第二十七章 自顶向下设计

As programs get larger and more complex, they become more difficult to design, code and maintain. As with any large project, it is often a good idea to break large, complex tasks into a series of small, simple tasks. Let’s imagine that we are trying to describe a common, everyday task, going to the market to buy food, to a person from Mars. We might describe the overall process as the following series of steps:

随着程序变得更加庞大和复杂，设计，编码和维护它们也变得更加困难。对于任意一个大项目而言， 把繁重，复杂的任务分割为细小且简单的任务，往往是一个好主意。想象一下，我们试图描述 一个平凡无奇的工作，一位火星人要去市场买食物。我们可能通过下面一系列步骤来形容整个过程：

* Get in car.
* Drive to market.
* Park car.
* Enter market.
* Purchase food.
* Return to car.
* Drive home.
* Park car.
* Enter house.
* 上车
* 开车到市场
* 停车
* 买食物
* 回到车中
* 开车回家
* 回到家中

However, a person from Mars is likely to need more detail. We could further break down the subtask “Park car” into this series of steps:

然而，火星人可能需要更详细的信息。我们可以进一步细化子任务“停车”为这些步骤：

* Find parking space.
* Drive car into space.
* Turn off motor.
* Set parking brake.
* Exit car.
* Lock car.
* 找到停车位
* 开车到停车位
* 关闭引擎
* 拉紧手刹
* 下车
* 锁车

The “Turn off motor” subtask could further be broken down into steps including “Turn off ignition,” “Remove ignition key” and so on, until every step of the entire process of going to the market has been fully defined.

这个“关闭引擎”子任务可以进一步细化为这些步骤，包括“关闭点火装置”，“移开点火匙”等等，直到 已经完整定义了要去市场买食物整个过程的每一个步骤。

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called top-down design. This technique allows us to break large complex tasks into many small, simple tasks. Top-down design is a common method of designing programs and one that is well suited to shell programming in particular.

这种先确定上层步骤，然后再逐步细化这些步骤的过程被称为自顶向下设计。这种技巧允许我们 把庞大而复杂的任务分割为许多小而简单的任务。自顶向下设计是一种常见的程序设计方法， 尤其适合 shell 编程。

In this chapter, we will use top-down design to further develop our report generator script.

在这一章中，我们将使用自顶向下的设计方法来进一步开发我们的报告产生器脚本。

### Shell 函数

Our script currently performs the following steps to generate the HTML document:

目前我们的脚本执行以下步骤来产生这个 HTML 文档：

* Open page.
* Open page header.
* Set page title.
* Close page header.
* Open page body.
* Output page heading.
* Output time stamp.
* Close page body.
* Close page.
* 打开网页
* 打开网页标头
* 设置网页标题
* 关闭网页标头
* 打开网页主体部分
* 输出网页标头
* 输出时间戳
* 关闭网页主体
* 关闭网页

For our next stage of development, we will add some additional tasks between steps 7 and 8. These will include:

为了下一阶段的开发，我们将在步骤7和8之间添加一些额外的任务。这些将包括：

* System uptime and load. This is the amount of time since the last shutdown or reboot and the average number of tasks currently running on the processor over several time intervals.
* Disk space. The overall use of space on the system’s storage devices.
* Home space. The amount of storage space being used by each user.
* 系统正常运行时间和负载。这是自上次关机或重启之后系统的运行时间，以及在几个时间间隔内当前运行在处理 中的平均任务量。
* 磁盘空间。系统中存储设备的总使用量。
* 家目录空间。每个用户所使用的存储空间数量。

If we had a command for each of these tasks, we could add them to our script simply through command substitution:

如果对于每一个任务，我们都有相应的命令，那么通过命令替换，我们就能很容易地把它们添加到我们的脚本中：

#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

CURRENT\_TIME=$(date +"%x %r %Z")

TIME\_STAMP="Generated $CURRENT\_TIME, by $USER"

cat << \_EOF\_

<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

<P>$TIME\_STAMP</P>

$(report\_uptime)

$(report\_disk\_space)

$(report\_home\_space)

</BODY>

</HTML>

\_EOF\_

We could create these additional commands two ways. We could write three separate scripts and place them in a directory listed in our PATH, or we could embed the scripts within our program as shell functions. As we have mentioned before, shell functions are “mini-scripts” that are located inside other scripts and can act as autonomous programs. Shell functions have two syntactic forms:

我们能够用两种方法来创建这些额外的命令。我们可以分别编写三个脚本，并把它们放置到 环境变量 PATH 所列出的目录下，或者我们也可以把这些脚本作为 shell 函数嵌入到我们的程序中。 我们之前已经提到过，shell 函数是位于其它脚本中的“微脚本”，作为自主程序。Shell 函数有两种语法形式：

function name {

commands

return

}

and

name () {

commands

return

}

where **name** is the name of the function and **commands** are a series of commands contained within the function.

