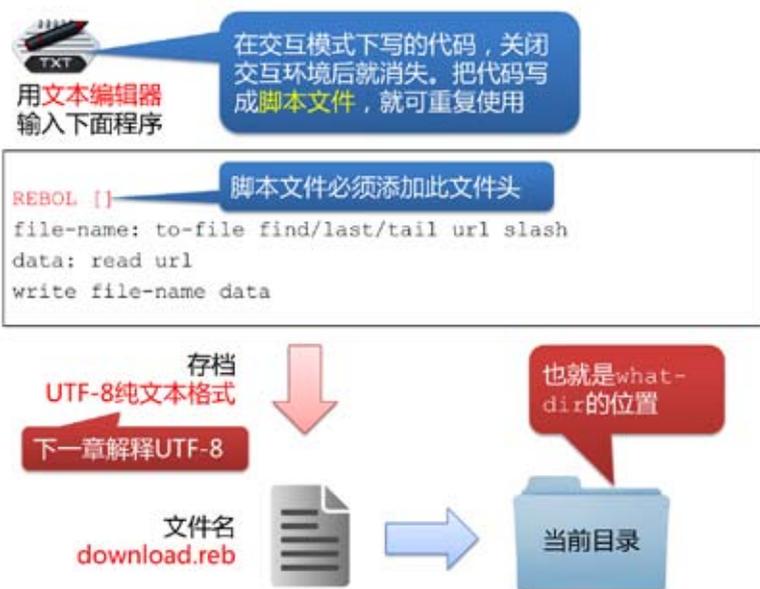




第3章

脚本文件

第1篇 编程原理



结束交互环境之后，历史记录就被清除了，下次无法延续。有些程序，我们觉得有价值，想把它保留下来，或许以后还用得着。REBOL 允许我们通过记事本等文本编辑软件，进行代码的编写并存档。以前面网站下载图片的程序为例来说，你可能想把它保留下来。做法如下：

1. 用任何一个文本编辑器（例如 Windows 的记事本或 Mac OS X 的文本编辑），将之前的 ABC 三行代码依次输入（或者从交互环境的历史记录中复制过来）。
2. 最前面加上 `REBOL []`，这称为 REBOL 文件头（header）。
3. 把此文件保存为纯文本格式，UTF-8 编码（下一章再解释什么是 UTF-8），保存在与 REBOL 解释器相同的目录，文件名 `download.reb`。

如果你使用微软 Windows 记事本，保存时的操作方式是：文件→另存为→选择目录→保存类型（所有文件）→输入文件名 `download.reb` →编码（UTF-8）→确定。如果你使用 Mac OS X 的文本编辑，保存时的操作方式是：格式（纯文本）→存储为→选择目录→输入文件名 `download.reb` →编码（UTF-8）。

REBOL 解释器没有限制我们脚本必须使用什么扩展名，一般常看到的 REBOL 脚本扩展名有 .r、.reb、.r3。建议使用 .reb 当扩展名。

编写 REBOL 脚本文件时，不要使用太华丽的编辑器（例如微软 Word），因为：

- 它可能会在文件中记录很多 REBOL 解释器不需要的信息（例如字体、颜色），这将导致我们的代码无法运行。
- 它可能会在我们编辑的时候，多余地帮我们做字符转换，例如把 " " 转为 “ ”，这将导致我们的代码无法运行。

第1篇 编程原理



```
(下面是执行刚才脚本文件的操作方式)
>> url: http://www.apple.com/index.html
== http://www.apple.com/index.html

>> do %download.reb
Script: "Untitled" Version: none Date: none

>> url: http://cdn.oreilly.com/images/site
wide-headers/ml-header-home-blinking.gif
== ... 省略 ...

>> do %download.reb
Script: "Untitled" Version: none Date: none

>> ls
index.html    ml-header-home-blinking.gif
... 省略 ...

(如果你的操作过程不顺利, 看下一页)
```

在交互环境下, 先设置好 `url` 单字, 然后在 `do` 的后面接着输入脚本文件名, 就可运行 REBOL 脚本文件。REBOL 中规定文件名称前面要加上 `%` 符号。也就是说, `download.reb` 必须写成 `%download.reb`。

如果脚本运行成功, 会看到一行文字, 打印出脚本的名称 (Script)、版本 (Version)、日期 (Date), 这些我们目前都没定义, 所以显示 `Untitled` 和 `none` (空值)。没有错误信息表示一切顺利。

你到当前的目录下会发现图片已经下载回来了, 用鼠标双击图片, 就可打开。如果运行不成功, 就看看错误信息, 从中分析, 找出问题所在, 并做出修改, 这是程序设计人员必须具备的基本素质。下一个页面是分析问题的过程示例。

```
(上页的操作过程不顺利? 这一页教你判断问题)
>> url: http://www.apple.com/index.html
== http://www.apple.com/index.html
>> do %download.reb
** Access error: cannot open: %download.reb
...省略
>> LS
REBOL3.exe history.txt
download.reb.txt
>> rename %download.reb.txt %download.reb
>> do %download.reb
>> size? %download.reb
== 94
```

如果输入 `do %download.reb`, 得到的错误信息是 `** Access error: cannot open: %download.reb`, 这表示文件不存在。可通过 `LS`, 看看当前目录下所有的文件, 证实文件确实不存在。

如果 `do %download.reb` 无法顺利运行, 而且在 `LS` 的结果中看到一个文件名为 `download.reb.txt`, 这表示我们之前在保存文件时没有做出正确的选择, 造成文件名称错误。我们可以通过 `rename` 改变它的名字, 做法如上图所示。

