第三十一章 疑难排解

As our scripts become more complex, it’s time to take a look at what happens when things go wrong and they don’t do what we want. In this chapter, we’ll look at some of the common kinds of errors that occur in scripts, and describe a few useful techniques that can be used to track down and eradicate problems.

随着我们的脚本变得越来越复杂，当脚本运行错误，执行结果出人意料的时候, 我们就应该查看一下原因了。 在这一章中，我们将会看一些脚本中出现地常见错误类型，同时还会介绍几个可以跟踪和消除问题的有用技巧。

### 语法错误

One general class of errors is syntactic. Syntactic errors involve mis-typing some element of shell syntax. In most cases, these kinds of errors will lead to the shell refusing to execute the script.

一个普通的错误类型是语法。语法错误涉及到一些 shell 语法元素的拼写错误。大多数情况下，这类错误 会导致 shell 拒绝执行此脚本。

In the following the discussions, we will use this script to demonstrate common types of errors:

在以下讨论中，我们将使用下面这个脚本，来说明常见的错误类型：

#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then

echo "Number is equal to 1."

else

echo "Number is not equal to 1."

fi

As written, this script runs successfully:

参看脚本内容，我们知道这个脚本执行成功了：

[me@linuxbox ~]$ trouble

Number is equal to 1.

#### 丢失引号

If we edit our script and remove the trailing quote from the argument following the first echo command:

如果我们编辑我们的脚本，并从跟随第一个 echo 命令的参数中，删除其末尾的双引号：

#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then

echo "Number is equal to 1.

else

echo "Number is not equal to 1."

fi

watch what happens:

观察发生了什么：

[me@linuxbox ~]$ trouble

/home/me/bin/trouble: line 10: unexpected EOF while looking for

matching `"'

/home/me/bin/trouble: line 13: syntax error: unexpected end of file

It generates two errors. Interestingly, the line numbers reported are not where the missing quote was removed, but rather much later in the program. We can see why, if we follow the program after the missing quote. bash will continue looking for the closing quote until it finds one, which it does immediately after the second echo command. bash becomes very confused after that, and the syntax of the if command is broken because the fi statement is now inside a quoted (but open) string.

这个脚本产生了两个错误。有趣地是，所报告的行号不是引号被删除的地方，而是程序中后面的文本行。 我们能知道为什么，如果我们跟随丢失引号文本行之后的程序。bash 会继续寻找右引号，直到它找到一个， 其就是这个紧随第二个 echo 命令之后的引号。找到这个引号之后，bash 变得很困惑，并且 if 命令的语法 被破坏了，因为现在这个 fi 语句在一个用引号引起来的（但是开放的）字符串里面。

In long scripts, this kind of error can be quite hard to find. Using an editor with syntax highlighting will help. If a complete version of vim is installed, syntax highlighting can be enabled by entering the command:

在冗长的脚本中，此类错误很难找到。使用带有语法高亮的编辑器将会帮助查找错误。如果安装了 vim 的完整版， 通过输入下面的命令，可以使语法高亮生效：

:syntax on

#### 丢失或意外的标记

Another common mistake is forgetting to complete a compound command, such as if or while. Let’s look at what happens if we remove the semicolon after the test in the if command:

另一个常见错误是忘记补全一个复合命令，比如说 if 或者是 while。让我们看一下，如果 我们删除 if 命令中测试之后的分号，会出现什么情况：

#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ] then

echo "Number is equal to 1."

else

echo "Number is not equal to 1."

fi

The result is this:

结果是这样的：

[me@linuxbox ~]$ trouble

/home/me/bin/trouble: line 9: syntax error near unexpected token

`else'

/home/me/bin/trouble: line 9: `else'

Again, the error message points to a error that occurs later than the actual problem. What happens is really pretty interesting. As we recall, if accepts a list of commands and evaluates the exit code of the last command in the list. In our program, we intend this list to consist of a single command, [, a synonym for test. The [ command takes what follows it as a list of arguments. In our case, three arguments: $number, =, and ]. With the semicolon removed, the word then is added to the list of arguments, which is syntactically legal. The following echo command is legal, too. It’s interpreted as another command in the list of commands that if will evaluate for an exit code. The else is encountered next, but it’s out of place, since the shell recognizes it as a reserved word (a word that has special meaning to the shell) and not the name of a command, hence the error message.

