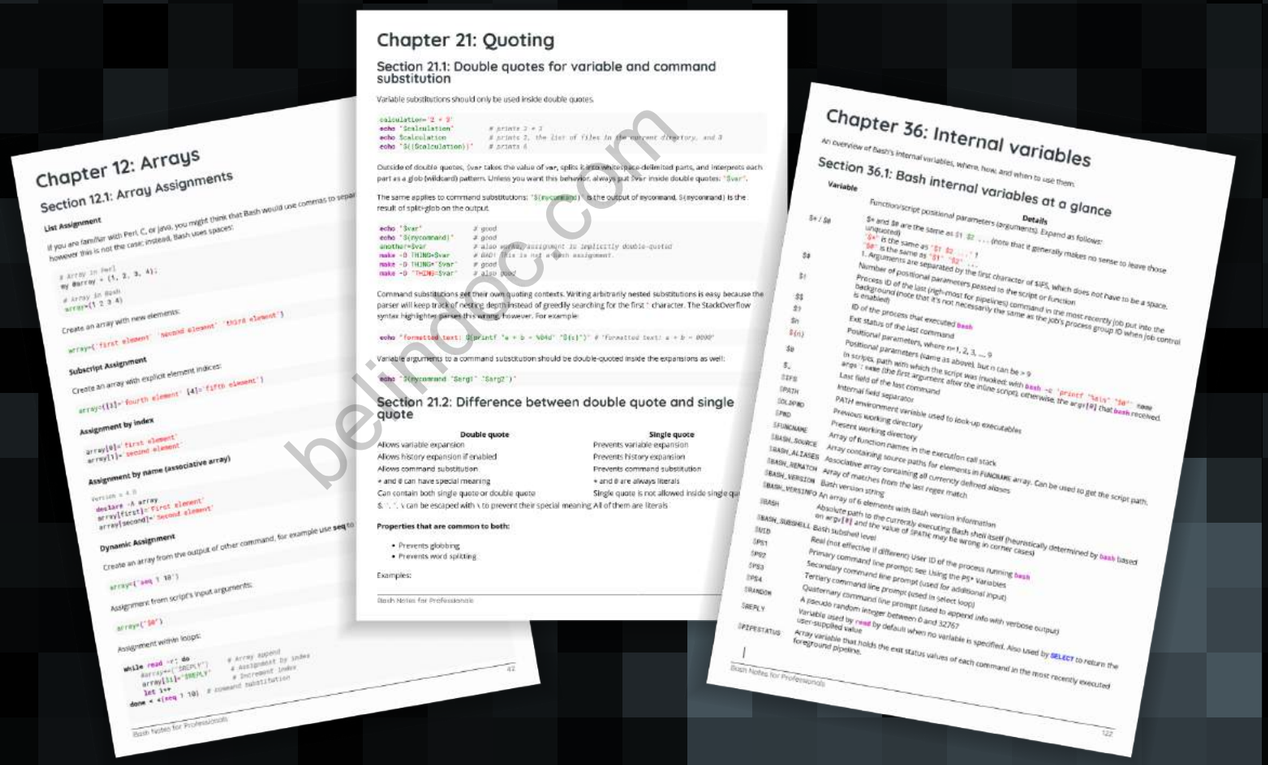
专业人员笔记

Bash

专业人员笔记

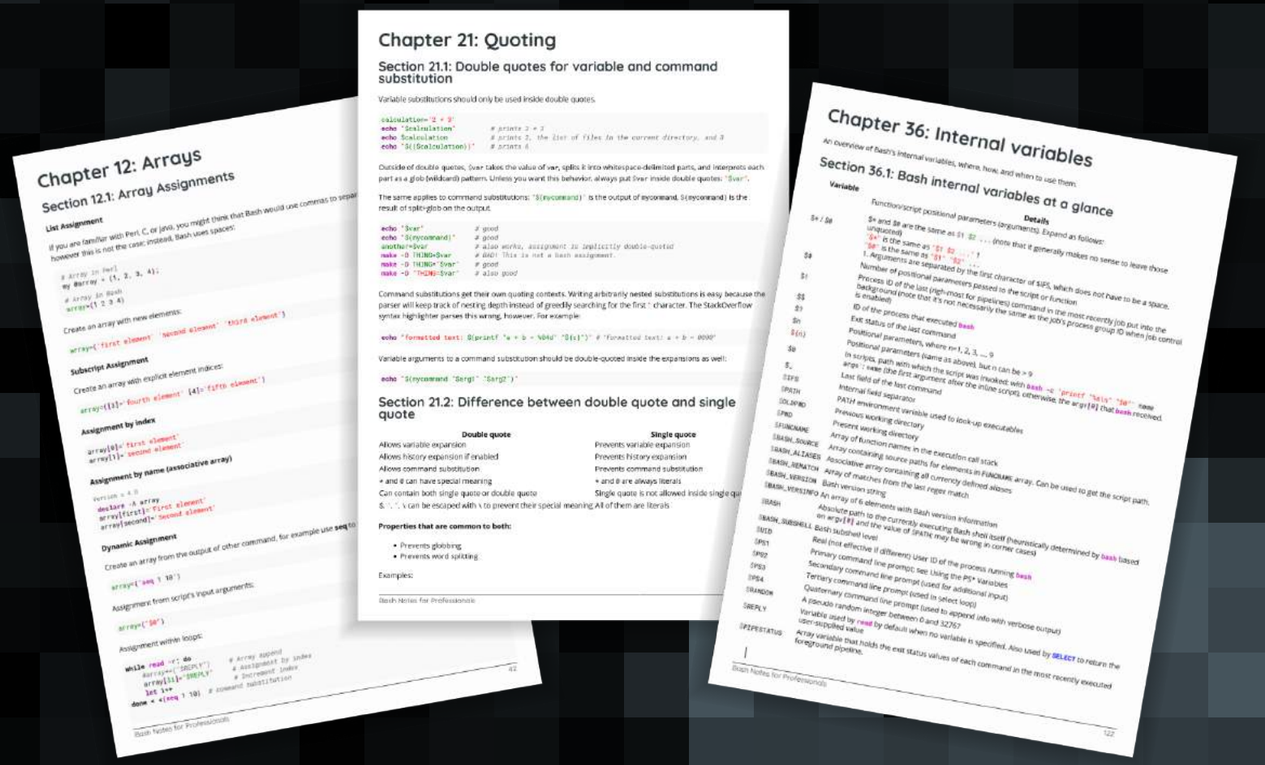


100多页

专业提示和技巧

Bash

Notes for Professionals



100+ pages

of professional hints and tricks

目录

Contents

关于

About

第1章：Bash入门

版本发布日期

0.99	1989-06-08
1.01	1989-06-23
2.0	1996-12-31
2.02	1998-04-20
2.03	1999-02-19
2.04	2001-03-21
2.05b	2002-07-17
3.0	2004-08-03
3.1	2005-12-08
3.2	2006-10-11
4.0	2009-02-20
4.1	2009-12-31
4.2	2011-02-13
4.3	2014-02-26
4.4	2016-09-15

第1.1节：你好，世界

交互式Shell

Bash shell通常用于交互式：它允许你输入和编辑命令，然后在你按下时执行它们
返回 关键。许多基于Unix和类Unix的操作系统使用Bash作为默认的shell
（尤其是Linux和macOS）。终端在启动时会自动进入一个交互式的Bash shell进程。

输出 Hello World 通过输入以下内容：

```
echo "Hello World"
#> Hello World  # 输出示例
```

备注

- 您可以通过在终端中直接输入 shell 的名称来更改 shell。例如：sh，bash 等。
- echo 是一个 Bash 内置命令，用于将接收到的参数写入标准输出。默认情况下，它会在输出末尾添加一个换行符。

非交互式Shell

Bash shell 也可以非交互式地从脚本中运行，使得 shell 不需要人工干预。
交互行为和脚本行为应当一致–这是Unix V7 Bourne shell及其衍生的Bash的重要设计考虑。因此，任何可以在命令行完成的操作都可以放入脚本文件中以便重用。

按照以下步骤创建一个Hello World脚本：

1. 创建一个名为 hello-world.sh 的新文件

Chapter 1: Getting started with Bash

Version Release Date

0.99	1989-06-08
1.01	1989-06-23
2.0	1996-12-31
2.02	1998-04-20
2.03	1999-02-19
2.04	2001-03-21
2.05b	2002-07-17
3.0	2004-08-03
3.1	2005-12-08
3.2	2006-10-11
4.0	2009-02-20
4.1	2009-12-31
4.2	2011-02-13
4.3	2014-02-26
4.4	2016-09-15

Section 1.1: Hello World

Interactive Shell

The Bash shell is commonly used **interactively:** It lets you enter and edit commands, then executes them when you press the Return key. Many Unix-based and Unix-like operating systems use Bash as their default shell (notably Linux and macOS). The terminal automatically enters an interactive Bash shell process on startup.

Output `Hello World` by typing the following:

```
echo "Hello World"
#> Hello World  # Output Example
```

Notes

- You can change the shell by just typing the name of the shell in terminal. For example: sh, bash, etc.
- echo is a Bash builtin command that writes the arguments it receives to the standard output. It appends a newline to the output, by default.

Non-Interactive Shell

The Bash shell can also be run **non-interactively** from a script, making the shell require no human interaction. Interactive behavior and scripted behavior should be identical – an important design consideration of Unix V7 Bourne shell and transitively Bash. Therefore anything that can be done at the command line can be put in a script file for reuse.

Follow these steps to create a `Hello World` script:

1. Create a new file called `hello-world.sh`

```
touch hello-world.sh
```

2. 通过运行 chmod +x hello-world.sh 使脚本可执行

3.添加以下代码：

```
#!/bin/bash
echo "Hello World"
```

第1行：脚本的第一行必须以字符序列 #! 开头，称为 shebang1。shebang 指示操作系统运行 /bin/bash，即 Bash shell，并将脚本路径作为参数传递给它。

例如 /bin/bash hello-world.sh

第2行: 使用 echo 命令将 Hello World 写入标准输出。

4. 使用以下任一方式从命令行执行 hello-world.sh 脚本：

   - ./hello-world.sh – 最常用且推荐的方式
   - /bin/bash hello-world.sh
   - bash hello-world.sh – 假设 /bin 在你的 $PATH 中
   - sh hello-world.sh

在实际生产环境中，你会省略 .sh 扩展名（这本身就容易误导，因为这是一个 Bash 脚本，而不是 sh 脚本），并且可能将文件移动到 PATH 中的某个目录，这样无论当前工作目录如何，都可以像系统命令 cat 或 ls 一样使用它。

常见错误包括：

1. 忘记给文件添加执行权限，即执行 chmod +x hello-world.sh，导致输出 ./hello-world.sh: 权限被拒绝。
2. 在 Windows 上编辑脚本，会产生 Bash 无法处理的错误换行符。

   一个常见的症状是：`command not found`，其中回车符将光标强制移到行首，覆盖了错误信息中冒号前的文本。

   可以使用 dos2unix程序修复该脚本。

   使用示例：`dos2unix hello-world.sh`

   dos2unix会直接编辑文件。

3. 使用sh ./hello-world.sh时，未意识到bash和sh是不同的shell，具有不同的特性（尽管由于Bash向后兼容，反过来的错误是无害的）。

   无论如何，仅依赖脚本的shebang行远比在每个脚本文件名前显式写**bash**或sh（或`python`、**`perl`**、**`awk`**、`ruby`等）要好得多。

   为了使脚本更具可移植性，常用的shebang行是使用*#!/usr/bin/env bash*
   而不是硬编码Bash的路径。这样，**/usr/bin/env**必须存在，但除此之外，**bash**只需


```
touch hello-world.sh
```

2. Make the script executable by running **chmod** `+x hello-world.sh`

3. Add this code:

```
#!/bin/bash
echo "Hello World"
```

Line 1: The first line of the script must start with the character sequence `#!`, referred to as *shebang*1. The shebang instructs the operating system to run **/bin/bash**, the Bash shell, passing it the script's path as an argument.

E.g. **/bin/bash** `hello-world.sh`

Line 2: Uses the **echo** command to write `Hello World` to the standard output.

4. Execute the `hello-world.sh` script from the command line using one of the following:

   - **./**`hello-world.sh` – most commonly used, and recommended
   - **/bin/bash** `hello-world.sh`
   - **bash** `hello-world.sh` – assuming **/bin** is in your `$PATH`
   - **sh** `hello-world.sh`

For real production use, you would omit the `.sh` extension (which is misleading anyway, since this is a Bash script, not a `sh` script) and perhaps move the file to a directory within your `PATH` so that it is available to you regardless of your current working directory, just like a system command such as **cat** or `ls`.

Common mistakes include:

1. Forgetting to apply execute permission on the file, i.e., **chmod** `+x hello-world.sh`, resulting in the output of **./**`hello-world.sh: Permission denied`.
2. Editing the script on Windows, which produces incorrect line ending characters that Bash cannot handle.

   A common symptom is : **command** `not found` where the carriage return has forced the cursor to the beginning of line, overwriting the text before the colon in the error message.

   The script can be fixed using the `dos2unix` program.

   An example use: `dos2unix hello-world.sh`

   `dos2unix` edits the file inline.

3. Using **sh** **./**`hello-world.sh`, not realizing that **bash** and `sh` are distinct shells with distinct features (though since Bash is backwards-compatible, the opposite mistake is harmless).

   Anyway, simply relying on the script's shebang line is vastly preferable to explicitly writing **bash** or `sh` (or `python` or **perl** or **awk** or `ruby` or...) before each script's file name.

   A common shebang line to use in order to make your script more portable is to use *#!/usr/bin/env bash* instead of hard-coding a path to Bash. That way, **/usr/bin/env** has to exist, but beyond that point, **bash** just

在你的PATH中即可。在许多系统上，/bin/bash不存在，你应该使用/usr/local/bin/bash或其他绝对路径；此更改避免了需要弄清楚这些细节。

1也称为sha-bang、hashbang、pound-bang、hash-pling。

第1.2节：使用变量的Hello World

创建一个名为`hello.sh`的新文件，内容如下，并使用`chmod +x hello.sh`赋予其可执行权限。

> 通过以下方式执行/运行：./hello.sh

```
#!/usr/bin/env bash

# 注意赋值操作符`=`两边不能有空格
whom_variable="World"

# 使用printf安全地输出数据
printf "Hello, %s" "$whom_variable"#> Hello,
 World
```

执行时，这将向标准输出打印Hello，World。

要告诉bash脚本的位置，需要非常具体地指向包含目录，通常使用`./`如果它是你的工作目录，其中`.`是当前目录的别名。如果你没有指定目录，**bash**会尝试在`$PATH`环境变量包含的目录之一中查找脚本。

以下代码接受一个参数$1，即第一个命令行参数，并以格式化字符串输出，格式为Hello,。

> 通过以下方式执行/运行：./hello.sh World

```
#!/usr/bin/env bash
printf "Hello, %s" "$1"#> Hell
o, World
```

需要注意的是$1必须用双引号括起来，而不是单引号。 "$1"会展开为第一个命令行参数，符合预期，而'$1'则会被当作字面字符串$1处理。

> **安全提示：**
> 请阅读Security implications of forgetting to quote a variable in bash shells以了解将变量文本放入双引号中的重要性。

第1.3节：带用户输入的Hello World

下面的代码会提示用户输入，然后将该输入作为字符串（文本）存储在变量中。该变量随后用于向用户显示消息。

needs to be on your `PATH`. On many systems, **`/bin/bash`** doesn't exist, and you should use **`/usr/local/bin/bash`** or some other absolute path; this change avoids having to figure out the details of that.

1 *Also referred to as sha-bang, hashbang, pound-bang, hash-pling.*

Section 1.2: Hello World Using Variables

Create a new file called `hello.sh` with the following content and give it executable permissions with **`chmod +x hello.sh`**.

> Execute/Run via: **`./`**`hello.sh`

```
#!/usr/bin/env bash

# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"

# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

This will print `Hello, World` to standard output when executed.

To tell bash where the script is you need to be very specific, by pointing it to the containing directory, normally with `./` if it is your working directory, where `.` is an alias to the current directory. If you do not specify the directory, **bash** tries to locate the script in one of the directories contained in the `$PATH` environment variable.

The following code accepts an argument $1, which is the first command line argument, and outputs it in a formatted string, following `Hello,`.

> Execute/Run via: **`./`**`hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

It is important to note that $1 has to be quoted in double quote, not single quote. `"$1"` expands to the first command line argument, as desired, while `'$1'` evaluates to literal string $1.

> **Security Note:**
> Read **Security implications of forgetting to quote a variable in bash shells** to understand the importance of placing the variable text within double quotes.

Section 1.3: Hello World with User Input

The following will prompt a user for input, and then store that input as a string (text) in a variable. The variable is then used to give a message to the user.

```
#!/usr/bin/env bash
echo  "你是谁？"
read name
echo "Hello, $name."
```

命令read在这里从标准输入读取一行数据到变量name。然后使用$name引用该变量，并用echo打印到标准输出。

示例输出：

```
$ ./hello_world.sh
你是谁？
马特
你好，马特。
```

这里用户输入了名字“马特”，这段代码用来输出你好，马特。。

如果你想在打印变量值时追加内容，可以在变量名周围使用大括号，如下例所示：

```
#!/usr/bin/env bash
echo  “你在做什么？”
read action
echo “你正在${action}。”
```

示例输出：

```
$ ./hello_world.sh
你在做什么？
睡眠
你正在睡觉。
```

这里当用户输入一个动作时，打印时会在该动作后面加上“ing”。

第1.4节：字符串中引用的重要性

引用对于bash中的字符串扩展很重要。通过引用，你可以控制bash如何解析和扩展你的字符串。

引用有两种类型：

- **弱引用**: *使用双引号："*
- **强引用**: *使用单引号：'*

如果你希望bash扩展你的参数，可以使用弱引用：

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

如果你不希望bash扩展你的参数，可以使用强引用：

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
```



```
#!/usr/bin/env bash
echo  "Who are you?"
read name
echo "Hello, $name."
```

The command `read` here reads one line of data from standard input into the variable `name`. This is then referenced using `$name` and printed to standard out using `echo`.

Example output:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Here the user entered the name "Matt", and this code was used to say `Hello, Matt.`.

And if you want to append something to the variable value while printing it, use curly brackets around the variable name as shown in the following example:

```
#!/usr/bin/env bash
echo  "What are you doing?"
read action
echo "You are ${action}ing."
```

Example output:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Here when user enters an action, "ing" is appended to that action while printing.

Section 1.4: Importance of Quoting in Strings

Quoting is important for string expansion in bash. With these, you can control how the bash parses and expands your strings.

There are two types of quoting:

- **Weak**: *uses double quotes: "*
- **Strong**: *uses single quotes: '*

If you want to bash to expand your argument, you can use **Weak Quoting**:

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

If you don't want to bash to expand your argument, you can use **Strong Quoting**:

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
```

```
#> Hello $world
```

你也可以使用转义来防止扩展：

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

有关初学者以外的更详细信息，你可以继续在这里阅读。

第1.5节：查看Bash内置命令的信息

```
help <command>
```

这将显示指定内置命令的Bash帮助（手册）页面。

例如，help unset 将显示：

```
unset: unset [-f] [-v] [-n] [name ...]
取消设置shell变量和函数的值及属性。

对于每个名称，移除对应的变量或函数。

选项：
-f    将每个名称视为一个 shell 函数
-v    将每个名称视为一个 shell 变量
-n    将每个名称视为名称引用，并取消设置变量本身
而不是它所引用的变量

无选项时，unset 首先尝试取消设置变量，如果失败，
则尝试取消设置函数。

某些变量无法取消设置；另请参见 `readonly`。

退出状态：
除非给出无效选项或名称为只读，否则返回成功。
```

要查看所有内置命令及其简短描述，请使用

```
help -d
```

第1.6节："调试"模式下的Hello World

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

参数-x 使你能够逐行跟踪脚本。这里有一个很好的例子：

```
$ cat hello.sh#!/
bin/bashecho
 "Hello World"
```



```
#> Hello $world
```

You can also use escape to prevent expansion:

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

For more detailed information other than beginner details, you can continue to read it here.

Section 1.5: Viewing information for Bash built-ins

```
help <command>
```

This will display the Bash help (manual) page for the specified built-in.

For example, `help unset` will show:

```
unset: unset [-f] [-v] [-n] [name ...]
Unset values and attributes of shell variables and functions.

For each NAME, remove the corresponding variable or function.

Options:
-f    treat each NAME as a shell function
-v    treat each NAME as a shell variable
-n    treat each NAME as a name reference and unset the variable itself
rather than the variable it references

Without options, unset first tries to unset a variable, and if that fails,
tries to unset a function.

Some variables cannot be unset; also see `readonly'.

Exit Status:
Returns success unless an invalid option is given or a NAME is read-only.
```

To see a list of all built-ins with a short description, use

```
help -d
```

Section 1.6: Hello World in "Debug" mode

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

The -x argument enables you to walk through each line in the script. One good example is here:

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
```

```
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World

expr: 非整数参数
```

上述提示的错误信息不足以追踪脚本；然而，使用以下方法能更好地帮助你定位脚本中的错误位置。

```
$ bash -x hello.sh
+ echo 你好，世界
你好，世界

+ adding_string_to_number=s
+ expr 5 + s
expr: 非整数参数
+ v=
```

第1.7节：处理命名参数

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
   --deploy) deploy="$2";;
--uglify) uglify="$2";;
   *) break;
 esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```



```
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World

expr: non-integer argument
```

The above prompted error is not enough to trace the script; however, using the following way gives you a better sense where to look for the error in the script.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World

+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
+ v=
```

Section 1.7: Handling Named Arguments

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
   --deploy) deploy="$2";;
   --uglify) uglify="$2";;
   *) break;
 esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

第2章：脚本shebang

第2.1节：Env shebang

要使用位于PATH环境变量中的bash可执行文件通过可执行文件env来执行脚本文件，脚本文件的第一行必须指明env可执行文件的绝对路径，并带上参数bash：

```
#!/usr/bin/env bash
```

只有当脚本像这样直接启动时，shebang中的env路径才会被解析和使用：

```
script.sh
```

脚本必须具有执行权限。

当明确指定bash解释器来执行脚本时，shebang会被忽略：

```
bash script.sh
```

第2.2节：直接shebang

要使用bash解释器执行脚本文件，脚本文件的**第一行**必须指明要使用的bash可执行文件的绝对路径：

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

只有当脚本被直接这样启动时，shebang中的bash路径才会被解析并使用：

```
./script.sh
```

脚本必须具有执行权限。

当明确指定bash解释器来执行脚本时，shebang会被忽略：

```
bash script.sh
```

第2.3节：其他shebang

内核识别两种程序。一种是二进制程序，通过其ELF(ExtenableLoadableFormat)头部识别，通常由编译器生成。另一种是各种类型的脚本。

如果文件的第一行以序列#!开头，那么接下来的字符串必须是解释器的路径名。
如果内核读取到这一行，它会调用该路径名指定的解释器，并将该行后续的所有单词作为参数传递给解释器。如果没有名为“something”或“wrong”的文件：

```
#!/bin/bash something wrong
echo "这行永远不会被打印"
```

bash尝试执行其参数“something wrong”，但该文件不存在。脚本文件名也会被添加进去。
要清楚地看到这一点，请使用一个 echo shebang：

Chapter 2: Script shebang

Section 2.1: Env shebang

To execute a script file with the bash executable found in the PATH environment variable by using the executable env, the **first line** of a script file must indicate the absolute path to the env executable with the argument bash:

```
#!/usr/bin/env bash
```

The env path in the shebang is resolved and used only if a script is directly launch like this:

```
script.sh
```

The script must have execution permission.

The shebang is ignored when a bash interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.2: Direct shebang

To execute a script file with the bash interpreter, the **first line** of a script file must indicate the absolute path to the bash executable to use:

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

The bash path in the shebang is resolved and used only if a script is directly launch like this:

```
./script.sh
```

The script must have execution permission.

The shebang is ignored when a bash interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.3: Other shebangs

There are two kinds of programs the kernel knows of. A binary program is identified by it's ELF (**E**xtenable**L**oadable**F**ormat) header, which is usually produced by a compiler. The second one are scripts of any kind.

If a file starts in the very first line with the sequence **#!** then the next string has to be a pathname of an interpreter. If the kernel reads this line, it calls the interpreter named by this pathname and gives all of the following words in this line as arguments to the interpreter. If there is no file named "something" or "wrong":

```
#!/bin/bash something wrong
echo "This line never gets printed"
```

bash tries to execute its argument "something wrong" which doesn't exist. The name of the script file is added too. To see this clearly use an **echo** shebang:

```
#"/bin/echo something wrong
# 现在调用名为“thisscript”的脚本，方法如下：
# thisscript one two
# 输出将是：
something wrong ./thisscript one two
```

一些程序如 awk 使用这种技术来运行存储在磁盘文件中的较长脚本。


```
#"/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

Some programs like **awk** use this technique to run longer scripts residing in a disk file.

第3章：目录导航

第3.1节：绝对目录与相对目录

要切换到一个绝对指定的目录，使用完整路径名，以斜杠 / 开头，如下所示：

```
cd /home/username/project/abc
```

如果您想切换到当前目录附近的目录，可以指定相对位置。例如，如果您已经在/home/username/project目录下，您可以这样输入子目录abc：

```
cd abc
```

如果你想进入当前目录的上一级目录，可以使用别名..。例如，如果你在/home/username/project/abc 并且想进入/home/username/project，那么你可以这样做：

```
cd ..
```

这也可以称为“向上”进入一个目录。

第3.2节：切换到上一个目录

对于当前的shell，这会带你进入之前所在的目录，无论它在哪里。

```
cd -
```

多次执行实际上会在当前目录和上一个目录之间“切换”。

第3.3节：切换到主目录

默认目录是主目录（$HOME，通常是/home/username），因此不带任何目录的cd命令会将你带到那里

```
cd
```

或者你可以更明确地写：

```
cd $HOME
```

主目录的快捷方式是~，因此也可以使用它。

```
cd ~
```

第3.4节：切换到脚本所在目录

一般来说，Bash脚本有两种类型：

1. 从当前工作目录操作的系统工具
2. 修改相对于其自身在文件系统中位置的文件的项目工具

对于第二类脚本，切换到脚本存放的目录是有用的。可以使用以下命令实现这一点：

Chapter 3: Navigating directories

Section 3.1: Absolute vs relative directories

To change to an absolutely specified directory, use the entire name, starting with a slash `/`, thus:

```
cd /home/username/project/abc
```

If you want to change to a directory near your current on, you can specify a relative location. For example, if you are already in `/home/username/project`, you can enter the subdirectory `abc` thus:

```
cd abc
```

If you want to go to the directory above the current directory, you can use the alias `..`. For example, if you were in `/home/username/project/abc` and wanted to go to `/home/username/project`, then you would do the following:

```
cd ..
```

This may also be called going "up" a directory.

Section 3.2: Change to the last directory

For the current shell, this takes you to the previous directory that you were in, no matter where it was.

```
cd -
```

Doing it multiple times effectively "toggles" you being in the current directory or the previous one.

Section 3.3: Change to the home directory

The default directory is the home directory (`$HOME`, typically `/home/username`), so `cd` without any directory takes you there

```
cd
```

Or you could be more explicit:

```
cd $HOME
```

A shortcut for the home directory is `~`, so that could be used as well.

```
cd ~
```

Section 3.4: Change to the Directory of the Script

In general, there are two types of Bash **scripts**:

1. System tools which operate from the current working directory
2. Project tools which modify files relative to their own place in the files system

For the second type of scripts, it is useful to change to the directory where the script is stored. This can be done with the following command:

```
cd "$(dirname "$(readlink -f "$0")")"
```

该命令执行三个命令：

1. readlink -f "$0" 用于确定当前脚本（$0）的路径
2. dirname 将脚本路径转换为其所在目录的路径
3. cd 将当前工作目录更改为 dirname 接收到的目录


```
cd "$(dirname "$(readlink -f "$0")")"
```

This command runs 3 commands:

1. **`readlink -f "$0"`** determines the path to the current script (`$0`)
2. **`dirname`** converts the path to script to the path to its directory
3. `cd` changes the current work directory to the directory it receives from **`dirname`**

第4章：列出文件

选项	描述
-a, --all	列出所有条目，包括以点开头的条目
-A, --almost-all	列出除.和..之外的所有条目
-c	按修改时间排序文件
-d, --directory	列出目录条目
-h, --human-readable	以人类可读格式显示大小（即K，M）
-H	与上述相同，但使用1000的幂而非1024
-l	以长列表格式显示内容
-o	长列表格式，不显示组信息
-r, --reverse	以相反顺序显示内容
-s, --size	以块为单位打印每个文件的大小
-S	按文件大小排序
--sort=WORD	按指定字段排序内容。（例如大小、版本、状态）
-t	按修改时间排序
-u	按最后访问时间排序
-v	按版本排序
-1	每行列出一个文件

第4.1节：以长列表格式列出文件

ls命令的-l选项以长列表格式打印指定目录的内容。如果未指定目录，则默认列出当前目录的内容。

```
ls -l /etc
```

示例输出：

```
总计 1204
drwxr-xr-x  3 root root    4096 4月  21 03:44 acpi
-rw-r--r--  1 root root    3028 4月  21 03:38 adduser.conf
drwxr-xr-x  2 root root    4096 6月  11 20:42 alternatives
...
```

输出首先显示总计，表示所列目录中所有文件的总大小（以块为单位）。然后它为所列目录中的每个文件显示八列信息。以下是输出中每列的详细说明：

列号	示例	描述
1.1	d	文件类型（见下表）
1.2	rwxr-xr-x	权限字符串
2	3	硬链接数
3	root	所有者名称
4	root	所有者组
5	4096	文件大小（字节）
6	4月21日 03:44	修改时间
7	acpi	文件名

Chapter 4: Listing Files

Option	Description
-a, --all	List all entries including ones that start with a dot
-A, --almost-all	List all entries excluding . and ..
-c	Sort files by change time
-d, --directory	List directory entries
-h, --human-readable	Show sizes in human readable format (i.e. K, M)
-H	Same as above only with powers of 1000 instead of 1024
-l	Show contents in long-listing format
-o	Long -listing format without group info
-r, --reverse	Show contents in reverse order
-s, --size	Print size of each file in blocks
-S	Sort by file size
--sort=WORD	Sort contents by a word. (i.e size, version, status)
-t	Sort by modification time
-u	Sort by last access time
-v	Sort by version
-1	List one file per line

Section 4.1: List Files in a Long Listing Format

The ls command's -l option prints a specified directory's contents in a long listing format. If no directory is specified then, by default, the contents of the current directory are listed.

```
ls -l /etc
```

Example Output:

```
total 1204
drwxr-xr-x  3 root root    4096 Apr 21 03:44 acpi
-rw-r--r--  1 root root    3028 Apr 21 03:38 adduser.conf
drwxr-xr-x  2 root root    4096 Jun 11 20:42 alternatives
...
```

The output first displays total, which indicates the total size in **blocks** of all the files in the listed directory. It then displays eight columns of information for each file in the listed directory. Below are the details for each column in the output:

Column No.	Example	Description
1.1	d	File type (see table below)
1.2	rwxr-xr-x	Permission string
2	3	Number of hard links
3	root	Owner name
4	root	Owner group
5	4096	File size in bytes
6	Apr 21 03:44	Modification time
7	acpi	File name

文件类型

文件类型可以是以下任意字符之一。

字符	文件类型
-	常规文件
b	块特殊文件
c	字符特殊文件
C	高性能（“连续数据”）文件
d	目录
D	门（仅限 Solaris 2.5+ 中的特殊 IPC 文件）
l	符号链接
M	离线（“迁移”）文件（Cray DMF）
n	网络特殊文件（HP-UX）
p	FIFO（命名管道）
P	端口（仅限 Solaris 10 及以上的特殊系统文件）
s	套接字
?	其他某种文件类型

第4.2节：列出最近修改的十个文件

下面将列出当前目录中最多十个最近修改的文件，使用长列表格式（-l）并按时间排序（-t）。

```
ls -lt | head
```

第4.3节：列出所有文件，包括点文件

一个dotfile是文件名以.开头的文件。这些文件通常被ls隐藏，除非特别请求，否则不会列出。

例如，以下是ls的输出：

```
$ ls
bin  pki
```

-a或--all选项将列出所有文件，包括dotfile。

```
$ ls -a
.   .ansible        .bash_logout   .bashrc  .lesshst  .puppetlabs  .viminfo
..  .bash_history   .bash_profile  bin      pki       .ssh
```

-A或--almost-all选项将列出所有文件，包括dotfile，但不列出隐含的.和..。注意，.是当前目录，..是父目录。

```
$ ls -A
.ansible        .bash_logout   .bashrc  .lesshst  .puppetlabs  .viminfo
.bash_history   .bash_profile  bin      pki       .ssh
```

第4.4节：不使用`ls`列出文件

使用 Bash shell 的文件名扩展和大括号扩展功能来获取文件名：

File Type

The file type can be one of any of the following characters.

Character	File Type
-	Regular file
b	Block special file
c	Character special file
C	High performance ("contiguous data") file
d	Directory
D	Door (special IPC file in Solaris 2.5+ only)
l	Symbolic link
M	Off-line ("migrated") file (Cray DMF)
n	Network special file (HP-UX)
p	FIFO (named pipe)
P	Port (special system file in Solaris 10+ only)
s	Socket
?	Some other file type

Section 4.2: List the Ten Most Recently Modified Files

The following will list up to ten of the most recently modified files in the current directory, using a long listing format (`-l`) and sorted by time (`-t`).

```
ls -lt | head
```

Section 4.3: List All Files Including Dotfiles

A **dotfile** is a file whose names begin with a `.`. These are normally hidden by `ls` and not listed unless requested.

For example the following output of `ls`:

```
$ ls
bin  pki
```

The `-a` or `--all` option will list all files, including dotfiles.

```
$ ls -a
.   .ansible        .bash_logout   .bashrc  .lesshst  .puppetlabs  .viminfo
..  .bash_history   .bash_profile  bin      pki       .ssh
```

The `-A` or `--almost-all` option will list all files, including dotfiles, but does not list implied `.` and `..`. Note that `.` is the current directory and `..` is the parent directory.

```
$ ls -A
.ansible        .bash_logout   .bashrc  .lesshst  .puppetlabs  .viminfo
.bash_history   .bash_profile  bin      pki       .ssh
```

Section 4.4: List Files Without Using `ls`

Use the Bash shell's filename expansion and brace expansion capabilities to obtain the filenames:

```
# 显示当前目录中的文件和目录
printf "%s" *

# 仅显示当前目录中的目录printf "%s" */


# 仅显示（部分）图像文件
printf "%s" *.{gif,jpg,png}
```

为了将文件列表捕获到变量中以便处理，通常的好做法是使用bash数组：

```
files=( * )

# 遍历它们
for file in "${files[@]}"; do
    echo "$file"
done
```

第4.5节：列出文件

ls 命令列出指定目录的内容，不包括点文件。如果未指定目录，则默认列出当前目录的内容。

列出的文件默认按字母顺序排序，如果不能在一行显示，则以列对齐方式排列。

```
$ ls
apt  configs  Documents  Fonts    Music      Programming  Templates  workspace
bin  Desktop  eclipse    git      Pictures   Public       Videos
```

第4.6节：以树状格式列出文件

tree 命令以树状格式列出指定目录的内容。如果未指定目录，则默认列出当前目录的内容。

示例输出：

```
$ tree /tmp
/tmp
├── 5037
├── adb.log
└── evince-20965
    └── image.FPWTJY.png
```

使用ree命令的-L选项限制显示深度，使用-d选项仅列出目录。

示例输出：

```
$ tree -L 1 -d /tmp
/tmp
└── evince-20965
```

第4.7节：按大小排序列出文件

ls命令的-S选项按文件大小降序排序文件。


```
# display the files and directories that are in the current directory
printf "%s\n" *

# display only the directories in the current directory
printf "%s\n" */

# display only (some) image files
printf "%s\n" *.{gif,jpg,png}
```

To capture a list of files into a variable for processing, it is typically good practice to use a bash array:

```
files=( * )

# iterate over them
for file in "${files[@]}"; do
    echo "$file"
done
```

Section 4.5: List Files

The `ls` command lists the contents of a specified directory, **excluding** dotfiles. If no directory is specified then, by default, the contents of the current directory are listed.

Listed files are sorted alphabetically, by default, and aligned in columns if they don't fit on one line.

```
$ ls
apt  configs  Documents  Fonts    Music      Programming  Templates  workspace
bin  Desktop  eclipse    git      Pictures   Public       Videos
```

Section 4.6: List Files in a Tree-Like Format

The `tree` command lists the contents of a specified directory in a tree-like format. If no directory is specified then, by default, the contents of the current directory are listed.

Example Output:

```
$ tree /tmp
/tmp
├── 5037
├── adb.log
└── evince-20965
    └── image.FPWTJY.png
```

Use the `tree` command's `-L` option to limit the display depth and the `-d` option to only list directories.

Example Output:

```
$ tree -L 1 -d /tmp
/tmp
└── evince-20965
```

Section 4.7: List Files Sorted by Size

The `ls` command's `-S` option sorts the files in descending order of file size.

```
$ ls -l -S ./Fruits
total 444
-rw-rw-rw- 1 root root 295303 7月 28 19:19 apples.jpg
-rw-rw-rw- 1 root root 102283 7月 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root  50197 7月 28 19:19 bananas.jpg
```

与-r选项一起使用时，排序顺序将被反转。

```
$ ls -l -S -r /Fruits
total 444
-rw-rw-rw- 1 root root  50197 7月 28 19:19 bananas.jpg
-rw-rw-rw- 1 root root 102283 7月 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root 295303 7月 28 19:19 apples.jpg
```



```
$ ls -l -S ./Fruits
total 444
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
```

When used with the `-r` option the sort order is reversed.

```
$ ls -l -S -r /Fruits
total 444
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
```

第5章：使用cat

选项	详细信息
-n	打印行号
-v	使用^和M-符号显示不可打印字符，除换行符(LFD)和制表符(TAB)外
-T	将TAB字符显示为^I
-E	将换行符(LF)显示为$
-e	与 -vE 相同
-b	对非空输出行编号，覆盖 -n
-A	等同于 -vET
-s	抑制重复的空输出行，s 指的是压缩

第5.1节：连接文件

这是 cat 的主要用途。

```
cat file1 file2 file3 > file_all
```

cat 也可以类似地用作管道的一部分来连接文件，例如。

```
cat file1 file2 file3 | grep foo
```

第5.2节：打印文件内容

```
cat file.txt
```

将打印文件的内容。

如果文件包含非ASCII字符，可以使用cat -v以符号方式显示这些字符。这在控制字符否则不可见的情况下非常有用。

```
cat -v unicode.txt
```

通常情况下，对于交互式使用，最好使用交互式分页器，如less或more。（less比more功能强大得多，建议更频繁地使用less而不是more。）

```
less file.txt
```

将文件内容作为命令的输入。通常被认为更好的方法（UUOC）是使用重定向。

```
tr A-Z a-z <file.txt   # 作为 cat file.txt | tr A-Z a-z 的替代方案
```

如果需要将内容从末尾开始反向列出，可以使用命令ac：

```
tac file.txt
```

如果想要带行号打印内容，则使用-n参数配合cat：

```
cat -n file.txt
```


Chapter 5: Using cat

Option	Details
-n	Print line numbers
-v	Show non-printing characters using ^ and M- notation except LFD and TAB
-T	Show TAB characters as ^I
-E	Show linefeed(LF) characters as $
-e	Same as -vE
-b	Number nonempty output lines, overrides -n
-A	equivalent to -vET
-s	suppress repeated empty output lines, s refers to squeeze

Section 5.1: Concatenate files

This is the primary purpose of `cat`.

```
cat file1 file2 file3 > file_all
```

`cat` can also be used similarly to concatenate files as part of a pipeline, e.g.

```
cat file1 file2 file3 | grep foo
```

Section 5.2: Printing the Contents of a File

```
cat file.txt
```

will print the contents of a file.

If the file contains non-ASCII characters, you can display those characters symbolically with `cat -v`. This can be quite useful for situations where control characters would otherwise be invisible.

```
cat -v unicode.txt
```

Very often, for interactive use, you are better off using an interactive pager like `less` or `more`, though. (`less` is far more powerful than `more` and it is advised to use `less` more often than `more`.)

```
less file.txt
```

To pass the contents of a file as input to a command. An approach usually seen as better (UUOC) is to use redirection.

```
tr A-Z a-z <file.txt   # as an alternative to cat file.txt | tr A-Z a-z
```

In case the content needs to be listed backwards from its end the command `tac` can be used:

```
tac file.txt
```

If you want to print the contents with line numbers, then use `-n` with `cat`:

```
cat -n file.txt
```

要以完全明确的字节形式显示文件内容，十六进制转储是标准解决方案。这适用于文件的非常简短片段，例如当你不知道确切编码时。标准的十六进制转储工具是od -cH，尽管其表示方式稍显繁琐；常用的替代工具包括xxd和hexdump。

```
$ printf 'Hëllö wörld' | xxd
0000000: 48c3 ab6c 6cc3 b620 77c3 b672 6c64       H..ll.. w..rld
```

第5.3节：写入文件

```
cat >file
```

它允许你在终端输入文本，文本将被保存到名为file的文件中。

```
cat >>file
```

将执行相同操作，但会将文本追加到文件末尾。

注意：`Ctrl+D`用于结束终端（Linux）上的文本输入

here文档可以用于将文件内容内联到命令行或脚本中：

```
cat <<END >file
Hello, World.
END
```

<<重定向符号后的标记是一个任意字符串，必须单独占据一行（无前导或尾随空白）以表示here文档的结束。你可以添加引号以防止shell执行命令替换和变量插值：

```
cat <<'fnord'
这里的内容不会被更改
fnord
```

（去掉引号后，here 会作为命令执行，$changed 会被替换为变量 changed 的值——如果未定义，则替换为空。）

第5.4节：显示不可打印字符

这对于查看是否存在不可打印字符或非ASCII字符非常有用。

例如，如果你从网页复制粘贴代码，可能会有类似 ” 的引号，而不是标准的 "。

```
$ cat -v file.txt
$ cat -vE file.txt # 有助于检测行尾空格。
```

例如。

```
$ echo '”    ' | cat -vE # echo | 将被实际文件替换。
M-bM-^@M-^]     $
```

你也可以使用 cat -A（A代表All），它等同于 cat -vET。它会显示TAB字符（显示为

To display the contents of a file in a completely unambiguous byte-by-byte form, a hex dump is the standard solution. This is good for very brief snippets of a file, such as when you don't know the precise encoding. The standard hex dump utility is `od -cH`, though the representation is slightly cumbersome; common replacements include xxd and `hexdump`.

```
$ printf 'Hëllö wörld' | xxd
0000000: 48c3 ab6c 6cc3 b620 77c3 b672 6c64       H..ll.. w..rld
```

Section 5.3: Write to a file

```
cat >file
```

It will let you write the text on terminal which will be saved in a file named *file*.

```
cat >>file
```

will do the same, except it will append the text to the end of the file.

N.B: `Ctrl+D` to end writing text on terminal (Linux)

A here document can be used to inline the contents of a file into a command line or a script:

```
cat <<END >file
Hello, World.
END
```

The token after the `<<` redirection symbol is an arbitrary string which needs to occur alone on a line (with no leading or trailing whitespace) to indicate the end of the here document. You can add quoting to prevent the shell from performing command substitution and variable interpolation:

```
cat <<'fnord'
Nothing in `here` will be $changed
fnord
```

(Without the quotes, `here` would be executed as a command, and `$changed` would be substituted with the value of the variable `changed` -- or nothing, if it was undefined.)

Section 5.4: Show non printable characters

This is useful to see if there are any non-printable characters, or non-ASCII characters.

e.g. If you have copy-pasted the code from web, you may have quotes like ” instead of standard ".

```
$ cat -v file.txt
$ cat -vE file.txt # Useful in detecting trailing spaces.
```

e.g.

```
$ echo '”    ' | cat -vE # echo | will be replaced by actual file.
M-bM-^@M-^]     $
```

You may also want to use `cat -A` (A for All) that is equivalent to `cat -vET`. It will display TAB characters (displayed

作为 ^I)、不可打印字符和每行末尾：

```
$ echo '" `' | cat -A
M-bM-^@M-^]^I`$
```

第5.5节：从标准输入读取

```
cat < file.txt
```

输出与 cat file.txt 相同，但它是从标准输入读取文件内容，而不是直接从文件读取。

```
printf "first lineSecond line" | cat -n
```

管道符 | 前的 echo 命令输出两行。cat 命令对输出进行处理，添加行号。

第5.6节：显示带行号的输出

使用 --number 标志在每行前打印行号。或者，-n 也能实现相同功能。

```
$ cat --number file

 1行 1
 2  line 2
 3
 4  line 4
 5  line 5
```

要在计数行时跳过空行，请使用--number-nonblank，或简写为-b。

```
$ cat -b file

 1行 1
 2  line 2

 3  line 4
 4  line 5
```

第5.7节：连接gzip压缩文件

由gzip压缩的文件可以直接连接成更大的gzip压缩文件。

```
cat file1.gz file2.gz file3.gz > combined.gz
```

这是gzip的一个特性，其效率低于先连接输入文件再进行gzip压缩的方式：

```
cat file1 file2 file3 | gzip > combined.gz
```

完整演示：

```
echo 'Hello world!' > hello.txt
echo 'Howdy world!' > howdy.txt
gzip hello.txt
gzip howdy.txt
```


as ^I), non printable characters and end of each line:

```
$ echo '" `' | cat -A
M-bM-^@M-^]^I`$
```

Section 5.5: Read from standard input

```
cat < file.txt
```

Output is same as `cat file.txt`, but it reads the contents of the file from standard input instead of directly from the file.

```
printf "first line\nSecond line\n" | cat -n
```

The echo command before | outputs two lines. The cat command acts on the output to add line numbers.

Section 5.6: Display line numbers with output

Use the `--number` flag to print line numbers before each line. Alternatively, `-n` does the same thing.

```
$ cat --number file

 1  line 1
 2  line 2
 3
 4  line 4
 5  line 5
```

To skip empty lines when counting lines, use the `--number-nonblank`, or simply `-b`.

```
$ cat -b file

 1  line 1
 2  line 2

 3  line 4
 4  line 5
```

Section 5.7: Concatenate gzipped files

Files compressed by `gzip` can be directly concatenated into larger gzipped files.

```
cat file1.gz file2.gz file3.gz > combined.gz
```

This is a property of `gzip` that is less efficient than concatenating the input files and gzipping the result:

```
cat file1 file2 file3 | gzip > combined.gz
```

A complete demonstration:

```
echo 'Hello world!' > hello.txt
echo 'Howdy world!' > howdy.txt
gzip hello.txt
gzip howdy.txt
```

```
cat hello.txt.gz howdy.txt.gz > greetings.txt.gz

gunzip greetings.txt.gz

cat greetings.txt
```

结果是

```
Hello world!
Howdy world!
```

注意，`greetings.txt.gz` 是一个 ***单一文件***，并且被解压为 ***单一文件*** greeting.txt。与此形成对比的是 `tar -czf hello.txt howdy.txt > greetings.tar.gz`，它将文件分别保存在 tar 包内。


```
cat hello.txt.gz howdy.txt.gz > greetings.txt.gz

gunzip greetings.txt.gz

cat greetings.txt
```

Which results in

```
Hello world!
Howdy world!
```

Notice that `greetings.txt.gz` is a ***single file*** and is decompressed as the ***single file*** `greeting.txt`. Contrast this with `tar -czf hello.txt howdy.txt > greetings.tar.gz`, which keeps the files separate inside the tarball.

第6章：Grep

第6.1节：如何在文件中搜索模式

在文件bar中查找单词foo：

```
grep foo ~/Desktop/bar
```

查找文件bar中所有不包含foo的行：

```
grep -v foo ~/Desktop/bar
```

查找所有以foo结尾的单词（通配符扩展）：

```
grep "*foo" ~/Desktop/bar
```


Chapter 6: Grep

Section 6.1: How to search a file for a pattern

To find the word **foo** in the file *bar* :

```
grep foo ~/Desktop/bar
```

To find all lines that **do not** contain foo in the file *bar* :

```
grep -v foo ~/Desktop/bar
```

To use find all words containing foo in the end (WIldcard Expansion):

```
grep "*foo" ~/Desktop/bar
```

第7章：别名

Shell别名是一种创建新命令或用自己的代码包装现有命令的简单方法。它们与shell函数有些重叠，但shell函数更灵活，因此通常应优先使用。

第7.1节：绕过别名

有时你可能想暂时绕过一个别名，而不禁用它。以一个具体的例子来说，
考虑这个别名：

```
alias ls='ls --color=auto'
```

假设你想在不禁用别名的情况下使用ls命令。你有几种选择：

- 使用`command`内置命令：`command ls`
- 使用命令的完整路径：`/bin/ls`
- 在命令名中任意位置添加\，例如：\ls，或l\s
- 引用命令：`"ls"`或`'ls'`

第7.2节：创建别名

```
alias word='command'
```

调用word将运行command。传递给别名的任何参数都会简单地附加到别名的目标命令后面：

```
alias myAlias='some command --with --options'
myAlias foo bar baz
```

然后shell将执行：

```
some command --with --options foo bar baz
```

要在同一个别名中包含多个命令，可以用&&将它们连接起来。例如：

```
alias print_things='echo "foo" && echo "bar" && echo "baz"'
```

第7.3节：删除别名

要删除现有别名，请使用：

```
unalias {alias_name}
```

示例：

```
# 创建别名
$ alias now='date'

# 预览别名
$ now
Thu Jul 21 17:11:25 CEST 2016

# 删除别名
$ unalias now
```


Chapter 7: Aliasing

Shell aliases are a simple way to create new commands or to wrap existing commands with code of your own. They somewhat overlap with shell functions, which are however more versatile and should therefore often be preferred.

Section 7.1: Bypass an alias

Sometimes you may want to bypass an alias temporarily, without disabling it. To work with a concrete example, consider this alias:

```
alias ls='ls --color=auto'
```

And let's say you want to use the `ls` command without disabling the alias. You have several options:

- Use the `command` builtin: `command ls`
- Use the full path of the command: `/bin/ls`
- Add a `\` anywhere in the command name, for example: `\ls`, or `l\s`
- Quote the command: `"ls"` or `'ls'`

Section 7.2: Create an Alias

```
alias word='command'
```

Invoking `word` will run `command`. Any arguments supplied to the alias are simply appended to the target of the alias:

```
alias myAlias='some command --with --options'
myAlias foo bar baz
```

The shell will then execute:

```
some command --with --options foo bar baz
```

To include multiple commands in the same alias, you can string them together with **`&&`**. For example:

```
alias print_things='echo "foo" && echo "bar" && echo "baz"'
```

Section 7.3: Remove an alias

To remove an existing alias, use:

```
unalias {alias_name}
```

Example:

```
# create an alias
$ alias now='date'

# preview the alias
$ now
Thu Jul 21 17:11:25 CEST 2016

# remove the alias
$ unalias now
```

```
# 测试是否已移除
$ now
-bash: now: command 未找到
```

第7.4节：BASH_ALIASES 是一个内部的 bash 关联数组

别名是命令的命名快捷方式，可以在交互式 bash 实例中定义和使用。它们保存在名为 BASH_ALIASES 的关联数组中。要在脚本中使用此变量，脚本必须在交互式 shell 中运行

```
#!/bin/bash -li
# 注意上面的 -li ! -l 使其表现得像登录 shell
# -i 使其表现得像交互式 shell
#
# shopt -s expand_aliases 在大多数情况下不起作用

echo 当前定义了 ${#BASH_ALIASES[*]} 个别名。

for ali in "${!BASH_ALIASES[@]}"; doprintf "
    alias: %-10s triggers: %s" "$ali" "${BASH_ALIASES[$ali]}"done
```

第7.5节：扩展别名

假设bar是 someCommand -flag1的别名。

在命令行输入bar然后按下 Ctrl + alt + e

你将得到 someCommand -flag1，替代了bar的位置。

第7.6节：列出所有别名

```
alias -p
```

将列出所有当前的别名。


```
# test if removed
$ now
-bash: now: command not found
```

Section 7.4: The BASH_ALIASES is an internal bash assoc array

Aliases are named shortcuts of commands, one can define and use in interactive bash instances. They are held in an associative array named BASH_ALIASES. To use this var in a script, it must be run within an interactive shell

```
#!/bin/bash -li
# note the -li above! -l makes this behave like a login shell
# -i makes it behave like an interactive shell
#
# shopt -s expand_aliases will not work in most cases

echo There are ${#BASH_ALIASES[*]} aliases defined.

for ali in "${!BASH_ALIASES[@]}"; do
   printf "alias: %-10s triggers: %s\n" "$ali" "${BASH_ALIASES[$ali]}"
done
```

Section 7.5: Expand alias

Assuming that `bar` is an alias for `someCommand -flag1`.

Type `bar` on the command line and then press Ctrl + alt + e

you'll get `someCommand -flag1` where `bar` was standing.

Section 7.6: List all Aliases

```
alias -p
```

will list all the current aliases.

第8章：作业和进程

第8.1节：作业处理

创建作业

要创建一个作业，只需在命令后附加一个&：

```
$ sleep 10 &
[1] 20024
```

你也可以通过按下 Ctrl + z：

```
$ sleep 10
^Z
[1]+  已停止                 sleep 10
```

后台和前台进程

要将进程切换到前台，使用命令fg并配合%

```
$ sleep 10 &
[1] 20024

$ fg %1
sleep 10
```

现在你可以与该进程交互。要将其切回后台，可以使用bg命令。由于终端会话被占用，你需要先按下停止进程的快捷键 Ctrl + z.

```
$ sleep 10
^Z
[1]+  已停止                 sleep 10

$ bg %1
[1]+ sleep 10 &
```

由于一些程序员的懒惰，如果只有一个进程，或者是列表中的第一个进程，所有这些命令也可以只用一个%。例如：

```
$ sleep 10 &
[1] 20024

$ fg %        # 将进程调到前台的命令 'fg %' 也可用。
sleep 10
```

或者只是

```
$ %           # 懒惰无界限，'%' 也可用。
sleep 10
```

此外，仅输入 fg 或 bg 而不带任何参数，会处理最后一个作业：

```
$ sleep 20 &
$ sleep 10 &
$ fg
```


Chapter 8: Jobs and Processes

Section 8.1: Job handling

Creating jobs

To create an job, just append a single **&** after the command:

```
$ sleep 10 &
[1] 20024
```

You can also make a running process a job by pressing Ctrl + z:

```
$ sleep 10
^Z
[1]+  Stopped                 sleep 10
```

Background and foreground a process

To bring the Process to the foreground, the command `fg` is used together with `%`

```
$ sleep 10 &
[1] 20024

$ fg %1
sleep 10
```

Now you can interact with the process. To bring it back to the background you can use the `bg` command. Due to the occupied terminal session, you need to stop the process first by pressing Ctrl + z.

```
$ sleep 10
^Z
[1]+  Stopped                 sleep 10

$ bg %1
[1]+ sleep 10 &
```

Due to the laziness of some Programmers, all these commands also work with a single `%` if there is only one process, or for the first process in the list. For Example:

```
$ sleep 10 &
[1] 20024

$ fg %        # to bring a process to foreground 'fg %' is also working.
sleep 10
```

or just

```
$ %           # laziness knows no boundaries, '%' is also working.
sleep 10
```

Additionally, just typing `fg` or `bg` without any argument handles the last job:

```
$ sleep 20 &
$ sleep 10 &
$ fg
```

```
sleep 10
^C
$ fg
sleep 20
```

终止正在运行的作业

```
$ sleep 10 &
[1] 20024

$ kill %1
[1]+  已终止               sleep 10
```

sleep 进程在后台运行，进程ID（pid）为 20024，作业编号为 1。要引用该进程，可以使用pid或作业编号。如果使用作业编号，必须在前面加上 %。kill 命令发送的默认终止信号是 SIGTERM，允许目标进程优雅退出。

下面显示了一些常见的kill信号。要查看完整列表，请运行 kill -l。

信号名称	信号值	作用
SIGHUP	1	挂断信号
SIGINT	2	键盘中断
SIGKILL	9	终止信号
SIGTERM	15	终止信号

启动并终止特定进程

杀死正在运行的进程最简单的方法可能是通过进程名称选择它，如下面使用pkill命令的示例

```
pkill -f test.py
```

（或者）更可靠的方法是使用pgrep搜索实际的进程ID

```
kill $(pgrep -f 'python test.py')
```

同样的结果也可以通过grep结合ps -ef | grep 进程名来获得，然后杀死与得到的pid（进程ID）相关联的进程。通过进程名选择进程在测试环境中很方便，但在生产环境中使用该脚本时可能非常危险：几乎不可能确定名称是否匹配你真正想要终止的进程。在这些情况下，以下方法实际上更安全。

使用以下方法启动最终将被终止的脚本。假设你想执行并最终终止的命令是python test.py。

```
#!/bin/bash

if [[ ! -e /tmp/test.py.pid ]]; then    # 检查文件是否已存在
    python test.py &                    #+如果存在则不运行另一个进程。
    echo $! > /tmp/test.py.pid
否则
    echo -n "错误：该进程已在运行，pid 为 "
    cat /tmp/test.py.pid
    echo
fi
```

这将在/tmp目录下创建一个包含python test.py进程pid的文件。如果该文件已存在，说明命令已经在运行，脚本将返回错误。


```
sleep 10
^C
$ fg
sleep 20
```

Killing running jobs

```
$ sleep 10 &
[1] 20024

$ kill %1
[1]+  Terminated              sleep 10
```

The sleep process runs in the background with process id (pid) `20024` and job number `1`. In order to reference the process, you can use either the pid or the job number. If you use the job number, you must prefix it with `%`. The default kill signal sent by `kill` is `SIGTERM`, which allows the target process to exit gracefully.

