

Bash 编程学习笔记

【前言】

建议：

- 1、bash 简单来说就是 Linux 的语言，擅长 bash 可以极大的加深对 Linux 理解和相关命令的应用。
- 2、你至少需要懂得 Linux 的一些极为基础的命令
- 3、你至少需要对任意一种现有的编程语言略有耳闻，并且打过交道
- 4、你最好有一台 Linux 机器可以随时使用，因为本文没讲太多硬道理，基本都是例子堆出来的。

倡议

- 1、诚邀各位师傅学习过程中对此笔记进行修正和内容丰富。对于贡献更新内容和提供的建议采纳后将更新在新一版的 pdf 中，并对为此贡献的师傅在页首致谢。
- 2、整个笔记的定位保证能在生活中的大部分场景满足大部分需求，对于深入的理论和少见的用法并没有过多的介绍，一是避免太过于枯燥，而是避免太多的理论性文字令人头大。由于鄙人才疏学浅，经验欠缺，如有错误还请师傅们谅解。
- 3、为了更好的阅读体验，本笔记的 **pdf 最新版** 也以发出，欢迎各位师傅们在**笔记结尾根据 Github 链接自行前往下载**。

致谢：

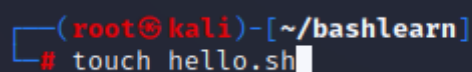
- 1、感谢 C++ 开发俱乐部 讨论群里的 Tom、罗上文、钱能 等几位前辈的慷慨解惑，有的知识点查了很多资料都难的其精髓，是他们指点迷津。
- 2、感谢我的女朋友，到现在还不让我遇到，让我一个人默默的学安全，不打扰我分毫。

【第一节】简单脚本编写

打印helloworld

流程：

首先，在Linux机器上创建一个hello.sh 文件



```
(root@kali)-[~/bashlearn]  
# touch hello.sh
```

在文件中输入如下字符：

```
#!/bin/bash  
echo hello world
```

保存文件后，在命令行输入 `./hello.sh`，之后hello world将会出现在命令行里

```
(root@kali)-[~/bashlearn]
# ./hello.sh
hello world
```

可能遇到的问题和疑惑

- 权限问题：一般直接创建的文件时没有执行权限的，这时需要我们自己为文件添加可执行权限，创建文件后在命令行里输入如下内容即可。

```
chmod +x hello.sh
```

- 什么是.sh文件？

粗暴一点理解的方式就是，我们在操作Linux的时候一般在命令行里一次只能输入和执行一条指令，然而如果我们把需要执行的命令写在.sh文件里，那么这个时候我们就可以一次执行很多条指令，并且指令执行的顺序是按照指令在文件里出现的顺序进行执行的。其实Windows里面的 bat 文件也是这个意思。

- 好像怎么弄都不行咋回事？

有的师傅发现怎么弄都不行，这可能是你的 bash 位置指定错了，注意看脚本内容的第一行是 `#!/bin/bash`，在我的 Linux 机器上，bash就在 /bin/bash 处，师傅如果遇到这么问题先确定一下自己机器的 bash 在哪里，再执行脚本文件。

简单的备份任务

流程：

直接在创建的.sh文件里输入以下命令，用来把 /home 目录里面的内容备份到/var目录下面。

```
#!/bin/bash
tar -czf /var/home-backup.tgz /home
```

执行：

```
(root@kali)-[~/bashlearn]
# bash backup.sh
tar: Removing leading `/' from member names
```

然后就可以看见备份文件：

```
(root@kali)-[/var]
# ls
backups  cache  home-backup.tgz  lib  local  lock  log
```

可能遇到的问题和疑惑

- bash backup.sh啥意思？

就是懒得给文件加执行权限这一步，然后直接输入bash让解释器用bash的方式解释文件内容。和加权限了然后执行文件一个意思。

【第二节】重定向

这几句话说给新人听：本节我个人认为有点难，比较绕，因为涉及到一些比较底层的东西，当然我在解释时会尽量简单生动，绕过太过于生涩的专业名词堆砌。

鉴于本节的难度我个人推荐的学习流程是先浅尝辄止的过一遍【流程】里的这些例子，然后仔细理解一下【可能遇到的问题和疑惑】里的东西，然后再好好理解一遍【流程】里的各个例子。

首先先了解一下 **文件描述符**，文件描述符有三种：stdin、stdout、stderr

流程

stdout 重定向到文件 (> 符号)

输入以下命令：

```
ls -l > ls-l.txt
```

正常情况下在Linux的terminal命令行输入 `ls -l` 会显示当前目录下所有文件或者文件夹的详细信息到屏幕上如下：

```
(root@kali)-[~/bashlearn]
# ls -l
total 8
-rw-r--r-- 1 root root 48 Nov 19 08:50 backup.sh
-rwxr-xr-x 1 root root 29 Nov 19 07:45 hello.sh
```

而当在命令行输入 `ls -l>ls-l.txt`，计算机会先在当前目录下创建一个文件名为 `ls-l.txt`，然后将屏幕上本来应该显示在屏幕上的信息输入到 `ls-l.txt` 文件里（当前目录这个文件名没有被使用，若已存在同名文件则会清空这个同名文件内容然后把内容输入进去），如下图：

```
(root@kali)-[~/bashlearn]
# ls -l>ls-l.txt

(root@kali)-[~/bashlearn]
# cat ls-l.txt
total 8
-rw-r--r-- 1 root root 48 Nov 19 08:50 backup.sh
-rwxr-xr-x 1 root root 29 Nov 19 07:45 hello.sh
-rw-r--r-- 1 root root 0 Nov 20 03:09 ls-l.txt
```

stderr重定向到文件 (2> 符号)

stderr和stdout差不多，所以这个和上面哪一个差不多，同样是把本应输出在屏幕的信息输出到文件里。只不过一个用在命令执行正确时，一个用在命令执行失败时。

例一：在终端输入以下命令：

```
grep da * 2> grep-errors.txt
```

如图：

```
(root@kali)-[~/bashlearn]
# grep da * 2>grep-errors.txt

(root@kali)-[~/bashlearn]
# cat grep-errors.txt

(root@kali)-[~/bashlearn]
```

因为这条指令正确执行了，并没有错误信息，所以 grep-errors.txt 文件为空。

例二：切换到普通用户去权限，然后输入以下命令：

```
ls -l ~/bashlearn 2>/home/kali/stderr.txt
```

命令的意思就是以普通用户身份去读取root目录下的信息，直接在终端执行命令显示如下：

```
(kali㉿kali)-[/root/bashlearn]
$ ls -l ~/bashlearn
ls: cannot access '/home/kali/bashlearn': No such file or directory
```

使用stdout后：

```
(kali㉿kali)-[/root/bashlearn]
$ ls -l ~/bashlearn >/home/kali/stderr1.txt
ls: cannot access '/home/kali/bashlearn': No such file or directory

(kali㉿kali)-[/root/bashlearn]
$ cat /home/kali/stderr1.txt
```

