第2章 周而复始：REPL简介

本章将学习如何设置编程环境并编写第一个Common Lisp程序。我们将使用由Matthew Danish和Mikel Evins开发的安装便利的Lisp in a Box环境，它封装了带有Emacs（强大且对Lisp较为友好的文本编辑器）的Common Lisp实现，以及SLIME（Superior Lisp Interaction Mode for Emacs）——构建在Emacs之上的Common Lisp开发环境。

上述组合提供了一个全新的Common Lisp开发环境，可以支持增量、交互式的开发风格，这是Lisp编程所特有的。SLIME环境提供了一个完全统一的用户接口，跟你所选择的操作系统和Common Lisp实现无关。为了能有一个具体的开发环境来进行讲解，我将使用这个Lisp in a Box环境。而那些想要探索其他开发环境的人，无论是使用某些商业Lisp供应商的图形化集成开发环境（IDE）还是基于其他编辑器的环境，在应用本书的内容时应该都不会有较大的困难。[①]

2.1 选择一个 Lisp 实现

首要的问题是选择一个Lisp实现。这对那些曾经使用诸如Perl、Python、Visual Basic（VB）、C#和Java语言的人来说，可能有些奇怪。Common Lisp和这些语言的区别在于，Common Lisp是标准化的，既不像Perl和Python那样是由善良的专制者所控制的某个实现，也不像VB、C#和Java那样是由某个公司控制的公认实现。任何打算阅读标准并实现该语言的人都可以自由地这样做。更进一步，对标准的改动必须在标准化组织美国国家标准学会（ANSI）所控制的程序下进行。

① 如果你之前不喜欢用Emacs，那你应该把Lisp in a Box当作一个刚好使用类Emacs编辑器作为文本编辑器的IDE，你不需要先成为Emacs大师再写Lisp程序。不过，使用一个基本支持Lisp的编辑器来编写Lisp程序将是件有趣的事情。至少，你需要一个可以帮你自动匹配括号并且知道如何自动缩进Lisp代码的编辑器。因为Emacs本身大部分是用一种叫做Elisp的Lisp方言写成的，所以它在相当程度上支持编辑Lisp代码。Emacs也在很大程度上参与到了Lisp的历史和Lisp黑客的文化中：最初的Emacs和它的前身TECMACS和TMACS，是由麻省理工学院（MIT）的Lisp程序员们写成的。Lisp机上的编辑器是完全用Lisp写成的某些版本的Emacs。最早的两种Lisp机Emacs，其命名方式沿用了使用递归缩略语的黑客传统，分别是EINE和ZWEI，意思是EINE Is Not Emacs以及ZWEI Was EINE Initially。后来的版本使用了ZWEI的一个派生版，其名字ZMACS也缺乏想象力。

这一设计可以避免任何个体（例如某个厂商）任意修改标准。[1]这样，Common Lisp标准就是Common Lisp供应商和Common Lisp程序员之间的一份协议。这份协议告诉你，如果所写的程序按照标准里描述的方式使用了某些语言特性，那么就可以指望程序在任何符合标准的实现里也能产生同样的行为。

另一方面，标准可能并没有覆盖程序所涉及的每个方面，某些方面故意没有定义，为的是允许实现者在这些领域里继续探索，在这些领域中，特定的语言风格尚未找到最佳的支持方式。因此每种实现都提供了一些依托并超越标准所规定范围的特性。根据你正打算进行的编程类型，有时选择一种带有其他所需特性的特定实现也是合理的。另外，如果我们正在向其他人交付Lisp源代码，例如库，那么你将需要尽可能地编写可移植的Common Lisp。假如所要编写的代码大部分可以移植，但其中要用到一些标准未曾定义过的功能时，Common Lisp提供的灵活方式可以编写出完全依赖于特定实现中某些特性的代码。第15章将看到这样的代码，在那里我们将开发一个简单的库来消除不同Lisp实现在对待文件名上的区别。

