

TURING 图灵程序设计丛书

Practical Common Lisp

实用Common Lisp编程

[美] Peter Seibel 著
田春 译

人民邮电出版社
北京

图书在版编目 (C I P) 数据

实用Common Lisp编程 / (美) 塞贝尔 (Seibel, P.) 著 ; 田春译. — 北京 : 人民邮电出版社, 2011.10
(图灵程序设计丛书)
书名原文: Practical Common Lisp
ISBN 978-7-115-26374-2

I. ①实… II. ①塞… ②田… III. ①LISP语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2011)第191310号

内 容 提 要

这是一本不同寻常的 Common Lisp 入门书。本书首先从作者的学习经历及语言历史出发，随后用 21 个章节讲述了各种基础知识，主要包括：REPL 及 Common Lisp 的各种实现、S- 表达式、函数与变量、标准宏与自定义宏、数字与字符以及字符串、集合与向量、列表处理、文件与文件 I/O 处理、类、FORMAT 格式、符号与包，等等。而接下来的 9 个章节则翔实地介绍了几个有代表性的实例，其中包含如何构建垃圾过滤器、解析二进制文件、构建 ID3 解析器，以及如何编写一个完整的 MP3 Web 应用程序等内容。最后还对一些未介绍内容加以延伸。

本书内容适合 Common Lisp 初学者及对之感兴趣的相关人士。

图灵程序设计丛书 实用Common Lisp编程

-
- ◆ 著 [美] Peter Seibel
 - 译 田 春
 - 责任编辑 傅志红
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
 - 邮编 100061 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 北京艺辉印刷有限公司印刷
 - ◆ 开本：800×1000 1/16
 - 印张：27.25
 - 字数：678千字 2011年10月第1版
 - 印数：1-3 500册 2011年10月北京第1次印刷
 - 著作权合同登记号 图字：01-2010-2360号

ISBN 978-7-115-26374-2

定价：89.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

www.TopSage.com

对本书的赞誉

“Peter Seibel 的 *Practical Common Lisp* 名副其实：对于那些想要学习并开始将 Common Lisp 用于实际工作的人来说，这是一本极佳的入门参考书。这本书写得非常好并且读起来很有趣，至少对于那些认为学习新语言很有趣的人来说是这样的。

“Seibel 在书中并未花大量时间来抽象地讨论 Lisp 在编程语言界的地位，而是选择了正确的切入点，通过一系列复杂度递增的编程示例来引导读者。这一思路把重点放在了那些有经验的程序员们最常使用的 Common Lisp 特性上，而没有纠缠于那些即便是专家也要通过查询手册才知道的 Common Lisp 语言特性的角落。Seibel 这一示例驱动思路所产生的结果就是，让读者准确地感受到了 Common Lisp 在以最小的代价构建复杂而具革命性的软件系统过程中所爆发的强大威力。

“在 Common Lisp 领域已经有许多采用了更具抽象和比较性的编写思路来创作的好书，但一本能够针对实际开发者讲解实现方式及相关原因的好书却是很有价值的，无论是对当前的 Common Lisp 用户，还是其他潜在用户来说都是这样。”

——Scott E. Fahlman, Carnegie Mellon 大学计算机科学教授

“这本书要是在我当初学 Lisp 的时候出版该多好。这并不是说当时没有其他关于 Common Lisp 的好书，但它们都不像本书这样务实和与时俱进。况且我们还不要忘了 Peter 讲述了一些在其他 Lisp 著作中被完全忽略的内容，诸如路径名、状况和再启动等主题。”

“如果你初学 Lisp 并且想要选择一个正确的切入点，那么就不要犹豫，赶快买下本书吧。一旦你读过并学会了它，接下来就可以继续阅读 Graham、Norvig、Keene 和 Steele 等人的‘经典’著作了。”

——Edi Weitz, Common Lisp Cookbook 的维护者和 CL-PPCRE 正则表达式库的编写者

“有经验的程序员可以从示例中学到极其有用的知识，而 Seibel 这本示例丰富的人门教材中用 Lisp 来讲解真是件令人高兴的事。尤其令人赞叹的是，这本书中包含的如此多的示例涵盖了当今程序员日常可能用到的众多问题领域，诸如 Web 开发和流媒体技术。”

——Philip Greenspun, Software Engineering for Internet Applications 的作者,
MIT 电子工程和计算机科学学院

“这是一本涵盖了 Common Lisp 广泛内容的优秀书籍，其中通过演示一些读者可以运行和扩展的实际应用，阐述了 Common Lisp 许多独一无二的特性。这本书不仅说明了 Common Lisp 是什么，还说明了为什么每个程序员都应当熟悉 Lisp。”

——John Foderaro, Franz Inc. 资深科学家

“Maxima 项目经常得到那些想要学习 Common Lisp 的潜在贡献者的垂询。我很高兴最终有一本书让我可以毫无保留地推荐给他们。Peter Seibel 那简明直接的风格使读者能够快速地领略 Common Lisp 的威力。他包含在书中的许多示例重点关注当代的编程问题，充分说明了 Lisp 绝不仅仅是一门学院派编程语言。这本书是该领域的一本极受欢迎的著作。”

——James Amundson, Maxima 项目负责人

“我喜欢那些分散在书中描述‘真实’和有应用程序的实践性章节。我们需要这样一本书来告诉世界，使用 Lisp 可以轻松地将字符串和数字组装成树和图。”

——Christian Queinnec 教授, Universite Paris 6 (Pierre et Marie Curie)

“学习一门编程语言最重要的一部分就是学习它正确的编程风格。这很难教授，但通过阅读这本书，你可以轻松地获取到。仅是阅读那些实践性的示例就足以让我成为能胜任各种语言的高级程序员。”

——Peter Scott, Lisp 程序员

“它提供了这门语言的全新视角，并且后面章节里的那些例子在你作为程序员的日常工作中也是很有用的。”

——Frank Buss, Lisp 程序员和 Slashdot 贡献者 (www.slashdot.org)

“如果你对 Lisp 感兴趣是因为它跟 Python 或 Perl 有关系，并且想要通过实际动手而不只是观看学习它，那么这将是一本极佳的入门书。”

——Chris McAvoy, Chicago Python 用户组 (www.chipy.org)

“终于有一本为我们写的 Lisp 书了。如果你想要学习如何编写一个阶乘函数，那么这本书并不适合你。Seibel 的书是为专业程序员所写的，它更关注工程师和艺术家，而非科学家，并在解决易于理解的现实问题过程中优雅而精细地体现出了这门语言的威力。”

“阅读大多数章节的感受就好像正在编写一个程序，一开始只知道很少的内容，然后越来越多，就像在搭建一个最终可以站在上面的平台那样。当 Seibel 在构建一个测试框架的过程中顺势引入宏的时候，我对如此一个简单的例子就让我能真正‘领会’它们的含义而感到震惊。书中的叙事性内容极其有用，这样的技术书籍更出类拔萃。恭喜你！”

——Keith Irwin, Lisp 程序员



“在学习 Lisp 的过程中，人们不知道一个特定函数的用途时会去查询 CL HyperSpec，但我发现如果只是阅读 HyperSpec 通常很难‘领会’其含义。当遇到这类问题时，我每次都会翻阅这本书，它是目前在教授编程方式方面最具可读性的来源，而绝不仅仅是平铺直叙。”

——Philip Haddad, Lisp 程序员

“在急速发展的 IT 行业，专业人员需要最强大的工具。这就是 Common Lisp 这种最强大、最灵活和最稳定的编程语言正获得广泛关注的原因。这是一本令人期待已久的书，它将帮助你驾驭 Common Lisp，以应对当今各种复杂的现实问题。”

——Marc Battyani, CL-PDF、CL-TYPESETTING 和 mod_lisp 的开发者

“不要认为 Common Lisp 只能用于数据库、单元测试框架、垃圾过滤器、ID3 解析器、Web 编程、Shoutcast 服务器、HTML 生成解释器和 HTML 生成编译器等领域，这些只是碰巧在这本书中被实现的几件事。”

——Tobias C. Rittweiler, Lisp 程序员

“当我遇到 Peter 时，他正在写这本书。我自问：‘为什么已经有了许多很好的入门书籍，他还要写另一本关于 Common Lisp 的书？’一年以后，我看到了这本新书的初稿，也意识到我最初的想法错了。这本书不是‘另一本’书。作者更关注实践方面而非语言的技术细节。我最初是通过阅读一本入门书学习 Lisp 的，我觉得我理解了这门语言，但也有这样的感觉：‘那又怎样？’这就意味着我完全不知道如何使用它。相反，这本书在用最初的几章讲解了最基本的语言概念以后就转向了‘实践’章节。读者会在跟随开发这些‘实用’项目的过程中逐渐学会更多的语言知识，同时这些项目将会合并成具有相当规模的产品。读完本书，读者会感到他们已经是专业的 Common Lisp 程序员了，因为他们已经‘完成’了一个大项目。我认为，Lisp 是唯一一门允许采用这种实践方式来介绍的语言。Peter 充分利用了这个语言特性，勾勒出对 Common Lisp 的有趣介绍。”

——Taiichi Yuasa 教授，京都大学计算机科学与通信学院

“本书展示了 Lisp 的威力，不仅是在传统领域，例如仅使用短短的 26 行代码就开发出一个完整的单元测试框架；而且还表现在一些全新领域上，诸如解析二进制 MP3 文件、构建浏览歌曲集的 Web 应用以及在 Web 上传输音频流。许多读者会很惊讶：Lisp 具有 Python 等脚本语言的简洁性、C++ 的高效性，以及设计自己语言扩展时无与伦比的灵活性。”

——Peter Norvig, Google 公司搜索质量组负责人, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* 的作者

译者序

很荣幸，我被授权翻译 *Practical Common Lisp* 一书。本书是自 1994 年 Common Lisp 语言标准化以来，国内出版的第一本 Common Lisp 的中文教材。

Lisp 语言家族最早诞生于 1959 年，它是人类历史上第二个高级程序设计语言（第一个是 Fortran）。那一年，人工智能（AI）专家 John McCarthy 发表了具有重大历史意义的第一篇 LISP 论文 “Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I”，其中介绍了一种运行在古老的 IBM 704 计算机上的列表处理语言 LISP (LIST Processing, 列表处理)，借助它可以轻松描述当时人工智能领域用到的各种算法。从此，Lisp 语言在包括 AI 领域在内的所有主流计算机分支上，都获得了长足的发展。Lisp 平台不但在 IBM PC 出现之前的几乎所有计算机硬件体系上均有移植，甚至在 20 世纪 80 年代还出现过专门用来运行 Lisp 程序的硬件——Lisp 机。1994 年，ANSI 标准化的 Common Lisp 语言将之前历史上的所有现存 Lisp 厂商的各种语言和平台特性做了一次伟大的总结，从此语言核心不再变化，不但标准化以前的历史遗留代码只通过少量修改就可以兼容现代 Lisp 平台，而且标准化以后写出的所有新代码也都几乎不经任何调整就可能运行在任何一种 Common Lisp 平台上，无论是带有原生或是字节码编译器的，还是间接转译成 C 语言的，或是运行在 JVM 上的。目前至少有 13 种不同的 Common Lisp 语言平台可以运行在现代计算机上，其中 10 种还在广泛使用中，远超过它们所在的操作系统上 C 和其他语言编译器的数量。可以说，Lisp 语言家族长达 50 年的发展史就是整个计算机发展史的缩影。

