第32章 结论：下一步是什么

到目前为止，我希望你终于明白本书的书名并不矛盾了。不过，确实有一些领域对你的编程实践非常重要而我并未谈及。例如，我没有提到如何开发图形用户界面（GUI），如何连接关系型数据库，如何解析XML或是如何编写用作多种网络协议客户端的程序。类似地，当你用Common Lisp来编写实际的应用程序时，还有两个非常重要的主题尚未讨论：优化Lisp代码，为交付而打包应用程序。

显然，我并不打算在这最后的一章里深入讨论所有这些主题，我只想给你一些指点，以便你去探索Lisp编程中你最关注的方面。

32.1 查找 Lisp 库

尽管Common Lisp自带的函数、数据类型和宏的标准库规模宏大，但它只提供了通用的编程构造。诸如编写GUI、与数据库进行通信以及解析XML之类的特定任务，都需要用到ANSI标准化语言之外所提供的第三方库。

要获取一个用来做你想做的事情的库，最简单方式可能就是简单地查看Lisp实现。多数实现至少提供了Lisp语言标准里没有指定的一些功能。Common Lisp厂商为证明其价值，尤其倾向于提供用于其实现的附加库。例如，Franz的Allegro Common Lisp企业版就自带了一些库，用于解析XML、进行SOAP通信、生成HTML、连接关系型数据库以及用多种方式构建图形界面。另一个卓越的商业Lisp平台LispWorks也提供了几个类似的库，包括广泛使用的可移植的GUI工具箱CAPI，用来开发能够运行在任何支持LispWorks的操作系统之上的GUI应用程序。

免费和开源的Common Lisp实现通常不包含许多打包的库，而是依赖于可移植的免费和开源库。但是，即便是那些实现，通常也会支持一些语言标准没有涉及的重要领域，例如网络和多线程。

使用与具体实现相关的库的唯一缺点是，它们将你捆绑在了提供这些库的实现上。如果你正在交付面向最终用户的应用程序，或者正在一台你所控制的服务器上部署基于服务器的应用程序，那么这可能不是什么大问题。但如果你想要编写可以共享给其他Lisp程序员的代码，或者只是简单地不想把自己捆绑在特定实现上，那么就有点恼人了。

对于可移植的库，其可移植性要么来源于它们完全是用标准的Common Lisp写成的，要么是

因为它们包含了适当的读取期条件化，从而可以工作在多个实现上[①]，获得它们的最佳途径就是借助Web。虽然说URL变化很快，通常会在打印出来后立即失效，不过可以将下面三个作为当前的最佳起始点。

- Common-Lisp.net （http://www.common-lisp.net/）是一个含有免费和开源Common Lisp项目的站点，它提供了版本控制、邮件列表以及项目页面的Web服务。在该站点启动后的一年半里，注册了将近100个项目。
- Common Lisp Open Code Collection （CLOCC），（http://clocc.sourceforge.net/）是一个较早的免费软件库聚集地，目的是在各个Common Lisp实现之间做到可移植，并试图不依赖于任何未包含在CLOCC的库。
- Cliki （http://www.cliki.net/）是一个提供用Common Lisp编写的各种免费软件的Wiki站点。尽管与其他任何Wiki站点一样，它可能随时发生变化，但它通常可以链接到相当多的库和多种开源Common Lisp实现。该站点运行所依赖的软件也是用Common Lisp写成的。

运行Debian或Gentoo发行版的Linux用户也可以轻松地安装越来越多的Lisp库，这些库与发行版的打包工具，即Debian的`apt-get`和Gentoo的`emerge`，打包在了一起。

我在这里不会推荐具体的库，因为这些库的情况日新月异。在对Perl、Python和Java的库集合觊觎了多年以后，Common Lisp程序员在过去的几年里也终于开始勇敢地接受挑战，开始为Common Lisp提供应有的开源及商业库了。

近年非常活跃的一个领域是GUI前端。Common Lisp不像Java和C#，但却跟Perl、Python和C的情况相似，不存在在开发GUI的单一方法。相反，这项工作依赖于你所使用的发行版和你想要支持的操作系统。

商业的Common Lisp实现通常提供了某种方式在它们支持的平台上构建GUI。其中LispWorks还提供了CAPI，前面提到过，这是一种可移植的GUI API。

在开源领域，你有几种选择。在Unix上，你可以使用CLX来编写底层的X-Window GUI，CLX是X-Window协议的一个纯Common Lisp的实现，几乎等价于C的xlib库。或者你可以使用几种对GTK和Tk这类高层次API和工具箱的绑定，这与你在Perl和Python里的工作方式差不多。

或者，如果你在寻找一些完全不同的工具，可以看一下Common Lisp Interface Manager (CLIM)。作为Symbolics Lisp Machine GUI框架的后裔，CLIM既强大又复杂。尽管事实上许多商业Common Lisp实现支持它，但并未见其被大量使用。不过在过去的几年里，一种CLIM的开源实现，McCLIM——目前放在Common-Lisp.net上，正在蓬勃地发展，因此也许我们正处于CLIM复苏的边缘。