但从目前而论，一个实现最重要的特点应该在于，它能否运行在我们所喜爱的操作系统上。Franz的Allegro Common Lisp开发者们放出了一个可用于本书的试用版产品，能够运行在Linux、Windows和Mac OS X上。试图寻找开源实现的人们有几种选择。SBCL（Steel Bank Common Lisp）是一个高质量的开源实现，它将程序编译成原生代码并且可以运行在广泛的Unix平台上，包括Linux和Mac OS X。SBCL来源于CMUCL（CMU Common Lisp），最早由卡内基-梅隆大学开发的一种Common Lisp，并且像CMUCL那样，大部分源代码都处于公共域（public domain），只有少量代码采用伯克利软件分发（BSD）风格的协议。CMUCL本身也是个不错的选择，尽管SBCL更容易安装并且现在支持21位Unicode。[2]而对于Mac OS X用户来说，OpenMCL[3]是一个极佳的选择，它可编译到机器码、支持线程，并且可以跟Mac OS X的Carbon和Cocoa工具箱很好地集成。还有另外一些开源和商业实现，参见第32章以获取更多相关资源的信息。

除非特别说明，本书中的所有Lisp代码都应该可以工作在任何符合标准的Common Lisp实现上，而SLIME通过为我们提供一个与Lisp交互的通用接口也可以消除不同实现之间的某些差异。本书里给出的程序输出来自运行在GNU/Linux上的Allegro平台。在某些情况下，其他Lisp可能会产生稍有不同的错误信息或调试输出。

① 实际上，该语言标准本身被修订的可能性微乎其微。尽管其中存在一些人们想改进的缺陷，但ANSI的流程不允许打开一个已有的标准进行细节修补，而且事实上也没有哪个打算改进的地方给任何人造成了任何严重的困难。Common Lisp 标准的未来很可能会通过事实上的标准来进行，就像是Perl和Python的“标准化过程”那样——在不同的实现者试验语言标准里没有定义的应用程序接口（API）和库的时候，其他实现者可能采纳他们的工作，或者人们将开发可移植的库来消除那些语言没有定义的特性在各种实现之间的区别。

② 最初SBCL从CMUCL里分离出来，主要是为了集中清理其内部代码结构，以使其更容易维护。这个代码分叉现在已经很友好了。bug修复经常在两个项目之间传递，并且有传闻某一天他们会重新合并在一起。

③ OpenMCL项目自从移植到Mac OS之外的平台后，更名为Clozure CL。——译者注

2.2　安装和运行Lisp in a Box

由于Lisp in a Box软件包可以让新Lisp程序员在一流的Lisp开发环境上近乎无痛起步，因此你需要做的就是根据你的操作系统和喜爱的Lisp平台从本书的Web站点http://www.gigamonkeys.com/book/上获取相应的安装包，然后按照安装说明提示来操作即可。

由于Lisp in a Box使用Emacs作为其编辑器，因此你至少得懂一点儿它的使用方法。最好的Emacs入门方法也许就是跟着它内置的向导走一遍。要想启动这个向导，选择帮助菜单的第一项Emacs tutorial即可。或者按住Ctrl键，输入h，然后放开Ctrl键，再按t。大多数Emacs命令都可以通过类似的组合键来访问。由于组合键用得如此普遍，因而Emacs用户使用了一种记号来描述组合键，以避免经常书写诸如“按住Ctrl键，输入h，然后放开Ctrl键，再按t”这样的组合。需要一起按的键，即键和弦，用连接号（-）连接；顺序按下的键或键和弦，用空格分隔。在一个键和弦里，C代表Ctrl键而M代表Meta键（也就是Alt键）。这样，我们可以将刚才描述的启动向导的按键组合直接写成：C-h t。

向导里还描述了其他有用的命令以及启动它们的组合键。Emacs还提供了大量在线文档，可以通过其内置的超文本浏览器Info来访问。阅读这些手册只需输入C-h i。这个Info系统也有其自身的向导，可以简单地在阅读手册时按下h来访问。最后，Emacs提供了好几种获取帮助信息的方式，全部绑定到了以C-h开头的组合键上。输入C-h ?就可以得到一个完整的列表。除了向导以外，还有两个最有用的帮助命令：一个是C-h k，告诉我们输入的任何组合键所对应的命令是什么；另一个是C-h w，告诉我们输入的命令名字所对应的组合键是什么。

对于那些拒绝看向导的人来说，有个至关重要的Emacs术语不得不提，那就是缓冲区（buffer）。当使用Emacs时，你所编辑的每个文件都将被表示成不同的缓冲区，在任一时刻只有一个缓冲区是“当前使用的”。当前缓冲区会接收所有的输入——无论是在打字还是调用任何命令。缓冲区也用来表示与Common Lisp这类程序的交互。因此，一个常用的操作就是“切换缓冲区”，就是说将一个不同的缓冲区设置为当前缓冲区，以便可以编辑某个特定的文件或者与特定的程序交互。这个命令是`switch-to-buffer`，对应的组合键是C-x b，使用时将提示你在Emacs框架的底部输入一个缓冲区的名字。当输入缓冲区的名字时，按Tab键将在输入的字符基础上对其补全，或者显示一个所有可能补全方式的列表。该提示同时推荐了一个默认缓冲区，你可以直接按回车（Return）键来选择它。也可以通过在缓冲区菜单里选择一个缓冲区来切换。