我从 2003 年大学三年级时开始学习 Common Lisp 语言，至今已有八个年头。当时学习它的动机基本上是出于对人工智能（传统的逻辑和推理、知识表示等方向）的个人兴趣。不过随后很快就发现，Common Lisp 是一门通用的编程语言，如果不考虑其历史渊源而只从语言本身的特性来观察的话，可以说它跟人工智能毫无关系。在 *Practical Common Lisp* 一书中，作者 Peter Seibel 也谈到这个问题。当今有太多的人对 Lisp 语言存在类似的误解，包括相当多学过早期 Lisp 语言的人还停留在列表（List）是 Lisp 语言的唯一复合数据类型的认识上。如果读者从头到尾学完了这本书，就会发现 Common Lisp 是一门特性丰富的大型编程语言，不但提供了现代编程语言普遍支持的各种数据类型（包括各种数值类型、字符串、数组、结构体和哈希表在内），还支持几乎所有的编程范式（面向过程的、函数式的以及面向对象的），尤其带有一套特性丰富且思想独到的面向对象编程接口 CLOS (Common Lisp Object System) 和 OO 扩展接口 MOP (Meta-Object Protocol)。如果要用一句话来描述 Common Lisp 中的 OO 与 C++/Java/SmallTalk 等语言的 OO 有

何不同，那就是 Common Lisp 对象系统完全不是基于消息传递的，而是基于广义函数的。有兴趣的读者应当仔细阅读本书的第 16 章和第 17 章，其中介绍了 CLOS 的一些入门内容。

不过 Lisp 语言最吸引人的地方还在于其与众不同的程序运行方式。从 C 语言一路学过来的人往往把一门语言的语法及其标准函数库视为语言的全部，因为一旦程序写好，编译器就会将整个代码编译成一个可执行程序或者被其他可执行程序使用的库。接下来语言本身是什么就不重要了，重要的是程序员写出了什么功能，甚至连编译器本身是什么都不重要，因为它只是一个黑箱，除了简单的优化开关之外几乎无法调整其行为。各种 Lisp 语言则采用完全不同的方式来运行 Lisp 程序：Lisp 平台本身是一个交互式的环境，它在很大程度上就是用其本身写成的。用户的 Lisp 代码以编译或解释的形式加载到 Lisp 环境中，然后跟 Lisp 语言或平台本身的代码直接融合在一起。换句话说，每一个 Lisp 程序都是对 Lisp 语言本身的某种形式的扩展。然后通过一个启动函数，整个程序得以运行。听到这里，读者似乎看到了 Python 或者 Ruby 的影子，但 Lisp 环境还有更绝的地方：几乎所有 Lisp 平台都允许用户将加载了用户代码的整个环境从内存中导出 (dump) 为一个磁盘文件。通过直接加载这个文件而不是默认的那个只含有 Lisp 本身的文件，可以迅速地重建导出前的 Lisp 环境，从而达到增量开发或者哪怕是快速加载已有 Lisp 程序的目的。最后，和其他语言很不同的一点是，Lisp 语言规范（至少 Common Lisp 是这样的）不但包括了如何定义某个程序组成部分（指的是变量、函数和类这些东西）的能力，还定义了从 Lisp 环境中清除任何程序组成部分以及就地修改它们的能力，并在语义和功能上确保了这些操作不会破坏运行中的 Lisp 代码。这导致了 Lisp 语言的另一个重要应用：通过加载补丁，Lisp 系统可以在运行中被任意修改，这对 24×7 的服务器端程序的平滑升级尤为有利。顺便说一句，Lisp 也是最早引入垃圾收集 (GC) 机制的编程语言，Lisp 环境中的任何对象，一旦失去了来自其他对象的引用，就会在某个时刻被 GC 系统从内存中清除掉。

读者可能已经注意到了我在不停地混用 Lisp 和 Common Lisp 两个概念。这有两层含义：首先，存在 Common Lisp 之外的 Lisp 语言，更准确地说是 Lisp 方言 (dialect)，至少包括了 Emacs Lisp、AutoLISP、Scheme、Racket (前身是 PLT Scheme) 和 Clojure，其中最后一个是在高速发展中的新兴 Lisp 方言；其次，所有 Lisp 家族的语言都有很多共性，除了上面描述中带有 Lisp 而非 Common Lisp 字样的部分以外，还有最大的也是初学者最容易看到的一点，那就是所有 Lisp 方言都使用前缀表达式和用小括号表示的列表，例如 $1+1$ 在 Lisp 中将写成 $(+ 1 1)$ 。很多初学者一开始都不适应前缀表达式，但我认为前缀表达式是有很多优点的：首先，它彻底消除了运算符结合性问题，令表达式毫无歧义可言；其次，它让语言处理器更加简单高效，避免了语法分析的困难。当然，一旦习惯了也就感觉没什么了。

学习本书对更好地使用其他 Lisp 方言无疑是大有帮助的。在翻阅书店里关于 AutoLISP (AutoCAD 计算机辅助设计软件的扩展语言) 的各种书籍时，我经常痛心疾首地发现这些图书的作者虽然精通 AutoCAD 所提供的 Lisp 编程接口，但写出的 AutoLISP 代码要么极为难看，要么缺乏效率、滥用内存。AutoLISP 在语法上跟 Common Lisp 非常接近，本书的大部分内容都适用于 AutoLISP。因此我强烈推荐所有 AutoLISP 程序员阅读本书以加强自身的 Lisp 素养。同样的问题对于 Emacs Lisp (GNU Emacs 文本编辑器的扩展语言) 来说也是一样的。Scheme 系的 Lisp 方

言区别相对大一些，如果连基本的变量和函数定义都在形式上完全不同的话（当然，思想上是没什么本质区别的），我恐怕初学者从本书中学得 Scheme 编程思想的机会不大，这种情况下还是推荐《计算机程序的构造和解释》、*Lisp in Small Pieces* 和 *The Little Schemer* 等书籍比较好。

本书可以作为其他 Common Lisp 语言教材的学习基础。在本书的最后一章里，作者给出了很多后续的教材，在此就不一一重复了。需要特别指出的是，另一本著名的 Common Lisp 教材 *On Lisp*（作者 Paul Graham，也就是《黑客与画家》一书的作者）多年前已经被我和我的几位朋友共同翻译成中文版，细心的读者可以从网上轻易地找到它。*On Lisp* 主要介绍 Common Lisp 的宏编程，这是 Common Lisp 区别于其他语言甚至其他 Lisp 方言的最重要特性。我相信一旦读者掌握了本书中关于宏的章节以后就可以阅读 *On Lisp* 中的进阶内容，从而将自身对编程语言的认识上升到一个新的高度，不过更加符合实用原则的思路还是先把本书读完。

Common Lisp 绝不是一门过时的编程语言，整个 Common Lisp 社区一直都在高速发展之中，近几年的发展尤为迅速。在我学习 Common Lisp 的这些年里，我亲眼目睹了几个 Common Lisp 平台从无到有（ECL、ABCL）或者发展壮大（SBCL、Clozure CL）的过程。经典平台（CMUCL、MCL）也得到了良好的维护并始终跟进操作系统的自然发展。随着计算机硬件的高速发展，即便相对保守的 Common Lisp 商业平台也开始或即将开始支持对称多处理器（SMP），其中 LispWorks 和 Scieneer CL 都以 SMP 支持作为主要卖点。第三方软件包长足发展，虽然尚未达到 Perl 社区 CPAN 的水平，但常用的工具包一应俱全，其中不乏高质量的大型项目。近年来最新的成果 Quicklisp 包管理平台，更是将 Common Lisp 第三方软件包的安装过程提升到了前所未有的便捷程度。免费平台越来越好，商业平台依然昂贵，开源工具蓬勃发展，所有这些都暗示着 Common Lisp 语言还保持着旺盛的生命力，唯一的问题是如何让更多的国内计算机领域爱好者了解它。这就是我翻译本书的目的所在。

过去 8 年里，我一直活跃在国内和国际 Common Lisp 社区的前沿。我在大学本科的最后两年学完了 Common Lisp 语言语法的主要部分，读完了包括本书在内的几本最经典的 Lisp 书籍，并已经能够在当时最常见的 CMUCL 平台（CMU Common Lisp）上编写一些简单的程序。后来在网易工作的 5 年里，我在工作之余从头研究了一遍 Lisp 语言的发展史，亲身体会了包括 Lisp 机在内的十几种不同的 Common Lisp 平台或实现，并自费购买了价值数千美元的商业开发环境 LispWorks，拥用三种主流操作系统上的 License。在网易从事 Linux 系统管理工作期间，我用 Common Lisp 从头实现了一万行源代码规模的 SNMP 简单网络管理协议工具包，它可以为任何服务器端 Common Lisp 程序添加通过 SNMP 协议进行远程监控的能力，也可以作为基于 Common Lisp 的网络监控系统的基础。我还在过去 3 年里参与维护了 Common Lisp 社区两个最重要的可移植网络库之一：usocket，并由于 SNMP 库的需要将其从原本只支持 TCP 扩展到了同时支持 UDP，其中对于 LispWorks 的 UDP 支持代码是完全从头写的，因为官方并不支持。2009 年，我向国际 Lisp 会议的投稿被接受，并作为会议论文集的一部分出版。我是长期担任水木社区函数型编程板块的板主之一，专门负责 Lisp 方向的讨论和技术分享。2011 年 7 月，我离开网易以后开始全职从事商业 Lisp 软件相关的开发工作。可能我还不是一个很好的译者，但作为一个经验丰富的 Common Lisp 程序员，我相信自己翻译这本书是合适的。

计算机领域每天都在高速发展，新语言和新技术的产生速度早已超过了一般人的学习速度。对于一个计算机领域的从业人员或爱好者来说，学习通常是为了更好地应用，把所有时间都用来学习而无暇具体应用也是本末倒置。在这种情况下，有选择地学习最有用、最不易变质的知识，以及甄别各种计算机知识的重要程度和相互关系的能力就显得非常重要的了。从计算机语言的发展历史来说，如果一门语言可以存活 50 年，那么它的内在生命力很可能保证其继续长期存活下去，一个人用这门语言写下的代码也将比其他语言的代码更有可能长久地造福后人。

