $RED $NETMAST $END"
echo -e  "Broadcast:     $RED $BROADCAST $END"
echo -e  "MAC Address:   $RED $MACADDR $END"
$GREEN---------主机网卡基本信息---------$END
```

文件执行后效果

```
[root@localhost ~]# /bin/bash get_netinfo_v2.sh
---------主机网卡基本信息---------
 HOSTNAME:      localhost
 IP:            10.0.0.12
 NetMask:       255.255.255.0
 Broadcast:     10.0.0.255
 MAC Address:   00:0c:29:23:23:8c
---------主机网卡基本信息---------
[root@localhost ~]#
```

2.3 全局变量

2.3.1 基本操作

基础知识

基本定义

全局变量是什么

全局变量就是：在当前系统的所有环境下都能生效的变量。

基本语法

```
查看全局环境变量
    env          只显示全局变量,一般结合 grep 和管道符来使用
    printenv     效果与env等同

    export       查看所有的环境变量，包括声明的过程等信息，一般不用
    declare -x   效果与export类似
```

```
定义全局变量方法一：
    变量=值
    export 变量
定义全局变量方法二：（最常用）
    export 变量=值
```

查看全局变量

```
查看所有的全局变量
[root@rocky9 ~]# env
XDG_SESSION_ID=4
HOSTNAME=localhost
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
...

查看制定的全局变量
[root@rocky9 ~]# env | grep SHELL
SHELL=/bin/bash
```

```
定制本地变量
[root@rocky9 ~]# envtype=local
[root@rocky9 ~]# echo $envtype
local

从全局变量中查看
[root@rocky9 ~]# env | grep envtype
[root@rocky9 ~]#
结果显示：
    无法从全局变量中查看本地变量的名称
```

```
方法1定制全局变量
[root@rocky9 ~]# echo $envtype
local
[root@rocky9 ~]# env | grep envtype
[root@rocky9 ~]# export envtype
[root@rocky9 ~]# env | grep envtype
envtype=local

方法2定制全局变量
[root@rocky9 ~]# export myuser=root
[root@rocky9 ~]# env | grep myuser
myuser=root
```

```
鸡肋方法定制全局变量
[root@rocky9 ~]# declare -x mydir=/root
[root@rocky9 ~]# env | grep mydir
mydir=/root

清理全局变量
[root@rocky9 ~]# unset envtype myuser mydir
```

2.3.2 文件体系

变量文件

变量文件

```
在linux环境中，有很多目录下的文件都可以定制一些作用范围更广的变量，这些文件或文件所在的目录有：
    作用范围在制定的用户范围：
        ~/.bashrc
        ~/.bash_profile
    作用的范围在系统范围：
        /etc/profile
        /etc/profile.d/env_file_name
```

简单实践

bashrc 或 bash_profile 实践

```
查看未知的变量名
[root@rocky9 ~]# echo $NAME

定制变量名到文件中
[root@rocky9 ~]# echo NAME=shuji >> ~/.bashrc
[root@rocky9 ~]# source ~/.bashrc
[root@rocky9 ~]# echo $NAME
shuji

新开一个终端查看效果
[root@rocky9 ~]# echo $NAME
shuji

新开一个普通用户的终端查看效果
[root@rocky9 ~]# su - python
[python@rocky9 ~]$ echo $NAME

[python@rocky9 ~]$
```

```
清理.bashrc 文件里的变量，然后清除当前环境下的变量名
unset NAME
```

profile实践

```
查看未知的变量名
[root@rocky9 ~]# echo $PROFILE
```

```
定制变量名到文件中
[root@rocky9 ~]# echo PROFILE=shuji >> /etc/profile
[root@rocky9 ~]# source /etc/profile
[root@rocky9 ~]# echo $PROFILE
shuji

新开一个终端查看效果
[root@rocky9 ~]# echo $PROFILE
shuji

新开一个普通用户的终端查看效果
[root@rocky9 ~]# su - python
[python@rocky9 ~]$ echo $PROFILE
shuji
[python@rocky9 ~]$
```

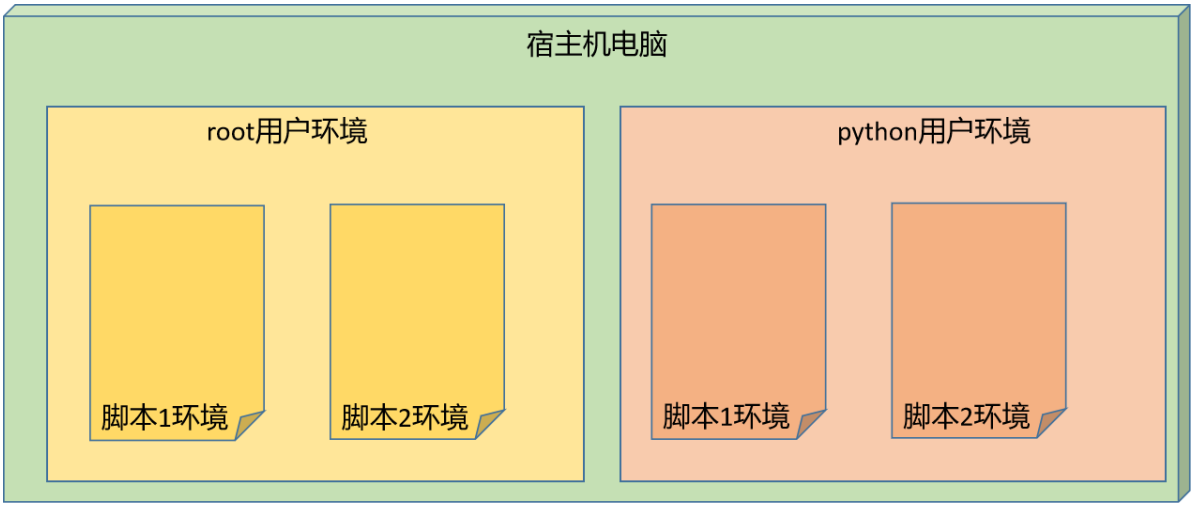
2.3.3 嵌套shell

export原理

原理解析

```
用户登录时:
    用户登录到Linux系统后，系统将启动一个用户shell。
    在这个shell中，可以使用shell命令或声明变量，也可以创建并运行 shell脚本程序。

运行脚本时:
    运行shell脚本程序时，系统将创建一个子shell。 此时，系统中将有两个shell
        - 一个是登录时系统启动的shell，另一个是系统为运行脚本程序创建的shell。
        当一个脚本程序运行完毕，它的脚本shell将终止，可以返回到执行该脚本之前的shell。
```




```
意义解读
    从这种意义上来说，用户可以有许多 shell，每个shell都是由某个shell（称为父shell）派生的。在子shell中定义的变量只在该子shell内有效。如果在一个shell脚本程序中定义了一个变量，当该脚本程序运行时，这个定义的变量只是该脚本程序内的一个局部变量，其他的shell不能引用它，要使某个变量的值可以在其他shell中被改变，可以使用export命令对已定义的变量进行输出。

    export命令将使系统在创建每一个新的shell时定义这个变量的一个拷贝。这个过程称之为变量输出。
```

实践解读

```
    当前父shell中定义变量中,分为局部变量和全局变量，不同点是：
        - 局部变量只能作用于本父shell，子shell无法继续使用
        - 如果使用了export将局部变量定义为全局变量，那么子shell创建的时候会继承父shell的全局
变量
```

嵌套实践

简单实践

```
查看父shell的脚本
[root@rocky9 ~]# cat father.sh
#!/bin/bash
# 定制全局变量
export _xing='王'
_name="wangshusen"
_age="42"
echo "父shell信息: $_xing$_name,$_age"
sleep 3

# 调用child.sh进行验证,最好放在同一目录下
/bin/bash child.sh
echo "父shell信息: $_xing$_name,$_age"
```

```
查看子shell的脚本
#!/bin/bash
# 显示父shell的全局变量
echo "子shell信息: $_xing$_name,$_age"