在特定的上下文环境中，其他组合键也可用于切换到特定的缓冲区。例如，当编辑Lisp源文件时，组合键C-c C-z可以切换到与Lisp进行交互的那个缓冲区。

2.3　放开思想：交互式编程

当启动Lisp in a Box时，应该可以看到一个带有类似下面提示符的缓冲区：

```
CL-USER>
```

这是Lisp的提示符。就像Unix或DOS shell提示符那样，在Lisp提示符的位置上输入表达式可以产生一定的结果。尽管如此，Lisp读取的是Lisp表达式而非一行shell命令，按照Lisp的规则来对它求值，然后打印结果。接着它再继续处理你输入的下一个表达式。正是由于这种无休止的读取、求值和打印的周期变化，因此它被称为读－求值－打印循环（read-eval-print loop），简称REPL。它也被称为顶层（top-level）、顶层监听器（top-level listener）或Lisp监听器。

借助REPL提供的环境就可以定义或重定义诸如变量、函数、类和方法等程序要素，求值任何Lisp表达式，加载含有Lisp源代码或编译后代码的文件，编译整个文件或者单独的函数，进入调试器，单步调试代码，以及检查个别Lisp对象的状态。

所有这些机制都是语言内置的，可以通过语言标准所定义的函数来访问。如果有必要，你只需使用REPL和一个知道如何正确缩进Lisp代码的文本编辑器，就可以构建出一个相当合理的编程环境来。但如果追求真正的Lisp编程体验，则需要一个像SLIME这样的环境，它可以同时通过REPL以及在编辑源文件时与Lisp进行交互。例如，没有必要把一个函数定义从源文件里复制并粘贴到REPL里，也不必因为改变了一个函数就把整个文件重新加载。Lisp环境应该允许编译单独的表达式和直接来自编辑器的整个文件或对其求值。

2.4 体验 REPL

为了测试REPL，需要一个可以被读取、求值和打印的Lisp表达式。最简单类型的Lisp表达式是一个数。在Lisp提示符下，可以输入10，接着敲回车键，然后看到类似下面的东西：

```
CL-USER> 10
10
```

第一个10是你输入的。Lisp 读取器，即REPL中的R，读取文本“10”并创建一个代表数字10的Lisp对象。这个对象是一个自求值（self-evaluating）对象，也就是说当把它送给求值器，即REPL中的E以后，它将对其自身求值。这个值随后被送到打印器里，打印出只有10的那行来。整个过程看起来似乎是费了九牛二虎之力却回到了原点，但如果你给了Lisp更有意义的信息，那么事情就变得有意思一些了。比如说，可以在Lisp提示符下输入(+ 2 3)。

```
CL-USER> (+ 2 3)
5
```

小括号里的东西构成了一个列表，上述列表包括三个元素：符号+，以及数字2和3。一般来说，Lisp对列表求值的时候会将第一个元素视为一个函数的名字，而其他元素作为即将求值的表达式则形成了该函数的实参。在本例里，符号+是加法函数的名字。2和3对自身求值后被传递给加法函数，从而返回了5。返回值5被传递给打印器从而得以输出。Lisp也可能以其他方式对列表求值，但我们现在还没必要讨论它。先从Hello World开始。

2.5 Lisp风格的“Hello, World”

没有“Hello, World”程序[1]的编程书籍是不完整的。事实上，想让REPL打印出“hello, world”再简单不过了。

```
CL-USER> "hello, world"
"hello, world"
```

其工作原理是，因为字符串和数字一样，带有Lisp读取器可以理解的字面语法并且是自求值对象：Lisp读取双引号里的字符串，求值的时候在内存里建立一个可以对自身求值的字符串对象，然后再以同样的语法打印出来。双引号本身不是在内存中的字符串对象的一部分——它们只是语法，用来告诉读取器读入一个字符串。而打印器之所以在打印字符串时带上它们，则是因为其试图以一种读取器可以理解的相同语法来打印对象。