总之，希望这本书能将读者顺利带入 Lisp 领域。学习一门新的语言总是要花些成本的，但我想说，和其他任何语言相比，花在理解 Lisp 上的时间和精力将绝对是物超所值的，即便相当多的读者可能没有机会在短期内将 Lisp 用于他们的日常工作。之所以这样说是有原因的：C 和 Lisp 是编程语言的两个极端，大多数人已经熟悉了 C 的那一端，但如果他们还熟悉另一端的话，那么迅速理解几乎所有其他的编程语言将不再是问题。



致 读 者

亲爱的读者：

Practical Common Lisp，这难道不矛盾吗？你可能和大多数程序员一样，知道一点儿 Lisp：要么是来自大学里的计算机科学课程，要么是学习了足够多的 Elisp 以定制 Emacs。或者可能你只是因为有人正喋喋不休地谈论 Lisp，而其认为这是一门历史上最伟大的语言。但是，你可能从未想过会在同一本书的书名里看到 Practical 和 Lisp。

但你确实已经在读这样一本书了，你一定还想知道更多。也许你相信学习 Lisp 将使你成为一名能够胜任所有语言的高级程序员。或者可能你只是想要知道那些 Lisp 狂热者们总是在喊些什么。或者可能你已经学了一些 Lisp，但还没有足够的能力去用它编写感兴趣的软件。

如果确实出现上述情形中的任何一种，那么本书就很适合你。Common Lisp 是一种 ANSI 标准化的工业级别的 Lisp 方言，通过使用它，我将向你展示如何编写那些远远超越了愚蠢的学术训练或简单编辑器定制的实际的软件。并且我将说明即便在其许多特性已被其他语言所采纳以后，Lisp 仍然有其独到之处。

与许多 Lisp 书籍所不同的是，本书不会只触及一些 Lisp 的伟大特性而让你自己思考如何使用它们。我讨论了你在编写实际程序中用到的所有语言特性，并用了多于三分之一的篇幅来开发具有一定复杂度的软件——基于统计的垃圾过滤器、用来解析二进制文件的库，以及一个带有完整在线 MP3 数据库和 Web 接口的通过网络流式传输 MP3 的服务器。

现在就翻开它，看看这门人类发明的最伟大的语言是多么的实用吧。

此致

Peter Seibel

致 谢

如果不是因为一些愉快的巧合，本书不会写出来，至少作者不是我。因此，我必须首先感谢 Franz 的 Steven Haflich。我们在 Bay Area Lispniks 见面会上相遇时，他邀请我和 Franz 的一些销售人员共进午餐，席间我们都认为需要有本 Lisp 新书。其次我必须感谢 Steve Sears，午餐中的一名销售人员，他随后将我引荐给 Franz 的总裁 Fritz Kunze。Fritz 曾经提到他正想请人来写一本 Lisp 书。当然，也非常感谢 Fritz 说服 Apress 出版社出版一本由我来写的新 Lisp 书。Apress 出版社在整个过程中给予了我鼓励和帮助。也要感谢 Franz 的 Sheng-Chuang Wu，提供了诸多帮助。

我在撰写本书时最不可或缺的资源是 comp.lang.lisp 新闻组。comp.lang.lisp 的忠实成员们不厌其烦地回答了有关 Lisp 及其历史的各种问题。我也经常翻阅该新闻组的 Google 存档，它是技术资料的宝库。因此，感谢 Google 提供了这一服务，也感谢所有 comp.lang.lisp 参与者们一直以来提供各种内容。特别地，我要感谢两位长期的 comp.lang.lisp 贡献者——Barry Margolin 和 Kent Pitman。Margolin 在我阅读该组资料时一直在提供各种琐碎的关于 Lisp 历史和他个人经验的资料；而 Pitman 除了作为一位语言标准的首席技术编辑和 Common Lisp HyperSpec 的开发者以外，还在 comp.lang.lisp 上撰写了成千上万字的帖子以阐述语言的多种方面及其来源。

在撰写本书时，其他不可或缺的资源包括用于 PDF 生成和排版的 Common Lisp 库以及由 Marc Battyani 编写的 CL-PDF 和 CL-TYPESETTING。我使用 CL-TYPESETTING 生成用于我个人编辑工作所用的 PDF 文件，并将 CL-PDF 作为我用来生成本书中线条图的 Common Lisp 程序的基础。

我还想感谢许多在网上阅读了本书部分草稿，并通过电子邮件指出笔误、提出问题或简单给出建议的人们。尽管无法提及所有人的名字，但对于一些提供了有价值的反馈的人要在这里特别指出：J. P. Massar（Bay Area Lispnik 成员，他在几次比萨午餐上极大地激发了我的灵感）、Gareth McCaughan、Chris Riesbeck、Bulent Murtezaoglu、Emre Sevinc、Chris Perkins 以及 Tayssir John Gabbour。我的几个非 Lisp 相关的朋友也参与审阅了某些章节，感谢 Marc Hedlund、Jolly Chen、Steve Harris、Sam Pullara、Sriram Srinivasan 和 William Grosso 提供反馈。另外要感谢 Scott Whitten 允许我使用图 26-1 中的那张照片。

我的技术审稿人 Steven Haflich、Mikel Evins、Barry Margolin 以及我的文字编辑 Kim Wimpsett，他们以数不清的方式提升了本书品质。当然，如果书中还有任何错误，那都是我的责任。同时也感谢 Apress 出版社其他参与本书出版工作的人们。

最后，也是最重要的，我要感谢我的家庭。感谢父母所做的一切，以及总是坚定地相信我可以完成本书的我的妻子 Lily。

排版约定

诸如 `xxx` 这样的内嵌文本是代码，它们通常是函数、变量和类等元素的名字，这些名字要么是我已经引入的，要么是我将要引入的。由语言标准所定义的名字会写成这样：`DEFUN`。更大块的示例代码的样式将会如下所示：

```
(defun foo (x y z)
  (+ x y z))
```

由于 Common Lisp 的语法以其规范性和简洁性著称，因此我采用简单的模版来描述各种 Lisp 形式^①的语法。例如，下面描述了 `DEFUN` 的语法，它是标准的函数定义宏：

```
(defun name (parameter*)
  [documentation-string]
  body-form*)
```

这些模板中以斜体书写的名称需要填入我将在正文中描述的具体名称或 Lisp 形式。斜体名字后跟的星号 (*) 则表示该名称出现了零次或更多次。而位于方括号 ([]) 内的名称代表可选的元素。偶尔也会出现以竖线 (|) 分隔的替代内容。模版中的所有其他内容（通常只是一些名字和括号）都是一些将会出现在 Lisp 形式中的字面文本。

最后，由于跟 Common Lisp 之间的许多交互都发生在交互性的“读-求值-打印”循环（即 REPL）中，所以将经常如下显示在 REPL 中对 Lisp 形式求值的结果：

```
CL-USER> (+ 1 2)
3
```

其中的 CL-USER> 是 Lisp 提示符，其后总是跟着需要求值的表达式，在本例中是 `(+ 1 2)`。求值的结果和生成的其他输出都会显示在接下来的几行里。有时，我也会在表达式后使用 → 来表示求值的结果，就像下面这样：

```
(+ 1 2) • 3
```

偶尔还会使用等价符号 (=) 来说明两个表达式是等价的，就像这样：

```
(+ 1 2 3) ≡ (+ (+ 1 2) 3)
```

^① Lisp 形式 (Lisp form) 是 Lisp 特有的一种语法构造，本书在不致引起歧义的上下文里，也会将其简称为“形式”。

——译者注

目 录

第 1 章 绪言：为什么是 Lisp	1
1.1 为什么是 Lisp	2
1.2 Lisp 的诞生	4
1.3 本书面向的读者	6
第 2 章 周而复始：REPL 简介	8
2.1 选择一个 Lisp 实现	8
2.2 安装和运行 Lisp in a Box	10
2.3 放开思想：交互式编程	10
2.4 体验 REPL	11
2.5 Lisp 风格的“Hello, World”	12
2.6 保存工作成果	13
第 3 章 实践：简单的数据库	17
3.1 CD 和记录	17
3.2 录入 CD	18
3.3 查看数据库的内容	19
3.4 改进用户交互	21
3.5 保存和加载数据库	23
3.6 查询数据库	24
3.7 更新已有的记录——WHERE 再战江湖	28
3.8 消除重复，获益良多	29
3.9 总结	33
第 4 章 语法和语义	34
4.1 括号里都可以有什么	34
4.2 打开黑箱	34
4.3 S-表达式	36
4.4 作为 Lisp 形式的 S-表达式	38
4.5 函数调用	39
4.6 特殊操作符	39
第 5 章 函数	46
4.7 宏	41
4.8 真、假和等价	42
4.9 格式化 Lisp 代码	43
第 6 章 变量	57
6.1 变量的基础知识	57
6.2 词法变量和闭包	60
6.3 动态变量	61
6.4 常量	65
6.5 赋值	65
6.6 广义赋值	66
6.7 其他修改位置的方式	67
第 7 章 宏：标准控制构造	69
7.1 WHEN 和 UNLESS	70
7.2 COND	71
7.3 AND、OR 和 NOT	72
7.4 循环	72
7.5 DOLIST 和 DOTIMES	73
7.6 DO	74
7.7 强大的 LOOP	76