使用stderr后：

```
(kali㉿kali)-[/root/bashlearn]
$ ls -l ~/bashlearn 2>/home/kali/stderr.txt

(kali㉿kali)-[/root/bashlearn]
$ cat /home/kali/stderr.txt
ls: cannot access '/home/kali/bashlearn': No such file or directory
```

解释：因为普通用户没有权限读取root目录，所以这条命令必然是会出错的，直接执行当命令执行失败后会把错误信息显示在屏幕；使用stdout依然显示在屏幕是因为命令执行错误，所以stdout没有生效；使用stderr后错误信息并没有显示在屏幕上，而是被重定向到了文件里，这是命令执行发生错误，所以stderr生效。正如上所说：stdout用在命令执行正确时，stderr用在命令执行失败时。

stdout重定向到stderr (1>&2 符号)

先在Linux中切换到普通用户，先在终端输入以下内容：

```
ls -l /home/kali /root
```

命令的意思是：以普通身份打开普通用户的目录和root用户的目录，普通用户目录能打开会正常显示，而root目录无法打开会报错，那么直接在终端输入指令会在屏幕上看到以下两部分内容：

```
(kali@kali)-[/root]
$ ls -l /home/kali /root
/home/kali:
total 40
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Desktop
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Documents
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Downloads
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Music
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Pictures
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Public
-rw-r--r-- 1 kali kali    0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali   68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali  633 Nov 20 06:38 stdout2stderr.txt
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Templates
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Videos
ls: cannot open directory '/root': Permission denied
```

然后在命令行输入以下指令：

```
ls -l /home/kali /root 2>/home/kali/stdout2stderr.txt 1>&2
```

会发下屏幕没有信息显示，打开文件可以看见正确信息和错误信息都被重定向到了文件里，如下：

```
(kali@kali)-[~]
$ ls -l /home/kali /root 2>/home/kali/stdout2stderr.txt 1>&2

(kali@kali)-[~]
$ cat stdout2stderr.txt
/home/kali:
total 36
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Desktop
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Documents
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Downloads
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Music
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Pictures
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Public
-rw-r--r-- 1 kali kali    0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali   68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali    0 Nov 20 06:38 stdout2stderr.txt
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Templates
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Videos
ls: cannot open directory '/root': Permission denied
```

命令的意思是：以普通身份打开两个目录，然后先把stderr重定向到文件，然后把stdout重定向到stderr（即相应文件），这个时候stdout和stderr都是指向这个文件的，然后把前面的内容发送到重定向文件里。因此屏幕上不直接显示信息而是在相应文件里才能看到stdout和stderr的信息。

这里给大家留一个思考题，下面这两个命令有啥区别：

1. `ls -l /home/kali /root 1>&2 2>/home/kali/stdout2stderr.txt`
2. `ls -l /home/kali /root 2>/home/kali/stdout2stderr.txt 1>&2`

stderr重定向到stdout（`2>&1` 符号）

此处还是以上面那个普通身份分别打开高权限目录和低权限目录为例子。

在终端执行以下命令：

```
ls -l /home/kali /root >/home/kali/stderr2stdout.txt 2>&1
```

会在文件中得到如下信息：

```
(kali@kali)-[~]
$ ls -l /home/kali /root >/home/kali/stderr2stdout.txt 2>61
ls: cannot open directory '/root': Permission denied

(kali@kali)-[~]
$ cat stderr2stdout.txt
/home/kali:
total 40
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Desktop
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Documents
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Downloads
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Music
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Pictures
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Public
-rw-r--r-- 1 kali kali 0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali 0 Nov 20 07:14 stderr2stdout.txt
-rw-r--r-- 1 kali kali 68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:13 stdout2stderr.txt
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Templates
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Videos
ls: cannot open directory '/root': Permission denied
```

解释：在上一个例子的加持下，这个例子就先的很好解释了。命令首先分别打开两个目录，然后对文件操作符进行重定向，即先把stdout指向文件，再把stderr指向stdout（即相应文件），这个时候再对文件流进行发送，即发送到了文件里。所以正确的和错误的信息都显示在了文件里。

stderr 和 stdout 重定向到文件（&> 符号）

还是以以一个普通身份分别打开高权限和低权限目录为例。在终端中输入以下命令：

```
ls -l /home/kali /root &>/home/kali/stderrANDstdout2file.txt
```

这时会在文件里发现以下信息：

```
(kali@kali)-[~]
$ ls -l /home/kali /root &>/home/kali/stderrANDstdout2file.txt

(kali@kali)-[~]
$ cat stderrANDstdout2file.txt
/home/kali:
total 44
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Desktop
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Documents
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Downloads
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Music
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Pictures
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Public
-rw-r--r-- 1 kali kali 0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:14 stderr2stdout.txt
-rw-r--r-- 1 kali kali 0 Nov 20 07:33 stderrANDstdout2file.txt
-rw-r--r-- 1 kali kali 68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:13 stdout2stderr.txt
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Templates
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Videos
ls: cannot open directory '/root': Permission denied
```

在上面两个例子的加持下，这个例子理解起来显得格外的轻松。&> 的意思就是同时把stdout和stderr都指向后面跟着的这个文件。因此可以在文件里同时看到正确的、错误的命令执行输出信息。

可能遇到的问题和疑惑

- 什么是文件描述符？

维基百科是这样定义的：文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。

简单点理解就是，在Linux中，一切皆文件，那么文件描述符就好比一个指针，定义了指针所指向的这一块内存的操作方式，不同的文件描述符所指向的内存里的数据将会通过不同的方式进行处理。说的再浅一点就是计算机通过文件描述符对计算机里面的各种流入流出的数据进行第一次基本的分类。

一般情况下，会先将其分为三类，即 stdin（记作 0）、stdout（记作 1）、stderr（记作 2）。分别代表了标准输入流、标准输出流、标准错误流。

- 什么是重定向？

就是改变数据流动的方向。就是数据的流动方向：

输入方向就是数据从哪里流向程序。数据默认从键盘流向程序，如果改变了它的方向，数据就从其它地方流入，这就是输入重定向。

输出方向就是数据从程序流向哪里。数据默认从程序流向显示器，如果改变了它的方向，数据就流向其它地方，这就是输出重定向。

重定向一般通过在命令间插入特定的符号来实现，一般是 `>`、`<`、`>>`、`<<`。

- `>`、`1>&2`、`2>&1`、`&>` 这些符号是啥意思？

`>`：代表输出重定向，就是把符号前面的命令执行后产生的信息输出到那个地方。在执行指令时，如果不使用这些符号，指令运行的结果会显示在屏幕上，而使用了该符号之后，指令的执行结果就会被显示在其他地方。与之相反的是输入重定向（`<` 符号）