① Common Lisp的读取期条件化和宏使得开发可移植库成为可能，这些库本身只是为了在不同实现中为语言标准没有指定的那些功能提供的API之上提供一个通用的API层。第15章的可移植路径名库就是这类库的一个例子，它在具体实现相关的API上尽可能地消除了对于语言标准的不同解释。

32.2　与其他语言接口

尽管许多有用的库都可以只用语言标准里指定的特性，以“纯粹的”Common Lisp来编写，并且还有更多的库可以使用由给定实现所提供的非标准功能用Lisp写出来，但有时使用来自C语言等其他语言的库会更为直接。

语言标准并未指定一种机制让Lisp代码得以调用其他语言写成的代码，甚至也没有要求具体实现提供这样一种机制。不过近年来，几乎所有的Common Lisp发行版都支持所谓的“外部函数接口”（Foreign Function Interface 或简称FFI）。[①]FFI的基本工作是让你给Lisp足够的信息以便其可以链接外部代码。因此，如果你打算调用一个来自某C库的函数，就需要告诉Lisp如何将传递给该函数的Lisp对象转化成C类型，然后再将该函数的返回值转化回Lisp对象。不过，每个实现都提供了它们自己的FFI，每种FFI都有稍微不同的功能和语法。某些FFI允许从C向Lisp回调，而另一些则不允许。Universal Foreign Function Interface（UFFI）项目提供了七八种不同Common Lisp实现上的可移植的FFI兼容层。它的工作方式是定义自己的宏，然后展开成其具体实现上的适当FFI代码。UFFI的思路是选取最底层的共同特征，这意味着它无法充分利用不同实现的FFI的所有特性，但它确实提供了一种构建基本C API外围Lisp封装的好方法。[②]

32.3　让它工作，让它正确，让它更快

人们总是说，并分别由Donald Knuth、C.A.R. Hoare和Edsger Dijkstra强调过，过早地进行优化是万恶之源。[③]Common Lisp是一门杰出的语言，使用它你在拥有充分表达能力的同时还可以得到很高的性能。如果你曾经听到过有关Lisp很慢的传言的话，可能会感到惊奇。在Lisp的早期岁月里，当计算机还在使用穿孔卡片来编程时，Lisp的高级特性可能确实让其慢于竞争对手，也即汇编语言和FORTRAN。但那已经是很久以前的事情了。在同一时期，Lisp曾经被用于从创建复杂AI系统到编写操作系统等一系列任务中，并且大量的工作被用来找出如何将Lisp编译成高效的代码。在本节里，我将讨论为什么说Common Lisp是用来编写高性能代码的杰出语言，并介绍相关的一些技术。

带有讽刺意味的是，说Lisp是一门用来编写高性能代码的优秀语言，首要原因正是Lisp编程的动态本质——这正是当初使Lisp的性能难以达到FORTRAN编译器水平的原因。Common Lisp的动态特性使其易于编写高性能代码，因为编写高效代码的第一步是要找到正确的算法和数据结构。

---

① 外部函数接口基本上等价于Java中的JNI，Perl中的XS或者Python中的扩展模块API。

② 在写这本书时，UFFI的两大缺点是缺少对从C到Lisp回调的支持（很多但并非全部实现的FFI都支持），以及缺少对CLISP的支持。后者的FFI很好但与其他实现的区别很大，以至于无法轻易集成到UFFI模型之中。

③ Knuth过去曾在其出版物中说过许多次，包括在他的1974年ACM图灵奖论文《作为艺术的计算机编程》和他的论文《带有goto语句的结构化程序》中。在他的论文《TeX的错误》中，他将该说法归功于 C.A.R. Hoare。而Hoare在一封于2004年发给phobia.com的Hans Genwitz的电子邮件中说他不记得这一说法的起源了，但他可以将其归功于Dijkstra。

Common Lisp的动态特性确保了代码的灵活性，这使其易于尝试不同的方法。给定有限的时间来编写一个程序，如果你不必花很多时间出入死胡同的话，多半会写出一个高性能的版本来。在Common Lisp中，你可以尝试一种思路，如果发现其不可行就立即切换到下一个，而不必花费大量时间来使编译器得以通过你的代码并等待编译完成。你可以先写出一个函数的某个直接而低效的版本（一份代码草图）来检查你的基本思路是否可行，如果是的话再将该函数替换成一个复杂但却高效的实现。并且就算发现整个思路存在问题，你也不必将时间浪费在调整一个不再需要的函数上，这意味着你有更多的时间来找出一个更好的方法。

