SHELL自动化运维

作者：itcast

版本：V 1.0

文档编号：

日期：2017年8月18日

内部架构文档

目录

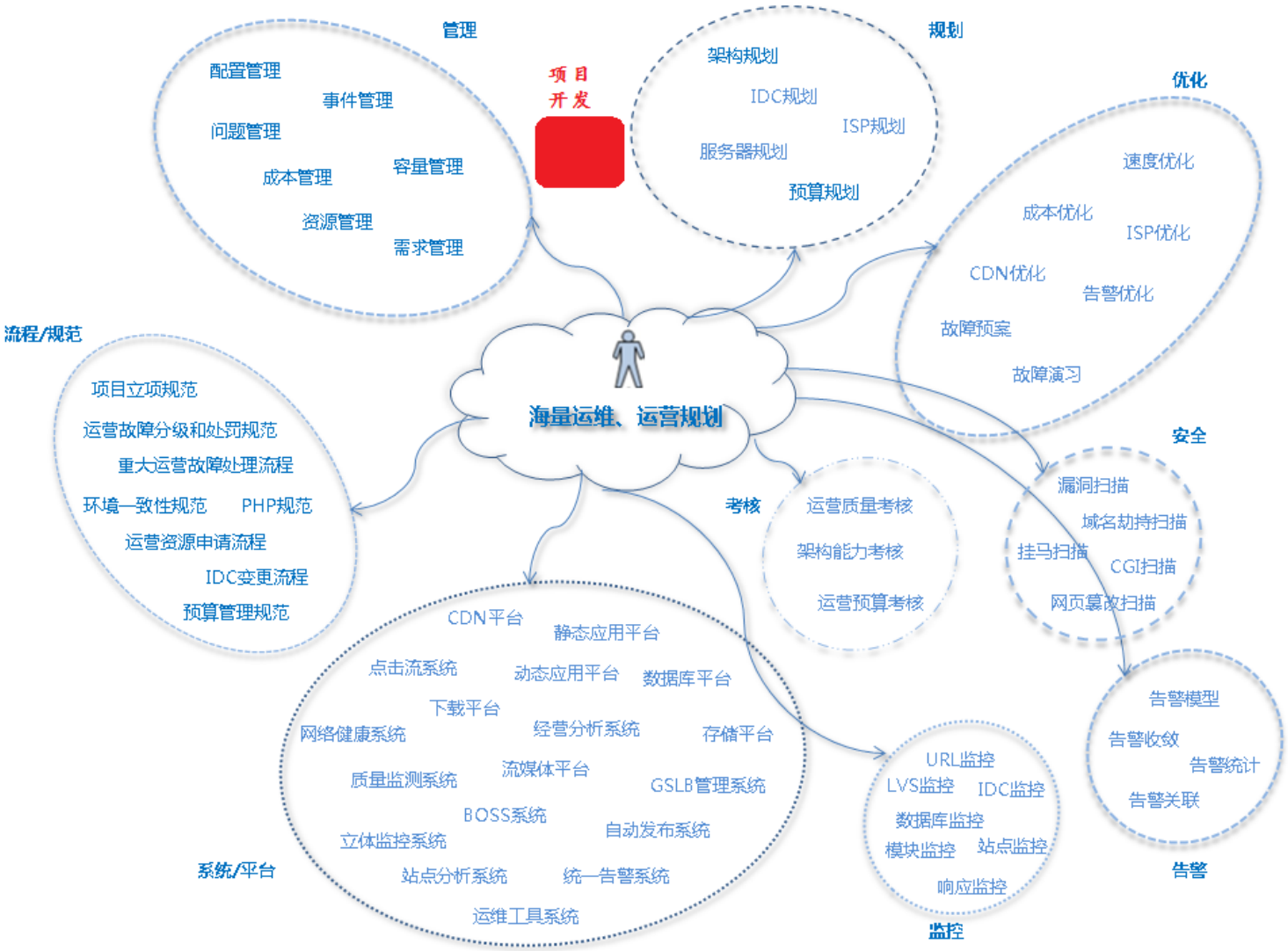
第 1 章 SHELL 快速入门

1.1 运维

运维是什么？

公司的技术岗位

运维的工作范围



以天天生鲜项目为例：

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

注意：

shell脚本就是shell编程的一种具体实现

1.2 shell简介

既然我们是来学shell，首先第一个问题：shell是什么？

1.2.1 什么是shell

shell的定义

在计算机科学中，Shell就是一个**命令解释器。**

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

shell位置图

总结：

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的分类

基本上shell分两大类：

图形界面shell和**命令行shell**

图形界面shell

图形界面shell就是我们常说的桌面

命令行式shell

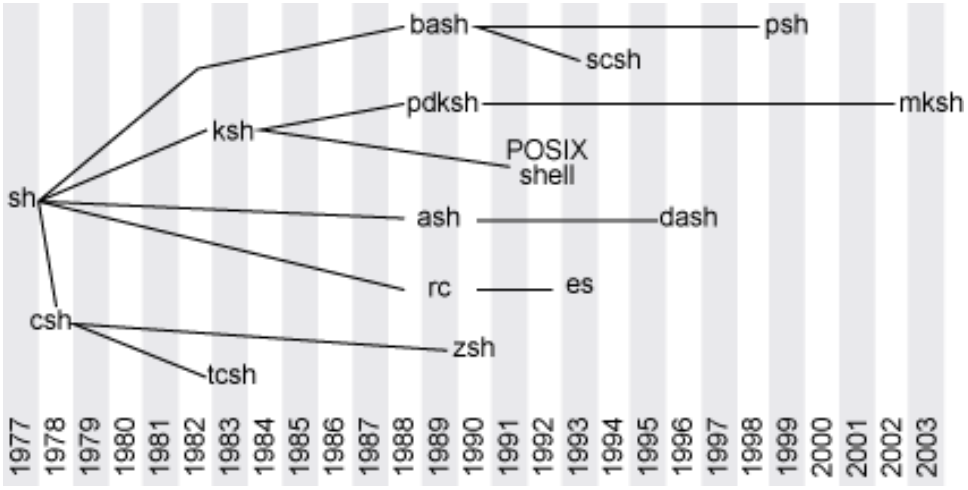
windows系统：

cmd.exe 命令提示字符

linux系统：

sh / csh / ksh / **bash** / ...

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



查看系统shell信息

查看当前系统的shell类型

```
echo $SHELL
```

查看当前系统环境支持的shell

```
[root@linux-node1 ~]# cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

1.2.3 shell 脚本

shell使用方式

手工方式：

手工敲击键盘,在shell的命令行输入命令,按Enter后,执行通过键盘输入的命令,然后shell返回并显示命令执行的结果.

重点：**逐行输入命令、逐行进行确认执行**

脚本方式：

就是说我们把手工执行的命令a，写到一个脚本文件b中，然后通过执行脚本b，达到执行命令a的效果.

shell脚本定义

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。

shell脚本示例

现在我们来使用脚本的方式来执行以下