`1>&2`：由上面知道**文件描述符 0 通常是标准输入（STDIN），1 是标准输出（STDOUT），2 是标准错误输出（STDERR）**。所以 `1>&2` 的意思就是把标准输出（STDOUT）重定向到标准错误输出（STDERR）。`&` 可以理解为取址，就是把 标准输出 指向 标准错误输出 所在的地方。

`2>&1`：由上一条可以知道，这个的意思就是把 标准错误输出 指向 标准输出 的地方。

`&>`：跟上面类似，这个符号的意思就是把 标准输出 和 标准错误输出 指向同一个地方。

【第三节】管道

管道以一个竖线为符号，像这样：`|`，他的意思就是把这个符号之前的命令执行完输出的结果当作这符号之后的命令执行之前的所具有的初始数据。

流程

如下图，在当前目录下有以下文件：

```

(kali㉿kali)-[~]
$ ls -l
total 48
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Desktop
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Documents
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Downloads
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Music
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Pictures
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Public
-rw-r--r-- 1 kali kali 0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:14 stderr2stdout.txt
-rw-r--r-- 1 kali kali 758 Nov 20 07:33 stderrANDstdout2file.txt
-rw-r--r-- 1 kali kali 68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:13 stdout2stderr.txt
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Templates
drwxr-xr-x 2 kali kali 4096 Aug 31 02:38 Videos

```

如果需要拿到当前目录下所有的 txt 文件名，按照已掌握的技能来看是：先把这些文件重定向到文件里，然后在筛选出文件里以 txt 结尾的文件，如下：

```

(kali㉿kali)-[~]
$ ls -l > pipdemo.txt

(kali㉿kali)-[~]
$ grep .txt pipdemo.txt
-rw-r--r-- 1 kali kali 0 Nov 21 08:10 pipdemo.txt
-rw-r--r-- 1 kali kali 0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:14 stderr2stdout.txt
-rw-r--r-- 1 kali kali 758 Nov 20 07:33 stderrANDstdout2file.txt
-rw-r--r-- 1 kali kali 68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:13 stdout2stderr.txt

```

这样子的话整个步骤就用了两步才拿到我们想要的输出结果。而如果使用管道符的话，在终端输入一次命令就能得到想要的输出内容，如下：

```

(kali㉿kali)-[~]
$ ls -l | grep .txt
-rw-r--r-- 1 kali kali 746 Nov 21 08:10 pipdemo.txt
-rw-r--r-- 1 kali kali 0 Nov 20 03:58 stderr1.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:14 stderr2stdout.txt
-rw-r--r-- 1 kali kali 758 Nov 20 07:33 stderrANDstdout2file.txt
-rw-r--r-- 1 kali kali 68 Nov 20 03:53 stderr.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:13 stdout2stderr.txt

```

发现了吗，这就是不一样，这里以一个很小的场景为例子可能感觉不到什么，能如果能在终端敲更少的字母而得到通用的结果，何乐而不为呢？

通道符号可以多次使用，如下图：

```

(kali㉿kali)-[~]
$ ls -l | grep "\.txt$" | grep out
-rw-r--r-- 1 kali kali 692 Nov 20 07:14 stderr2stdout.txt
-rw-r--r-- 1 kali kali 758 Nov 20 07:33 stderrANDstdout2file.txt
-rw-r--r-- 1 kali kali 692 Nov 20 07:13 stdout2stderr.txt

```


可能遇到的问题和疑惑

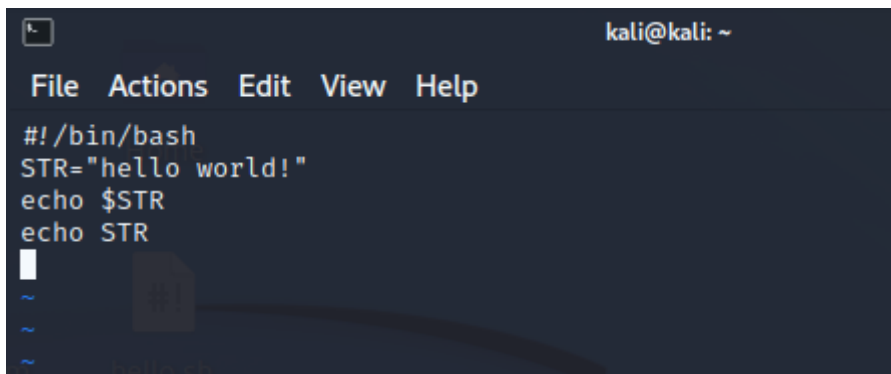
一般管道符使用不会出什么差错，出现差错一般就是检查前后命令是否由输入正确。确保前面的命令有输出，后面的命令需要输入。

【第四节】变量

流程

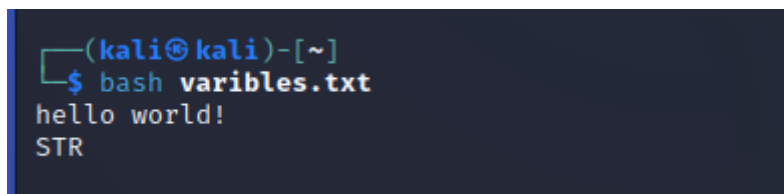
第一个例子

变量在许多编程语言中都存在。bash脚本语言中的变量没有类型，不需提前声明，可以直接创建并赋值，正所谓即开即用。如下图，创建文件名为 variables.txt 并在文件内添加如下内容：

A screenshot of a text editor window titled 'kali@kali: ~'. The menu bar includes 'File', 'Actions', 'Edit', 'View', and 'Help'. The script content is as follows:

```
#!/bin/bash
STR="hello world!"
echo $STR
echo STR
```

运行脚本输出，查看输出内容：

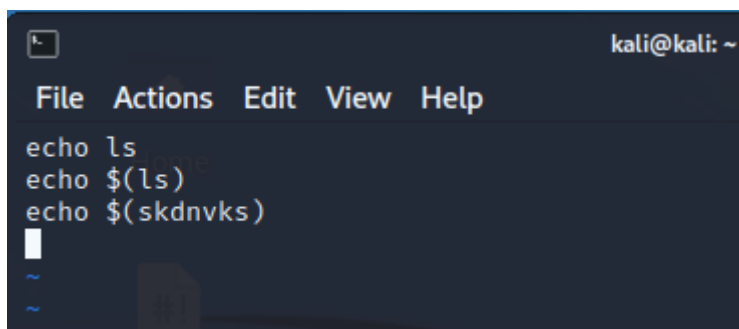
A screenshot of a terminal window showing the execution of the script. The prompt is '(kali@kali)-[~]'. The command '\$ bash variables.txt' has been entered, and the output is:

```
hello world!
STR
```

看到区别了吗，在脚本定义了一个变量叫 STR，并对其赋值为 hello world!，然后接下来两行分别进行输出，可见在bash中变量的使用需要使用到 \$ 符号。

第二个例子

在创建一个 demo.txt 文件，对文件内输入以下内容：

A screenshot of a text editor window titled 'kali@kali: ~'. The menu bar includes 'File', 'Actions', 'Edit', 'View', and 'Help'. The script content is as follows:

```
echo ls
echo $(ls)
echo $(skdnvks)
```