再次，错误信息指向一个错误，其出现的位置靠后于实际问题所在的文本行。所发生的事情真是相当有意思。我们记得， if 能够接受一系列命令，并且会计算列表中最后一个命令的退出代码。在我们的程序中，我们打算这个列表由 单个命令组成，即 [，测试的同义词。这个 [ 命令把它后面的东西看作是一个参数列表。在我们这种情况下， 有三个参数： $number，=，和 ]。由于删除了分号，单词 then 被添加到参数列表中，从语法上讲， 这是合法的。随后的 echo 命令也是合法的。它被解释为命令列表中的另一个命令，if 将会计算命令的 退出代码。接下来遇到单词 else，但是它出局了，因为 shell 把它认定为一个 保留字（对于 shell 有特殊含义的单词），而不是一个命令名，因此报告错误信息。

#### 预料不到的展开

It’s possible to have errors that only occur intermittently in a script. Sometimes the script will run fine and other times it will fail because of results of an expansion. If we return our missing semicolon and change the value of number to an empty variable, we can demonstrate:

可能有这样的错误，它们仅会间歇性地出现在一个脚本中。有时候这个脚本执行正常，其它时间会失败， 这是因为展开结果造成的。如果我们归还我们丢掉的分号，并把 number 的数值更改为一个空变量，我们 可以示范一下：

#!/bin/bash

# trouble: script to demonstrate common errors

number=

if [ $number = 1 ]; then

echo "Number is equal to 1."

else

echo "Number is not equal to 1."

fi

Running the script with this change results in the output:

运行这个做了修改的脚本，得到以下输出：

[me@linuxbox ~]$ trouble

/home/me/bin/trouble: line 7: [: =: unary operator expected

Number is not equal to 1.

We get this rather cryptic error message, followed by the output of the second echo command. The problem is the expansion of the number variable within the test command. When the command:

我们得到一个相当神秘的错误信息，其后是第二个 echo 命令的输出结果。这问题是由于 test 命令中 number 变量的展开结果造成的。当此命令：

[ $number = 1 ]

undergoes expansion with number being empty, the result is this:

经过展开之后，number 变为空值，结果就是这样：

[ = 1 ]

which is invalid and the error is generated. The = operator is a binary operator (it requires a value on each side), but the first value is missing, so the test command expects a unary operator (such as -z) instead. Further, since the test failed (because of the error), the if command receives a non-zero exit code and acts accordingly, and the second echo command is executed.

这是无效的，所以就产生了错误。这个 = 操作符是一个二元操作符（它要求每边都有一个数值），但是第一个数值是缺失的， 这样 test 命令就期望用一个一元操作符（比如 -z）来代替。进一步说，因为 test 命令运行失败了（由于错误）， 这个 if 命令接收到一个非零退出代码，因此执行第二个 echo 命令。

This problem can be corrected by adding quotes around the first argument in the test command:

通过为 test 命令中的第一个参数添加双引号，可以更正这个问题：

[ "$number" = 1 ]

Then when expansion occurs, the result will be this:

然后当展开操作发生地时候，执行结果将会是这样：

[ "" = 1 ]

which yields the correct number of arguments. In addition to empty strings, quotes should be used in cases where a value could expand into multi-word strings, as with filenames containing embedded spaces.

其得到了正确的参数个数。除了代表空字符串之外，引号应该被用于这样的场合，一个要展开 成多单词字符串的数值，及其包含嵌入式空格的文件名。

### 逻辑错误

Unlike syntactic errors, logical errors do not prevent a script from running. The script will run, but it will not produce the desired result, due to a problem with its logic. There are countless numbers of possible logical errors, but here are a few of the most common kinds found in scripts:

不同于语法错误，逻辑错误不会阻止脚本执行。虽然脚本会正常运行，但是它不会产生期望的结果， 归咎于脚本的逻辑问题。虽然有不计其数的可能的逻辑错误，但下面是一些在脚本中找到的最常见的 逻辑错误类型：