```
[root@linux-node1 ~]# vim itcast.sh
#!/bin/bash
# 这是临时 shell 脚本
echo 'nihao'
echo 'itcast'
```

脚本执行效果

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

第 2 章 SHELL 基础知识

2.1 shell脚本

我们在上面简单介绍了一下什么是shell脚本，现在我们来进一步的来介绍shell脚本的要求/格式/规范等内容

2.1.1 创建脚本

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

2.1.2 脚本执行

shell执行的方式

Shell脚本的执行通常可以采用以下几种方式

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

执行说明：

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

2、脚本文件具有可执行权限时使用。
使用频率：☆☆☆☆

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

2.1.2 脚本开发规范

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

2.2 变量

变量的学习我们主要从四个方面来学习：

变量的定义和分类

本地变量

全局变量

shell内置变量

2.2.1 什么是变量

变量是什么？

变量包括**两部分**：

变量名　　不变的

变量值　　变化的

我们一般所说的变量指的是：变量名

2.2.2 本地变量

本地变量是什么？

本地变量就是：在当前系统的某个环境下才能生效的变量，作用范围小。

本地变量包含两种：普通变量和命令变量

普通变量：

普通变量的定义方式有如下三种，接下来我们就分别说一下这三种方式：

方式一：

变量名=变量值

重点：

变量值必须是一个整体，中间没有特殊字符

方式二：

变量名='变量值'

重点：

我看到的内容，我就输出什么内容

方式三：

变量名="变量值"

重点：

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

习惯：

数字不加引号，其他默认加双引号

命令变量(熟练)

命令变量有两种定义方式，接下来我们就来介绍一下这两种方式

定义方式一：

变量名=`命令`

注意：

` 是反引号

定义方式二：

变量名=$(命令)

执行流程：

1、执行`或者$()范围内的命令

2、将命令执行后的结果，赋值给新的变量名A

2.2.3 全局变量

全局变量是什么

全局变量就是：当前shell以及其派生出来的子shell中都有效的变量。

查看全局变量命令

可以通过命令查看全局变量

env 只显示全局变量

定义全局变量

方法一：

变量=值

export 变量

方法二：（最常用）

export 变量=值

2.2.4 变量查看和取消

查看变量：

方式一：

$变量名

场景：

私下里，在命令行/脚本中使用

图省事

方式二：

"$变量名"

场景：

私下里，在命令行/脚本中使用

图省事

方式三：

${变量名}

场景：

echo " hello ${变量名} world "

使用频率较高

方式四：

"${变量名}"

场景：

标准使用方式

取消变量

unset 变量名

2.2.5 shell内置变量

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

和脚本文件有关

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

重点内置变量演示效果：

$0　　获取脚本的名称

示例：

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

$#　　获取当前脚本传入参数的数量

示例：

```
# cat num.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： $#"
```

$n　　获取当前脚本传入的第n个位置的参数

示例：

```
#!/bin/bash
# 获取指定位置的参数
echo "第一个位置的参数是： $1"
echo "第二个位置的参数是： $2"
echo "第三个位置的参数是： $3"
echo "第四个位置的参数是： $4"
```

$?　　获取文件执行或者命令执行的返回状态值

示例：

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

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

```
0
```

字符串精确截取

格式：${变量名:起始位置:截取长度}

示例：

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

默认值相关

场景一：

变量a如果有内容，那么就输出a的变量值

变量a如果没有内容，那么就输出默认的内容

格式：

```
${变量名:-默认值}
```

套餐示例：

如果我输入的参数为空，那么输出内容是 "您选择的套餐是： 套餐 1"

如果我输入的参数为n，那么输出内容是 "您选择的套餐是： 套餐 n"

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

场景二：

无论变量a是否有内容，都输出默认值

格式：

```
${变量名+默认值}
```

场景示例：

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

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

第 3 章 SHELL 进阶

这部分的知识，我们主要是从三个方面来学习：

测试语句

表达式(条件+计算)

linux常见符号和命令

3.1 表达式

要使Shell脚本程序具备一定的“逻辑能力”，面临的第一个问题就是：区分不同的情况以确定执行何种操作，下面我们就来学习解决这个问题---测试语句

3.1.1 测试语句

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）

测试语句形式

A: test 条件表达式

B: [条件表达式]

格式注意：

以上两种方法的作用完全一样，后者为常用。

但后者需要注意方括号[、]与条件表达式之间至少有一个空格。

test跟 [] 的意思一样

条件成立，状态返回值是0

条件不成立，状态返回值是1

操作注意：

[]两侧要有空格，表达式中的符号左右要有空格

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

3.1.2 条件表达式

我们这部分内容主要是介绍，测试语句中的 [条件表达式] 这一部分，测试的结果使用 echo $? 来查看

逻辑表达式

逻辑表达式一般用于判断多个条件之间的依赖关系。

常见的逻辑表达式有： && 和 ||

&&

命令1 && 命令2

如果命令1执行成功，那么我才执行命令2 -- 夫唱妇随

如果命令1执行失败，那么命令2也不执行

示例：

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

||

命令1 || 命令2

如果命令1执行成功，那么命令2不执行 -- 对着干

如果命令1执行失败，那么命令2执行

示例：

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

文件表达式

-f 判断输入内容是否是一个文件

示例：

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

-d 判断输入内容是否是一个目录

示例：

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

-x 判断输入内容是否可执行

示例：

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

数值操作符

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

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

字符串比较

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

实践

判断字符是否内容一致

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

```
root@ubuntu:~# echo $?
1
```

3.1.3 计算表达式

定义：

计算表达式，简单来说就是对具体的内容进行算数计算

格式：

方式一：

`$(())`　　`$(( 计算表达式 ))`

方式二：

`let`　　`let 计算表达式`

注意：

$(())中只能用+-*/和()运算符，并且只能做整数运算

$(())演示效果

格式：a=$((变量名a+1))

注意：

表达式范围内，空格不限制

计算示例

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

let演示效果

格式：let　变量名a=变量名a+1

注意：

表达式必须是一个整体，中间不能出现空格等特殊字符

let示例

```
root@ubuntu:~# i=1
root@ubuntu:~# let i=i+7
root@ubuntu:~# echo $i
8
```

3.2 linux常见符号

接下来我们来介绍几个linux的场景符号：

重定向符号、管道符、其他符号

3.2.1 重定向符号

在shell脚本中有两种常见的重定向符号 > 和 >>

> 符号

作用：

> 表示将符号左侧的内容，以覆盖的方式输入到右侧文件中

演示：

查看文件内容

```
admin-1@ubuntu:~$ cat file.txt
nihao
```

使用重定向符号给文件中增加内容

```
admin-1@ubuntu:~$ echo "file1.txt" > file.txt
```

再次查看文件内容

```
admin-1@ubuntu:~$ cat file.txt
file1.txt
```

>> 符号

作用：

>> 表示将符号左侧的内容，以**追加**的方式输入到右侧文件的末尾行中

演示：

查看文件内容

```
admin-1@ubuntu:~$ cat file.txt
file1.txt
```

使用重定向符号给文件中增加内容

```
admin-1@ubuntu:~$ echo "file2.txt" >> file.txt
```

再次查看文件内容

```
admin-1@ubuntu:~$ cat file.txt
file1.txt
file2.txt
```

3.2.2 管道符 |

定义：

| 这个就是管道符，传递信息使用的

使用格式：

命令1 | 命令2

管道符左侧命令1 执行后的结果，**传递**给管道符右侧的命令2使用

命令演示：

查看当前系统中的全局变量SHELL

```
admin-1@ubuntu:~$ env | grep SHELL
SHELL=/bin/bash
```

3.2.3 其他符号

后台展示符号 &

定义：

& 就是将一个命令从前台转到后台执行

使用格式：

命令 &

命令演示：

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

全部信息符号 2>&1

符号详解：

1 表示正确输出的信息

2 表示错误输出的信息

2>&1 代表所有输出的信息

符号示例

标准正确输出示例

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

标准错误输出示例

```
dsfadsfadsfa 2>> errfile
```

综合演练示例

脚本内容

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

脚本执行效果

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

1 和 2 综合演练

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

全部信息演练

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

linux系统垃圾桶

/dev/null 是linux下的一个设备文件，

这个文件类似于一个垃圾桶，特点是：容量无限大

3.3 常见命令详解

接下来我们介绍一些shell脚本中经常使用的linux命令：grep、sed、awk、find

3.3.1 grep命令详解

grep命令是我们常用的一个强大的文本搜索命令。

命令格式详解

grep [参数] <文件名>

注意：

我们在查看某个文件的内容的时候，是需要有<文件名>

grep命令在结合|(管道符)使用的情况下，后面的<文件名>是没有的

可以通过 grep --help 查看grep的帮助信息

参数详解

-c：只输出匹配行的计数。

-n：显示匹配行及行号。
-v：显示不包含匹配文本的所有行。

模板文件

```
admin-1@ubuntu:~$ cat find.txt
nihao aaa
nihao AAA
NiHao bbb
nihao CCC
```

-c:　输出匹配到aaa的个数

```
admin-1@ubuntu:~$ grep -c aaa find.txt
1
```

-n:　输出匹配内容，同时显示行号

```
admin-1@ubuntu:~$ grep -n CCC find.txt
4:nihao CCC
```

-v:　匹配到的内容部输出，输出不匹配的内容