# 同名变量 子shell 的优先级高于父shell,但是不会传递给父shell
_xing="王胖胖"
echo "子shell修改后的信息: $_xing"
```

```
执行测试效果
[root@rocky9 ~]# /bin/bash father.sh
父shell信息: 王wangshusen,42
子shell信息: 王,
子shell修改后的信息: 王胖胖
父shell信息: 王wangshusen,42
```

2.4 内置变量

2.4.1 脚本相关

基础知识

脚本相关的变量解析

序号	变量名	解析
1	$0	获取当前执行的shell脚本文件名
2	$n	获取当前执行的shell脚本的第n个参数值，n=1..9， 当n为0时表示脚本的文件名，如果n大于9就要用大括号括起来${10}
3	$#	获取当前shell命令行中参数的总个数
4	$?	获取执行上一个指令的返回值（0为成功，非0为失败）

简单实践

实践1 - $0 获取脚本的名称

```
[root@rocky9 ~]# cat get_name.sh
#!/bin/bash
# 获取脚本的名称
echo "我脚本的名称是： file.sh"
echo "我脚本的名称是：$0"
```

实践2 - $n 获取当前脚本传入的第n个位置的参数

```
[root@rocky9 ~]# cat get_args.sh
#!/bin/bash
# 获取指定位置的参数
echo "第一个位置的参数是： $1"
echo "第二个位置的参数是： $2"
echo "第三个位置的参数是： $3"
echo "第四个位置的参数是： $4"
```

实践3 - $# 获取当前脚本传入参数的数量

```
[root@rocky9 ~]# cat get_number.sh
#!/bin/bash
# 获取当前脚本传入的参数数量
echo "当前脚本传入的参数数量是： $#"
```

实践4 - $? 获取文件执行或者命令执行的返回状态值

```
[root@rocky9 ~]#  bash nihao
bash: nihao: No such file or directory
[root@rocky9 ~]#  echo $?
127

[root@rocky9 ~]#  ls
get_name.sh get_args.sh get_number.sh
[root@rocky9 ~]#  echo $?
0
```

```
```

2.4.2 字符串相关

基础知识

字符串相关的变量解析

```
字符串计数
    ${#file}              获取字符串的长度

字符串截取
    - 语法为${var:pos:length}    表示对变量var从pos开始截取length个字符，pos为空标示0
    ${file:0:5}           从0开始，截取5个字符
    ${file:5:5}           从5开始，截取5个字符
    ${file::5}            从0开始，截取5个字符
    ${file:0-6:3}         从倒数第6个字符开始，截取之后的3个字符
    ${file: -4}           返回字符串最后四个字节，-前面是"空格"
```

简单实践

字符串实践

```
定制字符串内容
[root@rocky9 ~]# string_context="dsjfdsafjkldjsklfajkdsa"
[root@rocky9 ~]# echo $string_context
dsjfdsafjkldjsklfajkdsa
```

```
获取字符串长度
[root@rocky9 ~]# echo ${#string_context}
23
```

```
从0开始，截取5个字符
[root@rocky9 ~]# echo ${string_context:0:5}
dsjfd

从5开始，截取5个字符
[root@rocky9 ~]# echo ${string_context:5:5}
safjk

从0开始，截取5个字符
[root@rocky9 ~]# echo ${string_context::5}
dsjfd

从倒数第6个字符开始，截取之后的3个字符
[root@rocky9 ~]# echo ${string_context:0-6:3}
ajk

返回字符串最后四个字节，-前面是"空格"
[root@rocky9 ~]# echo ${string_context: -4}
kdsa
```

```
```

2.4.3 默认值相关

基础知识

语法解读

```
格式一：${变量名:-默认值}
      变量a如果有内容，那么就输出a的变量值
      变量a如果没有内容，那么就输出默认的内容

格式二：${变量名+默认值}
      无论变量a是否有内容，都输出默认值
```

实践1 - 有条件的默认值

```
购买手机的时候选择套餐：
      如果我输入的参数为空，那么输出内容是 "您选择的套餐是： 套餐 1"
      如果我输入的参数为n，那么输出内容是 "您选择的套餐是： 套餐 n"
```

```
[root@rocky9 ~]# select_default_value.sh
#!/bin/bash
# 套餐选择演示
a="$1"
echo "您选择的手机套餐是： 套餐 ${a:-1}"
```

实践2 - 强制默认值

```
国家法律强制规定：
      不管我说国家法定结婚年龄是 多少岁，都输出 国家法定结婚年龄(男性)是 22 岁
```

```
[root@rocky9 ~]# froce_default_value.sh
#!/bin/bash
# 默认值演示示例二
a="$1"
echo "国家法定结婚年龄(男性)是 ${a+22} 岁"
```

2.4.4 其他相关

基础知识

脚本相关的变量解析

序号	变量名	解析
1	$_	在此之前执行的命令或脚本的第一个内容
2	$@	传给脚本的所有参数
3	$*	是以一个单字符串显示里所有向脚本传递的参数，与位置参数不同，参数可超过9个
4	$$	是脚本运行的当前进程的ID号，作用是方便以后管理它杀掉他
5	$!	前一条命令进程的ID号，作用是方便以后管理它杀掉他

简单实践

实践1 - 其他变量的作用

```
[root@rocky9 ~]# cat get_other.sh
#!/bin/sh
echo "脚本执行命令的第一个内容: $_"
echo "传递给当前脚本的所有参数是: $@"
echo "单字符串显示所有参数: $*"
echo "当前脚本执行时候的进程号是: $$"
sleep 5 &
echo "上一条命令执行时候的进程号是: $!"
```

实践2 - $$ 获取当前的进程号

```
查看当前的进程号
[root@rocky9 ~]# echo $$
4759
[root@rocky9 ~]# ps aux | grep 4759
root        4759  0.0  0.0 116712  3356 pts/1    Ss   00:11   0:00 -bash
root        5547  0.0  0.0 112828   984 pts/1    S+   02:00   0:00 grep --
color=auto 4759
杀死当前的进程
[root@rocky9 ~]# kill -9 4759
————————————————————————————————————
Session stopped
    - Press <return> to exit tab
    - Press R to restart session
    - Press S to save terminal output to file
```

实践3 - $@ 和 $* 的区别

```
定制father脚本
[root@rocky9 ~]# cat father.sh
#!/bin/bash
echo "$0: 所有的参数 $@"
echo "$0: 所有的参数 $*"
echo '将 $* 值传递给 child-1.sh 文件'
/bin/bash child-1.sh "$*"

echo '将 $@ 值传递给 child-2.sh 文件'
/bin/bash child-2.sh "$@"
```

```
定制两个child脚本
[root@rocky9 ~]# cat child-1.sh
#!/bin/bash
echo "$0: 获取所有的参数 $1"

[root@rocky9 ~]# cat child-2.sh
#!/bin/bash
echo "$0: 获取所有的参数 $1"
```

```
执行 father.sh 脚本
[root@rocky9 ~]# /bin/bash father.sh 1 2 3
father.sh: 所有的参数 1 2 3
father.sh: 所有的参数 1 2 3
将 $* 值传递给 child-1.sh 文件
child-1.sh: 获取所有的参数 1 2 3
将 $@ 值传递给 child-2.sh 文件
child-2.sh: 获取所有的参数 1
```

```
```

3 脚本交互

3.1 基础知识

3.1.1 shell登录解读

基础知识

shell配置文件

```
系统级别生效配置文件
    /etc/profile
        系统的每个用户设置环境信息,当用户第一次登录时,该文件被执行
    /etc/profile.d/*.sh
        被/etc/profile文件调用，执行当前目录下所有的文件中关于shell的设置
    /etc/bashrc
        为每一个运行bash shell的用户执行此文件.当bash shell被打开时,该文件被读取。
用户级别生效配置文件
    ~/.bash_profile
        设定用户专用的shell信息,当用户登录时,该文件仅仅执行一次
    ~/.bashrc
        该文件包含用户专用的bash信息,当登录时以及每次打开新的shell时,该文件被读取
```

```
用户退出生效配置文件
    ~/.bash_logout: 当每次退出系统(退出bash shell)时,执行该文件.
    ~/.bash_history:
        用户登录时自动读取其中的内容并加载到内存hiatory记录中
        logout时将内存中的history记录写入该文件中
```

shell的登录方式