这里的 name 是函数名，commands 是一系列包含在函数中的命令。

Both forms are equivalent and may be used interchangeably. Below we see a script that demonstrates the use of a shell function:

两种形式是等价的，可以交替使用。下面我们将查看一个说明 shell 函数使用方法的脚本：

1 #!/bin/bash

2

3 # Shell function demo

4

5 function funct {

6 echo "Step 2"

7 return

8 }

9

10 # Main program starts here

11

12 echo "Step 1"

13 funct

14 echo "Step 3"

As the shell reads the script, it passes over lines 1 through 11, as those lines consist of comments and the function definition. Execution begins at line 12, with an echo command. Line 13 calls the shell function funct and the shell executes the function just as it would any other command. Program control then moves to line 6, and the second echo command is executed. Line 7 is executed next. Its return command terminates the function and returns control to the program at the line following the function call (line 14), and the final echo command is executed. Note that in order for function calls to be recognized as shell functions and not interpreted as the names of external programs, shell function definitions must appear in the script before they are called.

随着 shell 读取这个脚本，它会跳过第1行到第11行的代码，因为这些文本行由注释和函数定义组成。 从第12行代码开始执行，有一个 echo 命令。第13行会调用 shell 函数 funct，然后 shell 会执行这个函数， 就如执行其它命令一样。这样程序控制权会转移到第六行，执行第二个 echo 命令。然后再执行第7行。 这个 return 命令终止这个函数，并把控制权交给函数调用之后的代码（第14行），从而执行最后一个 echo 命令。注意为了使函数调用被识别出是 shell 函数，而不是被解释为外部程序的名字，所以在脚本中 shell 函数定义必须出现在函数调用之前。

We’ll add minimal shell function definitions to our script:

我们将给脚本添加最小的 shell 函数定义：

#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

CURRENT\_TIME=$(date +"%x %r %Z")

TIME\_STAMP="Generated $CURRENT\_TIME, by $USER"

report\_uptime () {

return

}

report\_disk\_space () {

return

}

report\_home\_space () {

return

}

cat << \_EOF\_

<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

<P>$TIME\_STAMP</P>

$(report\_uptime)

$(report\_disk\_space)

$(report\_home\_space)

</BODY>

</HTML>

\_EOF\_

Shell function names follow the same rules as variables. A function must contain at least one command. The return command (which is optional) satisfies the requirement.

Shell 函数的命名规则和变量一样。一个函数必须至少包含一条命令。这条 return 命令（是可选的）满足要求。

### 局部变量

In the scripts we have written so far, all the variables (including constants) have been global variables. Global variables maintain their existence throughout the program. This is fine for many things, but it can sometimes complicate the use of shell functions. Inside shell functions, it is often desirable to have local variables. Local variables are only accessible within the shell function in which they are defined and cease to exist once the shell function terminates.

目前我们所写的脚本中，所有的变量（包括常量）都是全局变量。全局变量在整个程序中保持存在。 对于许多事情来说，这很好，但是有时候它会使 shell 函数的使用变得复杂。在 shell 函数中，经常期望 会有局部变量。局部变量只能在定义它们的 shell 函数中使用，并且一旦 shell 函数执行完毕，它们就不存在了。

Having local variables allows the programmer to use variables with names that may already exist, either in the script globally or in other shell functions, without having to worry about potential name conflicts.