第 8 章 如何自定义宏	78	11.9 序列谓词.....	119
8.1 Mac 的故事：只是一个故事	78	11.10 序列映射函数	120
8.2 宏展开期和运行期	79	11.11 哈希表	120
8.3 DEFMACRO	80	11.12 哈希表迭代	122
8.4 示例宏：do-primes	81		
8.5 宏形参	82		
8.6 生成展开式	83		
8.7 堵住漏洞	84		
8.8 用于编写宏的宏	88		
8.9 超越简单宏	90		
第 9 章 实践：建立单元测试框架	91		
9.1 两个最初的尝试	91		
9.2 重构	92		
9.3 修复返回值	94		
9.4 更好的结果输出	95		
9.5 抽象诞生	97		
9.6 测试层次体系	97		
9.7 总结	99		
第 10 章 数字、字符和字符串	101		
10.1 数字	101		
10.2 字面数值	102		
10.3 初等数学	104		
10.4 数值比较	106		
10.5 高等数学	107		
10.6 字符	107		
10.7 字符比较	107		
10.8 字符串	108		
10.9 字符串比较	109		
第 11 章 集合	111		
11.1 向量	111		
11.2 向量的子类型	113		
11.3 作为序列的向量	114		
11.4 序列迭代函数	114		
11.5 高阶函数变体	116		
11.6 整个序列上的操作	117		
11.7 排序与合并	118		
11.8 子序列操作	118		
第 12 章 LISP 名字的由来：列表 处理	123		
12.1 “没有列表”	123		
12.2 函数式编程和列表	126		
12.3 “破坏性” 操作	127		
12.4 组合回收性函数和共享结构	129		
12.5 列表处理函数	131		
12.6 映射	132		
12.7 其他结构	133		
第 13 章 超越列表：点对单元的其他 用法	134		
13.1 树	134		
13.2 集合	136		
13.3 查询表：alist 和 plist	137		
13.4 DESTRUCTURING-BIND	141		
第 14 章 文件和文件 I/O	143		
14.1 读取文件数据	143		
14.2 读取二进制数据	145		
14.3 批量读取	145		
14.4 文件输出	145		
14.5 关闭文件	146		
14.6 文件名	147		
14.7 路径名如何表示文件名	149		
14.8 构造新路径名	150		
14.9 目录名的两种表示方法	152		
14.10 与文件系统交互	153		
14.11 其他 I/O 类型	154		
第 15 章 实践：可移植路径名库	157		
15.1 API	157		
15.2 *FEATURES* 和读取期条件化	157		
15.3 列目录	159		
15.4 测试文件的存在	162		
15.5 遍历目录树	164		



第 16 章 重新审视面向对象：广义函数	165	19.3 状况处理器	205
16.1 广义函数和类	166	19.4 再启动	207
16.2 广义函数和方法	167	19.5 提供多个再启动	210
16.3 DEFGENERIC	168	19.6 状况的其他用法	211
16.4 DEFMETHOD	169		
16.5 方法组合	171		
16.6 标准方法组合	172		
16.7 其他方法组合	173		
16.8 多重方法	174		
16.9 未完待续	176		
第 17 章 重新审视面向对象：类	177		
17.1 DEFCLASS	177		
17.2 槽描述符	178		
17.3 对象初始化	179		
17.4 访问函数	182		
17.5 WITH-SLOTS 和 WITH-ACCESSORS	185		
17.6 分配在类上的槽	186		
17.7 槽和继承	187		
17.8 多重继承	188		
17.9 好的面向对象设计	190		
第 18 章 一些 FORMAT 秘诀	191		
18.1 FORMAT 函数	192		
18.2 FORMAT 指令	193		
18.3 基本格式化	194		
18.4 字符和整数指令	194		
18.5 浮点指令	196		
18.6 英语指令	197		
18.7 条件格式化	198		
18.8 迭代	199		
18.9 跳，跳，跳	201		
18.10 还有更多	202		
第 19 章 超越异常处理：状况和再启动	203		
19.1 Lisp 的处理方式	204		
19.2 状况	205		
第 20 章 特殊操作符	213		
20.1 控制求值	213		
20.2 维护词法环境	213		
20.3 局部控制流	216		
20.4 从栈上回退	219		
20.5 多值	223		
20.6 EVAL-WHEN	224		
20.7 其他特殊操作符	227		
第 21 章 编写大型程序：包和符号	228		
21.1 读取器是如何使用包的	228		
21.2 包和符号相关的术语	230		
21.3 三个标准包	230		
21.4 定义你自己的包	232		
21.5 打包可重用的库	234		
21.6 导入单独的名字	235		
21.7 打包技巧	236		
21.8 包的各种疑难杂症	237		
第 22 章 高阶 LOOP	240		
22.1 LOOP 的组成部分	240		
22.2 迭代控制	241		
22.3 计型数循环	241		
22.4 循环集合和包	242		
22.5 等价-然后迭代	243		
22.6 局部变量	244		
22.7 解构变量	245		
22.8 值汇聚	245		
22.9 无条件执行	247		
22.10 条件执行	247		
22.11 设置和拆除	248		
22.12 终止测试	250		
22.13 小结	251		
第 23 章 实践：垃圾邮件过滤器	252		
23.1 垃圾邮件过滤器的核心	252		

23.2 训练过滤器	255	第 26 章 实践：用 AllegroServe 进行 Web 编程	315
23.3 按单词来统计	257	26.1 30 秒介绍服务器端 Web 编程	315
23.4 合并概率	259	26.2 AllegroServe	317
23.5 反向卡方分布函数	261	26.3 用 AllegroServe 生成动态内容	320
23.6 训练过滤器	262	26.4 生成 HTML	321
23.7 测试过滤器	263	26.5 HTML 宏	324
23.8 一组工具函数	265	26.6 查询参数	325
23.9 分析结果	266	26.7 cookie	327
23.10 接下来的工作	268	26.8 小型应用框架	329
第 24 章 实践：解析二进制文件	269	26.9 上述框架的实现	330
24.1 二进制文件	269	第 27 章 实践：MP3 数据库	334
24.2 二进制格式基础	270	27.1 数据库	334
24.3 二进制文件中的字符串	271	27.2 定义模式	336
24.4 复合结构	273	27.3 插入值	338
24.5 设计宏	274	27.4 查询数据库	340
24.6 把梦想变成现实	275	27.5 匹配函数	342
24.7 读取二进制对象	277	27.6 获取结果	344
24.8 写二进制对象	279	27.7 其他数据库操作	346
24.9 添加继承和标记的结构	280	第 28 章 实践：Shoutcast 服务器	348
24.10 跟踪继承的槽	281	28.1 Shoutcast 协议	348
24.11 带有标记的结构	284	28.2 歌曲源	349
24.12 基本二进制类型	285	28.3 实现 Shoutcast	351
24.13 当前对象栈	288	第 29 章 实践：MP3 浏览器	357
第 25 章 实践：ID3 解析器	290	29.1 播放列表	357
25.1 ID3v2 标签的结构	291	29.2 作为歌曲源的播放列表	359
25.2 定义包	292	29.3 操作播放列表	362
25.3 整数类型	292	29.4 查询参数类型	365
25.4 字符串类型	294	29.5 样板 HTML	367
25.5 ID3 标签头	297	29.6 浏览页	368
25.6 ID3 帧	298	29.7 播放列表	371
25.7 检测标签补白	300	29.8 查找播放列表	373
25.8 支持 ID3 的多个版本	301	29.9 运行应用程序	374
25.9 版本化的帧基础类	303	第 30 章 实践：HTML 生成库，解释器部分	375
25.10 版本化的具体帧类	304	30.1 设计一个领域相关语言	375
25.11 你实际需要哪些帧	305	30.2 FOO 语言	376
25.12 文本信息帧	307		
25.13 评论帧	309		
25.14 从 ID3 标签中解出信息	310		

30.3 字符转义	379	31.3 FOO 宏	399
30.4 缩进打印器	380	31.4 公共 API	401
30.5 HTML 处理器接口	381	31.5 结束语	403
30.6 美化打印器后台	382	第 32 章 结论：下一步是什么 404	
30.7 基本求值规则	385	32.1 查找 Lisp 库	404
30.8 下一步是什么	389	32.2 与其他语言接口	406
第 31 章 实践：HTML 生成库，编译器 部分	390	32.3 让它工作，让它正确，让它更快	406
31.1 编译器	390	32.4 交付应用程序	413
31.2 FOO 特殊操作符	395	32.5 何去何从	415

绪言：为什么是Lisp



如果你认为编程最大的乐趣在于，可以用简明扼要的代码来清晰表达你的意图，完成许多事，那么使用Common Lisp编程可以说是用计算机所能做到的最有趣的事了。比起相当多的其他计算机语言，它可以让你更快地完成更多工作。

这是个大胆的断言。可我要怎样证明呢？当然不是用本章的只言片语了，而是这一整本书。你必须先学习一些Lisp知识才能体会到这一点。不过眼下，让我们先介绍一些轶事，即我本人迈向Lisp编程之路的经历。然后在下一节里，我将说明你可以通过学习Common Lisp得到些什么收获。

我是极少数的所谓第二代Lisp黑客之一。我父亲的计算机生涯开始于用汇编语言给他的机器编写操作系统，从而为他的物理学博士论文搜集资料。在成功帮助了几家物理实验室用上计算机系统之后，从20世纪80年代起，他彻底抛弃了物理学，转而为一家大型制药公司工作。当时那家公司里有个软件项目，正在开发用于模拟化工生产过程的软件，比如说，增大容器的尺寸将会怎样影响其年产量。原先的团队使用FORTRAN进行开发，已经耗费了该项目的一半预算和几乎全部的时间，却没能交付任何成果。那是在20世纪80年代，人工智能（AI）蓬勃发展，Lisp正在流行。于是，我父亲（当时还不是Lisp程序员）来到卡内基－梅隆大学（CMU），向那些正在开发后来被称为Common Lisp语言的人们咨询：如果用Lisp来开发这个项目是否合适？

CMU的人向我父亲演示了一些他们正在做的东西，于是他被说服了。然后他又说服老板让他的团队接手那个失败的项目并用Lisp重新开发。一年以后，仅仅使用了原先剩余的预算，他的团队就交付了一个可用的应用程序，而且还带有已经被原先团队所放弃的一些功能。我父亲将其团队成功的原因归结为他们使用了Lisp。

当然，仅此一例还不足以说明问题，而且也有可能我父亲对其成功原因认识有误，或者也许Lisp仅仅在那个年代才可以跟其他语言相媲美。如今我们有了大量精巧的新语言，它们中的许多都吸收了Lisp的特性。是否真的可以说，Lisp在现今也具有跟我父亲那个年代一样的优势吗？请继续往下读。

尽管父亲作了大量努力，但我在高中前没有学过一点儿Lisp。在大学时代我也没怎么用任何语言来写程序，是后来出现的Web让我回到了计算机的怀抱。我首先使用的是Perl，先是为*Mother Jones*杂志的网站构建在线论坛，待经验丰富以后转向Web商店Organic Online，在那里我为当时的大型Web站点工作，例如耐克公司在1996年奥运会期间上线的站点。后来我转向Java，并成为

WebLogic（现在是BEA的一部分）的早期开发者。离开WebLogic以后，我又作为另一个团队的首席程序员开始用Java编写事务消息系统。在此期间，对编程语言的广泛兴趣使我充分研究了诸如C、C++和Python这些主流语言，以及Smalltalk、Eiffel和Beta这些小众语言。

所以我对Perl和Java两种语言了如指掌，同时还熟悉其他不下六种语言。不过最终我还是意识到，我对编程语言的兴趣其实来源于记忆之中的我父亲的Lisp经历——语言之间真的有天壤之别，尽管所有的编程语言在形式上都是图灵等价的，但一些语言真的会比其他语言更快更好地完成某些事情，并且在使用的过程中还能给你带来更多的乐趣。不过可笑的是，我从没有在Lisp上花太多时间。于是我开始利用业余时间研究Lisp。无论我做什么，最令我兴奋的始终是可以很快地将思路变成可用的代码。