```
交互式登录
    方法1：密码登录
        直接通过终端输入账号密码登录
        复制终端
    方法2：su 变更shell操作用户的身份
        su - 用户名
        超级用户除外，需要键入该使用者的密码。
非交互式登录
    方法1：脚本执行
    方法2：su 用户名
```

```
登录shell的文件生效流程
    /etc/profile.d/*.sh
        -> /etc/profile
            -> /etc/bashrc
                -> ~/.bashrc
                    -> ~/.bash_profile
非登录shell的文件生效流程
    /etc/profile.d/*.sh
        -> /etc/bashrc
            -> ~/.bashrc
注意：
    若多配置文件中设置相同的变量，则后面配置文件中变量的值会覆盖前面配置文件中同一变量的值。
```

```
su的相关参数
    -：当前用户不仅切换为指定用户的身份，同时所用的工作环境也切换为此用户的环境。
    -l：同 - 的使用类似，完整切换工作环境，后面需要添加欲切换的使用者账号。
    -p：表示切换为指定用户的身份，但不改变当前的工作环境（不使用切换用户的配置文件）。
    -m：和 -p 一样；
    -c 命令：仅切换用户执行一次命令，执行后自动切换回来，该选项后通常会带有要执行的命令。
```

配置文件修改后生效的方法

```
修改profile和bashrc文件后需生效两种方法
    1. 重新启动shell进程
    2. source|. 配置文件

注意:
    source 会在当前shell中执行脚本,所有一般只用于执行置文件,或在脚本中调用另一个脚本的场景
```

简单实践

准备工作

```
为所有的shell相关的配置文件添加关键信息
echo "echo '1 - /etc/profile'" >> /etc/profile
echo "echo '2 - /etc/profile.d/2.sh'" >> /etc/profile.d/2.sh
echo "echo '3 - /etc/bashrc'" >> /etc/bashrc
echo "echo '4 - ~/.bash_profile'" >> ~/.bash_profile
echo "echo '5 - ~/.bashrc'" >> ~/.bashrc
```

非登录效果

```
[root@rocky9 ~]# su - python
上一次登录：五 6月 10 16:16:37 CST 2022pts/1 上
2 - /etc/profile.d/2.sh
1 - /etc/profile
3 - /etc/bashrc
[python@rocky9 ~]$ su root
密码：
2 - /etc/profile.d/2.sh
3 - /etc/bashrc
5 - ~/.bashrc
[root@rocky9 /home/python]# exit
exit
```

登录查看效果

```
切换标准root用户
[python@rocky9 ~]$ su - root
密码：
上一次登录：日 6月 12 12:41:11 CST 2022pts/2 上
2 - /etc/profile.d/2.sh
1 - /etc/profile
3 - /etc/bashrc
5 - ~/.bashrc
4 - ~/.bash_profile

新建终端效果
Last login: Sun Jun 12 12:35:59 2022 from 10.0.0.1
2 - /etc/profile.d/2.sh
1 - /etc/profile
3 - /etc/bashrc
5 - ~/.bashrc
4 - ~/.bash_profile
[root@rocky9 ~]#
```

清理环境

```
将刚才创建的5条命令执行反向操作
[root@rocky9 ~]# vim ...

退出当前shell环境
[root@rocky9 ~]# exit
登出
Session stopped
    - Press <return> to exit tab
    - Press R to restart session
    - Press S to save terminal output to file
```

3.1.2 子shell基础

基础知识

场景

```
    之前我们对于环境变量在多个shell环境中的应用进行了学习，那种操作量比较大。对于一些临时性的场景，我们在临时性的环境中，做一些操作，但是不希望对外部的环境造成影响，这个时候我们就涉及到了一些临时shell环境的实践。
    关于临时shell环境的创建，我们可以借助于()方法来实现。
```

临时shell

```
临时shell环境 - 启动子shell
    (命令列表)，在子shell中执行命令列表，退出子shell后,不影响后续环境操作。
临时shell环境 - 不启动子shell
    {命令列表}, 在当前shell中运行命令列表,会影响当前shell环境的后续操作。
```

简单实践

() 实践

```
查看当前shell的pid
[root@rocky9 ~]#  echo $BASHPID
11413
[root@rocky9 ~]# ps aux | grep bash
root       11413  0.0  0.0 116724  3160 pts/0    Ss   12:54   0:00 -bash
root       11660  0.0  0.0 112824   984 pts/0    R+   14:49   0:00 grep --
color=auto bash

查看子shell的pid
[root@rocky9 ~]# (echo $BASHPID; echo haha)
11661
haha
[root@rocky9 ~]# (echo $BASHPID; sleep 30)
11711

另开一个终端查看效果
[root@rocky9 ~]# pstree | grep sleep
        |-sshd-+-sshd---bash---bash---sleep
结果显示：
    在一个shell内部开启了另一个shell
```

```
子shell的操作不影响当前shell环境
[root@rocky9 ~]# (export SUBSHELL=subshell)
[root@rocky9 ~]# echo $SUBSHELL

[root@rocky9 ~]#

子shell中，查看命令执行效果
[root@rocky9 ~]# (cd /tmp;pwd)
/tmp
[root@rocky9 ~]# pwd
/root
```

{} 实践

```
查看当前shell的进程id号
[root@rocky9 ~]# echo $BASHPID
11676

在{}环境中查看当前shell的进程id号
[root@rocky9 ~]# { echo $BASHPID; }
11676
```

```
{} 环境中，操作命令会影响当前的shell环境
[root@rocky9 ~]# { export SUBSHELL=subshell; }
[root@rocky9 ~]# echo $SUBSHELL
subshell

子shell中，查看命令执行效果
[root@rocky9 ~]# { cd /tmp;pwd; }
/tmp
[root@rocky9 /tmp]# pwd
/tmp
```

```
```

3.1.3 子shell实践

CA创建

umask基础

```
umask 解读
    umask指的是文件权限默认的掩码，默认的值是022，也就是说
        默认创建的目录是777-022=755
        默认创建的文件是666-022-544
```

```
[root@rocky9 /data/scripts]# umask
0022
[root@rocky9 /data/scripts]# mkdir dir
[root@rocky9 /data/scripts]# touch file
[root@rocky9 /data/scripts]# ll
总用量 0
drwxr-xr-x 2 root root 6 6月  12 15:18 dir
-rw-r--r-- 1 root root 0 6月  12 15:18 file
```

CA手工实践

```
创建临时目录
[root@rocky9 ~]# mkdir /tmp/CA; cd /tmp/CA

生成私钥
[root@rocky9 /tmp/CA]# (umask 077;openssl genrsa -out ca.key 2048)
Generating RSA private key, 2048 bit long modulus
.....................................................+++
...................+++
e is 65537 (0x10001)
```

```
[root@rocky9 /tmp/CA]# ll
总用量 4
-rw------- 1 root root 1679 6月  12 15:35 ca.key

生成证书
[root@rocky9 /tmp/CA]# openssl req -new -x509 -key ca.key -out ca.crt -days 3650
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:CN
State or Province Name (full name) []:BJ
Locality Name (eg, city) [Default City]:BJ
Organization Name (eg, company) [Default Company Ltd]:BJ
Organizational Unit Name (eg, section) []:BJ
Common Name (eg, your name or your server's hostname) []:www.example.com
Email Address []:bj.example.com
```

```
查看生成的文件
[root@rocky9 /tmp/CA]# ls
ca.crt  ca.key

查看证书信息
[root@rocky9 /tmp/CA]# openssl x509 -in ca.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            d6:25:a6:0e:be:98:ec:48
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=CN, ST=BJ, L=BJ, O=BJ, OU=BJ,
CN=www.example.com/emailAddress=bj.example.com
```

脚本实践

创建脚本

```
查看脚本定制内容
[root@rocky9 /data/scripts]# cat ca_create.sh
#!/bin/bash
# 功能: 创建自建CA
# 版本: v0.1
# 作者: wangshusen
# 联系: www.sswang.com

# 定制普通环境变量
CA_DIR="tls"
CA_DOMAIN="$1"
CA_KEY='tls.key'
CA_CRT='tls.crt'

# 创建CA证书
mkdir ${CA_DIR}
(umask 077; cd ${CA_DIR}; openssl genrsa -out tls.key 2048)
```

```
openssl req -new -x509 -key ${CA_DIR}/${CA_KEY} -out ${CA_DIR}/${CA_CRT} -subj
"/CN=${CA_DOMAIN}" -days 365
```