1. Incorrect conditional expressions. It’s easy to incorrectly code an if/then/else and have the wrong logic carried out. Sometimes the logic will be reversed or it will be incomplete.
2. “Off by one” errors. When coding loops that employ counters, it is possible to overlook that the loop may require the counting start with zero, rather than one, for the count to conclude at the correct point. These kinds of errors result in either a loop “going off the end” by counting too far, or else missing the last iteration of the loop by terminating one iteration too soon.
3. Unanticipated situations. Most logic errors result from a program encountering data or situations that were unforeseen by the programmer. This can also include unanticipated expansions, such as a filename that contains embedded spaces that expands into multiple command arguments rather than a single filename.
4. 不正确的条件表达式。很容易编写一个错误的 if/then/else 语句，并且执行错误的逻辑。 有时候逻辑会被颠倒，或者是逻辑结构不完整。
5. “超出一个值”错误。当编写带有计数器的循环语句的时候，为了计数在恰当的点结束，循环语句 可能要求从 0 开始计数，而不是从 1 开始，这有可能会被忽视。这些类型的错误要不导致循环计数太多，而“超出范围”， 要不就是过早的结束了一次迭代，从而错过了最后一次迭代循环。
6. 意外情况。大多数逻辑错误来自于程序碰到了程序员没有预见到的数据或者情况。这也 可以包括出乎意料的展开，比如说一个包含嵌入式空格的文件名展开成多个命令参数而不是单个的文件名。

#### 防错编程

It is important to verify assumptions when programming. This means a careful evaluation of the exit status of programs and commands that are used by a script. Here is an example, based on a true story. An unfortunate system administrator wrote a script to perform a maintenance task on an important server. The script contained the following two lines of code:

当编程的时候，验证假设非常重要。这意味着要仔细得计算脚本所使用的程序和命令的退出状态。 这里有个实例，基于一个真实的故事。为了在一台重要的服务器中执行维护任务，一位不幸的系统管理员写了一个脚本。 这个脚本包含下面两行代码：

cd $dir\_name

rm \*

There is nothing intrinsically wrong with these two lines, as long as the directory named in the variable, dir\_name, exists. But what happens if it does not? In that case, the cd command fails, the script continues to the next line and deletes the files in the current working directory. Not the desired outcome at all! The hapless administrator destroyed an important part of the server because of this design decision.

从本质上来说，这两行代码没有任何问题，只要是变量 dir\_name 中存储的目录名字存在就可以。但是如果不是这样会发生什么事情呢？在那种情况下，cd 命令会运行失败， 脚本会继续执行下一行代码，将会删除当前工作目录中的所有文件。完成不是期望的结果！ 由于这种设计策略，这个倒霉的管理员销毁了服务器中的一个重要部分。

Let’s look at some ways this design could be improved. First, it might be wise to make the execution of rm contingent on the success of cd:

让我们看一些能够提高这个设计的方法。首先，在 cd 命令执行成功之后，再运行 rm 命令，可能是明智的选择。

cd $dir\_name && rm \*

This way, if the cd command fails, the rm command is not carried out. This is better, but still leaves open the possibility that the variable, dir\_name, is unset or empty, which would result in the files in the user’s home directory being deleted. This could also be avoided by checking to see that dir\_name actually contains the name of an existing directory:

这样，如果 cd 命令运行失败后，rm 命令将不会执行。这样比较好，但是仍然有可能未设置变量 dir\_name 或其变量值为空，从而导致删除了用户家目录下面的所有文件。这个问题也能够避免，通过检验变量 dir\_name 中包含的目录名是否真正地存在：

[[ -d $dir\_name ]] && cd $dir\_name && rm \*

Often, it is best to terminate the script with an error when an situation such as the one above occurs:

通常，当某种情况（比如上述问题）发生的时候，最好是终止脚本执行，并对这种情况提示错误信息：

if [[ -d $dir\_name ]]; then

if cd $dir\_name; then

rm \*

else

echo "cannot cd to '$dir\_name'" >&2

exit 1

fi

else

echo "no such directory: '$dir\_name'" >&2

exit 1

fi

Here, we check both the name, to see that it is that of an existing directory, and the success of the cd command. If either fails, a descriptive error message is sent to standard error and the script terminates with an exit status of one to indicate a failure.