尽管如此，这还不能算是一个“hello, world”*程序*，而更像是一个“hello, world”*值*。

向真正的程序迈进一步的方法是编写一段代码，其副作用可以将字符串“hello, world”打印到标准输出。Common Lisp提供了许多产生输出的方法，但最灵活的是**FORMAT**函数。**FORMAT**接受变长参数，但是只有两个必要的参数，分别代表着发送输出的位置以及字符串。在下一章里你将看到这个字符串是如何包含嵌入式指令，以便将其余的参数插入到字符串里的，就像`printf`或者Python的`string-%`那样。只要字符串里不包含一个~，那么它就会被原样输出。如果将`t`作为第一个参数传入，那么它将会发送其输出到标准输出。因此，一个将输出“hello, world”的**FORMAT**表达式应如下所示。[2]

```
CL-USER> (format t "hello, world")
hello, world
NIL
```

关于**FORMAT**表达式的结果，需要说明的一点是紧接着“`hello, world`”输出后的**NIL**。那个**NIL**是REPL输出的求值**FORMAT**表达式的结果。（**NIL**是Lisp版本的逻辑假和空值。更多内容见第4章。）和目前我们所见到的其他表达式不同的是，**FORMAT**表达式的副作用（在本例中是打印到标准输出）比其返回值更有意义。但Lisp中的每个表达式都会求值出某些结果。[3]

尽管如此，你是否已经写出了一个真正的“程序”呢？恐怕仍有争议。不过你离目标越来越近了。而且你正在体会REPL所带来的自底向上的编程风格：可以试验不同的方法，然后从已经

---

① 在Kernighan和Ritchie的C语言书极大促进了其流行以前，声誉卓著的“hello, world”就已经存在了。最初的“hello, world”似乎来源于*Brian Kernighan的A Tutorial Introduction to the Language B*，收录于1973年1月的*Bell Laboratories Computing Science Technical Report #8: The Programming Language B*。（目前可以在线查阅http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html。）

② 下面是同样可以打印出字符串“hello, world”的其他表达式：

```
(write-line "hello, world")
```

或是：

```
(print "hello, world")
```

③ 不过，当我们讨论到多重返回值的时候，你将看到编写一个求不出值的表达式在理论上是可能的，但即便是这样的表达式，在一个需要其值的上下文环境中也将被视为返回了**NIL**。

测试过的部分里构建出一个解决方案来。现在你已经写出了一个简单表达式来做想做的事，剩下的就是将其打包成一个函数了。函数是Lisp的基本程序构造单元，可以用类似下面的**DEFUN**表达式来定义：

```
CL-USER> (defun hello-world () (format t "hello, world"))
HELLO-WORLD
```

DEFUN后面的hello-world是这个函数的名字。在第4章里我们将看到究竟哪些字符可以在名字里使用，现在我们暂时假设包括-在内的很多在其他语言里非法的字符在Common Lisp里都是合法的。像hello-world这种用连字符而不是下划线（比如hello_world）或是内部大写（比如helloWorld）来形成复合词的方法，是标准的Lisp风格——更不用提那接近正常英语的排版了。名字后面的()是形参列表，在本例中为空，是因为该函数不带参数。其余的部分是函数体。

表面上看，这个表达式和你目前见到的所有其他表达式一样，只是另一个被REPL读取、求值和打印的表达式。这里的返回值是你所定义的函数名。[1]但是和**FORMAT**表达式一样，这个表达式的副作用比其返回值更有用。但与**FORMAT**表达式所不同的是，它的副作用是不可见的：当这个表达式被求值的时候，一个不带参数且函数体为(format t "hello, world")的新函数会被创建出来并被命名为HELLO-WORLD。

一旦定义了这个函数，你就可以像这样来调用它：

```
CL-USER> (hello-world)
hello, world
NIL
```

你将看到输出和直接对**FORMAT**表达式求值时是一样的，包括REPL打印出的**NIL**值。Common Lisp中的函数自动返回其最后求值的那个表达式的值。

2.6 保存工作成果

你可能会争辩说，这就是一个完整的“hello, world”程序了，但还有一个问题。如果退出Lisp然后重启，函数定义将会丢失。这么好的一个函数，你可能想要保存下来。

这很简单。只需创建一个文件，然后把定义保存在里面即可。在Emacs中可以通过输入C-x C-f来创建一个新文件，然后根据Emacs的提示输入文件的名字。文件存放的位置并不重要。Common Lisp源文件习惯上带有.lisp扩展名，尽管有些人用.cl来代替。