```
执行脚本
[root@rocky9 /data/scripts]# /bin/bash ca_create.sh www.example.com
Generating RSA private key, 2048 bit long modulus
....+++
.....................................+++
e is 65537 (0x10001)
[root@rocky9 /data/scripts]# ls tls/
tls.crt  tls.key

确认效果
[root@rocky9 /data/scripts]# openssl x509 -in tls/tls.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            e1:8b:55:da:65:04:fc:c7
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=www.example.com
```

3.2 脚本外交互

3.2.1 read基础

基础知识

场景需求

```
    虽然我们可以通过脚本传参的方式实现脚本一定程度的灵活性，但是生产工作中，有很多更加灵活的场景，需要我们在脚本运行的过程中，传递一些用户定制的具体信息。这个时候，普通的脚本参数就无法满足需求了。
    read 命令可以实现我们脚本内外的信息自由传递功能。
```

命令简介

```
    read命令是用于从终端或者文件中读取输入的内建命令，read命令读取整行输入，每行末尾的换行符不被读入。在read命令后面，如果没有指定变量名，读取的数据将被自动赋值给特定的变量REPLY。常用方式如下：
```

```
    read                从标准输入读取一行并赋值给特定变量REPLY。
    read answer         从标准输入读取输入并赋值给变量answer。
    read first last     从标准输入读取内容，将第一个单词放到first中，其他内容放在last中。
    read -s passwd      从标准输入读取内容，写入passwd，不输入效果
    read -n n name      从标准输入读取内容，截取n个字符，写入name，超过n个字符，直接退出
    read -p "prompt"    打印提示，等待输入，并将输入存储在REPLY中。
    read -r line        允许输入包含反斜杠。
    read -t second      指定超时时间,默认是秒，整数
    read -d sper        指定输入信息的截止符号
```

简单实践

命令操作

```
交互式接收用户信息
[root@rocky9 ~]# read
nihao-answer

接收用户输入给一个临时变量
[root@rocky9 ~]# read answer
nihao-answer
[root@rocky9 ~]# echo $answer
nihao-answer

接收多个信息，按照顺序交给不同的临时变量
[root@rocky9 ~]# read first last
first-1 last-2 end-3
[root@rocky9 ~]# echo $first
first-1
[root@rocky9 ~]# echo $last
last-2 end-3
```

实践2-静默显示

```
显式接收用户输入信息
[root@rocky9 ~]# read password
123456
[root@rocky9 ~]# echo $password
123456

隐式接收用户输入信息
[root@rocky9 ~]# read -s password
[root@rocky9 ~]# echo $password
abcdefg
```

实践3-提示用户输入信息

```
通过 -p 参数提示用户输入的信心
[root@rocky9 ~]# read -p "请输入登录用户名: " user
请输入登录用户名: root
[root@rocky9 ~]# echo $user
root
```

实践4-限制用户输入信息

```
[root@rocky9 ~]# read -n 6 -p "sss: " aaa
sss: 123456[root@rocky9 ~]# read -n 6 -p "只接收6个字符，超过自动退出: " string
只接收6个字符，超过自动退出: 123456[root@rocky9 ~]#
[root@rocky9 ~]# echo $string
123456
注意：
    -p + -s 的组合会导致不会自动换行，可以结合 echo的方式实现换行
```

实践5-等待时长

```
[root@rocky9 ~]# read -t 5 -p "等待5秒后自动退出！ " second
等待5秒后自动退出！ 4
[root@rocky9 ~]# echo $second
4
[root@rocky9 ~]# read -t 5 -p "等待5秒后自动退出！ " second
等待5秒后自动退出！ [root@rocky9 ~]#
```

```

```

3.2.2 案例实践

学习目标

这一节，我们从 登录模拟、堡垒机实践、小结 三个方面来学习。

登录模拟

需求

```
模拟shell终端工具的登录，功能过程如下：
    请输入用户名：
    请输入密码：
    您输入的用户名和密码是： xxx
```

脚本实践

```
[root@rocky9 ~]# cat simple_login.sh
#!/bin/bash
# 功能: 模拟shell登录
# 版本：v0.1
# 作者：wangshusen
# 联系：www.sswang.com

# 定制命令变量
OS_INFO=$(cat /etc/redhat-release)
KERNEL_INFO=$(uname -r)
OS_ARCH=$(uname -m)
HOSTNAME=$(hostname)

# 清屏
clear

# 输出提示信息
echo -e "\e[32m${OS_INFO} \e[0m"
echo -e "\e[32mKernel ${KERNEL_INFO} on an ${OS_ARCH} \e[0m"
echo "---------------------------------"
# 交互输入登陆信息
read -p "请输入用户名：" account
read -s -t30 -p "请输入登录密码：" password
echo
echo "---------------------------------"
# 输出用户输入信息
printf "您输入的用户名：\e[31m%s\e[0m您输入的密码：\e[31m%s\e[0m\n" ${account}
${password}
```

```
脚本执行效果
[root@10 ~]# /bin/bash simple_login.sh
CentOS Linux release 7.9.2009 (Core)
Kernel 3.10.0-1160.el7.x86_64 on an x86_64
---------------------------------
请输入用户名：root
请输入登录密码：
---------------------------------
您输入的用户名：root您输入的密码：123456
```

堡垒机实践

功能需求

```
模拟堡垒机的登录，功能过程如下：
    请选择要登录的主机
    请输入用户名：
    使用指定的用户连接远程主机
```

脚本实践

```
[root@rocky9 ~]# cat simple_jumpserver.sh
#!/bin/bash
# 功能：定制堡垒机的展示页面
# 版本：v0.2
# 作者：wangshusen
# 联系：www.sswang.com

# 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

# 远程连接主机
ssh $user@rocky9
```

```
脚本执行效果
[root@10 ~]# /bin/bash simple_jumpserver.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: p
请输入登录本地主机的用户名: root
The authenticity of host 'localhost (::1)' 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 'localhost' (ECDSA) to the list of known hosts.
root@rocky9's password:
Last login: Mon Jun 13 12:19:34 2022 from 10.0.0.1
[root@10 ~]# exit
登出
```

4 表达式

4.1 运算符

4.1.1 运算符基础

基础知识

需求

```
    根据我们之前的学习，通过现有的知识内容可以完成一个简单的功能操作，即使通过所谓的脚本参数可以实现一个脚本在多个数据值的情况下实现不同的结果。但是问题是，目前脚本本身还没有实现灵活的机制。
    所谓脚本级别的灵活机制，说的是脚本内部能够实现数据的操作和判断。而数据的操作也是判断过程中必须的一个条件组成部分。所以数据的操作是脚本的一个核心内容。
```

数据操作

```
    关于shell可以实施的数据操作，按照不同的业务场景主要可以分为如下两个方面：
    运算符 - 数据值之间的操作
        赋值运算
            - 结果值输出
            - 示例： =、*=、/=、%=、+=、-=、<<=、>>=、&=、^=、|=等
        二元运算
            - 数据值操作
            - 示例： +、-、*、/、%等
        高阶运算
            - 更高一级的数学运算
            - 示例：**、^、++、--、
        其他运算
            - 其他运算操作
            - 示例：<<、>>等
注意：
    这些所谓的运算符一般很难单独来使用，都需要结合计算表达式来实现相应的效果
```

```
表达式 - 数据值在特定场景的运算符操作
    计算表达式
        - 将多个值的运算操作执行起来
        - 示例：bc、let、expr、$(())等
    测试表达式
        - 判断结果值是否满足需求
        - 示例：test、[]等
    逻辑表达式
        - 多条件的多场景组合
        - 示例：&&、||、and、or、not、&、|等
    比较表达式
        - 判断数据之间的适配关系
        - 示例：-f|d|s、-r|x|w、-e、-n、==、!=、>、<、<=、>=等
    三元表达式
        - 多逻辑的简单计算表达式
        - 示例：expr?expr:expr
    集合表达式
        - 表达式之间的关联关系
        - 示例：expr1 , expr2、 [[ ]]、[ -a|o ]、[ ! ]等
```

赋值运算

基础知识

所谓的赋值运算，其实指的就是将一个值赋予一个变量名，虽然我们在变量一节中对于该知识点进行了一些基础性的操作，但是赋值运算仍然还有一些其他的表现样式，尤其是关于命令计算相关的。

简单的赋值操作