这里，我们检验了两种情况，一个名字，看看它是否为一个真正存在的目录，另一个是 cd 命令是否执行成功。 如果任一种情况失败，就会发送一个错误说明信息到标准错误，然后脚本终止执行，并用退出状态 1 表明脚本执行失败。

#### 验证输入

A general rule of good programming is that if a program accepts input, it must be able to deal with anything it receives. This usually means that input must be carefully screened, to ensure that only valid input is accepted for further processing. We saw an example of this in the previous chapter when we studied the read command. One script contained the following test to verify a menu selection:

一个良好的编程习惯是如果一个程序可以接受输入数据，那么这个程序必须能够应对它所接受的任意数据。这 通常意味着必须非常仔细地筛选输入数据，以确保只有有效的输入数据才能被程序用来做进一步地处理。在前面章节 中我们学习 read 命令的时候，我们遇到过一个这样的例子。一个脚本中包含了下面一条测试语句， 用来验证一个选择菜单：

[[ $REPLY =~ ^[0-3]$ ]]

This test is very specific. It will only return a zero exit status if the string returned by the user is a numeral in the range of zero to three. Nothing else will be accepted. Sometimes these sorts of tests can be very challenging to write, but the effort is necessary to produce a high quality script.

这条测试语句非常明确。只有当用户输入是一个位于 0 到 3 范围内（包括 0 和 3）的数字的时候， 这条语句才返回一个 0 退出状态。而其它任何输入概不接受。有时候编写这类测试条件非常具有挑战性， 但是为了能产出一个高质量的脚本，付出还是必要的。

Design Is A Function Of Time

**设计是时间的函数**

When I was a college student studying industrial design, a wise professor stated that the degree of design on a project was determined by the amount of time given to the designer. If you were given five minutes to design a device “that kills flies,” you designed a flyswatter. If you were given five months, you might come up with a laser-guided “anti-fly system” instead.

当我还是一名大学生，在学习工业设计的时候，一位明智的教授说过一个项目的设计程度是由 给定设计师的时间量来决定的。如果给你五分钟来设计一款能够 “杀死苍蝇” 的产品，你会设计出一个苍蝇拍。如果给你五个月的时间，你可能会制作出激光制导的 “反苍蝇系统”。

The same principle applies to programming. Sometimes a “quick and dirty” script will do if it’s only going to be used once and only used by the programmer. That kind of script is common and should be developed quickly to make the effort economical. Such scripts don’t need a lot of comments and defensive checks. On the other hand, if a script is intended for production use, that is, a script that will be used over and over for an important task or by multiple users, it needs much more careful development.

同样的原理适用于编程。有时候一个 “快速但粗糙” 的脚本就可以解决问题， 但这个脚本只能被其作者使用一次。这类脚本很常见，为了节省气力也应该被快速地开发出来。 所以这些脚本不需要太多的注释和防错检查。相反，如果一个脚本打算用于生产使用，也就是说， 某个重要任务或者多个客户会不断地用到它，此时这个脚本就需要非常谨慎小心地开发了。

### 测试

Testing is an important step in every kind of software development, including scripts. There is a saying in the open source world, “release early, release often,” which reflects this fact. By releasing early and often, software gets more exposure to use and testing. Experience has shown that bugs are much easier to find, and much less expensive to fix, if they are found early in the development cycle.

在各类软件开发中（包括脚本），测试是一个重要的环节。在开源世界中有一句谚语，“早发布，常发布”，这句谚语就反映出这个事实（测试的重要性）。 通过提早和经常发布，软件能够得到更多曝光去使用和测试。经验表明如果在开发周期的早期发现 bug，那么这些 bug 就越容易定位，而且越能低成本 的修复。

In a previous discussion, we saw how stubs can be used to verify program flow. From the earliest stages of script development, they are a valuable technique to check the progress of our work.