举个例子，在一次假期里，我差不多在Lisp上花了一个星期的时间，目标是实现一个基于遗传算法、用来下围棋的程序系统——我以前做Java程序员的时候已经写过了。虽然我那时的Common Lisp知识极其有限，还要不时地查询基本函数的用法，但我还是能感觉到比用Java重写同样的程序来得顺手——尽管自编写该程序的最初版本以来，我又多了几年的Java经验。

类似的经历还产生了我将在第24章里讨论的一个库。我以前在WebLogic的时候曾经用Java写了一个库，用来解析Java的类文件。虽然可以正常使用，但是代码有点乱，并且难以修改或扩展。几年来我曾多次重写这个库，并期望凭借日益提高的Java技巧可以找到某种方式，来消除其中大量的重复代码，可惜从未成功。但当我改用Common Lisp做这件事时，我只花了两天时间，最后不但得到了一个Java类文件的解析器，而且还产生一个可用于解析任何二进制文件的通用库。第24章将讨论这个库的工作原理，第25章会用它写一个MP3文件的内嵌ID3标签的解析器。

1.1 为什么是 Lisp

很难用绪言这一章的寥寥数页来说清楚为何一门语言的用户会喜欢这门语言，更难的是找出一个理由说服你花时间来学习某种语言。现身说法只能到此为止了。也许我喜欢Lisp是因为它刚好符合我的思维方式，甚至可能是遗传因素，因为我父亲也喜欢它。因此，在你开始学习Lisp之前，想要知道能得到什么回报也是合理的。

对某些语言来说，回报是相对明显的。举个例子，如果打算编写Unix上的底层代码，那你应该去学C。或者如果打算编写特定的跨平台应用程序，那你应该去学Java。而相当数量的公司仍在大量使用C++，如果你打算在这些公司里找到工作，那就应该去学C++。

尽管如此，对于多数语言来说，回报往往不是那么容易界定的，这里面存在包括个人感情在内的主观因素。Perl拥趸们经常说Perl“让简单的事情简单，让复杂的事情可行”，并且沉迷于“做事情永远都有不止一种方法”^①这一Perl格言所推崇的事实。另一方面，Python爱好者们认为Python是简洁的，并且Python代码之所以易于理解，是因为正如他们的格言所说：“做事情只有一种方法。”

那么Common Lisp呢？没有迹象表明采用Common Lisp可以立即得到诸如C、Java和C++那样显而易见的好处（当然了，除非刚好拥有一台Lisp机）。使用Lisp所获得的好处很大程度上取决

^① Perl作为“因特网的传送带/血管”也同样值得学习。

于使用经验。本书的其余部分将向读者展示Common Lisp的特性以及如何使用它们，让你自己去感受它。眼下，我只想让你先对Lisp哲学有个大致的感受。

Common Lisp中最接近格言的是一句类似禅语的描述：“可编程的编程语言。”虽然隐晦了一些，但这句话却道出了Common Lisp至今仍然雄踞其他语言之上的最大优势。Common Lisp比其他任何语言都更加遵循一种哲学——凡利于语言设计者的也利于语言使用者。这就意味着，当使用Common Lisp编程的时候，你永远不会遇到这种情况：语言里刚好缺乏某些可能令程序更容易编写的特性，因为正如你将在本书中看到的，你可以为语言添加任何想要的特性。

因此，Common Lisp程序倾向于把关于程序如何工作的想法更清楚地映射到实际所写的代码上。想法永远不会被过于紧凑的代码和不断重复的模式搞得含糊不清。这将使代码更加易于维护，因为不必在每次修改之前都先复查大量的相关代码。甚至，对程序行为的系统化调整也经常可以通过对实际代码作相对少量的修改来实现。这也意味着可以更快速地开发，编写更多的代码，也不必再花时间在语言的限制里寻求更简洁的方式来表达想法了。^①

Common Lisp也是一门适合做探索性编程的优秀语言。如果在刚开始编写程序的时候对整个工作机制还不甚明了，Common Lisp提供了一些特性可以有助于实现递进的交互式开发。

比如，将在下一章介绍的交互式“读-求值-打印”循环，它可以使你在开发过程中持续地与程序交互。编写新函数、测试它、修改它、尝试不同的实现方法，从而使思路不会因漫长的编译周期而停滞下来。^②

其他支持连贯的交互式编程风格的语言特性，还包括Lisp的动态类型机制以及Common Lisp状况系统（condition system）。由于前者的存在，只需花较少的时间就能让编译器把代码跑起来，然后把更多的时间放在如何实际运行和改进代码上^③，而后者甚至可以实现交互式地开发错误处理代码。

① 遗憾的是，在不同语言的生产力方面几乎没有实际的研究。一份阐明了在程序员和程序效率的组合上，Lisp相比于C++和Java脱颖而出的报告在<http://www.norvig.com/java-lisp.html>上有所讨论。

② 心理学家已经鉴别出大脑的一种称为连贯性（flow）的状态，这时我们可以产生高度的注意力和生产力。连贯性对于编程的重要性在近二十年以前就已经被认识到了，其最早在一本经典的关于编程的人为因素的书《人件：富有成果的项目和团队》（*Peopleware: Productive Projects and Teams*，Tom DeMarco和Timothy Lister著，Dorset House，1987年）里讨论过。连贯性的两个关键因素是人们需要15分钟才能进入连贯性状态，而即便短暂的打岔也会使人完全退出这一状态，从而需要另外15分钟才能重新进入状态。DeMarco和Lister，以及多数后来的作者，都很反感诸如电话铃和老板的贸然来访这类破坏连贯性的打岔。较少被考虑到但可能对程序员同样重要的是我们工具所造成的打岔，例如，那些在你测试最新代码之前需要经历冗长的编译过程的语言，其对于连贯性的影响可能跟吵闹的电话铃或者爱管闲事的老板同样有害。因此，Lisp作为一种可以令你保持连贯状态的语言，也是你应该了解它的一个理由。

③ 至少对某些人来说，这个观点很可能有争议。静态和动态类型的信仰之争在编程领域由来已久。如果你信奉C++和Java（或者是诸如Haskell和ML的静态类型函数式编程语言），并且拒绝生活在没有静态类型检查的环境里，你可能会就此合上本书了。不过，在此之前，你最好先查查“静态类型偏执狂”Robert Martin[*Design Object Oriented C++ Applications Using the Booch Method*(Prentice Hall，1995年)的作者]以及C++和Java领域的作者Bruce Eckel[*Thinking in C++* (Prentice Hall，1995年)和*Thinking in Java*(Prentice Hall，1998年)的作者]在他们的博客(<http://www.artima.com/weblogs/viewpost.jsp?thread=4639>和<http://www.mindview.net/WebLog/log-0025>)上是怎样自我描述的。另一方面，信奉SmallTalk、Python、Perl或者Ruby的人们应该会对Common Lisp的这方面感觉良好。

作为一门“可编程的编程语言”，Common Lisp除了支持各种小修小补以便开发人员更容易地编写某些程序之外，对于那些从根本上改变编程方式的新思想的支持也是绰绰有余的。例如，Common Lisp强大的对象系统CLOS (Common Lisp Object System)，其最初的实现就是一个用可移植的Common Lisp写成的库。而在这个库正式成为语言的一部分之前，Lisp程序员就可以体验其实际功能了。

目前来看，无论下一个流行的编程范例是什么，Common Lisp都极有可能在无需修改其语言核心部分的情况下将其吸纳进来。例如，最近有个Lisp程序员写了一个叫做AspectL的库，它为Common Lisp增加了对面向方面编程 (AOP) 的支持。^①如果AOP将主宰编程的未来，那么Common Lisp将可以直接支持它而无需对基础语言作任何修改，并且也不需要额外的预处理器和编译器。^②

1.2 Lisp 的诞生

Common Lisp是1956年John McCarthy发明的Lisp语言的现代版本。Lisp在1956年被设计用于“符号数据处理”^③，而Lisp这个名字本身就来源于其最擅长的工作：列表处理 (LISt Processing)。从那时起，Lisp得到了长足的发展。Common Lisp引人瞩目地具备一系列现代数据类型：将在第19章里介绍的状况系统提供了Java、Python和C++等语言的异常系统里所没有的充分灵活性，强大的面向对象编程支持，以及其他编程语言里完全不存在的一些语言机制。这一切怎么可能呢？怎么会进化出如此装备精良的语言来呢？

原来，McCarthy曾经是（现在也是）一名人工智能（AI）研究者，他在Lisp语言的最初版本里内置的很多特性，使其成为了AI编程的绝佳语言。在AI繁荣昌盛的20世纪80年代，Lisp始终是程序员们所偏爱的工具，广泛用于编写软件来求解包括自动定理证明、规划和调度以及计算机视觉在内的各种难题。这些问题都需要大量难于编写的软件，为此，AI程序员们需要一门强大的语言，而他们就将Lisp发展成了这样一门语言。另外，冷战也起了积极的作用——五角大楼向国防高级研究规划局（DARPA）投入了大量资金，其中的相当一部分用于研究诸如大规模战场模拟、自动规划以及自然语言接口等问题。这些研究人员也在使用Lisp并且持续地对其进行改进以满足自身需要。

推动Lisp特性进化的动力也同样推动了其他相关领域的发展——大型的AI问题无论如何编码总是要耗费大量的计算资源，而如果按照摩尔定律倒推20年，你就可以想象20世纪80年代的计

① AspectL是一个跟它的Java版前任AspectJ同样有趣的项目，后者由Common Lisp对象和元对象系统的设计者之一Gregor Kiczales所作。对许多Lisp程序员来说，AspectJ看起来就像是Kiczales尝试将其思想从Common Lisp向后移植到了Java。尽管如此，AspectL的作者Pascal Costanza认为，AOP许多有趣的思想可能对Common Lisp有用。当然，他能够将AspectL以库的形式实现的根本原因在于，Kiczales所设计的Common Lisp元对象协议 (Meta Object Protocol) 具有难以想象的灵活性。为了实现AspectJ，Kiczales不得不写了一个单独的编译器，将一种新语言编译成Java源代码。AspectL项目的主页是<http://common-lisp.net/project/aspectl/>。