```
admin-1@ubuntu:~$ grep -v ni find.txt
NiHao bbb
```

3.3.2 sed命令详解

sed 行文件编辑工具。因为它编辑文件是以行为单位的。

命令格式详解

命令格式：

sed [参数] '<匹配条件>[动作]' [文件名]

注意：

可以通过 sed --help 查看grep的帮助信息

参数详解：

参数为空　表示sed的操作效果，实际上不对文件进行编辑
-i　表示对文件进行编辑

匹配条件：

匹配条件分为两种：数字行号或者关键字匹配
关键字匹配格式：

'/关键字/'

注意：

隔离符号 / 可以更换成 @、#、！等符号
根据情况使用，如果关键字和隔离符号有冲突，就更换成其他的符号即可。

动作详解

-a　在匹配到的内容下一行增加内容
-i　在匹配到的内容上一行增加内容
-d　删除匹配到的内容
-s　替换匹配到的内容

注意：

上面的动作应该在参数为-i的时候使用，不然的话不会有效果

替换命令演示

关于替换，我们从三个方面来学习：

行号、列号、全体

命令格式：

sed -i [替换格式] [文件名]

注意：替换命令的写法

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

常见替换格式：

模板文件内容

```
admin-1@ubuntu:~$ cat sed.txt
nihao sed sed sed
nihao sed sed sed
nihao sed sed sed
```

替换**每行首个**匹配内容：

sed -i 's#原内容#替换后内容#' 文件名

示例：替换首每行的第1个sed为SED

```
admin-1@ubuntu:~$ sed -i 's#sed#SED#' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED sed sed
nihao SED sed sed
nihao SED sed sed
```

替换**全部**匹配内容：

sed -i 's#原内容#替换后内容#**g**' 文件名

示例：替换全部sed为des

```
admin-1@ubuntu:~$ sed -i 's#sed#SED#g' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED SED SED
nihao SED SED SED
nihao SED SED SED
```

指定**行号**替换**首个**匹配内容：

sed -i '**行号**s#原内容#替换后内容#' 文件名

示例：替换第2行的首个SED为sed

```
admin-1@ubuntu:~$ sed -i '2s#SED#sed#' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED SED SED
nihao sed SED SED
nihao SED SED SED
```

首行指定**列号**替换匹配内容：

sed -i 's#原内容#替换后内容#**列号**' 文件名

示例：替换每行的第2个SED为sed

```
admin-1@ubuntu:~$ sed -i 's#SED#sed#2' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED sed SED
```

```
nihao sed sed SED
nihao SED sed SED
```

指定行号列号匹配内容：

sed -i '行号s#原内容#替换后内容#列号' 文件名

示例：替换第3行的第2个SED为sed

```
admin-1@ubuntu:~$ sed -i '3s#SED#sed#2' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED sed SED
nihao sed sed SED
nihao SED sed sed
```

增加操作

作用：

在指定行号的下一行增加内容

格式：

sed -i '行号a\增加的内容' 文件名

注意：

如果增加多行，可以在行号位置写个范围值，彼此间使用逗号隔开，例如

sed -i '1,3a\增加内容' 文件名

演示效果：

指定行号下一行增加内容

```
admin-1@ubuntu:~$ sed -i '2a\zengjia-2' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED sed SED
nihao sed SED sed
zengjia-2
nihao SED sed sed
```

指定1~3每行都增加内容

```
admin-1@ubuntu:~$ sed -i '1,3a\tongshi-2' sed.txt
admin-1@ubuntu:~$ cat sed.txt
nihao SED sed SED
tongshi-2
nihao sed SED sed
tongshi-2
zengjia-2
tongshi-2
nihao SED sed sed
```

作用：

在指定行号的上一行增加内容

格式：

sed -i '行号i\增加的内容' 文件名

注意：

如果增加多行，可以在行号位置写个范围值，彼此间使用逗号隔开，例如

sed -i '1,3a\增加内容' 文件名

演示效果：

指定行号上一行增加内容

```
admin-1@ubuntu:~$ sed -i '1i\insert-1' sed.txt
admin-1@ubuntu:~$ cat sed.txt
insert-1
nihao SED sed SED
tongshi-2
nihao sed SED sed
tongshi-2
zengjia-2
tongshi-2
nihao SED sed sed
```

删除操作

作用：

指定行号删除

格式：

sed -i '行号**d**' 文件名

注意：

如果删除多行，可以在行号位置多写几个行号，彼此间使用**逗号**隔开，例如

sed -i '1,3d' 文件名

删除演练

删除第4行内容

```
admin-1@ubuntu:~$ sed -i '4d' sed.txt
admin-1@ubuntu:~$ cat sed.txt
insert-1
nihao SED sed SED
tongshi-2
tongshi-2
zengjia-2
tongshi-2
nihao SED sed sed
```

删除多行(3-5行)内容

```
admin-1@ubuntu:~$ sed -i '3,5d' sed.txt
admin-1@ubuntu:~$ cat sed.txt
insert-1
nihao SED sed SED
tongshi-2
nihao SED sed sed
```

3.3.3 awk命令详解

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

命令格式:

awk [参数] '[动作]' [文件名]

常见参数：

-F　　指定行的分隔符

常见动作：

print　显示内容

　　$0　　显示当前行所有内容

　　$n　　显示当前行的第n列内容

常见内置变量

FILENAME　当前输入文件的文件名，该变量是只读的

NR　　指定显示行的行号

NF　　输出最后一列的内容

OFS　　输出格式的列分隔符，缺省是空格

FS　　输入文件的列分融符，缺省是连续的空格和Tab

命令演示

模板文件内容

```
admin-1@ubuntu:~$ cat awk.txt
nihao awk awk awk
nihao awk awk awk
```

打印指定列内容

打印第1列的内容

```
admin-1@ubuntu:~$ awk '{print $1}' awk.txt
nihao
nihao
```

指定隔离分隔符，查看内容

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

设置显示分隔符，显示内容

```
admin-1@ubuntu:~$ awk 'BEGIN{OFS=":"} {print NR,$0}' awk.txt
1:nihao awk awk awk
2:nihao awk awk awk
```

3.3.4 find命令详解

命令格式：

find [路径] [参数] [关键字]

参数详解

-name 按照文件名查找文件。

-type 查找某一类型的文件，

诸如：

b - 块设备文件　　d - 目录　　c - 字符设备文件

p - 管道文件　　l - 符号链接文件　　f - 普通文件。

命令演示

在当前系统中查找一个叫awk的文件

```
admin-1@ubuntu:~$ sudo find /home/admin-1/ -name "awk.txt"
/home/admin-1/awk.txt
```

在当前系统中查找文件类型为普通文件的文件

```
admin-1@ubuntu:~$ find /tmp -type f
/tmp/.X0-lock
/tmp/vgauthsvclog.txt.0
/tmp/unity_support_test.0
/tmp/config-err-4igbXW
```

第4章 流程控制

在shell的语句中，流程控制主要分为两种：

简单流程控制语句：选择和循环

复杂流程控制语句：函数

4.1 简单流程控制语句

4.1.1 单分支if语句

语法格式

```
if [ 条件 ]
then
    指令
fi
```

场景：

单一条件，只有一个输出

单分支if语句示例

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

4.1.2 双分支if语句

语法格式

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

场景：

一个条件，两种结果

双分支if语句示例

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

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

4.1.3 多分支if语句

语法格式

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

场景：

n个条件，n+1个结果

多分支if语句示例

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

多if语句生产场景： 服务的启动

需求：

要求脚本执行需要有参数，通过传入参数来实现不同的功能。

参数和功能详情如下：

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

脚本内容

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

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

4.1.4 case选择语句

我们发现多if语句使用的时候，代码量很多，而且整体看起来确实有那么一丁点乱，有没有办法更好的实现这种效果呢？就是Case语句。

case 语句格式