Some common kill signals are shown below. To see a full list, run `kill -l`.

Signal name	Signal value	Effect
SIGHUP	1	Hangup
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal

Start and kill specific processes

Probably the easiest way of killing a running process is by selecting it through the process name as in the following example using `pkill` command as

```
pkill -f test.py
```

(or) a more fool-proof way using `pgrep` to search for the actual process-id

```
kill $(pgrep -f 'python test.py')
```

The same result can be obtained using `grep` over `ps -ef | grep name_of_process` then killing the process associated with the resulting pid (process id). Selecting a process using its name is convinient in a testing environment but can be really dangerous when the script is used in production: it is virtually impossible to be sure that the name will match the process you actually want to kill. In those cases, the following approach is actually much safe.

Start the script that will eventually killed with the following approach. Let's assume that the command you want to execute and eventually kill is `python test.py`.

```
#!/bin/bash

if [[ ! -e /tmp/test.py.pid ]]; then   # Check if the file already exists
    python test.py &                   #+and if so do not run another process.
    echo $! > /tmp/test.py.pid
else
    echo -n "ERROR: The process is already running with pid "
    cat /tmp/test.py.pid
    echo
fi
```

This will create a file in the `/tmp` directory containing the pid of the `python test.py` process. If the file already exists, we assume that the command is already running and the script return an error.

然后，当你想要终止它时，使用以下脚本：

```
#!/bin/bash

if [[ -e /tmp/test.py.pid ]]; then   # 如果文件不存在，则
    kill `cat /tmp/test.py.pid`      #+进程未运行，无需
    rm /tmp/test.py.pid              #+尝试终止它。
否则
    echo "test.py 未在运行"
fi
```

这将精确终止与你的命令关联的进程，而不依赖任何易变信息（如运行命令的字符串）。即使在文件不存在的情况下，脚本也会假设你想终止一个未运行的进程。

最后这个示例可以很容易地改进以支持多次运行相同命令（例如，向pid文件追加而不是覆盖），并处理进程在被终止前已退出的情况。

第8.2节：检查特定端口上运行的进程

检查端口8080上运行的进程

```
lsof -i :8080
```

第8.3节：取消后台作业的关联

```
$ gzip extremelylargefile.txt &
$ bg
$ disown %1
```

这允许长时间运行的进程在关闭你的shell（终端、ssh等）后继续运行。

第8.4节：列出当前作业

```
$ tail -f /var/log/syslog > log.txt
[1]+  已停止                  tail -f /var/log/syslog > log.txt

$ sleep 10 &

$ jobs
[1]+  已停止                  tail -f /var/log/syslog > log.txt
[2]-  运行中                  sleep 10 &
```

第8.5节：查找正在运行的进程信息

ps aux | grep <搜索词> 显示匹配搜索词的进程

示例：

```
root@server7:~# ps aux | grep nginx
root       315  0.0  0.3 144392  1020 ?        Ss   5月28日   0:00 nginx: 主进程
/usr/sbin/nginx
www-data  5647  0.0  1.1 145124  3048 ?        S    7月18日   2:53 nginx: 工作进程
www-data  5648  0.0  0.1 144392   376 ?        S    7月18日   0:00 nginx: 缓存管理进程
root     13134  0.0  0.3   4960   920 pts/0    S+   14:33   0:00 grep --color=auto nginx
root@server7:~#
```


Then, when you want to kill it use the following script:

```
#!/bin/bash

if [[ -e /tmp/test.py.pid ]]; then   # If the file do not exists, then the
    kill `cat /tmp/test.py.pid`      #+the process is not running. Useless
    rm /tmp/test.py.pid              #+trying to kill it.
else
    echo "test.py is not running"
fi
```

that will kill exactly the process associated with your command, without relying on any volatile information (like the string used to run the command). Even in this case if the file does not exist, the script assume that you want to kill a non-running process.

This last example can be easily improved for running the same command multiple times (appending to the pid file instead of overwriting it, for example) and to manage cases where the process dies before being killed.

Section 8.2: Check which process running on specific port

To check which process running on port 8080

```
lsof -i :8080
```

Section 8.3: Disowning background job

```
$ gzip extremelylargefile.txt &
$ bg
$ disown %1
```

This allows a long running process to continue once your shell (terminal, ssh, etc) is closed.

Section 8.4: List Current Jobs

```
$ tail -f /var/log/syslog > log.txt
[1]+  Stopped                 tail -f /var/log/syslog > log.txt

$ sleep 10 &

$ jobs
[1]+  Stopped                 tail -f /var/log/syslog > log.txt
[2]-  Running                 sleep 10 &
```

Section 8.5: Finding information about a running process

`ps aux | grep <search-term>` shows processes matching *search-term*

Example:

```
root@server7:~# ps aux | grep nginx
root       315  0.0  0.3 144392  1020 ?        Ss   May28   0:00 nginx: master process
/usr/sbin/nginx
www-data  5647  0.0  1.1 145124  3048 ?        S    Jul18   2:53 nginx: worker process
www-data  5648  0.0  0.1 144392   376 ?        S    Jul18   0:00 nginx: cache manager process
root     13134  0.0  0.3   4960   920 pts/0    S+   14:33   0:00 grep --color=auto nginx
root@server7:~#
```

这里，第二列是进程ID。例如，如果你想终止nginx进程，可以使用命令kill 5647。建议总是使用带有SIGTERM的kill命令，而不是SIGKILL。

第8.6节：列出所有进程

列出系统上所有进程有两种常用方法。两者都会列出所有用户运行的所有进程，虽然它们输出的格式不同（差异的原因是历史遗留问题）。

```
ps -ef   # 列出所有进程
ps aux   # 以另一种格式（BSD）列出所有进程
```

这可以用来检查某个应用程序是否正在运行。例如，检查 SSH 服务器（sshd）是否正在运行：

```
ps -ef | grep sshd
```


Here, second column is the process id. For example, if you want to kill the nginx process, you can use the command `kill 5647`. It is always adviced to use the `kill` command with `SIGTERM` rather than `SIGKILL`.

Section 8.6: List all processes

There are two common ways to list all processes on a system. Both list all processes running by all users, though they differ in the format they output (the reason for the differences are historical).

```
ps -ef   # lists all processes
ps aux   # lists all processes in alternative format (BSD)
```

This can be used to check if a given application is running. For example, to check if the SSH server (sshd) is running:

```
ps -ef | grep sshd
```

第9章：重定向

参数	详情
内部文件描述符	一个整数。
方向	以下之一 `>`, `<` 或 `<>`
外部文件描述符或路径	`&` 后跟一个表示文件描述符的整数或路径。

第9.1节：重定向标准输出

`>` 将当前命令的标准输出（即STDOUT）重定向到一个文件或另一个描述符。

这些示例将ls命令的输出写入文件file.txt

```
ls >file.txt
> file.txt ls
```

如果目标文件不存在，则创建该文件，否则该文件将被截断。

默认的重定向描述符是标准输出或1，当未指定时。此命令等同于之前显式指定标准输出的示例：

```
ls 1>file.txt
```

注意：重定向由执行的shell初始化，而非执行的命令，因此它在命令执行之前完成。

第9.2节：追加与截断

截断 `>`

1. 如果指定文件不存在，则创建该文件。
2.截断（清空文件内容）
3.写入文件

```
$ echo "第一行" > /tmp/lines
$ echo "第二行" > /tmp/lines

$ cat /tmp/lines
第二行
```

追加 `>>`

1. 如果指定文件不存在，则创建该文件。
2.追加文件（写入文件末尾）。

```
# 覆盖已有文件
$ echo "第一行" > /tmp/lines

# 追加第二行
$ echo "第二行" >> /tmp/lines

$ cat /tmp/lines
第一行
第二行
```


Chapter 9: Redirection

Parameter	Details
internal file descriptor	An integer.
direction	One of `>`, `<` or `<>`
external file descriptor or path	`&` followed by an integer for file descriptor or a path.

Section 9.1: Redirecting standard output

`>` redirect the standard output (aka `STDOUT`) of the current command into a file or another descriptor.

These examples write the output of the `ls` command into the file `file.txt`

```
ls >file.txt
> file.txt ls
```

The target file is created if it doesn't exists, otherwise this file is truncated.

The default redirection descriptor is the standard output or `1` when none is specified. This command is equivalent to the previous examples with the standard output explicitly indicated:

```
ls 1>file.txt
```

Note: the redirection is initialized by the executed shell and not by the executed command, therefore it is done before the command execution.

Section 9.2: Append vs Truncate

Truncate `>`

1. Create specified file if it does not exist.
2. Truncate (remove file's content)
3. Write to file

```
$ echo "first line" > /tmp/lines
$ echo "second line" > /tmp/lines

$ cat /tmp/lines
second line
```

Append `>>`

1. Create specified file if it does not exist.
2. Append file (writing at end of file).

```
# Overwrite existing file
$ echo "first line" > /tmp/lines

# Append a second line
$ echo "second line" >> /tmp/lines

$ cat /tmp/lines
first line
second line
```

第9.3节：重定向STDOUT和STDERR

像0和1这样的文件描述符是指针。我们通过重定向来改变文件描述符指向的内容。`>/dev/null` 表示1指向`/dev/null`。

首先我们将1（STDOUT）指向/dev/null，然后将2（STDERR）指向1所指向的位置。

```
# STDERR 重定向到 STDOUT：重定向到 /dev/null，
# 实际上将 STDERR 和 STDOUT 都重定向到 /dev/null
echo 'hello' > /dev/null 2>&1
```

版本 ≥ 4.0

这可以进一步简化为以下形式：

```
echo 'hello' &> /dev/null
```

然而，如果考虑到 shell 兼容性，这种形式在生产环境中可能不理想，因为它与 POSIX 标准冲突，引入了解析歧义，并且不支持此功能的 shell 会误解它：

```
# 实际代码
echo 'hello' &> /dev/null
echo 'hello' &> /dev/null 'goodbye'

# 期望行为
echo 'hello' > /dev/null 2>&1
echo 'hello' 'goodbye' > /dev/null 2>&1

# 实际行为
echo 'hello' &
echo 'hello' & goodbye > /dev/null
```

注意：`&>` 在 Bash 和 Zsh 中均能按预期工作。

第9.4节：使用命名管道

有时你可能想让一个程序输出内容并输入到另一个程序，但无法使用标准管道。

```
ls -l | grep ".log"
```

你可以简单地写入一个临时文件：

```
touch tempFile.txt
ls -l > tempFile.txt
grep ".log" < tempFile.txt
```

这对大多数应用来说都能正常工作，然而，没有人会知道 tempFile 是做什么用的，如果它包含了该目录下 ls -l 的输出，可能会被误删。这时命名管道就派上用场了：

```
mkfifo myPipe
ls -l > myPipe
grep ".log" < myPipe
```

`myPipe`在技术上是一个文件（Linux 中一切皆文件），所以让我们在刚创建的空目录中对管道执行`ls -l`命令：

Section 9.3: Redirecting both STDOUT and STDERR

File descriptors like `0` and `1` are pointers. We change what file descriptors point to with redirection. `>/dev/null` means `1` points to `/dev/null`.

First we point `1` (`STDOUT`) to `/dev/null` then point `2` (`STDERR`) to whatever `1` points to.

```
# STDERR is redirect to STDOUT: redirected to /dev/null,
# effectually redirecting both STDERR and STDOUT to /dev/null
echo 'hello' > /dev/null 2>&1
```

Version ≥ 4.0

This *can* be further shortened to the following:

```
echo 'hello' &> /dev/null
```

However, this form may be undesirable in production if shell compatibility is a concern as it conflicts with POSIX, introduces parsing ambiguity, and shells without this feature will misinterpret it:

```
# Actual code
echo 'hello' &> /dev/null
echo 'hello' &> /dev/null 'goodbye'

# Desired behavior
echo 'hello' > /dev/null 2>&1
echo 'hello' 'goodbye' > /dev/null 2>&1

# Actual behavior
echo 'hello' &
echo 'hello' & goodbye > /dev/null
```

NOTE: `&>` is known to work as desired in both Bash and Zsh.

Section 9.4: Using named pipes

Sometimes you may want to output something by one program and input it into another program, but can't use a standard pipe.

```
ls -l | grep ".log"
```

You could simply write to a temporary file:

```
touch tempFile.txt
ls -l > tempFile.txt
grep ".log" < tempFile.txt
```

This works fine for most applications, however, nobody will know what `tempFile` does and someone might remove it if it contains the output of `ls -l` in that directory. This is where a named pipe comes into play:

```
mkfifo myPipe
ls -l > myPipe
grep ".log" < myPipe
```

`myPipe` is technically a file (everything is in Linux), so let's do `ls -l` in an empty directory that we just created a pipe in:

```
mkdir pipeFolder
cd pipeFolder
mkfifo myPipe
ls -l
```

输出是：

```
prw-r--r-- 1 root root 0 7月 25 11:20 myPipe
```

注意权限中的第一个字符，它显示为管道，而不是文件。

现在让我们做点有趣的事情。

打开一个终端，记下目录（或者创建一个以便于清理），然后创建一个管道。

```
mkfifo myPipe
```

现在让我们往管道里放点东西。

```
echo "Hello from the other side" > myPipe
```

你会注意到程序挂起了，管道的另一端仍然关闭。让我们打开管道的另一端，让那些内容通过。

打开另一个终端，进入管道所在的目录（或者如果你知道路径，直接在管道名前加上路径）：

```
cat < myPipe
```

你会注意到，在输出了来自另一端的hello之后，第一个终端中的程序结束了，第二个终端中的程序也结束了。

现在反向运行命令。先运行cat < myPipe，然后向管道中echo一些内容。它仍然有效，因为程序会等待直到有内容写入管道才终止，因为它知道必须接收一些东西。

命名管道对于在终端之间或程序之间传递信息非常有用。

管道很小。一旦满了，写入方会阻塞，直到有读方读取内容，所以你需要在不同的终端运行读写程序，或者将其中一个放到后台运行：

```
ls -l /tmp > myPipe &
cat < myPipe
```

使用命名管道的更多示例：

- 示例1 - 所有命令在同一终端/同一shell中运行

```
$ { ls -l && cat file3; } >mypipe &
$ cat <mypipe
# 输出：打印 ls -l 数据，然后在屏幕上打印 file3 的内容
```

- 示例 2 - 所有命令在同一终端 / 同一 shell 中执行

```
$ ls -l >mypipe &
$ cat file3 >mypipe &
```



```
mkdir pipeFolder
cd pipeFolder
mkfifo myPipe
ls -l
```

The output is:

```
prw-r--r-- 1 root root 0 Jul 25 11:20 myPipe
```

Notice the first character in the permissions, it's listed as a pipe, not a file.

Now let's do something cool.

Open one terminal, and make note of the directory (or create one so that cleanup is easy), and make a pipe.

```
mkfifo myPipe
```

Now let's put something in the pipe.

```
echo "Hello from the other side" > myPipe
```

You'll notice this hangs, the other side of the pipe is still closed. Let's open up the other side of the pipe and let that stuff through.

Open another terminal and go to the directory that the pipe is in (or if you know it, prepend it to the pipe):

```
cat < myPipe
```

You'll notice that after `hello from the other side` is output, the program in the first terminal finishes, as does that in the second terminal.

Now run the commands in reverse. Start with `cat < myPipe` and then echo something into it. It still works, because a program will wait until something is put into the pipe before terminating, because it knows it has to get something.

Named pipes can be useful for moving information between terminals or between programs.

Pipes are small. Once full, the writer blocks until some reader reads the contents, so you need to either run the reader and writer in different terminals or run one or the other in the background:

```
ls -l /tmp > myPipe &
cat < myPipe
```

More examples using named pipes:

- Example 1 - all commands on the same terminal / same shell

```
$ { ls -l && cat file3; } >mypipe &
$ cat <mypipe
# Output: Prints ls -l data and then prints file3 contents on screen
```

- Example 2 - all commands on the same terminal / same shell

```
$ ls -l >mypipe &
$ cat file3 >mypipe &
```

```
$ cat <mypipe
# 输出：这将在屏幕上打印 mypipe 的内容。
```

请注意，首先显示的是 file3 的内容，然后显示 ls -l 的数据（后进先出配置）。

- 示例 3 - 所有命令在同一终端 / 同一 shell 中执行

```
$ { pipedata=$(<mypipe) && echo "$pipedata"; } &
$ ls >mypipe
# 输出：直接在屏幕上打印 ls 的输出
```

请注意，变量 $pipedata 在主终端 / 主 shell 中不可用，因为使用了 & 调用子shell，$pipedata 仅在该子shell中可用。

- 示例4 - 所有命令在同一终端/同一shell中执行

```
$ export pipedata
$ pipedata=$(<mypipe) &
$ ls -l *.sh >mypipe
$ echo "$pipedata"
#输出：正确打印mypipe的内容
```

由于变量的export声明，这会在主shell中正确打印$pipedata变量的值。主终端/主shell不会因为调用后台shell（&）而挂起。

第9.5节：重定向到网络地址

版本 ≥ 2.04

Bash 将某些路径视为特殊路径，并且可以通过写入这些路径进行一些网络通信 /dev/{udp|tcp}/host/port。Bash 不能设置监听服务器，但可以发起连接，对于 TCP 至少可以读取结果。

例如，要发送一个简单的网页请求，可以这样做：

```
exec 3</dev/tcp/www.google.com/80pri
ntf 'GET / HTTP/1.0\r\r' >&3cat <&3
```

这样 www.google.com 的默认网页内容将被打印到 stdout。

类似地

```
printf 'HI' >/dev/udp/192.168.1.1/6666
```

会向 192.168.1.1:6666 上的监听器发送包含 HI 的 UDP 消息

第9.6节：将错误信息打印到 stderr

错误信息通常包含在脚本中，用于调试或提供丰富的用户体验。
简单地写错误信息，如下所示：


```
$ cat <mypipe
#Output: This prints on screen the contents of mypipe.
```

Mind that first contents of `file3` are displayed and then the `ls -l` data is displayed (LIFO configuration).

- Example 3 - all commands on the same terminal / same shell

```
$ { pipedata=$(<mypipe) && echo "$pipedata"; } &
$ ls >mypipe
# Output: Prints the output of ls directly on screen
```

Mind that the variable `$pipedata` is not available for usage in the main terminal / main shell since the use of **&** invokes a subshell and `$pipedata` was only available in this subshell.

- Example 4 - all commands on the same terminal / same shell

```
$ export pipedata
$ pipedata=$(<mypipe) &
$ ls -l *.sh >mypipe
$ echo "$pipedata"
#Output : Prints correctly the contents of mypipe
```

This prints correctly the value of `$pipedata` variable in the main shell due to the export declaration of the variable. The main terminal/main shell is not hanging due to the invocation of a background shell (**&**).

Section 9.5: Redirection to network addresses

Version ≥ 2.04

Bash treats some paths as special and can do some network communication by writing to `/dev/{udp|tcp}/host/port`. Bash cannot setup a listening server, but can initiate a connection, and for TCP can read the results at least.

For example, to send a simple web request one could do:

```
exec 3</dev/tcp/www.google.com/80
printf 'GET / HTTP/1.0\r\n\r\n' >&3
cat <&3
```

and the results of `www.google.com`'s default web page will be printed to `stdout`.

Similarly

```
printf 'HI\n' >/dev/udp/192.168.1.1/6666
```

would send a UDP message containing `HI\n` to a listener on `192.168.1.1:6666`

Section 9.6: Print error messages to stderr

Error messages are generally included in a script for debugging purposes or for providing rich user experience.
Simply writing error message like this:

```
cmd || echo 'cmd failed'
```

对于简单情况可能有效，但这不是常用的方法。在此示例中，错误信息会污染脚本的实际输出，因为错误和成功输出都混合在stdout中。

简而言之，错误信息应该发送到stderr而不是stdout。非常简单：

```
cmd || echo 'cmd failed' >/dev/stderr
```

另一个例子：

```
if cmd; then
    echo '成功'
否则
    echo 'cmd failed' >/dev/stderr
fi
```

在上述示例中，成功信息将打印到stdout，而错误信息将打印到stderr。

打印错误信息的更好方法是定义一个函数：

```
err(){
    echo "E: $*" >>/dev/stderr
}
```

现在，当你需要打印错误时：

```
err "我的错误信息"
```

第9.7节：将多个命令重定向到同一文件

```
{
  echo "主目录内容"
  ls ~
} > output.txt
```

第9.8节：重定向标准输入（STDIN）

< 从其右侧参数读取并写入其左侧参数。

要将文件写入STDIN，我们应当读取/tmp/a_file并写入STDIN即0</tmp/a_file

注意：内部文件描述符默认是0（STDIN）用于<

```
$ echo "b" > /tmp/list.txt
$ echo "a" >> /tmp/list.txt
$ echo "c" >> /tmp/list.txt
$ sort < /tmp/list.txt
a
b
```



```
cmd || echo 'cmd failed'
```

may work for simple cases but it's not the usual way. In this example, the error message will pollute the actual output of the script by mixing both errors and successful output in stdout.

In short, error message should go to stderr not stdout. It's pretty simple:

```
cmd || echo 'cmd failed' >/dev/stderr
```

Another example:

```
if cmd; then
    echo 'success'
else
    echo 'cmd failed' >/dev/stderr
fi
```

In the above example, the success message will be printed on stdout while the error message will be printed on stderr.

A better way to print error message is to define a function:

```
err(){
    echo "E: $*" >>/dev/stderr
}
```

Now, when you have to print an error:

```
err "My error message"
```

Section 9.7: Redirecting multiple commands to the same file

```
{
  echo "contents of home directory"
  ls ~
} > output.txt
```

Section 9.8: Redirecting STDIN

< reads from its right argument and writes to its left argument.

To write a file into STDIN we should *read* /tmp/a_file and *write* into STDIN i.e 0</tmp/a_file

Note: Internal file descriptor defaults to 0 (STDIN) for <

```
$ echo "b" > /tmp/list.txt
$ echo "a" >> /tmp/list.txt
$ echo "c" >> /tmp/list.txt
$ sort < /tmp/list.txt
a
b
c
```

第9.9节：重定向标准错误输出（STDERR）

2是标准错误输出（STDERR）。

```
$ echo_to_stderr 2>/dev/null # 不输出任何内容
```

定义：

echo_to_stderr 是一个将"stderr"写入标准错误输出（STDERR）的命令

```
echo_to_stderr () {
    echo stderr >&2
}

$ echo_to_stderr
stderr
```

第9.10节：标准输入（STDIN）、标准输出（STDOUT）和标准错误输出（STDERR）说明

命令有一个输入（标准输入 STDIN）和两种输出，标准输出（STDOUT）和标准错误（STDERR）。

例如：

STDIN

```
root@server~# read
在此输入一些文本
```

标准输入用于向程序提供输入。（这里我们使用read内置命令从STDIN读取一行。）

STDOUT

```
root@server~# ls file
file
```

标准输出通常用于命令的“正常”输出。例如，ls列出文件，因此文件被发送到STDOUT。

STDERR

```
root@server~# ls anotherfile
ls: 无法访问'anotherfile'：没有那个文件或目录
```

标准错误（顾名思义）用于错误消息。因为该消息不是文件列表，所以它被发送到标准错误输出（STDERR）。

STDIN、STDOUT 和 STDERR 是三个标准流。它们通过数字而非名称被 shell 识别：

0 = 标准输入
1 = 标准输出
2 = 标准错误

默认情况下，STDIN 连接到键盘，STDOUT 和 STDERR 都显示在终端上。但是，我们

Section 9.9: Redirecting STDERR

2 is STDERR.

```
$ echo_to_stderr 2>/dev/null # echos nothing
```

Definitions:

echo_to_stderr is a command that writes "stderr" to STDERR

```
echo_to_stderr () {
    echo stderr >&2
}

$ echo_to_stderr
stderr
```

Section 9.10: STDIN, STDOUT and STDERR explained

Commands have one input (STDIN) and two kinds of outputs, standard output (STDOUT) and standard error (STDERR).

For example:

STDIN

```
root@server~# read
Type some text here
```

Standard input is used to provide input to a program. (Here we're using the **read** builtin to read a line from STDIN.)

STDOUT

```
root@server~# ls file
file
```

Standard output is generally used for "normal" output from a command. For example, ls lists files, so the files are sent to STDOUT.

STDERR

```
root@server~# ls anotherfile
ls: cannot access 'anotherfile': No such file or directory
```

Standard error is (as the name implies) used for error messages. Because this message is not a list of files, it is sent to STDERR.

STDIN, STDOUT and STDERR are the three *standard streams.* They are identified to the shell by a number rather than a name:

0 = Standard in
1 = Standard out
2 = Standard error

By default, STDIN is attached to the keyboard, and both STDOUT and STDERR appear in the terminal. However, we

可以将 STDOUT 或 STDERR 重定向到任何我们需要的位置。例如，假设你只需要标准输出，所有打印到标准错误的错误消息都应被抑制。这时我们使用描述符1和2。

将标准错误重定向到 /dev/null
以上一个例子为例，

```
root@server~# ls anotherfile 2>/dev/null
root@server~#
```

在这种情况下，如果有任何标准错误输出（STDERR），它将被重定向到/dev/null（一个会忽略所有输入的特殊文件），因此你不会在终端上看到任何错误输出。

can redirect either STDOUT or STDERR to whatever we need. For example, let's say that you only need the standard out and all error messages printed on standard error should be suppressed. That's when we use the descriptors 1 and 2.

Redirecting STDERR to /dev/null
Taking the previous example,

```
root@server~# ls anotherfile 2>/dev/null
root@server~#
```

In this case, if there is any STDERR, it will be redirected to /dev/null (a special file which ignores anything put into it), so you won't get any error output on the shell.

第10章：控制结构

[或 test 的参数	详细信息
文件操作符	详细信息
`-e "$file"`	如果文件存在则返回真。
`-d "$file"`	如果文件存在且是目录则返回真
`-f "$file"`	如果文件存在且是普通文件则返回真
`-h "$file"`	如果文件存在且是符号链接，则返回真
字符串比较器	详细信息
`-z "$str"`	如果字符串长度为零，则为真
`-n "$str`	如果字符串长度非零，则为真
`"$str" = "$str2"`	如果字符串 $str 等于字符串 $str2，则为真。不适合整数。可能有效，但结果不一致
`"$str" != "$str2"`	如果字符串不相等，则为真
整数比较器详情	
`"$int1" -eq "$int2"`	如果两个整数相等，则为真
`"$int1" -ne "$int2"`	如果两个整数不相等，则为真
`"$int1" -gt "$int2"`	如果int1大于int2，则为真
`"$int1" -ge "$int2"`	如果int1大于或等于int2，则为真
`"$int1" -lt "$int2"`	如果 int1 小于 int2，则为真
`"$int1" -le "$int2"`	如果 int1 小于或等于 int2，则为真

第10.1节：命令列表的条件执行

如何使用命令列表的条件执行

任何内置命令、表达式或函数，以及任何外部命令或脚本，都可以使用&&（与）和||（或）运算符进行条件执行。

例如，只有当cd命令成功时，才会打印当前目录。

```
cd my_directory && pwd
```

同样，如果cd命令失败，将退出，防止灾难发生：

```
cd my_directory || exit
rm -rf *
```

当以这种方式组合多个语句时，重要的是要记住（与许多C风格的语言不同）这些运算符没有优先级且是左结合的。

因此，该语句将按预期工作...

```
cd my_directory && pwd || echo "No such directory"
```

- 如果cd成功，&& pwd将执行并打印当前工作目录名称。除非pwd失败（极少见），否则|| echo ...不会被执行。
- 如果cd失败，&& pwd将被跳过，|| echo ...将会执行。

但这不会（如果你在想if...then...else的话）...

Chapter 10: Control Structures

Parameter to [or test	Details
File Operators	Details
`-e "$file"`	Returns true if the file exists.
`-d "$file"`	Returns true if the file exists and is a directory
`-f "$file"`	Returns true if the file exists and is a regular file
`-h "$file"`	Returns true if the file exists and is a symbolic link
String Comparators	Details
`-z "$str"`	True if length of string is zero
`-n "$str`	True if length of string is non-zero
`"$str" = "$str2"`	True if string $str is equal to string $str2. Not best for integers. It may work but will be inconsitent
`"$str" != "$str2"`	True if the strings are not equal
Integer Comparators	Details
`"$int1" -eq "$int2"`	True if the integers are equal
`"$int1" -ne "$int2"`	True if the integers are not equals
`"$int1" -gt "$int2"`	True if int1 is greater than int 2
`"$int1" -ge "$int2"`	True if int1 is greater than or equal to int2
`"$int1" -lt "$int2"`	True if int1 is less than int 2
`"$int1" -le "$int2"`	True if int1 is less than or equal to int2

Section 10.1: Conditional execution of command lists

How to use conditional execution of command lists

Any builtin command, expression, or function, as well as any external command or script can be executed conditionally using the **&&***(and)* and ||*(or)* operators.

For example, this will only print the current directory if the `cd` command was successful.

```
cd my_directory && pwd
```

Likewise, this will exit if the `cd` command fails, preventing catastrophe:

```
cd my_directory || exit
rm -rf *
```

When combining multiple statements in this manner, it's important to remember that (unlike many C-style languages) these operators have no precedence and are left-associative.

Thus, this statement will work as expected...

```
cd my_directory && pwd || echo "No such directory"
```

- If the `cd` succeeds, the **`&& pwd`** executes and the current working directory name is printed. Unless **`pwd`** fails (a rarity) the **`|| echo ...`** will not be executed.
- If the `cd` fails, the **`&& pwd`** will be skipped and the **`|| echo ...`** will run.

But this will not (if you're thinking `if...then...else`)...

```
cd my_directory && ls || echo "No such directory"
```

- 如果cd失败，&& ls将被跳过，|| echo ...将被执行。
- 如果cd成功，&& ls将被执行。
  - 如果ls成功，|| echo ...将被忽略。（到目前为止一切正常）
  - ***但是...如果`ls`失败，`|| echo ...`也会被执行。***

> ***是ls，而不是cd，才是前一个命令。***

为什么使用命令列表的条件执行

条件执行比if...then稍快一些，但其主要优势是允许函数和脚本提前退出，或称为“短路”。

与许多语言如C不同，后者需要显式为结构体和变量等分配内存（因此必须释放内存），bash在背后处理这些。在大多数情况下，我们不必在离开函数前清理任何东西。一个return语句会释放函数内所有本地资源，并在栈上的返回地址继续执行。

因此，尽早从函数返回或退出脚本可以显著提升性能，减少系统负载，避免不必要的代码执行。例如...

```
my_function () {

    ### 始终检查返回码

    # 需要一个参数。"" 视为假(1)
    [[ "$1" ]]              || return 1

    # 使用参数。失败则退出
    do_something_with "$1" || return 1
    do_something_else      || return 1

    # 成功！未检测到失败，否则不会执行到这里
    return 0
}
```

第10.2节：If语句

```
if [[ $1 -eq 1 ]]; then
    echo "第一个参数传入了1"
elif [[ $1 -gt 2 ]]; then
    echo "第一个参数没有传入2"
else
    echo "第一个参数既不是1，也不大于2。"
fi
```

结尾的 fi 是必须的，但 elif 和/或 else 子句可以省略。

在 then 之前的分号是将两条命令写在同一行的标准语法；只有当 then 移到下一行时，分号才可以省略。

重要的是要理解，方括号 [[并不是语法的一部分，而是被当作一个命令；测试的是该命令的退出码。因此，必须在方括号周围始终留有空格。


```
cd my_directory && ls || echo "No such directory"
```

- If the cd fails, the **`&& ls`** is skipped and the **`|| echo ...`** is executed.
- If the cd succeeds, the **`&& ls`** is executed.
  - If the ls succeeds, the **`|| echo ...`** is ignored. *(so far so good)*
  - ***BUT... if the `ls` fails, the `|| echo ...` will also be executed.***

> ***It is the `ls`***, not the cd, ***that is the previous command***.

Why use conditional execution of command lists

Conditional execution is a hair faster than `if...then` but its main advantage is allowing functions and scripts to exit early, or "short circuit".

Unlike many languages like C where memory is explicitly allocated for structs and variables and such (and thus must be deallocated), **`bash`** handles this under the covers. In most cases, we don't have to clean up anything before leaving the function. A **`return`** statement will deallocate everything local to the function and pickup execution at the return address on the stack.

Returning from functions or exiting scripts as soon as possible can thus significantly improve performance and reduce system load by avoiding the unnecessary execution of code. For example...

```
my_function () {

    ### ALWAYS CHECK THE RETURN CODE

    # one argument required. "" evaluates to false(1)
    [[ "$1" ]]              || return 1

    # work with the argument. exit on failure
    do_something_with "$1" || return 1
    do_something_else      || return 1

    # Success! no failures detected, or we wouldn't be here
    return 0
}
```

Section 10.2: If statement

```
if [[ $1 -eq 1 ]]; then
    echo "1 was passed in the first parameter"
elif [[ $1 -gt 2 ]]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

The closing `fi` is necessary, but the **`elif`** and/or the **`else`** clauses can be omitted.

The semicolons before **`then`** are standard syntax for combining two commands on a single line; they can be omitted only if **`then`** is moved to the next line.

It's important to understand that the brackets [[are not part of the syntax, but are treated as a command; it is the exit code from this command that is being tested. Therefore, you must always include spaces around the brackets.

这也意味着任何命令的结果都可以被测试。如果命令的退出码为零，则该语句被视为真。

```
if grep "foo" bar.txt; then
    echo "找到了foo"
else
    echo "未找到foo"
fi
```

数学表达式放在双括号内时，也会以相同方式返回0或1，并且也可以被测试：

```
if (( $1 + 5 > 91 )); then
    echo "$1 大于 86"
fi
```

你也可能会遇到带有单括号的 if 语句。这些语句在 POSIX 标准中有定义，并且保证在所有符合 POSIX 标准的 shell（包括 Bash）中都能正常工作。其语法与 Bash 中的非常相似：

```
if [ "$1" -eq 1 ]; then
    echo "第一个参数传入了 1"
elif [ "$1" -gt 2 ]; then
    echo "第一个参数没有传入2"
else
    echo "第一个参数既不是1，也不大于2。"
fi
```

第 10.3 节：遍历数组

for 循环：

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
done
```

或者

```
for ((i=0;i<${#arr[@]};i++));do
    echo "${arr[$i]}"
done
```

while 循环：

```
i=0
while [ $i -lt ${#arr[@]} ];do
    echo "${arr[$i]}"
    i=$(expr $i + 1)
done
```

或者

```
i=0
while (( $i < ${#arr[@]} ));do
    echo "${arr[$i]}"
    ((i++))
```


This also means that the result of any command can be tested. If the exit code from the command is a zero, the statement is considered true.

```
if grep "foo" bar.txt; then
    echo "foo was found"
else
    echo "foo was not found"
fi
```

Mathematical expressions, when placed inside double parentheses, also return 0 or 1 in the same way, and can also be tested:

```
if (( $1 + 5 > 91 )); then
    echo "$1 is greater than 86"
fi
```

You may also come across `if` statements with single brackets. These are defined in the POSIX standard and are guaranteed to work in all POSIX-compliant shells including Bash. The syntax is very similar to that in Bash:

```
if [ "$1" -eq 1 ]; then
    echo "1 was passed in the first parameter"
elif [ "$1" -gt 2 ]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

Section 10.3: Looping over an array

for loop:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
done
```

Or

```
for ((i=0;i<${#arr[@]};i++));do
    echo "${arr[$i]}"
done
```

while loop:

```
i=0
while [ $i -lt ${#arr[@]} ];do
    echo "${arr[$i]}"
    i=$(expr $i + 1)
done
```

Or

```
i=0
while (( $i < ${#arr[@]} ));do
    echo "${arr[$i]}"
    ((i++))
```

```
done
```

第10.4节：使用For循环遍历数字列表

```
#! /bin/bash

for i in {1..10}; do # {1..10} 展开为 "1 2 3 4 5 6 7 8 9 10"
    echo $i
done
```

输出如下：


```
done
```

Section 10.4: Using For Loop to List Iterate Over Numbers

```
#! /bin/bash

for i in {1..10}; do # {1..10} expands to "1 2 3 4 5 6 7 8 9 10"
    echo $i
done
```

This outputs the following:

```
1
2
3
4
5
6
7
8
8
10
```

Section 10.5: continue and break

Example for continue

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

Example for break

```
for i in [series]
do
    command 4
    if (condition) # Condition to break the loop
    then
            command 5 # Command if the loop needs to be broken
            break
    fi
    command 6 # Command to run if the "condition" is never true
done
```

Section 10.6: Loop break

Break multiple loop:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
```

```
    for j in "${arr[@]}";do
        echo "$j"
        break 2
    done
done
```

输出：

```
a
a
```

```
                )
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break
    done
done
```

输出：

```
a
a
b
a
c
a
d
a
e
a
f
a
```



```
    for j in "${arr[@]}";do
        echo "$j"
        break 2
    done
done
```

Output:

```
a
a
```

Break single loop:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break
    done
done
```

Output:

```
a
a
b
a
c
a
d
a
e
a
f
a
```

Section 10.7: While Loop

```
#! /bin/bash

i=0

while [ $i -lt 5 ] #While i is less than 5
do
    echo "i is currently $i"
    i=$[$i+1] #Not the lack of spaces around the brackets. This makes it a not a test expression
done #ends the loop
```

Watch that there are spaces around the brackets during the test (after the while statement). These spaces are necessary.

This loop outputs:

```
i is currently 0
i is currently 1
i is currently 2
i is currently 3
```

```
i当前是4
```

第10.8节：带C风格语法的For循环

C风格for循环的基本格式是：

```
for (( 变量赋值; 条件; 迭代过程 ))
```

注释：

- 在C风格的**for**循环中，变量的赋值可以包含空格，这与通常的赋值不同
- C风格的**for**循环中的变量前面不加$。

示例：

```
for (( i = 0; i < 10; i++ ))
do
    echo "当前迭代次数是 $i"
done
```

我们也可以在C风格的for循环中处理多个变量：

```
for (( i = 0, j = 0; i < 10; i++, j = i * i ))
do
    echo "$i 的平方等于 $j"
done
```

第10.9节：Until循环

Until循环会一直执行直到条件为真

```
i=5
until [[ i -eq 10 ]]; do #检查i是否等于10
    echo "i=$i" #打印i的值
    i=$((i+1)) #i加1
done
```

输出：

```
i=5
i=6
i=7
i=8
i=9
```

当 i 达到10时，until循环中的条件变为真，循环结束。

第10.10节：带case的switch语句

使用case语句可以将值与一个变量进行匹配。

传递给case的参数会被展开，并尝试与每个模式匹配。

如果找到匹配项，则执行直到;;之前的命令。


```
i is currently 4
```

Section 10.8: For Loop with C-style syntax

The basic format of C-style **for** loop is:

```
for (( variable assignment; condition; iteration process ))
```

Notes:

- The assignment of the variable inside C-style **for** loop can contain spaces unlike the usual assignment
- Variables inside C-style **for** loop aren't preceded with $.

Example:

```
for (( i = 0; i < 10; i++ ))
do
    echo "The iteration number is $i"
done
```

Also we can process multiple variables inside C-style **for** loop:

```
for (( i = 0, j = 0; i < 10; i++, j = i * i ))
do
    echo "The square of $i is equal to $j"
done
```

Section 10.9: Until Loop

Until loop executes until condition is true

```
i=5
until [[ i -eq 10 ]]; do #Checks if i=10
    echo "i=$i" #Print the value of i
    i=$((i+1)) #Increment i by 1
done
```

Output:

```
i=5
i=6
i=7
i=8
i=9
```

When `i` reaches 10 the condition in until loop becomes true and the loop ends.

Section 10.10: Switch statement with case

With the **case** statement you can match values against one variable.

The argument passed to **case** is expanded and try to match against each patterns.

If a match is found, the commands upto `;;` are executed.

```
case "$BASH_VERSION" in
 [34]*)
    echo {1..4}
    ;;
  *)
    seq -s" " 1 4
esac
```

模式不是正则表达式，而是 shell 模式匹配（也称为通配符）。

第10.11节：没有单词列表参数的for循环

```
for arg; do
    echo arg=$arg
done
```

没有单词列表参数的**for**循环将改为遍历位置参数。换句话说，
上述示例等同于以下代码：

```
for arg in "$@"; do
    echo arg=$arg
done
```

换句话说，如果你发现自己写了`for i in "$@"; do ...; done`，只需去掉`in`部分，直接写成`for i; do ...; done`即可。


```
case "$BASH_VERSION" in
 [34]*)
    echo {1..4}
    ;;
  *)
    seq -s" " 1 4
esac
```

Pattern are not regular expressions but shell pattern matching (aka globs).

Section 10.11: For Loop without a list-of-words parameter

```
for arg; do
    echo arg=$arg
done
```

A **for** loop without a list of words parameter will iterate over the positional parameters instead. In other words, the above example is equivalent to this code:

```
for arg in "$@"; do
    echo arg=$arg
done
```

In other words, if you catch yourself writing `for i in "$@"; do ...; done`, just drop the `in` part, and write simply `for i; do ...; done`.

第11章：true、false 和 ： 命令

第11.1节：无限循环

```
while true; do
    echo ok
done
```

或者

```
while :; do
   echo ok
done
```

或者

```
直到 false; 执行
    echo 确认
done
```

第11.2节：函数返回

```
函数 positive() {
    返回 0
}

函数 negative() {
    返回 1
}
```

第11.3节：总是/永不执行的代码

```
如果 true; 则
    echo 总是执行
fi
如果 false; 则
    echo 永不执行
fi
```


Chapter 11: true, false and : commands

Section 11.1: Infinite Loop

```
while true; do
    echo ok
done
```

or

```
while :; do
   echo ok
done
```

or

```
until false; do
    echo ok
done
```

Section 11.2: Function Return

```
function positive() {
    return 0
}

function negative() {
    return 1
}
```

Section 11.3: Code that will always/never be executed

```
if true; then
    echo Always executed
fi
if false; then
    echo Never executed
fi
```

第12章：数组

第12.1节：数组赋值

列表赋值

如果你熟悉 Perl、C 或 Java，可能会认为 Bash 会用逗号分隔数组元素，
但事实并非如此；Bash 使用空格分隔：

```
# Perl 中的数组
my @array = (1, 2, 3, 4);
```

```
# Bash 中的数组
array=(1 2 3 4)
```

创建一个包含新元素的数组：

```
array=('第一个元素' '第二个元素' '第三个元素')
```

下标赋值

创建一个带有显式元素索引的数组：

```
array=([3]='第四个元素' [4]='第五个元素')
```

按索引赋值

```
array[0]='第一个元素'
array[1]='第二个元素'
```

按名称赋值（关联数组）

版本 ≥ 4.0

```
declare -A array
array[first]='第一个元素'
array[second]='第二个元素'
```

动态赋值

从其他命令的输出创建数组，例如使用seq获取从1到10的范围：

```
array=(`seq 1 10`)
```

从脚本的输入参数赋值：

```
array=("$@")
```

循环内赋值：

```
while read -r; do
    #array+=("$REPLY")     # 数组追加
    array[$i]="$REPLY"     # 按索引赋值
    let i++                # 索引递增
done < <(seq 1 10)  # 命令替换
```


Chapter 12: Arrays

Section 12.1: Array Assignments

List Assignment

If you are familiar with Perl, C, or Java, you might think that Bash would use commas to separate array elements, however this is not the case; instead, Bash uses spaces:

```
# Array in Perl
my @array = (1, 2, 3, 4);
```

```
# Array in Bash
array=(1 2 3 4)
```

Create an array with new elements:

```
array=('first element' 'second element' 'third element')
```

Subscript Assignment

Create an array with explicit element indices:

```
array=([3]='fourth element' [4]='fifth element')
```

Assignment by index

```
array[0]='first element'
array[1]='second element'
```

Assignment by name (associative array)

Version ≥ 4.0

```
declare -A array
array[first]='First element'
array[second]='Second element'
```

Dynamic Assignment

Create an array from the output of other command, for example use **seq** to get a range from 1 to 10:

```
array=(`seq 1 10`)
```

Assignment from script's input arguments:

```
array=("$@")
```

Assignment within loops:

```
while read -r; do
    #array+=("$REPLY")     # Array append
    array[$i]="$REPLY"     # Assignment by index
    let i++                # Increment index
done < <(seq 1 10)  # command substitution
```

```
echo ${array[@]}    # 输出: 1 2 3 4 5 6 7 8 9 10
```

where $REPLY 始终是当前输入

第12.2节：访问数组元素

打印索引为0的元素

```
echo "${array[0]}"
```

版本 < 4.3

使用子串扩展语法打印最后一个元素

```
echo "${arr[@]: -1 }"
```

版本 ≥ 4.3

使用下标语法打印最后一个元素

```
echo "${array[-1]}"
```

分别打印所有元素，每个元素单独加引号

```
echo "${array[@]}"
```

将所有元素作为一个单独的带引号字符串打印

```
echo "${array[*]}"
```

从索引1开始，分别打印所有元素，每个元素单独加引号

```
echo "${array[@]:1}"
```

从索引1开始，分别打印3个元素，每个元素单独加引号

```
echo "${array[@]:1:3}"
```

字符串操作

如果指的是单个元素，则允许进行字符串操作：

```
array=(zero one two)
echo "${array[0]:0:3}" # 输出 zer（字符串 zero 中位置 0、1 和 2 的字符）
echo "${array[0]:1:3}" # 输出 ero（字符串 zero 中位置 1、2 和 3 的字符）
```

所以 ${array[$i]:N:M} 从字符串 ${array[$i]} 的第 N 个位置（从 0 开始）开始，输出 M 个后续字符组成的字符串。

第 12.3 节：数组修改

更改索引

初始化或更新数组中的特定元素

```
array[10]="第十一元素"    # 因为索引从 0 开始
```



```
echo ${array[@]}    # output: 1 2 3 4 5 6 7 8 9 10
```

where $REPLY is always the current input

Section 12.2: Accessing Array Elements

Print element at index 0

```
echo "${array[0]}"
```

Version < 4.3

Print last element using substring expansion syntax

```
echo "${arr[@]: -1 }"
```

Version ≥ 4.3

Print last element using subscript syntax

```
echo "${array[-1]}"
```

Print all elements, each quoted separately

```
echo "${array[@]}"
```

Print all elements as a single quoted string

```
echo "${array[*]}"
```

Print all elements from index 1, each quoted separately

```
echo "${array[@]:1}"
```

Print 3 elements from index 1, each quoted separately

```
echo "${array[@]:1:3}"
```

String Operations

If referring to a single element, string operations are permitted:

```
array=(zero one two)
echo "${array[0]:0:3}" # gives out zer (chars at position 0, 1 and 2 in the string zero)
echo "${array[0]:1:3}" # gives out ero (chars at position 1, 2 and 3 in the string zero)
```

so ${array[$i]:N:M} gives out a string from the Nth position (starting from 0) in the string ${array[$i]} with M following chars.

Section 12.3: Array Modification

Change Index

Initialize or update a particular element in the array

```
array[10]="elevenths element"    # because it's starting with 0
```

版本 ≥ 3.1

追加

修改数组，如果未指定下标，则在末尾添加元素。

```
array+=('第四个元素' '第五个元素')
```

用新的参数列表替换整个数组。

```
array=("${array[@]}" "第四个元素" "第五个元素")
```

在开头添加一个元素：

```
array=("新元素" "${array[@]}")
```

插入

在指定索引插入一个元素：

```
arr=(a b c d)
# 在索引2插入一个元素
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" #output: new
```

删除

使用unset内置命令删除数组索引：

```
arr=(a b c)
echo "${arr[@]}"   # 输出：a b c
echo "${!arr[@]}"  # 输出：0 1 2
unset -v 'arr[1]'
echo "${arr[@]}"   # 输出：a c
echo "${!arr[@]}"  # 输出：0 2
```

合并

```
array3=("${array1[@]}" "${array2[@]}")
```

这对稀疏数组也适用。

数组重新索引

如果数组中有元素被删除，或者不确定数组中是否存在空缺，这会很有用。要重新创建没有空缺的索引：

```
array=("${array[@]}")
```

第12.4节：数组迭代

数组迭代有两种形式，foreach和经典的for循环：

```
a=(1 2 3 4)
# foreach循环
```


Version ≥ 3.1

Append

Modify array, adding elements to the end if no subscript is specified.

```
array+=('fourth element' 'fifth element')
```

Replace the entire array with a new parameter list.

```
array=("${array[@]}" "fourth element" "fifth element")
```

Add an element at the beginning:

```
array=("new element" "${array[@]}")
```

Insert

Insert an element at a given index:

```
arr=(a b c d)
# insert an element at index 2
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" #output: new
```

Delete

Delete array indexes using the `unset` builtin:

```
arr=(a b c)
echo "${arr[@]}"   # outputs: a b c
echo "${!arr[@]}"  # outputs: 0 1 2
unset -v 'arr[1]'
echo "${arr[@]}"   # outputs: a c
echo "${!arr[@]}"  # outputs: 0 2
```

Merge

```
array3=("${array1[@]}" "${array2[@]}")
```

This works for sparse arrays as well.

Re-indexing an array

This can be useful if elements have been removed from an array, or if you're unsure whether there are gaps in the array. To recreate the indices without gaps:

```
array=("${array[@]}")
```

Section 12.4: Array Iteration

Array iteration comes in two flavors, foreach and the classic for-loop:

```
a=(1 2 3 4)
# foreach loop
```

```
for y in "${a[@]}"; do
    # 对$y进行操作
    echo "$y"
done
# 经典for循环
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # 对${a[$idx]}进行操作
    echo "${a[$idx]}"
done
```

你也可以迭代命令的输出：

```
a=($(tr ',' ' ' <<<"a,b,c,d")) # tr 可以将一个字符转换为另一个字符
for y in "${a[@]}"; do
    echo "$y"
done
```

第12.5节：数组长度

${#array[@]} 返回数组的长度 ${array[@]}：

```
array=('第一个元素' '第二个元素' '第三个元素')
echo "${#array[@]}" # 输出长度为3
```

这也适用于单个元素中的字符串：

```
echo "${#array[0]}"    # 输出元素0中字符串的长度：13
```

第12.6节：关联数组

版本 ≥ 4.0

声明一个关联数组

```
declare -A aa
```

在初始化或使用之前，必须声明关联数组。

初始化元素

您可以按如下方式逐个初始化元素：

```
aa[hello]=world
aa[ab]=cd
aa["带空格的键"]="hello world"
```

您也可以在一条语句中初始化整个关联数组：

```
aa=([hello]=world [ab]=cd ["带空格的键"]="hello world")
```

访问关联数组元素

```
echo ${aa[hello]}
# 输出: world
```

列出关联数组的键


```
for y in "${a[@]}"; do
    # act on $y
    echo "$y"
done
# classic for-loop
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # act on ${a[$idx]}
    echo "${a[$idx]}"
done
```

You can also iterate over the output of a command:

```
a=($(tr ',' ' ' <<<"a,b,c,d")) # tr can transform one character to another
for y in "${a[@]}"; do
    echo "$y"
done
```

Section 12.5: Array Length

`${#array[@]}` gives the length of the array `${array[@]}`:

```
array=('first element' 'second element' 'third element')
echo "${#array[@]}" # gives out a length of 3
```

This works also with Strings in single elements:

```
echo "${#array[0]}"    # gives out the lenght of the string at element 0: 13
```

Section 12.6: Associative Arrays

Version ≥ 4.0

Declare an associative array

```
declare -A aa
```

Declaring an associative array before initialization or use is mandatory.