如果确定文件存在正确的目录下, 文件名称也正确, 但还是无法顺利执行, 这时候我们就要怀疑是不是内容格式有错误。再强调一次, REBOL 要求脚本必须是纯文本格式, UTF-8 编码。关于文本格式与编码, 下一章会有说明。

某些文字编辑器保存文件的时候, 如果不做选择, 将默认采用富文本格式而不是纯文本格式。富文本格式内会包含字体、颜色等信息, 这类格式的文件都比较大, 通过 `size?` 函数就可以得知该文件的大小是多少字节(下一章会介绍字节), 如果我们发现它比纯文本代码格式所需要的 94 字节更多(你的程序中的空格与换行可能与我的不完全一致, 所以你的脚本大小不一定是 94 这个数字, 但应该很接近 94), 且大了很多, 则可能是富文本, 这时候你需要用编辑器重新将文件保存为之前规定的格式(UTF-8 纯文本)。按照前面的说明, 重新保存一次。



这是另一个脚本文件的范例。有点特别的是，我们在代码中使用了中文（REBOL 允许代码中出现各国文字）。我暂时不解释这段代码的细节。

与前面 `download.reb` 的例子一样：

1. 通过文本编辑器输入代码。
2. 保存时用 UTF-8 纯文本格式，文件名 `greeting.reb`，存放到目前目录。



The image shows a terminal window with three blue callout boxes on the left pointing to specific parts of the terminal output. The first callout, labeled '执行程序', points to the first line of the terminal output. The second callout, labeled '输入名字', points to the user's input 'Jerry'. The third callout, labeled '查询do的用法', points to the help text for the 'do' command.

```
(下面是执行刚才脚本文件的操作方式)
>> do %greeting.reb
Script: "Untitled" Version: none Date: none
你叫什么名字?
Jerry
Jerry 下午好

>> ? do
用法:
    DO value /args arg /next var
描述:
    执行一个方块、文件、 ... 省略
    DO是一个原生函数值。
参数:
    value - 通常是文件名、 ... 省略
```

同样，通过 `do` 函数来执行 `greeting.reb` 代码。你会看到此程序询问你叫什么名字，输入你的名字之后按下回车键，就会看到此程序对你打招呼。



我们已经写过 `download.reb` 与 `greeting.reb` 这两个脚本文件。既然程序要写在文件中才能保存，那么交互环境又有什么价值？

虽然 REBOL 程序员真正的程序设计工作都是在文本编辑器中完成的，但他们经常会同时打开交互环境，在上面做一些实验，验证自己的想法是否正确，或者查询一下说明文档，然后再回文本编辑器内继续写程序。

在交互环境与文本编辑器中写代码的差异不大：交互环境的程序通常很短，一行就是一个程序。交互环境不需要写 `REBOL []` 文件头。其他情况下，交互环境与文本编辑器几乎没有差别。

通过?可查文档。函数文档最多具有四部分

```
>> ? type?
1 用途:
   TYPE? value /word

2 描述:
   返回一个值的数据类型。
   TYPE?是个原生函数值。

3 参数:
   value (any-type!)

4 修饰字:
   /word - 返回一个单字
```

若为非函数的值，则查到的文档很简单

```
>> ? pi
PI是一个小数值: 3.14159265358979
```

本页旨在展示在线文档的组成，看不懂细节也不用担心

阅读在线文档，是程序员的基本功之一。REBOL 程序员经常会在交互环境下查询文档，用 ? 函数（? 是 help 的简写）来查询某些函数的用法，函数文档的内容分成四个部分，分别是用法（Usage）、描述（Description）、参数（Arguments）、修饰字（Refinements）：

1. 用法区说明代码中的写法。包括需要什么参数，可以有什么修饰字，且这些修饰字需要什么参数。在看这一部分时，我关注的是参数名称前面有没有紧接着 ' 或者 :。本页的例子中，参数名称 value 前面没有接上 : 或 '。关于 ' 与 : 的意义，第 14 章会详细解说。
2. 描述区说明此函数的用途、函数的种类（函数有 7 种，包括 native!、op!、closure!、command!、action!、function!、rebcodes!）。
3. 参数区说明函数需要的参数类型，以及相关解说。
4. 修饰字区列出该函数所有的修饰字，以及它们的目的与需要的参数。

这里有许多名词之前没见过，先不用担心，后面会在适当的时机详细解说。

【自我测验】

我们已经接触过这些函数，你记得这些函数的用途吗？



- | | | |
|-----------------------------------|------------------------------------|---------------------------------|
| <input type="checkbox"/> + | <input type="checkbox"/> / | <input type="checkbox"/> DO |
| <input type="checkbox"/> Q | <input type="checkbox"/> ECHO | <input type="checkbox"/> RENAME |
| <input type="checkbox"/> WHAT-DIR | <input type="checkbox"/> WHAT | <input type="checkbox"/> SIZE? |
| <input type="checkbox"/> CD | <input type="checkbox"/> PRINT | <input type="checkbox"/> EITHER |
| <input type="checkbox"/> POWER | <input type="checkbox"/> TO-STRING | <input type="checkbox"/> NOW |
| <input type="checkbox"/> WRITE | <input type="checkbox"/> READ | <input type="checkbox"/> ASK |
| <input type="checkbox"/> LS | <input type="checkbox"/> TO-FILE | <input type="checkbox"/> ? |
| <input type="checkbox"/> DELETE | <input type="checkbox"/> FIND | <input type="checkbox"/> TYPE? |

你能答对多少？不清楚的话可以翻到前面的内容复习，或者通过 `help` 函数（也就是 `?`）查询文件。

我成功编写脚本文件并执行之

如果你确定做到了，到本篇开始“学习目标”处打钩。