② 或者我们可以从另一个技术上更精确的方面来看待这件事：Common Lisp提供了内置的功能以方便集成嵌入式语言的编译器。

③ *Lisp 1.5 Programmer's Manual* (麻省理工学院出版社, 1962年)。

算资源是何等的贫乏了。Lisp工作者们不得不想尽办法从实现中获得更多的性能。现代Common Lisp实现就是这些早期工作的结晶，它们通常都带有相当专业的可产生原生机器码的编译器。感谢摩尔定律，今天我们从任何纯解释型语言里也能获得可接受的性能了，性能对于Common Lisp来说再也不是问题了。不过，你在第32章可以看到，通过使用适当的（可选）变量声明，一个好的Lisp编译器所生成的机器码，完全可以跟C编译器生成的机器码相媲美。

20世纪80年代也是Lisp机的年代，当时好几家公司（其中最著名的是Symbolics）都在生产可以在芯片上直接运行Lisp的计算机系统。Lisp因此成了系统编程语言，被广泛用于编写操作系统、编辑器、编译器，以及Lisp机上的大量其他软件。

事实上，到了20世纪80年代早期，几家AI实验室和Lisp机厂商都提供了他们自己的Lisp实现，众多的Lisp系统和方言让DARPA开始担心Lisp社区可能走向分裂。为了应对这些担忧，一个由Lisp黑客组成的草根组织于1981年成立，旨在结合既有Lisp方言之所长，定义一种新的称为Common Lisp的标准化Lisp语言。最后，他们的工作成果记录在了Guy Steele的*Common Lisp the Language (CLtL, Digital press, 1984年)*一书里，该书相当于Lisp的圣经。

到1986年的时候，首批Common Lisp实现诞生了，它们是在Common Lisp试图取代的那些方言的基础上写成的。1996年，美国国家标准学会（ANSI）发布了一个建立在CLtL之上并加以扩展的Common Lisp标准，其中增加了一些主要的新特性，包括CLOS和状况系统。但事情还没结束：跟此前的CLtL一样，ANSI标准有意为语言实现者保留一定的空间，以试验各种最佳的工作方式。一个完整的Lisp实现将带有丰富的运行时环境，并提供GUI微件、多线程控制和TCP/IP套接字等。今天的Common Lisp则进化得更像其他的开源语言——用户可以编写他们所需要的库并开放给其他人使用。在过去的几年里，开源Lisp库领域尤为活跃。

所以，一方面，Lisp是计算机科学领域的“经典语言”之一，构建在经过时间考验的各种思想之上。^①另一方面，它完全是一门现代的通用语言，其设计反映了尽可能高效可靠地求解实际问题的实用主义观点。Lisp“经典”遗产的唯一缺点是，许多人仍然生活在片面的Lisp背景之下，他们可能只是在McCarthy发明Lisp以来的近半个世纪中的某些特定时刻接触到了这门语言的某些方面。如果有人告诉你Lisp只能被解释执行，因此会很慢，或者你不得不用递归来干每件事，那么一定要问问他们究竟在谈论哪种Lisp方言，以及他们是否是在计算机远古时代学到这些东西的。^②

^① 首先由Lisp引进的编程思想包括if/then/else控制结构、递归函数调用、动态内存分配、垃圾收集、第一类(first-class)函数、词法闭包、交互式编程、增量编译以及动态类型。

^② 关于Lisp的一个最常见的说法是该语言“已死”。虽然Common Lisp确实不如Visual Basic或者Java这些语言使用广泛，但把一个继续用于新的开发并且继续吸引新用户的语言描述成“已死”看起来有些奇怪。一些近期的Lisp成功案例包括Paul Graham的Viaweb，在Yahoo买下了Paul的公司以后变成了Yahoo Store；由在线机票销售商Orbitz等使用的ITA Software的航空票务系统QPX；Naughty Dog运行在PlayStation 2上的游戏Jak and Daxter，在很大程度上是用Naughty Dog发明的一种称为GOAL的特定领域Lisp方言写成的，其编译器本身是用Common Lisp写的；还有自动机器人真空吸尘器Roomba，其软件是用L语言写的，后者是Common Lisp的向下兼容子集。也许最有说服力的是承载开源Common Lisp项目的Web站点Common-Lisp.net的成长，以及各种本地Lisp用户组过去几年里在数量上的显著增长。

我曾经学过Lisp，不过跟你所描述的不太一样

如果你以前用过Lisp，你对Lisp的认识很可能对学习Common Lisp没什么帮助。尽管Common Lisp取代了大多数它所继承下来的方言，但它并非仅存的Lisp方言。你要清楚自己是在何时何地认识Lisp的，很有可能你学的是某种其他方言。

除了Common Lisp以外，另一种仍然有着活跃用户群的通用Lisp方言是Scheme。Common Lisp从Scheme那里吸收了一些重要的特性，但从未试图取代它。

Scheme最早在MIT设计出来，然后很快用作本科计算机科学课程的教学语言，它一直被定位成一种与Common Lisp有所不同的语言。特别是Scheme的设计者们将注意力集中在使其语言核心尽可能地小而简单。这对作为教学语言来说非常有用，编程语言研究者们也很容易形式化地证明语言本身的有关命题。

这样设计的另一个好处是使得通过标准规范理解整个语言变得相对简单，但这样做带来的问题是缺失了许多在Common Lisp里已经标准化了的有用特性。个别的Scheme实现者可能以特定的实现方式提供了这些特性，但它们在标准中的缺失则使得编写可移植的Scheme代码比编写可移植的Common Lisp代码更加困难。

Scheme同样强调函数式的编程风格，并且使用了比Common Lisp更多的递归。如果在大学里学过Lisp并且感觉它只是一种没有现实应用的学术语言，那你八成是学了Scheme。当然，说Scheme具有这样的特征并不是很公正，只不过同样的说法用在Common Lisp身上更加不合适罢了，后者专门被设计成真实世界的工程语言，而不是一种理论上的“纯”语言。

如果学过Scheme，也应该当心，Scheme和Common Lisp之间的许多细微差别可能会使人犯错。这些差别也是Common Lisp和Scheme社区的狂热分子之间一些长期信仰之争的导火索。日后随着讨论的深入，我将指出其中最重要的差别。

另外两种仍然广泛使用的Lisp方言是Elisp和Autolisp。Elisp是Emacs编辑器的扩展语言，而Autolisp是Autodesk的AutoCAD计算机辅助设计工具的扩展语言。尽管用Elisp和Autolisp可能已经写出了超过任何其他方言的代码行数，但它们都不能用在各自的宿主应用程序之外，而且它们无论与Scheme还是Common Lisp相比都是相当过时的Lisp方言了。如果你曾经用过这些方言的一种，就需要做好准备，你可能要在Lisp时间机器里向前跳跃几十年了。

1.3 本书面向的读者

如果你对Common Lisp感兴趣，那么无论是否已经确定要使用它或者只是想一窥其门径，本书都挺适合你的。

如果你已经学会了一些Lisp，但却难于跨越从学术训练到真实程序之间的鸿沟，那么本书刚好可以帮你走上正途。而另一方面，你也不必带着学以致用的目的来阅读本书。

如果你是个顽强的实用主义者，想知道Common Lisp相比Perl、Python、Java、C和C#这些语言都有哪些优势，那么本书应该可以提供一些思路。或者你根本就没打算使用Lisp——可能是因为已经确信Lisp并不比已知的其他语言更好，但由于不熟悉它而无法反驳那些Lisp程序员。如果

是这样，本书将给你一个直奔Common Lisp主题的介绍。如果在读完本书以后，你仍然认为Common Lisp赶不上自己当前喜爱的其他语言，那么你将有充分理由来说明自己的观点了。

我不但介绍了该语言的语法和语义，还讲述了如何使用它来编写有用的软件。在本书的第一部分，我将谈及语言本身，同时穿插一些实践性章节，展示如何编写真实的代码。接着，在我阐述了该语言的绝大部分内容后——包括某些在其他书里往往留给你自己去考查的内容，将给出九个更实用的章节，帮助你编写一些中等大小的程序来做一些你可能认为有用的事：过滤垃圾邮件、解析二进制文件、分类MP3、通过网络播放MP3流媒体，以及为MP3目录和服务器提供Web接口。

读完本书后，你将熟悉该语言的所有最重要的特性以及它们是如何组织在一起的，而且你已经用Common Lisp写出了一些非凡的程序，并且可以凭借自身力量继续探索该语言了。尽管每个人的Lisp之路都有所不同，但我还是希望本书可以帮助你少走些弯路。那么，让我们就此上路吧。



第2章

周而复始：REPL简介

2

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，你需要一个可以帮助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各个方面，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机里，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被传递给打印机从而得以输出。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。而打印机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)

```

2

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>。

实践：简单的数据库



很明显，在你可以用Lisp构建真实软件之前，必须先学会这门语言。但是请想想看——你可能会觉得：“‘Practical Common Lisp’难道不矛盾吗？难道在确定一门语言真正有用之前就要先把它所有的细节都学完吗？”因此，我先给你一个小型的可以用Common Lisp来做的例子。本章将编写一个简单的数据库用来记录CD光盘。在第27章里，为我们的流式MP3服务器构建一个MP3数据库时还会用到类似的技术。事实上，它可以看成是整个MP3软件项目的一部分。毕竟，为了有大量的MP3可听，对我们所拥有并需要转换成MP3的CD加以记录是很有用的。

在本章，我只介绍足以使你理解代码工作原理所需的Lisp特性，但细节方面不会解释太多。目前你不需要执著于细节，接下来的几章将以一种更加系统化的方式介绍这里用到的所有Common Lisp控制结构以及更多内容。

关于术语方面，本章将讨论少量Lisp操作符。第4章将学到Common Lisp所提供的三种不同类型的操作符：函数、宏以及特殊操作符。对于本章来说，你并不需要知道它们的区别。尽管如此，在提及操作符时，我还是会适时地说成是函数、宏或特殊操作符，而不会笼统地用“操作符”这个词来表示。眼下你差不多可以认为函数、宏和特殊操作符是等价的。^①

另外请记住，我不会在这个继“hello, world”后写的首个程序中亮出所有最专业的Common Lisp技术来。本章的重点和意图也不在于讲解如何用Lisp编写数据库，而在于让你对Lisp编程有个大致的印象，并能看到即便相对简单的Lisp程序也可以有着丰富的功能。

3.1 CD 和记录

为了记录那些需要转换成MP3的CD，以及哪些CD应该先进行转换，数据库里的每条记录都将包含CD的标题和艺术家信息，一个关于有多少用户喜欢它的评级，以及一个表示其是否已经转换过的标记。因此，首先需要一种方式来表示一条数据库记录（也就是一张CD）。Common Lisp提供了大量可供选择的数据结构——从简单的四元素列表到基于Common Lisp对象系统（CLOS）的用户自定义类。

^① 尽管如此，在正式开始之前，至关重要的一点是，你必须忘记所有关于C预处理器所实现的#define风格“宏”的知识。Lisp宏是完全不同的东西。

眼下你只能选择该系列里最简单的方法——使用列表。你可以使用LIST函数来生成一个列表，如果正常执行的话，它将返回一个由其参数所组成的列表。