```
为变量赋值
[root@rocky9 ~]# echo $num $string

[root@rocky9 ~]# num=123 string=abc
[root@rocky9 ~]# echo $num $string
123 abc
```

```
赋值时候定制属性
[root@rocky9 ~]# declare stringnum=123
[root@rocky9 ~]# declare string=nihao
[root@rocky9 ~]# echo $stringnum $string
123 nihao
[root@rocky9 ~]# declare -i num=654
[root@rocky9 ~]# declare -i num2=aaa
[root@rocky9 ~]# echo $num $num2
654 0
```

```
获取特定数据
[root@rocky9 ~]# myuser=$(whoami)
[root@rocky9 ~]# echo $myuser
root
[root@rocky9 ~]# kernel_info=$(cat /etc/redhat-release)
[root@rocky9 ~]# echo $kernel_info
CentOS Linux release 7.9.2009 (Core)
```

4.1.2 简单计算

学习目标

这一节，我们从 $[]、let、(())、$(())、小结 五个方面来学习。

$[]

简介

```
    $[]方法，常用于整数计算场景，适合不太复杂的计算，运算结果是小数的也会自动取整。
    后面的几种也是一样
```

格式

```
方法1：
    $[计算表达式]
方法2：
    a=$[变量名a+1]

注意：
    这里的表达式可以不是一个整体
```

简单示例

```
简单运算
[root@rocky9 ~]# echo $[100/5]
20
[root@rocky9 ~]# echo $[ 2 + 5 ]
7

变量参与运算
[root@rocky9 ~]# a=6
[root@rocky9 ~]# a=$[a+1]
[root@rocky9 ~]# echo $a
7

运算结果取整
[root@rocky9 ~]# echo $[100/3]
33
```

let

简介

```
    let是另外一种相对来说比较简单的数学运算符号了
```

格式

```
let 变量名a=变量名a+1

注意：
    表达式必须是一个整体，中间不能出现空格等特殊字符
```

简单示例

```
简单运算
[root@rocky9 ~]# i=1
[root@rocky9 ~]# let i=i+7
[root@rocky9 ~]# echo $i
8

let表达式必须是一个整体
[root@rocky9 ~]# let i = i * 2
bash: let: =: 语法错误: 期待操作数 （错误符号是 "="）
[root@rocky9 ~]# let i=i * 2
bash: let: anaconda-ks.cfg: 语法错误: 无效的算术运算符 （错误符号是 ".cfg"）
[root@rocky9 ~]# let i=i*2
[root@rocky9 ~]# echo $i
16
```

(())

简介

```
(())的操作与let基本一致，相当于let替换成了 (())
```

格式

```
((变量计算表达式))
注意：
    对于 $(())中间的表达式，可以不是一个整体，不受空格的限制
```

简单实践

```
[root@rocky9 ~]# num1=34
[root@rocky9 ~]# ((num2=num1+34))
[root@rocky9 ~]# echo $num2
68
```

$(())

简介

```
$(())的操作，相当于 (()) + echo $变量名 的组合
```

格式

```
echo $((变量计算表达式))
注意：
    对于 $(())中间的表达式，可以不是一个整体，不受空格的限制
```

简单实践

```
[root@rocky9 ~]# num1=34
[root@rocky9 ~]# echo $((num2=num1+34))
68
```

4.1.3 赋值运算进阶

二元运算

简介

```
    所谓的二元运算，指的是 多个数字进行+-*/%等运算
```

简单实践

```
加法运算
[root@rocky9 ~]# echo $(( 4 + 4 ))
8
[root@rocky9 ~]# num1=3 num2=4
[root@rocky9 ~]# echo $(($num1 + $num2))
7

减法运算
[root@rocky9 ~]# echo $((8-2))
6
[root@rocky9 ~]# echo $(($num2-$num1))
1

乘法运算
[root@rocky9 ~]# echo $((8*2))
16
[root@rocky9 ~]# echo $(($num2*$num1))
12

除法运算
[root@rocky9 ~]# echo $((8/2))
4
[root@rocky9 ~]# echo $(($num2/$num1))
1

取余运算
[root@rocky9 ~]# echo $((8%3))
2
[root@rocky9 ~]# echo $(($num2%$num1))
1
```

案例实践-小学计算题

```
案例需求：鸡兔同笼，共有30个头，88只脚。求笼中鸡兔各有多少只？
```

```
查看脚本内容
[root@rocky9 ~]# cat count_head_feet.sh
#!/bin/bash
# 功能：小学计算题目
# 版本：v0.1
# 作者：wangshusen
# 联系：www.sswang.com
```

```
# 定制普通变量
head="$1"
feet="$2"

# 计算逻辑
# 每个头至少两只脚，多出的脚都是兔子的
rabbit_num=$(( ($feet - $head - $head) / 2))
chick_num=$[ $head - $rabbit_num ]

# 结果输出
echo -e "\e[31m\t鸡和兔的数量\e[0m"
echo -e "\e[32m============================="
echo "鸡的数量: ${chick_num}"
echo "兔的数量：${rabbit_num}"
echo -e "=============================\e[0m"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash count_head_feet.sh 30 88
        鸡和兔的数量
=============================
鸡的数量: 16
兔的数量：14
=============================
```

赋值运算

简介

```
    这里的赋值运算，是一种进阶版本的复制操作，常见的样式如下：
        样式1：+=、-=、*=、/=
            - 在自身的基础上进行二元运算，结果值还是自己
        样式2：++、--
            - 在自身的基础上进行递增和递减操作，结果值还是自己
```

简单实践

```
+=运算
[root@rocky9 ~]# num1=6
[root@rocky9 ~]# let value+=$num1    # 相当于 let value=value+$num1
[root@rocky9 ~]# echo $value
6

-=运算
[root@rocky9 ~]# let value-=2    # 相当于 let value=value-2
[root@rocky9 ~]# echo $value
4

*=运算
[root@rocky9 ~]# let value*=2    # 相当于 let value=value*2
[root@rocky9 ~]# echo $value
8

/=运算
[root@rocky9 ~]# let value/=2    # 相当于 let value=value/2
[root@rocky9 ~]# echo $value
```

```
4
```

```
i++运算
[root@rocky9 ~]# value=9
[root@rocky9 ~]# let value++        # 相当于 let value=value+1
[root@rocky9 ~]# echo $value
10
[root@rocky9 ~]# let value++        # 相当于 let value=value+1
[root@rocky9 ~]# echo $value
11

++i运算
[root@rocky9 ~]# let ++value        # 相当于 let value=value+1
[root@rocky9 ~]# echo $value
12
```

```
--运算
[root@rocky9 ~]# value=9
[root@rocky9 ~]# let value--        # 相当于 let value=value-1
[root@rocky9 ~]# echo $value
8
[root@rocky9 ~]# let value--        # 相当于 let value=value-1
[root@rocky9 ~]# echo $value
7
[root@rocky9 ~]# let --value        # 相当于 let value=value-1
[root@rocky9 ~]# echo $value
6
```

4.1.4 expr计算

基础知识

简介

```
    expr即可以做常见的整数运算，还可以做数字比较，字符串计算等操作。
```

格式

```
数字场景：
    expr 运算表达式
字符串场景：
    match：用户获取匹配到字符串的长度
        expr match 字符串 匹配内容
    substr：截取字符串
        expr substr 字符串 起始位置 截取长度
        注意：起始位置值>=1
    index：查找第一次匹配字符的位置
        expr index 字符串 字符
    length：计算字符串的长度
        expr length 字符串
```

简单实践

数学运算

```
数学运算
[root@rocky9 ~]# expr 1 + 2 - 3 \* 4 / 5 + \( 6 - 7 \) \* 8
-7
[root@rocky9 ~]# x=1
[root@rocky9 ~]# expr $x + 4
5
```

字符串运算

```
用户获取匹配到字符串的长度
[root@rocky9 ~]# file=jdslkfajkldsjafklds
[root@rocky9 ~]# expr match $file "k.*j"
0
[root@rocky9 ~]# expr match $file ".*j"
13