```
case 变量名 in
   值 1)
      指令 1
        ;;
   值 2)
      指令 2
        ;;
   值 3)
       指令 3
        ;;
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;

case语句示例

场景：在多if语句的基础上对脚本进行升级

需求：

要求脚本执行需要有参数，通过传入参数来实现不同的功能。

参数和功能详情如下：

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

脚本内容：

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

4.1.5 for循环语句

循环指定的所有元素，循环完毕之后就推出

语法格式

语法格式

```
for 值 in 列表
do
  执行语句
done
```

场景：

遍历列表

注意：

”for” 循环总是接收 “in” 语句之后的某种类型的字列表

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

对于固定次数的循环，可以通过seq命令来实现，就不需要变量的自增了

示例：遍历文件

```
#!/bin/bash
# for 语句的使用示例
file=`ls /data/scripts/python10`
for i in "${file}"
do
  echo "${i}"
done
```

示例2：遍历数字

```
#!/bin/bash
# for 语句示例 2
num=$(seq 10)
```

```
for i in "${num}"
do
  echo "${i}"
done
```

4.1.6 while循环语句

语法格式

```
while 条件
do
   执行语句
done
```

注意：

条件的类型：

命令、[[字符串表达式]]、((数字表达式))

场景：

只要条件满足，就一直循环下去

while语句示例

脚本内容

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

4.2 复杂流程控制语句

函数就是我们的复杂流程控制语句

4.2.1 函数基础知识

函数是什么？

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

简单函数格式：

定义函数：

```
函数名(){
    函数体
}
```

调用函数：

```
函数名
```

传参函数格式：

传参数

```
函数名 参数
```

函数体调用参数：

```
函数名(){
    函数体 $n
}
```

注意：

类似于shell内置变量中的位置参数

4.2.2 函数实践

简单函数定义和调用示例

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

函数传参和函数体内调用参数示例

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

函数调用脚本传参

```
#!/bin/bash
# 函数传参演示

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

# 函数传参
dayin $1
```

脚本传多参，函数分别调用示例

```
#!/bin/bash
# 函数的使用场景二
dayin(){
  echo "wo de mingzi shi $1"
  echo "wo de mingzi shi $2"
  echo "wo de mingzi shi $3"
}
dayin 111 df dfs
```

第 5 章 代码发布

5.1 代码发布简介

到现在为止我们学会了多个开发项目案例，也知道工作中如何做一个项目，但是一个项目方案的如何部署到公司的线

上服务器上，部署过程中都会经历哪些事情呢？接下来我们好好的来梳理一下。

5.1.1 代码发布介绍

什么是代码发布？

代码发布就是一句话：将我们的代码放到一台公司的互联网服务器上。
那么我们应该怎么来理解这句话呢？我们从三个方面来理解他。

如何理解这句话？

发布什么？
　　代码　　经过测试，功能完善，没有问题的代码
发布到哪里？
　　服务器　　所有人都能访问的到的一台服务器(有公网IP)
　　idc机房、阿里云、亚马逊、腾讯云、华为云、 ....
发布的效果？
　　web网页对外展示

5.1.2 发布方式

常见的代码发布方式有两种：手工方式和脚本方式。
这两种方式有什么区别呢？我们接下来好好的对比分析一下。

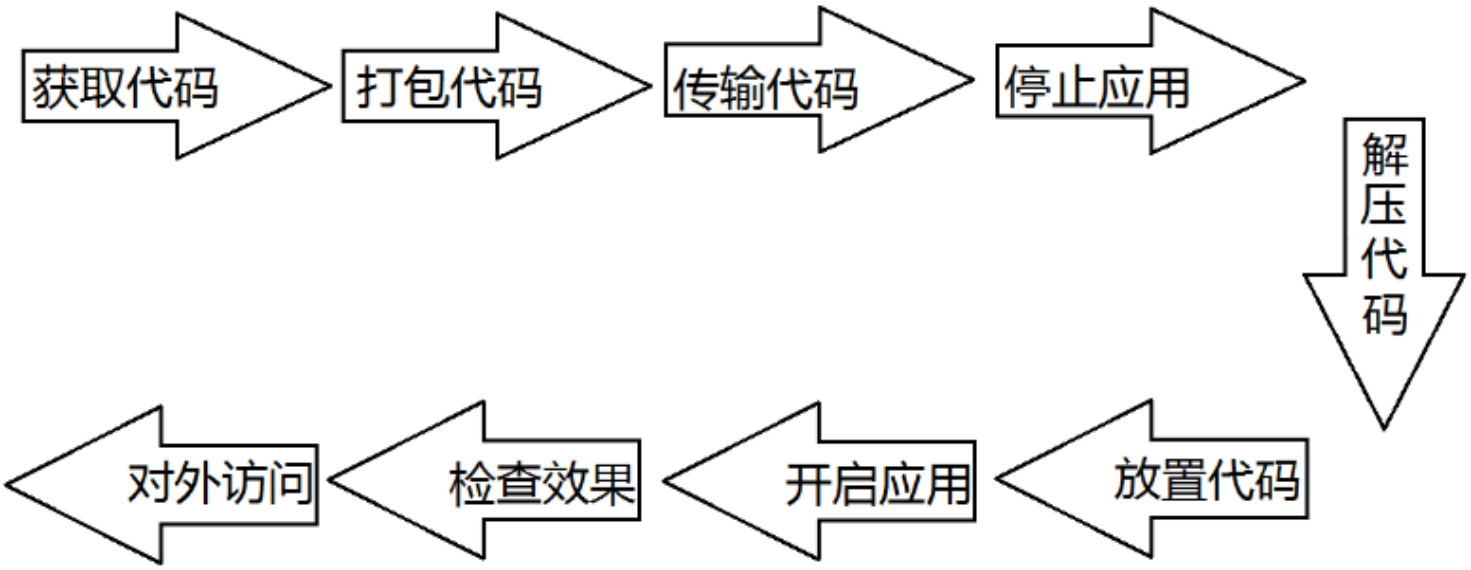
手工发布代码　　步行
　　慢
　　干扰因素多
　　不安全

脚本发布代码　　坐车
　　快
　　干扰因素少
　　安全

5.2 代码发布流程

5.2.1 流程简介

接下来我们来好好的说一下部署的流程：

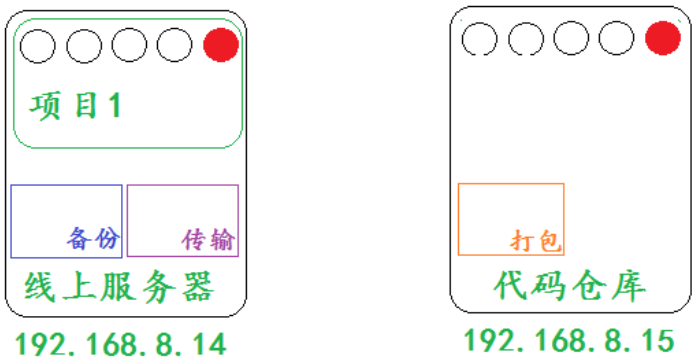


5.2.2 流程详解

接下来我们来对每个过程进行一个仔细的叙述

部署场景：

两台主机做部署动作



注意：

部署的文件就是两台主机右上角的红色内容

获取代码

代码仓库

集中式的

svn

分布式的

git

区别：

svn的几乎所有操作命令，都集中在我和代码仓库服务器处于网络连接状态。

git的几乎所有操作命令，可以在本地完成，和代码仓库服务器是否连接无关。

公司的代码仓库：

私有仓库 gitlab

内部服务器或者公网服务器

仓库权限

只有项目的开发人员才有权限，项目之外的人没有权限

代码权限：

开发、管理、查看

提交的方式：

代码版本号

打包代码

目的：

减少传输文件数量

减小传输文件大小

增强传输速率

常见打包方式：

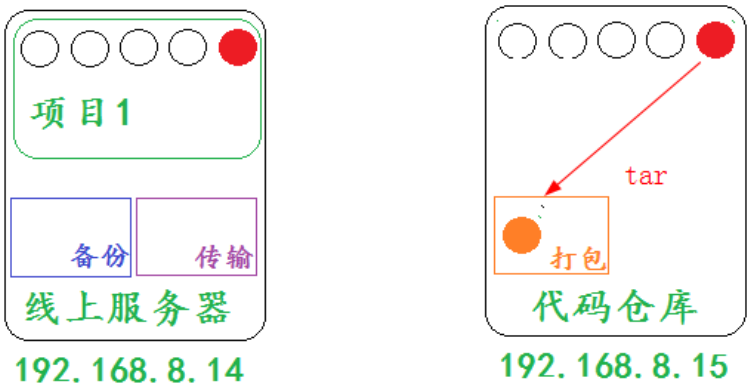
windows：

zip、rar...

linux：

tar、zip...