将脚本保存后运行，显示结果如下：

```
(kali㉿kali)-[~]
$ vim demo.txt

(kali㉿kali)-[~]
$ bash demo.txt
ls
demo.txt Desktop Documents Downloads Music Pictures pipdemo.txt Public stderr1.txt st
derr2stdout.txt stderrANDstdout2file.txt stderr.txt stdout2stderr.txt Templates varib
les.txt Videos
demo.txt: line 3: skdnvks: command not found

(kali㉿kali)-[~]
$ ls
demo.txt      Music          stderr1.txt    stdout2stderr.txt
Desktop       Pictures       stderr2stdout.txt Templates
Documents     pipdemo.txt   stderrANDstdout2file.txt variables.txt
Downloads     Public        stderr.txt     Videos
```

发现了吗，三个 echo 分别输出了三部分内容。可以明显看出，echo ls 的结果是把 ls 这个字符串打印在屏幕上；而 echo \$(ls) 是把在命令行里执行 ls 命令后输出的结果打印在了屏幕上。最后一个 echo 提示找不到这个指令。

因此，不难发现，在 bash 脚本中，\$ 符号的作用就是对其后面的内容进行解析执行，它先判断后面的内容是不是一个变量或者命令，如果是就执行并输出其执行结果，如果不是则报无此命令提示。

第三个例子

在对上面两个例子的掌握之后，这个例子就显得如鱼得水，毫无难度了，依然创建一个 demo1.txt 文件，并在其中输入以下内容：

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -cZPf $OF /home/kali/me
```

然后去看看 /var 目录下有什么东西发生了变化：

```
(kali㉿kali)-[/var]
$ ls -l
total 372
drwxr-xr-x  2 root root    4096 Nov  9 04:26 backups
drwxr-xr-x 15 root root    4096 Aug 31 02:35 cache
-rw-r--r--  1 root root 331813 Nov 19 08:51 home-backup.tgz
drwxr-xr-x 69 root root    4096 Aug 31 02:38 lib
drwxrwsr-x  2 root staff  4096 Jul 26 09:31 local
lrwxrwxrwx  1 root root      9 Aug  8 06:02 lock -> /run/lock
drwxr-xr-x 17 root root    4096 Nov 21 08:03 log
drwxrwsr-x  2 root mail    4096 Aug  8 06:02 mail
-rw-r--r--  1 root root    253 Nov 21 09:44 my-backup-20221121.tgz
drwxr-xr-x  2 root root    4096 Aug  8 06:02 opt
lrwxrwxrwx  1 root root      4 Aug  8 06:02 run -> /run
drwxr-xr-x  5 root root    4096 Aug  8 06:08 spool
drwxrwxrwt  7 root root    4096 Nov 21 09:39 tmp
drwxr-xr-x  3 root root    4096 Aug  8 06:05 www
```

发现了吗，/home/kali/me 文件被压缩成了一个 .tgz 结尾的压缩包，并被保存在了 /var 目录下，并且该压缩文件的名字中 20221121 就是 bash 脚本文件中 \$(date +%Y%m%d) 的执行结果。

```
(kali㉿kali)-[~]  
$ $(date +%Y%m%d)  
20221121: command not found
```

解析后发现它并不是一个指令，所以报了找不到指令提示，但是这不影响文件命名。

这里给大家留一个思考题：为什么 `command not found` 没有出现在文件命名中？

提示：与 `stdout` 和 `stderr` 有关。

局部变量

变量根据其作用域范围，才有了局部变量这一说法，局部变量只能由声明它的函数或块中访问。

在 Bash 中，局部变量的创建以 `local` 为关键字，其特性与其他编程语言基本是一样一样滴。一个例子了解一下：

创建一个 `demo2.sh`，并保存以下内容在文件里：

```
kali@kali: ~  
File Actions Edit View Help  
#!/bin/bash  
HELLO=Hello  
function hello(){  
    local HELLO=World  
    echo $HELLO  
}  
echo $HELLO  
hello  
echo $HELLO  
~ error near unexpected token `local'  
~ local HELLO=world'
```

为 `demo2.sh` 文件添加执行权限，然后运行查看输出：

```
(kali㉿kali)-[~]  
$ chmod +x demo2.sh  
  
(kali㉿kali)-[~]  
$ ./demo2.sh  
Hello  
World  
Hello
```

看见了吗，在脚本里一共使用了三次 `echo $HELLO` 命令，其中两次是在 `hello` 函数外使用，第二次是在 `hello` 函数内使用，从输出内容发现第二次调用时在函数内部创建了同名的局部变量，因此这一次的调用输出了不一样的值。因此关于局部变量的使用其实就遵循一个规则：在该变量被解析时总会从使用它在最近的一个作用域内被赋予的值。

可能遇到的问题和疑惑

- 什么玩意儿作用域，局部变量这俩是啥意思？

作用域这个概念可能有点生涩，简单说就是一个所属权的概念。

一个文件就是一个大作用域，它规定了这个文件里定义的东西是属于这个文件的，用以区别在一个项目中的其他文件里的同名变量；

文件里每一对大括号 `{ }` 包裹的范围就是一个局部作用域，在局部作用域里面创建的变量就叫做局部变量，它规定了这一对大括号里定义的东西是属于这个大括号所有的，用以区别其他大括号里的同名变量；

那要是一对大括号里要是还有一对大括号包裹着 `{ { } }` 呢？这就像俄罗斯套娃一样了，一样一样滴理解即可。

感觉听起来好麻烦啊，能说再直白一点吗？额……其实说白了作用域和局部变量这些说法就是为了解决遇到同名变量具体该解析那一块内存的问题。

试想一个场景，公司一个项目开发团队有 5 个人，假设这 5 个人在两个月里面为这个项目一共写了 5w 行代码，这 5w 行代码被写在了不同的 100 个文件里。

我们知道其实变量就是工程师们操作内存的一个手段，而函数则是工程师们对内存内数据快捷处理的一个手段。那在一个 5w 行代码的项目里面一定涉及到了大量变量的创建和使用，难免一个不小心其中一个工程师就创建了好几个同名的变量，或者两三个工程师创建了几个相同名字的变量，那计算机在遇到同名的变量时怎么分辨应该去哪一块内存拿这个变量的数据呢？

这个问题就是局部变量和作用域概念解决的问题，答案就是：当前被解析的这个变量在它最近的大括号里这个变量有没有被创建并赋值？如果有，那这个变量就是这个值；如果没有，找这个变量所在大括号外面的大括号有没有对这个变量进行创建赋值？如果有，那这个变量就是这个值；如果没有，找这个变量所更大一层大括号包裹的范围有没有对这个变量进行创建赋值，要是外面没有大括号了呢？找整个文件范围里有没有这个变量的赋值，要是这个文件里也没有呢？找这个文件引入的其他文件有没有这个变量的赋值。

就是这个逻辑，从近到远，由内而外一步步的找这个变量的值。

- 什么是 .sh 文件？

sh 文件是脚本文件，一般都是 bash 脚本，我们可以使用 sh 命令运行“sh xxx.sh”或者直接输入文件名进行运行。这与 C 语言的“.c”文件，python 的“.py”是一样的。

【第五节】条件判断

条件判断和其他语言英语，就是 if 那一套语法。

在这里说一个题外话，纠正一下很多人思维的一个误区：条件不属于循环，条件判断的大括号跟 break 和 continue 无关。

在 Bash 里面条件判断有三种形式：