截取字符串
[root@rocky9 ~]# expr substr $file 0 4
[root@rocky9 ~]# expr substr $file 1 4
jdsl

查找第一次匹配字符的位置
[root@rocky9 ~]# expr index $file b
0
[root@rocky9 ~]# expr index $file j
1

计算字符串的长度
[root@rocky9 ~]# expr length $file
19
```

```
```

4.1.5 bc计算

基础知识

简介

```
    bc是一种任意精度的计算语言，提供了语法结构，比如条件判断、循环等，功能是很强大的，还能进行进制转换。
```

```
常见参数
    -i 强制交互模式；
    -l 使用bc的内置库，bc里有一些数学库，对三角计算等非常实用；
    -q 进入bc交互模式时不再输出版本等多余的信息。
特殊变量
    ibase，obase 用于进制转换，ibase是输入的进制，obase是输出的进制，默认是十进制；
    scale 小数保留位数，默认保留0位。
```

简单实践

实践1-交互示例

```
在shell命令行直接输入bc即进入bc语言的交互模式
[root@rocky9 ~]# bc -l -q
4 / 3
1.33333333333333333333
3 + 4
7
5 * 8
40
exit
0
^C
(interrupt) Exiting bc.
[root@rocky9 ~]#
```

实践2 - 非交互示例

```
格式：echo "属性设置; 计算表达式" | bc
[root@rocky9 ~]# echo "scale=2; 9-8*2/5^2" | bc
8.36
[root@rocky9 ~]# echo "scale=2; sqrt(10)" | bc
3.16
[root@rocky9 ~]# echo '45.36-22.33' | bc
23.03
```

实践3 - 内存使用率统计

```
查看脚本效果
[root@rocky9 ~]# cat memory_info.sh
#!/bin/bash
# 功能：定制内存信息的使用率
# 版本：v0.1
# 作者：wangshusen
# 联系：www.sswang.com

# 定制基础信息
temp_file='/tmp/free.txt'
hostname=$(hostname)

# 获取内存信息
free -m > /tmp/free.txt
# 获得内存总量
memory_totle=$(grep -i "mem" /tmp/free.txt | tr -s " " | cut -d " " -f2)
# 获得内存使用的量
memory_use=$(grep -i "mem" /tmp/free.txt | tr -s " " | cut -d " " -f3)
# 获得内存空闲的量
memory_free=$(grep -i "mem" /tmp/free.txt | tr -s " " | cut -d " " -f4)

# 定制使用比例
# 获取内存使用率
percentage_use=$(echo "scale=2; $memory_use * 100 / $memory_totle" | bc)
# 定制内存空闲率
percentage_free=$(echo "scale=2; $memory_free * 100 / $memory_totle" | bc)
```

```
# 内容信息输出
echo -e "\e[31m\t${hostname} 内存使用信息统计\e[0m"
echo -e "\e[32m=========================================="
echo '内存总量:      ' ${memory_totle}
echo '内存使用量:    ' ${memory_use}
echo '内存空闲量:    ' ${memory_free}
echo '内存使用比率: ' ${percentage_use}
echo '内存空闲利率: ' ${percentage_free}
echo "=========================================="
echo -e "\e[0m"
```

```
脚本执行后效果
[root@rocky9 ~]# /bin/bash memory_info.sh
        localhost 内存使用信息统计
==========================================
内存总量:       3770
内存使用量:     238
内存空闲量:     3376
内存使用比率:  6.31
内存空闲利率:  89.54
==========================================
```

```
```

4.2 表达式

4.2.1 基础知识

基础知识

简介

```
    所谓的表达式，就是在场景需求的前提下，判断数据和运算符的操作是否满足需求。
```

语法格式:

```
格式
    真实值 操作符 真实值 比较运算符 预期值
示例
    3 + 4 > 6
要点:
    表达式应该具有判断的功能
```

测试表达式

简介

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

测试表达式

```
样式1: test 条件表达式
样式2: [ 条件表达式 ]
注意:
    以上两种方法的作用完全一样，后者为常用。
    但后者需要注意方括号[、]与条件表达式之间至少有一个空格。
    test跟 [] 的意思一样
        条件成立，状态返回值是0
        条件不成立，状态返回值是1
```

简单示例

```
test语法示例
[root@rocky9 ~]# test 1 == 1
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# test 1 == 2
[root@rocky9 ~]# echo $?
1

test -v语法测试变量有没有被设置值，亦可理解为变量是否为空。
[root@rocky9 ~]# echo $empty

[root@rocky9 ~]# test -v empty
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# empty=value
[root@rocky9 ~]# echo $?
0
```

```
[] 语法示例
[root@rocky9 ~]# [ 1 == 1 ]
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# [ 1 == 12 ]
[root@rocky9 ~]# echo $?
1
```

```
[] 格式解读
[root@rocky9 ~]# [ 1 == 12]
bash: [: 缺少 `]'
[root@rocky9 ~]# [ 1 == 12] ]
[root@rocky9 ~]# echo $?
1
```

```
```

4.2.2 逻辑表达式

基础知识

简介

```
    逻辑表达式一般用于判断多个条件之间的依赖关系。
    常见的逻辑表达式有：&&  和  ||，根据观察的角度不同含义也不同
```

语法解读

```
&&
    示例：命令1  &&  命令2
        如果命令1执行成功，那么我才执行命令2      -- 夫唱妇随
        如果命令1执行失败，那么命令2也不执行
```

```
||
    示例：命令1 || 命令2
        如果命令1执行成功，那么命令2不执行          -- 对着干
        如果命令1执行失败，那么命令2执行
```

```
!
    示例：！ 命令
        如果命令执行成功，则整体取反状态
```

组合使用

```
使用样式：
    命令1 && 命令2 || 命令3
        方便理解的样式  ( 命令1 && 命令2 ) || 命令3
功能解读：
    命令1执行成功的情况下，执行命令2
    命令2执行失败的情况下，执行命令3
注意：
    && 必须放到前面，|| 放到后面
```

简单实践

实践1-语法实践

```
&& 语法实践
[root@rocky9 ~]# [ 1 = 1 ] && echo "条件成立"
条件成立
[root@rocky9 ~]# [ 1 = 2 ] && echo "条件成立"

|| 语法实践
[root@rocky9 ~]# [ 1 = 2 ] || echo "条件不成立"
条件不成立
[root@rocky9 ~]# [ 1 = 1 ] || echo "条件不成立"
[root@rocky9 ~]#
```

实践2-案例实践

```
执行文件前保证具备执行权限
[root@rocky9 ~]# cat test_argnum.sh
#!/bin/bash
# && 和 || 演示

# 设定普通变量
arg_num=$#

[ $# == 1 ] && echo "脚本参数为1，允许执行脚本"
[ $# == 1 ] || echo "脚本参数不为1，不允许执行脚本"
```

```
脚本执行后效果
[root@rocky9 ~]# /bin/bash test_argnum.sh
脚本参数不为1，不允许执行脚本
[root@rocky9 ~]# /bin/bash test_argnum.sh 1
脚本参数为1，允许执行脚本
[root@rocky9 ~]# /bin/bash test_argnum.sh 1 2
脚本参数不为1，不允许执行脚本
```

实践3-取反

```
查看正常的字符串判断
[root@rocky9 ~]# [ aaa == aaa ]
[root@rocky9 ~]# echo $?
0

查看取反的效果判断
[root@rocky9 ~]# [ ! aaa == aaa ]
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# [ ! aaa == bbb ]
[root@rocky9 ~]# echo $?
0
```

实践4 - 组合使用

```
[root@rocky9 ~]# [ -d /etc ] && echo "目录存在" || echo "目录不存在"
目录存在
[root@rocky9 ~]# [ -d /etc1 ] && echo "目录存在" || echo "目录不存在"
目录不存在
```

实践5 - 主机网络连通性测试

```
查看脚本内容
[root@rocky9 ~]# cat host_network_test.sh
#!/bin/bash
# 功能：测试主机网络连通性
# 版本：v0.1
# 作者：wangshusen
# 联系：www.sswang.com