在之前的讨论中，我们知道了如何使用 stubs 来验证程序流程。在脚本开发的最初阶段，它们是一项有价值的技术 来检测我们的工作进度。

Let’s look at the file deletion problem above and see how this could be coded for easy testing. Testing the original fragment of code would be dangerous, since its purpose is to delete files, but we could modify the code to make the test safe:

让我们看一下上面的文件删除问题，为了轻松测试，看看如何修改这些代码。测试原本那个代码片段将是危险的，因为它的目的是要删除文件， 但是我们可以修改代码，让测试安全：

if [[ -d $dir\_name ]]; then

if cd $dir\_name; then

echo rm \* # TESTING

else

echo "cannot cd to '$dir\_name'" >&2

exit 1

fi

else

echo "no such directory: '$dir\_name'" >&2

exit 1

fi

exit # TESTING

Since the error conditions already output useful messages, we don’t have to add any. The most important change is placing an echo command just before the rm command to allow the command and its expanded argument list to be displayed, rather than the command actually being executed. This change allows safe execution of the code. At the end of the code fragment, we place an exit command to conclude the test and prevent any other part of the script from being carried out. The need for this will vary according to the design of the script.

因为在满足出错条件的情况下代码可以打印出有用信息，所以我们没有必要再添加任何额外信息了。 最重要的改动是仅在 rm 命令之前放置了一个 echo 命令， 为的是把 rm 命令及其展开的参数列表打印出来，而不是执行实际的 rm 命令语句。这个改动可以安全的执行代码。 在这段代码的末尾，我们放置了一个 exit 命令来结束测试，从而防止执行脚本其它部分的代码。 这个需求会因脚本的设计不同而变化。

We also include some comments that act as “markers” for our test-related changes. These can be used to help find and remove the changes when testing is complete.

我们也在代码中添加了一些注释，用来标记与测试相关的改动。当测试完成之后，这些注释可以帮助我们找到并删除所有的更改。

#### 测试案例

To perform useful testing, it’s important to develop and apply good test cases. This is done by carefully choosing input data or operating conditions that reflect edge and corner cases. In our code fragment (which is very simple), we want to know how the code performs under three specific conditions:

为了执行有用的测试，开发和使用好的测试案例是很重要的。这个要求可以通过谨慎地选择输入数据或者运行边缘案例和极端案例来完成。 在我们的代码片段中（是非常简单的代码），我们想要知道在下面的三种具体情况下这段代码是怎样执行的：

1. dir\_name contains the name of an existing directory
2. dir\_name contains the name of a non-existent directory
3. dir\_name is empty
4. dir\_name 包含一个已经存在的目录的名字
5. dir\_name 包含一个不存在的目录的名字
6. dir\_name 为空

By performing the test with each of these conditions, good test coverage is achieved.

通过执行以上每一个测试条件，就达到了一个良好的测试覆盖率。

Just as with design, testing is a function of time, as well. Not every script feature needs to be extensively tested. It’s really a matter of determining what is most important. Since it could be so potentially destructive if it malfunctioned, our code fragment deserves careful consideration during both its design and testing.

正如设计，测试也是一个时间的函数。不是每一个脚本功能都需要做大量的测试。问题关键是确定什么功能是最重要的。因为 测试若发生故障会存在如此潜在的破坏性，所以我们的代码片在设计和测试段期间都应值得仔细推敲。

### 调试

If testing reveals a problem with a script, the next step is debugging. “A problem” usually means that the script is, in some way, not performing to the programmers expectations. If this is the case, we need to carefully determine exactly what the script is actually doing and why. Finding bugs can sometimes involve a lot of detective work. A well designed script will try to help. It should be programmed defensively, to detect abnormal conditions and provide useful feedback to the user. Sometimes, however, problems are quite strange and unexpected and more involved techniques are required.

如果测试暴露了脚本中的一个问题，那下一步就是调试了。“一个问题”通常意味着在某种情况下，这个脚本的执行 结果不是程序员所期望的结果。若是这种情况，我们需要仔细确认这个脚本实际到底要完成什么任务，和为什么要这样做。 有时候查找 bug 要牵涉到许多监测工作。一个设计良好的脚本会对查找错误有帮助。设计良好的脚本应该具备防卫能力， 能够监测异常条件，并能为用户提供有用的反馈信息。 然而有时候，出现的问题相当稀奇，出人意料，这时候就需要更多的调试技巧了。