```
CL-USER> (list 1 2 3)
(1 2 3)
```

还可以使用一个四元素列表，将列表中的给定位置映射到记录中的给定字段。然而，使用另一类被称为属性表（property list, plist）的列表甚至更方便。属性表是这样一种列表：从第一个元素开始的所有相间元素都是一个用来描述接下来的那个元素的符号。目前我不会深入讨论关于符号的所有细节，基本上它就是一个名字。对于用来命名CD数据库字段的名字，你可以使用一种特殊类型的符号——关键字（keyword）符号。关键字符号是任何以冒号开始的名字，例如，:foo。下面是一个使用了关键字符号:a、:b和:c作为属性名的示例plist：

```
CL-USER> (list :a 1 :b 2 :c 3)
(:A 1 :B 2 :C 3)
```

注意，你可以使用和创建其他列表时同样的LIST函数来创建一个属性表，只是特殊的内容使其成为了属性表。

真正令属性表便于表达数据库记录的原因则是在于函数GETF的使用，它接受一个plist和一个符号，并返回plist中跟在那个符号后面的值，这使得plist成为了穷人的哈希表。当然，Lisp有真正的哈希表，但plist足以满足当前需要，并且可以更容易地保存在文件里（后面将谈及这点）。

```
CL-USER> (getf (list :a 1 :b 2 :c 3) :a)
1
CL-USER> (getf (list :a 1 :b 2 :c 3) :c)
3
```

理解了所有这些知识，你就可以轻松写出一个make-cd函数了，它以参数的形式接受4个字段，然后返回一个代表该CD的plist。

```
(defun make-cd (title artist rating ripped)
  (list :title title :artist artist :rating rating :ripped ripped))
```

单词DEFUN告诉我们上述形式正在定义一个新函数，函数名是make-cd。跟在名字后面的是形参列表，这个函数拥有四个形参：title、artist、rating和ripped。形参列表后面的都是函数体。本例中的函数体只有一个形式，即对LIST的调用。当make-cd被调用时，传递给该调用的参数将被绑定到形参列表中的变量上。例如，为了建立一个关于Kathy Mattea的名为Roses的CD的记录，你可以这样调用make-cd：

```
CL-USER> (make-cd "Roses" "Kathy Mattea" 7 t)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T)
```

3.2 录入 CD

只有单一记录还不能算是一个数据库，需要一些更大的结构来保存记录。出于简化目的，使用列表似乎也还不错。同样出于简化目的，也可以使用一个全局变量*db*，它可以用DEFVAR宏

来定义。名字中的星号是Lisp的全局变量命名约定。^①

```
(defvar *db* nil)
```

可以使用PUSH宏为*db*添加新的项。但稍微做得抽象一些可能更好，因此可以定义一个函数add-record来给数据库增加一条记录。

```
(defun add-record (cd) (push cd *db*))
```

3

现在可以将add-record和make-cd一起使用，来为数据库添加新的CD记录了。

```
CL-USER> (add-record (make-cd "Roses" "Kathy Mattea" 7 t))
((:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Fly" "Dixie Chicks" 8 t))
((:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Home" "Dixie Chicks" 9 t))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

那些每次调用add-record以后REPL所打印出来的东西是返回值，也就是函数体中最后一个表达式PUSH所返回的值，并且PUSH返回它正在修改的变量的新值。因此你看到的其实是每次新记录被添加以后整个数据库的值。

3.3 查看数据库的内容

无论何时，在REPL里输入*db*都可以看到*db*的当前值。

```
CL-USER> *db*
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

但这种查看输出的方式并不令人满意。可以用一个dump-db函数来将数据库转储成一个像下面这样的更适合人类阅读习惯的格式。

```
TITLE: Home
ARTIST: Dixie Chicks
RATING: 9
RIPPED: T

TITLE: Fly
ARTIST: Dixie Chicks
RATING: 8
RIPPED: T

TITLE: Roses
```

^① 使用全局变量也有一些缺点。例如，你每时每刻只能有一个数据库。在第27章，等学会了更多的语言特性以后，你可以构建一个更加灵活的数据库。但在第6章你会看到，即便是使用一个全局变量，在Common Lisp里也比其他语言更为灵活。

```
ARTIST: Kathy Mattea
RATING: 7
RIPPED: T
```

该函数如下所示。

```
(defun dump-db ()
  (dolist (cd *db*)
    (format t "~{~a: ~10t~a~%~}~%" cd)))
```

该函数的工作原理是使用**DOLIST**宏在*db*的所有元素上循环，依次绑定每个元素到变量cd上。而后使用**FORMAT**函数打印出每个cd的值。

不可否认，这个**FORMAT**调用多少显得有些晦涩。尽管如此，但**FORMAT**却并不比C或Perl的**printf**函数或者Python的**string-%**操作符更复杂。第18章将进一步讨论**FORMAT**的细节，目前我们只需记住这个调用就可以了。如第2章所述，**FORMAT**至少接受两个实参，第一个是它用来发送输出的流，t是标准输出流(*standard-output*)的简称。

FORMAT的第二个实参是一个格式字符串，内容既包括字面文本，也包括那些告诉**FORMAT**如何插入其余参数等信息的指令。格式指令以~开始（就像是**printf**指令以%开始那样）。**FORMAT**能够接受大量的指令，每一个都有自己的选项集。^①但目前我只关注那些对编写dump-db有用的选项。

~a指令是美化指令，它的意图是消耗一个实参，然后将其输出成人类可读的形式。这将使得关键字被渲染成不带前导冒号的形式，而字符串也不再有引号了。例如：

```
CL-USER> (format t "~a" "Dixie Chicks")
Dixie Chicks
NIL
```

或是：

```
CL-USER> (format t "~a" :title)
TITLE
NIL
```

~t指令用于制表。**~10t**告诉**FORMAT**产生足够的空格，以确保在处理下一个~a之前将光标移动10列。**~t**指令不使用任何实参。

```
CL-USER> (format t "~a:~10t~a" :artist "Dixie Chicks")
ARTIST: Dixie Chicks
NIL
```

现在事情变得稍微复杂一些了。当**FORMAT**看到~{的时候，下一个被使用的实参必须是一个列表。**FORMAT**在列表上循环操作，处理位于~{和~}之间的指令，同时在每次需要时，从列表上

^① 最酷的一个**FORMAT**指令是~R指令。曾经想知道如何用英语来说一个真正的大数吗？Lisp知道。求值这个：

```
(format nil "~r" 1606938044258990275541962092)
```

你将得到下面的结果（为清楚起见作了折行处理）：

```
one octillion six hundred six septillion nine hundred thirty-eight sextillion forty-four quintillion two hundred fifty-eight
quadrillion nine hundred ninety trillion two hundred seventy-five billion five hundred forty-one million nine hundred
sixty-two thousand ninety-two
```

使用尽可能多的元素。在dump-db里，FORMAT循环将在每次循环时从列表上消耗一个关键字和一个值。`~%`指令并不消耗任何实参，而只是告诉FORMAT来产生一个换行。然后在`~}`循环结束以后，最后一个`~%`告诉FORMAT再输出一个额外的换行，以便在每个CD数据间产生一个空行。

从技术上来讲，也可以使用FORMAT在整个数据库本身上循环，从而将dump-db函数变成只有一行。

```
(defun dump-db ()
  (format t "~~{(~a: ~10t~a~%~}~%~}" *db*))
```

3

这件事究竟算酷还是恐怖，完全看你怎么想。

3.4 改进用户交互

尽管add-record函数在添加记录方面做得很好，但对于普通用户来说却仍显得过于Lisp化了。并且如果他们想要添加大量的记录，这种操作也并不是很方便。因此你可能想要写一个函数来提示用户输入一组CD信息。这就意味着需要以某种方式来提示用户输入一条信息，然后读取它。下面让我们来写这个。

```
(defun prompt-read (prompt)
  (format *query-io* "~a: " prompt)
  (force-output *query-io*)
  (read-line *query-io*))
```

你用老朋友FORMAT来产生一个提示。注意到格式字符串里并没有`~%`，因此光标将停留在同一行里。对FORCE-OUTPUT的调用在某些实现里是必需的，这是为了确保Lisp在打印提示信息之前不会等待换行。

然后就可以使用名副其实的READ-LINE函数来读取单行文本了。变量`*query-io*`是一个含有关联到当前终端的输入流的全局变量（通过对全局变量的星号命名约定你也可以看出这点来）。`prompt-read`的返回值将是其最后一个形式，即调用READ-LINE所得到的值，也就是它所读取的字符串（不包括结尾的换行）。

你可以将已有的make-cd函数跟prompt-read组合起来，从而构造出一个函数，可以从依次提示输入每个值得到的数据中建立新的CD记录。

```
(defun prompt-for-cd ()
  (make-cd
    (prompt-read "Title")
    (prompt-read "Artist")
    (prompt-read "Rating")
    (prompt-read "Ripped [y/n]")))
```

这样已经差不多正确了。只是prompt-read总是返回字符串，对于Title和Artist字段来说可以，但对于Rating和Ripped字段来说就不太好，它们应该是数字和布尔值。花在验证用户输入数据上的努力可以是无止境的，而这取决于所要实现的用户接口专业程度。目前我们倾向于一种快餐式办法，你可以将关于评级的那个prompt-read包装在一个Lisp的PARSE-INTEGER函数

里，就像这样：

```
(parse-integer (prompt-read "Rating"))
```

不幸的是，**PARSE-INTEGER**的默认行为是当它无法从字符串中正确解析出整数，或者字符串里含有任何非数字的垃圾时直接报错。不过，它接受一个可选的关键字参数：`junk-allowed`，可以让其适当地宽容一些。

```
(parse-integer (prompt-read "Rating") :junk-allowed t)
```

但还有一个问题：如果无法在所有垃圾里找出整数的话，**PARSE-INTEGER**将返回NIL而不是整数。为了保持这个快餐式的思路，你可以把这种情况当作0来看待。Lisp的**OR**宏就是你在此时所需要的。它与Perl、Python、Java以及C中的“短路”符号||很类似，它接受一系列表达式，依次对它们求值，然后返回第一个非空的值（或者空值，如果它们全部是空值的话）。所以可以使用下面这样的语句：

```
(or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
```

来得到一个默认值0。

修复Ripped提示的代码就更容易了，只需使用Common Lisp的**Y-OR-N-P**函数：