Initialize elements

You can initialize elements one at a time as follows:

```
aa[hello]=world
aa[ab]=cd
aa["key with space"]="hello world"
```

You can also initialize an entire associative array in a single statement:

```
aa=([hello]=world [ab]=cd ["key with space"]="hello world")
```

Access an associative array element

```
echo ${aa[hello]}
# Out: world
```

Listing associative array keys

```
echo "${!aa[@]}"
#输出: hello ab 带空格的键
```

列出关联数组的值

```
echo "${aa[@]}"
#输出: world cd hello world
```

遍历关联数组的键和值

```
for key in "${!aa[@]}"; do
    echo "键:   ${key}"
    echo "值: ${array[$key]}"
done

# 输出:
# 键:   hello
# 值: world
# 键:   ab
# 值: cd
# 键:   key with space
# 值: hello world
```

计算关联数组元素数量

```
echo "${#aa[@]}"
# 输出: 3
```

第12.7节：遍历数组

我们的示例数组：

```
arr=(a b c d e f)
```

使用for..in循环：

```
for i in "${arr[@]}"; do
    echo "$i"
done
```

版本 ≥ 2.04

使用C风格的for循环：

```
for ((i=0;i<${#arr[@]};i++)); do
    echo "${arr[$i]}"
done
```

使用while循环：

```
i=0
while [ $i -lt ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

版本 ≥ 2.04


```
echo "${!aa[@]}"
#Out: hello ab key with space
```

Listing associative array values

```
echo "${aa[@]}"
#Out: world cd hello world
```

Iterate over associative array keys and values

```
for key in "${!aa[@]}"; do
    echo "Key:   ${key}"
    echo "Value: ${array[$key]}"
done

# Out:
# Key:   hello
# Value: world
# Key:   ab
# Value: cd
# Key:   key with space
# Value: hello world
```

Count associative array elements

```
echo "${#aa[@]}"
# Out: 3
```

Section 12.7: Looping through an array

Our example array:

```
arr=(a b c d e f)
```

Using a `for..in` loop:

```
for i in "${arr[@]}"; do
    echo "$i"
done
```

Version ≥ 2.04

Using C-style **`for`** loop:

```
for ((i=0;i<${#arr[@]};i++)); do
    echo "${arr[$i]}"
done
```

Using **`while`** loop:

```
i=0
while [ $i -lt ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

Version ≥ 2.04

使用带数字条件的while循环：

```
i=0
while (( $i < ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
完成
```

使用until循环：

```
i=0
until [ $i -ge ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

版本 ≥ 2.04

使用带数值条件的until循环：

```
i=0
until (( $i >= ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
完成
```

第12.8节：销毁、删除或取消设置数组

销毁、删除或取消设置数组：

```
unset array
```

销毁、删除或取消设置单个数组元素：

```
unset array[10]
```

第12.9节：从字符串创建数组

```
stringVar="Apple Orange Banana Mango"
arrayVar=(${stringVar// / })
```

字符串中的每个空格表示结果数组中的一个新元素。

```
echo ${arrayVar[0]} # 将打印 Apple
echo ${arrayVar[3]} # 将打印 Mango
```

同样，其他字符也可以用作分隔符。

```
stringVar="Apple+Orange+Banana+Mango"
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # 将打印 Apple
echo ${arrayVar[2]} # 将打印 Banana
```

第12.10节：已初始化索引列表

获取数组中已初始化索引的列表

Using **`while`** loop with numerical conditional:

```
i=0
while (( $i < ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Using an **`until`** loop:

```
i=0
until [ $i -ge ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

Version ≥ 2.04

Using an **`until`** loop with numerical conditional:

```
i=0
until (( $i >= ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Section 12.8: Destroy, Delete, or Unset an Array

To destroy, delete, or unset an array:

```
unset array
```

To destroy, delete, or unset a single array element:

```
unset array[10]
```

Section 12.9: Array from string

```
stringVar="Apple Orange Banana Mango"
arrayVar=(${stringVar// / })
```

Each space in the string denotes a new item in the resulting array.

```
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[3]} # will print Mango
```

Similarly, other characters can be used for the delimiter.

```
stringVar="Apple+Orange+Banana+Mango"
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[2]} # will print Banana
```

Section 12.10: List of initialized indexes

Get the list of inialized indexes in an array

```
$ arr[2]='second'
$ arr[10]='tenth'
$ arr[25]='twenty five'
$ echo ${!arr[@]}
2 10 25
```

第12.11节：将整个文件读入数组

一次性读取：

```
IFS=$" read -r -a arr < file
```

循环读取：

```
arr=()
while IFS= read -r line; do
  arr+=("$line")
done
```

版本 ≥ 4.0

使用mapfile或readarray（两者同义）：

```
mapfile -t arr < file
readarray -t arr < file
```

第12.12节：数组插入函数

此函数将在给定索引处向数组插入一个元素：

```
insert(){
    h='
################## 插入 ########################
# 用法:
#   insert 数组名 索引 元素
#
#   参数:
#       数组名    ：数组变量的名称
#       索引      ：插入的位置
#       元素      ：要插入的元素
################################################
    '
    [[ $1 = -h ]] && { echo "$h" >/dev/stderr; return 1; }
    declare -n __arr__=$1    # 引用数组变量
    i=$2                     # 插入的索引
    el="$3"                  # 要插入的元素
    # 处理错误
    [[ ! "$i" =~ ^[0-9]+$ ]] && { echo "E: insert: 索引必须是有效的整数" >/dev/stderr;
return 1; }
    (( $1 < 0 )) && { echo "E: insert: index 不能为负数" >/dev/stderr; return 1; }
    # 现在在 $i 处插入 $el
    __arr__=("${__arr__[@]:0:$i}" "$el" "${__arr__[@]:$i}")
}
```

用法：

```
insert array_variable_name index element
```



```
$ arr[2]='second'
$ arr[10]='tenth'
$ arr[25]='twenty five'
$ echo ${!arr[@]}
2 10 25
```


Section 12.11: Reading an entire file into an array

Reading in a single step:

```
IFS=$'\n' read -r -a arr < file
```

Reading in a loop:

```
arr=()
while IFS= read -r line; do
  arr+=("$line")
done
```

Version ≥ 4.0

Using `mapfile` or `readarray` (which are synonymous):

```
mapfile -t arr < file
readarray -t arr < file
```

Section 12.12: Array insert function

This function will insert an element into an array at a given index:

```
insert(){
    h='
################## insert ########################
# Usage:
#   insert arr_name index element
#
#   Parameters:
#       arr_name    : Name of the array variable
#       index       : Index to insert at
#       element     : Element to insert
################################################
    '
    [[ $1 = -h ]] && { echo "$h" >/dev/stderr; return 1; }
    declare -n __arr__=$1    # reference to the array variable
    i=$2                     # index to insert at
    el="$3"                  # element to insert
    # handle errors
    [[ ! "$i" =~ ^[0-9]+$ ]] && { echo "E: insert: index must be a valid integer" >/dev/stderr;
return 1; }
    (( $1 < 0 )) && { echo "E: insert: index can not be negative" >/dev/stderr; return 1; }
    # Now insert $el at $i
    __arr__=("${__arr__[@]:0:$i}" "$el" "${__arr__[@]:$i}")
}
```

Usage:

```
insert array_variable_name index element
```

示例：

```
arr=(a b c d)
echo "${arr[2]}" # 输出：c
# 现在调用 insert 函数并传入数组变量名，
# 要插入的索引
# 以及要插入的元素
insert arr 2 'New Element'
# 'New Element' 已插入到 arr 的索引 2 位置，现在打印它们
echo "${arr[2]}" # 输出：New Element
echo "${arr[3]}" # 输出：c
```


Example:

```
arr=(a b c d)
echo "${arr[2]}" # output: c
# Now call the insert function and pass the array variable name,
# index to insert at
# and the element to insert
insert arr 2 'New Element'
# 'New Element' was inserted at index 2 in arr, now print them
echo "${arr[2]}" # output: New Element
echo "${arr[3]}" # output: c
```

第13章：关联数组

第13.1节：检查关联数组

所有需要的用法都在此代码片段中展示：

```
#!/usr/bin/env bash

declare -A assoc_array=([key_string]=值                                    \
                        [one]="something"                                    \
                        [two]="another thing"                                \
                        [ three ]='注意空格！'                                \
                        [ " four" ]='稍后计算此键的空格数！'    \
                                                  [IMPORTANT]='空格是会累加的！！！'
 \
                        [1]='没有整数！'                                    \
                        [info]="为避免历史扩展 "                            \
                        [info2]="带单引号的感叹号引述"
                       )
echo # 只是一个空行
echo 现在这是 assoc_array 的值：
echo ${assoc_array[@]}
echo 不是很有用，
echo # 只是一个空行
echo 这个更好：

declare -p assoc_array     # -p == 打印

echo 仔细看看上面的空格\!\!\!
echo # 只是一个空行

echo 访问键
echo assoc_array 中的键是 ${!assoc_array[*]}
echo 注意间接引用操作符的使用 \!
echo # 只是一个空行


echo 现在我们逐行遍历 assoc_array
echo 注意 \! 间接引用操作符 的工作方式不同，
echo 如果 用于 assoc_array。
echo # 只是一个空行

for key in "${!assoc_array[@]}"; do # 使用 ! 间接引用访问键！！！！
    printf "key: \"%s\"value: \"%s\"" "$key" "${assoc_array[$key]}"done



echo 仔细观察上面键为 two、three 和 four 的条目中的空格！！！
echo # 只是一个空行
echo # 只是另一个空行

echo 使用整数作为键是有区别的！！！
i=1
echo 声明一个整数变量 i=1
echo # 只是一个空行
echo 在 integer_array 中 bash 识别算术上下文。
echo 在 assoc_array 中 bash 不识别算术上下文。
echo # 只是一个空行
echo 这样可以： \${assoc_array[\$i]}:  ${assoc_array[$i]}
echo 这样不行!!: \${assoc_array[i]}:  ${assoc_array[i]}
```


Chapter 13: Associative arrays

Section 13.1: Examining assoc arrays

All needed usage shown with this snippet:

```
#!/usr/bin/env bash

declare -A assoc_array=([key_string]=value                                    \
                        [one]="something"                                     \
                        [two]="another thing"                                 \
                        [ three ]='mind the blanks!'                          \
                        [ " four" ]='count the blanks of this key later!'     \
                                                  [IMPORTANT]='SPACES DO ADD UP!!!'
 \
                        [1]='there are no integers!'                          \
                        [info]="to avoid history expansion "                  \
                        [info2]="quote exclamation mark with single quotes"   \
                       )
echo # just a blank line
echo now here are the values of assoc_array:
echo ${assoc_array[@]}
echo not that useful,
echo # just a blank line
echo this is better:

declare -p assoc_array    # -p == print

echo have a close look at the spaces above\!\!\!
echo # just a blank line

echo accessing the keys
echo the keys in assoc_array are ${!assoc_array[*]}
echo mind the use of indirection operator \!
echo # just a blank line


echo now we loop over the assoc_array line by line
echo note the \! indirection operator which works differently,
echo if used with assoc_array.
echo # just a blank line

for key in "${!assoc_array[@]}"; do # accessing keys using ! indirection!!!!
   printf "key: \"%s\"\nvalue: \"%s\"\n\n" "$key" "${assoc_array[$key]}"
done



echo have a close look at the spaces in entries with keys two, three and four above\!\!\!
echo # just a blank line
echo # just another blank line

echo there is a difference using integers as keys\!\!\!
i=1
echo declaring an integer var i=1
echo # just a blank line
echo Within an integer_array bash recognizes artithmetic context.
echo Within an assoc_array bash DOES NOT recognize artithmetic context.
echo # just a blank line
echo this works: \${assoc_array[\$i]}:  ${assoc_array[$i]}
echo this NOT!!: \${assoc_array[i]}:  ${assoc_array[i]}
```

```
echo # 只是一个空行
echo # 只是一个空行
echo 一个 \${assoc_array[i]} 在大括号内具有字符串上下文，与 integer_array 相反
declare -i integer_array=( one two three )
echo "doing a: declare -i integer_array=( one two three )"
echo # 只是一个空行

echo 两种形式 都 可用： \${integer_array[i]} ： ${integer_array[i]}
echo 这也可以：  \${integer_array[\$i]} ： ${integer_array[$i]}
```



```
echo # just a blank line
echo # just a blank line
echo an \${assoc_array[i]} has a string context within braces in contrast to an integer_array
declare -i integer_array=( one two three )
echo "doing a: declare -i integer_array=( one two three )"
echo # just a blank line

echo both forms do work: \${integer_array[i]} : ${integer_array[i]}
echo and this too:  \${integer_array[\$i]} : ${integer_array[$i]}
```

第14章：函数

第14.1节：带参数的函数

在 helloJohn.sh 中：

```
#!/bin/bash

greet() {
   local name="$1"
   echo "你好，$name"
}

greet "约翰·多伊"
```

```
# 运行上述脚本
$ bash helloJohn.sh
你好，约翰·多伊
```

1. 如果你不对参数进行任何修改，就不需要复制到 local 变量中——直接 echo 即可“你好，$1”。
2. 你可以使用$1、$2、$3等来访问函数内部的参数。

> **注意：**对于超过9个的参数，$10不起作用（bash会将其识别为**$1**0），你需要使用${10}、${11}等。

3. $@表示函数的所有参数：

```
#!/bin/bash
foo() {
  echo "$@"
}

foo 1 2 3 # 输出 => 1 2 3
```

> 注意：你实际上应该总是像这里一样在"$@"周围使用双引号。

省略引号会导致shell展开通配符（即使用户特意用引号避免了这种情况），通常会引入不受欢迎的行为，甚至可能带来安全问题。

```
foo "带空格的字符串;" '$HOME' "*"
# 输出 => 带空格的字符串; $HOME *
```

4. 默认参数使用${1:-default_val}。例如：

```
#!/bin/bash
foo() {
  local val=${1:-25}
  echo "$val"
}
```


Chapter 14: Functions

Section 14.1: Functions with arguments

In helloJohn.sh:

```
#!/bin/bash

greet() {
  local name="$1"
  echo "Hello, $name"
}

greet "John Doe"
```

```
# running above script
$ bash helloJohn.sh
Hello, John Doe
```

1. If you don't modify the argument in any way, there is no need to copy it to a **local** variable - simply **echo** "Hello, $1".
2. You can use $1, $2, $3 and so on to access the arguments inside the function.

> **Note:** for arguments more than 9 $10 won't work (bash will read it as **$1**0), you need to do ${10}, ${11} and so on.

3. $@ refers to all arguments of a function:

```
#!/bin/bash
foo() {
  echo "$@"
}

foo 1 2 3 # output => 1 2 3
```

> **Note:** You should practically always use double quotes around "$@", like here.

Omitting the quotes will cause the shell to expand wildcards (even when the user specifically quoted them in order to avoid that) and generally introduce unwelcome behavior and potentially even security problems.

```
foo "string with spaces;" '$HOME' "*"
# output => string with spaces; $HOME *
```

4. for default arguments use ${1:-default_val}. Eg:

```
#!/bin/bash
foo() {
  local val=${1:-25}
  echo "$val"
}
```

```
foo     # 输出 => 25
foo 30  # 输出 => 30
```

5. 要要求参数，使用 ${var:?error message}

```
foo() {
   local val=${1:?必须提供参数}
   echo "$val"
}
```

第14.2节：简单函数

在 helloWorld.sh 中

```
#!/bin/bash

# 定义一个函数 greet
greet ()
{
    echo "Hello World!"
}

# 调用函数 greet
greet
```

运行脚本时，我们会看到我们的信息

```
$ bash helloWorld.sh
Hello World!
```

注意，通过 source 一个包含函数的文件，可以使这些函数在当前的 bash 会话中可用。

```
$ source helloWorld.sh   # 或者，更通用的写法，". helloWorld.sh"
$ greet
Hello World!
```

你可以在某些 shell 中 export 一个函数，使其对子进程可见。

```
bash -c 'greet'  # 失败
export -f greet  # 导出函数；注意 -f
bash -c 'greet'  # 成功
```

第14.3节：处理标志和可选参数

内置命令getopts可以在函数内部使用，以编写支持标志和可选参数的函数。这没有特别的难度，但必须适当处理getopts所涉及的值。
例如，我们定义了一个failwith函数，该函数在stderr上写入一条消息，并以代码1或作为-x选项参数提供的任意代码退出：

```
# failwith [-x STATUS] 类printf参数
#  使用给定的诊断消息失败退出
#
# -x 标志可用于传递自定义退出状态，而不是
# 值1。输出会自动添加换行符。

failwith()
```



```
foo     # output => 25
foo 30  # output => 30
```

5. to require an argument use ${var:?error message}

```
foo() {
  local val=${1:?Must provide an argument}
  echo "$val"
}
```

Section 14.2: Simple Function

In helloWorld.sh

```
#!/bin/bash

# Define a function greet
greet ()
{
    echo "Hello World!"
}

# Call the function greet
greet
```

In running the script, we see our message

```
$ bash helloWorld.sh
Hello World!
```

Note that sourcing a file with functions makes them available in your current bash session.

```
$ source helloWorld.sh   # or, more portably, ". helloWorld.sh"
$ greet
Hello World!
```

You can **export** a function in some shells, so that it is exposed to child processes.

```
bash -c 'greet'  # fails
export -f greet  # export function; note -f
bash -c 'greet'  # success
```

Section 14.3: Handling flags and optional parameters

The *getopts* builtin can be used inside functions to write functions that accommodate flags and optional parameters. This presents no special difficulty but one has to handle appropriately the values touched by *getopts*. As an example, we define a *failwith* function that writes a message on *stderr* and exits with code 1 or an arbitrary code supplied as parameter to the `-x` option:

```
# failwith [-x STATUS] PRINTF-LIKE-ARGV
#  Fail with the given diagnostic message
#
# The -x flag can be used to convey a custom exit status, instead of
# the value 1.  A newline is automatically added to the output.

failwith()
```

```
{
    local OPTIND OPTION OPTARG status

    status=1
    OPTIND=1

    while getopts 'x:' OPTION; do
        case ${OPTION} in
x)   status="${OPTARG}";;
            *)    1>&2 printf 'failwith: %s: 不支持的选项。' "${OPTION}";;esac

    done

    shift $(( OPTIND - 1 ))
    {
        printf '失败: '
        printf "$@"
        printf "} 1>
    &2
    exit "${status}"
}
```

此函数可以按如下方式使用：

```
failwith '%s: 文件未找到.' "${filename}"
failwith -x 70 '一般内部错误.'
```

等等。

注意，对于printf，变量不应作为第一个参数使用。如果要打印的消息由变量内容组成，应使用%s格式说明符来打印，如下所示

```
failwith '%s' "${message}"
```

第14.4节：打印函数定义

```
getfunc() {
    declare -f "$@"
}

function func(){
    echo "我是一个示例函数"
}

funcd="$(getfunc func)"
getfunc func # 或者 echo "$funcd"
```

输出：

```
func ()
{
    echo "我是一个示例函数"
}
```

第14.5节：接受命名参数的函数

```
foo() {
  while [[ "$#" -gt 0 ]]
```



```
{
    local OPTIND OPTION OPTARG status

    status=1
    OPTIND=1

    while getopts 'x:' OPTION; do
        case ${OPTION} in
            x)    status="${OPTARG}";;
            *)    1>&2 printf 'failwith: %s: Unsupported option.\n' "${OPTION}";;
        esac
    done

    shift $(( OPTIND - 1 ))
    {
        printf 'Failure: '
        printf "$@"
        printf '\n'
    } 1>&2
    exit "${status}"
}
```

This function can be used as follows:

```
failwith '%s: File not found.' "${filename}"
failwith -x 70 'General internal error.'
```

and so on.

Note that as for *printf*, variables should not be used as first argument. If the message to print consists of the content of a variable, one should use the `%s` specifier to print it, like in

```
failwith '%s' "${message}"
```

Section 14.4: Print the function definition

```
getfunc() {
    declare -f "$@"
}

function func(){
    echo "I am a sample function"
}

funcd="$(getfunc func)"
getfunc func # or echo "$funcd"
```

Output:

```
func ()
{
    echo "I am a sample function"
}
```

Section 14.5: A function that accepts named parameters

```
foo() {
  while [[ "$#" -gt 0 ]]
```

```
  do
    case $1 in
      -f|--follow)
        local FOLLOW="following"
        ;;
      -t|--tail)
        local TAIL="tail=$2"
        ;;
    esac
    shift
  done

  echo "FOLLOW: $FOLLOW"
  echo "TAIL: $TAIL"
}
```

示例用法：

```
foo -f
foo -t 10
foo -f --tail 10
foo --follow --tail 10
```

第14.6节：函数的返回值

Bash 中的return语句不像 C 函数那样返回一个值，而是以返回状态退出函数。你可以将其视为该函数的退出状态。

如果你想从函数返回一个值，那么像这样将值发送到stdout：

```
fun() {
    local var="要返回的示例值"
    echo "$var"
    #printf "%s" "$var"
}
```

现在，如果你执行：

```
var="$(fun)"
```

函数fun的输出将被存储在$var中。

第14.7节：函数的退出码是其最后一个命令的退出码

考虑这个示例函数，用于检查主机是否在线：

```
is_alive() {
    ping -c1 "$1" &> /dev/null
}
```

该函数向第一个函数参数指定的主机发送一次ping。ping的标准输出和错误输出都重定向到/dev/null，因此函数不会输出任何内容。但ping命令成功时退出码为0，失败时为非零。由于这是函数中的最后（且在此示例中唯一）命令，ping的退出码将作为函数本身的退出码。


```
  do
    case $1 in
      -f|--follow)
        local FOLLOW="following"
        ;;
      -t|--tail)
        local TAIL="tail=$2"
        ;;
    esac
    shift
  done

  echo "FOLLOW: $FOLLOW"
  echo "TAIL: $TAIL"
}
```

Example usage:

```
foo -f
foo -t 10
foo -f --tail 10
foo --follow --tail 10
```

Section 14.6: Return value from a function

The `return` statement in Bash doesn't return a value like C-functions, instead it exits the function with a return status. You can think of it as the exit status of that function.

If you want to return a value from the function then send the value to `stdout` like this:

```
fun() {
    local var="Sample value to be returned"
    echo "$var"
    #printf "%s\n" "$var"
}
```

Now, if you do:

```
var="$(fun)"
```

the output of `fun` will be stored in `$var`.

Section 14.7: The exit code of a function is the exit code of its last command

Consider this example function to check if a host is up:

```
is_alive() {
    ping -c1 "$1" &> /dev/null
}
```

This function sends a single ping to the host specified by the first function parameter. The output and error output of `ping` are both redirected to `/dev/null`, so the function will never output anything. But the `ping` command will have exit code 0 on success, and non-zero on failure. As this is the last (and in this example, the only) command of the function, the exit code of `ping` will be reused for the exit code of the function itself.

这一点在条件语句中非常有用。

例如，如果主机graucho在线，则使用ssh连接它：

```
if is_alive graucho; then
    ssh graucho
fi
```

另一个例子：反复检查直到主机graucho上线，然后使用ssh连接它：

```
while ! is_alive graucho; do
    sleep 5
done
ssh graucho
```


This fact is very useful in conditional statements.

For example, if host `graucho` is up, then connect to it with **`ssh`**:

```
if is_alive graucho; then
    ssh graucho
fi
```

Another example: repeatedly check until host `graucho` is up, and then connect to it with **`ssh`**:

```
while ! is_alive graucho; do
    sleep 5
done
ssh graucho
```

第15章：Bash参数扩展

字符$引入参数扩展、命令替换或算术扩展。要扩展的参数名称或符号可以用大括号括起来，括号是可选的，但用于保护要扩展的变量，防止紧跟其后的字符被误解为名称的一部分。

在Bash用户手册中了解更多。

第15.1节：修改字母字符的大小写

版本 ≥ 4.0

转换为大写

```
$ v="hello"
# 仅第一个字符
$ printf '%s' "${v^}"Hello

# 所有字符
$ printf '%s' "${v^^}"HELLO

# 另一种方法
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

转换为小写

```
$ v="BYE"
# 仅第一个字符
$ printf '%s' "${v,}"bYE

# 所有字符
$ printf '%s' "${v,,}"bye

# 另一种方法
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

切换大小写

```
$ v="Hello World"
# 所有字符
$ echo "${v~~}"
hELLO wORLD
$ echo "${v~}"
# 仅第一个字符
hello World
```

第15.2节：参数长度

```
# 字符串长度
$ var='12345'
$ echo "${#var}"
```


Chapter 15: Bash Parameter Expansion

The $ character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

Read more in the Bash User Manual.

Section 15.1: Modifying the case of alphabetic characters

Version ≥ 4.0

To uppercase

```
$ v="hello"
# Just the first character
$ printf '%s\n' "${v^}"
Hello
# All characters
$ printf '%s\n' "${v^^}"
HELLO
# Alternative
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

To lowercase

```
$ v="BYE"
# Just the first character
$ printf '%s\n' "${v,}"
bYE
# All characters
$ printf '%s\n' "${v,,}"
bye
# Alternative
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

Toggle Case

```
$ v="Hello World"
# All chars
$ echo "${v~~}"
hELLO wORLD
$ echo "${v~}"
# Just the first char
hello World
```

Section 15.2: Length of parameter

```
# Length of a string
$ var='12345'
$ echo "${#var}"
```

```
5
```

注意，这里指的是字符数的长度，这不一定等同于字节数（例如在UTF-8中，大多数字符由多个字节编码），也不一定等同于字形/字素的数量（其中一些是字符的组合），也不一定等同于显示宽度。

```
# 数组元素数量
$ myarr=(1 2 3)
$ echo "${#myarr[@]}"
3

# 也适用于位置参数
$ set -- 1 2 3 4
$ echo "${#@}"
4

# 但更常见且对其他shell更具移植性的方法是
$ echo "$#"
4
```

第15.3节：字符串中替换模式

第一次匹配：

```
$ a='I am a string'
$ echo "${a/a/A}"
我 Am a string
```

全部匹配：

```
$ echo "${a//a/A}"
我 Am A string
```

开头匹配：

```
$ echo "${a/#I/y}"
我 am a string
```

结尾匹配：

```
$ echo "${a/%g/N}"
I am a strinN
```

将模式替换为空：

```
$ echo "${a/g/}"
我是一个字符串
```

给数组项添加前缀：

```
$ A=(hello world)
$ echo "${A[@]/#/R}"
Rhello Rworld
```



```
5
```

Note that it's the length in number of *characters* which is not necessarily the same as the number of *bytes* (like in UTF-8 where most characters are encoded in more than one byte), nor the number of *glyphs/graphemes* (some of which are combinations of characters), nor is it necessarily the same as the display width.

```
# Number of array elements
$ myarr=(1 2 3)
$ echo "${#myarr[@]}"
3

# Works for positional parameters as well
$ set -- 1 2 3 4
$ echo "${#@}"
4

# But more commonly (and portably to other shells), one would use
$ echo "$#"
4
```

Section 15.3: Replace pattern in string

First match:

```
$ a='I am a string'
$ echo "${a/a/A}"
I Am a string
```

All matches:

```
$ echo "${a//a/A}"
I Am A string
```

Match at the beginning:

```
$ echo "${a/#I/y}"
y am a string
```

Match at the end:

```
$ echo "${a/%g/N}"
I am a strinN
```

Replace a pattern with nothing:

```
$ echo "${a/g/}"
I am a strin
```

Add prefix to array items:

```
$ A=(hello world)
$ echo "${A[@]/#/R}"
Rhello Rworld
```

第15.4节：子字符串和子数组

```
var='0123456789abcdef'

# 定义一个从零开始的偏移量
$ printf '%s' "${var:3}"3456789a
bcdef

# 子字符串的偏移量和长度
$ printf '%s' "${var:3:4}"
3456
```

版本 ≥ 4.2

```
# 负长度从字符串末尾开始计数
$ printf '%s' "${var:3:-5}"3456789a


# 负偏移从末尾开始计数
# 需要空格以避免与 ${var:-6} 混淆
$ printf '%s' "${var: -6}"abcdef


# 另一种写法：使用括号
$ printf '%s' "${var:(-6)}"abcdef


# 负偏移和负长度
$ printf '%s' "${var: -6:-5}"a
```

如果参数是位置参数或带下标数组的元素，同样适用这些扩展：

```
# 设置位置参数 $1
set -- 0123456789abcdef

# 定义偏移量
$ printf '%s' "${1:5}"56789ab
cdef

# 赋值给数组元素
myarr[0]='0123456789abcdef'

# 定义偏移量和长度
$ printf '%s' "${myarr[0]:7:3}"
789
```

类似的扩展适用于位置参数，其中偏移量是从1开始计数的：

```
# 设置位置参数 $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# 定义一个偏移量（注意这里也考虑了 $0（不是位置参数））

$ printf '%s' "${@:10}"
```


Section 15.4: Substrings and subarrays

```
var='0123456789abcdef'

# Define a zero-based offset
$ printf '%s\n' "${var:3}"
3456789abcdef

# Offset and length of substring
$ printf '%s\n' "${var:3:4}"
3456
```

Version ≥ 4.2

```
# Negative length counts from the end of the string
$ printf '%s\n' "${var:3:-5}"
3456789a

# Negative offset counts from the end
# Needs a space to avoid confusion with ${var:-6}
$ printf '%s\n' "${var: -6}"
abcdef

# Alternative: parentheses
$ printf '%s\n' "${var:(-6)}"
abcdef

# Negative offset and negative length
$ printf '%s\n' "${var: -6:-5}"
a
```

The same expansions apply if the parameter is a **positional parameter** or the **element of a subscripted array**:

```
# Set positional parameter $1
set -- 0123456789abcdef

# Define offset
$ printf '%s\n' "${1:5}"
56789abcdef

# Assign to array element
myarr[0]='0123456789abcdef'

# Define offset and length
$ printf '%s\n' "${myarr[0]:7:3}"
789
```

Analogous expansions apply to **positional parameters**, where offsets are one-based:

```
# Set positional parameters $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# Define an offset (beware $0 (not a positional parameter)
# is being considered here as well)
$ printf '%s\n' "${@:10}"
0
a
b
c
d
e
f
```

```
# 定义一个偏移量和长度
$ printf '%s' "${@:10:3}"
```



```
# Define an offset and a length
$ printf '%s\n' "${@:10:3}"
0
a
b

# No negative lengths allowed for positional parameters
$ printf '%s\n' "${@:10:-2}"
bash: -2: substring expression < 0

# Negative offset counts from the end
# Needs a space to avoid confusion with ${@:-10:2}
$ printf '%s\n' "${@: -10:2}"
7
8

# ${@:0} is $0 which is not otherwise a positional parameters or part
# of $@
$ printf '%s\n' "${@:0:2}"
/usr/bin/bash
1
```

Substring expansion can be used with **indexed arrays**:

```
# Create array (zero-based indices)
$ myarr=(0 1 2 3 4 5 6 7 8 9 a b c d e f)

# Elements with index 5 and higher
$ printf '%s\n' "${myarr[@]:12}"
c
d
e
f

# 3 elements, starting with index 5
$ printf '%s\n' "${myarr[@]:5:3}"
5
6
7

# The last element of the array
$ printf '%s\n' "${myarr[@]: -1}"
f
```

Section 15.5: Delete a pattern from the beginning of a string

Shortest match:

```
$ a='I am a string'
$ echo "${a#*a}"
m a string
```

Longest match:

```
$ echo "${a##*a}"
 string
```

第15.6节：参数间接引用

Bash的间接引用允许获取另一个变量中存储的变量名对应的值。变量示例：

```
$ red="the color red"
$ green="the color green"

$ color=red
$ echo "${!color}"
the color red
$ color=green
$ echo "${!color}"
the color green
```

一些展示间接展开用法的更多示例：

```
$ foo=10
$ x=foo
$ echo ${x}       #经典变量打印
 foo

$ foo=10
$ x=foo
$ echo ${!x}      #间接展开
 10
```

再举一个例子：

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${i}";done; }; argtester -ab -cd -ef
1   #i 展开为 1
2   #i 展开为 2
3   #i 展开为 3

$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab     # i=1 --> 展开为 $1 ---> 展开为传递给函数的第一个参数
-cd     # i=2 --> 展开为 $2 ---> 展开为传递给函数的第二个参数
-ef     # i=3 --> 展开为 $3 ---> 展开为传递给函数的第三个参数
```

第15.7节：参数扩展和文件名

您可以使用 Bash 参数扩展来模拟常见的文件名处理操作，如basename和dirname。

我们将使用以下路径作为示例：

```
FILENAME="/tmp/example/myfile.txt"
```

要模拟dirname并返回文件路径的目录名：

```
echo "${FILENAME%/*}"
#输出: /tmp/example
```

要模拟basename $FILENAME并返回文件路径的文件名：

```
echo "${FILENAME##*/}"
```


Section 15.6: Parameter indirection

Bash indirection permits to get the value of a variable whose name is contained in another variable. Variables example:

```
$ red="the color red"
$ green="the color green"

$ color=red
$ echo "${!color}"
the color red
$ color=green
$ echo "${!color}"
the color green
```

Some more examples that demonstrate the indirect expansion usage:

```
$ foo=10
$ x=foo
$ echo ${x}       #Classic variable print
foo

$ foo=10
$ x=foo
$ echo ${!x}      #Indirect expansion
10
```

One more example:

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${i}";done; }; argtester -ab -cd -ef
1   #i expanded to 1
2   #i expanded to 2
3   #i expanded to 3

$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab     # i=1 --> expanded to $1 ---> expanded to first argument sent to function
-cd     # i=2 --> expanded to $2 ---> expanded to second argument sent to function
-ef     # i=3 --> expanded to $3 ---> expanded to third argument sent to function
```

Section 15.7: Parameter expansion and filenames

You can use Bash Parameter Expansion to emulate common filename-processing operations like **basename** and **dirname**.

We will use this as our example path:

```
FILENAME="/tmp/example/myfile.txt"
```

To emulate **dirname** and return the directory name of a file path:

```
echo "${FILENAME%/*}"
#Out: /tmp/example
```

To emulate **basename** $FILENAME and return the filename of a file path:

```
echo "${FILENAME##*/}"
```

```
#输出：myfile.txt
```

模拟basename$FILENAME.txt并返回不带.txt.扩展名的文件名：

```
BASENAME="${FILENAME##*/}"
echo "${BASENAME%%.txt}"
#输出：myfile
```

第15.8节：默认值替换

> `${parameter:-word}`
>
> 如果parameter未设置或为空，则替换为word的展开值。否则，替换为parameter的值。

```
$ unset var
$ echo "${var:-XX}"      # 参数未设置 -> 发生XX的展开
XX
$ var=""                 # 参数为空 -> 发生XX的展开
$ echo "${var:-XX}"
XX
$ var=23                 # 参数不为空 -> 发生原始扩展
$ echo "${var:-XX}"
23
```

> `${parameter:=word}`
>
> 如果参数未设置或为空，则将 word 的扩展赋值给参数。然后替换为参数的值。位置参数和特殊参数不能以这种方式赋值。

```
$ unset var
$ echo "${var:=XX}"      # 参数未设置 -> 将 word 赋值为 XX
XX
$ echo "$var"
XX
$ var=""                 # 参数为空 -> 将 word 赋值为 XX
$ echo "${var:=XX}"
XX
$ echo "$var"
XX
$ var=23                 # 参数不为空 -> 不发生赋值
$ echo "${var:=XX}"
23
$ echo "$var"
23
```

第15.9节：从字符串末尾删除模式

最短匹配：

```
$ a='I am a string'
$ echo "${a%a*}"
我是
```

最长匹配：


```
#Out: myfile.txt
```

To emulate `basename $FILENAME .txt` and return the filename without the `.txt` extension:

```
BASENAME="${FILENAME##*/}"
echo "${BASENAME%%.txt}"
#Out: myfile
```

Section 15.8: Default value substitution

> `${parameter:-word}`
>
> If parameter is unset or null, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

```
$ unset var
$ echo "${var:-XX}"      # Parameter is unset -> expansion XX occurs
XX
$ var=""                 # Parameter is null -> expansion XX occurs
$ echo "${var:-XX}"
XX
$ var=23                 # Parameter is not null -> original expansion occurs
$ echo "${var:-XX}"
23
```

> `${parameter:=word}`
>
> If parameter is unset or null, the expansion of word is assigned to parameter. The value of parameter is then substituted. Positional parameters and special parameters may not be assigned to in this way.

```
$ unset var
$ echo "${var:=XX}"      # Parameter is unset -> word is assigned to XX
XX
$ echo "$var"
XX
$ var=""                 # Parameter is null -> word is assigned to XX
$ echo "${var:=XX}"
XX
$ echo "$var"
XX
$ var=23                 # Parameter is not null -> no assignment occurs
$ echo "${var:=XX}"
23
$ echo "$var"
23
```

Section 15.9: Delete a pattern from the end of a string

Shortest match:

```
$ a='I am a string'
$ echo "${a%a*}"
I am
```

Longest match:

```
$ echo "${a%%a*}"
我
```

第15.10节：扩展过程中的处理

变量不一定非得扩展为它们的值——在扩展过程中可以提取子字符串，这对于提取文件扩展名或路径部分非常有用。通配符字符保持其通常含义，因此.*表示字面上的点，后跟任意字符序列；它不是正则表达式。

```
$ v=foo-bar-baz
$ echo ${v%%-*}
foo
$ echo ${v%-*}
foo-bar
$ echo ${v##*-}
baz
$ echo ${v#*-}
bar-baz
```

也可以使用默认值来扩展变量——比如我想调用用户的编辑器，但如果他们没有设置，我想给他们vim。

```
$ EDITOR=nano
$ ${EDITOR:-vim} /tmp/some_file
# 打开 nano
$ unset EDITOR
$ $ ${EDITOR:-vim} /tmp/some_file
# 打开 vim
```

有两种不同的方式来执行此扩展，它们的区别在于相关变量是空还是未设置。使用 :- 会在变量未设置或为空时使用默认值，而 - 仅在变量未设置时使用默认值，但如果变量被设置为空字符串，则会使用该变量：

```
$ a="set"
$ b=""
$ unset c
$ echo ${a:-default_a} ${b:-default_b} ${c:-default_c}
set default_b default_c
$ echo ${a-default_a} ${b-default_b} ${c-default_c}
set default_c
```

类似于默认值，也可以给出替代值；当某个变量不可用时使用默认值，
当变量可用时使用替代值。

```
$ a="set"
$ b=""
$ echo ${a:+alternative_a} ${b:+alternative_b}
alternative_a
```

注意这些扩展可以嵌套，使用替代值在为命令行标志提供参数时尤其有用；

```
$ output_file=/tmp/foo
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# 展开为 wget -o /tmp/foo www.stackexchange.com
$ unset output_file
```



```
$ echo "${a%%a*}"
I
```

Section 15.10: Munging during expansion

Variables don't necessarily have to expand to their values - substrings can be extracted during expansion, which can be useful for extracting file extensions or parts of paths. Globbing characters keep their usual meanings, so `.*` refers to a literal dot, followed by any sequence of characters; it's not a regular expression.

```
$ v=foo-bar-baz
$ echo ${v%%-*}
foo
$ echo ${v%-*}
foo-bar
$ echo ${v##*-}
baz
$ echo ${v#*-}
bar-baz
```

It's also possible to expand a variable using a default value - say I want to invoke the user's editor, but if they've not set one I'd like to give them `vim`.

```
$ EDITOR=nano
$ ${EDITOR:-vim} /tmp/some_file
# opens nano
$ unset EDITOR
$ $ ${EDITOR:-vim} /tmp/some_file
# opens vim
```

There are two different ways of performing this expansion, which differ in whether the relevant variable is empty or unset. Using `:-` will use the default if the variable is either unset or empty, whilst `-` only uses the default if the variable is unset, but will use the variable if it is set to the empty string:

```
$ a="set"
$ b=""
$ unset c
$ echo ${a:-default_a} ${b:-default_b} ${c:-default_c}
set default_b default_c
$ echo ${a-default_a} ${b-default_b} ${c-default_c}
set default_c
```

Similar to defaults, alternatives can be given; where a default is used if a particular variable isn't available, an alternative is used if the variable is available.

```
$ a="set"
$ b=""
$ echo ${a:+alternative_a} ${b:+alternative_b}
alternative_a
```

Noting that these expansions can be nested, using alternatives becomes particularly useful when supplying arguments to command line flags;

```
$ output_file=/tmp/foo
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget -o /tmp/foo www.stackexchange.com
$ unset output_file
```

```
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# 展开为 wget  www.stackexchange.com
```

第15.11节：变量为空或未设置时出错

其语义类似于默认值替换，但不是替换为默认值，而是使用提供的错误信息报错。形式为${VARNAME?ERRMSG}和${VARNAME:?ERRMSG}。带:的形式在变量未设置或为空时都会报错，而不带:的形式仅在变量未设置时报错。如果发生错误，将输出ERRMSG并将退出码设置为1。

```
#!/bin/bash
FOO=
# ./script.sh: 第4行: FOO: 为空
echo "FOO 是 ${FOO:?EMPTY}"
# FOO 是
echo "FOO 是 ${FOO?UNSET}"
# ./script.sh: 第8行: BAR: 为空
echo "BAR 是 ${BAR:?EMPTY}"
# ./script.sh: 第10行: BAR: 未设置
echo "BAR 是 ${BAR?UNSET}"
```

要运行上面完整的示例，需要注释掉每个出错的 echo 语句才能继续。


```
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget  www.stackexchange.com
```

Section 15.11: Error if variable is empty or unset

The semantics for this are similar to that of default value substitution, but instead of substituting a default value, it errors out with the provided error message. The forms are `${VARNAME?ERRMSG}` and `${VARNAME:?ERRMSG}`. The form with `:` will error our if the variable is **unset** *or* **empty**, whereas the form without will only error out if the variable is *unset*. If an error is thrown, the `ERRMSG` is output and the exit code is set to `1`.

```
#!/bin/bash
FOO=
# ./script.sh: line 4: FOO: EMPTY
echo "FOO is ${FOO:?EMPTY}"
# FOO is
echo "FOO is ${FOO?UNSET}"
# ./script.sh: line 8: BAR: EMPTY
echo "BAR is ${BAR:?EMPTY}"
# ./script.sh: line 10: BAR: UNSET
echo "BAR is ${BAR?UNSET}"
```

The run the full example above each of the erroring echo statements needs to be commented out to proceed.

第16章：复制（cp）

选项	描述
-a,-archive	结合了 d、p 和 r 选项
-b, -备份	删除前，进行备份
-d, --no-deference	保留链接
-f, --force	删除现有目标文件而不提示用户
-i, --interactive	覆盖前显示提示
-l, --link	不复制，改为链接文件
-p, --preserve	尽可能保留文件属性
-R, --recursive	递归复制目录

第16.1节：复制单个文件

将foo.txt从/path/to/source/复制到/path/to/target/folder/

```
cp /path/to/source/foo.txt /path/to/target/folder/
```

将foo.txt从/path/to/source/复制到/path/to/target/folder/，命名为bar.txt

```
cp /path/to/source/foo.txt /path/to/target/folder/bar.txt
```

第16.2节：复制文件夹

将文件夹foo复制到文件夹bar

```
cp -r /path/to/foo /path/to/bar
```

如果在执行命令前文件夹bar已存在，则foo及其内容将被复制到文件夹bar中。
但是，如果在发出命令之前不存在bar文件夹，则会创建bar文件夹，并将foo的内容放入bar

Chapter 16: Copying (cp)

Option	Description
-a,-archive	Combines the d, p and r options
-b, -backup	Before removal, makes a backup
-d, --no-deference	Preserves links
-f, --force	Remove existing destinations without prompting user
-i, --interactive	Show prompt before overwriting
-l, --link	Instead of copying, link files instead
-p, --preserve	Preserve file attributes when possible
-R, --recursive	Recursively copy directories

Section 16.1: Copy a single file

Copy `foo.txt` from `/path/to/source/` to `/path/to/target/folder/`

```
cp /path/to/source/foo.txt /path/to/target/folder/
```

Copy `foo.txt` from `/path/to/source/` to `/path/to/target/folder/` into a file called `bar.txt`

```
cp /path/to/source/foo.txt /path/to/target/folder/bar.txt
```

Section 16.2: Copy folders

copy folder `foo` into folder `bar`

```
cp -r /path/to/foo /path/to/bar
```

if folder bar exists before issuing the command, then `foo` and its content will be copied into the folder `bar`.
However, if `bar` does not exist before issuing the command, then the folder `bar` will be created and the content of `foo` will be placed into `bar`

第17章：find命令

find是一个命令，用于递归搜索目录中符合条件的文件（或目录），然后对选中的文件执行某些操作。

find 搜索路径 选择条件 操作

第17.1节：按名称或扩展名搜索文件

要查找相对于pwd的特定名称的文件/目录：

```
$ find . -name "myFile.txt"
./myFile.txt
```

要查找具有特定扩展名的文件/目录，请使用通配符：

```
$ find . -name "*.txt"
./myFile.txt
./myFile2.txt
```

要查找匹配多个扩展名之一的文件/目录，请使用or标志：

```
$ find . -name "*.txt" -o -name "*.sh"
```

查找名称以 abc 开头，后跟一个字母字符，再跟一个数字结尾的文件/目录：

```
$ find . -name "abc[a-z][0-9]"
```

查找位于特定目录下的所有文件/目录

```
$ find /opt
```

仅搜索文件（不包括目录），使用 -type f：

```
find /opt -type f
```

仅搜索目录（不包括普通文件），使用 -type d：

```
find /opt -type d
```

第17.2节：对找到的文件执行命令

有时我们需要对大量文件运行命令。这可以使用 xargs 来完成。

```
find . -type d -print | xargs -r chmod 770
```

上述命令将递归查找相对于.（即当前工作目录）的所有目录（-type d），并对它们执行chmod 770。选项-r指定如果find未找到任何文件，xargs将不运行chmod。

如果你的文件名或目录名中包含空格字符，该命令可能会出错；解决方案是使用以下命令

Chapter 17: Find

find is a command to recursively search a directory for files(or directories) that match a criteria, and then perform some action on the selected files.

find search_path selection_criteria action

Section 17.1: Searching for a file by name or extension

To find files/directories with a specific name, relative to **pwd**:

```
$ find . -name "myFile.txt"
./myFile.txt
```

To find files/directories with a specific extension, use a wildcard:

```
$ find . -name "*.txt"
./myFile.txt
./myFile2.txt
```

To find files/directories matching one of many extensions, use the `or` flag:

```
$ find . -name "*.txt" -o -name "*.sh"
```

To find files/directories which name begin with abc and end with one alpha character following a one digit:

```
$ find . -name "abc[a-z][0-9]"
```

To find all files/directories located in a specific directory

```
$ find /opt
```

To search for files only (not directories), use `-type f`:

```
find /opt -type f
```

To search for directories only (not regular files), use `-type d`:

```
find /opt -type d
```

Section 17.2: Executing commands against a found file

Sometimes we will need to run commands against a lot of files. This can be done using **xargs**.

```
find . -type d -print | xargs -r chmod 770
```

The above command will recursively find all directories (`-type d`) relative to `.` (which is your current working directory), and execute `chmod 770` on them. The `-r` option specifies to **xargs** to not run **chmod** if **find** did not find any files.

If your files names or directories have a space character in them, this command may choke; a solution is to use the following

```
find . -type d -print0 | xargs -r -0 chmod 770
```

在上述示例中，-print0和-0标志指定文件名将使用null字节分隔，允许文件名中使用空格等特殊字符。这是GNU的扩展，可能在其他版本的find和 xargs中无法使用。

推荐的做法是跳过 xargs命令，让find自己调用子进程：

```
find . -type d -exec chmod 770 {} \;
```

这里，{}是一个占位符，表示你想在该位置使用文件名。 find将对每个文件单独执行chmod。

你也可以通过以下方式将所有文件名传递给chmod的单次调用：

```
find . -type d -exec chmod 770 {} +
```

这也是上述 xargs 代码片段的行为。（如果想对每个文件单独调用，可以使用 xargs -n1）。

第三种选择是让bash循环遍历 find 输出的文件名列表：

```
find . -type d | while read -r d; do chmod 770 "$d"; done
```

这在语法上是最笨拙的，但当你想对每个找到的文件运行多个命令时很方便。
然而，面对带有奇怪名称的文件名时，这是不安全的。

```
find . -type f | while read -r d; do mv "$d" "${d// /_}"; done
```

这将把文件名中的所有空格替换为下划线。（如果路径中包含
前置*目录*名中的空格，这个例子也无法正常工作。）

上述问题在于 **while read** -r 期望每行一个条目，但文件名可能包含换行符
（而且， **read** -r 会丢失任何尾随空白）。你可以通过反过来处理来解决这个问题：

```
find . -type d -exec bash -c 'for f; do mv "$f" "${f// /_}"; done' _ {} +
```

这样， -exec 以完全正确且可移植的形式接收文件名； bash -c 以多个参数的形式接收它们，这些参数会在 $@ 中找到，且正确加引号等。（脚本当然需要正确处理这些名称；每个包含文件名的变量都需要用双引号括起来。）神秘的 _ 是必要的，因为 bash -c 'script'的第一个参数用于填充 $0。

第17.3节：通过访问/修改时间查找文件

在一个 ext 文件系统中，每个文件都有一个存储的访问时间、修改时间和（状态）更改时间——要查看这些信息，你可以使用 stat myFile.txt；使用 find 中的标志，我们可以搜索在某个时间范围内被修改的文件。

查找在过去 2 小时内被修改的文件：

```
$ find . -mmin -120
```



```
find . -type d -print0 | xargs -r -0 chmod 770
```

In the above example, the `-print0` and `-0` flags specify that the file names will be separated using a `null` byte, and allows the use of special characters, like spaces, in the file names. This is a GNU extension, and may not work in other versions of **find** and **xargs**.

The preferred way to do this is to skip the **xargs** command and let **find** call the subprocess itself:

```
find . -type d -exec chmod 770 {} \;
```

Here, the {} is a placeholder indicating that you want to use the file name at that point. **find** will execute **chmod** on each file individually.

You can alternatively pass all file names to a *single* call of **chmod**, by using

```
find . -type d -exec chmod 770 {} +
```

This is also the behaviour of the above **xargs** snippets. (To call on each file individually, you can use **xargs -n1**).

A third option is to let bash loop over the list of filenames **find** outputs:

```
find . -type d | while read -r d; do chmod 770 "$d"; done
```

This is syntactically the most clunky, but convenient when you want to run multiple commands on each found file. However, this is **unsafe** in the face of file names with odd names.

```
find . -type f | while read -r d; do mv "$d" "${d// /_}"; done
```

which will replace all spaces in file names with underscores.(This example also won't work if there are spaces in leading *directory* names.)

The problem with the above is that **while read -r** expects one entry per line, but file names can contain newlines (and also, **read -r** will lose any trailing whitespace). You can fix this by turning things around:

```
find . -type d -exec bash -c 'for f; do mv "$f" "${f// /_}"; done' _ {} +
```

This way, the **-exec** receives the file names in a form which is completely correct and portable; the **bash -c** receives them as a number of arguments, which will be found in $@, correctly quoted etc. (The script will need to handle these names correctly, of course; every variable which contains a file name needs to be in double quotes.)

The mysterious _ is necessary because the first argument to **bash -c 'script'** is used to populate $0.

Section 17.3: Finding file by access / modification time

On an ext filesystem, each file has a stored Access, Modification, and (Status) Change time associated with it - to view this information you can use **stat myFile.txt**; using flags within *find*, we can search for files that were modified within a certain time range.

To find files that *have* been modified within the last 2 hours:

```
$ find . -mmin -120
```

查找在过去 2 小时内未被修改的文件：

```
$ find . -mmin +120
```

上述示例仅搜索 *修改* 时间——要搜索 **访问** 时间或 **更改** 时间，分别使用 a 或 c。

```
$ find . -amin -120
$ find . -cmin +120
```

通用格式：

-mmin n ：文件在 n 分钟前被修改
-mmin -n ：文件在不到 n 分钟前被修改
-mmin +n ：文件在 n 分钟前被修改

查找在过去2天内被修改的文件：

```
find . -mtime -2
```

查找在过去2天内未被修改的文件

```
find . -mtime +2
```

分别使用 -atime 和 -ctime 表示访问时间和状态更改时间。

通用格式：

-mtime n ：文件在 nx24 小时前被修改
-mtime -n ：文件在不到 nx24 小时前被修改
-mtime +n ：文件在超过 *nx24* 小时前被修改

查找在一段日期范围内***修改的文件***，从2007-06-07到2007-06-08：

```
find . -type f -newermt 2007-06-07 ! -newermt 2007-06-08
```

查找在一段时间戳范围内访问的文件（使用文件作为时间戳），从1小时前到10分钟前：

```
touch -t $(date -d '1 HOUR AGO' +%Y%m%d%H%M.%S) start_date
touch -t $(date -d '10 MINUTE AGO' +%Y%m%d%H%M.%S) end_date
timeout 10 find "$LOCAL_FOLDER" -newerat "start_date" ! -newerat "end_date" -print
```

通用格式：

-newerXY 参考：比较当前文件的时间戳与参考时间戳。 XY 可以是以下值之一： at（访问时间）， mt（修改时间）， ct（变更时间）等。 reference 是要比较的文件名，指定要比较的时间戳类型（访问、修改、变更）或描述绝对时间的 字符串。

第17.4节：根据大小查找文件

查找大于15MB的文件：

To find files that *have not* been modified within the last 2 hours:

```
$ find . -mmin +120
```

The above example are searching only on the *modified* time - to search on **a**ccess times, or **c**hanged times, use a, or c accordingly.

```
$ find . -amin -120
$ find . -cmin +120
```

General format:

`-mmin` n : File was modified *n* minutes ago
`-mmin -n` : File was modified less than *n* minutes ago
`-mmin` +n : File was modified more than *n* minutes ago

Find files that *have* been modified within the last 2 days:

```
find . -mtime -2
```

Find files that *have not* been modified within the last 2 days

```
find . -mtime +2
```

Use `-atime` and `-ctime` for access time and status change time respectively.

General format:

`-mtime` n : File was modified *nx24* hours ago
`-mtime -n` : File was modified less than *nx24* hours ago
`-mtime` +n : File was modified more than *nx24* hours ago

Find files modified in a ***range of dates***, from 2007-06-07 to 2007-06-08:

```
find . -type f -newermt 2007-06-07 ! -newermt 2007-06-08
```

Find files accessed in a ***range of timestamps*** (using files as timestamp), from 1 hour ago to 10 minutes ago:

```
touch -t $(date -d '1 HOUR AGO' +%Y%m%d%H%M.%S) start_date
touch -t $(date -d '10 MINUTE AGO' +%Y%m%d%H%M.%S) end_date
timeout 10 find "$LOCAL_FOLDER" -newerat "start_date" ! -newerat "end_date" -print
```

General format:

`-newerXY reference` : Compares the timestamp of the current file with reference. `XY` could have one of the following values: `at` (access time), `mt` (modification time), `ct` (change time) and more. `reference` is the *name of a file* whe want to compare the timestamp specified (access, modification, change) or a *string* describing an absolute time.

Section 17.4: Finding files according to size

Find files larger than 15MB:

```
find -type f -size +15M
```

查找小于12KB的文件：

```
find -type f -size -12k
```

查找大小正好为12KB的文件：

```
find -type f -size 12k
```

或者

```
find -type f -size 12288c
```

或者

```
find -type f -size 24b
```

或者

```
find -type f -size 24
```

通用格式：

```
find [options] -size n[cwbkMG]
```

> 查找大小为 n 块的文件，其中 +n 表示大于 n 块，-n 表示小于 n 块，n（无任何符号）表示正好为 n 块

块大小：

c1.: 字节
w2.: 2 字节
b3.: 512 字节（默认）
k4. : 1 KB
M5. : 1 MB
G6. : 1 GB

第 17.5 节：过滤路径

-path 参数允许指定一个模式来匹配结果的路径。该模式也可以匹配文件名本身。

要查找路径（文件夹或名称）中任何位置包含log的文件：

```
find . -type f -path '*log*'
```

仅查找名为log的文件夹内的文件（任意层级）：

```
find . -type f -path '*/log/*'
```

仅查找名为log或data的文件夹内的文件：


```
find -type f -size +15M
```

Find files less than 12KB:

```
find -type f -size -12k
```

Find files exactly of 12KB size:

```
find -type f -size 12k
```

Or

```
find -type f -size 12288c
```

Or

```
find -type f -size 24b
```

Or

```
find -type f -size 24
```

General format:

```
find [options] -size n[cwbkMG]
```

> Find files of n-block size, where +n means more than n-block, -n means less than n-block and n (without any sign) means exactly n-block

Block size:

1. c: bytes
2. w: 2 bytes
3. b: 512 bytes (default)
4. k: 1 KB
5. M: 1 MB
6. G: 1 GB

Section 17.5: Filter the path

The -path parameter allows to specify a pattern to match the path of the result. The pattern can match also the name itself.

To find only files containing log anywhere in their path (folder or name):

```
find . -type f -path '*log*'
```

To find only files within a folder called log (on any level):

```
find . -type f -path '*/log/*'
```

To find only files within a folder called log or data:

```
find . -type f -path '*/log/*' -o -path '*/data/*'
```

查找所有文件，except包含在名为bin的文件夹中的文件：

```
find . -type f -not -path '*/bin/*'
```

查找所有文件，except包含在名为bin的文件夹中或日志文件：

```
find . -type f -not -path '*log' -not -path '*/bin/*'
```

第17.6节：按类型查找文件

要查找文件，请使用 -type f 标志

```
$ find . -type f
```

要查找目录，请使用 -type d 标志

```
$ find . -type d
```

要查找块设备，请使用 -type b 标志

```
$ find /dev -type b
```

要查找符号链接，请使用 -type l 标志

```
$ find . -type l
```

第17.7节：按特定扩展名查找文件

要查找当前路径下所有具有某个扩展名的文件，可以使用以下 `find` 语法。它通过利用 bash 的内置 glob 结构来匹配所有具有该 .extension 的名称。

```
find /目录/搜索 -maxdepth 1 -type f -name "*.extension"
```

要仅从当前目录查找所有类型为.txt的文件，执行

```
find . -maxdepth 1 -type f -name "*.txt"
```



```
find . -type f -path '*/log/*' -o -path '*/data/*'
```

To find all files **except** the ones contained in a folder called `bin`:

```
find . -type f -not -path '*/bin/*'
```

To find all file all files **except** the ones contained in a folder called `bin` or log files:

```
find . -type f -not -path '*log' -not -path '*/bin/*'
```

Section 17.6: Finding files by type

To find files, use the `-type f` flag

```
$ find . -type f
```

To find directories, use the `-type d` flag

```
$ find . -type d
```

To find block devices, use the `-type b` flag

```
$ find /dev -type b
```

To find symlinks, use the `-type l` flag

```
$ find . -type l
```

Section 17.7: Finding files by specific extension

To find all the files of a certain extension within the current path you can use the following `find` syntax. It works by making use of `bash`'s built-in `glob` construct to match all the names having the `.extension`.

```
find /directory/to/search -maxdepth 1 -type f -name "*.extension"
```

To find all files of type `.txt` from the current directory alone, do

```
find . -maxdepth 1 -type f -name "*.txt"
```

第18章：使用sort命令

选项	含义
-u	使输出的每一行唯一

sort是一个Unix命令，用于按顺序排列文件中的数据。

第18.1节：sort命令输出

sort命令用于对行列表进行排序。

从文件输入

```
sort  file.txt
```

来自命令的输入

您可以对任何命令输出进行排序。示例中是对符合某个模式的文件列表进行排序。

```
find * -name  pattern  |  sort
```

第18.2节：使输出唯一

如果输出的每一行都需要唯一，添加-u选项。

显示文件夹中文件所有者

```
ls -l | awk '{print $3}' | sort -u
```

第18.3节：数字排序

假设我们有这个文件：

```
test>>cat file
10.格兰芬多
4.霍格沃茨
2.哈利
3.邓布利多
1.分院帽
```

要按数字顺序排序此文件，请使用带 -n 选项的 sort 命令：

```
test>>sort -n file
```

这将按如下顺序排序文件：

```
1.分院帽
2.哈利
3.邓布利多
4.霍格沃茨
10.格兰芬多
```

反转排序顺序：要反转排序顺序，请使用 -r 选项

Chapter 18: Using sort

Option	Meaning
-u	Make each lines of output unique

sort is a Unix command to order data in file(s) in a sequence.