# 定制普通变量
host_addr="$1"
```

```
# 脚本基本判断
[ -z ${host_addr} ] && echo "请输入待测试主机ip" && exit
[ $# -ne 1 ] && echo "请保证输入1个脚本参数" && exit

# 测试主机网络
net_status=$(ping -c1 -w1 ${host_addr} >/dev/null 2>&1 && echo "正常" || echo "异常")

# 信息输出
echo -e "\e[31m\t主机网络状态信息\e[0m"
echo -e "\e[32m================================="
echo "${host_addr} 网络状态: ${net_status}"
echo -e "=================================\e[0m"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash host_network_test.sh
请输入待测试主机ip
[root@rocky9 ~]# /bin/bash host_network_test.sh aa bb
请保证输入1个脚本参数
[root@rocky9 ~]# /bin/bash host_network_test.sh 10.0.0.12
        主机网络状态信息
=================================
10.0.0.12 网络状态: 正常
=================================
[root@rocky9 ~]# /bin/bash host_network_test.sh 10.0.0.15
        主机网络状态信息
=================================
10.0.0.15 网络状态: 异常
=================================
```

4.2.3 字符串表达式

基础知识

简介

```
    所谓的字符串表达式，主要是判断 比较运算符 两侧的值的内容是否一致，由于bash属于弱类型语言，所以，默认情况下，无论数字和字符，都会可以被当成字符串进行判断。
```

符号解读

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

内容空值判断
    -z  str                 空值判断，获取字符串长度，长度为0，返回True
    -n  "str"               非空值判断，获取字符串长度，长度不为0，返回True
                            注意：str外侧必须携带"",否则无法判断
```

简单实践

实践1-内容比较判断

```
判断字符串内容是否一致
[root@rocky9 ~]# test aaa == bbb
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# test aaa != bbb
[root@rocky9 ~]# echo $?
0

判断数字内容是否一致
[root@rocky9 ~]# num1=234 num2=456
[root@rocky9 ~]# test $num1 == $num2
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# test $num1 != $num2
[root@rocky9 ~]# echo $?
0
```

实践2-空值判断

```
判断内容是否为空
[root@rocky9 ~]# string=nihao
[root@rocky9 ~]# test -z $string
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# test -z $string1
[root@rocky9 ~]# echo $?
0

判断内容是否为不空，可以理解为变量是否被设置
[root@rocky9 ~]# unset str
[root@rocky9 ~]# [ -n $str ]
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# [ -n "$str" ]
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# str=value
[root@rocky9 ~]# [ -n "$str" ]
[root@rocky9 ~]# echo $?
0
```

实践3-脚本实践

```
查看脚本内容
[root@rocky9 ~]# cat simple_login_string.sh
#!/bin/bash
# 功能: 模拟shell登录
# 版本: v0.1
# 作者: wangshusen
# 联系: www.sswang.com

# 定制命令变量
OS_INFO=$(cat /etc/redhat-release)
KERNEL_INFO=$(uname -r)
OS_ARCH=$(uname -m)
HOSTNAME=$(hostname)
```

```
# 清屏
clear

# 输出提示信息
echo -e "\e[32m${OS_INFO} \e[0m"
echo -e "\e[32mKernel ${KERNEL_INFO} on an ${OS_ARCH} \e[0m"
echo "----------------------------------"
# 交互输入登陆信息
read -p "请输入用户名：" account
[ -z $account ] && read -p "请输入用户名：" account
read -s -t30 -p "请输入登录密码：" password
echo
echo "----------------------------------"
# 输出用户输入信息
[ $account == 'root' ] && [ $password == '123456' ] && echo "登录成功" || echo "登录失败"
```

```
脚本执行测试
[root@rocky9 ~]# /bin/bash simple_login_string.sh
CentOS Linux release 7.9.2009 (Core)
Kernel 3.10.0-1160.el7.x86_64 on an x86_64
----------------------------------
请输入用户名：root
请输入登录密码：
----------------------------------
登录成功
[root@rocky9 ~]# /bin/bash simple_login_string.sh
CentOS Linux release 7.9.2009 (Core)
Kernel 3.10.0-1160.el7.x86_64 on an x86_64
----------------------------------
请输入用户名：
请输入用户名：root1
请输入登录密码：
----------------------------------
登录失败
[root@rocky9 ~]#
```

4.2.4 文件表达式

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

> 所谓的文件表达式，主要是判断文件相关的权限和属性信息的。

表达式解读

```
文件属性判断
    -d  检查文件是否存在且为目录文件
    -f  检查文件是否存在且为普通文件
    -S  检查文件是否存在且为socket文件
```

```
    -L  检查文件是否存在且为链接文件
    -O  检查文件是否存在并且被当前用户拥有
    -G  检查文件是否存在并且默认组为当前用户组

文件权限判断
    -r  检查文件是否存在且可读
    -w  检查文件是否存在且可写
    -x  检查文件是否存在且可执行

文件存在判断
    -e  检查文件是否存在
    -s  检查文件是否存在且不为空

文件新旧判断
    file1 -nt file2  检查file1是否比file2新
    file1 -ot file2  检查file1是否比file2旧
    file1 -ef file2  检查file1是否与file2是同一个文件，判定依据的是i节点
```

简单实践

实践1- 文件属性判断

```
[root@rocky9 ~]# [ -f weizhi.sh ] && echo "是一个文件"
[root@rocky9 ~]# [ -f weizhi.sddh ] || echo "不是一个文件"
不是一个文件
[root@rocky9 ~]# [ -d weizhi.sddh ] || echo "不是一个目录"
不是一个目录
[root@rocky9 ~]# [ -d /tmp ] && echo "是一个目录"
是一个目录
```

实践2-文件权限判断

```
[root@rocky9 ~]# [ -x memory_info.sh ] || echo "文件没有执行权限"
文件没有执行权限
[root@rocky9 ~]# [ -x memory_info.sh ] || chmod +x memory_info.sh
[root@rocky9 ~]# [ -x memory_info.sh ] && ./memory_info.sh
        localhost 内存使用信息统计
==========================================
内存总量:       3770
内存使用量:     242
内存空闲量:     3372
内存使用比率:   6.41
内存空闲利率:   89.44
==========================================
```

实践3-文件存在判断

```
文件内容空值判断
[root@rocky9 ~]# touch nihao.txt
[root@rocky9 ~]# [ -s nihao.txt ] || echo "文件为空"
文件为空
[root@rocky9 ~]# echo nihao > file.txt
[root@rocky9 ~]# [ -s file.txt ] && echo "文件不为空"
文件不为空
```

```
文件存在与否判断
[root@rocky9 ~]# [ -e file.txt ] && echo "文件存在"
文件存在
[root@rocky9 ~]# [ -e file.txt1 ] || echo "文件不存在"
文件不存在
```

4.2.5 数字表达式

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

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

语法解读

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

简单实践

实践1-命令实践

```
[root@rocky9 ~]# [ 3 -gt 2 ] && echo "3 大于 2"
3 大于 2
[root@rocky9 ~]# [ 3 -ne 2 ] && echo "3 不等于 2"
3 不等于 2
[root@rocky9 ~]# [ 3 -eq 3 ] && echo "3 等于 3"
3 等于 3
```

实践2-脚本安全

```
查看脚本内容
[root@rocky9 ~]# cat test_argnum.sh
#!/bin/bash
# -eq 和 -ne 演示

# 设定普通变量
arg_num=$#

[ $arg_num -eq 1 ] && echo "脚本参数为1，允许执行脚本"
[ $arg_num -ne 1 ] && echo "脚本参数不为1，不允许执行脚本"
```

```
脚本执行效果
root@rocky9 ~]# /bin/bash test_argnum.sh
脚本参数不为1，不允许执行脚本
[root@rocky9 ~]# /bin/bash test_argnum.sh 1
脚本参数为1，允许执行脚本
[root@rocky9 ~]# /bin/bash test_argnum.sh 1 2
脚本参数不为1，不允许执行脚本
```

4.3 表达式进阶

4.3.1 [[]] 测试进阶

基础知识

简介

```
    我们之前学习过 test 和 [ ] 测试表达式，这些简单的测试表达式，仅仅支持单条件的测试。如果需要针对多条件测试场景的话，我们就需要学习[[  ]] 测试表达式了。
    我们可以将 [[  ]] 理解为增强版的 [ ]，它不仅仅支持多表达式，还支持扩展正则表达式和通配符。
```