场景演示：



传输代码

传输方式：

有网情况下

多种方式：

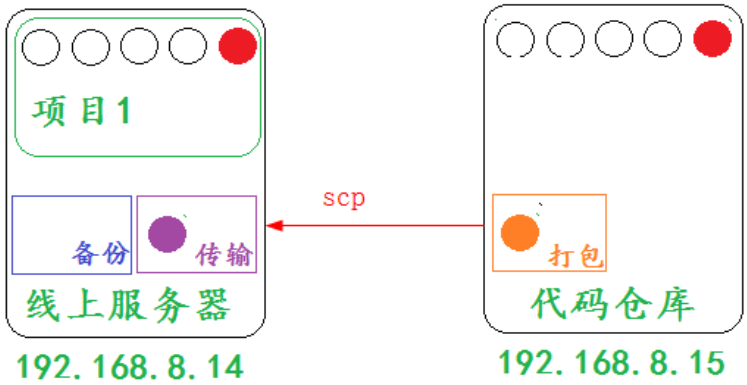
git、ftp、**scp**、共享挂载 cp、rsync

没有网情况下

物理方式：

U盘或者硬盘

场景效果：



关闭应用

开启什么应用，就关闭什么应用

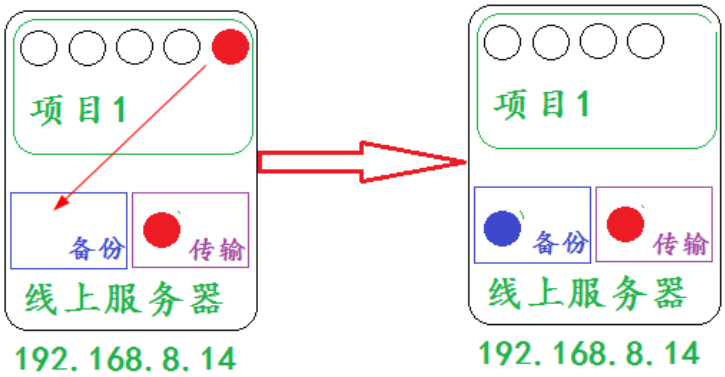
解压代码：

tar xf ...

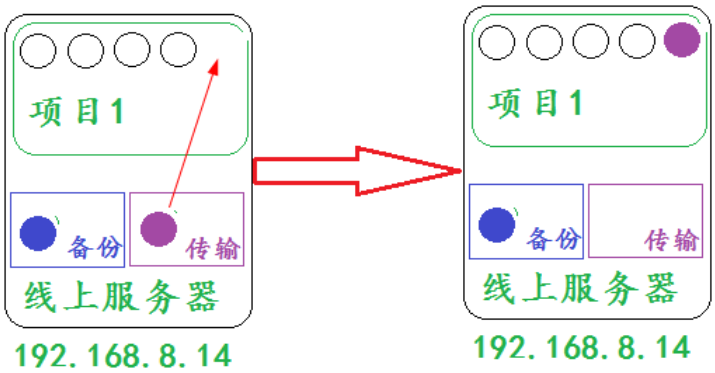
放置代码

为了避免我们在放置代码过程中，对老文件造成影响，所以我们放置代码一般分为两步：备份老文件和放置新文件。

备份原文件



放置新文件



注意：

两个文件的名称是一样的，只是内容不同

对整个应用项目来说，两个文件没有区别

开启应用

关闭什么应用就开启什么应用

检查

查看浏览器效果

5.2.3 技术关键点

文件的压缩和解压

文件的压缩

压缩格式：

```
tar zcvf 压缩后的文件名  将要压缩的文件
```

文件的解压

解压格式：

```
tar xzf 压缩后的文件名
```

命令参数详解

- z 指定压缩文件的格式为 tar.gz
- c 表示压缩
- v 显示详细过程
- f 指定压缩文件
- x 解压

查看压缩文件内容

zcat 压缩文件

文件的传输

scp传输工具：

命令格式：

scp 要传输的文件 要放置的位置

将本地文件推送到远程主机

```
scp python.tar.gz root@192.168.8.15:/root/
```

将远程主机的文件拉取到本地

```
scp root@192.168.8.15:/root/python.tar.gz ./
```

远端主机文件放置位置的表示形式：

远程连接的用户@远程主机:远程主机的目录路径

远端主机文件位置的表示形式：

远程连接的用户@远程主机:远程主机的文件路径

文件的备份

文件的备份要有一定的标志符号，我们就使用目前通用的时间戳的形式来表示

date命令详解：

命令格式：date [option]

常见参数：

%F 显示当前日期格式，%Y-%m-%d

%T 显示当前时间格式，%H:%M:%S

演示效果：

显示当前日期

```
# date +%F
2017-09-28
```

显示当前时间

```
# date +%T
03:06:30
```

根据上面的参数介绍，我们可以指定命令显示的格式，

年月日：date +%Y%m%d

时分秒：date +%H%M%S

演示效果：

显示当前日期

```
# date +%Y%m%d
20170928
```

显示当前时间

```
# date +%H%M%S
030643
```

指定时间戳格式：

年月日时分秒：date +%Y%m%d%H%M%S

时间戳演示效果：

指定的时间戳格式

```
# date +%Y%m%d%H%M%S
20170928030742
```

备份命令效果格式：

方式一：复制备份

```
cp nihao nihao-$(date +%Y%m%d%H%M%S)
```

方式二：移动备份

```
mv nihao nihao-$(date +%Y%m%d%H%M%S)
```

我们为了避免在放置新文件时候，出现验证操作，我们确定采用方式二：

第 6 章 环境部署

环境部署这块，我们需要有一个项目的基础环境，然后在这个基础环境上，再根据项目需求搭建一个能让项目代码正常运行的环境。所以这块的知识，我们从三个方面来学习

基础环境

方案分析

项目环境部署

6.1 基础环境

6.1.1 基础目录环境

使用root用户进行所有的操作

```
sudo passwd root
```

在windows中使用xshell连接linux

```
vi /etc/ssh/sshd_config
#PermitRootLogin prohibit-password
PermitRootLogin yes
service ssh restart
```

创建基本目录

```
# mkdir /data/{server,logs,backup,softs,virtual,scripts,codes} -p
# ls /data/
backup  logs  scripts  server  softs  virtual  codes
```

查看

```
admin-1@ubuntu:/data# tree -L 1 /data/
/data/
├── backup          备份
├── codes           代码
├── logs            日志
├── scripts         脚本
├── server          服务
├── softs           软件
└── virtual         虚拟环境
```

在windows中使用xshell传输文件

```
sudo apt install lrzsz
rz
```

6.1.2 主机网络环境

问题：

我们上面进行文件传输的时候，发现一个问题，每次进行文件传输都需要进行密码验证，这对我们来说，有些一丁点不舒服，那么有没有办法，让我舒服一点？

答案就是：主机间免秘钥认证

知识点：

什么是主机间免秘钥认证？

就是我和另外一台主机做任何事情，不需要输入密码认证，非常很方便。

方案详解

我们要做主机间免秘钥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

方案实施

1、生成秘钥对

```
ssh-keygen -t rsa
    -t      指定秘钥的类型
        rsa 秘钥类型

秘钥目录：/root/.ssh/
    私钥        id_rsa              钥匙
    公钥        id_rsa.pub          锁
```

2、编辑认证文件

```
admin-1@ubuntu:~/.ssh# cat /root/.ssh/authorized_keys
ssh-rsa AAAAB3N...PVp admin-1@ubuntu
```