说Common Lisp是用于开发高性能软件的优秀语言，第二个原因在于，大多数Common Lisp实现都带有成熟的编译器，可产生相当高效的机器码。我将很快谈及如何帮助这些编译器来生成可与C编译器生成的代码相媲美的代码，但不作改进的这些实现已经比那些不成熟的实现或使用更简单的编译器或解释器的语言快得多了。另外，由于Lisp编译器在运行期可用，因此Lisp程序员拥有一些其他语言难以比拟的可能性——程序可以在运行期生成Lisp代码，然后再被编译成机器码来运行。如果生成的代码打算运行足够多次的话，那么由此带来的好处将是巨大的。或者，即便不使用运行期的编译器，闭包也给了你另一种将机器码与运行期数据混合使用的方式。例如，CL-PPCRE正则表达式库运行在CMUCL上的时候，在某些基准测试上比Perl的正则表达式引擎还要快，即便Perl的引擎是用高度优化的C写成的。这在很大程度上是因为在Perl中一个正则表达式被转化成了本质上是字节码的东西，然后被插入到了正则引擎中，而CL-PPCRE直接将正则表达式转译成了编译了的闭包树，这些闭包通过正常的函数调用机制来调用彼此。[1]

不过，即便使用了正确的算法和高质量的编译器，你可能仍然无法达到你所需要的原始速度。那么接下来就要思考性能分析和调优了。在Lisp中，和在任何语言中一样，关键在于首先要进行分析以找到你程序中最花时间的性能瓶颈，然后再考虑如何让这些部分提速。[2]

你有许多种方式来完成性能分析。语言标准提供了一些基本的工具用于测量特定的形式在执行时花了多长时间。特别是，**TIME**宏可以包装在任何形式之外，并返回由其形式所返回的任何值，不过在这之前它会向***TRACE-OUTPUT***打印一条消息，指出它花费了多长时间来运行，以及它使用了多少内存。该消息的确切形式是由具体实现定义的。

可以使用**TIME**来完成一些简单粗暴的分析，缩小你搜索性能瓶颈的范围。例如，假设你有一个花费很长时间来运行的函数，并且它还调用了其他两个类似下面的函数：

```
(defun foo ()
  (bar)
  (baz))
```

① CL-PPCRE还利用了另一个我没有讨论过的Common Lisp特性，即编译器宏（compiler macro）。编译器宏是一个特殊类型的宏，它提供了优化一个特定的函数调用的机会，方法是将对该函数的调用转化成更高效的代码。CL-PPCRE为其接受正则表达式参数的函数定义了编译器宏。这个编译器宏通过在编译期解析正则表达式来优化那些带有常量正则表达式的函数调用，而不是将它们留到运行期再处理。关于编译器宏的更多信息，参见你喜爱的任何一本Common Lisp参考书中的**DEFINE-COMPILER-MACRO**部分。

② “过早地优化”中的词汇“过早”完全可以被定义成“在分析之前”。请记住就算你可以将一些代码的速度提高到几乎不需要花时间的程度，你的整个程序的性能提升也仅限于那段代码在程序运行时间中所占的比率。

如果你想要查看bar和baz究竟哪一个花了更多的时间，那么可以将foo的定义改成下面这样：

```
(defun foo ()
  (time (bar))
  (time (baz)))
```

现在你可以调用foo了，而Lisp将会打印出两份报告，一个是bar的，另一个是baz的。具体的格式是与实现相关的。下面是它们在Allegro Common Lisp中的样子：

```
CL-USER> (foo)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time  105 msec
; space allocation:
;  24,172 cons cells, 1,696 other bytes, 0 static bytes
; cpu time (non-gc) 540 msec user, 10 msec system
; cpu time (gc)     170 msec user, 0 msec system
; cpu time (total)  710 msec user, 10 msec system
; real time  1,046 msec
; space allocation:
;  270,172 cons cells, 1,696 other bytes, 0 static bytes
```

当然，如果输出中带有一个标签的话就更易于阅读了。如果你大量使用这种技术，也许值得去定义一个类似下面这样的宏：

```
(defmacro labeled-time (form)
  `(progn
    (format *trace-output* "~2&~a" ',form)
    (time ,form)))
```

如果你在foo中将**TIME**替换成labeled-time，那么将得到下面的输出：

```
CL-USER> (foo)

(BAR)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time  131 msec
; space allocation:
;  24,172 cons cells, 1,696 other bytes, 0 static bytes

(BAZ)
; cpu time (non-gc) 490 msec user, 0 msec system
; cpu time (gc)     190 msec user, 10 msec system
; cpu time (total)  680 msec user, 10 msec system
; real time  1,088 msec
; space allocation:
;  270,172 cons cells, 1,696 other bytes, 0 static bytes
```

从这些输出中，很明显可以看出foo的大多数时间都花在了baz上。

当然，如果你想要分析的Lisp形式被重复调用的话，那么来自**TIME**的输出就显得笨拙了。你可以使用函数**GET-INTERNAL-REAL-TIME**和**GET-INTERNAL-RUN-TIME**来构造你自己的测量工