Section 18.1: Sort command output

`sort` command is used to sort a list of lines.

Input from a file

```
sort file.txt
```

Input from a command

You can sort any output command. In the example a list of file following a pattern.

```
find * -name pattern | sort
```

Section 18.2: Make output unique

If each lines of the output need to be unique, add `-u` option.

To display owner of files in folder

```
ls -l | awk '{print $3}' | sort -u
```

Section 18.3: Numeric sort

Suppose we have this file:

```
test>>cat file
10.Gryffindor
4.Hogwarts
2.Harry
3.Dumbledore
1.The sorting hat
```

To sort this file numerically, use sort with -n option:

```
test>>sort -n file
```

This should sort the file as below:

```
1.The sorting hat
2.Harry
3.Dumbledore
4.Hogwarts
10.Gryffindor
```

Reversing sort order: To reverse the order of the sort use the -r option

要反转上述文件的排序顺序，请使用：

```
sort -rn file
```

这将按如下顺序排序文件：

```
10.格兰芬多
4.霍格沃茨
3.邓布利多
2.哈利
1.分院帽
```

第18.4节：按键排序

假设我们有这个文件：

```
test>>cat 霍格沃茨
哈利          马尔福          罗伊娜          赫尔加
  格兰芬多    斯莱特林        拉文克劳        赫奇帕奇
  赫敏          戈伊尔          洛哈特          唐克斯
罗恩            斯内普          奥利凡德        纽特
  罗恩          戈伊尔          弗立维克        斯普劳特
```

要使用某列作为键对该文件进行排序，请使用 k 选项：

```
test>>sort -k 2 霍格沃茨
```

这将以第2列作为键对文件进行排序：

```
罗恩          戈伊尔          弗利维克        斯普劳特
  赫敏          戈伊尔          洛哈特          唐克斯
  哈利          马尔福          罗威娜          赫尔加
  格兰芬多    斯莱特林        拉文克劳        赫奇帕奇
  罗恩          斯内普          奥利凡德        纽特
```

现在如果我们需要用主键和次键一起对文件进行排序，使用：

```
sort -k 2,2 -k 1,1 Hogwarts
```

这将首先以第2列作为主键对文件进行排序，然后以第1列作为次键对文件进行排序：

```
赫敏          戈伊尔          洛哈特          唐克斯
  罗恩          戈伊尔          弗利维克        斯普劳特
  哈利          马尔福          罗威娜          赫尔加
  格兰芬多    斯莱特林        拉文克劳        赫奇帕奇
  罗恩          斯内普          奥利凡德        纽特
```

如果我们需要用多个键对文件进行排序，那么每个 -k 选项都需要指定排序的结束位置。所以 -k1,1 表示从第一列开始排序，到第一列结束排序。

-t 选项

在前面的例子中，文件使用了默认的分隔符——制表符。如果对使用非默认分隔符的文件进行排序，则需要使用 -t 选项来指定分隔符。假设我们有如下文件：

```
test>>cat file
```


To reverse the sort order of the above file use:

```
sort -rn file
```

This should sort the file as below:

```
10.Gryffindor
4.Hogwarts
3.Dumbledore
2.Harry
1.The sorting hat
```

Section 18.4: Sort by keys

Suppose we have this file:

```
test>>cat Hogwarts
  Harry         Malfoy        Rowena        Helga
  Gryffindor    Slytherin     Ravenclaw     Hufflepuff
  Hermione      Goyle         Lockhart      Tonks
  Ron           Snape         Olivander     Newt
  Ron           Goyle         Flitwick      Sprout
```

To sort this file using a column as key use the k option:

```
test>>sort -k 2 Hogwarts
```

This will sort the file with column 2 as the key:

```
  Ron           Goyle         Flitwick      Sprout
  Hermione      Goyle         Lockhart      Tonks
  Harry         Malfoy        Rowena        Helga
  Gryffindor    Slytherin     Ravenclaw     Hufflepuff
  Ron           Snape         Olivander     Newt
```

Now if we have to sort the file with a secondary key along with the primary key use:

```
sort -k 2,2 -k 1,1 Hogwarts
```

This will first sort the file with column 2 as primary key, and then sort the file with column 1 as secondary key:

```
  Hermione      Goyle         Lockhart      Tonks
  Ron           Goyle         Flitwick      Sprout
  Harry         Malfoy        Rowena        Helga
  Gryffindor    Slytherin     Ravenclaw     Hufflepuff
  Ron           Snape         Olivander     Newt
```

If we need to sort a file with more than 1 key , then for every -k option we need to specify where the sort ends. So -k1,1 means start the sort at the first column and end sort at first column.

-t option

In the previous example the file had the default delimeter - tab. In case of sorting a file that has non-default delimeter we need the -t option to specify the delimeter. Suppose we have the file as below:

```
test>>cat file
```

```
5.|格兰芬多
4.|霍格沃茨
2.|哈利
3.|邓布利多
1.|分院帽
```

要根据第二列对该文件进行排序，请使用：

```
test>>sort -t "|" -k 2 file
```

这将按以下方式对文件进行排序：

```
3.|邓布利多
5.|格兰芬多
2.|哈利
4.|霍格沃茨
1.|分院帽
```



```
5.|Gryffindor
4.|Hogwarts
2.|Harry
3.|Dumbledore
1.|The sorting hat
```

To sort this file as per the second column, use:

```
test>>sort -t "|" -k 2 file
```

This will sort the file as below:

```
3.|Dumbledore
5.|Gryffindor
2.|Harry
4.|Hogwarts
1.|The sorting hat
```

第19章：资源获取

第19.1节：资源获取文件

资源获取文件不同于执行，所有命令都在当前
bash会话的上下文中进行评估——这意味着任何定义的变量、函数或别名将在整个会话中持续存在。

创建您希望引用的文件sourceme.sh

```
#!/bin/bash

export A="hello_world"
alias sayHi="echo Hi"
sayHello() {
    echo 你好
}
```

从你的会话中，加载该文件

```
$ source sourceme.sh
```

从此以后，你可以使用该被加载文件中的所有资源

```
$ echo $A
hello_world

$ sayHi
Hi

$ sayHello
你好
```

注意命令 . 与 source 同义，因此你可以直接使用

```
$ . sourceme.sh
```

第19.2节：加载虚拟环境

在一台机器上开发多个应用时，将依赖项分离到虚拟环境中会很有用。

使用virtualenv时，这些环境会被加载到你的shell中，这样当你运行命令时，命令就来自该虚拟环境。

这通常通过pip安装。

```
pip install https://github.com/pypa/virtualenv/tarball/15.0.2
```

创建一个新环境

```
virtualenv --python=python3.5 my_env
```

激活环境

Chapter 19: Sourcing

Section 19.1: Sourcing a file

Sourcing a file is different from execution, in that all commands are evaluated within the context of the current bash session - this means that any variables, function, or aliases defined will persist throughout your session.

Create the file you wish to source `sourceme.sh`

```
#!/bin/bash

export A="hello_world"
alias sayHi="echo Hi"
sayHello() {
    echo Hello
}
```

From your session, source the file

```
$ source sourceme.sh
```

From hencefourth, you have all the resources of the sourced file available

```
$ echo $A
hello_world

$ sayHi
Hi

$ sayHello
Hello
```

Note that the command `.` is synonymous to **`source`**, such that you can simply use

```
$ . sourceme.sh
```

Section 19.2: Sourcing a virtual environment

When developing several applications on one machine, it becomes useful to separate out dependencies into virtual environments.

With the use of `virtualenv`, these environments are sourced into your shell so that when you run a command, it comes from that virtual environment.

This is most commonly installed using `pip`.

```
pip install https://github.com/pypa/virtualenv/tarball/15.0.2
```

Create a new environment

```
virtualenv --python=python3.5 my_env
```

Activate the environment

```
source my_env/bin/activate
```



```
source my_env/bin/activate
```

第20章：Here文档和here字符串

第20.1节：使用here文档执行命令

```
ssh -p 21 example@example.com <<EOF
  echo '打印当前目录'
echo "\$(pwd)"
  ls -a
find '*.txt'
EOF
```

$ 被转义是因为我们不希望它被当前shell展开，即$(pwd) 是在远程shell上执行的。

另一种方式：

```
ssh -p 21 example@example.com <<'EOF'
  echo '打印当前目录'
echo "$(pwd)"
  ls -a
find '*.txt'
EOF
```

注意：结束标记EOF 应位于行首（前面无空格）。如果需要缩进，可以使用制表符（tab），前提是你的heredoc以<<-开始。更多信息请参见“缩进here文档”和“限制字符串”示例。

第20.2节：缩进here文档

你可以用制表符缩进这里文档中的文本，你需要使用<<-重定向操作符，而不是<<：

```
$ cat <<- EOF
这是一些用制表符`缩进的内容。
你不能用空格缩进，必须使用制表符。
Bash 会删除这些行前面的空白。
__注意__：复制此示例时，务必将空格替换为制表符。
EOF

这是一些用制表符__缩进的内容。
你不能用空格缩进，必须使用制表符。
Bash 会删除这些行前的空格。
__注意__：复制此示例时，请务必将空格替换为制表符。
```

此用法的一个实际案例（如man bash中提到）是在shell脚本中，例如：

```
if cond; then
    cat <<- EOF
    hello
    there
    EOF
fi
```

通常会像此if语句中那样缩进代码块内的行，以提高可读性。如果没有<<-

Chapter 20: Here documents and here strings

Section 20.1: Execute command with here document

```
ssh -p 21 example@example.com <<EOF
  echo 'printing pwd'
  echo "\$(pwd)"
  ls -a
  find '*.txt'
EOF
```

$ is escaped because we do not want it to be expanded by the current shell i.e `$(pwd)` is to be executed on the remote shell.

Another way:

```
ssh -p 21 example@example.com <<'EOF'
  echo 'printing pwd'
  echo "$(pwd)"
  ls -a
  find '*.txt'
EOF
```

Note: The closing EOF **should** be at the beginning of the line (No whitespaces before). If indentation is required, tabs may be used if you start your heredoc with `<<-`. See the Indenting here documents and Limit Strings examples for more information.

Section 20.2: Indenting here documents

You can indent the text inside here documents with tabs, you need to use the `<<-` redirection operator instead of `<<`:

```
$ cat <<- EOF
    This is some content indented with tabs `\t`.
    You cannot indent with spaces you __have__ to use tabs.
    Bash will remove empty space before these lines.
    __Note__: Be sure to replace spaces with tabs when copying this example.
EOF

This is some content indented with tabs _\t_.
You cannot indent with spaces you __have__ to use tabs.
Bash will remove empty space before these lines.
__Note__: Be sure to replace spaces with tabs when copying this example.
```

One practical use case of this (as mentioned in `man bash`) is in shell scripts, for example:

```
if cond; then
    cat <<- EOF
    hello
    there
    EOF
fi
```

It is customary to indent the lines within code blocks as in this `if` statement, for better readability. Without the `<<-`

操作符语法，我们将不得不这样编写上述代码：

```
if cond; then
    cat << EOF
hello
there
EOF
fi
```

这样读起来非常不舒服，在更复杂的实际脚本中情况会更糟。

第20.3节：创建文件

here文档的一个经典用法是通过输入内容来创建文件：

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

here文档是位于 << EOF 和 EOF 之间的行。

这个here文档成为 cat 命令的输入。 cat 命令只是输出它的输入，使用输出重定向符号 > 我们将其重定向到文件 fruits.txt。

结果，fruits.txt 文件将包含以下内容：

```
apple
orange
lemon
```

输出重定向的常规规则适用：如果 fruits.txt 之前不存在，则会被创建。如果之前存在，则会被截断。

第20.4节：Here字符串

版本 ≥ 2.05b

你可以像这样使用here字符串向命令提供输入：

```
$ awk '{print $2}' <<< "hello world - how are you?"
world

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

你也可以用 here 字符串来给 while 循环输入数据：

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```


operator syntax, we would be forced to write the above code like this:

```
if cond; then
    cat << EOF
hello
there
EOF
fi
```

That's very unpleasant to read, and it gets much worse in a more complex realistic script.

Section 20.3: Create a file

A classic use of here documents is to create a file by typing its content:

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

The here-document is the lines between the `<< EOF` and `EOF`.

This here document becomes the input of the **`cat`** command. The **`cat`** command simply outputs its input, and using the output redirection operator **`>`** we redirect to a file `fruits.txt`.

As a result, the `fruits.txt` file will contain the lines:

```
apple
orange
lemon
```

The usual rules of output redirection apply: if `fruits.txt` did not exist before, it will be created. If it existed before, it will be truncated.

Section 20.4: Here strings

Version ≥ 2.05b

You can feed a command using here strings like this:

```
$ awk '{print $2}' <<< "hello world - how are you?"
world

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

You can also feed a **`while`** loop with a here string:

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```

第20.5节：使用 sudo 运行多个命令

```
sudo -s <<EOF
  a='var'
echo '使用 sudo 运行多个命令'
  mktemp -d
echo "\$a"
EOF
```

- $a 需要转义以防止被当前 shell 展开

或者

```
sudo -s <<'EOF'
  a='var'
echo '使用 sudo 运行多个命令'
  mktemp -d
echo "$a"
EOF
```

第20.6节：限制字符串

Heredoc 使用limitstring来确定何时停止读取输入。终止的 limitstring 必须

- 位于行首。
- 是该行唯一的文本 注意： 如果使用 <<-，limitstring 可以以制表符开头

正确示例：

```
cat <<limitstring
line 1
第2行
限制字符串
```

这将输出：

```
第1行
第2行
```

错误用法：

```
cat <<limitstring
第1行
第2行
 limitstring
```

由于最后一行的limitstring并不完全位于行首，shell会继续等待更多输入，直到看到一行以limitstring开头且不包含其他内容。只有那时，它才会停止等待输入，并继续将here-document传递给cat命令。

注意，当你在初始limitstring前加上连字符时，解析前会移除行首的所有制表符，因此数据和限制字符串可以用制表符缩进（便于在shell脚本中阅读）。

```
cat <<-limitstring
第1行     在单词line和has前各有一个制表符
            第2行有两个前导制表符
```


Section 20.5: Run several commands with sudo

```
sudo -s <<EOF
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "\$a"
EOF
```

- $a needs to be escaped to prevent it to be expanded by the current shell

Or

```
sudo -s <<'EOF'
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "$a"
EOF
```

Section 20.6: Limit Strings

A heredoc uses the *limitstring* to determine when to stop consuming input. The terminating limitstring **must**

- Be at the start of a line.
- Be the only text on the line **Note:** If you use `<<-` the limitstring can be prefixed with tabs `\t`

Correct:

```
cat <<limitstring
line 1
line 2
limitstring
```

This will output:

```
line 1
line 2
```

Incorrect use:

```
cat <<limitstring
line 1
line 2
 limitstring
```

Since `limitstring` on the last line is not exactly at the start of the line, the shell will continue to wait for further input, until it sees a line that starts with `limitstring` and doesn't contain anything else. Only then it will stop waiting for input, and proceed to pass the here-document to the `cat` command.

Note that when you prefix the initial limitstring with a hyphen, any tabs at the start of the line are removed before parsing, so the data and the limit string can be indented with tabs (for ease of reading in shell scripts).

```
cat <<-limitstring
        line 1    has a tab each before the words line and has
            line 2 has two leading tabs
```

```
        limitstring
```

将产生

```
第1行 在单词line和has前各有一个制表符
第2行有两个前导制表符
```

去除前导制表符（但不去除内部制表符）。


```
        limitstring
```

will produce

```
line 1    has a tab each before the words line and has
line 2 has two leading tabs
```

with the leading tabs (but not the internal tabs) removed.

第21章：引用

第21.1节：用于变量和命令替换的双引号

变量替换应仅在双引号内使用。

```
calculation='2 * 3'
echo "$calculation"         # 输出 2 * 3
echo $calculation           # 输出 2、当前目录下的文件列表和 3
echo "$(($calculation))"    # 输出 6
```

在双引号外，$var 取 var 的值，将其拆分为以空白字符分隔的部分，并将每个部分解释为通配符模式。除非你想要这种行为，否则请始终将 $var 放在双引号内："$var"。

命令替换同样适用："$(mycommand)" 是 mycommand 的输出，$(mycommand) 是对输出进行拆分+通配符扩展后的结果。

```
echo "$var"            # 好的
echo "$(mycommand)"    # 好的
another=$var           # 也可以，赋值时隐式双引号
make -D THING=$var     # 错误！这不是bash赋值。
make -D THING="$var"   # 好的
make -D "THING=$var"   # 也好
```

命令替换有自己的引号上下文。编写任意嵌套的替换很容易，因为解析器会跟踪嵌套深度，而不是贪婪地寻找第一个 " 字符。然而，StackOverflow的语法高亮解析是错误的。例如：

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # “formatted text: a + b = 0000”
```

命令替换中的变量参数在扩展时也应使用双引号：

```
echo "$(mycommand "$arg1" "$arg2")"
```

第21.2节：双引号和单引号的区别

双引号	单引号
允许变量扩展	防止变量扩展
如果启用，允许历史扩展	防止历史扩展
允许命令替换	防止命令替换
* 和 @ 可能具有特殊含义	* 和 @ 总是字面量
可以包含单引号或双引号	单引号内不允许出现单引号
$, `, ", \ 可以用 \ 转义以防止它们的特殊含义	它们全部都是字面量

两者共有的属性：

- 防止通配符扩展
- 防止单词拆分

示例：

Chapter 21: Quoting

Section 21.1: Double quotes for variable and command substitution

Variable substitutions should only be used inside double quotes.

```
calculation='2 * 3'
echo "$calculation"         # prints 2 * 3
echo $calculation           # prints 2, the list of files in the current directory, and 3
echo "$(($calculation))"    # prints 6
```

Outside of double quotes, `$var` takes the value of `var`, splits it into whitespace-delimited parts, and interprets each part as a glob (wildcard) pattern. Unless you want this behavior, always put `$var` inside double quotes: `"$var"`.

The same applies to command substitutions: `"$(mycommand)"` is the output of `mycommand`, `$(mycommand)` is the result of split+glob on the output.

```
echo "$var"            # good
echo "$(mycommand)"    # good
another=$var           # also works, assignment is implicitly double-quoted
make -D THING=$var     # BAD! This is not a bash assignment.
make -D THING="$var"   # good
make -D "THING=$var"   # also good
```

Command substitutions get their own quoting contexts. Writing arbitrarily nested substitutions is easy because the parser will keep track of nesting depth instead of greedily searching for the first `"` character. The StackOverflow syntax highlighter parses this wrong, however. For example:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # “formatted text: a + b = 0000”
```

Variable arguments to a command substitution should be double-quoted inside the expansions as well:

```
echo "$(mycommand "$arg1" "$arg2")"
```

Section 21.2: Difference between double quote and single quote

Double quote	Single quote
Allows variable expansion	Prevents variable expansion
Allows history expansion if enabled	Prevents history expansion
Allows command substitution	Prevents command substitution
`*` and `@` can have special meaning	`*` and `@` are always literals
Can contain both single quote or double quote	Single quote is not allowed inside single quote
$, `, ", \ can be escaped with \ to prevent their special meaning	All of them are literals

Properties that are common to both:

- Prevents globbing
- Prevents word splitting

Examples:

```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\""
"'"
$ a='var'
$ echo '$a'
$a
$ echo "$a"
var
```

第21.3节：换行符和控制字符

换行符可以包含在单引号字符串或双引号字符串中。注意，反斜杠加换行符不会产生换行，换行被忽略。

```
newline1='
'
newline2="
"
newline3=$t"emp
ty=\

echo "Line${newline1}break"
echo "Line${newline2}break"
echo "Line${newline3}break"
echo "这里没有换行${empty}"
```

在美元引号字符串中，反斜杠加字母或反斜杠加八进制数可以用来插入控制字符，就像许多其他编程语言中一样。

```
echo $'制表符: []'ech
o $'再次制表符: [\009]'echo $'
换页符: [\f]'echo $'换行符'
```

第21.4节：引用文本字面量

本段中的所有示例都会打印该行

```
!"#$&'()*;<=>?  @[\]^`{|}~
```

反斜杠引用下一个字符，即下一个字符被字面解释。唯一的例外是换行符：反斜杠-换行符扩展为空字符串。

```
echo !'"#$&'()*;<=>?  @[\]^`{|}~
```

所有单引号（直引号'，也称为撇号）之间的文本都按字面打印。即使反斜杠也表示其自身，且无法直接包含单引号；相反，可以结束字面字符串，用反斜杠包含一个字面单引号，然后重新开始字面字符串。因此，4字符序列'\''实际上允许在字面字符串中包含单引号。

```
echo '!"#$&'\''()*;<=>?  @[\]^`{|}~'
```



```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\""
"'"
$ a='var'
$ echo '$a'
$a
$ echo "$a"
var
```

Section 21.3: Newlines and control characters

A newline can be included in a single-quoted string or double-quoted string. Note that backslash-newline does not result in a newline, the line break is ignored.

```
newline1='
'
newline2="
"
newline3=$'\n'
empty=\

echo "Line${newline1}break"
echo "Line${newline2}break"
echo "Line${newline3}break"
echo "No line break${empty} here"
```

Inside dollar-quote strings, backslash-letter or backslash-octal can be used to insert control characters, like in many other programming languages.

```
echo $'Tab: [\t]'
echo $'Tab again: [\009]'
echo $'Form feed: [\f]'
echo $'Line\nbreak'
```

Section 21.4: Quoting literal text

All the examples in this paragraph print the line

```
!"#$&'()*;<=>?  @[\]^`{|}~
```

A backslash quotes the next character, i.e. the next character is interpreted literally. The one exception is a newline: backslash-newline expands to the empty string.

```
echo \!\"\#\$\&\'\(\)\*\;\<\=\>\?\ \ \@\[\\\]\^\`\{\|\}\~
```

All text between single quotes (forward quotes `'`, also known as apostrophe) is printed literally. Even backslash stands for itself, and it's impossible to include a single quote; instead, you can stop the literal string, include a literal single quote with a backslash, and start the literal string again. Thus the 4-character sequence `'\''` effectively allow to include a single quote in a literal string.

```
echo '!"#$&'\''()*;<=>?  @[\]^`{|}~'
```

```
#          ^^^^
```

美元符号加单引号开始一个字符串字面量$'...'，类似许多其他编程语言，其中反斜杠引用下一个字符。

```
echo $'!"#$&\'()*;<=>?  @[\\]^`{|}~'
#          ^^           ^^
```

双引号"界定半字面字符串，其中只有字符" \ $和`保留其特殊含义。
这些字符前面需要加反斜杠（注意如果反斜杠后面跟着其他字符，反斜杠保持不变）。双引号主要用于包含变量或命令替换时。

```
echo "!\"#\$&'()*;<=>?  @[\\]^\`{|}~"
#      ^^               ^^  ^^
echo "!\"#\$&'()*;<=>?  @[\]^\`{|}~"
#      ^^               ^  ^^      \[ 打印 \[
```

交互式使用时，要注意 ！ 会在双引号内触发历史扩展："!oops" 会查找包含 oops 的旧命令；"\!oops" 不会进行历史扩展，但会保留反斜杠。脚本中不会发生这种情况。


```
#          ^^^^
```

Dollar-single-quote starts a string literal `$'...'` like many other programming languages, where backslash quotes the next character.

```
echo $'!"#$&\'()*;<=>?  @[\\]^`{|}~'
#          ^^           ^^
```

Double quotes `"` delimit semi-literal strings where only the characters `" \ $` and `` ` `` retain their special meaning. These characters need a backslash before them (note that if backslash is followed by some other character, the backslash remains). Double quotes are mostly useful when including a variable or a command substitution.

```
echo "!\"#\$&'()*;<=>?  @[\\]^\`{|}~"
#      ^^               ^^  ^^
echo "!\"#\$&'()*;<=>?  @[\]^\`{|}~"
#      ^^               ^  ^^      \[ prints \[
```

Interactively, beware that `!` triggers history expansion inside double quotes: `"!oops"` looks for an older command containing `oops`; `"\!oops"` doesn't do history expansion but keeps the backslash. This does not happen in scripts.

第22章：条件表达式

第22.1节：文件类型测试

-e 条件操作符用于测试文件是否存在（包括所有文件类型：目录等）。

```
if [[ -e $filename ]]; then
   echo "$filename 存在"
fi
```

还有针对特定文件类型的测试。

```
if [[ -f $filename ]]; then
   echo "$filename 是一个普通文件"
elif [[ -d $filename ]]; then
   echo "$filename 是一个目录"
elif [[ -p $filename ]]; then
   echo "$filename 是一个命名管道"
elif [[ -S $filename ]]; then
   echo "$filename 是一个命名套接字"
elif [[ -b $filename ]]; then
   echo "$filename 是一个块设备"
elif [[ -c $filename ]]; then
   echo "$filename 是一个字符设备"
fi
if [[ -L $filename ]]; then
   echo "$filename 是一个符号链接（指向任意文件类型）"
fi
```

对于符号链接，除了-L之外，这些测试适用于目标，并且对于断开的链接返回假。

```
if [[ -L $filename || -e $filename ]]; then
   echo "$filename 存在（但可能是一个损坏的符号链接）"
fi

if [[ -L $filename && ! -e $filename ]]; then
   echo "$filename 是一个损坏的符号链接"
fi
```

第22.2节：字符串比较与匹配

字符串比较使用==运算符来比较带引号的字符串。使用!=运算符来取反比较结果。

```
if [[ "$string1" == "$string2" ]]; then
   echo "\$string1 和 \$string2 相同"
fi
if [[ "$string1" != "$string2" ]]; then
   echo "\$string1 和 \$string2 不相同"
fi
```

如果右侧没有加引号，则它是一个通配符模式，$string1 会与之匹配。

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
  # 测试为真
```


Chapter 22: Conditional Expressions

Section 22.1: File type tests

The `-e` conditional operator tests whether a file exists (including all file types: directories, etc.).

```
if [[ -e $filename ]]; then
  echo "$filename exists"
fi
```

There are tests for specific file types as well.

```
if [[ -f $filename ]]; then
  echo "$filename is a regular file"
elif [[ -d $filename ]]; then
  echo "$filename is a directory"
elif [[ -p $filename ]]; then
  echo "$filename is a named pipe"
elif [[ -S $filename ]]; then
  echo "$filename is a named socket"
elif [[ -b $filename ]]; then
  echo "$filename is a block device"
elif [[ -c $filename ]]; then
  echo "$filename is a character device"
fi
if [[ -L $filename ]]; then
  echo "$filename is a symbolic link (to any file type)"
fi
```

For a symbolic link, apart from `-L`, these tests apply to the target, and return false for a broken link.

```
if [[ -L $filename || -e $filename ]]; then
  echo "$filename exists (but may be a broken symbolic link)"
fi

if [[ -L $filename && ! -e $filename ]]; then
  echo "$filename is a broken symbolic link"
fi
```

Section 22.2: String comparison and matching

String comparison uses the == operator between *quoted* strings. The `!=` operator negates the comparison.

```
if [[ "$string1" == "$string2" ]]; then
  echo "\$string1 and \$string2 are identical"
fi
if [[ "$string1" != "$string2" ]]; then
  echo "\$string1 and \$string2 are not identical"
fi
```

If the right-hand side is not quoted then it is a wildcard pattern that `$string1` is matched against.

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
  # the test is true
```

```
  echo "字符串 $string 匹配模式 $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
  # 测试为假
  echo "字符串 $string 不匹配模式 $pattern"
fi
```

操作符 < 和 > 按字典序比较字符串（字符串没有小于等于或大于等于的操作符）。

有用于空字符串的一元测试。

```
if [[ -n "$string" ]]; then
  echo "$string 不是空的"
fi
if [[ -z "${string// }" ]]; then
  echo "$string 为空或仅包含空格"
fi
if [[ -z "$string" ]]; then
  echo "$string 为空"
fi
```

上面，-z 检查可能表示 $string 未设置，或者设置为空字符串。要区分空和未设置，使用：

```
if [[ -n "${string+x}" ]]; then
    echo "$string 已设置，可能为空字符串"
fi
if [[ -n "${string-x}" ]]; then
    echo "$string 未设置或设置为非空字符串"
fi
if [[ -z "${string+x}" ]]; then
    echo "$string 未设置"
fi
if [[ -z "${string-x}" ]]; then
    echo "$string 被设置为空字符串"
fi
```

其中 x 是任意的。或者以 table form 形式：

```
+-------+-------+-----------+
$string 是： | 未设置 | 空字符串 | 非空字符串 |
+----------------------+-------+-------+-----------+
| [[ -z ${string} ]]     | true  | true  | false     |
| [[ -z ${string+x} ]]   | true  | false | false     |
| [[ -z ${string-x} ]]   | false | true  | false     |
| [[ -n ${string} ]]     | false | false | true      |
| [[ -n ${string+x} ]]   | false | true  | true      |
| [[ -n ${string-x} ]]   | true  | false | true      |
+----------------------+-------+-------+-----------+
```

或者，可以在 case 语句中检查状态：

```
case ${var+x$var} in
  (x) echo 空字符串;;
  ("") echo 未设置;;
  (x*[![:blank:]]*) echo 非空白;;
  (*) echo 空白
```



```
  echo "The string $string matches the pattern $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
  # the test is false
  echo "The string $string does not match the pattern $pattern"
fi
```

The < and > operators compare the strings in lexicographic order (there are no less-or-equal or greater-or-equal operators for strings).

There are unary tests for the empty string.

```
if [[ -n "$string" ]]; then
  echo "$string is non-empty"
fi
if [[ -z "${string// }" ]]; then
  echo "$string is empty or contains only spaces"
fi
if [[ -z "$string" ]]; then
  echo "$string is empty"
fi
```

Above, the -z check may mean $string is unset, or it is set to an empty string. To distinguish between empty and unset, use:

```
if [[ -n "${string+x}" ]]; then
    echo "$string is set, possibly to the empty string"
fi
if [[ -n "${string-x}" ]]; then
    echo "$string is either unset or set to a non-empty string"
fi
if [[ -z "${string+x}" ]]; then
    echo "$string is unset"
fi
if [[ -z "${string-x}" ]]; then
    echo "$string is set to an empty string"
fi
```

where x is arbitrary. Or in table form:

```
                        +-------+-------+-----------+
$string is: | unset | empty | non-empty |
+-----------------------+-------+-------+-----------+
| [[ -z ${string} ]]     | true  | true  | false     |
| [[ -z ${string+x} ]]   | true  | false | false     |
| [[ -z ${string-x} ]]   | false | true  | false     |
| [[ -n ${string} ]]     | false | false | true      |
| [[ -n ${string+x} ]]   | false | true  | true      |
| [[ -n ${string-x} ]]   | true  | false | true      |
+-----------------------+-------+-------+-----------+
```

Alternatively, the state can be checked in a case statement:

```
case ${var+x$var} in
  (x) echo empty;;
  ("") echo unset;;
  (x*[![:blank:]]*) echo non-blank;;
  (*) echo blank
```

```
esac
```

其中 [:blank:] 是特定于区域设置的水平空白字符（制表符、空格等）。

第22.3节：命令退出状态的测试

退出状态 0：成功
退出状态非 0：失败

测试命令的退出状态：

```
if command;then
    echo '成功'
else
    echo '失败'
fi
```

第22.4节：单行测试

你可以这样做：

```
[[ $s = '某物' ]] && echo '匹配' || echo "不匹配"
[[ $s == '某物' ]] && echo '匹配' || echo "不匹配"
[[ $s != '某物' ]] && echo "不匹配" || echo "匹配"
[[ $s -eq 10 ]] && echo '相等' || echo "不相等"
(( $s == 10 )) && echo '相等' || echo '不相等'
```

退出状态的单行测试：

```
command && echo '退出状态为0' || echo '非0退出'
cmd && cmd1 && echo '之前的命令都成功' || echo '其中一个失败'
cmd || cmd1 #如果cmd失败则尝试cmd1
```

第22.5节：文件比较

```
if [[ $file1 -ef $file2 ]]; then
  echo "$file1 和 $file2 是同一个文件"
fi
```

“同一文件”意味着就地修改其中一个文件会影响另一个文件。即使两个文件名称不同，它们也可以是同一个文件，例如如果它们是硬链接，或者它们是指向相同目标的符号链接，或者其中一个是指向另一个的符号链接。

如果两个文件内容相同，但它们是不同的文件（因此修改其中一个不会影响另一个），那么 -ef 会报告它们为不同文件。如果你想逐字节比较两个文件，请使用 cmp 工具。

```
if cmp -s -- "$file1" "$file2"; then
  echo "$file1 和 $file2 内容相同"
else
  echo "$file1 和 $file2 不同"
fi
```

要生成文本文件之间差异的可读列表，请使用 diff 工具。

```
if diff -u "$file1" "$file2"; then
```



```
esac
```

Where [:blank:] is locale specific horizontal spacing characters (tab, space, etc).

Section 22.3: Test on exit status of a command

Exit status 0: success
Exit status other than 0: failure

To test on the exit status of a command:

```
if command;then
    echo 'success'
else
    echo 'failure'
fi
```

Section 22.4: One liner test

You can do things like this:

```
[[ $s = 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s == 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s != 'something' ]] && echo "didn't match" || echo "matched"
[[ $s -eq 10 ]] && echo 'equal' || echo "not equal"
(( $s == 10 )) && echo 'equal' || echo 'not equal'
```

One liner test for exit status:

```
command && echo 'exited with 0' || echo 'non 0 exit'
cmd && cmd1 && echo 'previous cmds were successful' || echo 'one of them failed'
cmd || cmd1 #If cmd fails try cmd1
```

Section 22.5: File comparison

```
if [[ $file1 -ef $file2 ]]; then
  echo "$file1 and $file2 are the same file"
fi
```

“Same file” means that modifying one of the files in place affects the other. Two files can be the same even if they have different names, for example if they are hard links, or if they are symbolic links with the same target, or if one is a symbolic link pointing to the other.

If two files have the same content, but they are distinct files (so that modifying one does not affect the other), then -ef reports them as different. If you want to compare two files byte by byte, use the **cmp** utility.

```
if cmp -s -- "$file1" "$file2"; then
  echo "$file1 and $file2 have identical contents"
else
  echo "$file1 and $file2 differ"
fi
```

To produce a human-readable list of differences between text files, use the **diff** utility.

```
if diff -u "$file1" "$file2"; then
```

```
  echo "$file1 和 $file2 内容相同"
否则
: # 文件之间的差异已被列出
fi
```

第22.6节：文件访问测试

```
if [[ -r $filename ]]; then
  echo "$filename 是一个可读文件"
fi
if [[ -w $filename ]]; then
  echo "$filename 是一个可写文件"
fi
if [[ -x $filename ]]; then
  echo "$filename 是一个可执行文件"
fi
```

这些测试会考虑权限和所有权，以确定脚本（或从脚本启动的程序）是否可以访问该文件。

注意竞争条件（TOCTOU）：仅仅因为测试现在成功，并不意味着下一行仍然有效。通常最好尝试访问文件并处理错误，而不是先测试然后再处理错误，以防文件在此期间发生了变化。

第22.7节：数值比较

数值比较使用-eq运算符及其相关运算符

```
if [[ $num1 -eq $num2 ]]; then
  echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
  echo "$num1 <= $num2"
fi
```

共有六个数值运算符：

- -eq 等于
- -ne 不等于
- -le 小于或等于
- -lt 小于
- -ge 大于或等于
- -gt 大于

请注意，< 和 > 运算符在 [[...]] 中比较的是字符串，而不是数字。

```
if [[ 9 -lt 10 ]]; then
  echo "9 在数字顺序中位于 10 之前"
fi
if [[ 9 > 10 ]]; then
  echo "9 在字典顺序中位于 10 之后"
fi
```

两边必须是以十进制（或以零开头的八进制）书写的数字。或者，使用 ((...)) 算术表达式语法，它以类似 C/Java/... 的语法执行整数计算。


```
  echo "$file1 and $file2 have identical contents"
else
  : # the differences between the files have been listed
fi
```

Section 22.6: File access tests

```
if [[ -r $filename ]]; then
  echo "$filename is a readable file"
fi
if [[ -w $filename ]]; then
  echo "$filename is a writable file"
fi
if [[ -x $filename ]]; then
  echo "$filename is an executable file"
fi
```

These tests take permissions and ownership into account to determine whether the script (or programs launched from the script) can access the file.

Beware of race conditions (TOCTOU): just because the test succeeds now doesn't mean that it's still valid on the next line. It's usually better to try to access a file, and handle the error, rather than test first and then have to handle the error anyway in case the file has changed in the meantime.

Section 22.7: Numerical comparisons

Numerical comparisons use the `-eq` operators and friends

```
if [[ $num1 -eq $num2 ]]; then
  echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
  echo "$num1 <= $num2"
fi
```

There are six numeric operators:

- `-eq` equal
- `-ne` not equal
- `-le` less or equal
- `-lt` less than
- `-ge` greater or equal
- `-gt` greater than

Note that the `<` and `>` operators inside `[[ ... ]]` compare strings, not numbers.

```
if [[ 9 -lt 10 ]]; then
  echo "9 is before 10 in numeric order"
fi
if [[ 9 > 10 ]]; then
  echo "9 is after 10 in lexicographic order"
fi
```

The two sides must be numbers written in decimal (or in octal with a leading zero). Alternatively, use the `((...))` arithmetic expression syntax, which performs **integer** calculations in a C/Java/...-like syntax.

```
x=2
if ((2*x == 4)); then
   echo "2 乘以 2 等于 4"
fi
((x += 1))
echo "2 加 1 等于 $x"
```



```
x=2
if ((2*x == 4)); then
  echo "2 times 2 is 4"
fi
((x += 1))
echo "2 plus 1 is $x"
```

第23章：带参数的脚本编写

第23.1节：多参数解析

要解析大量参数，推荐的做法是使用while循环、case语句和shift。

`shift`用于弹出参数序列中的第一个参数，使原本的*$2*变为*$1*。这对于逐个处理参数非常有用。

```
#!/bin/bash

# 加载用户定义的参数
while [[ $# > 0 ]]
do
        case "$1" in

                -a|--valueA)
                        valA="$2"
                        shift
                        ;;

                -b|--valueB)
                        valB="$2"
                        shift
                        ;;

                --help|*)
                        echo "用法:"
                        echo "    --valueA \"值\""
                        echo "    --valueB \"值\""
                        echo "    --help"
                        exit 1
                        ;;
        esac
        shift
done

echo "A: $valA"
echo "B: $valB"
```

输入与输出

```
$ ./multipleParams.sh --help
用法：
    --valueA "value"
    --valueB "value"
    --help

$ ./multipleParams.sh
A:
B:

$ ./multipleParams.sh --valueB 2
A:
B: 2

$ ./multipleParams.sh --valueB 2 --valueA "hello world"
A: hello world
```


Chapter 23: Scripting with Parameters

Section 23.1: Multiple Parameter Parsing

To parse lots of parameters, the preferred way of doing this is using a *while* loop, a *case* statement, and *shift*.

`shift` is used to pop the first parameter in the series, making what used to be *$2*, now be *$1*. This is useful for processing arguments one at a time.

```
#!/bin/bash

# Load the user defined parameters
while [[ $# > 0 ]]
do
        case "$1" in

                -a|--valueA)
                        valA="$2"
                        shift
                        ;;

                -b|--valueB)
                        valB="$2"
                        shift
                        ;;

                --help|*)
                        echo "Usage:"
                        echo "    --valueA \"value\""
                        echo "    --valueB \"value\""
                        echo "    --help"
                        exit 1
                        ;;
        esac
        shift
done

echo "A: $valA"
echo "B: $valB"
```

Inputs and Outputs

```
$ ./multipleParams.sh --help
Usage:
    --valueA "value"
    --valueB "value"
    --help

$ ./multipleParams.sh
A:
B:

$ ./multipleParams.sh --valueB 2
A:
B: 2

$ ./multipleParams.sh --valueB 2 --valueA "hello world"
A: hello world
```

```
B: 2
```

第23.2节：使用for循环解析参数

一个提供以下选项的简单示例：

选项	备用选项	详细信息
-h	--help	显示帮助
-v	--version	显示版本信息
-dr 路径	--doc-root 路径	一个带有次级参数（路径）的选项
-i	--install	布尔选项（真/假）
-*	--	无效选项

```
#!/bin/bash
dr=''
install=false

skip=false
for op in "$@";do
    if $skip;then skip=false;continue;fi
    case "$op" in
        -v|--version)
            echo "$ver_info"
            shift
            exit 0
            ;;
        -h|--help)
            echo "$help"
            shift
            exit 0
            ;;
        -dr|--doc-root)
            shift
            if [[ "$1" != "" ]]; then
                dr="${1/%\//}"
                shift
                skip=true
            否则
                echo "E: Arg missing for -dr option"
                exit 1
            fi
            ;;
        -i|--install)
            install=true
            shift
            ;;
-*)
            echo "E: Invalid option: $1"
            shift
            exit 1
            ;;
    esac
done
```

第23.3节：包装脚本

包装脚本是一种包装另一个脚本或命令的脚本，用于提供额外功能或只是为了使某些操作不那么繁琐。

B: 2

Section 23.2: Argument parsing using a for loop

A simple example which provides the options:

Opt	Alt. Opt	Details
-h	--help	Show help
-v	--version	Show version info
-dr path	--doc-root path	An option which takes a secondary parameter (a path)
-i	--install	A boolean option (true/false)
-*	--	Invalid option

```
#!/bin/bash
dr=''
install=false

skip=false
for op in "$@";do
    if $skip;then skip=false;continue;fi
    case "$op" in
        -v|--version)
            echo "$ver_info"
            shift
            exit 0
            ;;
        -h|--help)
            echo "$help"
            shift
            exit 0
            ;;
        -dr|--doc-root)
            shift
            if [[ "$1" != "" ]]; then
                dr="${1/%\//}"
                shift
                skip=true
            else
                echo "E: Arg missing for -dr option"
                exit 1
            fi
            ;;
        -i|--install)
            install=true
            shift
            ;;
        -*)
            echo "E: Invalid option: $1"
            shift
            exit 1
            ;;
    esac
done
```

Section 23.3: Wrapper script

Wrapper script is a script that wraps another script or command to provide extra functionalities or just to make something less tedious.

例如，新的GNU/Linux系统中实际的grep被一个名为grep的包装脚本替代。它的样子是这样的：

```
#!/bin/sh
exec grep -E "$@"
```

因此，当你在这样的系统中运行grep时，实际上是运行grep -E，并转发所有参数。

一般情况下，如果你想用另一个脚本mexmp来运行示例脚本/命令exmp，那么包装脚本mexmp看起来会是这样的：

```
#!/bin/sh
exmp "$@" # 在"$@"之前添加其他选项
# 或者
#full/path/to/exmp "$@"
```

第23.4节：访问参数

执行Bash脚本时，传入脚本的参数根据其位置命名：$1是第一个参数的名称，$2是第二个参数的名称，依此类推。

缺失的参数会被简单地评估为空字符串。检查参数是否存在可以按如下方式进行：

```
if [ -z "$1" ]; then
    echo "未提供参数"
fi
```

获取所有参数

$@ 和 $* 是与所有脚本参数交互的方式。参考 Bash 手册页，我们看到：

- $*：展开为位置参数，从第一个开始。当展开发生在双引号内时，它会展开为一个单词，参数值之间用 IFS 特殊变量的第一个字符分隔。
- $@：展开为位置参数，从第一个开始。当展开发生在双引号内时，每个参数展开为一个单独的单词。

获取参数数量

$# 获取传入脚本的参数数量。一个典型的用例是检查是否传入了适当数量的参数：

```
if [ $# -eq 0 ]; then
    echo "未提供参数"
fi
```

示例 1

遍历所有参数并检查它们是否为文件：

```
for item in "$@"
do
    if [[ -f $item ]]; then
        echo "$item 是一个文件"
```


For example, the actual **egrep** in new GNU/Linux system is being replaced by a wrapper script named **egrep**. This is how it looks:

```
#!/bin/sh
exec grep -E "$@"
```

So, when you run **egrep** in such systems, you are actually running **grep -E** with all the arguments forwarded.

In general case, if you want to run an example script/command exmp with another script mexmp then the wrapper mexmp script will look like:

```
#!/bin/sh
exmp "$@" # Add other options before "$@"
# or
#full/path/to/exmp "$@"
```

Section 23.4: Accessing Parameters

When executing a Bash script, parameters passed into the script are named in accordance to their position: $1 is the name of the first parameter, $2 is the name of the second parameter, and so on.

A missing parameter simply evaluates to an empty string. Checking for the existence of a parameter can be done as follows:

```
if [ -z "$1" ]; then
    echo "No argument supplied"
fi
```

Getting all the parameters

$@ and $* are ways of interacting with all the script parameters. Referencing the Bash man page, we see that:

- $*: Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable.
- $@: Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.

Getting the number of parameters

$# gets the number of parameters passed into a script. A typical use case would be to check if the appropriate number of arguments are passed:

```
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
fi
```

Example 1

Loop through all arguments and check if they are files:

```
for item in "$@"
do
    if [[ -f $item ]]; then
        echo "$item is a file"
```

```
    fi
done
```

示例 2

遍历所有参数并检查它们是否为文件：

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}

    if [[ -f $item ]]; then
        echo "$item 是一个文件"
    fi
done
```

第 23.5 节：在 Bash 中将字符串拆分为数组

假设我们有一个字符串参数，想要按逗号拆分它

```
my_param="foo,bar,bash"
```

要按逗号拆分这个字符串，我们可以使用；

```
IFS=',' read -r -a array <<< "$my_param"
```

这里，IFS 是一个称为内部字段分隔符的特殊变量，用于定义在某些操作中将模式分割成标记时使用的字符或字符集。

访问单个元素的方法：

```
echo "${array[0]}"
```

遍历元素的方法：

```
for element in "${array[@]}"
do
    echo "$element"
done
```

获取索引和值的方法：

```
for index in "${!array[@]}"
do
    echo "$index ${array[index]}"
done
```



```
    fi
done
```

Example 2

Loop through all arguments and check if they are files:

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}

    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Section 23.5: Split string into an array in Bash

Let's say we have a String parameter and we want to split it by comma

```
my_param="foo,bar,bash"
```

To split this string by comma we can use;

```
IFS=',' read -r -a array <<< "$my_param"
```

Here, IFS is a special variable called Internal field separator which defines the character or characters used to separate a pattern into tokens for some operations.

To access an individual element:

```
echo "${array[0]}"
```

To iterate over the elements:

```
for element in "${array[@]}"
do
    echo "$element"
done
```

To get both the index and the value:

```
for index in "${!array[@]}"
do
    echo "$index ${array[index]}"
done
```

第24章：Bash历史替换

第24.1节：快速参考

与历史的交互

```
# 列出所有之前的命令
history

# 清除历史记录，如果你不小心输入了密码，这很有用
history -c
```

事件指定符

```
# 展开为bash历史中的第n行
!n

# 展开为最后一条命令
!!

# 展开为以 "text" 开头的最后一条命令
!text

# 展开为包含 "text" 的最后一条命令
!?text

# 展开为 n 行前的命令
!-n

# 展开为最后一条命令，将首次出现的 "foo" 替换为 "bar"
^foo^bar^

# 展开为当前命令
!#
```

单词指定符

这些由:与它们所指的事件标识符分开。如果单词标识符不以数字开头，则冒号可以省略：!^与!:^是相同的。

```
# 展开为最近命令的第一个参数
!^

# 展开为最近命令的最后一个参数（!!:$ 的简写）
!$

# 展开为最近命令的第三个参数
!:3

# 展开为最近命令的第 x 到第 y 个参数（包含 x 和 y）
# x 和 y 可以是数字或锚字符 ^ $
!:x-y

# 展开为最近命令的所有参数，除第 0 个之外
# 等同于 :^-$
!*
```

修饰符

这些修饰前面的事件或单词标识符。

```
# 使用 sed 语法进行替换扩展
```


Chapter 24: Bash history substitutions

Section 24.1: Quick Reference

Interaction with the history

```
# List all previous commands
history

# Clear the history, useful if you entered a password by accident
history -c
```

Event designators

```
# Expands to line n of bash history
!n

# Expands to last command
!!

# Expands to last command starting with "text"
!text

# Expands to last command containing "text"
!?text

# Expands to command n lines ago
!-n

# Expands to last command with first occurrence of "foo" replaced by "bar"
^foo^bar^

# Expands to the current command
!#
```

Word designators

These are separated by `:` from the event designator they refer to. The colon can be omitted if the word designator doesn't start with a number: `!^` is the same as `!:^`.

```
# Expands to the first argument of the most recent command
!^

# Expands to the last argument of the most recent command (short for !!:$)
!$

# Expands to the third argument of the most recent command
!:3

# Expands to arguments x through y (inclusive) of the last command
# x and y can be numbers or the anchor characters ^ $
!:x-y

# Expands to all words of the last command except the 0th
# Equivalent to :^-$
!*
```

Modifiers

These modify the preceding event or word designator.