注意：

认证文件内容和8.15机器的公钥文件内容一致

保证文件内容是一整行

3、编辑ssh配置文件

```
admin-1@ubuntu:~/.ssh# cat /etc/ssh/sshd_config
...
    AuthorizedKeysFile  %h/.ssh/authorized_keys
```

注意：

直接取消该行的注释即可

4、配置文件生效

重启ssh服务

```
service ssh restart
```

5、验证操作

```
ssh root@192.168.8.15
```

1和5是在8.14上操作

2-4是在8.15上操作

6.2 方案分析

6.2.1 需求

需求：

部署一个环境，支持我们的django项目正常运行

6.2.2 需求分析

分析：

2、python环境 ---> 3、python虚拟环境

1、django环境部署

4、django软件安装

5、项目基本操作

6、应用基本操作

7、view和url配置

8、问题：只有本机能访问

9、方案代理---- 10、nginx

11、nginx实现代理

13、pcre软件安装

12、nginx软件安装

14、nginx基本操作

15、nginx代理的配置

16、目录结构

17、查看配置文件

18、找到对应的代理配置项

19、启动django

20、启动nginx

21、整个项目调试

6.2.3 部署方案

环境部署方案

一、django环境部署

1.1 python虚拟环境

1.2 django环境部署

1.2.1 django软件安装

1.2.2 项目基本操作

1.2.3 应用基本操作

1.2.4 view和url配置

二、nginx代理django

2.1 nginx软件安装

2.1.1 pcre软件安装

2.1.2 nginx软件安装

2.1.3 nginx基本操作

2.2 nginx代理配置

2.2.1 目录结构查看

2.2.2 配置文件查看

2.2.3 编辑代理配置项

6.3 项目环境部署

6.3.1 python虚拟环境

软件安装

安装虚拟环境软件

```
sudo pip install virtualenv
sudo pip install virtualenvwrapper
```

虚拟环境基本操作

创建

```
mkvirtualenv 虚拟环境名称
```

进入

```
workon 虚拟环境名称
```

退出

```
deactivate
```

删除

```
remove 虚拟环境名称
```

6.3.2 django环境

django软件安装

注意：先进入虚拟环境

解压

```
cd /data/soft
tar xf Django-1.10.7.tar.gz
```

查看

```
cd Django-1.10.7
cat INSTALL or README
```

安装

```
python setup.py install
```

拓展知识点：

python类型软件的安装流程

普通：

解压　安装

特殊：

解压　编译　安装

编译：python setup.py build

django项目操作

创建项目

```
cd /data/server
django-admin startproject itcast
```

django应用操作

创建应用

```
cd /data/server/itcast
python manage.py startapp test1
```

注册应用

```
# vim itcast/settings.py


INSTALL_APP = [
    。。。
    'test1',
]
```

view和url配置

view 配置文件生效

```
admin-1@ubuntu:/data/soft# cat /data/server/itcast/test1/views.py
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def hello(resquest):
   return HttpResponse("itcast V1.0")
```

url文件配置

```
admin-1@ubuntu:/data/soft# cat /data/server/itcast/itcast/urls.py
...
from test1.views import *

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^hello/$', hello),
]
```

启动django:

```
cd /data/server/itcast
python  manage.py runserver
```

6.3.3 nginx环境

pcre软件安装

解压

```
cd /data/soft/
tar xzf pcre-8.39.tar.gz
```

查看帮助

```
cd pcre-8.39
INSTALL 或者 README
```

配置

```
./configure
```

编译

```
make
```

安装

```
make install
```

拓展知识点：

linux中软件安装的一般流程

解压

tar

解压文件，获取真正的配置文件

配置

configure

根据默认的配置项或者更改配置项，生成编译配置文件(Makefile)

编译

make

根据 Makefile 内容，编译生成指定的软件所需要的所有文件

安装

make install

将编译生成的所有文件，转移到软件指定安装的目录下面

--prefix

nginx软件安装

解压

```
cd /data/soft/
tar xf nginx-1.10.2.tar.gz
```

配置

```
cd nginx-1.10.2/
./configure --prefix=/data/server/nginx --without-http_gzip_module
```

编译

```
make
```

安装

```
make install
```

nginx简单操作

检查

```
/data/server/nginx/sbin/nginx -t
```

开启

```
/data/server/nginx/sbin/nginx
```

关闭

```
/data/server/nginx/sbin/nginx -s stop
```

重载

```
/data/server/nginx/sbin/nginx -s reload
```

6.3.4 nginx代理django

nginx配置简介

nginx的目录结构

```
admin-1@ubuntu:/data/server/nginx# tree -L 2 /data/server/nginx/
/data/server/nginx/
├── ...
├── conf              配置文件目录
|   ...
```

```
|   ├── nginx.conf        默认的配置文件
|  ...
├── ...
├── html              网页文件
|   ├── 50x.html
|   └── index.html
├── logs              日志目录
|   ├── access.log
|   └── error.log
├── ...
├── sbin              执行文件目录
|   └── nginx
├── ...
```

nginx配置文件介绍

全局配置段

http配置段

server配置段　　项目或者应用

location配置段　　url配置

nginx代理配置

需求：

访问192.168.8.14/hello/ 跳转到 127.0.0.1:8000/hello/的django服务

代理配置项

```
#location ~ \.php$ {
#    proxy_pass   http://127.0.0.1;
#}
```

编辑配置文件实现代理功能

配置内容

```
62: location /hello/ {
63:     proxy_pass http://127.0.0.1:8000;
64: }
```

配置文件生效

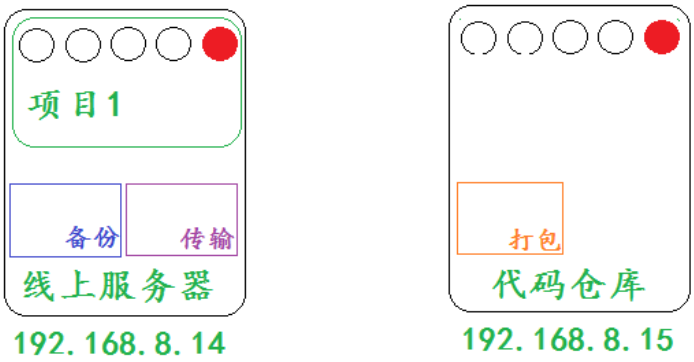
```
/data/server/nginx/sbin/nginx -t
/data/server/nginx/sbin/nginx -s reload
```