```
if ... then ... fi

if ... then ... else ... fi

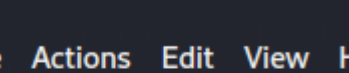
if ... then ... elif ... then ... else ... fi
```

每一个 if 后面都是判断语句，每一个 then 与 else 之后都是判断成立执行的代码与判断不成立执行的代码。

流程

第一个例子

创建 ifthen.sh 并输入一下代码：



The screenshot shows a terminal window with a dark background. At the top right, the prompt 'kali@kali: ~' is visible. Below the menu bar (File, Actions, Edit, View, Help), the command '#!/bin/bash' is entered. The script body consists of an if statement: 'if ["foo" = "foo"]; then' followed by an indented 'echo it is equal' and a 'fi' statement. A cursor is positioned at the end of the 'fi' line.

```
kali@kali: ~  
File Actions Edit View Help  
#!/bin/bash  
  
if [ "foo" = "foo" ]; then  
    echo it is equal  
fi  
~
```

代码内容对两个字符串进行了基本判断，如果相等，则输出 `it is equal` 字符串。`fi` 代表 `then` 后面的语句结束

注意，bash脚本对规范很严格，像图片里一样，字母与符号之间该有的空格一个都不能少！否则就会得到如下让人很懵逼的报错信息：

```
kali@kali: ~  
File Actions Edit View Help  
File Actions Edit View Help  
1 if 和 then 之间没打空格  
—(kali@kali)-[~]  
$ bash ifthen.sh  
ifthen.sh: line 3: syntax error near unexpected token `then'  
ifthen.sh: line 3: `if["foo"]="foo"]; then'  
  
—(kali@kali)-[~]  
$ bash ifthen.sh  
ifthen.sh: line 3: [foo=foo]: command not found  
  
—(kali@kali)-[~]  
$ bash ifthen.sh  
ifthen.sh: line 3: [1=1]: command not found  
  
—(kali@kali)-[~]  
$ bash ifthen.sh  
ifthen.sh: line 3: [foo=foo]: command not found  
  
—(kali@kali)-[~]  
$ bash ifthen.sh  
ifthen.sh: line 3: [foo=foo]: command not found  
  
—(kali@kali)-[~]  
$ bash ifthen.sh  
it is equal  
  
—(kali@kali)-[~]  
$
```

输入命令运行代码，可以得到以下理想中的结果：

```
(kali㉿kali)-[~]  
$ bash ifthen.sh  
it is equal
```


第二个例子

创建 ifthenelse.sh 并输入以下代码：

```
kali@kali: ~  
File Actions Edit View Help  
#!/bin/bash  
if [ "foo" = "foo1" ]; then  
    echo it is equal  
else  
    echo it is unequal  
fi  
~  
~  
~
```

代码的内容是对 foo 和 foo1 进行比较，根据比较结果进行不同的输出。运行代码后可以获得以下结果：

```
(kali@kali)-[~]  
$ bash ifthenelse.sh  
it is unequal
```

第三个例子

创建 ifwithvar.sh 并输入以下代码：

```
kali@kali: ~  
File Actions Edit View Help  
#!/bin/bash  
var1="PHP"  
var2="C++"  
if [ "$var1" = "$var2" ]; then  
    echo it is equal  
else  
    echo its is unequal  
fi  
~  
~  
~
```

这时 if 中的判断语句也可以去掉，像如下输入：

```
kali@kali: ~  
File Actions Edit View Help  
#!/bin/bash  
var1="PHP"  
var2="C++"  
if [ $var1 = $var2 ]; then  
    echo it is equal  
else  
    echo its is unequal  
fi  
~  
~  
~
```

运行代码后可以得到一样的结果，如下图：

```
(kali㉿kali)-[~]
└─$ bash ifwithvar.sh
its is unequal
└─$ bash ifwithvar.sh
its is unequal
```

可能遇到的问题和疑惑

- 一点点比较都要写这么多，有没有简单的比较方法？

有！好好参悟一下这一行代码：

```
[ 3 -gt 2 ]&& echo 1 ||echo 0
```

这一行代码直接在命令行就能完成运行，不需要单独创建文本。代码的意思就是 如果方括号里的语句成立，就输出1，否则输出0。

以上内容只对 Bash 条件判断中的 if 语法进行了一定演示，实际上，bash 的判断语句十分丰富，并且有很多种类，这里埋一颗种子，各位师傅以后遇到了在深入学习也不错。

【第六节】循环：for、while、until

流程

第一个例子

for: for循环与其他编程语言略有不同。基本上，让我们在字符串中对一个个单词进行迭代（或者说是遍历）。

创建 for.sh 并输入以下代码：

```
kali@kali: ~
File Actions Edit View Help
#!/bin/bash
for i in $( ls |grep txt ); do
    echo item: $i
done
```

在第二行：声明*i*为变量，它将接受\$(ls |grep txt) 中包含的不同值。

第三行：意思是输出当前的变量*i*的值如果需要。实际上，在 do 到done 之间的代码都是对当前 变量*i* 的值进行操作的代码；这在当前例子中是输出其值，其实也可以根据实际需要做更多更复杂的事情。

第四行：“done”代表当前 \$i 值的相关操作代码已完成。

分别运行脚本和直接在命令行运行脚本中遍历的命令，可以看见就是对命令执行输出的文本中的单词进行遍历输出：

```
(kali㉿kali)-[~]
└─$ bash for.sh
item: demo1.txt
item: demo.txt
item: pipdemo.txt
item: stderr1.txt
item: stderr2stdout.txt
item: stderrANDstdout2file.txt
item: stderr.txt
item: stdout2stderr.txt
item: variables.txt

(kali㉿kali)-[~]
└─$ ls |grep txt
demo1.txt
demo.txt
pipdemo.txt
stderr1.txt
stderr2stdout.txt
stderrANDstdout2file.txt
stderr.txt
stdout2stderr.txt
variables.txt
```

再看另一种类型的 for 循环写法，创建 for1.sh 并输入以下代码：

```
kali@kali: ~
File Actions Edit View Help
#!/bin/bash
for i in `ls |grep txt`;
do
    echo $i
done
~
~
```

这个和上面那种写法差不多，输出结果是一样的，不过好像这种看起来好像更好看一些。

第二个例子

while：如果控制表达式为真，while将执行一段代码，只有当它为假（或在执行的代码中发现显式中断）时，while才会停止。

创建一个 while.sh 文件，并输入以下代码