```
# Replacement in the expansion using sed syntax
```

```
# 允许在 s 之前使用标志和替代分隔符
:s/foo/bar/ # 将第一个 foo 替换为 bar
:gs|foo|bar| # 将所有 foo 替换为 bar

# 从最后一个参数中移除前导路径（“尾部”）
:t

# 从最后一个参数中移除尾部路径（“头部”）
:h

# 从最后一个参数中移除文件扩展名
:r
```

如果 Bash 变量HISTCONTROL包含ignorespace或ignoreboth（或者，替代地，HISTIGNORE包含模式[ ]*），你可以通过在命令前加一个空格来防止命令被保存到 Bash 历史记录中：

```
# 该命令不会被保存到历史记录中
 foo

# 该命令会被保存
bar
```

第24.2节：使用 sudo 重复执行上一个命令

```
$ apt-get install r-base
E: 无法打开锁文件/var/lib/dpkg/lock - 打开失败(13: 权限被拒绝)
E: 无法锁定管理目录(/var/lib/dpkg/)，你是 root 用户吗？
$ sudo !!
sudo apt-get install r-base
[sudo] 用户 <user> 的密码:
```

第24.3节：通过模式在命令历史中搜索

按下 `control r` 并输入一个模式。

例如，如果你最近执行过`man 5 crontab`，可以通过*开始输入*“crontab”快速找到它。
提示符将变成这样：

```
(reverse-i-search)`cr': man 5 crontab
```

这里的`cr'是我目前输入的字符串。这是一个增量搜索，因此随着你继续输入，搜索结果会更新为匹配包含该模式的最新命令。

按左箭头或右箭头键在运行前编辑匹配的命令，或按 `enter` 键运行命令。

默认情况下，搜索会找到与模式匹配的最近执行的命令。要在历史记录中更早地查找，请按 `control r` 再次。您可以反复按下，直到找到所需的命令。

第24.4节：使用 !#:N 切换到新创建的目录

```
$ mkdir backup_download_directory && cd !#:1
mkdir backup_download_directory && cd backup_download_directory
```

这将替换当前命令的第N个参数。在示例中，!#:1 被替换为第一个


```
# Allows flags before the s and alternate separators
:s/foo/bar/ #substitutes bar for first occurrence of foo
:gs|foo|bar| #substitutes bar for all foo

# Remove leading path from last argument ("tail")
:t

# Remove trailing path from last argument ("head")
:h

# Remove file extension from last argument
:r
```

If the Bash variable `HISTCONTROL` contains either `ignorespace` or `ignoreboth` (or, alternatively, `HISTIGNORE` contains the pattern `[ ]*`), you can prevent your commands from being stored in Bash history by prepending them with a space:

```
# This command won't be saved in the history
 foo

# This command will be saved
bar
```

Section 24.2: Repeat previous command with sudo

```
$ apt-get install r-base
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
$ sudo !!
sudo apt-get install r-base
[sudo] password for <user>:
```

Section 24.3: Search in the command history by pattern

Press `control r` and type a pattern.

For example, if you recently executed `man 5 crontab`, you can find it quickly by *starting to type* "crontab". The prompt will change like this:

```
(reverse-i-search)`cr': man 5 crontab
```

The `cr' there is the string I typed so far. This is an incremental search, so as you continue typing, the search result gets updated to match the most recent command that contained the pattern.

Press the left or right arrow keys to edit the matched command before running it, or the `enter` key to run the command.

By default the search finds the most recently executed command matching the pattern. To go further back in the history press `control r` again. You may press it repeatedly until you find the desired command.

Section 24.4: Switch to newly created directory with !#:N

```
$ mkdir backup_download_directory && cd !#:1
mkdir backup_download_directory && cd backup_download_directory
```

This will substitute the Nth argument of the current command. In the example `!#:1` is replaced with the first

参数，即 backup_download_directory。

第24.5节：使用 !$

您可以使用 !$ 来减少命令行中的重复：

```
$ echo ping
ping
$ echo !$
ping
```

您也可以在重复的基础上进行扩展

```
$ echo !$ pong
ping pong
$ echo !$, 一个很棒的游戏
pong，一个很棒的游戏
```

注意，在上一个例子中我们没有得到 ping pong，一个很棒的游戏，因为传递给前一个命令的最后一个参数是 pong，我们可以通过添加引号来避免此类问题。继续这个例子，我们的最后一个参数是 game：

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

第24.6节：用替换重复执行前一个命令

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

该命令将在之前执行的命令中将 1 替换为 2。它只会替换字符串的第一个出现位置，等同于 !!:s/1/2/。

如果你想替换 所有 出现的位置，必须使用 !!:gs/1/2/ 或 !!:as/1/2/。

argument, i.e. backup_download_directory.

Section 24.5: Using !$

You can use the `!$` to reduce repetition when using the command line:

```
$ echo ping
ping
$ echo !$
ping
```

You can also build upon the repetition

```
$ echo !$ pong
ping pong
$ echo !$, a great game
pong, a great game
```

Notice that in the last example we did not get `ping pong, a great game` because the last argument passed to the previous command was `pong`, we can avoid issue like this by adding quotes. Continuing with the example, our last argument was `game`:

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

Section 24.6: Repeat the previous command with a substitution

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

This command will replace `1` with `2` in the previously executed command. It will only replace the first occurrence of the string and is equivalent to `!!:s/1/2/`.

If you want to replace *all* occurrences, you have to use `!!:gs/1/2/` or `!!:as/1/2/`.

第25章：数学

第25.1节：使用dc进行数学运算

dc 是Unix上最古老的程序之一。

它使用逆波兰表示法，这意味着你先将数字入栈，然后进行操作。例如1+1写成1 1+。

要打印栈顶元素，使用命令p

```
echo '2 3 + p' | dc
5
```

或者

```
dc <<< '2 3 + p'
5
```

你可以多次打印栈顶元素

```
dc <<< '1 1 + p 2 + p'
2
4
```

对于负数使用 _前缀

```
dc <<< '_1 p'
-1
```

你也可以使用大写字母A到F表示数字10到15，.作为小数点

```
dc <<< 'A.4 p'
10.4
```

dc使用的是任意精度，这意味着精度仅受可用内存限制。默认情况下，
精度设置为0位小数

```
dc <<< '4 3 / p'
1
```

我们可以使用命令k来增加精度。2k将使用

```
dc <<< '2k 4 3 / p'
1.33

dc <<< '4k 4 3 / p'
1.3333
```

你也可以在多行上使用它

```
dc << EOF
1 1 +
3 *
p
```


Chapter 25: Math

Section 25.1: Math using dc

dc is one of the oldest programs on Unix.

It uses reverse polish notation, which means that you first stack numbers, then operations. For example 1+1 is written as 1 1+.

To print an element from the top of the stack use command p

```
echo '2 3 + p' | dc
5

or

dc <<< '2 3 + p'
5
```

You can print the top element many times

```
dc <<< '1 1 + p 2 + p'
2
4
```

For negative numbers use _ prefix

```
dc <<< '_1 p'
-1
```

You can also use capital letters from A to F for numbers between 10 and 15 and . as a decimal point

```
dc <<< 'A.4 p'
10.4
```

dc is using abitrary precision which means that the precision is limited only by the available memory. By default the precision is set to 0 decimals

```
dc <<< '4 3 / p'
1
```

We can increase the precision using command k. 2k will use

```
dc <<< '2k 4 3 / p'
1.33

dc <<< '4k 4 3 / p'
1.3333
```

You can also use it over multiple lines

```
dc << EOF
1 1 +
3 *
p
```

```
EOF
6
```

> bc 是 dc 的预处理器。

第25.2节：使用bash功能进行数学运算

算术计算也可以在不调用任何其他程序的情况下完成，如下所示：

乘法：

```
echo $((5 * 2))
10
```

除法：

```
echo $((5 / 2))
2
```

取模：

```
echo $((5 % 2))
1
```

指数运算：

```
echo $((5 ** 2))
25
```

第25.3节：使用bc进行数学运算

bc 是一种任意精度计算器语言。它可以交互使用，也可以从命令行执行。

例如，它可以输出表达式的结果：

```
echo '2 + 3' | bc
5

echo '12 / 5' | bc
2
```

对于浮点运算，你可以导入标准库 bc -l：

```
echo '12 / 5' | bc -l
2.40000000000000000000
```

它可以用于比较表达式：

```
echo '8 > 5' | bc
1

echo '10 == 11' | bc
0
```



```
EOF
6
```

> bc is a preprocessor for dc.

Section 25.2: Math using bash capabilities

Arithmetic computation can be also done without involving any other programs like this:

Multiplication:

```
echo $((5 * 2))
10
```

Division:

```
echo $((5 / 2))
2
```

Modulo:

```
echo $((5 % 2))
1
```

Exponentiation:

```
echo $((5 ** 2))
25
```

Section 25.3: Math using bc

bc is an arbitrary precision calculator language. It could be used interactively or be executed from command line.

For example, it can print out the result of an expression:

```
echo '2 + 3' | bc
5

echo '12 / 5' | bc
2
```

For floating-post arithmetic, you can import standard library `bc -l`:

```
echo '12 / 5' | bc -l
2.40000000000000000000
```

It can be used for comparing expressions:

```
echo '8 > 5' | bc
1

echo '10 == 11' | bc
0
```

```
echo '10 == 10 && 8 > 3' | bc
1
```

第25.4节：使用expr进行数学运算

`expr` 或者 `计算表达式` 计算一个表达式并将结果写到标准输出

基本算术运算

```
expr 2 + 3
5
```

乘法时，需要对*符号进行转义

```
expr 2 \* 3
6
```

你也可以使用变量

```
a=2
expr $a + 3
5
```

请记住它只支持整数，所以像这样的表达式

```
expr 3.0 / 2
```

将抛出错误 `expr: 不是十进制数字: '3.0'`。

它支持正则表达式来匹配模式

```
expr 'Hello World' : 'Hell\(.*\)rld'
o Wo
```

或者查找搜索字符串中第一个字符的索引

> 这将在Mac OS X上抛出 **`expr: 语法错误`**，因为它使用的是**BSD expr**，后者没有index命令，而Linux上的expr通常是**GNU expr**

```
expr index hello l
3

expr index 'hello' 'lo'
3
```



```
echo '10 == 10 && 8 > 3' | bc
1
```

Section 25.4: Math using expr

`expr` or `Evaluate expressions` evaluates an expression and writes the result on standard output

Basic arithmetics

```
expr 2 + 3
5
```

When multiplying, you need to escape the `*` sign

```
expr 2 \* 3
6
```

You can also use variables

```
a=2
expr $a + 3
5
```

Keep in mind that it only supports integers, so expression like this

```
expr 3.0 / 2
```

will throw an error `expr: not a decimal number: '3.0'`.

It supports regular expression to match patterns

```
expr 'Hello World' : 'Hell\(.*\)rld'
o Wo
```

Or find the index of the first char in the search string

> This will throw `expr: syntax error` on **Mac OS X**, because it uses **BSD expr** which does not have the index command, while expr on Linux is generally **GNU expr**

```
expr index hello l
3

expr index 'hello' 'lo'
3
```

第26章：Bash算术

参数	详细信息
EXPRESSION	要计算的表达式

第26.1节：使用(( ))进行简单算术运算

```
#!/bin/bash
echo $(( 1 + 2 ))
```

输出：3

```
# 使用变量
#!/bin/bash
var1=4
var2=5
((output=$var1 * $var2))print
f "%d" "$output"
```

输出：20

第26.2节：算术命令

- `let`

```
let num=1+2
let num="1+2"
let 'num= 1 + 2'
let num=1 num+=2
```

如果有空格或通配符字符，则需要使用引号。否则会报错：

```
let num = 1 + 2    #错误
let 'num = 1 + 2'  #正确
let a[1] = 1 + 1   #错误
let 'a[1] = 1 + 1' #正确
```

- `(( ))`

```
((a=$a+1))     #给 a 加 1
((a = a + 1))  #同上
((a += 1))     #同上
```

我们可以在 `if` 中使用 `(())`。示例：

```
if (( a > 1 )); then echo "a 大于 1"; fi
```

`(())` 的输出可以赋值给变量：

```
result=$((a + 1))
```

或者直接用于输出：

```
echo "a + 1 的结果是 $((a + 1))"
```


Chapter 26: Bash Arithmetic

Parameter	Details
EXPRESSION	Expression to evaluate

Section 26.1: Simple arithmetic with (( ))

```
#!/bin/bash
echo $(( 1 + 2 ))
```

Output: 3

```
# Using variables
#!/bin/bash
var1=4
var2=5
((output=$var1 * $var2))
printf "%d\n" "$output"
```

Output: 20

Section 26.2: Arithmetic command

- `let`

```
let num=1+2
let num="1+2"
let 'num= 1 + 2'
let num=1 num+=2
```

You need quotes if there are spaces or globbing characters. So those will get error:

```
let num = 1 + 2    #wrong
let 'num = 1 + 2'  #right
let a[1] = 1 + 1   #wrong
let 'a[1] = 1 + 1' #right
```

- `(( ))`

```
((a=$a+1))     #add 1 to a
((a = a + 1))  #like above
((a += 1))     #like above
```

We can use `(())` in `if`. Some Example:

```
if (( a > 1 )); then echo "a is greater than 1"; fi
```

The output of `(())` can be assigned to a variable:

```
result=$((a + 1))
```

Or used directly in output:

```
echo "The result of a + 1 is $((a + 1))"
```

第26.3节：使用 expr 进行简单算术运算

```
#!/bin/bash
expr 1 + 2
```

输出：3

Section 26.3: Simple arithmetic with expr

```
#!/bin/bash
expr 1 + 2
```

Output: 3

第27章：作用域范围

第27.1节：动态作用域的实际应用

动态作用域意味着变量查找发生在函数被调用的作用域中，而不是函数被定义的作用域中。

```
$ x=3
$ func1 () { echo "在 func1 中: $x"; }
$ func2 () { local x=9; func1; }
$ func2
在 func1 中: 9
$ func1
在 func1 中: 3
```

在词法作用域语言中，func1 会始终在全局作用域中查找 x 的值，因为 func1 是在局部作用域中定义的。

在动态作用域语言中，func1 会在它被*调用*的作用域中查找。当它从 func2 内部被调用时，它首先在 func2 的函数体内查找 x 的值。如果那里没有定义，它会在调用 func2 的全局作用域中查找。

Chapter 27: Scoping

Section 27.1: Dynamic scoping in action

Dynamic scoping means that variable lookups occur in the scope where a function is *called*, not where it is *defined*.

```
$ x=3
$ func1 () { echo "in func1: $x"; }
$ func2 () { local x=9; func1; }
$ func2
in func1: 9
$ func1
in func1: 3
```

In a lexically scoped language, `func1` would *always* look in the global scope for the value of `x`, because `func1` is *defined* in the local scope.

In a dynamically scoped language, `func1` looks in the scope where it is *called*. When it is called from within `func2`, it first looks in the body of `func2` for a value of `x`. If it weren't defined there, it would look in the global scope, where `func2` was called from.

第28章：进程替换

第28.1节：比较来自网络的两个文件

以下内容比较了使用进程替代而非创建临时文件的 diff 命令比较两个文件的方法。

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

第28.2节：用命令的输出作为 while 循环的输入

这将用 grep 命令的输出作为 while 循环的输入：

```
while IFS=":" read -r user _
do
    # "$user" 保存了 /etc/passwd 中的用户名
done < <(grep "hello" /etc/passwd)
```

第28.3节：文件的连接

众所周知，不能在同一条命令中同时使用同一个文件作为输入和输出。例如，

```
$ cat header.txt body.txt >body.txt
```

doesn't do what you want。到 cat 读取 body.txt 时，文件已经被重定向截断，且内容为空。最终结果是 body.txt 只会包含 header.txt 的内容。

有人可能会想通过进程替换来避免这个问题，也就是说，命令

```
$ cat header.txt <(cat body.txt) > body.txt
```

会强制将 body.txt 的原始内容以某种方式先保存到某个缓冲区，然后再通过重定向截断该文件。但这并不奏效。括号中的 cat 只有在所有文件描述符都设置好之后才开始读取文件，就像外层的 cat 一样。在这种情况下，尝试使用进程替换是没有意义的。

将一个文件内容添加到另一个文件前面的唯一方法是创建一个中间文件：

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

这正是 sed、perl 或类似程序在使用"原地编辑"（通常是

第28.4节：同时通过多个程序处理文件流

这条命令使用 wc -l 统计大文件的行数，同时用 gzip 进行压缩。两者并发运行。

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

通常 tee 会将其输入写入一个或多个文件（以及标准输出）。我们可以用 tee 将输入写入命令而不是文件

Chapter 28: Process substitution

Section 28.1: Compare two files from the web

The following compares two files with `diff` using process substitution instead of creating temporary files.

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

Section 28.2: Feed a while loop with the output of a command

This feeds a `while` loop with the output of a `grep` command:

```
while IFS=":" read -r user _
do
    # "$user" holds the username in /etc/passwd
done < <(grep "hello" /etc/passwd)
```

Section 28.3: Concatenating files

It is well known that you cannot use the same file for input and output in the same command. For instance,

```
$ cat header.txt body.txt >body.txt
```

doesn't do what you want. By the time `cat` reads `body.txt`, it has already been truncated by the redirection and it is empty. The final result will be that `body.txt` will hold the contents of `header.txt` *only*.

One might think to avoid this with process substitution, that is, that the command

```
$ cat header.txt <(cat body.txt) > body.txt
```

will force the original contents of `body.txt` to be somehow saved in some buffer somewhere before the file is truncated by the redirection. It doesn't work. The `cat` in parentheses begins reading the file only after all file descriptors have been set up, just like the outer one. There is no point in trying to use process substitution in this case.

The only way to prepend a file to another file is to create an intermediate one:

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

which is what `sed` or `perl` or similar programs do under the carpet when called with an *edit-in-place* option (usually `-i`).

Section 28.4: Stream a file through multiple programs at once

This counts the number of lines in a big file with `wc -l` while simultaneously compressing it with `gzip`. Both run concurrently.

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Normally `tee` writes its input to one or more files (and stdout). We can write to commands instead of files with `tee`

>(command).

这里命令 wc -l >&2 统计从 tee 读取的行数（而 tee 又从 bigfile 读取）。(行数被发送到标准错误输出（>&2），以避免与 gzip 的输入混淆。）tee 的标准输出同时被传递给 gzip。

第28.5节：使用paste命令

```
# 使用paste命令的进程替换很常见
# 用于比较两个目录的内容
paste <( ls /path/to/directory1 ) <( ls /path/to/directory2 )
```

第28.6节：避免使用子shell

进程替换的一个主要方面是它让我们在从shell管道命令时避免使用子shell。

下面用一个简单的例子来演示。我当前文件夹中有以下文件：

```
$ find . -maxdepth 1 -type f -print
foo bar zoo foobar foozoo barzoo
```

如果我将输出通过 while/read 循环来递增计数器，代码如下：

```
count=0
find . -maxdepth 1 -type f -print | while IFS= read -r _; do
    ((count++))
done
```

$count 现在不包含 6，因为它是在子shell上下文中被修改的。下面显示的任何命令都会在子shell上下文中运行，子shell终止后，所使用变量的作用域将丢失。

```
command &
command | command
( command )
```

进程替换将通过避免使用管道符号 | 操作符来解决该问题，如下所示

```
count=0
while IFS= read -r _; do
    ((count++))
done < <(find . -maxdepth 1 -type f -print)
```

这将保留 count 变量的值，因为没有调用子shell。

>(command).

Here the command `wc -l >&2` counts the lines read from `tee` (which in turn is reading from `bigfile`). (The line count is sent to stderr (`>&2`) to avoid mixing with the input to `gzip`.) The stdout of `tee` is simultaneously fed into `gzip`.

Section 28.5: With paste command

```
# Process substitution with paste command is common
# To compare the contents of two directories
paste <( ls /path/to/directory1 ) <( ls /path/to/directory2 )
```

Section 28.6: To avoid usage of a sub-shell

One major aspect of process substitution is that it lets us avoid usage of a sub-shell when piping commands from the shell.

This can be demonstrated with a simple example below. I have the following files in my current folder:

```
$ find . -maxdepth 1 -type f -print
foo bar zoo foobar foozoo barzoo
```

If I pipe to a `while`/`read` loop that increments a counter as follows:

```
count=0
find . -maxdepth 1 -type f -print | while IFS= read -r _; do
    ((count++))
done
```

`$count` now does *not* contain `6`, because it was modified in the sub-shell context. Any of the commands shown below are run in a sub-shell context and the scope of the variables used within are lost after the sub-shell terminates.

```
command &
command | command
( command )
```

Process substitution will solve the problem by avoiding use the of pipe | operator as in

```
count=0
while IFS= read -r _; do
    ((count++))
done < <(find . -maxdepth 1 -type f -print)
```

This will retain the `count` variable value as no sub-shells are invoked.

第29章：可编程补全

第29.1节：使用函数的简单补全

```
_mycompletion() {
    local command_name="$1" # 在此示例中未使用
    local current_word="$2"
    local previous_word="$3" # 在此示例中未使用
    # COMPREPLY 是一个数组，必须填充可能的补全项
    # compgen 用于过滤匹配的补全项
    COMPREPLY=( $(compgen -W 'hello world' -- "$current_word") )
}
complete -F _mycompletion mycommand
```

使用示例：

```
$ mycommand [TAB][TAB]
hello world
$ mycommand h[TAB][TAB]
$ mycommand hello
```

第29.2节：选项和文件名的简单补全

```
# 以下shell函数将用于为
# “nuance_tune”命令生成补全项。
_nuance_tune_opts ()
{
  local curr_arg prev_arg
  curr_arg=${COMP_WORDS[COMP_CWORD]}
  prev_arg=${COMP_WORDS[COMP_CWORD-1]}

  # “config”选项需要一个文件参数，因此获取当前目录中的文件列表。# 这里使用 case 语句可
  能不是必需的，但为其他标志自定义参数留有余地。

  case "$prev_arg" in
    -config)
      COMPREPLY=( $( /bin/ls -1 ) )
      return 0
      ;;
    esac

    # 使用 compgen 为所有已知选项提供补全。
    COMPREPLY=( $(compgen -W '-analyze -experiment -generate_groups -compute_thresh -config -output
-help -usage -force -lang -grammar_overrides -begin_date -end_date -group -dataset -multiparses -
dump_records -no_index -confidencelevel -nrecs -dry_run -rec_scripts_only -save_temp -full_trc -
single_session -verbose -ep -unsupervised -write_manifest -remap -noreparse -upload -reference -
target -use_only_matching -histogram -stepsize' -- $curr_arg ) );
}

                                # -o 参数告诉 Bash 在适用时将补全作为文件名处理。


complete -o filenames -F _nuance_tune_opts nuance_tune
```


Chapter 29: Programmable completion

Section 29.1: Simple completion using function

```
_mycompletion() {
    local command_name="$1" # not used in this example
    local current_word="$2"
    local previous_word="$3" # not used in this example
    # COMPREPLY is an array which has to be filled with the possible completions
    # compgen is used to filter matching completions
    COMPREPLY=( $(compgen -W 'hello world' -- "$current_word") )
}
complete -F _mycompletion mycommand
```

Usage Example:

```
$ mycommand [TAB][TAB]
hello world
$ mycommand h[TAB][TAB]
$ mycommand hello
```

Section 29.2: Simple completion for options and filenames

```
# The following shell function will be used to generate completions for
# the "nuance_tune" command.
_nuance_tune_opts ()
{
  local curr_arg prev_arg
  curr_arg=${COMP_WORDS[COMP_CWORD]}
  prev_arg=${COMP_WORDS[COMP_CWORD-1]}

  # The "config" option takes a file arg, so get a list of the files in the
  # current dir.  A case statement is probably unnecessary here, but leaves
  # room to customize the parameters for other flags.
  case "$prev_arg" in
    -config)
      COMPREPLY=( $( /bin/ls -1 ) )
      return 0
      ;;
    esac

    # Use compgen to provide completions for all known options.
    COMPREPLY=( $(compgen -W '-analyze -experiment -generate_groups -compute_thresh -config -output
-help -usage -force -lang -grammar_overrides -begin_date -end_date -group -dataset -multiparses -
dump_records -no_index -confidencelevel -nrecs -dry_run -rec_scripts_only -save_temp -full_trc -
single_session -verbose -ep -unsupervised -write_manifest -remap -noreparse -upload -reference -
target -use_only_matching -histogram -stepsize' -- $curr_arg ) );
}

# The -o parameter tells Bash to process completions as filenames, where applicable.


complete -o filenames -F _nuance_tune_opts nuance_tune
```

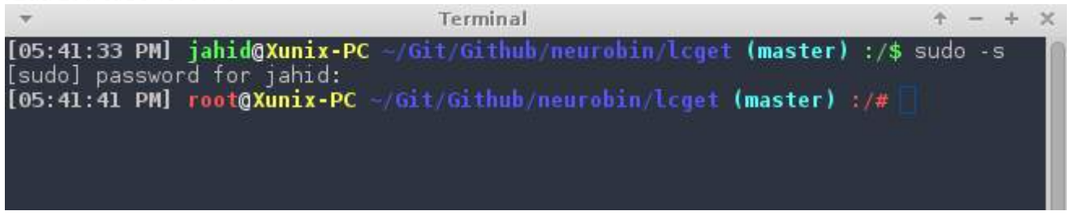
第30章：自定义 PS1

第30.1节：为终端提示符着色和自定义

这是作者设置个人PS1变量的方式：

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}
#如果你是Mac用户，请使用下面的函数
gitPS1ForMac(){
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
timeNow(){
    echo "$(date +%r)"
}
if [ "$color_prompt" = yes ]; then
  if [ x$EUID = x0 ]; then
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;31m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;31m\]:/#\[\033[00m\] '
  else
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;32m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;32m\]:/$\[\033[00m\] '
  fi
否则
    PS1='[$(timeNow)] \u@\h \w$(gitPS1) :/$ '
fi
```

这就是我的提示符的样子：



颜色参考：

```
# 颜色
txtblk='\e[0;30m' # 黑色 - 常规
txtred='\e[0;31m' # 红色
txtgrn='\e[0;32m' # 绿色
txtylw='\e[0;33m' # 黄色
txtblu='\e[0;34m' # 蓝色
txtpur='\e[0;35m' # 紫色
txtcyn='\e[0;36m' # 青色
txtwht='\e[0;37m' # 白色
bldblk='\e[1;30m' # 黑色 - 粗体
bldred='\e[1;31m' # 红色
bldgrn='\e[1;32m' # 绿色
bldylw='\e[1;33m' # 黄色
bldblu='\e[1;34m' # 蓝色
bldpur='\e[1;35m' # 紫色
bldcyn='\e[1;36m' # 青色
```

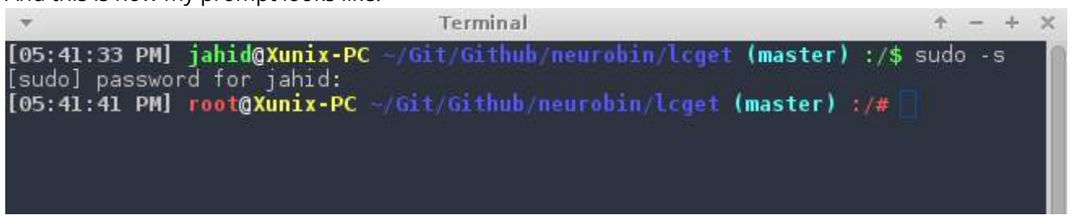

Chapter 30: Customizing PS1

Section 30.1: Colorize and customize terminal prompt

This is how the author sets their personal PS1 variable:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}
#Please use the below function if you are a mac user
gitPS1ForMac(){
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
timeNow(){
    echo "$(date +%r)"
}
if [ "$color_prompt" = yes ]; then
  if [ x$EUID = x0 ]; then
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;31m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;31m\]:/#\[\033[00m\] '
  else
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;32m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;32m\]:/$\[\033[00m\] '
  fi
else
    PS1='[$(timeNow)] \u@\h \w$(gitPS1) :/$ '
fi
```

And this is how my prompt looks like:



Color reference:

```
# Colors
txtblk='\e[0;30m' # Black - Regular
txtred='\e[0;31m' # Red
txtgrn='\e[0;32m' # Green
txtylw='\e[0;33m' # Yellow
txtblu='\e[0;34m' # Blue
txtpur='\e[0;35m' # Purple
txtcyn='\e[0;36m' # Cyan
txtwht='\e[0;37m' # White
bldblk='\e[1;30m' # Black - Bold
bldred='\e[1;31m' # Red
bldgrn='\e[1;32m' # Green
bldylw='\e[1;33m' # Yellow
bldblu='\e[1;34m' # Blue
bldpur='\e[1;35m' # Purple
bldcyn='\e[1;36m' # Cyan
```

```
bldwht='\e[1;37m' # 白色
unkblk='\e[4;30m' # 黑色 - 下划线
undred='\e[4;31m' # 红色
undgrn='\e[4;32m' # 绿色
undylw='\e[4;33m' # 黄色
undblu='\e[4;34m' # 蓝色
undpur='\e[4;35m' # 紫色
undcyn='\e[4;36m' # 青色
undwht='\e[4;37m' # 白色
bakblk='\e[40m'   # 黑色 - 背景
bakred='\e[41m'   # 红色
badgrn='\e[42m'   # 绿色
bakylw='\e[43m'   # 黄色
bakblu='\e[44m'   # 蓝色
bakpur='\e[45m'   # 紫色
bakcyn='\e[46m'   # 青色
bakwht='\e[47m'   # 白色
txtrst='\e[0m'    # 文本重置
```

注释：

- 在 ~/.bashrc 或 /etc/bashrc 或 ~/.bash_profile 或 ~./profile 文件中（取决于操作系统）进行更改并保存。
- 对于 `root` 用户，你可能还需要编辑 `/etc/bash.bashrc` 或 `/root/.bashrc` 文件
- 保存文件后运行 `source ~/.bashrc`（根据发行版不同）。
- 注意：如果你已在 ~/.bashrc 中保存了更改，记得在你的 ~/.bash_profile 中添加 source ~/.bashrc，这样每次终端应用启动时，PS1 的更改都会被加载。

第30.2节：在终端提示符中显示git分支名称

你可以在PS1变量中使用函数，只需确保用单引号括起来或对特殊字符进行转义：

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}

PS1='\u@\h:\w$(gitPS1)$ '
```

它会给你一个类似这样的提示符：

```
user@Host:/path (master)$
```

注释：

- 在 ~/.bashrc 或 /etc/bashrc 或 ~/.bash_profile 或 ~./profile 文件中（取决于操作系统）进行更改并保存。
- 保存文件后，运行 source ~/.bashrc（发行版特定）。

第30.3节：在终端提示符显示时间

```
timeNow(){
    echo "$(date +%r)"
}
```



```
bldwht='\e[1;37m' # White
unkblk='\e[4;30m' # Black - Underline
undred='\e[4;31m' # Red
undgrn='\e[4;32m' # Green
undylw='\e[4;33m' # Yellow
undblu='\e[4;34m' # Blue
undpur='\e[4;35m' # Purple
undcyn='\e[4;36m' # Cyan
undwht='\e[4;37m' # White
bakblk='\e[40m'   # Black - Background
bakred='\e[41m'   # Red
badgrn='\e[42m'   # Green
bakylw='\e[43m'   # Yellow
bakblu='\e[44m'   # Blue
bakpur='\e[45m'   # Purple
bakcyn='\e[46m'   # Cyan
bakwht='\e[47m'   # White
txtrst='\e[0m'    # Text Reset
```

Notes:

- Make the changes in `~/.bashrc` or `/etc/bashrc` or `~/.bash_profile` or `~./profile` file (depending on the OS) and save it.
- For `root` you might also need to edit the `/etc/bash.bashrc` or `/root/.bashrc` file
- Run `source ~/.bashrc` (distro specific) after saving the file.
- Note: if you have saved the changes in `~/.bashrc`, then remember to add `source ~/.bashrc` in your `~/.bash_profile` so that this change in PS1 will be recorded every time the Terminal application starts.

Section 30.2: Show git branch name in terminal prompt

You can have functions in the PS1 variable, just make sure to single quote it or use escape for special chars:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}

PS1='\u@\h:\w$(gitPS1)$ '
```

It will give you a prompt like this:

```
user@Host:/path (master)$
```

Notes:

- Make the changes in `~/.bashrc` or `/etc/bashrc` or `~/.bash_profile` or `~./profile` file (depending on the OS) and save it.
- Run `source ~/.bashrc` (distro specific) after saving the file.

Section 30.3: Show time in terminal prompt

```
timeNow(){
    echo "$(date +%r)"
}
```

```
PS1='[$(timeNow)] \u@\h:\w$ '
```

它会给你一个类似这样的提示符：

```
[05:34:37 PM] user@Host:/path$
```

注释：

- 在 ~/.bashrc 或 /etc/bashrc 或 ~/.bash_profile 或 ~./profile 文件中（取决于操作系统）进行更改并保存。
- 保存文件后，运行 source ~/.bashrc（发行版特定）。

第30.4节：使用PROMPT_COMMAND显示git分支

如果你在一个git仓库的文件夹内，显示当前所在的分支可能会很方便。在~/.bashrc或/etc/bashrc中添加以下内容（需要安装git才能使用）：

```
function prompt_command {
    # 检查我们是否在git仓库内
    if git status > /dev/null 2>&1; then
        # 只获取分支名称
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)
    else
        export GIT_STATUS=""
    fi
}
# 每次显示 PS1 时都会调用此函数
PROMPT_COMMAND=prompt_command

PS1="\$GIT_STATUS \u@\h:\w\$ "
```

如果我们在 git 仓库中的某个文件夹内，这将输出：

> branch user@machine:~$

如果我们在普通文件夹内：

> user@machine:~$

第30.5节：更改 PS1 提示符

要更改 PS1，只需更改 PS1 shell 变量的值。该值可以设置在 ~/.bashrc 或 /etc/bashrc 文件中，具体取决于发行版。PS1 可以更改为任何纯文本，例如：

```
PS1="hello "
```

除了纯文本外，还支持多种反斜杠转义的特殊字符：

格式	操作
\a	一个 ASCII 响铃字符（07）
\d	日期，格式为“星期 月 日”（例如，“周二 5月 26日”）


```
PS1='[$(timeNow)] \u@\h:\w$ '
```

It will give you a prompt like this:

```
[05:34:37 PM] user@Host:/path$
```

Notes:

- Make the changes in ~/.bashrc or /etc/bashrc or ~/.bash_profile or ~./profile file (depending on the OS) and save it.
- Run source ~/.bashrc (distro specific) after saving the file.

Section 30.4: Show a git branch using PROMPT_COMMAND

If you are inside a folder of a git repository it might be nice to show the current branch you are on. In ~/.bashrc or /etc/bashrc add the following (git is required for this to work):

```
function prompt_command {
    # Check if we are inside a git repository
    if git status > /dev/null 2>&1; then
        # Only get the name of the branch
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)
    else
        export GIT_STATUS=""
    fi
}
# This function gets called every time PS1 is shown
PROMPT_COMMAND=prompt_command

PS1="\$GIT_STATUS \u@\h:\w\$ "
```

If we are in a folder inside a git repository this will output:

> branch user@machine:~$

And if we are inside a normal folder:

> user@machine:~$

Section 30.5: Change PS1 prompt

To change PS1, you just have to change the value of PS1 shell variable. The value can be set in ~/.bashrc or /etc/bashrc file, depending on the distro. PS1 can be changed to any plain text like:

```
PS1="hello "
```

Besides the plain text, a number of backslash-escaped special characters are supported:

Format	Action
\a	an ASCII bell character (07)
\d	the date in “Weekday Month Date” format (e.g., “Tue May 26”)

\D {格式}	格式参数传递给 strftime(3)，结果插入到提示字符串中；空格式会生成特定于区域设置的时间表示。大括号是必需的
\e	一个 ASCII 转义字符 (033)
\h	主机名直到第一个‘.’
\H	主机名
\j	当前由 shell 管理的作业数量
\l	shell 终端设备名称的基本名
	换行符
\r	回车符
\s	shell的名称，$0的基本名（最后一个斜杠之后的部分）
	当前时间，24小时制 HH:MM:SS 格式
\T	当前时间，12小时制 HH:MM:SS 格式
\@	当前时间，12小时制 am/pm 格式
\A	当前时间，24小时制 HH:MM 格式
\u	当前用户的用户名
\v	bash 的版本（例如，2.00）
\V	bash 的发行版本，版本号加补丁级别（例如，2.00.0）
\w	当前工作目录，$HOME 用波浪号表示
\W	当前工作目录的基本名称，$HOME 用波浪号缩写
\!	该命令的历史编号
\#	该命令的命令编号
\$	如果有效用户ID为0，则为#，否则为$
nn*	对应八进制数字 nnn 的字符
\	反斜杠
\[	开始一段不可打印字符序列，该序列可用于在提示符中嵌入终端控制序列
\]	结束一段不可打印字符序列

例如，我们可以将 PS1 设置为：

```
PS1="\u@\h:\w\$ "
```

它将输出：

> user@machine:~$

第30.6节：显示上一个命令的返回状态和时间

有时我们需要一个视觉提示来指示上一个命令的返回状态。以下代码片段将其放置在 PS1 的开头。

请注意，每次生成新的 PS1 时都应调用 __stat() 函数，否则它将保持为 .bashrc 或 .bash_profile 中最后一个命令的返回状态。

```
# -ANSI-颜色代码- #
Color_Off="\033[0m"
###-常规-###
Red="\033[0;31m"
Green="\033[0;32m"
```


\D{format}	the format is passed to strftime(3) and the result is inserted into the prompt string; an empty format results in a locale-specific time representation. The braces are required
\e	an ASCII escape character (033)
\h	the hostname up to the first ‘.’
\H	the hostname
\j	the number of jobs currently managed by the shell
\l	the basename of the shell’s terminal device name
\n	newline
\r	carriage return
\s	the name of the shell, the basename of $0 (the portion following the final slash)
\t	the current time in 24-hour HH:MM:SS format
\T	the current time in 12-hour HH:MM:SS format
\@	the current time in 12-hour am/pm format
\A	the current time in 24-hour HH:MM format
\u	the username of the current user
\v	the version of bash (e.g., 2.00)
\V	the release of bash, version + patch level (e.g., 2.00.0)
\w	the current working directory, with $HOME abbreviated with a tilde
\W	the basename of the current working directory, with $HOME abbreviated with a tilde
\!	the history number of this command
\#	the command number of this command
\$	if the effective UID is 0, a #, otherwise a $
\nnn*	the character corresponding to the octal number nnn
\	a backslash
\[	begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
\]	end a sequence of non-printing characters

So for example, we can set PS1 to:

```
PS1="\u@\h:\w\$ "
```

And it will output:

> user@machine:~$

Section 30.6: Show previous command return status and time

Sometimes we need a visual hint to indicate the return status of previous command. The following snippet make put it at the head of the PS1.

Note that the __stat() function should be called every time a new PS1 is generated, or else it would stick to the return status of last command of your .bashrc or .bash_profile.

```
# -ANSI-COLOR-CODES- #
Color_Off="\033[0m"
###-Regular-###
Red="\033[0;31m"
Green="\033[0;32m"
```

```
Yellow="\033[0;33m"
####-加粗-####

function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✓ $Color_Off "
    else
        echo -en "$Red ✘ $Color_Off "
    fi
}

PS1='$(__stat)'
PS1+="[] "
PS1+="\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m $ "export

PS1
```


```
Yellow="\033[0;33m"
####-Bold-####

function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✓ $Color_Off "
    else
        echo -en "$Red ✘ $Color_Off "
    fi
}

PS1='$(__stat)'
PS1+="[\t] "
PS1+="\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m \n$ "

export PS1
```

第31章：大括号扩展

第31.1节：修改文件扩展名

```
$ mv filename.{jar,zip}
```

这将扩展为 `mv filename.jar filename.zip` 。

第31.2节：创建按月份和年份分组的目录

```
$ mkdir 20{09..11}-{01..12}
```

输入ls命令将显示已创建以下目录：

```
2009-01 2009-04 2009-07 2009-10 2010-01 2010-04 2010-07 2010-10 2011-01 2011-04 2011-07 2011-10
2009-02 2009-05 2009-08 2009-11 2010-02 2010-05 2010-08 2010-11 2011-02 2011-05 2011-08 2011-11
2009-03 2009-06 2009-09 2009-12 2010-03 2010-06 2010-09 2010-12 2011-03 2011-06 2011-09 2011-12
```

在示例中，在9前面加一个0可以确保数字前面填充一个0。你也可以用多个零来填充数字，例如：

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

第31.3节：创建dotfiles的备份

```
$ cp .vimrc{,.bak}
```

这会展开成命令 `cp .vimrc .vimrc.bak`。

第31.4节：使用递增

```
$ echo {0..10..2}
0 2 4 6 8 10
```

第三个参数用于指定增量，即 `{start..end..increment}`

使用增量不限于数字

```
$ for c in {a..z..5}; do echo -n $c; done
afkpuz
```

第31.5节：使用大括号扩展创建列表

Bash 可以轻松地从字母数字字符创建列表。

```
# 从 a 到 z 的列表
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# 从 z 到 a 反向
$ echo {z..a}
```


Chapter 31: Brace Expansion

Section 31.1: Modifying filename extension

```
$ mv filename.{jar,zip}
```

This expands into `mv filename.jar filename.zip`.

Section 31.2: Create directories to group files by month and year

```
$ mkdir 20{09..11}-{01..12}
```

Entering the `ls` command will show that the following directories were created:

```
2009-01 2009-04 2009-07 2009-10 2010-01 2010-04 2010-07 2010-10 2011-01 2011-04 2011-07 2011-10
2009-02 2009-05 2009-08 2009-11 2010-02 2010-05 2010-08 2010-11 2011-02 2011-05 2011-08 2011-11
2009-03 2009-06 2009-09 2009-12 2010-03 2010-06 2010-09 2010-12 2011-03 2011-06 2011-09 2011-12
```

Putting a `0` in front of `9` in the example ensures the numbers are padded with a single `0`. You can also pad numbers with multiple zeros, for example:

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

Section 31.3: Create a backup of dotfiles

```
$ cp .vimrc{,.bak}
```

This expands into the command `cp .vimrc .vimrc.bak`.

Section 31.4: Use increments

```
$ echo {0..10..2}
0 2 4 6 8 10
```

A third parameter to specify an increment, i.e. `{start..end..increment}`

Using increments is not constrained to just numbers

```
$ for c in {a..z..5}; do echo -n $c; done
afkpuz
```

Section 31.5: Using brace expansion to create lists

Bash can easily create lists from alphanumeric characters.

```
# list from a to z
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# reverse from z to a
$ echo {z..a}
```

```
z y x w v u t s r q p o n m l k j i h g f e d c b a

# 位数
$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# 带前导零
$ echo {01..20}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

# 逆序数字
$ echo {20..1}
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

# 带前导零的逆序
$ echo {20..01}
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01

# 组合多个大括号
$ echo {a..d}{1..3}
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

大括号扩展是最先进行的扩展，因此不能与其他扩展结合使用。

只能使用字符和数字。

这将不起作用：`echo {$(date +$H)..24}`

第31.6节：创建带子目录的多个目录

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

这将创建一个名为oplevel的顶层文件夹，在oplevel内创建九个文件夹，分别命名为`sublevel_01`、`sublevel_02`、等等。然后在这些子级目录内创建`child1`、`child2`、`child3`文件夹，结构如下：

```
顶层/子层级_01/子项1
顶层/子层级_01/子项2
顶层/子层级_01/子项3
顶层/子层级_02/子项1
```

等等。我发现这对于使用一个bash命令为我的特定用途创建多个文件夹和子文件夹非常有用。替换变量有助于自动化/解析传递给脚本的信息。


```
z y x w v u t s r q p o n m l k j i h g f e d c b a

# digits
$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# with leading zeros
$ echo {01..20}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

# reverse digit
$ echo {20..1}
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

# reversed with leading zeros
$ echo {20..01}
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01

# combining multiple braces
$ echo {a..d}{1..3}
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

Brace expansion is the very first expansion that takes place, so it cannot be combined with any other expansions.

Only chars and digits can be used.

This won't work: `echo {$(date +$H)..24}`

Section 31.6: Make Multiple Directories with Sub-Directories

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

This will create a top level folder called `toplevel`, nine folders inside of `toplevel` named `sublevel_01`, `sublevel_02`, etc. Then inside of those sublevels: `child1`, `child2`, `child3` folders, giving you:

```
toplevel/sublevel_01/child1
toplevel/sublevel_01/child2
toplevel/sublevel_01/child3
toplevel/sublevel_02/child1
```

and so on. I find this very useful for creating multiple folders and sub folders for my specific purposes, with one bash command. Substitute variables to help automate/parse information given to the script.

第32章：getopts：智能位置参数解析

参数	详细信息
选项字符串	要识别的选项字符
名称	存储解析选项的位置名称

第32.1节：pingnmap

```
#!/bin/bash
# 脚本名称：pingnmap
# 场景：X公司的系统管理员厌倦了单调的ping和nmap操作，
# 于是他决定用脚本简化这项工作。
# 他希望完成的任务是
# 1. 对给定的IP地址/域名进行最多5次的Ping操作。和/或
# 2. 检查给定IP地址/域名的特定端口是否开放。
# 而getopts是她的救援工具。
# 选项的简要概述
# n ：用于nmap
# t ：用于ping
# i ：输入IP地址的选项
# p ：输入端口的选项
# v ：获取脚本版本的选项


while getopts ':nti:p:v' opt
# 在开头加:可以抑制无效选项的错误
do
case "$opt" in
   'i')ip="${OPTARG}"
       ;;
   'p')port="${OPTARG}"
       ;;
   'n')nmap_yes=1;
       ;;
   't')ping_yes=1;
       ;;
   'v')echo "pingnmap version 1.0.0"
       ;;
    *) echo "无效选项 $opt"
       echo "用法："
       echo "pingmap -[n|t[i|p]|v]"
       ;;
esac
done
if  [ ! -z "$nmap_yes" ] && [ "$nmap_yes" -eq "1" ]
then
   if [ ! -z "$ip" ] && [ ! -z "$port" ]
   then
     nmap -p "$port" "$ip"
   fi
fi

if  [ ! -z "$ping_yes" ] && [ "$ping_yes" -eq "1" ]
then
   if [ ! -z "$ip" ]
   then
     ping -c 5 "$ip"
```


Chapter 32: getopts : smart positional-parameter parsing

Parameter	Detail
optstring	The option characters to be recognized
name	Then name where parsed option is stored

Section 32.1: pingnmap

```
#!/bin/bash
# Script name : pingnmap
# Scenario : The systems admin in company X is tired of the monotonous job
# of pinging and nmapping, so he decided to simplify the job using a script.
# The tasks he wish to achieve is
# 1. Ping - with a max count of 5 -the given IP address/domain. AND/OR
# 2. Check if a particular port is open with a given IP address/domain.
# And getopts is for her rescue.
# A brief overview of the options
# n : meant for nmap
# t : meant for ping
# i : The option to enter the IP address
# p : The option to enter the port
# v : The option to get the script version


while getopts ':nti:p:v' opt
#putting : in the beginnnig suppresses the errors for invalid options
do
case "$opt" in
   'i')ip="${OPTARG}"
       ;;
   'p')port="${OPTARG}"
       ;;
   'n')nmap_yes=1;
       ;;
   't')ping_yes=1;
       ;;
   'v')echo "pingnmap version 1.0.0"
       ;;
    *) echo "Invalid option $opt"
       echo "Usage : "
       echo "pingmap -[n|t[i|p]|v]"
       ;;
esac
done
if  [ ! -z "$nmap_yes" ] && [ "$nmap_yes" -eq "1" ]
then
   if [ ! -z "$ip" ] && [ ! -z "$port" ]
   then
     nmap -p "$port" "$ip"
   fi
fi

if  [ ! -z "$ping_yes" ] && [ "$ping_yes" -eq "1" ]
then
   if [ ! -z "$ip" ]
   then
     ping -c 5 "$ip"
```

```
   fi
fi
shift $(( OPTIND - 1 )) # 处理额外参数
if [ ! -z "$@" ]
then
  echo "末尾存在无效参数：$@"
fi
```

输出

```
$ ./pingnmap -nt -i google.com -p 80

开始 Nmap 6.40 ( http://nmap.org ) 于 2016-07-23 14:31 IST
google.com (216.58.197.78) 的 Nmap 扫描报告
主机在线 (延迟 0.034秒)。
216.58.197.78 的 rDNS 记录：maa03s21-in-f14.1e100.net
端口    状态 服务
80/tcp 开放  http

Nmap 完成：扫描了 1 个 IP 地址（1 个主机在线），耗时 0.22 秒
PING google.com (216.58.197.78) 56(84) 字节的数据。
来自 maa03s21-in-f14.1e100.net (216.58.197.78) 的 64 字节数据：icmp_seq=1 ttl=57 时间=29.3 毫秒
来自 maa03s21-in-f14.1e100.net (216.58.197.78) 的 64 字节数据：icmp_seq=2 ttl=57 时间=30.9 毫秒
来自 maa03s21-in-f14.1e100.net (216.58.197.78) 的 64 字节数据：icmp_seq=3 ttl=57 时间=34.7 毫秒
来自 maa03s21-in-f14.1e100.net (216.58.197.78) 的 64 字节数据：icmp_seq=4 ttl=57 时间=39.6 毫秒
来自 maa03s21-in-f14.1e100.net (216.58.197.78) 的 64 字节数据：icmp_seq=5 ttl=57 时间=32.7 毫秒

--- google.com ping 统计信息 ---
5 个数据包已发送，5 个已接收，丢包率 0%，耗时 4007 毫秒
rtt 最小/平均/最大/偏差 = 29.342/33.481/39.631/3.576 毫秒
$ ./pingnmap -v
pingnmap 版本 1.0.0
$ ./pingnmap -h
无效选项？
用法：
pingmap -[n|t[i|p]|v]
$ ./pingnmap -v
pingnmap 版本 1.0.0
$ ./pingnmap -h
无效选项？
用法：
pingmap -[n|t[i|p]|v]
```



```
   fi
fi
shift $(( OPTIND - 1 )) # Processing additional arguments
if [ ! -z "$@" ]
then
  echo "Bogus arguments at the end : $@"
fi
```

Output

```
$ ./pingnmap -nt -i google.com -p 80

Starting Nmap 6.40 ( http://nmap.org ) at 2016-07-23 14:31 IST
Nmap scan report for google.com (216.58.197.78)
Host is up (0.034s latency).
rDNS record for 216.58.197.78: maa03s21-in-f14.1e100.net
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
PING google.com (216.58.197.78) 56(84) bytes of data.
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=1 ttl=57 time=29.3 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=2 ttl=57 time=30.9 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=3 ttl=57 time=34.7 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=4 ttl=57 time=39.6 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=5 ttl=57 time=32.7 ms

--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 29.342/33.481/39.631/3.576 ms
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
```

第33章：调试

第33.1节：使用“-n”检查脚本语法

“-n”标志允许您检查脚本的语法，而无需执行它：

```
~> $ bash -n testscript.sh
testscript.sh: 第128行：在寻找匹配的`"'时遇到意外的文件结尾（EOF）
testscript.sh: 第130行：语法错误：意外的文件结尾
```

第33.2节：使用bashdb调试

Bashdb是一个类似于gdb的工具，你可以设置断点（在某行或某函数处）、打印变量内容、重新启动脚本执行等。

你通常可以通过包管理器安装，例如在Fedora上：

```
sudo dnf install bashdb
```

或者从主页获取。然后你可以用脚本作为参数运行它：

```
bashdb <你的脚本>
```

以下是一些入门命令：

```
l - 显示本地代码行，再次按l向下滚动
s - 单步执行到下一行
print $VAR - 输出变量内容
restart - 重新运行bash脚本，执行前会重新加载脚本。
eval - 评估一些自定义命令，例如：eval echo hi

b  在某行设置断点
c - 继续执行直到某个断点
i b - 查看断点信息
d  - 删除指定行号的断点

shell - 在执行过程中启动一个子shell，方便操作变量
```

更多信息，建议查阅手册：
http://www.rodericksmith.plus.com/outlines/manuals/bashdbOutline.html

另见主页：
http://bashdb.sourceforge.net/

第33.3节：使用“-x”调试bash脚本

使用“-x”启用执行行的调试输出。它可以用于整个会话或脚本，也可以在脚本中以编程方式启用。

运行启用调试输出的脚本：

```
$ bash -x myscript.sh
```


Chapter 33: Debugging

Section 33.1: Checking the syntax of a script with "-n"

The -n flag enables you to check the syntax of a script without having to execute it:

```
~> $ bash -n testscript.sh
testscript.sh: line 128: unexpected EOF while looking for matching `"'
testscript.sh: line 130: syntax error: unexpected end of file
```

Section 33.2: Debugging using bashdb

Bashdb is a utility that is similar to gdb, in that you can do things like set breakpoints at a line or at a function, print content of variables, you can restart script execution and more.

You can normally install it via your package manager, for example on Fedora:

```
sudo dnf install bashdb
```

Or get it from the homepage. Then you can run it with your script as a paramater:

```
bashdb <YOUR SCRIPT>
```

Here are a few commands to get you started:

```
l - show local lines, press l again to scroll down
s - step to next line
print $VAR - echo out content of variable
restart - reruns bashscript, it re-loads it prior to execution.
eval - evaluate some custom command, ex: eval echo hi

b  set breakpoint on some line
c - continue till some breakpoint
i b - info on break points
d  - delete breakpoint at line #

shell - launch a sub-shell in the middle of execution, this is handy for manipulating variables
```

For more information, I recommend consulting the manual:
http://www.rodericksmith.plus.com/outlines/manuals/bashdbOutline.html

See also homepage:
http://bashdb.sourceforge.net/

Section 33.3: Debugging a bash script with "-x"

Use "-x" to enable debug output of executed lines. It can be run on an entire session or script, or enabled programmatically within a script.

Run a script with debug output enabled:

```
$ bash -x myscript.sh
```

或者

```
$ bash --debug myscript.sh
```

在 bash 脚本中开启调试。调试也可以选择性地重新开启，尽管脚本退出时调试输出会自动重置。

```
#!/bin/bash
set -x    # 启用调试
# 这里是一些代码
set +x    # 关闭调试输出。
```


Or

```
$ bash --debug myscript.sh
```

Turn on debugging within a bash script. It may optionally be turned back on, though debug output is automatically reset when the script exits.

```
#!/bin/bash
set -x    # Enable debugging
# some code here
set +x    # Disable debugging output.
```

第34章：模式匹配和正则表达式

第34.1节：从字符串的正则匹配中获取捕获组

```
a='我是一串包含数字1234的简单字符串'
pat='(.*) ([0-9]+)'
[[ "$a" =~ $pat ]]
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

输出：

```
我是一串简单的字符串，包含数字1234
我是一串简单的字符串，包含数字
1234
```

第34.2节：当通配符没有匹配任何内容时的行为

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

如果通配符没有匹配任何内容，结果由选项 nullglob 和 failglob 决定。如果两者都未设置，Bash将在没有匹配时返回通配符本身

```
$ echo no*match
no*match
```

如果激活了nullglob，则不会返回任何内容（null）：

```
$ shopt -s nullglob
$ echo no*match

$
```

如果激活了failglob，则会返回错误信息：

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```


Chapter 34: Pattern matching and regular expressions

Section 34.1: Get captured groups from a regex match against a string

```
a='I am a simple string with digits 1234'
pat='(.*) ([0-9]+)'
[[ "$a" =~ $pat ]]
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

Output:

```
I am a simple string with digits 1234
I am a simple string with digits
1234
```

Section 34.2: Behaviour when a glob does not match anything

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

In case the glob does not match anything the result is determined by the options `nullglob` and `failglob`. If neither of them are set, Bash will return the glob itself if nothing is matched

```
$ echo no*match
no*match
```

If `nullglob` is activated then nothing (`null`) is returned:

```
$ shopt -s nullglob
$ echo no*match

$
```

If `failglob` is activated then an error message is returned:

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```

注意，failglob选项优先于nullglob选项，即如果同时设置了nullglob和failglob，则在没有匹配的情况下会返回错误。

第34.3节：检查字符串是否匹配正则表达式

版本 ≥ 3.0

检查字符串是否正好由8位数字组成：

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

第34.4节：正则表达式匹配

```
pat='[^0-9]+([0-9]+)'
s='我是一串包含数字的字符串 1024'
[[ $s =~ $pat ]] # $pat 必须不加引号
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
```

输出：

```
我是一串包含数字的字符串 1024
1024
```

除了将正则表达式赋值给变量（$pat），我们也可以这样写：

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

说明

- 构造 `[[ $s =~ $pat ]]` 用于执行正则表达式匹配
- 捕获的分组，即匹配结果，存储在名为 BASH_REMATCH 的数组中
- 数组 *BASH_REMATCH* 的第0个索引是完整匹配内容
- BASH_REMATCH 数组中的第 i 个索引是第 i 个捕获组，其中 i = 1, 2, 3 ...

第34.5节：* 通配符

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```


Notice, that the `failglob` option supersedes the `nullglob` option, i.e., if `nullglob` and `failglob` are both set, then - in case of no match - an error is returned.

Section 34.3: Check if a string matches a regular expression

Version ≥ 3.0

Check if a string consists in exactly 8 digits:

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

Section 34.4: Regex matching

```
pat='[^0-9]+([0-9]+)'
s='I am a string with some digits 1024'
[[ $s =~ $pat ]] # $pat must be unquoted
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
```

Output:

```
I am a string with some digits 1024
1024
```

Instead of assigning the regex to a variable (`$pat`) we could also do:

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

Explanation

- The `[[ $s =~ $pat ]]` construct performs the regex matching
- The captured groups i.e the match results are available in an array named *BASH_REMATCH*
- The 0th index in the *BASH_REMATCH* array is the total match
- The *i'th* index in the *BASH_REMATCH* array is the *i'th* captured group, where *i* = 1, 2, 3 ...

Section 34.5: The * glob

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

星号 * 可能是最常用的通配符。它简单地匹配任意字符串

```
$ echo *acy
macy stacy tracy
```

单个 * 不会匹配位于子文件夹中的文件和文件夹

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

第34.6节：** 通配符

版本 ≥ 4.0

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -s globstar
```

Bash能够将两个相邻的星号解释为一个通配符。在启用globstar选项后，这可以用来匹配目录结构中更深层的文件夹

```
echo **
emptyfolder 文件夹 文件夹/anotherfolder 文件夹/anotherfolder/内容
文件夹/anotherfolder/内容/deepfolder 文件夹/anotherfolder/内容/deepfolder/文件
文件夹/subfolder 文件夹/subfolder/内容 文件夹/subfolder/内容/deepfolder
文件夹/subfolder/内容/deepfolder/文件 macy stacy tracy
```

“**”可以被视为路径展开，无论路径有多深。此示例匹配任何以 deep 开头的文件或文件夹，无论其嵌套多深：

```
$ echo **/deep*
文件夹/anotherfolder/内容/deepfolder 文件夹/subfolder/内容/deepfolder
```

第34.7节：? 通配符

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
```


The asterisk * is probably the most commonly used glob. It simply matches any String

```
$ echo *acy
macy stacy tracy
```

A single * will not match files and folders that reside in subfolders

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

Section 34.6: The ** glob

Version ≥ 4.0

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -s globstar
```

Bash is able to interpret two adjacent asterisks as a single glob. With the `globstar` option activated this can be used to match folders that reside deeper in the directory structure

```
echo **
emptyfolder folder folder/anotherfolder folder/anotherfolder/content
folder/anotherfolder/content/deepfolder folder/anotherfolder/content/deepfolder/file
folder/subfolder folder/subfolder/content folder/subfolder/content/deepfolder
folder/subfolder/content/deepfolder/file macy stacy tracy
```

The ** can be thought of a path expansion, no matter how deep the path is. This example matches any file or folder that starts with `deep`, regardless of how deep it is nested:

```
$ echo **/deep*
folder/anotherfolder/content/deepfolder folder/subfolder/content/deepfolder
```

Section 34.7: The ? glob

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
```

```
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

“?”仅匹配恰好一个字符

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

第34.8节：[ ] 通配符

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

如果需要匹配特定字符，则可以使用 '[]'。'[]' 中的任何字符都会被精确匹配一次。

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

然而，[] 通配符比这更灵活。它还允许负向匹配，甚至匹配字符范围和字符类。负向匹配是通过在 [后面使用 ！或 ^ 作为第一个字符来实现的。

我们可以通过以下方式匹配 stacy

```
$ echo [!t][^r]acy
stacy
```

这里我们告诉 bash，我们只想匹配那些不以 t 开头，第二个字母不是 r，且以 acy 结尾的文件。

范围可以通过用连字符（-）分隔一对字符来匹配。任何位于这两个括号字符之间（包括两端）的字符都会被匹配。例如，[r-t] 等同于 [rst]

```
$ echo [r-t][r-t]acy
stacy tracy
```

字符类可以通过 [:class:] 来匹配，例如，为了匹配包含空白字符的文件

```
$ echo *[[:blank:]]*
file with space
```



```
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

The ? simply matches exactly one character

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

Section 34.8: The [ ] glob

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

If there is a need to match specific characters then '[]' can be used. Any character inside '[]' will be matched exactly once.

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

The [ ] glob, however, is more versatile than just that. It also allows for a negative match and even matching ranges of characters and character classes. A negative match is achieved by using `!` or `^` as the first character following `[`. We can match `stacy` by

```
$ echo [!t][^r]acy
stacy
```

Here we are telling bash the we want to match only files which do not not start with a `t` and the second letter is not an `r` and the file ends in `acy`.

Ranges can be matched by seperating a pair of characters with a hyphen (-). Any character that falls between those two enclosing characters - inclusive - will be matched. E.g., `[r-t]` is equivalent to `[rst]`

```
$ echo [r-t][r-t]acy
stacy tracy
```

Character classes can be matched by `[:class:]`, e.g., in order to match files that contain a whitespace

```
$ echo *[[:blank:]]*
file with space
```

第34.9节：匹配隐藏文件

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Bash 内置选项 dotglob 允许匹配隐藏文件和文件夹，即以 . 开头的文件和文件夹。

```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

第34.10节：大小写不敏感匹配

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

设置选项 nocaseglob 将使通配符匹配时忽略大小写

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

第34.11节：扩展通配符

版本 ≥ 2.02

准备

```
$ mkdir 通配
$ cd 通配
$ mkdir -p 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/
touch macy stacy tracy "带空格的文件" 文件夹/{子文件夹,另一个}文件夹/内容/深层文件夹/文件
.隐藏文件
$ shopt -u nullglob
```


Section 34.9: Matching hidden files

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

The Bash built-in option *dotglob* allows to match hidden files and folders, i.e., files and folders that start with a `.`

```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

Section 34.10: Case insensitive matching

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Setting the option `nocaseglob` will match the glob in a case insensitive manner

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

Section 34.11: Extended globbing

`Version ≥ 2.02`

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
```

```
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Bash内置的extglob选项可以扩展通配符的匹配能力

```
shopt -s extglob
```

以下子模式构成有效的扩展通配符：

- `?(pattern-list)` – 匹配给定模式的零次或一次出现
- `*(pattern-list)` – 匹配给定模式的零次或多次出现
- `+(pattern-list)` – 匹配一个或多个给定模式的出现
- `@(pattern-list)` – 匹配给定模式中的任意一个
- !(pattern-list) – 匹配除给定模式之外的任何内容

pattern-list 是由 | 分隔的通配符列表。

```
$ echo *([r-t])acy
stacy tracy

$ echo *([r-t]|m)acy
macy stacy tracy

$ echo ?([a-z])acy
macy
```

pattern-list 本身可以是另一个嵌套的扩展通配符。在上面的例子中，我们已经看到可以用 *(r-t) 匹配 tracy 和 stacy。这个扩展通配符本身可以用在取反扩展通配符!(pattern-list) 中，以匹配 macy

```
$ echo !(*([r-t]))acy
macy
```

它匹配任何不以零个或多个字母 r、s 和 t 开头的内容，这样只有 macy 可能匹配。


```
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Bash's built-in `extglob` option can extend a glob's matching capabilities

```
shopt -s extglob
```

The following sub-patterns comprise valid extended globs:

- `?(pattern-list)` – Matches zero or one occurrence of the given patterns
- `*(pattern-list)` – Matches zero or more occurrences of the given patterns
- `+(pattern-list)` – Matches one or more occurrences of the given patterns
- `@(pattern-list)` – Matches one of the given patterns
- `!(pattern-list)` – Matches anything except one of the given patterns

The `pattern-list` is a list of globs separated by `|`.

```
$ echo *([r-t])acy
stacy tracy

$ echo *([r-t]|m)acy
macy stacy tracy

$ echo ?([a-z])acy
macy
```

The `pattern-list` itself can be another, nested extended glob. In the above example we have seen that we can match `tracy` and `stacy` with `*(r-t)`. This extended glob itself can be used inside the negated extended glob `!(pattern-list)` in order to match `macy`

```
$ echo !(*([r-t]))acy
macy
```

It matches anything that does **not** start with zero or more occurrences of the letters `r`, `s` and `t`, which leaves only `macy` as possible match.