一旦创建了文件，就可以向其中写入之前在REPL里输入过的定义。需要注意的是，在输入了开括号和单词**DEFUN**以后，在Emacs窗口的底部，SLIME将会提示它所期待的参数。具体的形式将取决于所使用的具体Common Lisp实现，但其形式可能会如下所示。

```
(defun name varlist &rest body)
```

在开始输入每一个新的列表元素时，这个信息就会消失，但当每次输入了空格以后，它又会重新出现。在文件中输入这个定义的时候，你可能选择将函数从形参列表那里打断成两行。如果

① 我将在第4章里解释为何所有的名字都被转换成大写的。

输入回车然后按Tab键，SLIME将自动把第二行缩进到合适的位置，如下所示。[1]

```
(defun hello-world ()
  (format t "hello, world"))
```

SLIME也会帮助匹配括号——当输入了闭括号时，它将闪烁显示对应的开括号。也可以输入C-c C-q来调用命令slime-close-parens-at-point，它将插入必要数量的闭括号以匹配当前的所有开括号。

可用几种方式将这个定义输入到Lisp环境中。最简单的是，当光标位于DEFUN定义内部的任何位置或者刚好在其后面时，输入C-c C-c，这将启动slime-compile-defun命令，将当前定义发给Lisp进行求值并编译。为了确认这个过程有效，你可以对hello-world作些修改，重新编译它然后回到REPL，使用C-c C-z或者C-x b再次调用它。例如，你可以使其更合乎语法。

```
(defun hello-world ()
  (format t "Hello, world!"))
```

接下来，用C-c C-c进行重新编译，然后输入C-c C-z来切换到REPL试一下新版本。

```
CL-USER> (hello-world)
Hello, world!
NIL
```

你可能需要保存工作文件。在hello.lisp缓冲区里，输入C-x C-s可以启动Emacs命令save-buffer。

现在尝试从源文件中重新加载这个函数，这需要退出并重启Lisp环境。执行退出操作可以使用一个SLIME快捷键：在REPL中输入一个逗号。在Emacs窗口底部，将提示你输入一个命令。输入quit（或sayoonara），然后按回车。这将退出Lisp并且关闭所有SLIME创建的缓冲区，包括REPL缓冲区。[2]现在用M-x slime重启SLIME。

可以顺便试试直接调用hello-world。

```
CL-USER> (hello-world)
```

此时SLIME将弹出一个新的缓冲区并带有类似下面的内容：

```
attempt to call 'HELLO-WORLD' which is an undefined function.
   [Condition of type UNDEFINED-FUNCTION]
Restarts:
  0: [TRY-AGAIN] Try calling HELLO-WORLD again.
  1: [RETURN-VALUE] Return a value instead of calling HELLO-WORLD.
  2: [USE-VALUE] Try calling a function other than HELLO-WORLD.
  3: [STORE-VALUE] Setf the symbol-function of HELLO-WORLD and call it again.
  4: [ABORT] Abort handling SLIME request.
  5: [ABORT] Abort entirely from this process.
Backtrace:
  0: (SWANK::DEBUG-IN-EMACS #<UNDEFINED-FUNCTION @ #x716b082a>)
  1: ((FLET SWANK:SWANK-DEBUGGER-HOOK SWANK::DEBUG-IT))
```

① 你也可以在REPL里将定义输入成两行，因为REPL读取整个表达式，而不是按行。

② SLIME快捷键并不是Common Lisp的一部分，它们是SLIME的命令。

```
  2: (SWANK:SWANK-DEBUGGER-HOOK #<UNDEFINED-FUNCTION @ #x716b082a> ➥
#<Function SWANK-DEBUGGER-HOOK>)
  3: (ERROR #<UNDEFINED-FUNCTION @ #x716b082a>)
  4: (EVAL (HELLO-WORLD))
  5: (SWANK::EVAL-REGION "(hello-world)
" T)
```

天哪！发生了什么事？原来，你试图调用了一个不存在的函数。不过尽管输出了很多东西，Lisp实际上正在很好地处理这一情况。跟Java或者Python不同，Common Lisp不会只是放弃——抛出一个异常并从栈上退回，而且它也绝对不会仅仅因为调用了一个不存在的函数就开始做核心转储（core dump）。事实上Lisp会转到调试器。

在调试器中时，你仍然拥有对Lisp的完全访问权限，所以可以通过求值表达式来检查程序的状态，甚至可以直接修复一些东西。不过目前先别担心它们。直接输入q退出调试器，然后回到REPL里。调试器缓冲区将会消失，而REPL将显示：

```
CL-USER> (hello-world)
; Evaluation aborted
CL-USER>
```