```
kali@kali: ~
File Actions Edit View Help
#!/bin/bash
counter=0
while [ $counter -lt 10 ]; do
    echo the counter is $counter
    let counter+=1
done
~
~
```

代码的意思就是 设置一个变量叫 counter 并设置初始值为0，然后每次循环和 10 比较，如果小于 10 就输出变量的值然后把变量的值加 1 然后进行下一次循环，直到不满足判断就停止。

第三行：中括号里的内容就是上一节没讲的 bash 语法里其他判断形式。`-lt` 的意思就是 小于，如果 `$counter` 的值小于 10，则条件成立。

第五行：`let` 的意思就是为变量赋值的一个关键字。如果不加这个 `let` 的话，程序也不会报错，不过会输出其他结果，可以思考一下为什么。

运行程序后会得到以下输出：

```
(kali㉿kali)-[~]
$ bash while.sh
the counter is 0
the counter is 1
the counter is 2
the counter is 3
the counter is 4
the counter is 5
the counter is 6
the counter is 7
the counter is 8
the counter is 9
```

第三个例子

`until`：除非代码在控制表达式求值为 `false` 时执行，否则直到循环几乎等于 `while` 循环。

创建 `until.sh` 文件，并输入以下代码：

```
kali@kali: ~
File Actions Edit View Help
#!/bin/bash
counter=20
until [ $counter -lt 10 ]; do
    echo counter $counter
    let counter-=1
done
~
~
```

代码的意思就是先执行 `do` 后面的语句，然后循环完了一次在条件判断里面比对一次，当符合条件了就结束循环。

运行代码后，可以得到以下结果：

```
(kali㉿kali)-[~]
$ bash until.sh
counter 20
counter 19
counter 18
counter 17
counter 16
counter 15
counter 14
counter 13
counter 12
counter 11
counter 10
```

其实像这类的判断、循环等语句各种编程语言都是千篇一律的，触类旁通，懂了任何一种，其他的就好理解了。

可能遇到的问题和疑惑

- `let` 是个啥玩意儿?

`let` 命令是 BASH 中用于计算的工具，用于执行一个或多个表达式，变量计算中不需要加上 `$` 来表示变量。如果表达式中包含了空格或其他特殊字符，则必须引起来。简而言之就是涉及到算数运算的最好添上这个关键字。

- `-lt` 是啥玩意儿?

在 `bash` 里面，除了可以用等号进行判断，也可以用符号判断，如例子里面的 `-lt` 其实是 `less than` 的首字母缩写，就是小于的意思，意思是前者小于后者。

条件判断语法除了 `-lt` 还有以下：

字符串类	
<code>str1 = str2</code>	当两个串有相同内容、长度时为真
<code>str1 != str2</code>	当串 <code>str1</code> 和 <code>str2</code> 不等时为真
<code>-n str1</code>	当串的长度大于0时为真(串非空)
<code>-z str1</code>	当串的长度为0时为真(空串)
<code>str1</code>	当串 <code>str1</code> 为非空时为真

数字类	
<code>int1 -eq int2</code>	两数相等为真
<code>int1 -ne int2</code>	两数不等为真
<code>int1 -gt int2</code>	<code>int1</code> 大于 <code>int2</code> 为真
<code>int1 -ge int2</code>	<code>int1</code> 大于等于 <code>int2</code> 为真
<code>int1 -lt int2</code>	<code>int1</code> 小于 <code>int2</code> 为真
<code>int1 -le int2</code>	<code>int1</code> 小于等于 <code>int2</code> 为真

文件类	
-r file	用户可读为真
-w file	用户可写为真
-x file	用户可执行为真
-f file	文件为正规文件为真
-d file	文件为目录为真
-c file	文件为字符特殊文件为真
-b file	文件为块特殊文件为真
-s file	文件大小非0时为真
-t file	当文件描述符(默认为1)指定的设备为终端时为真

逻辑类	
-a	与
-o	或
!	非

具体使用也都差不多，不妨像上一节一样，各位师傅在以后实际生产中，需要用到的时候再深入研究也可以嘞！

【第七节】函数

好像任何编程语言都有 函数 这个功能吧？函数的意义就是让某些处理同一个事情的代码归类在一起，不同的代码做不同的事情，做不同事情的代码聚集在一个地方，从而让整个代码结构不至于松松散散的看起来很凌乱。

流程

第一个例子

创建一个 function.sh 文件并输入以下内容：

```

kali@kali: ~
File  Actions  Edit  View  Help
1  #!/bin/bash
2  function quit (){
3      exit
4  }
5  function hello {
6      echo hello!
7  }
8  hello
9  quit
10
11 echo foo
12
13

```

quit 函数：第 2 ~ 4 行就是 quit 函数的内容，这个函数的作用就是推出程序脚本执行，当脚本执行到 exit 时，就会停止并退出。

hello 函数：第 5 ~ 7 行就是 hello 函数的内容，函数的作用就如大括号内所示，打印 hello! 字符串到屏幕上。

运行代码，可以得到以下内容：

```
(kali㉿kali)-[~]  
$ bash function.sh  
hello!
```

可以看到：echo foo 没有执行把相应单词输出在屏幕上。而这正式因为 exit 生效了造成的。

函数基本形式为：

```
[ function ] name () {}
```

前面的 function 单词可以省略不写，如果写上了 function 单词那么后面的小括号可以省略，就如上图在 hello 函数中就没有写小括号。

第二个例子

这个例子将要揭示bash函数的传参过程，创建一个funcwitharg.sh 文本并输入以下内容：