具，它们返回一个按照常量INTERNAL-TIME-UNITS-PER-SECOND的值逐秒递增的数值。GET-INTERNAL-REAL-TIME测量的是“挂钟时间”，也就是实际流逝的时间量，而GET-INTERNAL-RUN-TIME则测量某种具体实现所定义的值，例如Lisp实际执行的时间量，或是Lisp执行用户代码而不包括垃圾收集等内部例行公事在内的时间量。下面是一个简单而有用的分析工具，它使用了一些宏和GET-INTERNAL-RUN-TIME：

```
(defparameter *timing-data* ())

(defmacro with-timing (label &body body)
  (with-gensyms (start)
    `(let ((,start (get-internal-run-time)))
       (unwind-protect (progn ,@body)
         (push (list ',label ,start (get-internal-run-time)) *timing-data*)))))

(defun clear-timing-data ()
  (setf *timing-data* ()))

(defun show-timing-data ()
  (loop for (label time count time-per %-of-total) in (compile-timing-data) do
        (format t "~3d% ~a: ~d ticks over ~d calls for ~d per.~%"
                %-of-total label time count time-per)))

(defun compile-timing-data ()
  (loop with timing-table = (make-hash-table)
     with count-table = (make-hash-table)
     for (label start end) in *timing-data*
     for time = (- end start)
     summing time into total
     do
       (incf (gethash label timing-table 0) time)
       (incf (gethash label count-table 0))
     finally
       (return
         (sort
          (loop for label being the hash-keys in timing-table collect
                (let  ((time (gethash label timing-table))
                       (count (gethash label count-table)))
                  (list label time count
                        (round (/ time count)) (round (* 100 (/ time total))))))
          #'> :key #'fifth))))
```

这个分析器可以让你将with-timing包装在任意Lisp形式之外。每当该形式被执行时，其开始和结束的时刻都将记录下来并关联到你提供的标签上。函数show-timing-data可以输出一个表，显示在带有不同标签的代码段中分别花费的时间，如下所示：

```
CL-USER> (show-timing-data)
 84% BAR: 650 ticks over 2 calls for 325 per.
 16% FOO: 120 ticks over 5 calls for 24 per.
NIL
```

你可以明显地从多个角度让这段分析代码变得更专业。此外，你的Lisp实现也很有可能提供了它自己的分析工具，并且由于它们具有对实现内部的访问权限，因此能够得到对用户层代码未

必可见的一些信息。

你一旦在代码中发现了瓶颈所在，就可以开始设法调优了。当然，你应当尝试的第一件事是寻找一个更有效的基本算法，这是获得最大性能提升的最佳方法。但假设你已经使用了一个适当的算法，那么接下来就是“代码精修”阶段了，即局部优化你的代码，让其绝对不做任何多余的工作。

Common Lisp中用于代码精修的主要工具是它的各种可选的声明。Common Lisp声明背后的基本思想是，它们用来向编译器提供信息以帮助其生成更好的代码。

举一个简单的例子，考虑下面这个Common Lisp函数：

```
(defun add (x y) (+ x y))
```

我在第10章里提到过，如果你比较这个Lisp函数和看起来等价的C函数之间的性能：

```
int add (int x, int y) { return x + y; }
```

那么你很可能会发现Common Lisp版本慢很多，即便当你的Common Lisp实现号称是带有高质量的原生编译器时。

这是因为Common Lisp版本的函数做了太多的事——Common Lisp编译器甚至不知道a和b的值是数字，因此必须生成代码在运行期检查。并且一旦检测出它们确实是数字，它还需要检测这些数字的类型是整数、比值、浮点数还是复数，然后派发到适当的用于实际类型的加法程序上。并且就算a和b都是整数（你所关心的情形），加法程序也不得不考虑加法的结果可能过大而无法表示成一个fixnum，即单个机器字可表示的数，从而可能不得不分配一个bignum对象。

而在C语言中，由于所有变量的类型都是声明了的，编译器精确地知道a和b将保存何种类型的值。并且由于C语言中的算术在加法的结果过大而无法用返回值的类型表示时只是简单地溢出，不做任何溢出检测，也不会在数学意义上的和过大而无法填入单个机器字时分配一个bignum对象。

这样，尽管Common Lisp代码的行为更有可能从数学意义上讲是正确的，但是C版本很可能直接被编译成一两个机器指令。不过，如果你愿意为Common Lisp编译器提供跟C编译器同样的信息，包括参数和返回值的类型，以及在通用性和错误检查方面接受类似C的妥协的话，那么Common Lisp函数也可以被编译成一两个指令。

这就是声明的用处。声明的主要用法是为了告诉编译器关于变量和其他表达式的类型。例如，你可以通过编写下面的函数来告诉编译器，add的两个参数都是fixnum：

```
(defun add (x y)
  (declare (fixnum x y))
  (+ x y))
```

其中的**DECLARE**表达式并非Lisp形式，相反，它是**DEFUN**语法的一部分，并且必须出现在函数体中其他任何代码之前。[1]该声明声称为形参x和y传递的参数将总是fixnum的。换句话说，这是一

① 声明可以出现在引入新变量的多数Lisp形式中，例如**LET**、**LET***和**DO**家族的循环宏。**LOOP**有其自己的用于声明循环变量类型的语法。第20章里提过的特殊操作符**LOCALLY**，就是专门用来创建一个可书写声明的作用域的。