相比直接中止调试器，在它里面显然可以做更多的事情——例如我们将在第19章里看到调试器是如何与错误处理系统集成在一起的。尽管如此，目前最重要的是，要知道总是可以通过按q来退出并回到REPL里。

回到REPL里可以再试一次。问题在于Lisp不知道`hello-world`的定义，因此你需要让Lisp知道保存在hello.lisp文件中的定义。有几种方式可以做到这一点。可以切换回含有那个文件的缓冲区（使用C-x b，在其提示时输入hello.lisp），然后就像之前那样用C-c C-c重新编译那个定义。或者可以加载整个文件，当文件里含有大量定义时，这将更加便利，在REPL里像这样使用**LOAD**函数：

```
CL-USER> (load "hello.lisp")
; Loading /home/peter/my-lisp-programs/hello.lisp
T
```

那个T表示文件被正确加载了。[①]使用**LOAD**加载一个文件，本质上等价于以文件中出现的顺序在REPL下逐个输入每一个表达式，因此在调用了**LOAD**之后，`hello-world`就应该有定义了：

```
CL-USER> (hello-world)
Hello, world!
NIL
```

另一种加载文件中有用定义的方法是，先用**COMPILE-FILE**编译，然后再用**LOAD**加载编译后产生的文件，也就是FASL文件——快速加载文件（fast-load file）的简称。**COMPILE-FILE**将返回FASL文件的名字，所以我们可以在REPL里像下面这样进行编译和加载：

① 如果由于某种原因，**LOAD**没有正常执行，你将得到另一个错误并退回到调试器。如果发生了这样的事，多半是由于Lisp没有找到那个文件，而这可能是因为Lisp所认为的当前工作目录和你文件所在的位置不一样。这种情况下，你可以键入q退出调试器，然后使用SLIME快捷命令cd来改变Lisp所认为的当前目录——输入一个逗号，然后在提示命令时输入cd，以及hello.lisp被保存到的目录名。

```
CL-USER> (load (compile-file "hello.lisp"))
;;; Compiling file hello.lisp
;;; Writing fasl file hello.fasl
;;; Fasl write complete
; Fast loading /home/peter/my-lisp-programs/hello.fasl
T
```

SLIME还支持不使用REPL来加载和编译文件。在一个源代码缓冲区时，你可以使用C-c C-l调用命令slime-load-file来加载文件。Emacs将会提示你给出要加载的文件名，同时将当前的文件名作为默认值，直接回车就可以了。或者可以输入C-c C-k来编译并加载那个当前缓冲区所关联的文件。在一些Common Lisp实现里，对代码进行编译将使其速度更快一些；在其他实现里可能不会，因为它们总是编译所有东西。

这些内容应该足够给你一个关于Lisp编程如何工作的大致印象了。当然我还没有涉及所有的技术和窍门，但你已经见到其本质要素了——通过与REPL的交互来尝试一些东西，加载和测试新代码，调整和调试它们。资深的Lisp黑客们经常会保持一个Lisp映像日复一日地运行，不断地添加、重定义和测试他们的程序。

同样地，甚至当部署Lisp程序以后，往往仍有一种方式可以进入REPL。第26章将介绍如何使用REPL和SLIME来跟正在运行一个Web服务器的Lisp进行交互，同时它还在伺服Web页面。甚至有可能用SLIME连接到运行在另一台不同机器里的Lisp，从而允许你像本地环境那样去调试一个远程服务器。

一个更加令人印象深刻的案例是1998年发生在NASA的Deep Space 1任务中的远程调试。在宇宙飞船升空半年以后，一小段Lisp代码正准备控制飞船以进行为期两天的一系列实验。不幸的是，代码里的一个难以察觉的静态条件逃过了地面测试期间的检测并且已经升空了。当这个错误在距地球一亿英里外的地方出现时，地面团队得以诊断并修复了运行中的代码，使得实验顺利地完成。[①]一个程序员如此描述了这件事：

调试一个运行在一亿英里之外且价值一亿美元硬件上的程序是件有趣的经历。一个运行在宇宙飞船上的读－求值－打印循环，在查找和修复这个问题的过程中，真是无价之宝啊。

你还没有准备好将任何Lisp代码发送到太空，不过在接下来一章里，你就将亲身参与编写一个比“hello, world”更有趣一点儿的程序了。

① http://www.flownet.com/gat/jpl-lisp.html。