```
kali@kali: ~  
File Actions Edit View Help  
1 #!/bin/bash  
2 function quit {  
3     exit  
4 }  
5 function e {  
6     echo $1  
7 }  
8  
9 e hello  
10 e world  
11 quit  
12 echo foo  
13
```

这个代码和上述代码差不多，唯一的区别就是在 e 函数里面的 echo \$1，这个 \$1 就是为了接受传进来的参数用的，例如在第 9 行和第 10 行，函数 e 在调用中接收到了后面紧跟的参数，从而使得参数的内容可以参与函数运行过程。

运行上述代码会出现以下内容：

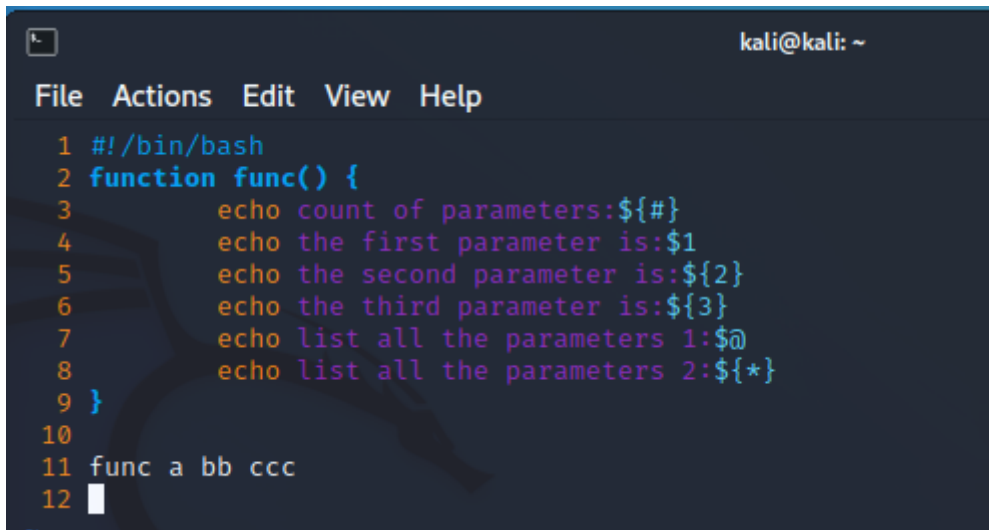
```
(kali㉿kali)-[~]  
$ bash funcwitharg.sh  
hello  
world
```

可能遇到的问题和疑惑

- 传参那个是个什么原理？

很简单，若函数带有参数，那么参数个数，各个参数内容，所有参数，都有特殊变量表示。见下图，创建 testarg.sh，输入以下代码：

（先说明一点，图中表示 **引用变量的大括号** 是可写可不写的，如图第 3、5、6、8 行便在引用变量周围加上了大括号，而第 4、7 行则没添加括号。



```
kali@kali: ~  
File Actions Edit View Help  
1 #!/bin/bash  
2 function func() {  
3     echo count of parameters:${#}  
4     echo the first parameter is:$1  
5     echo the second parameter is:${2}  
6     echo the third parameter is:${3}  
7     echo list all the parameters 1:$@  
8     echo list all the parameters 2:${*}  
9 }  
10  
11 func a bb ccc  
12
```

代码意思是：

第三行：传入参数总数是：

第四行：传入的第一个参数的值是：

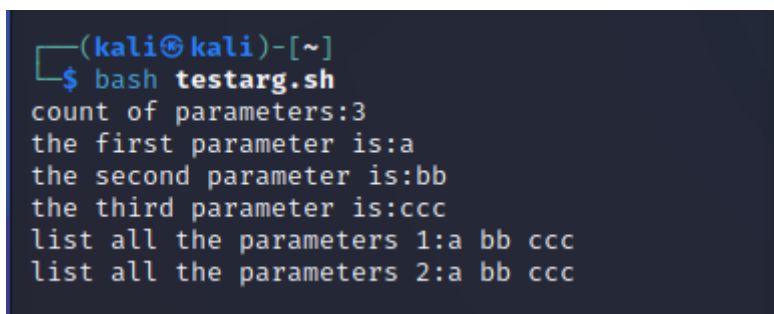
第五行：传入的第二个参数的值是：

第六行：传入的第三个参数的值是：

第七行：所有传入参数的值是：

第八行：所有传入参数的值是（第二种写法）：

运行代码，会得到以下结果：



```
(kali@kali)-[~]  
$ bash testarg.sh  
count of parameters:3  
the first parameter is:a  
the second parameter is:bb  
the third parameter is:ccc  
list all the parameters 1:a bb ccc  
list all the parameters 2:a bb ccc
```

其实在 bash 编程的函数中也还有很多的技巧和方法，为了保持愉悦的心情，各位师傅可以在未来实际生产中遇到了再深入探讨学习也不错哦。

【第八节】交互操作

流程

第一个例子

使用 select 制作简单菜单

创建 userinterface.sh 文件，在文件中输入以下内容：

```
kali@kali: ~  
File Actions Edit View Help  
1 #!/bin/bash  
2 OPTIONS="quit hello"  
3 select opt in $OPTIONS; do  
4     if [ "$opt" = "quit" ]; then  
5         echo done  
6         exit  
7     elif [ "$opt" = "hello" ]; then  
8         echo hello world  
9     else  
10        echo bad option  
11    fi  
12 done  
13
```

代码的意思：先设定了一个 OPTIONS 菜单，里面的词语代表相应功能的名称，之后像循环一样每次等待用户选择，根据选择进行判断，在本例子中，如果用户输入的是 quit 则退出程序；如果用户输入的是 hello，则输出 hello world 到屏幕上。然后等待用户的新一轮选择，直到用户选择退出程序停止执行。

运行代码后会获得如下菜单：

```
(kali@kali)-[~]  
$ bash userinterface.sh  
1) quit  
2) hello  
#? █
```

输入相应功能序号即可跳转执行相应功能：

```
(kali@kali)-[~]  
$ bash userinterface.sh  
1) quit  
2) hello  
#? 2  
hello world  
#? █
```

一个简单的菜单选择功能就此完成

第二个例子

命令行交互操作对目录打包压缩

创建脚本 usingcammand.sh 文件并输入一下代码：

```
kali@kali: ~  
File Actions Edit View Help  
1 #!/bin/bash  
2 if [ -z $1 ]; then  
3     echo usage:$0 directory  
4     exit  
5 fi  
6 SRCD=$1  
7 TGT="/var/backups/"  
8 OF=home-$(date +%Y%m%d).tgz  
9 sudo tar -cZf $TGT$OF $SRCD  
10
```

代码意思

`-z $1`：就是看是否存在该参数，这个判断语句的意思就是如果当前没接收到参数就提示没有当前目录并退出。

之后就是将参数指定的目录传进变量 `SRCD`，同时将压缩文件的目录定义在 `TGT` 变量里，之后通过 `OF` 变量对每一个压缩文件分别命名。

运行代码时不添加参数，输出如下：

```
(kali@kali)-[~]  
$ bash usingcommand.sh  
usage:usingcommand.sh directory
```

运行代码文件时添加参数后效果如下：

```
(kali@kali)-[~]  
$ bash usingcommand.sh test.sh  
[sudo] password for kali:
```

因为需要访问高权限目录，输入密码后可在对应目录下看到生成的压缩文件：

```
(kali@kali)-[~]  
$ sudo ls /var/backups |grep home  
home-20221124.tgz
```

学到这里我终于知道 Linux 的命令行工具是怎么接受参数的了呜呜呜。原来就是这个原因。

【第九节】其他值得一学的

读取用户输入

创建一个 `read.sh` 文件，并在文件中输入以下内容：


```
kali@kali: ~  
File Actions Edit View Help  
1 #!/bin/bash  
2 echo input your name:  
3 read NAME  
4 echo hi $NAME!  
~  
~  
~
```

运行文件后，可以看到以下输出：

```
(kali@kali)-[~]  
$ bash read.sh  
input your name:  
█
```

输入参数值后可以得到内容：

```
(kali@kali)-[~]  
$ bash read.sh  
input your name:  
shungli923  
hi shungli923!
```

同样，read也可以用于读取多个参数，创建 read1.sh 文件，并输入以下内容：