个对编译器的承诺，并且编译器被允许生成假设你所言为真的代码。

为了声明返回值的类型，你可以将形式(+ x y)包装在**THE**特殊操作符中。该操作符接受一个诸如**FIXNUM**这样的类型说明符和一个形式，从而告诉编译器该形式将求值到给定的类型上。这样，为了给Common Lisp编译器相当于C编译器得到的所有关于add的信息，你可以写成下面这样：

```
(defun add (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

不过，即便是这个版本也需要更多的一个声明，才能让Common Lisp编译器像C编译器那样生成快速但危险的代码。**OPTIMIZE**声明被用来告诉编译器如何平衡5个量：被生成的代码的速度；运行期错误检查的程度；代码的内存使用，包括代码本身的大小和运行期的内存占用；随代码提供的调试信息的数量；编译过程本身的速度。**OPTIMIZE**声明由一个或多个列表构成，其中每个列表都含有符号**SPEED**、**SAFETY**、**SPACE**、**DEBUG**和**COMPILATION-SPEED**中的一个，以及一个从0到3的数值，包括0和3在内。该数值指定了编译器应当给予对应量的相对权重，其中3代表最重要，而0意味着完全不重要。这样，为了让Common Lisp将add编译到跟C编译器差不多的程度，你可以写成下面这样：

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

当然，现在这个Lisp版本将只承担相当于C版本的义务了。如果传递的参数不是fixnum或者加法溢出了，那么结果将是数学上错误的或者更坏。另外，如果有人使用错误的参数数量来调用add，那么结果可能也会很糟。因此，你应该仅在你的程序已经正常工作以后才使用这类声明，并且应该只在性能分析表明它们可以带来不同的效果时才添加它们。如果没有它们的时候也能得到合理的性能，那么就不要使用它们。但如果性能分析显示你的代码中有一个真正的热点并且你需要对其调优，那么就放手去做吧。由于你可以这样去使用声明，因而很少只出于性能考虑来用C重写代码。FFI可以用来访问已有的C代码，但声明可在需要接近C的性能时使用。当然，你希望给定的一段Common Lisp代码在多大程度上接近于C和C++的性能，完全取决于你想让它们有多像C代码。

Lisp中内置的另一个代码调优工具是函数**DISASSEMBLE**。该函数的确切行为是与实现相关的，因为它依赖于具体实现编译代码的方式——是否编译成机器码、字节码或其他的某种形式。但其基本思想是，它可以向你展示编译一个指定的函数后所生成的代码。

于是，你可以使用**DISASSEMBLE**来查看声明是否对生成的代码产生了任何效果。并且如果你的Lisp实现使用了一个原生编译器，同时你还懂你所在平台上的汇编语言的话，那么就可以精确地看到当你调用一个函数时究竟发生了什么。例如，你可以使用**DISASSEMBLE**来感受没有声明的第一个版本的add和最终版本之间的区别。首先，定义并编译最初的版本。

```
(defun add (x y) (+ x y))
```

然后在REPL中用该函数的名字来调用**DISASSEMBLE**。在Allegro中，它可以显示类似下面的由编译器所生成的类似汇编语言的代码输出：

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x737496f4:
   0: 55          pushl  ebp
   1: 8b ec       movl   ebp,esp
   3: 56          pushl  esi
   4: 83 ec 24    subl   esp,$36
   7: 83 f9 02    cmpl   ecx,$2
  10: 74 02       jz     14
  12: cd 61       int    $97   ; SYS::TRAP-ARGERR
  14: 80 7f cb 00 cmpb   [edi-53],$0        ; SYS::C_INTERRUPT-PENDING
  18: 74 02       jz     22
  20: cd 64       int    $100  ; SYS::TRAP-SIGNAL-HIT
  22: 8b d8       movl   ebx,eax
  24: 0b da       orl    ebx,edx
  26: f6 c3 03    testb  bl,$3
  29: 75 0e       jnz    45
  31: 8b d8       movl   ebx,eax
  33: 03 da       addl   ebx,edx
  35: 70 08       jo     45
  37: 8b c3       movl   eax,ebx
  39: f8          clc
  40: c9          leave
  41: 8b 75 fc    movl   esi,[ebp-4]
  44: c3          ret
  45: 8b 5f 8f    movl   ebx,[edi-113]    ; EXCL::+_2OP
  48: ff 57 27    call   *[edi+39]   ; SYS::TRAMP-TWO
  51: eb f3       jmp    40
  53: 90          nop
; No value
```

很明显，其中做了很多事。如果你熟悉x86汇编语言的话，就很可能看出具体的内容。现在编译下面的带有完整声明的add版本。

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

现在再次反汇编add并查看这些声明是否产生了什么效果。

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x7374dc34:
   0: 03 c2    addl eax,edx
   2: f8         clc
   3: 8b 75 fc movl esi,[ebp-4]
```