第35章：更改Shell

第35.1节：查找当前Shell

有几种方法可以确定当前的 shell

```
echo $0
ps -p $$
echo $SHELL
```

第35.2节：列出可用的 shell

列出可用的登录 shell：

```
cat /etc/shells
```

示例：

```
$ cat /etc/shells
# /etc/shells: 有效的登录 shell
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

第35.3节：更改 shell

要更改当前的 bash，请运行以下命令

```
export SHELL=/bin/bash
exec /bin/bash
```

要更改启动时打开的bash，编辑.profile并添加这些行

Chapter 35: Change shell

Section 35.1: Find the current shell

There are a few ways to determine the current shell

```
echo $0
ps -p $$
echo $SHELL
```

Section 35.2: List available shells

To list available login shells :

```
cat /etc/shells
```

Example:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Section 35.3: Change the shell

To change the current bash run these commands

```
export SHELL=/bin/bash
exec /bin/bash
```

to change the bash that opens on startup edit `.profile` and add those lines

第36章：内部变量

Bash内部变量的概述，何时、何地以及如何使用它们。

第36.1节：Bash内部变量一览

变量	详细信息
$* / $@	函数/脚本的位置参数（参数）。展开如下： $* 和 $@ 与 $1 $2 ... 相同（注意通常不加引号使用它们没有意义） "$*" 与 "$1 $2 ..." 相同， "$@" 与 "$1" "$2" ... 相同 1. 参数由 $IFS 的第一个字符分隔，该字符不必是空格。
$#	传递给脚本或函数的位置参数数量
$!	最近放入后台的作业中最后一个（管道中最右边的）命令的进程ID（注意，当启用作业控制时，这不一定与作业的进程组ID相同）
$$	执行bash的进程ID
$?	最后一个命令的退出状态
$n	位置参数，其中 n=1, 2, 3, ..., 9
${n}	位置参数（同上），但 n 可以大于 9
$0	在脚本中，脚本被调用时的路径；使用bash-c'printf "%s" "$0"' nameargs': name（内联脚本后的第一个参数），否则为argv[0]，即bash接收到的。
$_	上一个命令的最后一个字段
$IFS	内部字段分隔符
$PATH	用于查找可执行文件的PATH环境变量
$OLDPWD	上一个工作目录
$PWD	当前工作目录
$FUNCNAME	执行调用栈中的函数名数组
$BASH_SOURCE	包含FUNCNAME数组中元素的源路径的数组。可用于获取脚本路径。
$BASH_ALIASES	包含所有当前定义别名的关联数组
$BASH_REMATCH	上一次正则匹配的匹配结果数组
$BASH_VERSION	Bash版本字符串
$BASH_VERSINFO	包含6个元素的Bash版本信息数组
$BASH	当前执行的Bash shell本身的绝对路径（通过bash基于argv[0]和$PATH的值启发式确定；在某些极端情况下可能不准确）
$BASH_SUBSHELL	Bash子shell级别
$UID	运行bash进程的真实（如果不同则无效）用户ID
$PS1	主命令行提示符；参见使用 PS* 变量
$PS2	次级命令行提示符（用于额外输入）
$PS3	三级命令行提示符（用于 select 循环）
$PS4	四级命令行提示符（用于在详细输出中附加信息）
$RANDOM	介于 0 到 32767 之间的伪随机整数
$REPLY	默认情况下由read使用的变量，当未指定变量时使用。也被SELECT用来返回用户提供的值
$PIPESTATUS	数组变量，保存最近执行的前台管道中每个命令的退出状态值。

Chapter 36: Internal variables

An overview of Bash's internal variables, where, how, and when to use them.

Section 36.1: Bash internal variables at a glance

Variable	Details
$* / $@	Function/script positional parameters (arguments). Expand as follows: $* and $@ are the same as $1 $2 ... (note that it generally makes no sense to leave those unquoted) "$*" is the same as "$1 $2 ..." 1 "$@" is the same as "$1" "$2" ... 1. Arguments are separated by the first character of $IFS, which does not have to be a space.
$#	Number of positional parameters passed to the script or function
$!	Process ID of the last (righ-most for pipelines) command in the most recently job put into the background (note that it's not necessarily the same as the job's process group ID when job control is enabled)
$$	ID of the process that executed **bash**
$?	Exit status of the last command
$n	Positional parameters, where n=1, 2, 3, ..., 9
${n}	Positional parameters (same as above), but n can be > 9
$0	In scripts, path with which the script was invoked; with **bash** -c 'printf "%s\n" "$0"' name args': name (the first argument after the inline script), otherwise, the argv[0] that **bash** received.
$_	Last field of the last command
$IFS	Internal field separator
$PATH	PATH environment variable used to look-up executables
$OLDPWD	Previous working directory
$PWD	Present working directory
$FUNCNAME	Array of function names in the execution call stack
$BASH_SOURCE	Array containing source paths for elements in FUNCNAME array. Can be used to get the script path.
$BASH_ALIASES	Associative array containing all currently defined aliases
$BASH_REMATCH	Array of matches from the last regex match
$BASH_VERSION	Bash version string
$BASH_VERSINFO	An array of 6 elements with Bash version information
$BASH	Absolute path to the currently executing Bash shell itself (heuristically determined by **bash** based on argv[0] and the value of $PATH; may be wrong in corner cases)
$BASH_SUBSHELL	Bash subshell level
$UID	Real (not effective if different) User ID of the process running **bash**
$PS1	Primary command line prompt; see Using the PS* Variables
$PS2	Secondary command line prompt (used for additional input)
$PS3	Tertiary command line prompt (used in select loop)
$PS4	Quaternary command line prompt (used to append info with verbose output)
$RANDOM	A pseudo random integer between 0 and 32767
$REPLY	Variable used by **read** by default when no variable is specified. Also used by **SELECT** to return the user-supplied value
$PIPESTATUS	Array variable that holds the exit status values of each command in the most recently executed foreground pipeline.

> 变量赋值前后不能有空格。应写成 a=123，不能写成 a = 123。后者（等号两边有空格）单独出现时表示运行命令 a，参数为=和123，尽管它也出现在字符串比较操作符中（语法上是[或[[或你使用的任何测试命令的参数）。

第36.2节：$@

"$@"展开为所有命令行参数，作为独立的单词。它不同于"$*"，后者将所有参数展开为一个单词。

"$@"特别适合用于循环遍历参数以及处理带空格的参数。

假设我们在一个脚本中，该脚本以两个参数调用，如下所示：

```
$ ./script.sh "␣1␣2␣" "␣3␣␣4␣"
```

变量$*或$@将展开为$1␣$2，进而展开为1␣2␣3␣4，因此下面的循环：

```
for var in $*; do # 同样的 for var in $@; do
    echo ||<"$var"||>
done
```

将会打印两者

```
<1>
<2>
<3>
<4>
```

而 "$*" 会被展开成 "$1␣$2"，进而展开成 "␣1␣2␣␣␣3␣␣4␣"，因此循环：

```
for var in "$*"; do
    echo \\<"$var"\\>
done
```

只会调用一次 echo 并打印

```
<␣1␣2␣␣␣3␣␣4␣>
```

最后 "$@" 会展开成 "$1" "$2"，进而展开成 "␣1␣2␣" "␣3␣␣4␣"，因此循环

```
for var in "$@"; do
    echo \\<"$var"\\>
done
```

将会打印

```
<␣1␣2␣>
<␣3␣␣4␣>
```

从而保留了参数内部的空格和参数之间的分隔。请注意

> Variable Assignment must have no space before and after. a=123 not a = 123. The latter (an equal sign surrounded by spaces) in isolation means run the command a with the arguments = and 123, though it is also seen in the string comparison operator (which syntactically is an argument to [or [[or whichever test you are using).

Section 36.2: $@

"$@" expands to all of the command line arguments as separate words. It is different from "$*", which expands to all of the arguments as a single word.

"$@" is especially useful for looping through arguments and handling arguments with spaces.

Consider we are in a script that we invoked with two arguments, like so:

```
$ ./script.sh "␣1␣2␣" "␣3␣␣4␣"
```

The variables $* or $@ will expand into $1␣$2, which in turn expand into 1␣2␣3␣4 so the loop below:

```
for var in $*; do # same for var in $@; do
    echo \\<"$var"\\>
done
```

will print for both

```
<1>
<2>
<3>
<4>
```

While "$*" will be expanded into "$1␣$2" which will in turn expand into "␣1␣2␣␣␣3␣␣4␣" and so the loop:

```
for var in "$*"; do
    echo \\<"$var"\\>
done
```

will only invoke **echo** once and will print

```
<␣1␣2␣␣␣3␣␣4␣>
```

And finally "$@" will expand into "$1" "$2", which will expand into "␣1␣2␣" "␣3␣␣4␣" and so the loop

```
for var in "$@"; do
    echo \\<"$var"\\>
done
```

will print

```
<␣1␣2␣>
<␣3␣␣4␣>
```

thereby preserving both the internal spacing in the arguments and the arguments separation. Note that the

构造for var in "$@"; do ... 是非常常见且惯用的写法，它是for循环的默认形式，并且可以缩写为for var; do ...。

第36.3节：$#

要获取命令行参数或位置参数的数量，请输入：

```
#!/bin/bash
echo "$#"
```

当使用三个参数运行时，上述示例将输出：

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

第36.4节：$HISTSIZE

最大记忆命令数：

```
~> $ echo $HISTSIZE
1000
```

第36.5节：$FUNCNAME

获取当前函数名称的方法 - 输入：

```
my_function()
{
    echo "这个函数是 $FUNCNAME"    # 这将输出 "这个函数是 my_function"
}
```

如果在函数外输入此指令，将不会返回任何内容：

```
my_function

echo "这个函数是 $FUNCNAME"    # 这将输出 "这个函数是"
```

第36.6节：$HOME

用户的主目录

```
~> $ echo $HOME
/home/user
```

第36.7节：$IFS

包含bash在循环等操作中用于拆分字符串的内部字段分隔符字符串。默认是空白字符：

（换行）、（制表符）和空格。将其更改为其他字符可以让你使用不同的字符拆分字符串：

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
```


construction `for var in "$@"; do ...` is so common and idiomatic that it is the default for a for loop and can be shortened to `for var; do ...`.

Section 36.3: $#

To get the number of command line arguments or positional parameters - type:

```
#!/bin/bash
echo "$#"
```

When run with three arguments the example above will result with the output:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

Section 36.4: $HISTSIZE

The maximum number of remembered commands:

```
~> $ echo $HISTSIZE
1000
```

Section 36.5: $FUNCNAME

To get the name of the current function - type:

```
my_function()
{
    echo "This function is $FUNCNAME"    # This will output "This function is my_function"
}
```

This instruction will return nothing if you type it outside the function:

```
my_function

echo "This function is $FUNCNAME"    # This will output "This function is"
```

Section 36.6: $HOME

The home directory of the user

```
~> $ echo $HOME
/home/user
```

Section 36.7: $IFS

Contains the Internal Field Separator string that bash uses to split strings when looping etc. The default is the white space characters: `\n` (newline), `\t` (tab) and space. Changing this to something else allows you to split strings using different characters:

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
```

```
done
```

上述命令的输出是：

```
a
b
c
d
注
释
：
```

- 这就是所谓的单词拆分现象的原因。

第36.8节：$OLDPWD

OLDPWD（OLDPrint工作目录）包含上一次cd命令之前的目录：

```
~> $ cd 目录
directory> $ echo $OLDPWD
/家目录/用户
```

第36.9节：$PWD

PWD（打印工作目录）你当前所在的工作目录：

```
~> $ echo $PWD
/home/user
~> $ cd 目录
directory> $ echo $PWD
/home/user/directory
```

第36.10节：$1 $2 $3 等等

传递给脚本的命令行或函数的位置参数：

```
#!/bin/bash
# $n 是第 n 个位置参数
echo "$1"
echo "$2"
echo "$3"
```

上述命令的输出是：

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

如果位置参数的数量超过九个，必须使用花括号。

```
# “set -- ”设置位置参数
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# 以下行将输出10，而不是1，因为$1的值数字1
# 会与后面的0连接
echo $10   # 输出 1
echo ${10} # 输出 ten
```



```
done
```

The output of the above is:

```
a
b
c
d
```

Notes:

- This is responsible for the phenomenon known as word splitting.

Section 36.8: $OLDPWD

OLDPWD (**OLDP**rint**W**orking**D**irectory) contains directory before the last `cd` command:

```
~> $ cd directory
directory> $ echo $OLDPWD
/home/user
```

Section 36.9: $PWD

PWD (**P**rint**W**orking**D**irectory) The current working directory you are in at the moment:

```
~> $ echo $PWD
/home/user
~> $ cd directory
directory> $ echo $PWD
/home/user/directory
```

Section 36.10: $1 $2 $3 etc..

Positional parameters passed to the script from either the command line or a function:

```
#!/bin/bash
# $n is the n'th positional parameter
echo "$1"
echo "$2"
echo "$3"
```

The output of the above is:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

If number of positional argument is greater than nine, curly braces must be used.

```
#  "set -- " sets positional parameters
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# the following line will output 10 not 1 as the value of $1 the digit 1
# will be concatenated with the following 0
echo $10   # outputs 1
echo ${10} # outputs ten
```

```
# 为了清楚显示这一点：
set -- arg{1..12}
echo $10
echo ${10}
```

第36.11节：$*

将返回所有位置参数组成的单个字符串。

testscript.sh:

```
#!/bin/bash
echo "$*"
```

运行脚本并传入多个参数：

```
./testscript.sh firstarg secondarg thirdarg
```

输出：

```
firstarg secondarg thirdarg
```

第36.12节：$!

最后一个后台运行作业的进程ID（pid）：

```
~> $ ls &
testfile1 testfile2
[1]+  完成                    ls
~> $ echo $!
21715
```

第36.13节：$?

上一个执行的函数或命令的退出状态。通常0表示正常，其他值表示失败：

```
~> $ ls *.blah;echo $?
ls: 无法访问 *.blah: 没有那个文件或目录
2

 > $ ls;echo $?
testfile1 testfile2
0
```

第36.14节：$$

当前进程的进程ID（pid）：

```
~> $ echo $$
13246
```

第36.15节：$RANDOM

每次引用此参数时，都会生成一个介于0到32767之间的随机整数。给


```
# to show this clearly:
set -- arg{1..12}
echo $10
echo ${10}
```

Section 36.11: $*

Will return all of the positional parameters in a single string.

testscript.sh:

```
#!/bin/bash
echo "$*"
```

Run the script with several arguments:

```
./testscript.sh firstarg secondarg thirdarg
```

Output:

```
firstarg secondarg thirdarg
```

Section 36.12: $!

The Process ID (pid) of the last job run in the background:

```
~> $ ls &
testfile1 testfile2
[1]+  Done                    ls
~> $ echo $!
21715
```

Section 36.13: $?

The exit status of the last executed function or command. Usually 0 will mean OK anything else will indicate a failure:

```
~> $ ls *.blah;echo $?
ls: cannot access *.blah: No such file or directory
2
~> $ ls;echo $?
testfile1 testfile2
0
```

Section 36.14: $$

The Process ID (pid) of the current process:

```
~> $ echo $$
13246
```

Section 36.15: $RANDOM

Each time this parameter is referenced, a random integer between 0 and 32767 is generated. Assigning a value to

该变量赋值会为随机数生成器（source）设定种子。

```
~> $ echo $RANDOM
1349
```

第36.16节：$BASHPID

当前Bash实例的进程ID（pid）。这与$$变量不同，但通常结果相同。这是Bash 4中新增加的功能，Bash 3中不支持。

```
~> $ echo "\$\$ pid = $$  BASHPID = $BASHPID"
$$ pid = 9265  BASHPID = 9265
```

第36.17节：$BASH_ENV

指向Bash启动文件的环境变量，该文件在脚本调用时被读取。

第36.18节：$BASH_VERSINFO

一个包含完整版本信息的数组，分割成多个元素，比起$BASH_VERSION更方便，尤其是当你只想查看主版本号时：

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSINFO[$i] = ${BASH_VERSINFO[$i]}"; done
BASH_VERSINFO[0] = 3
BASH_VERSINFO[1] = 2
BASH_VERSINFO[2] = 25
BASH_VERSINFO[3] = 1
BASH_VERSINFO[4] = release
BASH_VERSINFO[5] = x86_64-redhat-linux-gnu
```

第36.19节：$BASH_VERSION

显示当前运行的bash版本，这可以帮助你判断是否可以使用某些高级功能：

```
~> $ echo $BASH_VERSION
4.1.2(1)-release
```

第36.20节：$EDITOR

任何脚本或程序调用的默认编辑器，通常是 vi 或 emacs。

```
~> $ echo $EDITOR
vi
```

第36.21节：$HOSTNAME

系统启动时分配的主机名。

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```


this variable seeds the random number generator (source).

```
~> $ echo $RANDOM
27119
~> $ echo $RANDOM
1349
```

Section 36.16: $BASHPID

Process ID (pid) of the current instance of Bash. This is not the same as the $$ variable, but it often gives the same result. This is new in Bash 4 and doesn't work in Bash 3.

```
~> $ echo "\$\$ pid = $$  BASHPID = $BASHPID"
$$ pid = 9265  BASHPID = 9265
```

Section 36.17: $BASH_ENV

An environment variable pointing to the Bash startup file which is read when a script is invoked.

Section 36.18: $BASH_VERSINFO

An array containing the full version information split into elements, much more convenient than `$BASH_VERSION` if you're just looking for the major version:

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSINFO[$i] = ${BASH_VERSINFO[$i]}"; done
BASH_VERSINFO[0] = 3
BASH_VERSINFO[1] = 2
BASH_VERSINFO[2] = 25
BASH_VERSINFO[3] = 1
BASH_VERSINFO[4] = release
BASH_VERSINFO[5] = x86_64-redhat-linux-gnu
```

Section 36.19: $BASH_VERSION

Shows the version of bash that is running, this allows you to decide whether you can use any advanced features:

```
~> $ echo $BASH_VERSION
4.1.2(1)-release
```

Section 36.20: $EDITOR

The default editor that will be involked by any scripts or programs, usually vi or emacs.

```
~> $ echo $EDITOR
vi
```

Section 36.21: $HOSTNAME

The hostname assigned to the system during startup.

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```

第36.22节：$HOSTTYPE

该变量标识硬件，有助于确定执行哪个二进制文件：

```
~> $ echo $HOSTTYPE
x86_64
```

第36.23节：$MACHTYPE

类似于上面的$HOSTTYPE，这个变量还包含操作系统和硬件的信息

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

第36.24节：$OSTYPE

返回有关运行在机器上的操作系统类型的信息，例如。

```
~> $ echo $OSTYPE
linux-gnu
```

第36.25节：$PATH

查找命令二进制文件的搜索路径。常见示例包括/usr/bin和/usr/local/bin。

当用户或脚本尝试运行命令时，会按顺序搜索$PATH中的路径，以找到具有执行权限的匹配文件。

在$PATH中的目录由:字符分隔。

```
~> $ echo "$PATH"
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
```

例如，给定上述$PATH，如果你在提示符下输入lss，shell将会查找
/usr/kerberos/bin/lss，然后/usr/local/bin/lss，然后/bin/lss，然后/usr/bin/lss，按此顺序，最后才确定没有该命令。

第36.26节：$PPID

脚本或shell的父进程ID（pid），即调用当前脚本或shell的进程。

```
~> $ echo $$

13015
```

第36.27节：$SECONDS

脚本运行的秒数。如果在shell中显示，这个数字可能会很大：

```
~ > $ echo $SECONDS
98834
```


Section 36.22: $HOSTTYPE

This variable identifies the hardware, it can be useful in determining which binaries to execute:

```
~> $ echo $HOSTTYPE
x86_64
```

Section 36.23: $MACHTYPE

Similar to `$HOSTTYPE` above, this also includes information about the OS as well as hardware

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

Section 36.24: $OSTYPE

Returns information about the type of OS running on the machine, eg.

```
~> $ echo $OSTYPE
linux-gnu
```

Section 36.25: $PATH

The search path for finding binaries for commands. Common examples include `/usr/bin` and `/usr/local/bin`.

When a user or script attempts to run a command, the paths in `$PATH` are searched in order to find a matching file with execute permission.

The directories in `$PATH` are separated by a : character.

```
~> $ echo "$PATH"
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
```

So, for example, given the above `$PATH`, if you type `lss` at the prompt, the shell will look for `/usr/kerberos/bin/lss`, then `/usr/local/bin/lss`, then `/bin/lss`, then `/usr/bin/lss`, in this order, before concluding that there is no such command.

Section 36.26: $PPID

The Process ID (pid) of the script or shell's parent, meaning the process than invoked the current script or shell.

```
~> $ echo $$
13016
~> $ echo $PPID
13015
```

Section 36.27: $SECONDS

The number of seconds a script has been running. This can get quite large if shown in the shell:

```
~> $ echo $SECONDS
98834
```

第36.28节：$SHELLOPTS

bash 启动时提供的只读选项列表，用于控制其行为：

```
~> $ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

第 36.29 节：$_

输出上一个执行命令的最后一个字段，常用于将某些内容传递给另一个
命令：

```
~> $ ls *.sh;echo $_
testscript1.sh  testscript2.sh
testscript2.sh
```

如果在其他命令之前使用，它会返回脚本路径：

test.sh：

```
#!/bin/bash
echo "$_"
```

输出：

```
~> $ ./test.sh # 运行 test.sh
./test.sh
```

注意： 这不是获取脚本路径的万无一失的方法

第36.30节：$GROUPS

包含用户所属组编号的数组：

```
#!/usr/bin/env bash
echo 您被分配到以下组：
for group in ${GROUPS[@]}; do
    IFS=: read -r name dummy number members < <(getent group $group )printf
     "name: %-10s number: %-15s members: %s" "$name" "$number" "$members"done
```

第36.31节：$LINENO

输出当前脚本中的行号。主要用于调试脚本时。

```
#!/bin/bash
# 这是第2行
echo something  # 这是第3行
echo $LINENO # 将输出4
```

第36.32节：$SHLVL

当执行 bash 命令时，会打开一个新的 shell。环境变量 $SHLVL 保存当前 shell 所运行的层级数。

Section 36.28: $SHELLOPTS

A readonly list of the options bash is supplied on startup to control its behaviour:

```
~> $ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

Section 36.29: $_

Outputs the last field from the last command executed, useful to get something to pass onwards to another command:

```
 ~> $ ls *.sh;echo $_
testscript1.sh  testscript2.sh
testscript2.sh
```

It gives the script path if used before any other commands:

test.sh:

```
#!/bin/bash
echo "$_"
```

Output:

```
~> $ ./test.sh # running test.sh
./test.sh
```

Note: This is not a foolproof way to get the script path

Section 36.30: $GROUPS

An array containing the numbers of groups the user is in:

```
#!/usr/bin/env bash
echo You are assigned to the following groups:
for group in ${GROUPS[@]}; do
   IFS=: read -r name dummy number members < <(getent group $group )
   printf "name: %-10s number: %-15s members: %s\n" "$name" "$number" "$members"
done
```

Section 36.31: $LINENO

Outputs the line number in the current script. Mostly useful when debugging scripts.

```
#!/bin/bash
# this is line 2
echo something  # this is line 3
echo $LINENO # Will output 4
```

Section 36.32: $SHLVL

When the bash command is executed a new shell is opened. The $SHLVL environment variable holds the number of shell levels the *current* shell is running on top of.

在一个新的终端窗口中，执行以下命令会根据所使用的 Linux 发行版产生不同的结果。

```
echo $SHLVL
```

使用 Fedora 25 时，输出为“3”。这表明，当打开一个新的 shell 时，初始的 bash 命令会执行并完成一个任务。初始的 bash 命令执行一个子进程（另一个 bash 命令），该子进程又执行最终的 bash 命令以打开新的 shell。新 shell 打开时，作为另外两个 shell 进程的子进程运行，因此输出为“3”。

在以下示例中（假设用户正在运行 Fedora 25），新 shell 中 $SHLVL 的输出将被设置为“3”。每执行一个 bash 命令，$SHLVL 会增加一。

```
~> $ echo $SHLVL
3

 > $ bash
~> $ echo $SHLVL
4

 > $ bash
~> $ echo $SHLVL
5
```

可以看到，执行 'bash' 命令（或执行一个 bash 脚本）会打开一个新的 shell。相比之下，source 一个脚本会在当前 shell 中运行代码。

test1.sh

```
#!/usr/bin/env bash
echo "来自 test1.sh 的问候。我的 shell 级别是 $SHLVL"
source "test2.sh"
```

test2.sh

```
#!/usr/bin/env bash
echo "来自 test2.sh 的问候。我的 shell 级别是 $SHLVL"
```

run.sh

```
#!/usr/bin/env bash
echo "来自 run.sh 的问候。我的 shell 级别是 $SHLVL"
./test1.sh
```

执行：

```
chmod +x test1.sh && chmod +x run.sh
./run.sh
```

输出：

```
来自 run.sh 的问候。我的 shell 级别是 4
来自 test1.sh 的问候。我的 shell 级别是 5
来自 test2.sh 的问候。我的 shell 级别是 5
```


In a *new* terminal window, executing the following command will produce different results based on the Linux distribution in use.

```
echo $SHLVL
```

Using *Fedora 25*, the output is "3". This indicates, that when opening a new shell, an initial bash command executes and performs a task. The initial bash command executes a child process (another bash command) which, in turn, executes a final bash command to open the new shell. When the new shell opens, it is running as a child process of 2 other shell processes, hence the output of "3".

In the following example (given the user is running Fedora 25), the output of $SHLVL in a new shell will be set to "3". As each bash command is executed, $SHLVL increments by one.

```
~> $ echo $SHLVL
3
~> $ bash
~> $ echo $SHLVL
4
~> $ bash
~> $ echo $SHLVL
5
```

One can see that executing the 'bash' command (or executing a bash script) opens a new shell. In comparison, sourcing a script runs the code in the current shell.

test1.sh

```
#!/usr/bin/env bash
echo "Hello from test1.sh. My shell level is $SHLVL"
source "test2.sh"
```

test2.sh

```
#!/usr/bin/env bash
echo "Hello from test2.sh. My shell level is $SHLVL"
```

run.sh

```
#!/usr/bin/env bash
echo "Hello from run.sh. My shell level is $SHLVL"
./test1.sh
```

Execute:

```
chmod +x test1.sh && chmod +x run.sh
./run.sh
```

Output:

```
Hello from run.sh. My shell level is 4
Hello from test1.sh. My shell level is 5
Hello from test2.sh. My shell level is 5
```

第 36.33 节：$UID

一个只读变量，存储用户的 ID 号：

```
~> $ echo $UID
12345
```


Section 36.33: $UID

A read only variable that stores the users' ID number:

```
~> $ echo $UID
12345
```

第 37 章：作业控制

第 37.1 节：列出后台进程

```
$ jobs
[1]   运行中              sleep 500 &  (当前目录: ~)
[2]-  运行中              sleep 600 &  (当前目录: ~)
[3]+  运行中              ./Fritzing &
```

第一栏显示作业 ID。紧跟作业 ID 的 + 和 - 符号分别表示默认作业和当前默认作业结束后下一个候选默认作业。当使用 fg 或 bg 命令且不带任何参数时，将使用默认作业。

第二栏显示作业状态。第三栏是启动该进程所用的命令。

最后一个字段（wd: ~）表示睡眠命令是从工作目录~（主目录）启动的。

第37.2节：将后台进程调到前台

```
$ fg %2
sleep 600
```

%2 指定作业编号2。如果 fg 不带任何参数，则会将最后一个放入后台的进程调到前台。

```
$ fg %?sle
sleep 500
```

?sle 指包含 "sle" 的后台进程命令。如果多个后台命令包含该字符串，则会产生错误。

第37.3节：重新启动停止的后台进程

```
$ bg
[8]+ sleep 600 &
```

第37.4节：在后台运行命令

```
$ sleep 500 &
[1] 7582
```

将 sleep 命令放入后台。7582 是后台进程的进程 ID。

第37.5节：停止前台进程

按 Ctrl + Z 停止前台进程并将其放入后台

```
$ sleep 600
^Z
[8]+  已停止              sleep 600
```


Chapter 37: Job Control

Section 37.1: List background processes

```
$ jobs
[1]   Running             sleep 500 &  (wd: ~)
[2]-  Running             sleep 600 &  (wd: ~)
[3]+  Running             ./Fritzing &
```

First field shows the job ids. The + and - sign that follows the job id for two jobs denote the default job and next candidate default job when the current default job ends respectively. The default job is used when the `fg` or `bg` commands are used without any argument.

Second field gives the status of the job. Third field is the command used to start the process.

The last field (wd: ~) says that the sleep commands were started from the working directory ~ (Home).

Section 37.2: Bring a background process to the foreground

```
$ fg %2
sleep 600
```

%2 specifies job no. 2. If fg is used without any arguments if brings the last process put in background to the foreground.

```
$ fg %?sle
sleep 500
```

`?sle` refers to the baground process command containing "sle". If multiple background commands contain the string, it will produce an error.

Section 37.3: Restart stopped background process

```
$ bg
[8]+ sleep 600 &
```

Section 37.4: Run command in background

```
$ sleep 500 &
[1] 7582
```

Puts the sleep command in background. 7582 is the process id of the background process.

Section 37.5: Stop a foreground process

Press Ctrl + Z to stop a foreground process and put it in background

```
$ sleep 600
^Z
[8]+  Stopped             sleep 600
```

第38章：case 语句

第38.1节：简单的 case 语句

在所有 bash 版本支持的最简单形式中，case 语句执行与模式匹配的 case。
;; 操作符在第一个匹配后（如果有）跳出。

```
#!/bin/bash

var=1
case $var in
1)
  echo "Antartica"
 ;;
2)
  echo "Brazil"
 ;;
3)
  echo "Cat"
 ;;
esac
```

输出：

```
Antartica
```

第38.2节：带穿透的case语句

版本 ≥ 4.0

自bash 4.0起，引入了一个新操作符;&，提供了穿透机制。

#!/bin/bash

```
var=1
case $var in
1)
  echo "南极洲"
  ;&
2)
  echo "巴西"
  ;&
3)
   echo "猫"
  ;&
esac
```

输出：

```
南极洲
巴西
猫
```

第38.3节：仅当后续模式匹配时才继续执行

版本 ≥ 4.0

Chapter 38: Case statement

Section 38.1: Simple case statement

In its simplest form supported by all versions of bash, case statement executes the case that matches the pattern.
;; operator breaks after the first match, if any.

```
#!/bin/bash

var=1
case $var in
1)
  echo "Antartica"
 ;;
2)
  echo "Brazil"
 ;;
3)
  echo "Cat"
 ;;
esac
```

Outputs:

```
Antartica
```

Section 38.2: Case statement with fall through

Version ≥ 4.0

Since bash 4.0, a new operator **;&** was introduced which provides fall through mechanism.

#!/bin/bash

```
var=1
case $var in
1)
  echo "Antartica"
  ;&
2)
  echo "Brazil"
  ;&
3)
  echo "Cat"
  ;&
esac
```

Outputs:

```
Antartica
Brazil
Cat
```

Section 38.3: Fall through only if subsequent pattern(s) match

Version ≥ 4.0

自Bash 4.0起，引入了另一个操作符;;&，它也提供了仅当后续case语句中的模式（如果有）匹配时的继续执行。

```
#!/bin/bash

var=abc
case $var in
a*)
  echo "南极洲"
  ;;&
xyz)
  echo "巴西"
  ;;&
*b*)
  echo "猫"
  ;;&
esac
```

输出：

```
南极洲
猫
```

在下面的例子中，abc 同时匹配第一个和第三个情况，但不匹配第二个情况。因此，第二个情况不会被执行。

Since Bash 4.0, another operator `;;&` was introduced which also provides fall through *only if* the patterns in subsequent case statement(s), if any, match.

```
#!/bin/bash

var=abc
case $var in
a*)
  echo "Antartica"
  ;;&
xyz)
  echo "Brazil"
  ;;&
*b*)
  echo "Cat"
  ;;&
esac
```

Outputs:

```
Antartica
Cat
```

In the below example, the `abc` matches both first and third case but not the second case. So, second case is not executed.

第39章：逐行（和/或逐字段）读取文件（数据流，变量）？

参数	详细信息
IFS	内部字段分隔符
文件	文件名/路径
`-r`	与 read 一起使用时防止反斜杠被解释
`-t`	移除由 readarray 读取的每行末尾的换行符
`-d 分隔符`	继续读取直到遇到 DELIM 的第一个字符（使用 read），而不是换行符

第 39.1 节：逐行循环读取文件

```
while IFS= read -r line; do
   echo "$line"
done <file
```

如果文件末尾可能没有换行符，则：

```
while IFS= read -r line || [ -n "$line" ]; do
   echo "$line"
done <file
```

第 39.2 节：逐字段循环处理命令输出

假设字段分隔符是：

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done < <(ping google.com)
```

或者使用管道：

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done
```

第39.3节：将文件的行读入数组

```
readarray -t arr <file
```

或者使用循环：

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <file
```


Chapter 39: Read a file (data stream, variable) line-by-line (and/or field-by-field)?

Parameter	Details
IFS	Internal field separator
file	A file name/path
`-r`	Prevents backslash interpretation when used with read
`-t`	Removes a trailing newline from each line read by `readarray`
`-d DELIM`	Continue until the first character of DELIM is read (with **`read`**), rather than newline

Section 39.1: Looping through a file line by line

```
while IFS= read -r line; do
   echo "$line"
done <file
```

If file may not include a newline at the end, then:

```
while IFS= read -r line || [ -n "$line" ]; do
   echo "$line"
done <file
```

Section 39.2: Looping through the output of a command field by field

Let's assume that the field separator is :

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done < <(ping google.com)
```

Or with a pipe:

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done
```

Section 39.3: Read lines of a file into an array

```
readarray -t arr <file
```

Or with a loop:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <file
```

第39.4节：将字符串的行读入数组

```
var='line 1
line 2
line3'
readarray -t arr <<< "$var"
```

或者使用循环：

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <<< "$var"
```

第39.5节：逐行循环遍历字符串

```
var='line 1
line 2
line3'
while IFS= read -r line; do
   echo "-$line-"
done <<< "$var"
```

或者

```
readarray -t arr <<< "$var"
for i in "${arr[@]}";do
    echo "-$i-"
done
```

第39.6节：逐行循环遍历命令行输出

```
while IFS= read -r line;do
    echo "**$line**"
done < <(ping google.com)
```

或者使用管道：

```
ping google.com |
while IFS= read -r line;do
    echo "**$line**"
done
```

第39.7节：逐字段读取文件

假设文件file中的字段分隔符是:（冒号）。

```
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "$field"
done <file
```

对于内容：

```
first : se
```


Section 39.4: Read lines of a string into an array

```
var='line 1
line 2
line3'
readarray -t arr <<< "$var"
```

or with a loop:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <<< "$var"
```

Section 39.5: Looping through a string line by line

```
var='line 1
line 2
line3'
while IFS= read -r line; do
   echo "-$line-"
done <<< "$var"
```

or

```
readarray -t arr <<< "$var"
for i in "${arr[@]}";do
    echo "-$i-"
done
```

Section 39.6: Looping through the output of a command line by line

```
while IFS= read -r line;do
    echo "**$line**"
done < <(ping google.com)
```

or with a pipe:

```
ping google.com |
while IFS= read -r line;do
    echo "**$line**"
done
```

Section 39.7: Read a file field by field

Let's assume that the field separator is : (colon) in the file *file*.

```
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "$field"
done <file
```

For a content:

```
first : se
```

```
con
d:
    Thi rd:
    Fourth
```

输出是：

```
**第一**
** 第二
第二
d**
**
第三**
**
    Fourth
**
```

第39.8节：逐字段读取字符串

假设字段分隔符是：

```
var='line: 1
line: 2
line3'
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "-$field-"
done <<< "$var"
```

输出：

```
-line-
- 1
line-
- 2
line3
-
```

第39.9节：将文件的字段读入数组

假设字段分隔符是：

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <file
```

第39.10节：将字符串的字段读入数组

假设字段分隔符是：

```
var='1:2:3:4:
newline'
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <<< "$var"
```


con
d:
    Thi rd:
    Fourth

The output is:

```
**first **
** se
con
d**
**
    Thi rd**
**
    Fourth
**
```

Section 39.8: Read a string field by field

Let's assume that the field separator is :

```
var='line: 1
line: 2
line3'
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "-$field-"
done <<< "$var"
```

Output:

```
-line-
- 1
line-
- 2
line3
-
```

Section 39.9: Read fields of a file into an array

Let's assume that the field separator is :

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <file
```

Section 39.10: Read fields of a string into an array

Let's assume that the field separator is :

```
var='1:2:3:4:
newline'
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <<< "$var"
```

```
echo "${arr[4]}"
```

输出：

```
换行符
```

第39.11节：逐行逐字段读取文件（/etc/passwd）

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS=: read -r username password userid groupid comment homedir cmdshell
do
  echo "$username, $userid, $comment $homedir"
done < $FILENAME
```

在unix密码文件中，用户信息按行存储，每行包含一个用户的信息，字段之间用冒号（:）分隔。在此示例中，读取文件时逐行读取，同时使用冒号字符作为分隔符将该行拆分成多个字段，这由IFS的值指示。

示例输入

```
mysql:x:27:27:MySQL服务器:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio系统守护进程:/var/run/pulse:/sbin/nologin
sshd:x:74:74:特权分离SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

示例输出

```
mysql, 27, MySQL服务器 /var/lib/mysql
pulse, 497, PulseAudio系统守护进程 /var/run/pulse
sshd, 74, 特权分离SSH /var/empty/sshd
tomcat, 91, Apache Tomcat /usr/share/tomcat6
webalizer, 67, Webalizer /var/www/usage
```

要逐行读取并将整行赋值给变量，以下是示例的修改版本。
注意这里只提到了一个名为line的变量。

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
  echo "$line"
done < $FILENAME
```

示例输入

```
mysql:x:27:27:MySQL服务器:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio系统守护进程:/var/run/pulse:/sbin/nologin
sshd:x:74:74:特权分离SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

示例输出


```
echo "${arr[4]}"
```

Output:

```
newline
```

Section 39.11: Reads file (/etc/passwd) line by line and field by field

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS=: read -r username password userid groupid comment homedir cmdshell
do
  echo "$username, $userid, $comment $homedir"
done < $FILENAME
```

In unix password file, user information is stored line by line, each line consisting of information for a user separated by colon (:) character. In this example while reading the file line by line, the line is also split into fields using colon character as delimiter which is indicated by the value given for IFS.

Sample input

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Sample Output

```
mysql, 27, MySQL Server /var/lib/mysql
pulse, 497, PulseAudio System Daemon /var/run/pulse
sshd, 74, Privilege-separated SSH /var/empty/sshd
tomcat, 91, Apache Tomcat /usr/share/tomcat6
webalizer, 67, Webalizer /var/www/usage
```

To read line by line and have the entire line assigned to variable, following is a modified version of the example. Note that we have only one variable by name line mentioned here.

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
  echo "$line"
done < $FILENAME
```

Sample Input

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Sample Output

```
mysql:x:27:27:MySQL服务器:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio系统守护进程:/var/run/pulse:/sbin/nologin
sshd:x:74:74:特权分离SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```



```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

第40章：文件执行顺序

.bash_profile、.bash_login、.bashrc 和 .profile 都差不多：设置和定义
函数、变量等。

主要区别是 .bashrc 在打开非登录但交互式窗口时调用，
.bash_profile 和其他文件则在登录 shell 时调用。许多人无论如何都会让他们的 .bash_profile 或类似文件调用 .bashrc。

第40.1节：.profile 与 .bash_profile（及 .bash_login）

.profile 在大多数 shell 启动时都会被读取，包括 bash。然而，.bash_profile 用于 bash 特定的配置。
通用初始化代码放在 .profile 中。如果是 bash 特有的，使用 .bash_profile。

.profile 实际上并非专为 bash 设计，而 .bash_profile 是。（.profile 是为 Bourne 及其他类似 shell 设计的，bash 基于它们）如果找不到 .bash_profile，bash 会回退使用 .profile。

.bash_login 是 .bash_profile 的备用文件，如果找不到 .bash_profile，通常最好使用 .bash_profile 或 .profile 。

Chapter 40: File execution sequence

.bash_profile, .bash_login, .bashrc, and .profile all do pretty much the same thing: set up and define functions, variables, and the sorts.

The main difference is that .bashrc is called at the opening of a non-login but interactive window, and .bash_profile and the others are called for a login shell. Many people have their .bash_profile or similar call .bashrc anyway.

Section 40.1: .profile vs .bash_profile (and .bash_login)

.profile is read by most shells on startup, including bash. However, .bash_profile is used for configurations specific to bash. For general initialization code, put it in .profile. If it's specific to bash, use .bash_profile.

.profile isn't actually designed for bash specifically, .bash_profile is though instead. (.profile is for Bourne and other similar shells, which bash is based off) Bash will fall back to .profile if .bash_profile isn't found.

.bash_login is a fallback for .bash_profile, if it isn't found. Generally best to use .bash_profile or .profile instead.

第41章：拆分文件

有时将一个文件拆分成多个独立的文件是很有用的。如果你有大文件，拆分成更小的块可能是个好主意。

第41.1节：拆分文件

运行 split 命令且不带任何选项时，会将文件拆分成一个或多个独立文件，每个文件最多包含1000行。

```
split 文件
```

这将创建名为 xaa、xab、xac 等文件，每个文件最多包含1000行。如你所见，默认情况下所有文件都以字母 x 为前缀。如果原始文件少于1000行，则只会创建一个这样的文件。

要更改前缀，请在命令行末尾添加你想要的前缀。

```
split 文件 customprefix
```

现在将创建名为 customprefixaa、customprefixab、customprefixac 等文件。要指定每个文件输出

的行数，请使用 -l 选项。以下命令将文件拆分为最多5000行的文件。

```
split -l5000 file
```

或者

```
split --lines=5000 文件
```

或者，您可以指定最大字节数而不是行数。这可以通过使用-b或--bytes
选项来完成。例如，允许最大为1MB

```
split --bytes=1MB 文件
```


Chapter 41: Splitting Files

Sometimes it's useful to split a file into multiple separate files. If you have large files, it might be a good idea to break it into smaller chunks

Section 41.1: Split a file

Running the split command without any options will split a file into 1 or more separate files containing up to 1000 lines each.

```
split file
```

This will create files named `xaa`, `xab`, `xac`, etc, each containing up to 1000 lines. As you can see, all of them are prefixed with the letter `x` by default. If the initial file was less than 1000 lines, only one such file would be created.

To change the prefix, add your desired prefix to the end of the command line

```
split file customprefix
```

Now files named `customprefixaa`, `customprefixab`, `customprefixac` etc. will be created

To specify the number of lines to output per file, use the `-l` option. The following will split a file into a maximum of 5000 lines

```
split -l5000 file
```

OR

```
split --lines=5000 file
```

Alternatively, you can specify a maximum number of bytes instead of lines. This is done by using the `-b` or `--bytes` options. For example, to allow a maximum of 1MB

```
split --bytes=1MB file
```

第42章：使用scp进行文件传输

第42.1节：scp传输文件

要安全地将文件传输到另一台机器，请输入：

```
scp file1.txt tom@server2:$HOME
```

此示例展示了将file1.txt从我们的主机传输到server2的用户tom的主目录。

第42.2节：scp传输多个文件

scp 也可以用来将多个文件从一台服务器传输到另一台服务器。下面是一个示例，将所有扩展名为.txt的文件从my_folder目录传输到server2。在下面的示例中，所有文件将被传输到用户om的主目录。

```
scp /my_folder/*.txt tom@server2:$HOME
```

第42.3节：使用scp下载文件

要从远程服务器下载文件到本地机器，输入：

```
scp tom@server2:$HOME/file.txt /local/machine/path/
```

此示例展示如何将名为file.txt的文件从用户om的主目录下载到我们本地机器的当前目录。

Chapter 42: File Transfer using scp

Section 42.1: scp transferring file

To transfer a file securely to another machine - type:

```
scp file1.txt tom@server2:$HOME
```

This example presents transferring `file1.txt` from our host to `server2`'s user `tom`'s home directory.

Section 42.2: scp transferring multiple files

`scp` can also be used to transfer multiple files from one server to another. Below is example of transferring all files from `my_folder` directory with extension `.txt` to `server2`. In Below example all files will be transferred to user `tom` home directory.

```
scp /my_folder/*.txt tom@server2:$HOME
```

Section 42.3: Downloading file using scp

To download a file from remote server to the local machine - type:

```
scp tom@server2:$HOME/file.txt /local/machine/path/
```

This example shows how to download the file named `file.txt` from user `tom`'s home directory to our local machine's current directory.

第43章：管道

第43.1节：使用|&

|& 将第一个命令的标准输出和标准错误连接到第二个命令，而|仅连接第一个命令的标准输出到第二个命令。

在此示例中，页面通过curl下载。使用-v选项时，curl会将一些信息写入stderr，包括下载的页面写入stdout。页面标题可以在<title>和</title>之间找到。

```
curl -vs 'http://www.google.com/' |& awk '/Host:/{print}
/<title>/{match($0,/<title>(.*)<\/title>/,a);print a[1]}'
```

输出为：

```
> 主机：www.google.com
Google
```

但是使用|会打印更多信息，即那些发送到stderr的内容，因为只有stdout被传递给下一个命令。在此示例中，除最后一行（Google）外，所有行均由curl发送到stderr：

```
* 主机名未在DNS缓存中找到
*   正在尝试连接172.217.20.228...
* 已连接到www.google.com（172.217.20.228）端口80（#0）
> GET / HTTP/1.1
> 用户代理：curl/7.35.0
> 主机：www.google.com
> 接受：*/*
>
* HTTP 1.0，假设正文后关闭连接
< HTTP/1.0 200 OK
< 日期：2016年7月24日星期日19:04:59 GMT
< 过期时间：-1
< 缓存控制：私有，最大存活时间=0
< 内容类型：text/html；字符集=ISO-8859-1
< P3P：CP="这不是P3P策略！请参见
https://www.google.com/support/accounts/answer/151657?hl=en 获取更多信息。
< 服务器: gws
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: NID=82=jX0yZLPPUE7u13kKNevUCDg8yG9Ze_C03o0IM-
EopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWF-
oEoq2T0sBTuQVdsIFULj9b2O8X35O0sAgUnc3a3JnTRBqelMcuS9QkQA; expires=Mon, 23-Jan-2017 19:04:59 GMT;
path=/; domain=.google.com; HttpOnly
< Accept-Ranges: none
< Vary: Accept-Encoding
< X-Cache: MISS 来自 jetsib_appliance
< X-Loop-Control: 5.202.190.157 81E4F9836653D5812995BA53992F8065
< Connection: close
<
{ [数据未显示]
* 关闭连接 0
谷歌
```


Chapter 43: Pipelines

Section 43.1: Using |&

|& connects standard output and standard error of the first command to the second one while | only connects standard output of the first command to the second command.

In this example, the page is downloaded via `curl`. with `-v` option `curl` writes some info on `stderr` including , the downloaded page is written on `stdout`. Title of page can be found between **`<title>`** and **`</title>`**.

```
curl -vs 'http://www.google.com/' |& awk '/Host:/{print}
/<title>/{match($0,/<title>(.*)<\/title>/,a);print a[1]}'
```

Output is:

```
> Host: www.google.com
Google
```

But with | a lot more information will be printed, i.e. those that are sent to `stderr` because only `stdout` is piped to the next command. In this example all lines except the last line (Google) were sent to `stderr` by `curl`:

```
* Hostname was NOT found in DNS cache
*   Trying 172.217.20.228...
* Connected to www.google.com (172.217.20.228) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: www.google.com
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Date: Sun, 24 Jul 2016 19:04:59 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< P3P: CP="This is not a P3P policy! See
https://www.google.com/support/accounts/answer/151657?hl=en for more info."
< Server: gws
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: NID=82=jX0yZLPPUE7u13kKNevUCDg8yG9Ze_C03o0IM-
EopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWF-
oEoq2T0sBTuQVdsIFULj9b2O8X35O0sAgUnc3a3JnTRBqelMcuS9QkQA; expires=Mon, 23-Jan-2017 19:04:59 GMT;
path=/; domain=.google.com; HttpOnly
< Accept-Ranges: none
< Vary: Accept-Encoding
< X-Cache: MISS from jetsib_appliance
< X-Loop-Control: 5.202.190.157 81E4F9836653D5812995BA53992F8065
< Connection: close
<
{ [data not shown]
* Closing connection 0
Google
```

第43.2节：显示所有分页的流程

```
ps -e | less
```

ps -e 显示所有进程，其输出通过 | 连接到 more 的输入，less 对结果进行分页显示。

第43.3节：修改命令的连续输出

```
~$ ping -c 1 google.com # 未修改的输出
PING google.com (16.58.209.174) 56(84) 字节的数据。
64来自 wk-in-f100.1e100.net (16.58.209.174)的字节： icmp_seq=1 ttl=53 time=47.4 毫秒
~$ ping google.com | grep -o '^[0-9]\+[^()]\+' # 修改后的输出
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
64来自 wk-in-f100.1e100.net 的字节
...
```

管道符（|）将 ping 的 stdout 连接到 grep 的 stdin，后者会立即处理它。其他一些命令如 sed 默认对其 stdin 进行缓冲，这意味着它必须接收足够的数据后才会打印任何内容，可能导致后续处理的延迟。

Section 43.2: Show all processes paginated

```
ps -e | less
```

`ps -e` shows all the processes, its output is connected to the input of more via `|`, `less` paginates the results.

Section 43.3: Modify continuous output of a command

```
~$ ping -c 1 google.com # unmodified output
PING google.com (16.58.209.174) 56(84) bytes of data.
64 bytes from wk-in-f100.1e100.net (16.58.209.174): icmp_seq=1 ttl=53 time=47.4 ms
~$ ping google.com | grep -o '^[0-9]\+[^()]\+' # modified output
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
...
```

The pipe (`|`) connects the `stdout` of **`ping`** to the `stdin` of **`grep`**, which processes it immediately. Some other commands like **`sed`** default to buffering their `stdin`, which means that it has to receive enough data, before it will print anything, potentially causing delays in further processing.

第44章：管理 PATH 环境变量

参数	详细信息
PATH	路径环境变量

第44.1节：向PATH环境变量添加路径

PATH环境变量通常定义在~/.bashrc或~/.bash_profile或/etc/profile或~/.profile或/etc/bash.bashrc（发行版特定的Bash配置文件）中

```
$ echo $PATH
                /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:
/usr/lib/jvm/jdk1.8.0_92/bin:/usr/lib/jvm/jdk1.8.0_92/db/bin:/usr/lib/jvm/jdk1.8.0_92/jre/bin
```

现在，如果我们想向PATH变量添加一个路径（例如 ~/bin）：

```
PATH=~/bin:$PATH
# 或者
PATH=$PATH:~/bin
```

但这只会修改当前 shell（及其子 shell）中的 PATH。一旦退出 shell，这个修改将会消失。

要使其永久生效，我们需要将那段代码添加到 ~/.bashrc（或其他相应）文件中，并重新加载该文件。

如果你在终端运行以下代码，它将永久地将 ~/bin 添加到 PATH 中：

```
echo 'PATH=~/bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

说明：

- echo 'PATH=~/bin:$PATH' >> ~/.bashrc 会在 ~/.bashrc 文件末尾添加一行 PATH=~/bin:$PATH（你也可以用文本编辑器来做这件事）
- source ~/.bashrc 会重新加载 ~/.bashrc 文件

这是一段代码（在终端运行），它会检查某个路径是否已包含，只有在未包含时才添加该路径：

```
path=~/bin            # 要包含的路径
bashrc=~/.bashrc      # 要写入并重新加载的 bash 文件
# 运行以下代码，保持不变
echo $PATH | grep -q "\(^\|:\)$path\(:\|/\{0,1\}$\)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

第44.2节：从 PATH 环境变量中移除路径

要从PATH环境变量中移除某个路径，您需要编辑~/.bashrc或~/.bash_profile或/etc/profile或~/.profile或/etc/bash.bashrc（取决于发行版）文件，并删除该路径的赋值。

您也可以不去查找确切的赋值，而是在最终阶段直接对$PATH进行替换。

下面的命令将安全地从$PATH中移除$path：

Chapter 44: Managing PATH environment variable

Parameter	Details
PATH	Path environment variable

Section 44.1: Add a path to the PATH environment variable

The PATH environment variable is generally defined in ~/.bashrc or ~/.bash_profile or /etc/profile or ~/.profile or /etc/bash.bashrc (distro specific Bash configuration file)

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:
/usr/lib/jvm/jdk1.8.0_92/bin:/usr/lib/jvm/jdk1.8.0_92/db/bin:/usr/lib/jvm/jdk1.8.0_92/jre/bin
```

Now, if we want to add a path (e.g ~/bin) to the PATH variable:

```
PATH=~/bin:$PATH
# or
PATH=$PATH:~/bin
```

But this will modify the PATH only in the current shell (and its subshell). Once you exit the shell, this modification will be gone.

To make it permanent, we need to add that bit of code to the ~/.bashrc (or whatever) file and reload the file.

If you run the following code (in terminal), it will add ~/bin to the PATH permanently:

```
echo 'PATH=~/bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

Explanation:

- `echo 'PATH=~/bin:$PATH' >> ~/.bashrc` adds the line `PATH=~/bin:$PATH` at the end of ~/.bashrc file (you could do it with a text editor)
- `source ~/.bashrc` reloads the ~/.bashrc file

This is a bit of code (run in terminal) that will check if a path is already included and add the path only if not:

```
path=~/bin            # path to be included
bashrc=~/.bashrc      # bash file to be written and reloaded
# run the following code unmodified
echo $PATH | grep -q "\(^\|:\)$path\(:\|/\{0,1\}$\)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

Section 44.2: Remove a path from the PATH environment variable

To remove a PATH from a PATH environment variable, you need to edit ~/.bashrc or ~/.bash_profile or /etc/profile or ~/.profile or /etc/bash.bashrc (distro specific) file and remove the assignment for that particular path.

Instead of finding the exact assignment, you could just do a replacement in the $PATH in its final stage.

The following will safely remove $path from $PATH:

```
path=~/bin
PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
```

要使其永久生效，您需要将其添加到bash配置文件的末尾。

您可以用函数的方式来实现：

```
rpath(){
     for path in "$@";do
         PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
    done
echo "$PATH"
}

PATH="$(rpath ~/bin /usr/local/sbin /usr/local/bin)"
PATH="$(rpath /usr/games)"
# 等等 ...
```

这将使处理多个路径更容易。

注释：

- 您需要将这些代码添加到 Bash 配置文件（~/.bashrc 或其他文件）中。
- 运行 source ~/.bashrc 以重新加载 Bash 配置文件（~/.bashrc）。


```
path=~/bin
PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
```

To make it permanent, you will need to add it at the end of your bash configuration file.