```
kali@kali: ~  
File Actions Edit View Help  
1 #!/bin/bash  
2 echo enter your nicknames  
3 read NN1 NN2  
4 echo $NN1 and $NN2  
5 █  
~  
~  
~
```

运行脚本文件并填写内容后可以获取以下输出：

```
(kali@kali)-[~]  
$ bash read1.sh  
enter your nicknames  
shungli shungli923  
shungli and shungli923
```

这便是 read 方法的使用。

算数运算

相信很多人已经发现了，直接在脚本文件里输入 `echo 1+1` 输出的值不是 2，而是一个 1+1 的字符串，怎么可以输出算数运算的结果呢？有两种方法

一：双括号

至于为什么是双括号，我愿意贴出图片，师傅们自行感悟

```

(kali㉿kali)-[~]
$ echo 1+1
1+1

(kali㉿kali)-[~]
$ echo $((1+1))
2

(kali㉿kali)-[~]
$ echo $(1+1)
1+1: command not found

(kali㉿kali)-[~]
$ echo $(((1+1)))
2

```

看图片，也就是说至少都得两个括号才行。

二：中括号

如图：

```

(kali㉿kali)-[~]
$ echo ${1+1}
2

(kali㉿kali)-[~]
$ echo ${[1+1]}
zsh: bad base syntax

```

在实际操作中，如果尝试多一点就会发现上面这两种方法只能用于整数运算，对于分数毫无抵抗力，那怎么进行分数运算呢？

```

(kali㉿kali)-[~]
$ echo ${1/4}
0

(kali㉿kali)-[~]
$ echo $((1/4))
0

```

在 bash 里，可以通过 bc 指令来运算分数，具体用法见下，在命令行输入如下代码：发现没安装，需要安装。

```

(kali㉿kali)-[~]
$ echo 1/4|bc -l -y
Command 'bc' not found, but can be installed with:
sudo apt install bc
Do you want to install it? (N/y)

```

安装过程中却发现不能安装，报这个错：

```
(kali㉿kali)-[~]
└─$ sudo apt install bc
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
E: Unable to locate package bc
```

这个错误说明找不到安装包，这个时候需要换一个源 也就是 sources.list，进入以下文件：`/etc/apt/sources.list`，在网上随便找一个中科大的，清华的，阿里的源替换了即可，然后命令行输入 `apt-get update`，等待之后就可以进行安装了。

```
(root㉿kali)-[~]
└─# vim /etc/apt/sources.list

(root㉿kali)-[~]
└─# apt-get update
Get:1 http://mirrors.ustc.edu.cn/kali kali-rolling InRelease [30.6 kB]
Get:2 http://mirrors.ustc.edu.cn/kali kali-rolling/non-free Sources [132 kB]
Get:3 http://mirrors.ustc.edu.cn/kali kali-rolling/contrib Sources [74.3 kB]
Get:4 http://mirrors.ustc.edu.cn/kali kali-rolling/main Sources [15.1 MB]
Get:5 http://mirrors.ustc.edu.cn/kali kali-rolling/main amd64 Packages [18.9 MB]
Get:6 http://mirrors.ustc.edu.cn/kali kali-rolling/main amd64 Contents (deb) [43.4 MB]
Get:7 http://mirrors.ustc.edu.cn/kali kali-rolling/non-free amd64 Packages [237 kB]
Get:8 http://mirrors.ustc.edu.cn/kali kali-rolling/non-free amd64 Contents (deb) [901 kB]
Get:9 http://mirrors.ustc.edu.cn/kali kali-rolling/contrib amd64 Packages [111 kB]
Get:10 http://mirrors.ustc.edu.cn/kali kali-rolling/contrib amd64 Contents (deb) [161 kB]
Fetched 79.0 MB in 28s (2,809 kB/s)
Reading package lists... Done
```

安装后再次输入以下指令：

```
(kali㉿kali)-[~]
└─$ echo 1/4|bc -l
.2500000000000000000000
```

至于为什么小数点后面有这么多位的 0，怎么把这么多的 0 给消除掉，欢迎师傅们有机会深入探索！

使用 sed 和 awk

这俩是什么的有多强大，相信各位师傅都略有耳闻，此处权当抛砖引玉，若要深入研究，欢迎查看官方文档。

sed:

sed 是一个非交互式编辑器。可以实现不用通过在屏幕上移动光标来更改文件，而是使用 sed 指令，再加上要编辑的文件名来对文件内容进行操作。也可以将sed描述为过滤器。

举一个例子：

```
(root㉿kali)-[/home/kali]
└─# sed 's/to_be_replaced/replaced/g' /tmp/dummy
replaced
```

sed 将字符串“to_be_replaced”替换为字符串“replaced”，并从/tmp/dmmy文件中读取。结果将发送到 stdout（通常是控制台），但您也可以在上方的行末尾使用重定向，以便sed将输出发送到重定向文件。

举两个例子：

```
(root@kali)-[/home/kali]
# sed 4,6d demo333.txt

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16

(root@kali)-[/home/kali]
#
```

sed 显示除第 4 行至第 6 行之外的所有行。但原始文件不会被此命令更改。

awk:

awk 是处理文本文件的一个应用程序，它依次处理文件的每一行，并读取里面的每一个字段。对于日志、CSV 那样的每行格式相同的文本文件，awk 的方便几乎无与伦比。

创建一个示例文件 demo555.txt 包含以下内容：

```
kali@kali: ~
File Actions Edit View Help
1 test123
2 test
3 tteesstt
4
~
~
~
~
```

输入以下命令可以得到结果：

```
(root@kali)-[/home/kali]
# awk '/test/ {print}' demo555.txt
test123
test

(root@kali)-[/home/kali]
# awk '/test/ {i=i+1} END {print i}' demo555.txt
2
```

师傅们这里可以根据输入命令和输出结果来推敲一下命令的意思。

关于 awk 的用法和骚操作还有很多很多，此处仅作一个引子，有兴趣的师傅在实际需要的时候在深入了解也不错哦！

【结尾】总结

整个笔记的定位保证能在生活中的大部分场景满足大部分需求，对于深入的理论和少见的用法并没有过多的介绍，一是避免太过于枯燥，而是避免太多的理论性文字令人头大。由于鄙人才疏学浅，经验欠缺，如有错误还请师傅们谅解，再次诚邀各位师傅对本笔记进行纠错和内容丰富！

附 PDF 下载链接:

Github 下载链接: <https://github.com/shungli923/WowBigBug/tree/main>/代码笔记

百度云盘下载地址: <https://pan.baidu.com/s/1VHGK5T8Sbvel9JBgL8uZYA> 提取码: mqjt

要是两个链接都用不上, 来我公众号后台回复 **【bash笔记】**, 获取 **该笔记最新版本 PDF 链接**



扫码关注我们

公众号| WowBigBug

致敬默默无闻的安全人