```
   6: c3          ret
   7: 90          nop
; No value
```

嗯，看来确实有效果。

32.4 交付应用程序

另一个对于实践有重要意义但我却没有在本书谈论过的主题，是如何交付用Lisp编写的软件。我忽略该主题的主要原因是有许多种方式可以做到这点，并且具体哪一种是最好的取决于你需要交付的软件类型、目标用户以及所使用的Common Lisp实现。在本节里，我将对其中一些不同的选择做一个概括。

如果你在编写打算共享给其他Lisp程序员的代码，那么发行它们的最直接方式就是提供源代码。[①]你可以将一个简单的库以单个源代码文件的形式来发布，然后程序员们可以用**LOAD**将其加载到他们的Lisp映像里，或者有可能先用**COMPILE-FILE**将其编译然后再加载。

那些跨越多个源文件的更加复杂的库和应用提出了更大的挑战——为了加载和编译这些代码，所有文件都需要以正确的顺序来编译和加载。例如，一个含有宏定义的文件必须在你编译任何使用了这些宏的文件之前加载，而一个含有**DEFPACKAGE**形式的文件也必须在用到该包的任何文件甚至成为**READ**之前被加载。Lisp程序员将其称为“系统定义”问题，并通常使用所谓的“系统定义机制”或“系统定义工具”来处理它们，后者类似于make或ant这样的构建工具。跟make和ant相似，系统定义工具允许你指定不同文件之间的依赖关系，并帮助你以正确的顺序来加载和编译文件，其间试图只做必要的工作，例如只重新编译那些发生了改变的文件。

近年来，最为广泛使用的系统定义工具是ASDF，它的全称是Another System Definition Facility。[②]ASDF背后的基本思想是，你在ASD文件中定义系统，然后ASDF提供了一些系统上的操作，包括加载或编译它们等。一个系统也可以定义成依赖于其他的系统，后者将在必要时被加载。例如，下面给出了html.asd的内容，它是来自第31章和第32章的FOO库的ASD文件：

```
(defpackage :com.gigamonkeys.html-system (:use :asdf :cl))
(in-package :com.gigamonkeys.html-system)

(defsystem html
  :name "html"
  :author "Peter Seibel <peter@gigamonkeys.com>"
  :version "0.1"
  :maintainer "Peter Seibel <peter@gigamonkeys.com>"
  :license "BSD"
```

① **COMPILE-FILE**所产生的FASL文件是与实现相关的，并且不一定能在同一个Common Lisp实现的不同版本间兼容。这样，使用它们就不是分发Lisp代码的一种良好方式了。它们可用于为特定实现的已知版本里运行的应用程序提供补丁。追加一个补丁的方法就是加载这个FASL文件，而由于FASL可以包含任意代码，它可被用来通过提供新的代码定义来升级已有的数据。

② ASDF最初是由SBCL的开发者之一Daniel Barlow所编写的，并且长久以来就是SBCL的一部分，也以独立库的形式来分发。它最近已经被采纳并包含在诸如OpenMCL和Allegro等其他实现中。

```
:description "HTML and CSS generation from sexps."
:long-description ""
:components
((:file "packages")
 (:file "html" :depends-on ("packages"))
 (:file "css" :depends-on ("packages" "html")))
:depends-on (:macro-utilities))
```

如果你在变量asdf:*central-registry*[1]中所列出的目录里添加了一个到该文件的符号链接，那么你就可以通过键入