You can do it in a functional way:

```
rpath(){
    for path in "$@";do
        PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
    done
    echo "$PATH"
}

PATH="$(rpath ~/bin /usr/local/sbin /usr/local/bin)"
PATH="$(rpath /usr/games)"
# etc ...
```

This will make it easier to handle multiple paths.

Notes:

- You will need to add these codes in the Bash configuration file (~/.bashrc or whatever).
- Run `source ~/.bashrc` to reload the Bash configuration (~/.bashrc) file.

第45章：单词拆分

参数	详细信息
IFS	内部字段分隔符
-x	打印命令及其参数的执行过程（Shell 选项）

第45.1节：什么、何时以及为什么？

当 shell 执行参数扩展、命令替换、变量或算术扩展时，它会扫描结果中的单词边界。如果发现任何单词边界，则结果会在该位置被拆分成多个单词。单词边界由 shell 变量IFS（内部字段分隔符）定义。IFS 的默认值是空格、制表符和换行符，即如果不显式防止，单词拆分将发生在这三种空白字符上。

```
set -x
var='我是
a
多行字符串'
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

在上述示例中，fun 函数的执行方式如下：

```
fun 我是 多行字符串
```

```
fun 我是 多行字符串
```

> $var 被拆分成 5 个参数，只有 我、是 和 多行 会被打印。

第 45.2 节：单词拆分的负面影响

```
$  a ='我 是 一个 带空格的字符串'
$ [ $a = $a ] || echo "未匹配"
bash: [: 参数过多
didn't match
```

> **[$a = $a] 被解释为 [我是一个带空格的字符串 = 我是一个带空格的字符串]。 [是** the **test** 命令，其中 我是一个带空格的字符串 不是单个参数，而是 **6** 参数！！

```
$ [ $a = something ] || echo "didn't match"
bash: [: 参数过多
didn't match
```

> [$a = something] 被解释为 [我是一个带空格的字符串 = something]

```
$ [ $(grep . file) = 'something' ]
```


Chapter 45: Word splitting

Parameter	Details
IFS	Internal field separator
-x	Print commands and their arguments as they are executed (Shell option)

Section 45.1: What, when and Why?

When the shell performs *parameter expansion*, *command substitution*, *variable or arithmetic expansion*, it scans for word boundaries in the result. If any word boundary is found, then the result is split into multiple words at that position. The word boundary is defined by a shell variable `IFS` (Internal Field Separator). The default value for IFS are space, tab and newline, i.e. word splitting will occur on these three white space characters if not prevented explicitly.

```
set -x
var='I am
a
multiline string'
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

In the above example this is how the `fun` function is being executed:

```
fun I am a multiline string
```

```
fun I am a multiline string
```

> `$var` is split into 5 args, only `I`, `am` and `a` will be printed.

Section 45.2: Bad effects of word splitting

```
$ a='I am a string with spaces'
$ [ $a = $a ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = $a ]` was interpreted as `[ I am a string with spaces = I am a string with spaces ]`. [is the **test** command for which `I am a string with spaces` is not a single argument, rather it's **6** arguments!!

```
$ [ $a = something ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = something ]` was interpreted as `[ I am a string with spaces = something ]`

```
$ [ $(grep . file) = 'something' ]
```

```
bash: [: 参数过多
```

> grep 命令返回一个带空格的多行字符串，所以你可以想象有多少参数在那里...:D

查看基础的内容：什么、何时以及为什么。

第45.3节：单词拆分的有用性

有些情况下，单词拆分是有用的：

填充数组：

```
arr=($(grep -o '[0-9]\+' file))
```

> 这将用在file中找到的所有数字值填充arr

遍历以空格分隔的单词：

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

输出：

```
W: foo
W: bar
W: baz
```

传递不包含空白字符的以空格分隔的参数：

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

或者

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

> 这将安装这些软件包。如果你用双引号括住$packs，则会报错。

> 未加引号的$packs会将所有以空格分隔的软件包名称作为参数传递给apt-get，而加引号后会将$packs字符串作为单个参数传递，apt-get会尝试安装名为apache2 php php-mbstring php-mysql（以第一个为例）的软件包，显然该软件包不存在。


```
bash: [: too many arguments
```

> The `grep` command returns a multiline string with spaces, so you can just imagine how many arguments are there...:D

See what, when and why for the basics.

Section 45.3: Usefulness of word splitting

There are some cases where word splitting can be useful:

Filling up array:

```
arr=($(grep -o '[0-9]\+' file))
```

> This will fill up `arr` with all numeric values found in *file*

Looping through space separated words:

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

Output:

```
W: foo
W: bar
W: baz
```

Passing space separated parameters which don't contain white spaces:

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

or

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

> This will install the packages. If you double quote the `$packs` then it will throw an error.

> Unquoetd `$packs` is sending all the space separated package names as arguments to `apt-get`, while quoting it will send the `$packs` string as a single argument and then `apt-get` will try to install a package named `apache2 php php-mbstring php-mysql` (for the first one) which obviously doesn't exist

查看基础的内容：什么、何时以及为什么。

第45.4节：按分隔符拆分的变化

我们可以简单地将分隔符从空格替换为换行符，如下面的示例所示。

```
echo $sentence | tr " " ""
```

它会将变量sentence的值拆分，并逐行显示。

第45.5节：使用IFS拆分

为了更清楚起见，我们创建一个名为showarg的脚本：

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

现在让我们看看区别：

```
$ var="这是一个例子"
$ showarg $var
4参数：<这是> <一个> <例子> <>
```

> $var 被拆分成4个参数。 IFS 是空白字符，因此在空格处发生了单词拆分

```
$ var="这是/一个/例子"
$ showarg $var
1参数：<这是/一个/例子>
```

> 上面没有发生单词拆分，因为没有找到 IFS 字符。

现在让我们设置 IFS=/

```
$ IFS=/
$ var="这是/一个/例子"
$ showarg $var
4参数：<这是> <一个> <例子> <>
```

> 变量 $var 被拆分成4个参数，而不是单个参数。

第45.6节：IFS与单词拆分

如果你不了解IFS与单词拆分的关系，看看是什么、什么时候以及为什么

我们将IFS设置为空格字符：

```
设置 -x
var='I am
```


See what, when and why for the basics.

Section 45.4: Splitting by separator changes

We can just do simple replacement of separators from space to new line, as following example.

```
echo $sentence | tr " " "\n"
```

It'll split the value of the variable `sentence` and show it line by line respectively.

Section 45.5: Splitting with IFS

To be more clear, let's create a script named `showarg`:

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

Now let's see the differences:

```
$ var="This is an example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

> `$var` is split into 4 args. `IFS` is white space characters and thus word splitting occurred in spaces

```
$ var="This/is/an/example"
$ showarg $var
1 args: <This/is/an/example>
```

> In above word splitting didn't occur because the `IFS` characters weren't found.

Now let's set `IFS=/`

```
$ IFS=/
$ var="This/is/an/example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

> The `$var` is splitting into 4 arguments not a single argument.

Section 45.6: IFS & word splitting

See what, when and why if you don't know about the affiliation of IFS to word splitting

let's set the IFS to space character only:

```
set -x
var='I am
```

```
a
多行字符串'
IFS=' '
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

这次单词拆分只会在空格上生效。函数fun将这样执行：

```
fun I 'am
a
多行' 字符串
```

> $var 被拆分成3个参数。 I， ama多行 和 字符串 将被打印

让我们将IFS设置为仅换行符：

```
IFS=$"...
```

现在fun将像这样执行：

```
fun '我是一段' 多行字符串'
```

> $var被分割成3个参数。 我是一段，多行字符串将被打印出来

让我们看看如果将IFS设置为空字符串会发生什么：

```
IFS=
...
```

这次fun将像这样执行：

```
fun '我是一段
a
多行字符串'
```

> $var 未拆分，即保持为单个参数。

你可以通过将 IFS 设置为空字符串来防止单词拆分

防止单词拆分的一般方法是使用双引号：

```
fun "$var"
```

将在上述所有情况下防止单词拆分，即fun函数将只用一个参数执行。


```
a
multiline string'
IFS=' '
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

This time word splitting will only work on spaces. The `fun` function will be executed like this:

```
fun I 'am
a
multiline' string
```

> `$var` is split into 3 args. `I`, `am\na\nmultiline` and `string` will be printed

Let's set the IFS to newline only:

```
IFS=$'\n'
...
```

Now the `fun` will be executed like:

```
fun 'I am' a 'multiline string'
```

> `$var` is split into 3 args. `I am`, `a`, `multiline string` will be printed

Let's see what happens if we set IFS to nullstring:

```
IFS=
...
```

This time the `fun` will be executed like this:

```
fun 'I am
a
multiline string'
```

> `$var` is not split i.e it remained a single arg.

You can prevent word splitting by setting the IFS to nullstring

A general way of preventing word splitting is to use double quote:

```
fun "$var"
```

will prevent word splitting in all the cases discussed above i.e the `fun` function will be executed with only one argument.

第46章：使用 printf 避免日期问题

在 Bash 4.2 中，引入了 printf 的内置时间转换：格式说明符%(datefmt)T使得 printf 输出与 strftime 理解的格式字符串datefmt对应的日期时间字符串。

第46.1节：获取当前日期

```
$ printf '%(%F)T'
2016-08-17
```

第46.2节：将变量设置为当前时间

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```


Chapter 46: Avoiding date using printf

In Bash 4.2, a shell built-in time conversion for `printf` was introduced: the format specification `%(datefmt)T` makes `printf` output the date-time string corresponding to the format string `datefmt` as understood by `strftime`.

Section 46.1: Get the current date

```
$ printf '%(%F)T\n'
2016-08-17
```

Section 46.2: Set variable to current time

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```

第47章：使用“trap”响应信号和系统事件

参数	含义
-p	列出当前已安装的trap
-l	列出信号名称及对应编号

第47.1节：简介：清理临时文件

您可以使用trap命令来“捕获”信号；这相当于C语言及大多数其他编程语言中的signal()或sigaction()调用，用于捕捉信号。

使用trap最常见的用途之一是在预期或意外退出时清理临时文件。

遗憾的是，很多shell脚本并没有做到这一点 :-(

```
#!/bin/sh

# 创建一个清理函数
cleanup() {
  rm --force -- "${tmp}"
}

# 捕获特殊的“EXIT”组，该组在shell退出时总会执行。
trap cleanup EXIT

# 创建一个临时文件
tmp="$(mktemp -p /tmp tmpfileXXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# 不需要 rm -f "$tmp"。使用 EXIT 的好处是即使发生错误或使用了 exit，清理操作仍然会执行。
```

第47.2节：捕获 SIGINT 或 Ctrl+C

trap 会在子shell中重置，因此 sleep 仍会响应由 ^C 发送的 SIGINT 信号（通常会退出），但父进程（即shell脚本）不会。

```
#!/bin/sh

# 在信号2（SIGINT，即 ^C 发送的信号）时运行命令
sigint() {
    echo "子shell已被终止！"
}
trap sigint INT

# 或者使用无操作命令以无输出
#trap : INT

# 这将在第一次 ^C 时被终止
echo "正在休眠..."
sleep 500

echo "正在休眠..."
sleep 500
```


Chapter 47: Using "trap" to react to signals and system events

Parameter	Meaning
-p	List currently installed traps
-l	List signal names and corresponding numbers

Section 47.1: Introduction: clean up temporary files

You can use the **trap** command to "trap" signals; this is the shell equivalent of the `signal()` or `sigaction()` call in C and most other programming languages to catch signals.

One of the most common uses of **trap** is to clean up temporary files on both an expected and unexpected exit.

Unfortunately not enough shell scripts do this :-(

```
#!/bin/sh

# Make a cleanup function
cleanup() {
  rm --force -- "${tmp}"
}

# Trap the special "EXIT" group, which is always run when the shell exits.
trap cleanup EXIT

# Create a temporary file
tmp="$(mktemp -p /tmp tmpfileXXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# No rm -f "$tmp" needed. The advantage of using EXIT is that it still works
# even if there was an error or if you used exit.
```

Section 47.2: Catching SIGINT or Ctl+C

The trap is reset for subshells, so the **sleep** will still act on the `SIGINT` signal sent by `^C` (usually by quitting), but the parent process (i.e. the shell script) won't.

```
#!/bin/sh

# Run a command on signal 2 (SIGINT, which is what ^C sends)
sigint() {
    echo "Killed subshell!"
}
trap sigint INT

# Or use the no-op command for no output
#trap : INT

# This will be killed on the first ^C
echo "Sleeping..."
sleep 500

echo "Sleeping..."
sleep 500
```

还有一种变体，允许你在一秒内按两次 ^C 来退出主程序：

```
last=0
allow_quit() {
    [ $(date +%s) -lt $(( $last + 1 )) ] && exit
    echo "连续按两次 ^C 以退出"
    last=$(date +%s)
}
trap allow_quit INT
```

第47.3节：积累一份在退出时运行的陷阱工作列表

你是否曾经忘记添加一个trap来清理临时文件或在退出时执行其他工作？

你是否曾经设置了一个trap，结果取消了另一个trap？

这段代码使得在退出时逐条添加要执行的操作变得简单，而不是在代码某处写一个庞大的`trap`语句，这样更容易被遗忘。

```
# on_exit 和 add_on_exit
# 用法：
#   add_on_exit rm -f /tmp/foo
#   add_on_exit echo "我正在退出"
#   tempfile=$(mktemp)
#   add_on_exit rm -f "$tempfile"
# 基于 http://www.linuxjournal.com/content/use-bash-trap-statement-cleanup-temporary-files
function on_exit()
{
    for i in "${on_exit_items[@]}"
    do
        eval $i
    done
}
function add_on_exit()
{
    local n=${#on_exit_items[*]}
    on_exit_items[$n]="$*"
    if [[ $n -eq 0 ]]; then
        trap on_exit EXIT
    fi
}
```

第47.4节：退出时终止子进程

Trap表达式不必是单独的函数或程序，也可以是更复杂的表达式。

通过结合jobs -p和kill，我们可以在退出时终止shell启动的所有子进程：

```
trap 'jobs -p | xargs kill' EXIT
```

第47.5节：响应终端窗口大小变化

有一个信号WINCH（窗口变化，WINdowCHange），当调整终端窗口大小时会触发该信号。

```
declare -x rows cols

update_size(){
```


And a variant which still allows you to quit the main program by pressing ^C twice in a second:

```
last=0
allow_quit() {
    [ $(date +%s) -lt $(( $last + 1 )) ] && exit
    echo "Press ^C twice in a row to quit"
    last=$(date +%s)
}
trap allow_quit INT
```

Section 47.3: Accumulate a list of trap work to run at exit

Have you ever forgotten to add a `trap` to clean up a temporary file or do other work at exit?

Have you ever set one trap which canceled another?

This code makes it easy to add things to be done on exit one item at a time, rather than having one large `trap` statement somewhere in your code, which may be easy to forget.

```
# on_exit and add_on_exit
# Usage:
#   add_on_exit rm -f /tmp/foo
#   add_on_exit echo "I am exiting"
#   tempfile=$(mktemp)
#   add_on_exit rm -f "$tempfile"
# Based on http://www.linuxjournal.com/content/use-bash-trap-statement-cleanup-temporary-files
function on_exit()
{
    for i in "${on_exit_items[@]}"
    do
        eval $i
    done
}
function add_on_exit()
{
    local n=${#on_exit_items[*]}
    on_exit_items[$n]="$*"
    if [[ $n -eq 0 ]]; then
        trap on_exit EXIT
    fi
}
```

Section 47.4: Killing Child Processes on Exit

Trap expressions don't have to be individual functions or programs, they can be more complex expressions as well.

By combining `jobs -p` and `kill`, we can kill all spawned child processes of the shell on exit:

```
trap 'jobs -p | xargs kill' EXIT
```

Section 47.5: react on change of terminals window size

There is a signal WINCH (WINdowCHange), which is fired when one resizes a terminal window.

```
declare -x rows cols

update_size(){
```

```
  rows=$(tput lines) # 获取终端的实际行数
  cols=$(tput cols)  # 获取终端的实际列数
  echo DEBUG 终端窗口没有 $rows 行，宽度为 $cols 个字符
}

trap update_size WINCH
```



```
  rows=$(tput lines) # get actual lines of term
  cols=$(tput cols)  # get actual columns of term
  echo DEBUG terminal window has no $rows lines and is $cols characters wide
}

trap update_size WINCH
```

第48章：命令链和操作

有一些方法可以将命令串联起来。简单的方法如仅用分号（;），更复杂的方法如根据某些条件运行的逻辑链。第三种是管道命令，它实际上将输出数据传递给链中的下一个命令。

第48.1节：统计文本模式出现次数

使用管道可以使一个命令的输出成为下一个命令的输入。

```
ls -1 | grep -c ".conf"
```

在这种情况下，ls命令的输出被用作grep命令的输入。结果将是文件名中包含“.conf”的文件数量。

这可以用来构建所需长度的后续命令链：

```
ls -1 | grep ".conf" | grep -c .
```

第48.2节：将root命令输出传输到用户文件

通常人们希望将以root身份执行的命令结果展示给其他用户。 tee 命令可以轻松地以用户权限将命令（以root身份运行）的输出写入文件：

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

只有 ifconfig 以root身份运行。

第48.3节：使用 && 和 || 进行命令的逻辑链式连接

&& 连接两个命令。只有第一个命令成功退出时，第二个命令才会运行。 || 也连接两个命令。
但第二个命令只有在第一个命令失败退出时才会运行。

```
[ a = b  ] && echo "yes" || echo "no"

# 如果你想在逻辑链中运行更多命令，可以使用大括号
# 来指定一组命令块
# 在闭合大括号前需要加上分号，以便bash能区分大括号的其他用法

[ a = b ] && { echo "让我看看。"
               echo "嗯，是的，我认为是真的" ; } \
          || { echo "既然我处于否定状态，我认为 "
               echo "这是假的。a不是b。"  ; }
# 注意换行续行符 |
# 仅在将yes块与||连接时需要
```

第48.4节：使用分号进行命令的串行链式连接

分号仅用于分隔两个命令。

```
echo "我是第一个" ; echo "我是第二个" ; echo "我是第三个"
```


Chapter 48: Chain of commands and operations

There are some means to chain commands together. Simple ones like just a ; or more complex ones like logical chains which run depending on some conditions. The third one is piping commands, which effectively hands over the output data to the next command in the chain.

Section 48.1: Counting a text pattern ocurrence

Using a pipe makes the output of a command be the input of the next one.

```
ls -1 | grep -c ".conf"
```

In this case the output of the ls command is used as the input of the grep command. The result will be the number of files that include ".conf" in their name.

This can be used to contruct chains of subsequent commands as long as needed:

```
ls -1 | grep ".conf" | grep -c .
```

Section 48.2: transfer root cmd output to user file

Often one want to show the result of a command executed by root to other users. The **tee** command allows easily to write a file with user perms from a command running as root:

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

Only **ifconfig** runs as root.

Section 48.3: logical chaining of commands with && and ||

&& chains two commands. The second one runs only if the first one exits with success. **||** chains two commands. But second one runs only if first one exits with failure.

```
[ a = b  ] && echo "yes" || echo "no"

# if you want to run more commands within a logical chain, use curly braces
# which designate a block of commands
# They do need a ; before closing bracket so bash can diffentiate from other uses
# of curly braces
[ a = b ] && { echo "let me see."
               echo "hmmm, yes, i think it is true" ; } \
          || { echo "as i am in the negation i think "
               echo "this is false. a is a not b."  ; }
# mind the use of line continuation sign \
# only needed to chain yes block with || ....
```

Section 48.4: serial chaining of commands with semicolon

A semicolon separates just two commands.

```
echo "i am first" ; echo "i am second" ; echo " i am third"
```

第48.5节：使用 | 链接命令

| 将左侧命令的输出作为右侧命令的输入。请注意，这在一个子shell中执行。因此，您无法在管道中设置调用进程的变量值。

```
find . -type f -a -iname '*.mp3'  | \
       while read filename; do
         mute --noise "$filename"
       done
```


Section 48.5: chaining commands with |

The | takes the output of the left command and pipes it as input the right command. Mind, that this is done in a subshell. Hence you cannot set values of vars of the calling process within a pipe.

```
find . -type f -a -iname '*.mp3'  | \
       while read filename; do
             mute --noise "$filename"
       done
```

第49章：Shell 类型

第49.1节：启动交互式 shell

```
bash
```

第49.2节：检测 shell 类型

```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

第49.3节：dot文件简介

在Unix中，以点号开头的文件和目录通常包含特定程序或一系列程序的设置。dot文件通常对用户是隐藏的，因此你需要运行ls-a才能看到它们。

一个dot文件的例子是.bash_history，它包含最近执行的命令，假设用户正在使用Bash。

当你进入Bash shell时，有各种文件会被sourced。下图取自本网站，展示了启动时选择要source哪些文件的决策过程。

Chapter 49: Type of Shells

Section 49.1: Start an interactive shell

```
bash
```

Section 49.2: Detect type of shell

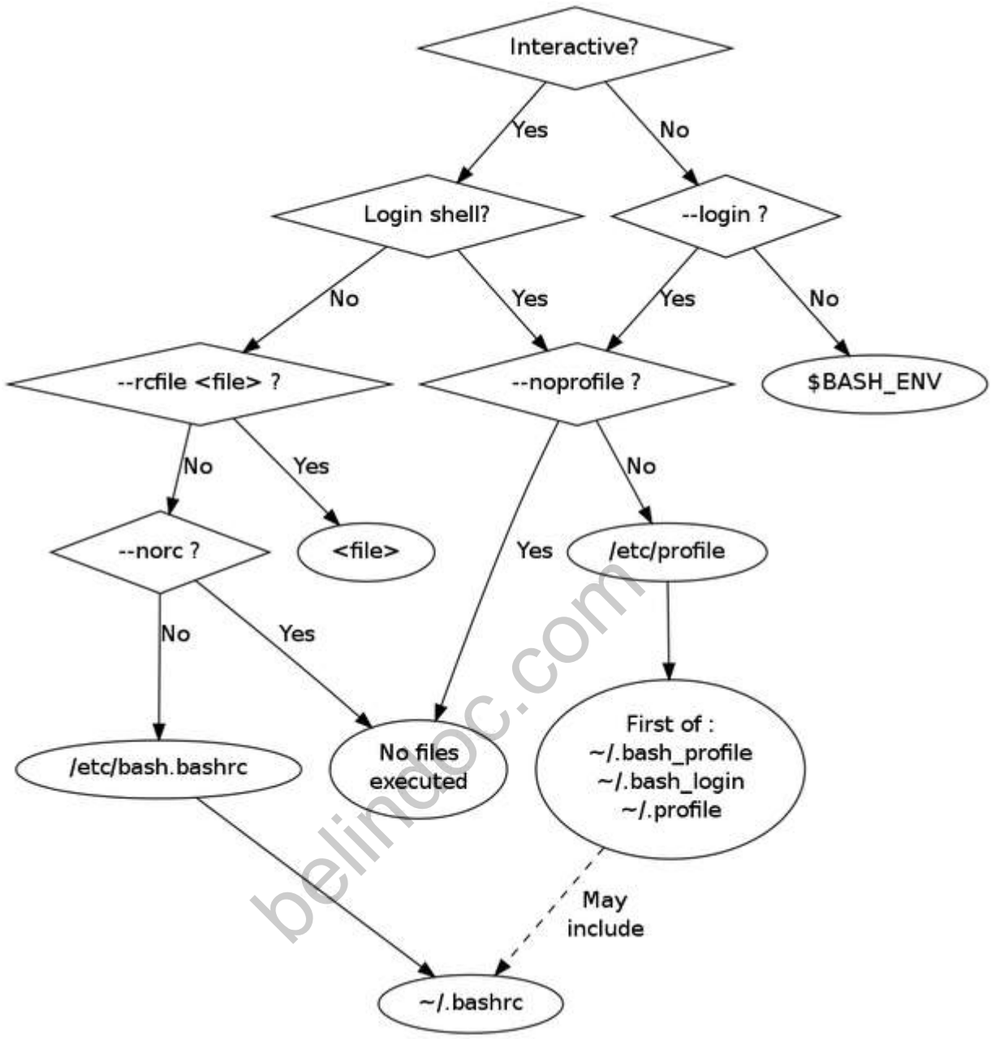
```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

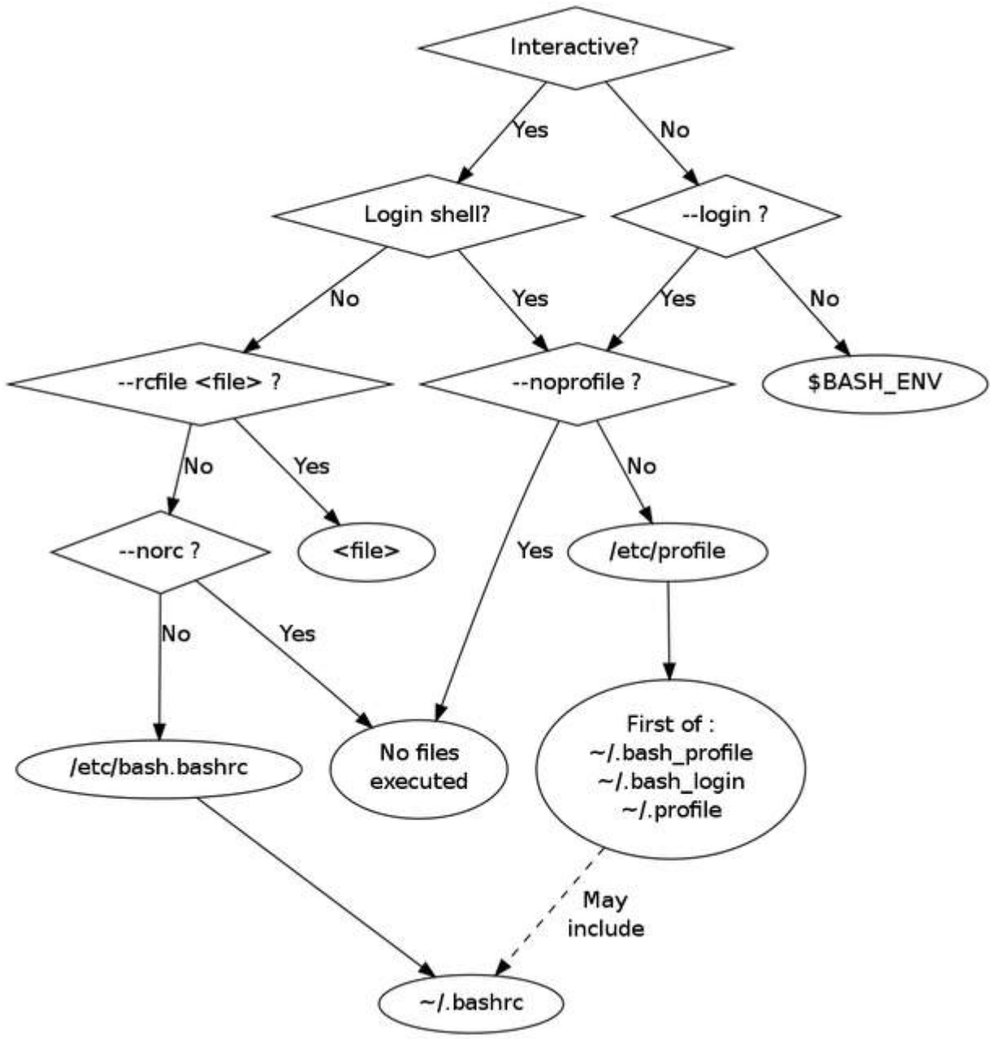
Section 49.3: Introduction to dot files

In Unix, files and directories beginning with a period usually contain settings for a specific program/a series of programs. Dot files are usually hidden from the user, so you would need to run `ls -a` to see them.

An example of a dot file is `.bash_history`, which contains the latest executed commands, assuming the user is using Bash.

There are various files that are sourced when you are dropped into the Bash shell. The image below, taken from this site, shows the decision process behind choosing which files to source at startup.





第50章：彩色脚本输出（跨平台）

第50.1节：color-output.sh

在bash脚本的开头部分，可以定义一些变量，作为辅助工具，在脚本运行期间为终端输出着色或进行其他格式化。

不同平台使用不同的字符序列来表示颜色。然而，有一个名为tput的工具，它适用于所有*nix系统，并通过一致的跨平台API返回平台特定的终端着色字符串。

例如，存储将终端文本变为红色或绿色的字符序列：

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

或者，存储将文本重置为默认外观的字符序列：

```
reset=$(tput sgr0)
```

然后，如果BASH脚本需要显示不同颜色的输出，可以通过以下方式实现：

cho "${green}成功！${reset}" echo "${red}失败。${reset}"

Chapter 50: Color script output (cross-platform)

Section 50.1: color-output.sh

In the opening section of a bash script, it's possible to define some variables that function as helpers to color or otherwise format the terminal output during the run of the script.

Different platforms use different character sequences to express color. However, there's a utility called `tput` which works on all *nix systems and returns platform-specific terminal coloring strings via a consistent cross-platform API.

For example, to store the character sequence which turns the terminal text red or green:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

Or, to store the character sequence which resets the text to default appearance:

```
reset=$(tput sgr0)
```

Then, if the BASH script needed to show different colored outputs, this can be achieved with:

cho "${green}Success!${reset}" echo "${red}Failure.${reset}"

第51章：协同进程

第51.1节：Hello World

```
# 创建协同进程
coproc bash

# 向其发送命令（echo a）
echo 'echo Hello World' >&"${COPROC[1]}"

# 从其输出读取一行
read line <&"${COPROC[0]}"

# 显示该行
echo "$line"
```

输出是 "Hello World"。

Chapter 51: co-processes

Section 51.1: Hello World

```
# create the co-process
coproc bash

# send a command to it (echo a)
echo 'echo Hello World' >&"${COPROC[1]}"

# read a line from its output
read line <&"${COPROC[0]}"

# show the line
echo "$line"
```

The output is "Hello World".

第52章：变量类型

第52.1节：声明弱类型变量

declare 是bash的内部命令。（内部命令使用 **help** 来显示“手册页”）。它用于显示和定义变量或显示函数体。

语法：declare [选项] [名称[=值]]...

```
# 选项用于定义
# 一个整数
declare -i myInteger
declare -i anotherInt=10
# 一个带值的数组
declare -a anArray=( one two three)
# 一个关联数组
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# 注意bash识别[]内的字符串上下文

# 存在一些修饰符
# 大写内容
声明 -u big='这将是大写'
# 小写同理
declare -l small='THIS WILL BE LOWERCASE'

# 只读数组
declare -ra constarray=( eternal true and unchangeable )

# 将整数导出到环境
declare -xi importantInt=42
```

你也可以使用 + 来移除给定的属性。大多没什么用，只是为了完整性。

显示变量和/或函数也有一些选项

```
# 打印已定义的变量和函数
declare -f
# 限制输出仅显示函数
declare -F # 如果调试，打印定义的行号和文件名
```


Chapter 52: Typing variables

Section 52.1: declare weakly typed variables

declare is an internal command of bash. (internal command use **help** for displaying "manpage"). It is used to show and define variables or show function bodies.

Syntax: **declare [options] [name[=value]]...**

```
# options are used to define
# an integer
declare -i myInteger
declare -i anotherInt=10
# an array with values
declare -a anArray=( one two three)
# an assoc Array
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# note that bash recognizes the string context within []

# some modifiers exist
# uppercase content
declare -u big='this will be uppercase'
# same for lower case
declare -l small='THIS WILL BE LOWERCASE'

# readonly array
declare -ra constarray=( eternal true and unchangeable )

# export integer to environment
declare -xi importantInt=42
```

You can use also the + which takes away the given attribute. Mostly useless, just for completness.

To display variables and/or functions there are some options too

```
# printing definded vars and functions
declare -f
# restrict output to functions only
declare -F # if debugging prints line number and filename defined in too
```

第53章：特定时间的作业

第53.1节：在特定时间执行一次任务

注意：大多数现代发行版默认未安装 at。

要在除当前时间之外的某个时间执行一次任务，例如下午5点，可以使用

```
echo "somecommand &" | at 5pm
```

如果想捕获输出，可以用常规方式：

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

at 支持多种时间格式，所以你也可以这样说

```
echo "somecommand &" | at now + 2 minutes
echo "somecommand &" | at 17:00
echo "somecommand &" | at 17:00 Jul 7
echo "somecommand &" | at 4pm 12.03.17
```

如果未指定年份或日期，则假定为你指定时间的下一次出现时间。因此，如果你指定的小时今天已经过去，则假定为明天；如果指定的月份今年已经过去，则假定为明年。

这也可以像预期的那样与nohup一起使用。

```
echo "nohup somecommand > out.txt 2>err.txt &" | 在下午5点执行
```

还有一些用于控制定时任务的命令：

- atq 列出所有定时任务（atqueue）
- atrm 删除一个定时任务（atremove）
- batch 基本上与 at 相同，但仅在系统负载低于0.8时运行任务

所有命令均适用于当前登录用户的任务。如果以 root 用户登录，则当然处理系统范围内的任务。

第53.2节：使用systemd.timer重复执行指定时间的任务

systemd 提供了一个现代化的 cron 实现。要周期性执行脚本，需要一个服务文件和一个定时器文件。服务和定时器文件应放置在 /etc/systemd/{system,user} 目录下。服务文件内容如下：

```
[Unit]
Description=我的脚本或程序表现最佳，这就是描述

[服务]
# 类型很重要！
类型=简单
# 调用的程序|脚本。始终使用绝对路径
# 并重定向STDIN和STDERR，因为执行时没有终端
ExecStart=/绝对/路径/到/某命令 >>/路径/到/输出 2>/路径/到/STDERR输出
#没有安装部分！！！！由定时器功能本身处理。
#[安装]
```


Chapter 53: Jobs at specific times

Section 53.1: Execute job once at specific time

Note: **at** is not installed by default on most of modern distributions.

To execute a job once at some other time than now, in this example 5pm, you can use

```
echo "somecommand &" | at 5pm
```

If you want to catch the output, you can do that in the usual way:

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

`at` understands many time formats, so you can also say

```
echo "somecommand &" | at now + 2 minutes
echo "somecommand &" | at 17:00
echo "somecommand &" | at 17:00 Jul 7
echo "somecommand &" | at 4pm 12.03.17
```

If no year or date are given, it assumes the next time the time you specified occurs. So if you give a hour that already passed today, it will assume tomorrow, and if you give a month that already passed this year, it will assume next year.

This also works together with nohup like you would expect.

```
echo "nohup somecommand > out.txt 2>err.txt &" | at 5pm
```

There are some more commands to control timed jobs:

- **atq** lists all timed jobs (**atq**ueue)
- **atrm** removes a timed job (**atrm**ove)
- **batch** does basically the same like at, but runs jobs only when system load is lower than 0.8

All commands apply to jobs of the user logged in. If logged in as root, system wide jobs are handled of course.

Section 53.2: Doing jobs at specified times repeatedly using systemd.timer

systemd provides a modern implementation of **cron**. To execute a script periodical a service and a timer file ist needed. The service and timer files should be placed in /etc/systemd/{system,user}. The service file:

```
[Unit]
Description=my script or programm does the very best and this is the description

[Service]
# type is important!
Type=simple
# program|script to call. Always use absolute pathes
# and redirect STDIN and STDERR as there is no terminal while being executed
ExecStart=/absolute/path/to/someCommand >>/path/to/output 2>/path/to/STDERRoutput
#NO install section!!!! Is handled by the timer facitlities itself.
#[Install]
```

```
#WantedBy=multi-user.target
```

接下来是定时器文件：

```
[Unit]
描述=我的第一个systemd定时器
[计时器]
# 日期/时间规格的语法是 Y-m-d H:M:S
# 星号 * 表示“每个”，也可以给出逗号分隔的列表
# *-*-* *,15,30,45:00 表示每年、每月、每天、每小时，
# 在第15、30、45分钟和零秒时执行

OnCalendar=*-*-* *:01:00
# 这个任务在每小时的第1分钟0秒运行，例如 13:01:00
```


#WantedBy=multi-user.target

Next the timer file:

```
[Unit]
Description=my very first systemd timer
[Timer]
# Syntax for date/time specifications is  Y-m-d H:M:S
# a * means "each", and a comma separated list of items can be given too
# *-*-* *,15,30,45:00  says every year, every month, every day, each hour,
# at minute 15,30,45 and zero seconds

OnCalendar=*-*-* *:01:00
# this one runs each hour at one minute zero second e.g. 13:01:00
```

第54章：处理系统提示

转义	详细信息
\a	一个响铃字符。
\d	日期，格式为“星期几 月份 日期”（例如，“周二 5月 26日”）。
\D{FORMAT}	FORMAT 会传递给 `strftime'(3)，结果会插入到提示字符串中；空的 FORMAT 会生成特定于区域设置的时间表示。大括号是必须的。
\e	一个转义字符。当然，\033 也可以使用。
\h	主机名，直到第一个`.`为止。（即不包括域名部分）
\H	主机名，包含域名部分
\j	当前由 shell 管理的作业数量。
\l	shell 终端设备名称的基本名。
	换行符。
\r	回车符。
\s	shell 的名称，即 `$0` 的基本名（最后一个斜杠之后的部分）。
	时间，24 小时制 HH:MM:SS 格式。
\T	时间，12 小时制 HH:MM:SS 格式。
@	时间，采用12小时制上午/下午格式。
\A	时间，24小时制 HH:MM 格式。
\u	当前用户的用户名。
\v	Bash 的版本（例如，2.00）
\V	Bash 的发行版本，版本号 + 补丁级别（例如，2.00.0）
\w	当前工作目录，$HOME 用波浪号缩写（使用 $PROMPT_DIRTRIM 变量）。
\W	$PWD 的基本名称，$HOME 用波浪号缩写。
!	该命令的历史编号。
#	该命令的命令编号。
$	如果有效用户ID是0，#，否则$。
\NNN	ASCII码为八进制值NNN的字符。
\	反斜杠。
\[	开始一段非打印字符序列。这可以用来将终端控制序列嵌入到提示符中。
\]	结束一段非打印字符序列。

第54.1节：使用PROMPT_COMMAND环境变量

当交互式bash实例中的最后一个命令执行完毕后，评估后的PS1变量会被显示。在实际显示PS1之前，bash会检查PROMPT_COMMAND是否被设置。该变量的值必须是一个可调用的程序或脚本。如果该变量被设置，则在显示PS1提示符之前会调用该程序/脚本。

```
# 只是一个简单的函数，我们用来演示
# 我们检查时间，如果小时是12且分钟小于59
lunchbreak(){
    if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
        # 并打印彩色 \033[ 开始转义序列
        # 5; 是闪烁属性
        # 2；表示加粗
        # 31 表示红色
```


Chapter 54: Handling the system prompt

Escape	Details
\a	A bell character.
\d	The date, in "Weekday Month Date" format (e.g., "Tue May 26").
\D{FORMAT}	The FORMAT is passed to `strftime'(3) and the result is inserted into the prompt string; an empty FORMAT results in a locale-specific time representation. The braces are required.
\e	An escape character. \033 works of course too.
\h	The hostname, up to the first `.'. (i.e. no domain part)
\H	The hostname eventually with domain part
\j	The number of jobs currently managed by the shell.
\l	The basename of the shell's terminal device name.
\n	A newline.
\r	A carriage return.
\s	The name of the shell, the basename of `$0' (the portion following the final slash).
\t	The time, in 24-hour HH:MM:SS format.
\T	The time, in 12-hour HH:MM:SS format.
@	The time, in 12-hour am/pm format.
\A	The time, in 24-hour HH:MM format.
\u	The username of the current user.
\v	The version of Bash (e.g., 2.00)
\V	The release of Bash, version + patchlevel (e.g., 2.00.0)
\w	The current working directory, with $HOME abbreviated with a tilde (uses the $PROMPT_DIRTRIM variable).
\W	The basename of $PWD, with $HOME abbreviated with a tilde.
!	The history number of this command.
#	The command number of this command.
$	If the effective uid is 0, **#**, otherwise **$**.
\NNN	The character whose ASCII code is the octal value NNN.
\	A backslash.
\[	Begin a sequence of non-printing characters. This could be used to embed a terminal control sequence into the prompt.
\]	End a sequence of non-printing characters.

Section 54.1: Using the PROMPT_COMMAND envrionment variable

When the last command in an interactive bash instance is done, the evaluated PS1 variable is displayes. Before actually displaying PS1 bash looks whether the PROMPT_COMMAND is set. This value of this var must be a callable program or script. If this var is set this program/script is called BEFORE the PS1 prompt is displayed.

```
# just a stupid function, we will use to demonstrate
# we check the date if Hour is 12 and Minute is lower than 59
lunchbreak(){
   if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
      # and print colored \033[ starts the escape sequence
      # 5; is blinking attribute
      # 2; means bold
      # 31 says red
```

```
        printf "\033[5;1;31m注意午休时间\033[0m";else

        printf "\033[33m仍在工作...\033[0m";fi;

}

# 激活它
export PROMPT_COMMAND=lunchbreak
```

第54.2节：使用PS2

当命令延续到多行且bash等待更多按键时，会显示PS2。
当输入类似**while...do..done**的复合命令时，也会显示PS2。

```
export PS2="请完成此命令？"# 现在输入一个至少延续两行的命令以查看PS2
```

第54.3节：使用PS3

当执行select语句时，它会显示带编号的给定项目，然后显示PS3提示符：

```
export PS3="  选择你的语言请输入前面的数字 : "
select lang in EN CA FR DE; do
    # 在此检查输入直到有效。
    break
done
```

第54.4节：使用PS4

当bash处于调试模式时显示PS4。

```
#!/usr/bin/env bash

# 开启调试
set -x

# 定义一个stupid_func函数
stupid_func(){
    echo 我是stupid_func的第1行
    echo 我是stupid_func的第2行
}

# 设置PS4"DEBUG"提示符
export PS4='DEBUG 级别:$SHLVL 子shell级别: $BASH_SUBSHELL 源文件:${BASH_SOURCE}行号#:${LINENO} 函数:${FUNCNAME[0]:+${FUNCNAME[0]}(): }语句: '

# 一个普通的语句
echo something

# 函数调用
stupid_func

# 在子shell中运行的一条命令管道
( ls -l | grep 'x' )
```



```
      printf "\033[5;1;31mmind the lunch break\033[0m\n";
   else
      printf "\033[33mstill working...\033[0m\n";
   fi;
}

# activating it
export PROMPT_COMMAND=lunchbreak
```

Section 54.2: Using PS2

PS2 is displayed when a command extends to more than one line and bash awaits more keystrokes. It is displayed too when a compound command like **while...do..done** and alike is entered.

```
export PS2="would you please complete this command?\n"
# now enter a command extending to at least two lines to see PS2
```

Section 54.3: Using PS3

When the select statement is executed, it displays the given items prefixed with a number and then displays the PS3 prompt:

```
export PS3="  To choose your language type the preceding number : "
select lang in EN CA FR DE; do
   # check input here until valid.
   break
done
```

Section 54.4: Using PS4

PS4 is displayes when bash is in debugging mode.

```
#!/usr/bin/env bash

# switch on debugging
set -x

# define a stupid_func
stupid_func(){
   echo I am line 1 of stupid_func
   echo I am line 2 of stupid_func
}

# setting the PS4 "DEBUG" prompt
export PS4='\nDEBUG level:$SHLVL subshell-level: $BASH_SUBSHELL \nsource-file:${BASH_SOURCE}
line#:${LINENO} function:${FUNCNAME[0]:+${FUNCNAME[0]}(): }\nstatement: '

# a normal statement
echo something

# function call
stupid_func

# a pipeline of commands running in a subshell
( ls -l | grep 'x' )
```

第54.5节：使用PS1

PS1是普通的系统提示符，表示bash正在等待输入命令。它支持一些转义序列，并且可以执行函数或程序。由于bash需要在显示的提示符后定位光标，因此它需要知道如何计算提示符字符串的有效长度。为了表示PS1变量中非打印字符序列，使用转义大括号：\[非打印字符序列 \]。以上内容同样适用于所有PS*变量。

（黑色插入符表示光标）

```
#所有非转义序列的内容将被原样打印
export PS1="literal sequence "  # 提示符现在是：
literal sequence ■

# |u == 用户 |h == 主机 |w == 当前工作目录
# 注意单引号，避免被 shell 解释
export PS1='\u@\h:\w > ' # |u == 用户, |h == 主机, |w 当前工作目录
looser@主机:/某些/路径 > ■

# 在 PS1 中执行一些命令
# 以下行将在用户为 root 时将前景色设置为红色，
# 否则重置属性为默认
# $( (($EUID == 0)) &&  tput setaf 1)
# 之后我们用
# $(  tput sgr0 ) 重置属性为默认
# 假设是 root 用户：
PS1="\[$( (($EUID == 0)) &&  tput setaf 1 \]\u\[$(tput sgr0)\]@\w:\w \$ "
looser@主机:/某些/路径 > ■  # 如果不是 root 则显示，否则显示 <red>root<default>@主机....
```


Section 54.5: Using PS1

PS1 is the normal system prompt indicating that bash waits for commands being typed in. It understands some escape sequences and can execute functions or progams. As bash has to position the cursor after the displayes prompt, it needs to know how to calculate the effective length of the prompt string. To indicate non printing sequences of chars within the PS1 variable escaped braces are used: **\[** *a non printing sequence of chars* **\]**. All being said holds true for all PS* vars.

(The black caret indicates cursor)

```
#everything not being an escape sequence will be literally printed
export PS1="literal sequence "  # Prompt is now:
literal sequence ■

# \u == user \h == host \w == actual working directory
# mind the single quotes avoiding interpretation by shell
export PS1='\u@\h:\w > ' # \u == user, \h == host, \w actual working dir
looser@host:/some/path > ■

# executing some commands within PS1
# following line will set foreground color to red, if user==root,
# else it resets attributes to default
# $( (($EUID == 0)) &&  tput setaf 1)
# later we do reset attributes to default with
# $(  tput sgr0 )
# assuming being root:
PS1="\[$( (($EUID == 0)) &&  tput setaf 1 \]\u\[$(tput sgr0)\]@\w:\w \$ "
looser@host:/some/path > ■  # if not root else <red>root<default>@host....
```

第55章：cut命令

参数	详细信息
-f, --fields	基于字段的选择
-d, --delimiter	基于字段选择的分隔符
-c, --characters	基于字符的选择，忽略分隔符或报错
-s, --only-delimited	抑制没有分隔符字符的行（否则按原样打印）
--complement	反向选择（提取除指定字段/字符之外的所有内容）
--output-delimiter	指定输出分隔符，当其与输入分隔符不同时使用

cut命令是一种快速提取文本文件行中部分内容的方法。它属于最早的Unix命令之一。其最流行的实现版本是Linux上的GNU版本和MacOS上的FreeBSD版本，但每种Unix系统都有自己的版本。详见下文的差异说明。输入行可以从stdin读取，也可以从命令行参数中列出的文件读取。

第55.1节：仅允许一个分隔符字符

不能使用多个分隔符：如果你指定类似-d ",;:"，有些实现只会使用第一个字符作为分隔符（本例中为逗号）。其他实现（例如GNU cut）则会报错。

```
$ cut -d ",;:" -f2 <<<"J.Smith,1 Main Road,cell:1234567890;land:4081234567"
cut: the delimiter must be a single character
尝试`cut --help' 获取更多信息。
```

第55.2节：重复的分隔符被解释为空字段

```
$ cut -d, -f1,3 <<<"a,,b,c,d,e"
a,b
```

这很明显，但对于以空格分隔的字符串，有些人可能不太明显

```
$ cut -d ' ' -f1,3 <<<"a  b c d e"
a b
```

cut 不能像shell和其他程序那样解析参数。

第55.3节：无引用

无法保护分隔符。电子表格和类似的CSV处理软件通常可以识别文本引用字符，从而定义包含分隔符的字符串。使用cut则不行。

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"'
"1
```

第55.4节：提取，而非操作

您只能提取行的部分内容，不能重新排序或重复字段。

```
$ cut -d, -f2,1 <<<'John,Smith,USA' ## 就像 -f1,2 一样
```


Chapter 55: The cut command

Parameter	Details
-f, --fields	Field-based selection
-d, --delimiter	Delimiter for field-based selection
-c, --characters	Character-based selection, delimiter ignored or error
-s, --only-delimited	Suppress lines with no delimiter characters (printed as-is otherwise)
--complement	Inverted selection (extract all *except* specified fields/characters
--output-delimiter	Specify when it has to be different from the input delimiter

The `cut` command is a fast way to extract parts of lines of text files. It belongs to the oldest Unix commands. Its most popular implementations are the GNU version found on Linux and the FreeBSD version found on MacOS, but each flavor of Unix has its own. See below for differences. The input lines are read either from `stdin` or from files listed as arguments on the command line.

Section 55.1: Only one delimiter character

You cannot have more than one delimiter: if you specify something like `-d ",;:"`, some implementations will use only the first character as a delimiter (in this case, the comma.) Other implementations (e.g. GNU `cut`) will give you an error message.

```
$ cut -d ",;:" -f2 <<<"J.Smith,1 Main Road,cell:1234567890;land:4081234567"
cut: the delimiter must be a single character
Try `cut --help' for more information.
```

Section 55.2: Repeated delimiters are interpreted as empty fields

```
$ cut -d, -f1,3 <<<"a,,b,c,d,e"
a,b
```

is rather obvious, but with space-delimited strings it might be less obvious to some

```
$ cut -d ' ' -f1,3 <<<"a  b c d e"
a b
```

`cut` cannot be used to parse arguments as the shell and other programs do.

Section 55.3: No quoting

There is no way to protect the delimiter. Spreadsheets and similar CSV-handling software usually can recognize a text-quoting character which makes it possible to define strings containing a delimiter. With `cut` you cannot.

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"'
"1
```

Section 55.4: Extracting, not manipulating

You can only extract portions of lines, not reorder or repeat fields.

```
$ cut -d, -f2,1 <<<'John,Smith,USA' ## Just like -f1,2
```

```
John,Smith
$ cut -d, -f2,2 <<<'John,Smith,USA' ## 就像 -f2
Smith
```



```
John,Smith
$ cut -d, -f2,2 <<<'John,Smith,USA' ## Just like -f2
Smith
```

第56章：Windows 10上的Bash

第56.1节：自述文件

在Windows中使用Bash的更简单方法是安装Git for Windows。它附带了Git Bash，这是一个真正的Bash。你可以通过以下快捷方式访问它：

```
开始 > 所有程序 > Git > Git Bash
```

像grep、ls、find、sed、vi等命令都可以使用。

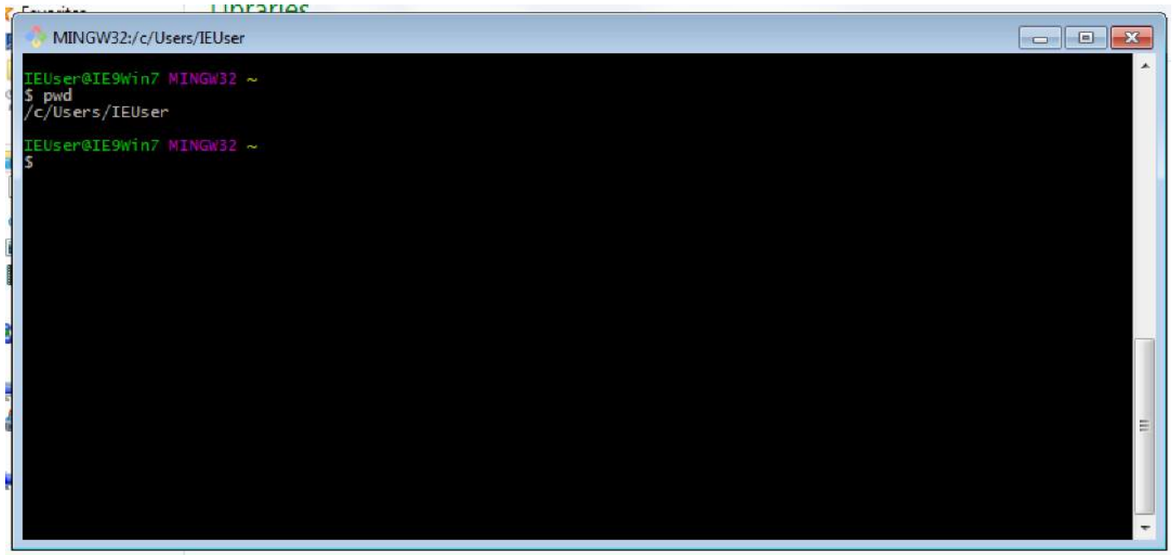
Chapter 56: Bash on Windows 10

Section 56.1: Readme

The simpler way to use Bash in Windows is to install Git for Windows. It's shipped with Git Bash which is a real Bash. You can access it with shortcut in :

```
Start > All Programs > Git > Git Bash
```

Commands like `grep`, `ls`, `find`, `sed`, `vi` etc is working.



第57章：cut命令

选项	描述
-b LIST，--bytes=LIST	打印LIST参数中列出的字节
-c LIST，--characters=LIST	打印LIST参数中指定位置的字符
-f LIST，--fields=LIST	打印字段或列
-d 分隔符	用于分隔列或字段

在Bash中，cut命令用于将文件分割成几个较小的部分。

第57.1节：显示文件的第一列

假设你有一个看起来像这样的文件

```
约翰·史密斯 31
罗伯特·琼斯 27
...
```

该文件有3列，以空格分隔。要只选择第一列，请执行以下操作。

```
cut -d ' ' -f1 filename
```

这里的-d标志指定分隔符，即记录之间的分隔符。-f标志指定字段或列号。将显示以下输出

```
约翰
罗伯特
...
```

第57.2节：显示文件的第x列到第y列

有时，显示文件中的一系列列是有用的。假设你有这个文件

```
苹果 加利福尼亚 2017 1.00 47
芒果 俄勒冈 2015 2.30 33
```

要选择前三列，执行

```
cut -d ' ' -f1-3 filename
```

这将显示以下输出

```
苹果 加利福尼亚 2017
芒果 俄勒冈 2015
```


Chapter 57: Cut Command

Option	Description
-b LIST, --bytes=LIST	Print the bytes listed in the LIST parameter
-c LIST, --characters=LIST	Print characters in positions specified in LIST parameter
-f LIST, --fields=LIST	Print fields or columns
-d DELIMITER	Used to separate columns or fields

In Bash, the `cut` command is useful for dividing a file into several smaller parts.

Section 57.1: Show the first column of a file

Suppose you have a file that looks like this

```
John Smith 31
Robert Jones 27
...
```

This file has 3 columns separated by spaces. To select only the first column, do the following.

```
cut -d ' ' -f1 filename
```

Here the -d flag, specifies the delimiter, or what separates the records. The -f flag specifies the field or column number. This will display the following output

```
John
Robert
...
```

Section 57.2: Show columns x to y of a file

Sometimes, it's useful to display a range of columns in a file. Suppose you have this file

```
Apple California 2017 1.00 47
Mango Oregon 2015 2.30 33
```

To select the first 3 columns do

```
cut -d ' ' -f1-3 filename
```

This will display the following output

```
Apple California 2017
Mango Oregon 2015
```

第58章：全局变量和局部变量

默认情况下，bash中的每个变量都是全局的，对每个函数、脚本甚至外部shell都是如此，如果你在脚本中声明变量。

如果你希望变量仅在函数内局部使用，可以使用local，使该变量成为一个新的变量，独立于全局作用域，其值只在该函数内部可访问。

第58.1节：全局变量

```
var="hello"

function foo(){
    echo $var
}

foo
```

显然会输出“hello”，但反过来也成立：

```
function foo()  {
    var="hello"
}

foo
echo $var
```

也将输出 "hello"

第58.2节：局部变量

```
function foo() {
    local var
    var="hello"
}

foo
echo $var
```

不会输出任何内容，因为var是函数foo的局部变量，其值在函数外部不可见。

第58.3节：两者混合使用

```
var="hello"

function foo(){
    local var="sup?"
    echo "函数内部，var=$var"
}

foo
echo "函数外部，var=$var"
```

将输出

Chapter 58: global and local variables

By default, every variable in bash is **global** to every function, script and even the outside shell if you are declaring your variables inside a script.