拥有局部变量允许程序员使用的局部变量名，可以与已存在的变量名相同，这些变量可以是全局变量， 或者是其它 shell 函数中的局部变量，却不必担心潜在的名字冲突。

Here is an example script that demonstrates how local variables are defined and used:

这里有一个实例脚本，其说明了怎样来定义和使用局部变量：

#!/bin/bash

# local-vars: script to demonstrate local variables

foo=0 # global variable foo

funct\_1 () {

local foo # variable foo local to funct\_1

foo=1

echo "funct\_1: foo = $foo"

}

funct\_2 () {

local foo # variable foo local to funct\_2

foo=2

echo "funct\_2: foo = $foo"

}

echo "global: foo = $foo"

funct\_1

echo "global: foo = $foo"

funct\_2

echo "global: foo = $foo"

As we can see, local variables are defined by preceding the variable name with the word local. This creates a variable that is local to the shell function in which it is defined. Once outside the shell function, the variable no longer exists. When we run this script, we see the results:

正如我们所看到的，通过在变量名之前加上单词 local，来定义局部变量。这就创建了一个只对其所在的 shell 函数起作用的变量。在这个 shell 函数之外，这个变量不再存在。当我们运行这个脚本的时候， 我们会看到这样的结果：

[me@linuxbox ~]$ local-vars

global: foo = 0

funct\_1: foo = 1

global: foo = 0

funct\_2: foo = 2

global: foo = 0

We see that the assignment of values to the local variable foo within both shell functions has no effect on the value of foo defined outside the functions.

我们看到对两个 shell 函数中的局部变量 foo 赋值，不会影响到在函数之外定义的变量 foo 的值。

This feature allows shell functions to be written so that they remain independent of each other and of the script in which they appear. This is very valuable, as it helps prevent one part of a program from interfering with another. It also allows shell functions to be written so that they can be portable. That is, they may be cut and pasted from script to script, as needed.

这个功能就允许 shell 函数能保持各自以及与它们所在脚本之间的独立性。这个非常有价值，因为它帮忙 阻止了程序各部分之间的相互干涉。这样 shell 函数也可以移植。也就是说，按照需求， shell 函数可以在脚本之间进行剪切和粘贴。

### 保持脚本运行

While developing our program, it is useful to keep the program in a runnable state. By doing this, and testing frequently, we can detect errors early in the development process. This will make debugging problems much easier. For example, if we run the program, make a small change, then run the program again and find a problem, it’s very likely that the most recent change is the source of the problem. By adding the empty functions, called stubs in programmer-speak, we can verify the logical flow of our program at an early stage. When constructing a stub, it’s a good idea to include something that provides feedback to the programmer, which shows the logical flow is being carried out. If we look at the output of our script now:

当开发程序的时候，保持程序的可执行状态非常有用。这样做，并且经常测试，我们就可以在程序 开发过程的早期检测到错误。这将使调试问题容易多了。例如，如果我们运行这个程序，做一个小的修改， 然后再次执行这个程序，最后发现一个问题，非常有可能这个最新的修改就是问题的来源。通过添加空函数， 程序员称之为占位符，我们可以在早期阶段证明程序的逻辑流程。当构建一个占位符的时候， 能够包含一些为程序员提供反馈信息的代码是一个不错的主意，这些信息展示了正在执行的逻辑流程。 现在看一下我们脚本的输出结果：

[me@linuxbox ~]$ sys\_info\_page

<HTML>

<HEAD>

<TITLE>System Information Report For twin2</TITLE>

</HEAD>

<BODY>

<H1>System Information Report For linuxbox</H1>

<P>Generated 03/19/2009 04:02:10 PM EDT, by me</P>

</BODY>

</HTML>

we see that there are some blank lines in our output after the time stamp, but we can’t be sure of the cause. If we change the functions to include some feedback:

我们看到时间戳之后的输出结果中有一些空行，但是我们不能确定这些空行产生的原因。如果我们 修改这些函数，让它们包含一些反馈信息：

report\_uptime () {

echo "Function report\_uptime executed."

return

}

report\_disk\_space () {

echo "Function report\_disk\_space executed."

return

}

report\_home\_space () {

echo "Function report\_home\_space executed."

return

}

and run the script again:

然后再次运行这个脚本：

[me@linuxbox ~]$ sys\_info\_page

<HTML>

<HEAD>

<TITLE>System Information Report For linuxbox</TITLE>

</HEAD>

<BODY>

<H1>System Information Report For linuxbox</H1>

<P>Generated 03/20/2009 05:17:26 AM EDT, by me</P>

Function report\_uptime executed.

Function report\_disk\_space executed.

Function report\_home\_space executed.

</BODY>

</HTML>

we now see that, in fact, our three functions are being executed.