第 7 章 手工代码发布

7.1 方案分析

发布需求：

手工方式部署代码

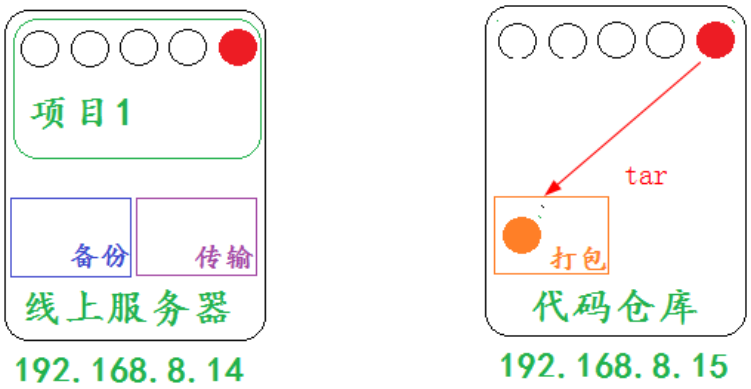


发布方案：

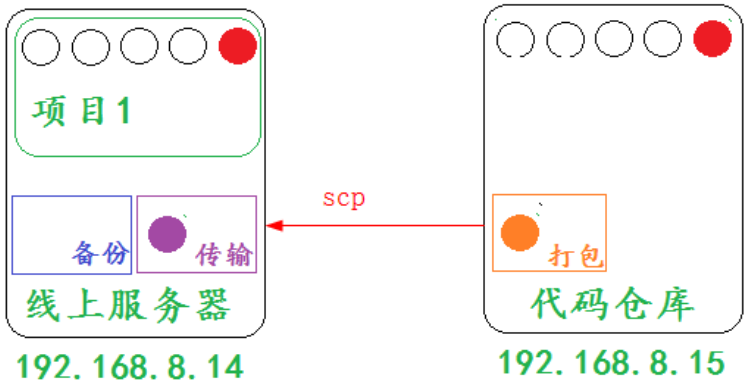
获取代码

sed -i 's#文件原始的内容#替换后的内容#g' 要更改到文件名

打包代码



传输代码

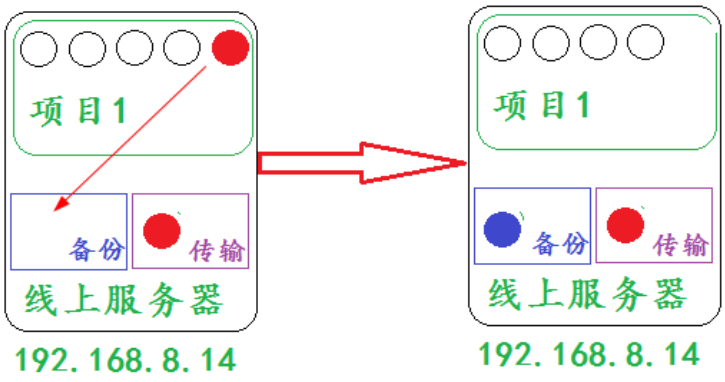


关闭应用

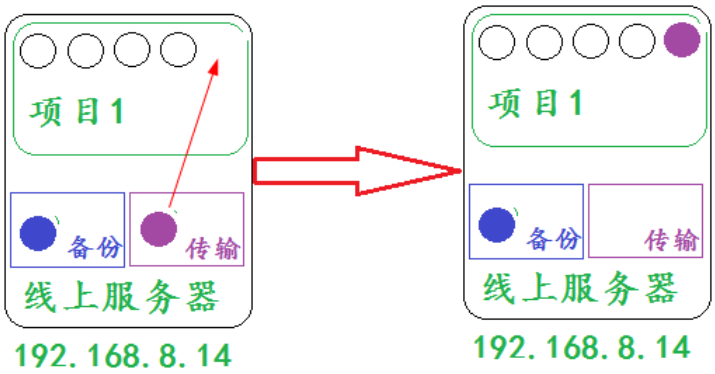
解压代码

放置代码

备份老文件



放置新文件



开启应用

检查

7.2 方案实施

获取代码

```
mkdir /data/codes -p
cd /data/codes
sed -i 's#1.0#1.1#' django/views.py
sed -i  's#原内容#替换后内容#g' 文件
```

分隔符：# / @

打包代码

```
cd /data/codes/
tar zcf django.tar.gz django
```

注意：

前面两步在192.168.8.15 上操作

传输代码

```
scp root@192.168.8.15:/data/codes/django.tar.gz ./
```

关闭应用

关闭nginx应用

```
/data/server/nginx/sbin/nginx -s stop
```

关闭django应用

根据端口查看进程号，

```
lsof -Pti :8000
```

杀死进程号

```
kill 56502
```

一条命令搞定它：

```
kill $(lsof -Pti :8000)
```

解压代码

```
cd /data/codes
tar xf django.tar.gz
```

放置代码

备份老文件

需求：备份的格式：

文件名-时间戳

时间戳：年月日时分秒

date +%Y%m%d%H%M%S

```
mv /data/server/itcast/test1/views.py /data/backup/views.py-$(date +%Y%m%d%H%M%S)
```

放置新文件

```
cd /data/codes
mv django/views.py /data/server/itcast/test1/
```

开启应用

开启django应用

```
source /data/virtual/venv/bin/activate
cd /data/server/itcast/
python manage.py runserver >> /dev/null 2>&1 &
deactivate
```

开启nginx应用

```
/data/server/nginx/sbin/nginx
```

检查

```
netstat -tnulp | grep :80
```

第 8 章 脚本发布代码

关于脚本发布代码部分呢，我们将这个代码部署流程拆分成两部分：简单脚本 (远端主机上执行) 和大型脚本 (线上机器执行)，这样我们能从两个方面来学习生产中的脚本如何编写。

8.1 简单脚本编写

8.1.1 命令罗列

实现简单的功能--- 简单的命令罗列

```
admin-1@ubuntu:/data/scripts# cat tar_code.sh
#!/bin/bash
# 功能：打包代码
# 脚本名：tar_code.sh
# 作者：python n 期全体
# 版本：V 0.1
# 联系方式：长安街 1 号 太和殿旁边 中南海 1 号厅


cd /data/codes
[ -f django.tar.gz ] && rm -f django.tar.gz
tar zcf django.tar.gz django
```

脚本编写完成后，进行测试：

```
sed -i 's#1.1#1.2#' /data/codes/django/views.py
bash /data/scripts/tar_code.sh
```

查看压缩文件内容

```
zcat django.tar.gz
```

8.1.2 固定内容变量化

脚本优化之 固定内容变量化

```
admin-1@ubuntu:/data/scripts# cat tar_code.sh
#!/bin/bash
# 功能：打包代码
# 脚本名：tar_code.sh
# 作者：python n 期全体
# 版本：V 0.2
# 联系方式：长安街 1 号 太和殿旁边 中南海 1 号厅

FILE='django.tar.gz'
CODE_DIR='/data/codes'
CODE_PRO='django'

cd "${CODE_DIR}"
[ -f "${FILE}" ] && rm -f "${FILE}"
tar zcf "${FILE}" "${CODE_PRO}"
```

脚本编写完成后，进行测试：

```
sed -i 's#1.2#1.3#' /data/codes/django/views.py
bash /data/scripts/tar_code.sh
```

查看压缩文件内容

```
zcat django.tar.gz
```

8.1.3 功能函数化

脚本优化之 功能函数化

```
admin-1@ubuntu:/data/scripts# cat tar_code.sh
#!/bin/bash
# 功能：打包代码
# 脚本名：tar_code.sh
# 作者：python n 期全体
# 版本：V 0.3
# 联系方式：长安街 1 号 太和殿旁边 中南海 2 号厅

FILE='django.tar.gz'
CODE_DIR='/data/codes'
CODE_PRO='django'

code_tar(){
    cd "${CODE_DIR}"
    [ -f "${FILE}" ] && rm -f "${FILE}"
    tar zcf "${FILE}" "${CODE_PRO}"
}
code_tar
```

脚本编写完成后，进行测试：

```
sed -i 's#1.2#1.3#' /data/codes/django/views.py
bash /data/scripts/tar_code.sh
```

查看压缩文件内容

```
zcat /data/codes/django.tar.gz
```

8.1.4 远程执行

远程命令执行

格式：

ssh 远程主机登录用户名@远程主机ip地址 "执行命令"

效果

```
admin-1@ubuntu:/data/server/itcast# ssh root@192.168.8.15 "ifconfig eth0"
eth0      Link encap:Ethernet  HWaddr 00:0c:29:f7:ca:d4
          inet addr:192.168.8.15  Bcast:192.168.56.255  Mask:255.255.255.0
          ...
```

远程执行脚本测试

远程更新文件内容

```
ssh root@192.168.8.15 "sed -i /'s#1.4#1.5#' /data/codes/django/views.py"
```

远程查看脚本

```
ssh root@192.168.8.15 "ls /data/scripts"
```

远程执行脚本

```
ssh root@192.168.8.15 "/bin/bash /data/scripts/tar_code.sh"
```

远程检查更新效果

```
ssh root@192.168.8.15 "zcat /data/codes/django.tar.gz"
```

8.2 大型脚本编写

本地是指 192.168.8.14 主机(线上服务器)