```
(asdf:operate 'asdf:load-op :html)
```

来以正确的顺序编译和加载文件packages.lisp、html.lisp以及html-macros.lisp，并首先保证:macro-utilities系统已被编译和加载。对于其他的ASD文件示例，你可以查看本书的源代码，即来自每个实用章节的代码都被定义成一个系统，并带有ASD文件中表达的适当的跨系统依赖关系。

你将发现大多数免费和开源的Common Lisp库都带有一个ASD文件。它们中的一些还有可能使用其他系统定义工具，例如相对比较古老的MK:DEFSYSTEM，或者甚至是库作者自己设计的工具，但整个流行趋势是向ASDF发展的。[2]

当然，尽管ASDF让Lisp程序员可以容易地安装Lisp库了，但它对于你想要给不了解或不关心Lisp的最终用户打包一个应用程序却不能带来多大的帮助。如果你正在分发一个纯面向最终用户的应用程序，那么你必定要提供一些东西，让用户可以下载、安装和运行而无需知道任何有关Lisp的知识。你不能期待他们能独立地下载并安装一个Lisp实现。并且你还应该让他们能像运行其他应用程序一样运行你的程序——通过双击Windows或OS X上的一个图标，或者在Unix命令行下输入该程序的名字即可。

不过，C程序通常依赖于作为操作系统一部分的那些构成C“运行时环境”的特定共享库（在Windows上是DLL），Lisp程序与此不同，它必须包含一个Lisp运行时环境，也就是当你启动Lisp时运行的那个程序，其中的某些功能可能是你的应用程序所不需要的。

更复杂的问题在于，“程序”这个概念在Lisp中并没有很好的定义。正如你在本书中所看到的，在Lisp中开发软件是一个不断修改你的Lisp映像中的定义和数据集的增量过程。“程序”只是映像所达到的一个特定状态，通过加载含有创建适当定义和数据的代码的.lisp或.fasl文件来改变。随后你可以将Lisp应用程序分发成一个Lisp运行时环境外加一组FASL文件，以及一个可执行程序负责启动运行时环境、加载FASL文件并以某种方式调用适当的启动函数。不过，由于事实上加载FASL可能需要花很多时间，尤其是当你需要做一些计算来设置环境的状态时，因此多数Common Lisp实现都提供了一种导出映像文件的方式，即将一个运行中的Lisp环境的状态保存在

① 在不支持符号链接的Windows上，其工作方式略有不同，但也是差不多的。

② 另一个工具ASDF-INSTALL，构建在ASDF和MK:DEFSYSTEM之上，提供了一种从网络上自动下载和安装库的简单方式。学习ASDF-INSTALL的最佳途径是阅读Edi Weitz的“A tutorial for ASDF-INSTALL”（http://www.weitz.de/asdf-install/）。

一个称为“映像文件”（image file）或“核心”（core）的文件里。当一个Lisp运行时环境启动时，它做的第一件事就是加载映像文件，这比通过加载FASL文件来重建所有状态花费的时间要少得多。

正常情况下，这个映像文件是一个只含有语言所定义的标准包和具体实现所提供的附加包的默认映像。但在多数实现里，你都有某种方法可以指定一个不同的映像文件。这样，不必将一个应用程序打包成一个Lisp运行时环境外加一堆FASL，你可以将其打包成一个Lisp运行时环境外加单个映像文件包含你应用程序的所有定义和数据。然后你所需要的就是一个程序，它可以用适当的映像文件来启动Lisp运行时环境，并调用作为该应用程序入口点的那个函数。

这就是事情开始依赖于具体实现和操作系统的地方了。一些Common Lisp实现，尤其是诸如Allegro和LispWorks这样的商业实现，提供了用来构建这样一个可执行程序的工具。例如，Allegro的企业版提供了一个函数`excl:generate-application`，它可以创建出一个目录，其中含有共享库形式的Lisp运行时环境、一个映像文件以及一个用给定映像启动Lisp运行时环境的可执行程序。类似地，LispWorks专业版中的“delivery”机制允许你构建所有程序的单一可执行文件。在Unix上，通过使用多种免费和开源的实现，你也可以从本质上达到同样的效果，只是使用一个shell脚本来开始可能会更容易一些。

在Mac OS X上，事情甚至变得更奇妙了。由于Mac OS X上的所有应用程序都打包成了.app应用程序捆绑（bundle），其本质上就是带有特定结构的一个目录，因此将Lisp应用程序的所有部分打包成一个可以双击的.app应用程序捆绑完全没有任何困难。Mikel Evins的Bosco工具可以让创建OpenMCL上的应用程序的.app捆绑变得更容易。

当然，近年来的另一种分发应用程序的流行方式是将其作为服务器端应用程序。这是Common Lisp真正擅长的方式。你可以选取一种最适合你的操作系统和Common Lisp实现组合，并且不需要担心如何打包面向最终用户的应用程序。并且Common Lisp的交互式调试和开发特性也可以调试和升级一个运行中的服务器，这在那些不够动态的语言里要么根本是不可能的，要么也要求你为此构建大量的特定基础设施才可行。

32.5 何去何从

就这么多了。欢迎来到Lisp的精彩世界。现在你的最佳选择（如果你还没有开始的话）就是开始亲手编写你自己的Lisp代码。选择一个令你感兴趣的项目，然后用Common Lisp来完成它。然后再做另一个。就这样不断地进行下去。

不过，如果你还需要更进一步的指点，本节提供了一些可供参考的地方。对于初学者来说，可以查看本书位于http://www.gigamonkeys.com/book/的Web站点，这里有那些实用章节的源代码、勘误以及指向Web上其他Lisp资源的链接。

另外，除了我在32.1节所提到的站点之外，你可能还需要浏览Common Lisp HyperSpec（也称为HyperSpec或CLHS），一个ANSI语言标准的HTML版本，它由Kent Pitman制作并通过LispWorks发布在http://www.lispworks.com/documentation/HyperSpec/index.html。HyperSpec并不是

一个学习向导，但在你未从ANSI购买语言标准的打印副本的情况下，它是你可以获得的关于这门语言的权威指导，并且它更适合于日常使用。[①]