现在我们看到，事实上，执行了三个函数。

With our function framework in place and working, it’s time to flesh out some of the function code. First, the report\_uptime function:

我们的函数框架已经各就各位并且能工作，是时候更新一些函数代码了。首先，是 report\_uptime 函数：

report\_uptime () {

cat <<- \_EOF\_

<H2>System Uptime</H2>

<PRE>$(uptime)</PRE>

\_EOF\_

return

}

It’s pretty straightforward. We use a here document to output a section header and the output of the uptime command, surrounded by <PRE> tags to preserve the formatting of the command. The report\_disk\_space function is similar:

这些代码相当直截了当。我们使用一个 here 文档来输出标题和 uptime 命令的输出结果，命令结果被 <PRE> 标签包围， 为的是保持命令的输出格式。这个 report\_disk\_space 函数类似：

report\_disk\_space () {

cat <<- \_EOF\_

<H2>Disk Space Utilization</H2>

<PRE>$(df -h)</PRE>

\_EOF\_

return

}

This function uses the df -h command to determine the amount of disk space. Lastly, we’ll build the report\_home\_space function:

这个函数使用 df -h 命令来确定磁盘空间的数量。最后，我们将建造 report\_home\_space 函数：

report\_home\_space () {

cat <<- \_EOF\_

<H2>Home Space Utilization</H2>

<PRE>$(du -sh /home/\*)</PRE>

\_EOF\_

return

}

We use the du command with the -sh options to perform this task. This, however, is not a complete solution to the problem. While it will work on some systems (Ubuntu, for example), it will not work on others. The reason is that many systems set the permissions of home directories to prevent them from being world-readable, which is a reasonable security measure. On these systems, the report\_home\_space function, as written, will only work if our script is run with superuser privileges. A better solution would be to have the script could adjust its behavior according to the privileges of the user. We will take this up in the next chapter.

我们使用带有 -sh 选项的 du 命令来完成这个任务。然而，这并不是此问题的完整解决方案。虽然它会 在一些系统（例如 Ubuntu）中起作用，但是在其它系统中它不工作。这是因为许多系统会设置家目录的 权限，以此阻止其它用户读取它们，这是一个合理的安全措施。在这些系统中，这个 report\_home\_space 函数， 只有用超级用户权限执行我们的脚本时，才会工作。一个更好的解决方案是让脚本能根据用户的使用权限来 调整自己的行为。我们将在下一章中讨论这个问题。

Shell Functions In Your .bashrc File

你的 .bashrc 文件中的 shell 函数

Shell functions make excellent replacements for aliases, and are actually the preferred method of creating small commands for personal use. Aliases are very limited in the kind of commands and shell features they support, whereas shell functions allow anything that can be scripted. For example, if we liked the report\_disk\_space shell function that we developed for our script, we could create a similar function named ds for our .bashrc file:

Shell 函数是更为完美的别名替代物，实际上是创建较小的个人所用命令的首选方法。别名 非常局限于命令的种类和它们支持的 shell 功能，然而 shell 函数允许任何可以编写脚本的东西。 例如，如果我们喜欢 为我们的脚本开发的这个 report\_disk\_space shell 函数，我们可以为我们的 .bashrc 文件 创建一个相似的名为 ds 的函数：

ds () {

echo “Disk Space Utilization For $HOSTNAME”

df -h

}

### 总结归纳

In this chapter, we have introduced a common method of program design called top- down design, and we have seen how shell functions are used to build the stepwise refinement that it requires. We have also seen how local variables can be used to make shell functions independent from one another and from the program in which they are placed. This makes it possible for shell functions to be written in a portable manner and to be reusable by allowing them to be placed in multiple programs; a great time saver.

这一章中，我们介绍了一种常见的程序设计方法，叫做自顶向下设计，并且我们知道了怎样 使用 shell 函数按照要求来完成逐步细化的任务。我们也知道了怎样使用局部变量使 shell 函数 独立于其它函数，以及其所在程序的其它部分。这就有可能使 shell 函数以可移植的方式编写， 并且能够重复使用，通过把它们放置到多个程序中；节省了大量的时间。

### 拓展阅读

* The Wikipedia has many articles on software design philosophy. Here are a couple of good ones:
* Wikipedia 上面有许多关于软件设计原理的文章。这里是一些好文章：

<http://en.wikipedia.org/wiki/Top-down_design>

<http://en.wikipedia.org/wiki/Subroutines>