编写大型脚本有一个流程：

先写框架 -- 再填命令 -- 完善功能

8.2.1 功能框架

需求：先将脚本所涉及的所有业务流程跑通

方案：

使用函数来体现

脚本实施：

```
#!/bin/bash
# 功能：打包代码
# 脚本名：deploy.sh
# 作者：python n 期全体
# 版本：V 0.1
# 联系方式：长安街 1 号 太和殿旁边 国务院 1 号厅

# 获取代码
get_code(){
  echo "获取代码"
}

# 打包代码
tar_code(){
  echo "打包代码"
}
```

```
# 传输代码
scp_code(){
  echo "传输代码"
}

# 关闭应用
stop_serv(){
  echo "关闭应用"
  echo "关闭 nginx 应用"
  echo "关闭 django 应用"
}

# 解压代码
untar_code(){
  echo "解压代码"
}

# 放置代码
fangzhi_code(){
  echo "放置代码"
  echo "备份老文件"
  echo "放置新文件"
}

# 开启应用
start_serv(){
  echo "开启应用"
  echo "开启 django 应用"
  echo "开启 nginx 应用"
}

# 检查
check(){
  echo "检查项目"
}

# 部署函数
deploy_pro(){
  get_code
  tar_code
  scp_code
  stop_serv
  untar_code
  fangzhi_code
  start_serv
  check
}

# 主函数
main(){
```

```
  deploy_pro
}

# 执行主函数
main
```

8.2.2 命令填充

需求：

在流程跑通的情况下，执行完整的代码部署过程

方案：

在流程框架中，填写执行没有任何问题的命令

脚本实施：

```
#!/bin/bash
# 功能：打包代码
# 脚本名：deploy.sh
# 作者：python n 期全体
# 版本：V 0.2
# 联系方式：长安街 1 号 太和殿旁边 国务院 2 号厅

# 获取代码
get_code(){
  echo "获取代码"
}

# 打包代码
tar_code(){
  echo "打包代码"
  ssh root@192.168.8.15 "/bin/bash /data/scripts/tar_code.sh"
}

# 传输代码
scp_code(){
  echo "传输代码"
  cd /data/codes
  [ -f django.tar.gz ] && rm -f django.tar.gz
  scp root@192.168.8.15:/data/codes/django.tar.gz ./
}

# 关闭应用
stop_serv(){
  echo "关闭应用"
  echo "关闭 nginx 应用"
  /data/server/nginx/sbin/nginx -s stop
  echo "关闭 django 应用"
  kill $(lsof -Pti :8000)
}

# 解压代码
```

```
untar_code(){
  echo "解压代码"
  cd /data/codes
  tar xf django.tar.gz
}

# 放置代码
fangzhi_code(){
  echo "放置代码"
  echo "备份老文件"
  mv /data/server/itcast/test1/views.py /data/backup/views.py-$(date +%Y%m%d%H%M%S)
  echo "放置新文件"
  mv /data/codes/django/views.py /data/server/itcast/test1/
}

# 开启应用
start_serv(){
  echo "开启应用"
  echo "开启 django 应用"
  export WORKON_HOME=/data/virtual
  source /usr/local/bin/virtualenvwrapper.sh
  python manage.py runserver >> /dev/null 2>&1 &
  deactivate
  echo "开启 nginx 应用"
  /data/server/nginx/sbin/nginx
}
# 检查
check(){
  echo "检查项目"
  netstat -tnulp | grep ':80'
}

...
```

8.2.3 增加日志功能

需求：

1、追踪记录
2、数据说话

方案：

增加日志功能
1、日志文件
/data/logs/deploy.log
2、日志格式
日期 时间 脚本名称 步骤

日期：date +%F
时间：date +%T
脚本：$0

脚本实施：

```
#!/bin/bash
...
LOG_FILE='/data/logs/deploy.log'
# 增加日志功能
write_log(){
  DATE=$(date +%F)
  TIME=$(date +%T)
  buzhou="$1"
  echo "${DATE} ${TIME} $0 : ${buzhou}" >> "${LOG_FILE}"
}

# 获取代码
get_code(){
  ...
  write_log "获取代码"
}

# 打包代码
tar_code(){
  ...
  write_log "打包代码"
}

# 传输代码
scp_code(){
  ...
  write_log "传输代码"
}

# 关闭应用
stop_serv(){
  ...
  write_log "关闭应用"
  ...
  write_log "关闭 nginx 应用"
  ...
  write_log "关闭 django 应用"
}

# 解压代码
untar_code(){
  ...
  write_log "解压代码"
}

# 放置代码
fangzhi_code(){
  ...
  write_log "放置代码"
```

```
  ...
  write_log "备份老文件"
  ...
  write_log "放置新文件"
}

# 开启应用
start_serv(){
  ...
  write_log "开启应用"
  ...
  write_log "开启 django 应用"
  ...
  write_log "开启 nginx 应用"
}
# 检查
check(){
  ...
  write_log "检查项目"
}


...
```

8.2.4 增加锁文件功能

需求：

同一时间段内，只允许有一个用户来执行这个脚本

如果脚本执行的时候，有人在执行，那么输出报错：

脚本 deploy.sh 正在运行，请稍候...

设计：

锁文件　　/tmp/deploy.pid

脚本执行的时候，需要创建锁文件

脚本执行结束的时候，需要删除锁文件

脚本实施

```
#!/bin/bash
...
PID_FILE='/tmp/deploy.pid'
...
# 增加锁文件功能
add_lock(){
  echo "增加锁文件"
  touch "${PID_FILE}"
  write_log "增加锁文件"
}

# 删除锁文件功能
del_lock(){
  echo "删除锁文件"
```

```
  rm -f "${PID_FILE}"
  write_log "删除锁文件"
}

# 部署函数
deploy_pro(){
  add_lock
  ...
  del_lock
}

# 脚本报错信息
err_msg(){
  echo "脚本 $0 正在运行，请稍候..."
}

# 主函数
main(){
   if [ -f "${PID_FILE}" ]
   then
     err_msg
   else
     deploy_pro
   fi
}

# 执行主函数
main
```

8.2.5 脚本流程知识点填充

需求：

如果我给脚本输入的参数是deploy，那么脚本才执行，否则的话，提示该脚本的使用帮助信息，然后退出

提示信息：脚本 deploy.sh 的使用方式： deploy.sh [deploy]

分析：

脚本传参，就需要在脚本内部进行调用参数

使用 $1 来完成

脚本内容就需要对传参的内容进行判断

使用case语句来完成

脚本的帮助信息

使用一个usage函数来完成

方案：

1、脚本的传参

脚本执行：bash deploy.sh deploy

2、脚本的帮助信息

完成一个usage函数

3、位置参数的调用

在main函数中，结合case语句，对传入的参数进行判断

传入参数匹配到deploy，那么就执行代码部署流程

如果匹配不到，那么输出脚本的帮助信息，然后退出

脚本实施

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

# 脚本帮助信息
usage(){
  echo "脚本 $0 的使用方式: $0 [deploy]"
  exit
}

# 主函数
main(){
  case "$1" in
    "deploy")
      if [ -f "${PID_FILE}" ]
      then
         err_msg
      else
        deploy_pro
      fi
    ;;
    *)
      usage
    ;;
  esac
}

# 执行主函数
main $1
```

8.2.6 输入参数安全优化

需求：

对脚本传入的参数进行判断，如果传入的参数数量不对，直接提示脚本使用方式，然后退出

方案：

条件表达式 + $#

脚本实施

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

# 执行主函数
if [ $# -eq 1 ]
then
  main $1
else
  usage
fi
```

8.3脚本调试功能

我们介绍脚本调试的时候呢，主要分三种方式来介绍：

-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

8.4 生产脚本编写总结

8.4.1 简单脚本编写总结

1、手工执行的命令一定要可执行
1、命令简单罗列
2、固定的内容变量化
3、功能函数化

8.4.2 复杂脚本编写总结

1、手工执行的命令一定要可执行
2、根据发布流程编写脚本的框架
3、将手工执行的命令填充到对应的框架函数内部
4、增加日志功能，方便跟踪脚本历史执行记录
5、主函数中逻辑流程控制好
6、设计安全的方面：
　增加锁文件，保证代码发布的过程中不受干扰，
　输入参数数量
　输入参数匹配
　脚本帮助信息
7、调试脚本

8.4.3 注意事项：

1、命令一定要保证能正常执行
2、成对的符号，要成对写，避免丢失
3、函数调用，
　写好函数后，一定要在主函数中进行调用
4、避免符号出现中文
5、命令变量的写法一定要规范
6、固定的内容一定要变量实现，方便以后更改
7、日志的输出
8、脚本的传参和函数的传参要区别对待