如果你想要接触其他Lisp程序员，那么Usenet的com.lang.lisp新闻组和Freenode IRC网络上的#lisp频道就是两个最主要的会面场所。还有一些Lisp相关的博客，其中的多数都被聚合到了位于http://planet.lisp.org/的Planet Lisp站点上。

并且你要保持关注所有这些论坛上关于你所在区域里的本地Lisp用户组的公告——在过去的几年里，Lisp用户群正在世界上的许多城市里陆续出现，从纽约到奥克兰，从科隆到慕尼黑，从日内瓦到赫尔辛基。

如果你想要继续啃书本，那么这里有一些推荐书目。如果你想要一本放在桌面上又厚又精美的参考书，可以选择David Margolies的*ANSI Common Lisp Reference Book*（Apress，2005年）。[②]

想了解Common Lisp对象系统的更多内容，你可以从Sonya E. Keena的*Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*（Addison-Wesley，1989年）开始。然后如果你真的想要成为一个对象技术专家或者只是想要激发灵感的话，可以阅读Gregor Kiczales、Jim des Rivières和Daniel G. Bobrow的*The Art of the Metaobject Protocol*（MIT Press，1991年）。这本书也称为AMOP，它说明了元对象协议是什么以及你为何需要它，并且还描述了一个被许多Common Lisp实现所支持的元对象协议的事实标准。

两本覆盖通用Common Lisp技术的书籍是Peter Norvig的*Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*（Morgan Kaufmann，1992年）以及Paul Graham的*On Lisp: Advanced Techniques for Common Lisp*（Prentice Hall，1994年）。前者提供了对各种人工智能相关技术的全面介绍，其中教授了许多关于如何编写良好Common Lisp代码的内容，而后者在对宏的处理上尤为优秀。

如果你是那种对事情的工作原理究根问底的人，那么Christian Queinnec的*Lisp in Small Pieces*（Cambridge University Press，1996年）提供了编程语言理论和使用Lisp实现技术的完美融合。尽管该书主要集中在Scheme而非Common Lisp上，但两者应用的是同样的原则。

那些更喜欢用理论的眼光来看事物的读者，或者读者只是想知道作为MIT的计算机学院新生是什么感觉，可以去看Harold Abelson、Gerald Jay Sussman和Julie Sussman的*Structure and Interpretation of Computer Programs*第二版（MIT Press，1996年），这是使用Scheme来教授重要编程概念的经典计算机科学文献。每个程序员都可以从该书中受益良多，不过要注意的是Scheme

① SLIME带有一个Elisp库，它允许你自动跳转到任何由标准所定义的名字在HyperSpec中的对应项上。你还可以下载一份HyperSpec的完整副本以实现本地的离线浏览。

② 另一本经典的参考书是Guy Steele的*Common Lisp: The Language*（Digital Press，1984和1990年）。该书的第一版，也称为CLtL1，在很多年里都是该语言的事实标准。在等待官方的ANSI标准完成时，Guy Steele——标准化委员会的成员之一，决定发布第二版以填补CLtL1和未来标准之间的鸿沟。该书的第二版，现在称为CLtL2，在本质上是标准化委员会在接近完成的一个特定时间里的工作的快照，接近但并不等于最终的标准。因此，CLtL2与最终的标准在一些方面还有所区别，这使得它不是一个很好的日常参考。不过，它是一份极其有用的历史文献，尤其是它说明了某些特性在完成之前被标准所丢弃从而没能成为标准的前因后果，以及为何某些特性采用了标准中所定义的方式。

和Common Lisp之间有许多重要的区别。

一旦你习惯了Lisp的思维方式，就可能想了解它的更多内容。没人可以在不懂Smalltalk的情况下号称自己真正理解了面向对象，因此你可能需要从Adele Goldberg和David Robson的*Smalltalk-80: The Language*（Addison Wesley，1989年）开始，它是对Smalltalk核心的标准介绍。在这之后，Kent Beck的*Smalltalk Best Practice Patterns*（Prentice Hall，1997年）是一本面向Smalltalk程序员的完美教材，其中的许多内容都可以应用在任何面向对象的语言里。

另一方面，Bertrand Meyer的*Object-Oriented Software Construction*（Prentice Hall，1997年）给出了Eiffel发明人对静态语言思想的杰出解释，Eiffel是Simula和Algol的一个后裔，常常被人忽视。它含有很多值得思考的东西，即便是对于那些工作在诸如Common Lisp这类动态语言的程序员来说也是这样。特别是Meyer关于契约式设计（Design By Contract）的思想对于应该如何使用Common Lisp的状况系统也有很大参考价值。

尽管不是关于计算机的，但James Surowiecki的*The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies, and Nations*（Doubleday，2004年）一书却给出了下面这个问题的一个很好的答案："如果Lisp这么好的话，那为什么不是每个人都用它呢？"具体参见第53页开始的一节"Plank-Road Fever"。

最后，为了寻找一些乐趣，并且也是为了了解Lisp和Lisp程序员对黑客文化的影响，可以看看Eric S. Raymond所编译的*The New Hacker's Dictionary*第三版（MIT Press，1996年），该书基于Guy Steele早先所编辑的*The Hacker's Dictionary*（Harper & Row，1983年）。

但是不要让所有这些建议影响你的编程，真正学好一门语言的唯一方法是去实际使用它。如果你已经学到这里了，那么肯定已经准备好要这样做了。那么祝你玩得开心！