语法解析

```
基本格式：
    [[ 源内容 操作符 匹配内容 ]]

操作符解析：
    == 左侧源内容可以被右侧表达式精确匹配
    =~ 左侧源内容可以被右侧表达式模糊匹配
```

简单实践

实践1-内容的基本匹配

```
定制默认的的变量
[root@rocky9 ~]# string=value
[root@rocky9 ~]# echo $string
value
[root@rocky9 ~]# [[ $string == value ]]
[root@rocky9 ~]# echo $?
0

[[]] 支持通配符
[root@rocky9 ~]# [[ $string == v* ]]
[root@rocky9 ~]# echo $?
0

使用""取消正则，则内容匹配失败
[root@rocky9 ~]# [[ $string == v"*" ]]
[root@rocky9 ~]# echo $?
1

使用\取消正则，则内容匹配失败
[root@rocky9 ~]# [[ $string == v\* ]]
[root@rocky9 ~]# echo $?
1
```

实践2-文件的匹配

```
定制文件名称
[root@rocky9 ~]# script_name=file.sh
[root@rocky9 ~]# [[ $script_name == *.sh ]]
[root@rocky9 ~]# echo $?
0

尝试其他匹配
[root@rocky9 ~]# [[ $script_name == *.txt ]]
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# [[ $script_name != *.txt ]]
[root@rocky9 ~]# echo $?
0
```

4.3.2 集合基础

基础知识

简介

```
    所谓的集合，主要是针对多个条件表达式组合后的结果，尤其是针对于逻辑场景的组合。初中数学的相关逻辑示意图：
```

A条件	B条件	非A [! A]	A 或 B [A | B]	A 与 B [A & B]
真	真	假	真	真
真	假	假	真	假
假	真	真	真	假
假	假	真	假	假

表现样式

```
两个条件
    1 - 真        0 - 假
三种情况：
    与 - &        或 - |        非 - !

注意：
    这里的 0 和 1 ，千万不要与条件表达式的状态值混淆
```

```
与关系：
    0 与 0 = 0        0 & 0 = 0
    0 与 1 = 0        0 & 1 = 0
    1 与 0 = 0        1 & 0 = 0
    1 与 1 = 1        1 & 1 = 1
```

```
或关系：
    0 或 0 = 0        0 | 0 = 0
    0 或 1 = 1        0 | 1 = 1
    1 或 0 = 1        1 | 0 = 1
    1 或 1 = 1        1 | 1 = 1
```

```
非关系:
    非 1 = 0          ! true = false
    非 0 = 1          ! false = true
```

简单实践

实践1- 简单判断

```
或实践
[root@rocky9 ~]# echo $[ 0 | 1 ]
1
[root@rocky9 ~]# echo $[ 0 | 0 ]
0
[root@rocky9 ~]# echo $[ 1 | 0 ]
1
[root@rocky9 ~]# echo $[ 1 | 1 ]
1
```

```
与实践
[root@rocky9 ~]# echo $[ 1 & 1 ]
1
[root@rocky9 ~]# echo $[ 1 & 0 ]
0
[root@rocky9 ~]# echo $[ 0 & 1 ]
0
[root@rocky9 ~]# echo $[ 0 & 0 ]
0
```

```
非实践
[root@rocky9 ~]# true
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# false
[root@rocky9 ~]# echo $?
1

[root@rocky9 ~]# echo $[ ! 0 ]
1
[root@rocky9 ~]# echo $[ ! 1 ]
0
```

4.3.3 逻辑组合

基础知识

简介

```
所谓的条件组合，指的是在同一个场景下的多个条件的综合判断效果。
```

语法解析

```
方法1：
    [ 条件1 -a 条件2 ]      - 两个条件都为真，整体为真，否则为假
    [ 条件1 -o 条件2 ]      - 两个条件都为假，整体为假，否则为真
方法2：
    [[ 条件1 && 条件2 ]]    - 两个条件都为真，整体为真，否则为假
    [[ 条件1 || 条件2 ]]    - 两个条件都为假，整体为假，否则为真
```

简单实践

实践1-[]组合实践

```
[root@rocky9 ~]# user=root pass=123456
[root@rocky9 ~]# [ $user == "root" -a $pass == "123456" ]
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# [ $user == "root" -a $pass == "1234567" ]
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# [ $user == "root" -o $pass == "1234567" ]
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# [ $user == "root1" -o $pass == "1234567" ]
[root@rocky9 ~]# echo $?
1
```

实践2 - [[]]组合实践

```
[root@rocky9 ~]# [[ $user == "root" && $pass == "123456" ]]
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# [[ $user == "root" && $pass == "1234567" ]]
[root@rocky9 ~]# echo $?
1
[root@rocky9 ~]# [[ $user == "root" || $pass == "1234567" ]]
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# [[ $user == "root1" || $pass == "1234567" ]]
[root@rocky9 ~]# echo $?
1
```

4.3.4 综合实践

堡垒机登录

脚本功能-扩充用户名和密码验证功能

```
[root@rocky9 ~]# cat simple_jumpserver.sh
#!/bin/bash
# 功能：定制堡垒机的展示页面
# 版本：v0.3
# 作者：wangshusen
# 联系：www.sswang.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
[[ ${user} == ${login_user} && ${password} == ${login_pass} ]] && echo "主机登录验证成功" || echo "您输入的用户名或密码有误"
```

```
脚本执行效果
[root@rocky9 ~]# /bin/bash simple_jumpserver.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.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
```

信息检测

脚本功能-检测公司网站的存活

判断网站的命令

```
[root@rocky9 ~]# wget --spider -T5 -q -t2 www.baidu.com
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# wget --spider -T5 -q -t2 www.baidu.com1
[root@rocky9 ~]# echo $?
4
[root@rocky9 ~]# curl -s -o /dev/null www.baidu.com
[root@rocky9 ~]# echo $?
0
[root@rocky9 ~]# curl -s -o /dev/null www.baidu.com1
[root@rocky9 ~]# echo $?
6
```

脚本的核心内容

```
[root@rocky9 ~]# cat site_healthcheck.sh
#!/bin/bash
# 功能：定制站点的检测功能
# 版本：v0.1
# 作者：wangshusen
# 联系：www.sswang.com

# 定制普通变量
site_addr="$1"
# 脚本基本判断
[ -z ${site_addr} ] && echo "请输入待测试站点域名" && exit
[ $# -ne 1 ] && echo "请保证输入1个脚本参数" && exit

# 检测平台的信息提示
echo -e "\e[32m-----------检测平台支持的检测类型-----------
 1: wget
 2: curl
----------------------------------------"'\033[0m'

# 选择检测类型
read -p "请输入网站的检测方法: " check_type
site_status=$([ ${check_type} == 1 ] && wget --spider -T5 -q -t2 ${site_addr} &&
echo "正常" || echo "异常")
site_status=$([ ${check_type} == 2 ] && curl -s -o /dev/null ${site_addr} &&
echo "正常" || echo "异常")
# 信息输出
echo
echo -e "\e[31m\t站点状态信息\e[0m"
echo -e "\e[32m================================="
echo "${site_addr} 站点状态: ${site_status}"
echo -e "=================================\e[0m"
```

脚本执行效果

```
[root@rocky9 ~]# /bin/bash site_healthcheck.sh
请输入待测试站点域名
[root@rocky9 ~]# /bin/bash site_healthcheck.sh aa bb
请保证输入1个脚本参数
[root@rocky9 ~]# /bin/bash site_healthcheck.sh www.baidu.com
-----------检测平台支持的检测类型-----------
 1: wget
 2: curl
```

```
----------------------------------------
请输入网站的检测方法: 1

        站点状态信息
================================
www.baidu.com 站点状态: 异常
================================
[root@rocky9 ~]# /bin/bash site_healthcheck.sh www.baidu.com1
-----------检测平台支持的检测类型-----------
 1: wget
 2: curl
----------------------------------------
请输入网站的检测方法: 2

        站点状态信息
================================
www.baidu.com1 站点状态: 异常
================================
```