#### 找到问题区域

In some scripts, particularly long ones, it is sometimes useful to isolate the area of the script that is related to the problem. This won’t always be the actual error, but isolation will often provide insights into the actual cause. One technique that can be used to isolate code is “commenting out” sections a script. For example, our file deletion fragment could be modified to determine if the removed section was related to an error:

在一些脚本中，尤其是一些代码比较长的脚本，有时候隔离脚本中与出现的问题相关的代码区域对查找问题很有效。 隔离的代码区域并不总是真正的错误所在，但是隔离往往可以深入了解实际的错误原因。可以用来隔离代码的一项 技巧是“添加注释”。例如，我们的文件删除代码可以修改成这样，从而决定注释掉的这部分代码是否导致了一个错误：

if [[ -d $dir\_name ]]; then

if cd $dir\_name; then

rm \*

else

echo "cannot cd to '$dir\_name'" >&2

exit 1

fi

# else

#

echo "no such directory: '$dir\_name'" >&2

#

exit 1

fi

By placing comment symbols at the beginning of each line in a logical section of a script, we prevent that section from being executed. Testing can then be performed again, to see if the removal of the code has any impact on the behavior of the bug.

通过给脚本中的一个逻辑区块内的每条语句的开头添加一个注释符号，我们就阻止了这部分代码的执行。然后可以再次执行测试， 来看看清除的代码是否影响了错误的行为。

#### 追踪

Bugs are often cases of unexpected logical flow within a script. That is, portions of the script are either never being executed, or are being executed in the wrong order or at the wrong time. To view the actual flow of the program, we use a technique called tracing.

在一个脚本中，错误往往是由意想不到的逻辑流导致的。也就是说，脚本中的一部分代码或者从未执行，或是以错误的顺序， 或在错误的时间给执行了。为了查看真实的程序流，我们使用一项叫做追踪（tracing）的技术。

One tracing method involves placing informative messages in a script that display the location of execution. We can add messages to our code fragment:

一种追踪方法涉及到在脚本中添加可以显示程序执行位置的提示性信息。我们可以添加提示信息到我们的代码片段中：

echo "preparing to delete files" >&2

if [[ -d $dir\_name ]]; then

if cd $dir\_name; then

echo "deleting files" >&2

rm \*

else

echo "cannot cd to '$dir\_name'" >&2

exit 1

fi

else

echo "no such directory: '$dir\_name'" >&2

exit 1

fi

echo "file deletion complete" >&2

We send the messages to standard error to separate them from normal output. We also do not indent the lines containing the messages, so it is easier to find when it’s time to remove them.

我们把提示信息输出到标准错误输出，让其从标准输出中分离出来。我们也没有缩进包含提示信息的语句，这样 想要删除它们的时候，能比较容易找到它们。

Now when the script is executed, it’s possible to see that the file deletion has been performed:

当这个脚本执行的时候，就可能看到文件删除操作已经完成了：

[me@linuxbox ~]$ deletion-script

preparing to delete files

deleting files

file deletion complete

[me@linuxbox ~]$

bash also provides a method of tracing, implemented by the -x option and the set command with the -x option. Using our earlier trouble script, we can activate tracing for the entire script by adding the -x option to the first line:

bash 还提供了一种名为追踪的方法，这种方法可通过 -x 选项和 set 命令加上 -x 选项两种途径实现。 拿我们之前的 trouble 脚本为例，给该脚本的第一行语句添加 -x 选项，我们就能追踪整个脚本。

#!/bin/bash -x

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then

echo "Number is equal to 1."

else

echo "Number is not equal to 1."

fi

When executed, the results look like this:

当脚本执行后，输出结果看起来像这样:

[me@linuxbox ~]$ trouble

+ number=1

+ '[' 1 = 1 ']'

+ echo 'Number is equal to 1.'

Number is equal to 1.