If you want your variable to be local to a function, you can use `local` to have that variable a new variable that is independent to the global scope and whose value will only be accessible inside that function.

Section 58.1: Global variables

```
var="hello"

function foo(){
    echo $var
}

foo
```

Will obviously output "hello", but this works the other way around too:

```
function foo()  {
    var="hello"
}

foo
echo $var
```

Will also output `"hello"`

Section 58.2: Local variables

```
function foo() {
    local var
    var="hello"
}

foo
echo $var
```

Will output nothing, as var is a variable local to the function foo, and its value is not visible from outside of it.

Section 58.3: Mixing the two together

```
var="hello"

function foo(){
    local var="sup?"
    echo "inside function, var=$var"
}

foo
echo "outside function, var=$var"
```

Will output

```
函数内部，变量=sup？
函数外部，变量=hello
```



```
inside function, var=sup?
outside function, var=hello
```

第59章：CGI脚本

第59.1节：请求方法：GET

通过GET调用CGI脚本非常简单。
首先你需要脚本的编码后的URL。

然后你添加一个问号?，后面跟变量。

- 每个变量应有两个部分，用=分隔。
  第一部分应始终是每个变量的唯一名称，
  而第二部分仅包含其值
- 变量由&分隔
- 字符串的总长度不应超过**255**个字符
- 名称和值需要进行HTML编码（替换：**</ , / ? : @ & = + $**）
  提示：
  使用html表单时，请求方法可以自动生成。
  使用Ajax时，可以通过encodeURI和encodeURIComponent对所有内容进行编码

示例：

```
http://www.example.com/cgi-bin/script.sh?var1=Hello%20World!&var2=This%20is%20a%20Test.&
```

服务器应仅通过**跨域资源共享**（CORS）进行通信，以提高请求的安全性。在
本示例中，我们使用**CORS**来确定要使用的数据类型。

我们可以选择多种数据类型，最常见的有...

- **text/html**
- **text/plain**
- **application/json**

发送请求时，服务器还会创建许多环境变量。目前最重要的环境变量是$REQUEST_METHOD和$QUERY_STRING。

请求方法必须是GET，别无其他！
查询字符串包括所有html编码的数据。

脚本

```
#!/bin/bash

# CORS 是通信的方式，所以先响应服务器
echo "Content-type: text/html"     # 设置我们想使用的数据类型
echo ""     # 我们不需要更多规则，空行启动此功能。

# CORS 是固定的，从现在开始的任何通信都将像读取 html 文档一样。
# 因此我们需要以 html 格式创建任何标准输出！

# 创建 html 结构并发送到标准输出
echo "<!DOCTYPE html>"
echo "<html><head>"

# 内容将根据请求方法创建
if [ "$REQUEST_METHOD" = "GET" ]; then
```


Chapter 59: CGI Scripts

Section 59.1: Request Method: GET

It is quite easy to call a CGI-Script via `GET`.
First you will need the `encoded url` of the script.

Then you add a question mark ? followed by variables.

- Every variable should have two sections separated by **=**.
  First section should be always a unique name for each variable,
  while the second part has values in it only
- Variables are separated by **&**
- Total length of the string should not rise above **255** characters
- Names and values needs to be html-encoded (replace: **</ , / ? : @ & = + $**)
  Hint:
  When using **html-forms** the request method can be generated by it self.
  With **Ajax** you can encode all via **encodeURI** and **encodeURIComponent**

Example:

```
http://www.example.com/cgi-bin/script.sh?var1=Hello%20World!&var2=This%20is%20a%20Test.&
```

The server should communicate via **Cross-Origin Resource Sharing** (CORS) only, to make request more secure. In this showcase we use **CORS** to determine the `Data-Type` we want to use.

There are many `Data-Types` we can choose from, the most common are...

- **text/html**
- **text/plain**
- **application/json**

When sending a request, the server will also create many environment variables. For now the most important environment variables are `$REQUEST_METHOD` and `$QUERY_STRING`.

The **Request Method** has to be `GET` nothing else!
The **Query String** includes all the `html-endoded data`.

The Script

```
#!/bin/bash

# CORS is the way to communicate, so lets response to the server first
echo "Content-type: text/html"    # set the data-type we want to use
echo ""    # we don't need more rules, the empty line initiate this.

# CORS are set in stone and any communication from now on will be like reading a html-document.
# Therefor we need to create any stdout in html format!

# create html scructure and send it to stdout
echo "<!DOCTYPE html>"
echo "<html><head>"

# The content will be created depending on the Request Method
if [ "$REQUEST_METHOD" = "GET" ]; then
```

```
    # 请注意，环境变量 $REQUEST_METHOD 和 $QUERY_STRING 可以被 shell 直接处理。

    # 必须过滤输入以避免跨站脚本攻击。

    Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=\([^&]*\).*$/\1/p')     # 读取 "var1" 的值
    Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))     # HTML 解码

    Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=\([^&]*\).*$/\1/p')
    Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))

    # 为标准输出创建内容
    echo "<title>Bash-CGI 示例 1</title>"
    echo "</head><body>"
    echo "<h1>Bash-CGI 示例 1</h1>"
    echo "<p>QUERY_STRING: ${QUERY_STRING}<br>var1=${Var1_Dec}<br>var2=${Var2_Dec}</p>"    # 将值打印到标准
输出

否则

    echo "<title>456 错误的请求方法</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>请求数据出错。<br>请求方法必须仅为\"GET\"！</p>"

fi

echo "<hr>"
echo "$SERVER_SIGNATURE"     # 另一个环境变量
echo "</body></html>"     # 关闭html

exit 0
```

该html文档将如下所示...

```
<html><head>
<title>Bash-CGI 示例 1</title>
</head><body>
<h1>Bash-CGI 示例 1</h1>
<p>QUERY_STRING: var1=Hello%20World!&amp;var2=This%20is%20a%20Test.&amp;<br>var1=Hello
World!<br>var2=This is a Test.</p>
<hr>
<address>Apache/2.4.10 (Debian) 服务器在 example.com 端口 80</address>
```

变量的输出将如下所示 ...

```
var1=Hello%20World!&var2=This%20is%20a%20Test.&
Hello World!
这是一个测试。
Apache/2.4.10 (Debian) 服务器在 example.com 端口 80
```

负面副作用...

- 所有的编码和解码看起来不太美观，但这是必要的
- 请求将是公开可读的，并留下一个痕迹
- 请求的大小是有限制的


```
    # Note that the environment variables $REQUEST_METHOD and $QUERY_STRING can be processed by the
shell directly.
    # One must filter the input to avoid cross site scripting.

    Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=\([^&]*\).*$/\1/p')    # read value of "var1"
    Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))    # html decode

    Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=\([^&]*\).*$/\1/p')
    Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))

    # create content for stdout
    echo "<title>Bash-CGI Example 1</title>"
    echo "</head><body>"
    echo "<h1>Bash-CGI Example 1</h1>"
    echo "<p>QUERY_STRING: ${QUERY_STRING}<br>var1=${Var1_Dec}<br>var2=${Var2_Dec}</p>"    # print
the values to stdout

else

    echo "<title>456 Wrong Request Method</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Requesting data went wrong.<br>The Request method has to be \"GET\" only!</p>"

fi

echo "<hr>"
echo "$SERVER_SIGNATURE"    # an other environment variable
echo "</body></html>"    # close html

exit 0
```

The **html-document** will look like this ...

```
<html><head>
<title>Bash-CGI Example 1</title>
</head><body>
<h1>Bash-CGI Example 1</h1>
<p>QUERY_STRING: var1=Hello%20World!&amp;var2=This%20is%20a%20Test.&amp;<br>var1=Hello
World!<br>var2=This is a Test.</p>
<hr>
<address>Apache/2.4.10 (Debian) Server at example.com Port 80</address>


</body></html>
```

The **output** of the variables will look like this ...

```
var1=Hello%20World!&var2=This%20is%20a%20Test.&
Hello World!
This is a Test.
Apache/2.4.10 (Debian) Server at example.com Port 80
```

Negative side effects...

- All the encoding and decoding does not look nice, but is needed
- The Request will be public readable and leave a tray behind
- The size of a request is limited

- 需要防护跨站脚本攻击（XSS）

第59.2节：请求方法：POST /w JSON

使用请求方法POST结合SSL使数据传输更安全。

此外...

- 大部分编码和解码已不再需要
- URL将对所有人可见，因此需要进行URL编码。
  数据将单独发送，因此应通过SSL进行保护
- 数据大小几乎无限制
- 仍需防护跨站脚本攻击（XSS）

为了保持此演示简单，我们希望接收**JSON数据**
且通信应通过**跨域资源共享**（CORS）。

下面的脚本还将演示两种不同的内容类型。

```
#!/bin/bash

exec 2>/dev/null    # 我们不希望任何错误信息打印到标准输出
trap "response_with_html && exit 0" ERR    # 发生错误时用HTML消息响应
并关闭脚本

function response_with_html(){
    echo "Content-type: text/html"
    echo ""
    echo "<!DOCTYPE html>"
    echo "<html><head>"
    echo "<title>456</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>尝试与服务器通信失败。</p>"
    echo "<hr>"
    echo "$SERVER_SIGNATURE"
    echo "</body></html>"
}

function response_with_json(){
    echo "Content-type: application/json"
    echo ""
    echo "{\"message\": \"Hello World!\"}"
}

if [ "$REQUEST_METHOD" = "POST" ]; then

    # 环境变量 $CONTENT_TYPE 描述接收到的数据类型
    case "$CONTENT_TYPE" in
application/json)
        # 环境变量 $CONTENT_LENGTH 描述读取数据的大小
        read -n "$CONTENT_LENGTH" QUERY_STRING_POST         # 读取数据流

        # 以下几行将防止 XSS 并检查有效的 JSON 数据。
        # 但这些符号在发送到此脚本之前需要以某种方式进行编码
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\$//g;s/`//g;s/\*//g;s/\\//g' )        # 移除一些符号（如 \ * ` $ '）以防止 Bash 和 SQL 中的 XSS 攻击。
```


- Needs protection against Cross-Side-Scripting (XSS)

Section 59.2: Request Method: POST /w JSON

Using Request Method `POST` in combination with `SSL` makes datatransfer more secure.

In addition...

- Most of the encoding and decoding is not needed any more
- The URL will be visible to any one and needs to be url encoded.
  The data will be send separately and therefor should be secured via SSL
- The size of the data is almost unlitmited
- Still needs protection against Cross-Side-Scripting (XSS)

To keep this showcase simple we want to receive **JSON Data**
and communication should be over **Cross-Origin Resource Sharing** (CORS).

The following script will also demonstrate two different **Content-Types**.

```
#!/bin/bash

exec 2>/dev/null    # We don't want any error messages be printed to stdout
trap "response_with_html && exit 0" ERR    # response with an html message when an error occurred
and close the script

function response_with_html(){
    echo "Content-type: text/html"
    echo ""
    echo "<!DOCTYPE html>"
    echo "<html><head>"
    echo "<title>456</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Attempt to communicate with the server went wrong.</p>"
    echo "<hr>"
    echo "$SERVER_SIGNATURE"
    echo "</body></html>"
}

function response_with_json(){
    echo "Content-type: application/json"
    echo ""
    echo "{\"message\": \"Hello World!\"}"
}

if [ "$REQUEST_METHOD" = "POST" ]; then

    # The environment variabe $CONTENT_TYPE describes the data-type received
    case "$CONTENT_TYPE" in
    application/json)
        # The environment variabe $CONTENT_LENGTH describes the size of the data
        read -n "$CONTENT_LENGTH" QUERY_STRING_POST         # read datastream

        # The following lines will prevent XSS and check for valide JSON-Data.
        # But these Symbols need to be encoded somehow before sending to this script
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\$//g;s/`//g;s/\*//g;s/\\//g' )        # removes some symbols (like \ * ` $ ') to prevent XSS
with Bash and SQL.
```

```
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;//ba')    #
移除大部分HTML声明以防止文档中的XSS攻击JSON=$(echo "$QUERY_STRING_P
        OST" | jq .)        # json编码 - 这是一种相当安全的校验有效json代码的方法

    ;;
    *)
        response_with_html
        exit 0
    ;;
    esac

否则
    response_with_html
    exit 0
fi

# 一些命令 ...

response_with_json

exit 0
```

当通过POST向此脚本发送JSON数据时，您将收到{"message":"Hello World!"}作为响应。其他所有情况将返回HTML文档。

变量$JSON也很重要。该变量不含XSS，但仍可能包含错误的值，需要先进行验证。请牢记这一点。

这段代码在没有JSON的情况下也能类似工作。
您可以通过这种方式获取任何数据。
您只需根据需要更改Content-Type即可。

示例：

```
if [ "$REQUEST_METHOD" = "POST" ]; then
    case "$CONTENT_TYPE" in
application/x-www-form-urlencoded)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    text/plain)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    ;;
    esac
fi
```

最后但同样重要的是，不要忘记响应所有请求，否则第三方程序将无法知道它们是否成功


```
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;//ba')    #
removes most html declarations to prevent XSS within documents
        JSON=$(echo "$QUERY_STRING_POST" | jq .)        # json encode - This is a pretty save way
to check for valide json code
    ;;
    *)
        response_with_html
        exit 0
    ;;
    esac

else
    response_with_html
    exit 0
fi

# Some Commands ...

response_with_json

exit 0
```

You will get `{"message":"Hello World!"}` as an answer when sending **JSON-Data** via `POST` to this Script. Every thing else will receive the html document.

Important is also the varialbe `$JSON`. This variable is free of XSS, but still could have wrong values in it and needs to be verify first. Please keep that in mind.

This code works similar without JSON.
You could get any data this way.
You just need to change the `Content-Type` for your needs.

Example:

```
if [ "$REQUEST_METHOD" = "POST" ]; then
    case "$CONTENT_TYPE" in
    application/x-www-form-urlencoded)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    text/plain)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    ;;
    esac
fi
```

Last but not least, don't forget to response to all requests, otherwise third party programms won't know if they succeeded

第60章：Select关键字

Select关键字可用于以菜单格式获取输入参数。

第60.1节：Select关键字可用于以菜单格式获取输入参数

假设你想让用户从菜单中选择关键字，我们可以创建一个类似的脚本

```
#!/usr/bin/env bash

select os in "linux" "windows" "mac"
do
    echo "${os}"
    break
done
```

说明：这里使用SELECT关键字来循环遍历一组将在命令提示符下呈现供用户选择的项目。注意break关键字，用于在用户做出选择后跳出循环。否则，循环将无限进行！

结果：运行此脚本后，将显示这些项目的菜单，并提示用户进行选择。选择后，将显示所选值，然后返回命令提示符。

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```


Chapter 60: Select keyword

Select keyword can be used for getting input argument in a menu format.

Section 60.1: Select keyword can be used for getting input argument in a menu format

Suppose you want the `user` to **`SELECT`** keywords from a menu, we can create a script similar to

```
#!/usr/bin/env bash

select os in "linux" "windows" "mac"
do
    echo "${os}"
    break
done
```

Explanation: Here **`SELECT`** keyword is used to loop through a list of items that will be presented at the command prompt for a user to pick from. Notice the **`break`** keyword for breaking out of the loop once the user makes a choice. Otherwise, the loop will be endless!

Results: Upon running this script, a menu of these items will be displayed and the user will be prompted for a selection. Upon selection, the value will be displayed, returning back to command prompt.

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```

第61章：何时使用eval

首先：要清楚自己在做什么！其次，虽然应尽量避免使用eval，但如果使用它能使代码更简洁，也可以使用。

第61.1节：使用Eval

例如，考虑以下将$@的内容设置为给定变量内容的示例：

```
a=(1 2 3)
eval set -- "${a[@]}"
```

这段代码通常与**getopt**或**getopts**一起使用，将$@设置为上述选项解析器的输出，
但是，你也可以用它来创建一个简单的pop函数，该函数可以静默且直接地操作变量，
而无需将结果存储回原始变量：

```
isnum()
{
    # 参数是否为整数？
    local re='^[0-9]+$'
    if [[ -n $1 ]]; then
        [[ $1 =~ $re ]] && return 0
        return 1
    否则
        return 2
    fi
}

isvar()
{
    if isnum "$1"; then
        return 1
    fi
    local arr="$(eval eval -- echo -n "\$$1")"
    if [[ -n ${arr[@]} ]]; then
        return 0
    fi
    return 1
}

pop()
{
    if [[ -z $@ ]]; then
        return 1
    fi

    local var=
    local isvar=0
    local arr=()

    if isvar "$1"; then # 让我们检查这是否是一个变量或仅仅是一个裸数组
        var="$1"
        isvar=1
        arr=($(eval eval -- echo -n "\${$1[@]}")) # 如果它是一个变量，获取其内容
    else
        arr=($@)
    fi
```


Chapter 61: When to use eval

First and foremost: know what you're doing! Secondly, while you should avoid using **eval**, if its use makes for cleaner code, go ahead.

Section 61.1: Using Eval

For example, consider the following that sets the contents of $@ to the contents of a given variable:

```
a=(1 2 3)
eval set -- "${a[@]}"
```

This code is often accompanied by **getopt** or **getopts** to set $@ to the output of the aforementioned option parsers, however, you can also use it to create a simple pop function that can operate on variables silently and directly without having to store the result to the original variable:

```
isnum()
{
    # is argument an integer?
    local re='^[0-9]+$'
    if [[ -n $1 ]]; then
        [[ $1 =~ $re ]] && return 0
        return 1
    else
        return 2
    fi
}

isvar()
{
    if isnum "$1"; then
        return 1
    fi
    local arr="$(eval eval -- echo -n "\$$1")"
    if [[ -n ${arr[@]} ]]; then
        return 0
    fi
    return 1
}

pop()
{
    if [[ -z $@ ]]; then
        return 1
    fi

    local var=
    local isvar=0
    local arr=()

    if isvar "$1"; then # let's check to see if this is a variable or just a bare array
        var="$1"
        isvar=1
        arr=($(eval eval -- echo -n "\${$1[@]}")) # if it is a var, get its contents
    else
        arr=($@)
    fi
```

```
    # 我们需要反转 $@ 的内容，以便将最后一个元素移除

    arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # 将 $@ 设置为 ${arr[@]}，以便对其执行 shift 操作。
    eval set -- "${arr[@]}"

    shift # 移除最后一个元素

    # 将数组恢复到原始顺序
    arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # 为了用户和裸数组的方便，输出内容
    echo "${arr[@]}"

    if ((isvar)); then
        # 将原始变量的内容设置为新的修改后数组
        eval -- "$var=(${arr[@]})"
    fi
}
```

第61.2节：使用Eval和Getopt

虽然对于类似pop的函数可能不需要eval，但每当你使用getopt时，eval是必需的：

考虑以下接受-h选项的函数：

```
f()
{
    local __me__="${FUNCNAME[0]}"
    local argv="$(getopt -o 'h' -n $__me__ -- "$@")"

    eval set -- "$argv"

    while :; do
        case "$1" in
-h)
            echo "LOLOLOLOL"
            return 0
            ;;
--)
            shift
            break
            ;;
    done

    echo "$@"
}
```

没有 `eval set -- "$argv"` 生成

```
-h --
```

而不是期望的(`-h --`)，随后进入无限循环，因为

```
-h --
```

不匹配`--`或`-h`。


```
    # we need to reverse the contents of $@ so that we can shift
    # the last element into nothingness
    arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # set $@ to ${arr[@]} so that we can run shift against it.
    eval set -- "${arr[@]}"

    shift # remove the last element

    # put the array back to its original order
    arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # echo the contents for the benefit of users and for bare arrays
    echo "${arr[@]}"

    if ((isvar)); then
        # set the contents of the original var to the new modified array
        eval -- "$var=(${arr[@]})"
    fi
}
```

Section 61.2: Using Eval with Getopt

While eval may not be needed for a pop like function, it is however required whenever you use `getopt`:

Consider the following function that accepts -h as an option:

```
f()
{
    local __me__="${FUNCNAME[0]}"
    local argv="$(getopt -o 'h' -n $__me__ -- "$@")"

    eval set -- "$argv"

    while :; do
        case "$1" in
            -h)
              echo "LOLOLOLOL"
              return 0
              ;;
            --)
              shift
              break
              ;;
    done

    echo "$@"
}
```

Without `eval set -- "$argv"` generates

```
-h --
```

instead of the desired `(-h --)` and subsequently enters an infinite loop because

```
-h --
```

doesn't match `--` or `-h`.

第62章：使用Bash进行网络操作

Bash通常用于服务器和集群的管理与维护。应包含有关网络操作中常用命令的信息，何时为哪个目的使用哪个命令，以及其独特和/或有趣应用的示例/样本

第62.1节：网络命令

```
ifconfig
```

上述命令将显示机器上所有活动的接口，并提供以下信息

1. 分配给接口的IP地址
2. 接口的MAC地址
3. 广播地址
4. 发送和接收的字节数

一些示例

```
ifconfig -a
```

上述命令也会显示禁用的接口

```
ifconfig eth0
```

上述命令只会显示 eth0 接口

```
ifconfig eth0 192.168.1.100 netmask 255.255.255.0
```

上述命令会为 eth0 接口分配静态 IP

```
ifup eth0
```

上述命令会启用 eth0 接口

```
ifdown eth0
```

以下命令将禁用 eth0 接口

```
ping
```

上述命令（数据包互联网分组器）用于测试两个节点之间的连通性

```
ping -c2 8.8.8.8
```

上述命令将对谷歌服务器进行ping测试，持续2秒钟。

```
traceroute
```

上述命令用于故障排除，以确定到达目标所经过的跳数。

```
netstat
```


Chapter 62: Networking With Bash

Bash is often commonly used in the management and maintenance of servers and clusters. Information pertaining to typical commands used by network operations, when to use which command for which purpose, and examples/samples of unique and/or interesting applications of it should be included

Section 62.1: Networking commands

```
ifconfig
```

The above command will show all active interface of the machine and also give the information of

1. IP address assign to interface
2. MAC address of the interface
3. Broadcast address
4. Transmit and Receive bytes

Some example

```
ifconfig -a
```

The above command also show the disable interface

```
ifconfig eth0
```

The above command will only show the eth0 interface

```
ifconfig eth0 192.168.1.100 netmask 255.255.255.0
```

The above command will assign the static IP to eth0 interface

```
ifup eth0
```

The above command will enable the eth0 interface

```
ifdown eth0
```

The below command will disable the eth0 interface

```
ping
```

The above command (Packet Internet Grouper) is to test the connectivity between the two nodes

```
ping -c2 8.8.8.8
```

The above command will ping or test the connectivity with google server for 2 seconds.

```
traceroute
```

The above command is to use in troubleshooting to find out the number of hops taken to reach the destination.

```
netstat
```

上述命令（网络统计）提供连接信息及其状态

```
dig www.google.com
```

上述命令（域信息聚合器）查询DNS相关信息

```
nslookup www.google.com
```

上述命令查询DNS并找出对应网站名称的IP地址。

```
route
```

上述命令用于检查网络路由信息。它基本上显示路由表

```
router add default gw 192.168.1.1 eth0
```

上述命令将在路由表中为eth0接口添加默认网络路由，网关为192.168.1.1。

```
route del default
```

上述命令将从路由表中删除默认路由

The above command (Network statistics) give the connection info and their state

```
dig www.google.com
```

The above command (domain information grouper) query the DNS related information

```
nslookup www.google.com
```

The above command query the DNS and find out the IP address of corresponding the website name.

```
route
```

The above command is used to check the Netwrok route information. It basically show you the routing table

```
router add default gw 192.168.1.1 eth0
```

The above command will add the default route of network of eth0 Interface to 192.168.1.1 in routing table.

```
route del default
```

The above command will delete the default route from the routing table

第63章：并行

选项	描述
`-j n`	并行运行 n 个作业
`-k`	保持相同顺序
`-X`	带上下文替换的多个参数
`--colsep 正则表达式`	根据正则表达式拆分输入以进行位置替换
`{} {.} {/} {/.} {#}`	替换字符串
`{3} {3.} {3/} {3/.}`	位置替换字符串
`-S ssh登录`	示例：foo@server.example.com
`--trc {}.bar`	--transfer --return {}.bar --cleanup 的简写
`--onall`	对所有 ssh 登录运行给定命令并带参数
`--nonall`	对所有 ssh 登录运行给定命令但不带参数
`--pipe`	将标准输入（stdin）拆分为多个作业。
`--recend str`	--pipe 的记录结束分隔符。
`--recstart str`	用于 --pipe 的记录起始分隔符。

GNU Linux中的任务可以使用GNU parallel进行并行处理。一个任务可以是单个命令或一个小脚本，需要针对输入中的每一行运行。典型的输入包括文件列表、主机列表、用户列表、URL列表或表格列表。任务也可以是从管道读取的命令。

第63.1节：对文件列表的重复任务进行并行处理

如果利用更多计算机资源（如CPU和内存），许多重复性任务可以更高效地完成。下面是一个并行运行多个任务的示例。

假设你有一个<文件列表>，比如来自ls的输出。同时，假设这些文件是bz2压缩的，需要按以下顺序对它们进行操作。

1. 使用bzcat将bz2文件解压到标准输出
2. 使用grep<某些关键词>筛选包含特定关键词的行
3. 将输出通过管道合并成一个压缩的gzip文件，使用gzip

使用while循环运行可能如下所示

```
filenames="file_list.txt"
while read -r line
do
name="$line"
     ## 抓取包含 puppies 的行
     bzcat $line | grep puppies | gzip >> output.gz
done < "$filenames"
```

使用 GNU Parallel，我们可以通过简单地执行以下命令同时运行 3 个并行任务

```
parallel -j 3 "bzcat {} | grep puppies" ::: $( cat filelist.txt ) | gzip > output.gz
```

该命令简单、简洁且在文件数量和文件大小较大时更高效。任务由 parallel 启动，选项 -j 3 启动 3 个并行任务，输入通过 ::: 传入并行任务。输出最终被管道传递给 gzip > output.gz

Chapter 63: Parallel

Option	Description
`-j n`	Run n jobs in parallel
`-k`	Keep same order
`-X`	Multiple arguments with context replace
`--colsep regexp`	Split input on regexp for positional replacements
`{} {.} {/} {/.} {#}`	Replacement strings
`{3} {3.} {3/} {3/.}`	Positional replacement strings
`-S sshlogin`	`Example: foo@server.example.com`
`--trc {}.bar`	Shorthand for --transfer --return {}.bar --cleanup
`--onall`	Run the given command with argument on all sshlogins
`--nonall`	Run the given command with no arguments on all sshlogins
`--pipe`	Split stdin (standard input) to multiple jobs.
`--recend str`	Record end separator for --pipe.
`--recstart str`	Record start separator for --pipe.

Jobs in GNU Linux can be parallelized using GNU parallel. A job can be a single command or a small script that has to be run for each of the lines in the input. The typical input is a list of files, a list of hosts, a list of users, a list of URLs, or a list of tables. A job can also be a command that reads from a pipe.

Section 63.1: Parallelize repetitive tasks on list of files

Many repetitive jobs can be performed more efficiently if you utilize more of your computer's resources (i.e. CPU's and RAM). Below is an example of running multiple jobs in parallel.

Suppose you have a `< list of files >`, say output from `ls`. Also, let these files are bz2 compressed and the following order of tasks need to be operated on them.

1. Decompress the bz2 files using **`bzcat`** to stdout
2. Grep (e.g. filter) lines with specific keyword(s) using **`grep`** `<some key word>`
3. Pipe the output to be concatenated into one single gzipped file using **`gzip`**

Running this using a while-loop may look like this

```
filenames="file_list.txt"
while read -r line
do
name="$line"
     ## grab lines with puppies in them
     bzcat $line | grep puppies | gzip >> output.gz
done < "$filenames"
```

Using GNU Parallel, we can run 3 parallel jobs at once by simply doing

```
parallel -j 3 "bzcat {} | grep puppies" ::: $( cat filelist.txt ) | gzip > output.gz
```

This command is simple, concise and more efficient when number of files and file size is large. The jobs gets initiated by `parallel`, option `-j 3` launches 3 parallel jobs and input to the parallel jobs is taken in by `:::`. The output is eventually piped to **`gzip`** `> output.gz`

第 63.2 节：并行化标准输入（STDIN）

现在，假设我们有一个大文件（例如 30 GB）需要逐行转换。假设我们有一个脚本，`convert.sh`，执行此 **<task>**。我们可以将该文件的内容通过管道传递给标准输入，让 parallel 以*块*的形式接收并处理，例如

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

其中 <stdin> 可以来自任何东西，例如 cat <file>。

作为一个可复现的示例，我们的任务将是 nl -n rz。取任意文件，我的文件是 data.bz2，并将其传递给 <stdin>

```
bzcat data.bz2 | nl | parallel --pipe --block 10M -k nl -n rz | gzip > ouptput.gz
```

上述示例从 bzcat data.bz2 | nl 获取 <stdin>，其中我加入了 nl 仅作为概念验证，表明最终输出 output.gz 会按接收顺序保存。然后，parallel 将 <stdin> 分割成大小为10MB的块，并对每个块通过 nl -n rz 处理，处理方式是右对齐地附加数字（详见 nl --help）。选项 --pipe 告诉 parallel 将 <stdin> 分割成多个作业，--block 指定块大小。选项 -k 指定必须保持顺序。

你的最终输出应类似于

```
000001       1  <data>
000002       2  <data>
000003       3  <data>
000004       4  <data>
000005       5  <data>
 ...
000587  552409  <data>
000588  552410  <data>
000589  552411  <data>
000590  552412  <data>
000591  552413  <data>
```

我的原始文件有552,413行。第一列表示并行作业，第二列表示传递给parallel的原始行号（分块处理）。你应该注意到第二列（以及文件的其余部分）的顺序是保持不变的。

Section 63.2: Parallelize STDIN

Now, let's imagine we have 1 large file (e.g. 30 GB) that needs to be converted, line by line. Say we have a script, `convert.sh`, that does this **<task>**. We can pipe contents of this file to stdin for parallel to take in and work with in *chunks* such as

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

where **<stdin>** can originate from anything such as `cat` **<file>**.

As a reproducible example, our task will be `nl -n rz`. Take any file, mine will be `data.bz2`, and pass it to **<stdin>**

```
bzcat data.bz2 | nl | parallel --pipe --block 10M -k nl -n rz | gzip > ouptput.gz
```

The above example takes **<stdin>** from `bzcat data.bz2 | nl`, where I included `nl` just as a proof of concept that the final output `output.gz` will be saved in the order it was received. Then, `parallel` divides the **<stdin>** into chunks of size 10 MB, and for each chunk it passes it through `nl -n rz` where it just appends a numbers rightly justified (see `nl --help` for further details). The options `--pipe` tells `parallel` to split **<stdin>** into multiple jobs and `-- block` specifies the size of the blocks. The option `-k` specifies that ordering must be maintained.

Your final output should look something like

```
000001       1  <data>
000002       2  <data>
000003       3  <data>
000004       4  <data>
000005       5  <data>
 ...
000587  552409  <data>
000588  552410  <data>
000589  552411  <data>
000590  552412  <data>
000591  552413  <data>
```

My original file had 552,413 lines. The first column represents the parallel jobs, and the second column represents the original line numbering that was passed to `parallel` in chunks. You should notice that the order in the second column (and rest of the file) is maintained.

第64章：URL解码

第64.1节：简单示例

编码的URL

> http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty

使用此命令解码URL

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9A-F]\)/\\\\\x\1/g" | xargs -0 echo -e
```

解码后的URL（命令结果）

> http://www.foo.com/index.php?id=qwerty

第64.2节：使用printf解码字符串

```
#!bin/bash

$ string='Question%20-%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-%20Use%20printf%20%3A)'
$ printf '%b' "${string//%/\\x}"# 结果


Question - "我如何解码一个百分号编码的字符串？"
Answer   - 使用 printf :)
```


Chapter 64: Decoding URL

Section 64.1: Simple example

Encoded URL

> http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty

Use this command to decode the URL

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9A-F]\)/\\\\\x\1/g" | xargs -0 echo -e
```

Decoded URL (result of command)

> http://www.foo.com/index.php?id=qwerty

Section 64.2: Using printf to decode a string

```
#!bin/bash

$ string='Question%20-%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-%20Use%20printf%20%3A)'
$ printf '%b\n' "${string//%/\\x}"

# the result
Question - "how do I decode a percent encoded string?"
Answer   - Use printf :)
```

第65章：设计模式

在Bash中实现一些常见的设计模式

第65.1节：发布/订阅（Pub/Sub）模式

当一个Bash项目变成一个库时，添加新功能可能会变得困难。函数名、变量和参数通常需要在使用它们的脚本中进行更改。在这种情况下，解耦代码并使用事件驱动设计模式是有帮助的。在该模式中，外部脚本可以订阅一个事件。当该事件被触发（发布）时，脚本可以执行它注册到该事件的代码。

pubsub.sh：

```
#!/usr/bin/env bash

#
# 将此脚本目录的路径保存到全局环境变量中
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# 将包含所有注册事件的数组
#
EVENTS=()

function action1() {
    echo "动作 #1 已执行 ${2}"
}

function action2() {
    echo "操作 #2 已执行"
}

#
# @desc   :: 注册一个事件
# @param  :: string $1 - 事件名称。基本上是函数名的别名
# @param  :: string $2 - 要调用的函数名
# @param  :: string $3 - 包含被调用函数的脚本的完整路径
#
function subscribe() {
    EVENTS+=("${1};${2};${3}")
}

#
# @desc   :: 发布一个事件
# @param  :: string $1 - 被发布的事件名称
#
function publish() {
    for event in ${EVENTS[@]}; do
        local IFS=";"
        read -r -a event <<< "$event"
        if [[  "${event[0]}" ==  "${1}" ]]; then
            ${event[1]} "$@"
        fi
    done
}

#
# 注册我们的事件及其处理函数
```


Chapter 65: Design Patterns

Accomplish some common design patterns in Bash

Section 65.1: The Publish/Subscribe (Pub/Sub) Pattern

When a Bash project turns into a library, it can become difficult to add new functionality. Function names, variables and parameters usually need to be changed in the scripts that utilize them. In scenarios like this, it is helpful to decouple the code and use an event driven design pattern. In said pattern, an external script can subscribe to an event. When that event is triggered (published) the script can execute the code that it registered with the event.

pubsub.sh:

```
#!/usr/bin/env bash

#
# Save the path to this script's directory in a global env variable
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# Array that will contain all registered events
#
EVENTS=()

function action1() {
    echo "Action #1 was performed ${2}"
}

function action2() {
    echo "Action #2 was performed"
}

#
# @desc   :: Registers an event
# @param  :: string $1 - The name of the event. Basically an alias for a function name
# @param  :: string $2 - The name of the function to be called
# @param  :: string $3 - Full path to script that includes the function being called
#
function subscribe() {
    EVENTS+=("${1};${2};${3}")
}

#
# @desc   :: Public an event
# @param  :: string $1 - The name of the event being published
#
function publish() {
    for event in ${EVENTS[@]}; do
        local IFS=";"
        read -r -a event <<< "$event"
        if [[  "${event[0]}" ==  "${1}" ]]; then
            ${event[1]} "$@"
        fi
    done
}

#
# Register our events and the functions that handle them
```

```
    #
subscribe "/do/work"           "action1" "${DIR}"
    subscribe "/do/more/work"      "action2" "${DIR}"
    subscribe "/do/even/more/work" "action1" "${DIR}"

    #
    # 执行我们的事件
    #
publish "/do/work"
    publish "/do/more/work"
    publish "/do/even/more/work" "again"
```

运行：

```
chmod +x pubsub.sh
./pubsub.sh
```



```
    #
    subscribe "/do/work"           "action1" "${DIR}"
    subscribe "/do/more/work"      "action2" "${DIR}"
    subscribe "/do/even/more/work" "action1" "${DIR}"

    #
    # Execute our events
    #
    publish "/do/work"
    publish "/do/more/work"
    publish "/do/even/more/work" "again"
```

Run:

```
chmod +x pubsub.sh
./pubsub.sh
```

第66章：陷阱

第66.1节：赋值时的空白字符

赋值时空白字符很重要。

```
foo = 'bar' # 错误
foo= 'bar'  # 错误
foo='bar'   # 正确
```

前两个会导致语法错误（或者更糟，执行错误的命令）。最后一个示例会正确地
将变量`$foo`设置为文本 "bar"。

第66.2节：失败的命令不会停止脚本执行

在大多数脚本语言中，如果函数调用失败，可能会抛出异常并停止程序执行。
Bash命令没有异常机制，但它们有退出码。非零退出码表示失败，然而，
非零退出码不会停止程序的执行。

这可能导致危险的（虽然确实是人为设计的）情况，如下所示：

```
#!/bin/bash
cd ~/不存在/的/目录
rm -rf *
```

如果cd到该目录失败，Bash会忽略失败并继续执行下一条命令，清空你运行脚本的目录。

解决此问题的最佳方法是使用set命令：

```
#!/bin/bash
set -e
cd ~/不存在/的/目录
rm -rf *
```

set -e 告诉Bash如果任何命令返回非零状态，则立即退出脚本。

第66.3节：文件中缺少最后一行

C标准规定文件应以换行符结尾，因此如果EOF出现在一行的末尾，该行可能会被某些命令忽略。例如：

```
$ echo 'onetwothree\c' > file.txt$ cat file.txt


one
two
three

$ while read line ; do echo "line $line" ; done < file.txt
one
two
```

为了确保上述示例中的操作正确执行，添加一个测试，使得如果最后一行不为空，循环将继续。

Chapter 66: Pitfalls

Section 66.1: Whitespace When Assigning Variables

Whitespace matters when assigning variables.

```
foo = 'bar' # incorrect
foo= 'bar'  # incorrect
foo='bar'   # correct
```

The first two will result in syntax errors (or worse, executing an incorrect command). The last example will correctly set the variable `$foo` to the text "bar".

Section 66.2: Failed commands do not stop script execution

In most scripting languages, if a function call fails, it may throw an exception and stop execution of the program. Bash commands do not have exceptions, but they do have exit codes. A non-zero exit code signals failure, however, a non-zero exit code will not stop execution of the program.

This can lead to dangerous (although admittedly contrived) situations like so:

```
#!/bin/bash
cd ~/non/existent/directory
rm -rf *
```

If `cd`-ing to this directory fails, Bash will ignore the failure and move onto the next command, wiping clean the directory from where you ran the script.

The best way to deal with this problem is to make use of the set command:

```
#!/bin/bash
set -e
cd ~/non/existent/directory
rm -rf *
```

`set -e` tells Bash to exit the script immediately if any command returns a non-zero status.

Section 66.3: Missing The Last Line in a File

The C standard says that files should end with a new line, so if EOF comes at the end of a line, that line may not be missed by some commands. As an example:

```
$ echo 'one\ntwo\nthree\c' > file.txt

$ cat file.txt
one
two
three

$ while read line ; do echo "line $line" ; done < file.txt
one
two
```

To make sure this works correctly for in the above example, add a test so that it will continue the loop if the last line is not empty.

```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
二
三
```



```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
two
three
```

附录 A：键盘快捷键

A.1 节：编辑快捷键

快捷键	描述
Ctrl + a	移动到行首
Ctrl + e	移动到行尾
Ctrl + k	删除从当前光标位置到行尾的文本。
Ctrl + u	删除从当前光标位置到行首的文本
Ctrl + w	删除当前光标位置后面的单词
Alt + b	向后移动一个单词
Alt + f	向前移动一个单词
Ctrl + Alt + e	shell 扩展行
Ctrl + y	将最近删除的文本粘回光标处的缓冲区。
Alt + y	在已删除的文本中旋转。只有在前一个命令是时，您才能执行此操作 Ctrl + y 或者 Alt + y.

杀死文本将删除文本，但会保存它，以便用户可以通过粘贴(yank)将其重新插入。类似于剪切和粘贴，但文本被放置在一个杀死环(kill ring)中，这允许存储多组文本以便在命令行上重新粘贴。

您可以在emacs手册中了解更多信息。

A.2节：回忆快捷键

快捷键	描述
Ctrl + r	向后搜索历史记录
Ctrl + p	历史中的上一个命令
Ctrl + n	历史中的下一个命令
Ctrl + g	退出历史搜索模式
Alt + .	使用上一个命令的最后一个单词
	重复以获取上一个加一命令的最后一个单词
Alt + n Alt + .	使用上一个命令的第n个单词
!! + 返回	再次执行上一个命令（当你忘记使用sudo时很有用：sudo !!）

A.3节：宏

快捷键	描述
Ctrl + x, (	开始录制宏
Ctrl + x,)	停止录制宏
Ctrl + x, e	执行最后录制的宏

A.4节：自定义键绑定

使用bind命令可以定义自定义键绑定。

下面的示例绑定了一个 Alt + w 到>/dev/null 2>&1:

```
bind '"\ew":"\" >/dev/null 2>&1\""
```


Appendix A: Keyboard shortcuts

Section A.1: Editing Shortcuts

Shortcut	Description
Ctrl + a	move to the beginning of the line
Ctrl + e	move to the end of the line
Ctrl + k	Kill the text from the current cursor position to the end of the line.
Ctrl + u	Kill the text from the current cursor position to the beginning of the line
Ctrl + w	Kill the word behind the current cursor position
Alt + b	move backward one word
Alt + f	move forward one word
Ctrl + Alt + e	shell expand line
Ctrl + y	Yank the most recently killed text back into the buffer at the cursor.
Alt + y	Rotate through killed text. You can only do this if the prior command is Ctrl + y or Alt + y.

Killing text will delete text, but save it so that the user can reinsert it by yanking. Similar to cut and paste except that the text is placed on a kill ring which allows for storing more than one set of text to be yanked back on to the command line.

You can find out more in the emacs manual.

Section A.2: Recall Shortcuts

Shortcut	Description
Ctrl + r	search the history backwards
Ctrl + p	previous command in history
Ctrl + n	next command in history
Ctrl + g	quit history searching mode
Alt + .	use the last word of the previous command
	repeat to get the last word of the previous + 1 command
Alt + n Alt + .	use the nth word of the previous command
!! + Return	execute the last command again (useful when you forgot sudo: `sudo !!`)

Section A.3: Macros

Shortcut	Description
Ctrl + x, (	start recording a macro
Ctrl + x,)	stop recording a macro
Ctrl + x, e	execute the last recorded macro

Section A.4: Custome Key Bindings

With the `bind` command it is possible to define custom key bindings.

The next example bind an Alt + w to `>/dev/null 2>&1`:

```
bind '"\ew":"\" >/dev/null 2>&1\""
```

如果想立即执行该行，请添加\C-m(输入)到它：

```
bind '"\ew":"\" >/dev/null 2>&1\C-m\""
```

A.5节：作业控制

快捷键	描述
Ctrl + c	停止当前作业
Ctrl + z	挂起当前作业（发送SIGTSTP信号）

If you want to execute the line immediately add \C-m (Enter) to it:

```
bind '"\ew":"\" >/dev/null 2>&1\C-m\""
```

Section A.5: Job Control

Shortcut	Description
Ctrl + c	Stop the current job
Ctrl + z	Suspend the current job (send a SIGTSTP signal)

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

阿贾伊·桑加莱	第一章
阿金卡	第20章
亚历山德罗·马斯科洛	第11章和第26章
阿列克谢·马古拉	第9章、第12章、第36章和第61章
阿米尔·拉楚姆	第8章
阿尼尔	第一章
anishsane	第5章
安托万·博尔维	第9章
阿尔切马尔	第9章
阿罗尼卡尔	第12章
阿沙里	第36章
阿什坎	第36章和第43章
巴茨	第17章
本杰明·W.	第1、9、12、15、24、31、36、46和47章
binki	第21章
Blachshma	第一章
鲍勃·巴格威尔	第一章
博斯蒂安	第7章
BrunoLM	第14章
布赖登·吉布森	第9章
泡泡弹	第1、5、8、12和24章
布尔哈德	第一章
BurnsBA	第22章
地毯吸烟者	第47章
cb0	第28章
钱德拉哈斯·阿鲁里	第6章
混沌	第9章
查尼凯	第50章
切普纳	第15、27和46章
克里斯·拉西斯	第34章
克里斯托弗·博托姆斯	第1、3和5章
codeforester	第12章
Cody	第66章
杨科林	第一章
牛叫声	第30章
CraftedCart	第一章
CrazyMax	第64章
criw	第36章
丹尼尔·凯弗	第67章
丹尼	第一章
达里奥	第28、36和55章
大卫·格雷森	第9章
迪帕克·K·M	第20章
deepmax	第25章
depperm	第4章和第35章
dhimanta	第62章
dimo414	第14章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Ajay Sangale	Chapter 1
Ajinkya	Chapter 20
Alessandro Mascolo	Chapters 11 and 26
Alexej Magura	Chapters 9, 12, 36 and 61
Amir Rachum	Chapter 8
Anil	Chapter 1
anishsane	Chapter 5
Antoine Bolvy	Chapter 9
Archemar	Chapter 9
Arronical	Chapter 12
Ashari	Chapter 36
Ashkan	Chapters 36 and 43
Batsu	Chapter 17
Benjamin W.	Chapters 1, 9, 12, 15, 24, 31, 36, 46 and 47
binki	Chapter 21
Blachshma	Chapter 1
Bob Bagwill	Chapter 1
Bostjan	Chapter 7
BrunoLM	Chapter 14
Brydon Gibson	Chapter 9
Bubblepop	Chapters 1, 5, 8, 12 and 24
Burkhard	Chapter 1
BurnsBA	Chapter 22
Carpetsmoker	Chapter 47
cb0	Chapter 28
Chandrahas Aroori	Chapter 6
chaos	Chapter 9
charneykaye	Chapter 50
chepner	Chapters 15, 27 and 46
Chris Rasys	Chapter 34
Christopher Bottoms	Chapters 1, 3 and 5
codeforester	Chapter 12
Cody	Chapter 66
Colin Yang	Chapter 1
Cows quack	Chapter 30
CraftedCart	Chapter 1
CrazyMax	Chapter 64
criw	Chapter 36
Daniel Käfer	Chapter 67
Danny	Chapter 1
Dario	Chapters 28, 36 and 55
David Grayson	Chapter 9
Deepak K M	Chapter 20
deepmax	Chapter 25
depperm	Chapters 4 and 35
dhimanta	Chapter 62
dimo414	Chapter 14

dingalapadum	第7章和第16章
divyum	第1章和第14章
DocSalvager	第10章
Doctor J	第28章
DonyorM	第10章
Dr Beco	第36章
Dunatotatos	第51章
Echoes_86	第17章
埃德加·罗克jān	第10章
edi9999	第14章
埃里克·雷努夫	第9章
fedorqui	第12、15、17、20、28和34章
fifaltra	第8和53章
流量	第18章
加文	第9、26、33和36章
乔治·瓦西里乌	第9、15和58章
吉勒斯	第21章和第22章
格伦·杰克曼	第1、4、5和7章
格雷克西斯	第15章
格里沙·列维特	第36章
gzh	第10章
hedgar2017	第9、15和22章
霍尔特·约翰逊	第4章
I0_ol	第64章
伊恩	第4章和第20章
IamaTacos	第35章
伊南奇·古穆斯	第一章
伊尼安	第17章和第28章
intboolstring	第4章、第5章和第7章
贾希德	第1、5、9、10、12、14、15、17、20、21、22、23、30、34、39、43、44和45章
詹姆斯·泰勒	第23章
杰米·梅茨格	第31章
jandob	第29章
janos	第7、10、12、14、20和24章
杰弗里·林	第49章
JepZ	第3章
jerblack	第12章
陈杰西	第15、26和45章
JHS	第7、19和67章
jimsug	第24章
约翰·库格尔曼	第12章
乔恩	第63章
乔恩·埃里克森	第9章
乔尼·亨利	第4章
乔迪	第48章
贾德·罗杰斯	第9章和第67章
凯鲁姆·塞纳纳亚克	第23章
ksoni	第30章
leftaroundabout	第17章
列奥·乌菲姆采夫	第33章
liborm	第9章
lynxlynxlynx	第43章
m02ph3u5	第67章

dingalapadum	Chapters 7 and 16
divyum	Chapters 1 and 14
DocSalvager	Chapter 10
Doctor J	Chapter 28
DonyorM	Chapter 10
Dr Beco	Chapter 36
Dunatotatos	Chapter 51
Echoes_86	Chapter 17
Edgar Rokjān	Chapter 10
edi9999	Chapter 14
Eric Renouf	Chapter 9
fedorqui	Chapters 12, 15, 17, 20, 28 and 34
fifaltra	Chapters 8 and 53
Flows	Chapter 18
Gavyn	Chapters 9, 26, 33 and 36
George Vasiliou	Chapters 9, 15 and 58
Gilles	Chapters 21 and 22
glenn jackman	Chapters 1, 4, 5 and 7
Grexis	Chapter 15
Grisha Levit	Chapter 36
gzh	Chapter 10
hedgar2017	Chapters 9, 15 and 22
Holt Johnson	Chapter 4
I0_ol	Chapter 64
Iain	Chapters 4 and 20
IamaTacos	Chapter 35
Inanc Gumus	Chapter 1
Inian	Chapters 17 and 28
intboolstring	Chapters 4, 5 and 7
Jahid	Chapters 1, 5, 9, 10, 12, 14, 15, 17, 20, 21, 22, 23, 30, 34, 39, 43, 44 and 45
James Taylor	Chapter 23
Jamie Metzger	Chapter 31
jandob	Chapter 29
janos	Chapters 7, 10, 12, 14, 20 and 24
Jeffrey Lin	Chapter 49
JepZ	Chapter 3
jerblack	Chapter 12
Jesse Chen	Chapters 15, 26 and 45
JHS	Chapters 7, 19 and 67
jimsug	Chapter 24
John Kugelman	Chapter 12
Jon	Chapter 63
Jon Ericson	Chapter 9
Jonny Henly	Chapter 4
jordi	Chapter 48
Judd Rogers	Chapters 9 and 67
Kelum Senanayake	Chapter 23
ksoni	Chapter 30
leftaroundabout	Chapter 17
Leo Ufimtsev	Chapter 33
liborm	Chapter 9
lynxlynxlynx	Chapter 43
m02ph3u5	Chapter 67

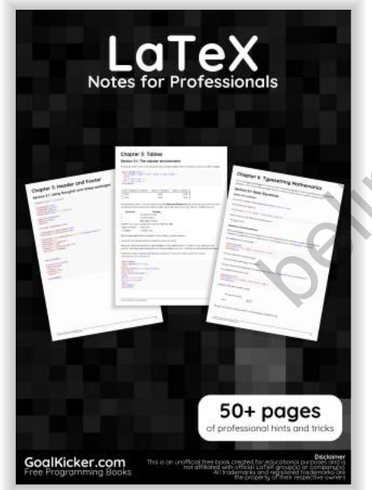
markjwill	第12章
马库斯·V.	第4章
马泰乌什·皮奥特罗夫斯基	第12章
马特·克拉克	第1、9、14、17、19和23章
mattmc	第36和65章
迈克尔·勒·巴尔比耶格吕内瓦尔德	第14章
迈克·梅茨格	第8章
miken32	第9和10章
米萨·拉佐维奇	第4章和第30章
莫希玛·乔杜里	第18章和第41章
航海的	第34章
尼尔·王	第12章
Neui	第8章
Ocab19	第58章
ormaaj	第12章
大阪	第4章
签名	第38章
帕维尔·卡热维茨	第25章
彼得·乌纳克	第31章
phs	第47章
普扬·霍斯拉维	第9章
拉法·莫亚诺	第42章
重启	第42章
里卡多·佩特拉利亚	第8章
理查德·汉密尔顿	第4、16、41和57章
赖克尔	第1和40章
罗曼·皮塔克	第47章
鲁特	第5、8和9章
萨米尔·斯里瓦斯塔瓦	第8章
萨米克	第4、5、10、12、14和37章
塞缪尔	第5章
萨基布·罗卡迪亚	第67章
萨蒂亚纳拉扬·拉奥	第一章
斯克罗夫	第66章
谢尔盖	第14章
sjsam	第1章和第32章
Sk606	第8章、第12章和第33章
天网	第45章
SLePort	第5章和第10章
斯特凡·沙泽拉斯	第15章和第36章
斯托博尔	第20章
苏莱曼	第59章
桑迪普	第一章
西尔万·布加特	第2、4、9、14和15章
托马斯·尚皮翁	第56章
蒂姆·里亚维克	第25章
准时汤姆	第47章
特雷弗·克拉克	第一章
tripleee	第1、5、14、17和36章
tversteeg	第30章
uhelp	第2、7、13、20、31、36、47、48、52、53和54章
U·纳加斯瓦米	第12、13和60章

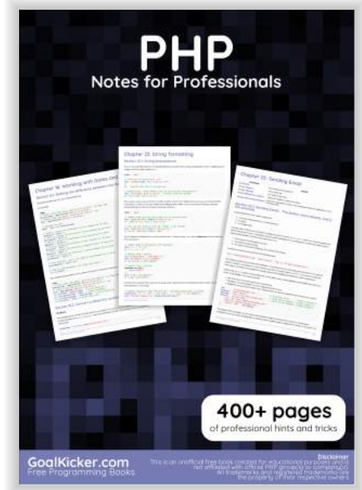
markjwill	Chapter 12
Markus V.	Chapter 4
Mateusz Piotrowski	Chapter 12
Matt Clark	Chapters 1, 9, 14, 17, 19 and 23
mattmc	Chapters 36 and 65
Michael Le Barbier Grünewald	Chapter 14
Mike Metzger	Chapter 8
miken32	Chapters 9 and 10
Misa Lazovic	Chapters 4 and 30
Mohima Chaudhuri	Chapters 18 and 41
nautical	Chapter 34
NeilWang	Chapter 12
Neui	Chapter 8
Ocab19	Chapter 58
ormaaj	Chapter 12
Osaka	Chapter 4
P.P.	Chapter 38
Pavel Kazhevets	Chapter 25
Peter Uhnak	Chapter 31
phs	Chapter 47
Pooyan Khosravi	Chapter 9
Rafa Moyano	Chapter 42
Reboot	Chapter 42
Riccardo Petraglia	Chapter 8
Richard Hamilton	Chapters 4, 16, 41 and 57
Riker	Chapters 1 and 40
Roman Piták	Chapter 47
Root	Chapters 5, 8 and 9
Sameer Srivastava	Chapter 8
Samik	Chapters 4, 5, 10, 12, 14 and 37
Samuel	Chapter 5
Saqib Rokadia	Chapter 67
satyanarayan rao	Chapter 1
Scroff	Chapter 66
Sergey	Chapter 14
sjsam	Chapters 1 and 32
Sk606	Chapters 8, 12 and 33
Skynet	Chapter 45
SLePort	Chapters 5 and 10
Stephane Chazelas	Chapters 15 and 36
Stobor	Chapter 20
suleiman	Chapter 59
Sundeep	Chapter 1
Sylvain Bugat	Chapters 2, 4, 9, 14 and 15
Thomas Champion	Chapter 56
Tim Rijavec	Chapter 25
TomOnTime	Chapter 47
Trevor Clarke	Chapter 1
tripleee	Chapters 1, 5, 14, 17 and 36
tversteeg	Chapter 30
uhelp	Chapters 2, 7, 13, 20, 31, 36, 47, 48, 52, 53 and 54
UNagaswamy	Chapters 12, 13 and 60

user1336087	第1章和第26章
vielmetti	第5章
vmaroli	第39章
沃伦·哈珀	第9章
文中	第30章
威尔	第12章、第15章和第21章
威尔·巴恩韦尔	第24章
威廉·珀塞尔	第1章、第36章和第49章
沃伊切赫·卡齐奥尔	第36章
沃尔夫冈	第9章
xhienne	第5章
ymbirtt	第15章
zarak	第8章、第24章和第31章
Zaz	第一章
Мона_Сах	第28章
南山竹	第1、5、9、12和17章

user1336087	Chapters 1 and 26
vielmetti	Chapter 5
vmaroli	Chapter 39
Warren Harper	Chapter 9
Wenzhong	Chapter 30
Will	Chapters 12, 15 and 21
Will Barnwell	Chapter 24
William Pursell	Chapters 1, 36 and 49
Wojciech Kazior	Chapter 36
Wolfgang	Chapter 9
xhienne	Chapter 5
ymbirtt	Chapter 15
zarak	Chapters 8, 24 and 31
Zaz	Chapter 1
Мона_Сах	Chapter 28
南山竹	Chapters 1, 5, 9, 12 and 17

你可能也喜欢





You may also like