With tracing enabled, we see the commands performed with expansions applied. The leading plus signs indicate the display of the trace to distinguish them from lines of regular output. The plus sign is the default character for trace output. It is contained in the PS4 (prompt string 4) shell variable. The contents of this variable can be adjusted to make the prompt more useful. Here, we modify the contents of the variable to include the current line number in the script where the trace is performed. Note that single quotes are required to prevent expansion until the prompt is actually used:

追踪生效后，我们看到脚本命令展开后才执行。行首的加号表明追踪的迹象，使其与常规输出结果区分开来。 加号是追踪输出的默认字符。它包含在 PS4（提示符4）shell 变量中。可以调整这个变量值让提示信息更有意义。 这里，我们修改该变量的内容，让其包含脚本中追踪执行到的当前行的行号。注意这里必须使用单引号是为了防止变量展开，直到 提示符真正使用的时候，就不需要了。

[me@linuxbox ~]$ export PS4='$LINENO + '

[me@linuxbox ~]$ trouble

5 + number=1

7 + '[' 1 = 1 ']'

8 + echo 'Number is equal to 1.'

Number is equal to 1.

To perform a trace on a selected portion of a script, rather than the entire script, we can use the set command with the -x option:

我们可以使用 set 命令加上 -x 选项，为脚本中的一块选择区域，而不是整个脚本启用追踪。

#!/bin/bash

# trouble: script to demonstrate common errors

number=1

set -x # Turn on tracing

if [ $number = 1 ]; then

echo "Number is equal to 1."

else

echo "Number is not equal to 1."

fi

set +x # Turn off tracing

We use the set command with the -x option to activate tracing and the +x option to deactivate tracing. This technique can be used to examine multiple portions of a troublesome script.

我们使用 set 命令加上 -x 选项来启动追踪，+x 选项关闭追踪。这种技术可以用来检查一个有错误的脚本的多个部分。

#### 执行时检查数值

It is often useful, along with tracing, to display the content of variables to see the internal workings of a script while it is being executed. Applying additional echo statements will usually do the trick:

伴随着追踪，在脚本执行的时候显示变量的内容，以此知道脚本内部的工作状态，往往是很用的。 使用额外的 echo 语句通常会奏效。

#!/bin/bash

# trouble: script to demonstrate common errors

number=1

echo "number=$number" # DEBUG

set -x # Turn on tracing

if [ $number = 1 ]; then

echo "Number is equal to 1."

else

echo "Number is not equal to 1."

fi

set +x # Turn off tracing

In this trivial example, we simply display the value of the variable number and mark the added line with a comment to facilitate its later identification and removal. This technique is particularly useful when watching the behavior of loops and arithmetic within scripts.

在这个简单的示例中，我们只是显示变量 number 的数值，并为其添加注释，随后利于其识别和清除。 当查看脚本中的循环和算术语句的时候，这种技术特别有用。

### 总结

In this chapter, we looked at just a few of the problems that can crop up during script de- velopment. Of course, there are many more. The techniques described here will enable finding most common bugs. Debugging is a fine art that can be developed through experience, both in knowing how to avoid bugs (testing constantly throughout development) and in finding bugs (effective use of tracing).

在这一章中，我们仅仅看了几个在脚本开发期间会出现的问题。当然，还有很多。这章中描述的技术对查找 大多数的常见错误是有效的。调试是一种艺术，可以通过开发经验，在知道如何避免错误(整个开发过程中不断测试) 以及在查找 bug（有效利用追踪）两方面都会得到提升。

### 拓展阅读

* The Wikipedia has a couple of short articles on syntactic and logical errors:
* Wikipedia 上面有两篇关于语法和逻辑错误的短文：

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

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

* There are many online resources for the technical aspects of bash programming:
* 网上有很多关于技术层面的 bash 编程的资源：

<http://mywiki.wooledge.org/BashPitfalls>

<http://tldp.org/LDP/abs/html/gotchas.html>

<http://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html>

* Eric Raymond’s The Art of Unix Programming is a great resource for learning the basic concepts found in well-written Unix programs. Many of these ideas apply to shell scripts:
* 想要学习从编写良好的 Unix 程序中得知的基本概念，可以参考 Eric Raymond 的《Unix 编程的艺术》这本 伟大的著作。书中的许多想法都能适用于 shell 脚本：

<http://www.faqs.org/docs/artu/>

<http://www.faqs.org/docs/artu/ch01s06.html>

* For really heavy-duty debugging, there is the Bash Debugger:
* 对于真正的高强度的调试，参考这个 Bash Debugger：

<http://bashdb.sourceforge.net/>