```
(y-or-n-p "Ripped [y/n]: ")
```

事实上，这将是**prompt-for-cd**中最健壮的部分，因为**Y-OR-N-P**会在输入了没有以y、Y、n或者N开始的内容时重新提示输入。

将所有这些内容放在一起，就得到了一个相当健壮的**prompt-for-cd**函数了。

```
(defun prompt-for-cd ()
  (make-cd
    (prompt-read "Title")
    (prompt-read "Artist")
    (or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
    (y-or-n-p "Ripped [y/n]: ")))
```

最后可以将**prompt-for-cd**包装在一个不停循环直到用户完成的函数里，以此来搞定这个“添加大量CD”的接口。可以使用**LOOP**宏的一种简单形式，它不断执行一个表达式体，最后通过调用**RETURN**来退出。例如：

```
(defun add-cds ()
  (loop (add-record (prompt-for-cd))
    (if (not (y-or-n-p "Another? [y/n]: ")) (return))))
```

现在可以使用**add-cds**来添加更多CD到数据库里了。

```
CL-USER> (add-cds)
Title: Rockin' the Suburbs
Artist: Ben Folds
Rating: 6
Ripped [y/n]: y
Another? [y/n]: y
Title: Give Us a Break
Artist: Limpopo
```

```

Rating: 10
Ripped [y/n]: y
Another? [y/n]: y
Title: Lyle Lovett
Artist: Lyle Lovett
Rating: 9
Ripped [y/n]: y
Another? [y/n]: n
NIL

```

3



3.5 保存和加载数据库

用一种便利的方式来给数据库添加新记录是件好事。但如果让用户不得不在每次退出并重启Lisp以后再重新输入所有记录，他们是绝对不会高兴的。幸好，借助用来表示数据的数据结构，可以相当容易地将数据保存在文件里并在稍后重新加载。下面是一个`save-db`函数，它接受一个文件名作为参数并且保存当前数据库的状态：

```
(defun save-db (filename)
  (with-open-file (out filename
                        :direction :output
                        :if-exists :supersede)
    (with-standard-io-syntax
      (print *db* out))))
```

`WITH-OPEN-FILE`宏会打开一个文件，将文件流绑定到一个变量上，执行一组表达式，然后再关闭这个文件。它还可以保证即便在表达式体求值出错时也可以正确关闭文件。紧跟着`WITH-OPEN-FILE`的列表并非函数调用而是`WITH-OPEN-FILE`语法的一部分。它含有用来保存要在`WITH-OPEN-FILE`主体中写入的文件流的变量名，这个值必须是文件名，紧随其后是一些控制如何打开文件的选项。这里用`:direction :output`指定了正在打开一个用于写入的文件，以及用`:if-exists :supersede`说明当存在同名的文件时想要覆盖已存在的文件。

一旦已经打开了文件，所需做的就只是使用`(print *db* out)`将数据库的内容打印出来。跟`FORMAT`不同的是，`PRINT`会将Lisp对象打印成一种可以被Lisp读取器读回来的形式。`WITH-STANDARD-IO-SYNTAX`确保那些影响`PRINT`行为的特定变量可以被设置成它们的标准值。当把数据读回来时，你将使用同样的宏来确保Lisp读取器和打印机的操作彼此兼容。

`save-db`的实参应该是一个含有用户打算用来保存数据库的文件名字符串。该字符串的确切形式取决于正在使用什么操作系统。例如，在Unix系统上可能会这样调用`save-db`：

```
CL-USER> (save-db "/home/peter/my-cds.db")
(:TITLE "Lyle Lovett" :ARTIST "Lyle Lovett" :RATING 9 :RIPPED T)
(:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T)
(:TITLE "Rockin' the Suburbs" :ARTIST "Ben Folds" :RATING 6 :RIPPED T)
(:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
(:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 9 :RIPPED T))
```

而在Windows下，文件名可能会是“c:/my-cds.db”或“c:\\my-cds.db”。^①

可以在任何文本编辑器里打开这个文件来查看它的内容。所看到的东西应该和直接在REPL里输入*db*时看到的东西差不多。

将数据加载回数据库的函数其形式也差不多。

```
(defun load-db (filename)
  (with-open-file (in filename)
    (with-standard-io-syntax
      (setf *db* (read in)))))
```

这次不需要在WITH-OPEN-FILE的选项里指定:direction了，因为你要的是默认值:input。并且与打印相反，所做的是使用函数READ来从流中读入。这是与REPL使用的相同的读取器，可以读取你在REPL提示符下输入的任何Lisp表达式。但本例中只是读取和保存表达式，并不会对它求值。WITH-STANDARD-IO-SYNTAX宏再一次确保READ使用和save-db在打印数据时相同的基本语法。

SETF宏是Common Lisp最主要的赋值操作符。它将其第一个参数设置成其第二个参数的求值结果。因此在load-db里，变量*db*将含有从文件中读取的对象，也就是由save-db所写入的那些列表组成的列表。需要特别注意一件事——load-db会破坏其被调用之前*db*里面的东西。因此，如果已经用add-record或者add-cds添加了尚未用save-db保存的记录，就将失去它们。

3.6 查询数据库

有了保存和重载数据库的方法，并且可以用一个便利的用户接口来添加新记录，因此很快就会出现足够多的记录，但你并不想为了查看它里面有什么而每次都把整个数据库导出来。需要采用一种方式来查询数据，比如类似于下面这样的语句。

```
(select :artist "Dixie Chicks")
```

然后就可以列出艺术家Dixie Chicks的所有记录。这又证明了当初选择用列表来保存记录是明智的。

函数REMOVE-IF-NOT接受一个谓词和一个原始列表，然后返回一个仅包含原始列表中匹配该谓词的所有元素的新列表。换句话说，它删除了所有不匹配该谓词的元素。然而REMOVE-IF-NOT并没有真的删除任何东西——它会创建一个新列表，而不会去碰原始列表。这就好比是在一个文件上运行grep。谓词参数可以是任何接受单一参数并能返回布尔值的函数——除了NIL表示假以外其余的都表示真。

举个例子，假如要从一个由数字组成的列表里抽出所有偶数来，就可以像下面这样来使用REMOVE-IF-NOT：

```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

^① Windows事实上可以理解文件名中的正斜杠 (/)，尽管它正常情况下是使用反斜杠 (\) 作为目录分隔符的。这是一个很方便的特性，否则的话我们将不得不写成双反斜杠，因为反斜杠是Lisp字符串的转义字符。

这里的谓词是函数**EVENP**，当其参数是偶数时返回真。那个有趣的#’记号是“获取函数，其名如下”的简称。没有#’的话，Lisp将把evenp作为一个变量名来对待并查找该变量的值，而不是将其看作函数。

你也可以向**REMOVE-IF-NOT**传递一个匿名函数。例如，如果**EVENP**不存在，你也可以像下面这样来写前面给出的表达式：

```
CL-USER> (remove-if-not #'(lambda (x) (= 0 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

在这种情况下，谓词是下面这个匿名函数：

```
(lambda (x) (= 0 (mod x 2)))
```

它会检查其实参与取模2时等于0的情况（换句话说，就是偶数）。如果想要用匿名函数来抽出所有的奇数，就可以这样写：

```
CL-USER> (remove-if-not #'(lambda (x) (= 1 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(1 3 5 7 9)
```

注意，**lambda**并不是函数的名字——它只是一个表明你正在定义匿名函数的指示器。^①但除了缺少名字以外，一个**LAMBDA**表达式看起来很像一个**DEFUN**：单词**lambda**后面紧跟着形参列表，然后是函数体。

为了用**REMOVE-IF-NOT**从数据库里选出所有Dixie Chicks的专辑，需要可以在一条记录的艺术家字段是“Dixie Chicks”时返回真的函数。请记住，我们之所以选择plist来表达数据库的记录，是因为函数**GETF**可以从plist里抽出给定名称的字段来。因此假设cd是保存着数据库单一记录的变量名，那么可以使用表达式(**getf cd :artist**)来抽出艺术家名字来。当给函数**EQUAL**赋予字符串参数时，可以逐个字符地比较它们。因此(**equal (getf cd :artist) "Dixie Chicks"**)将测试一个给定CD的艺术家字段是否等于“Dixie Chicks”。所需做的只是将这个表达式包装在一个**LAMBDA**形式里，从而得到一个匿名函数，然后传递给**REMOVE-IF-NOT**。

```
CL-USER> (remove-if-not
  #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")) *db*)
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

假设要将整个表达式包装进一个接受艺术家名字作为参数的函数里，可以写成这样：

```
(defun select-by-artist (artist)
  (remove-if-not
  #'(lambda (cd) (equal (getf cd :artist) artist))
  *db*))
```

这个匿名函数的代码直到其被**REMOVE-IF-NOT**调用才会运行，注意看它是如何访问到变量**artist**的。在这种情况下，匿名函数不仅使你免于编写一个正规函数，而且还可编写出一个其部分含义(**artist**值)取自上下文环境的函数。

^① 单词**lambda**用于Lisp是因为该语言早期跟**lambda**演算之间的关系，后者是一种为研究数学函数而发明的数学形式化方法。

以上就是select-by-artist。尽管如此，通过艺术家来搜索只是你想要支持的各种查询方法的一种，还可以编写其他几个函数，诸如select-by-title、select-by-rating、select-by-title-and-artist，等等。但它们之间除了匿名函数的内容以外就没有其他区别了。换个做法，可以做出一个更加通用的select函数来，它接受一个函数作为其实参。

```
(defun select (selector-fn)
  (remove-if-not selector-fn *db*))
```

但是#'哪里去了？这是因为你并不希望REMOVE-IF-NOT在此使用一个名为selector-fn的函数。它应该使用的是一个作为select的实参传递到变量selector-fn里的匿名函数。不过，在对select的调用中，#'还是会出现。

```
CL-USER> (select #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")))
(:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
(:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

但这样看起来相当乱。所幸可以将匿名函数的创建过程包装起来。

```
(defun artist-selector (artist)
  #'(lambda (cd) (equal (getf cd :artist) artist)))
```

这是一个返回函数的函数，并且返回的函数里引用了一个似乎在artist-selector返回以后将不会存在的变量。^①尽管现在可能看起来有些奇怪，但它确实可以按照你所想象的方式来工作——如果用参数"Dixie Chicks"调用artist-selector，那么将得到一个可以匹配其:artist字段为"Dixie Chicks"的CD的匿名函数，而如果用"Lyle Lovett"来调用它，就将得到另一个匹配:artist字段为"Lyle Lovett"的函数。所以现在可以像下面这样来重写前面的select调用了：

```
CL-USER> (select (artist-selector "Dixie Chicks"))
(:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
(:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

现在只需要用更多的函数来生成选择器了。但正如不想编写select-by-title、select-by-rating等雷同的东西那样，你也不会想去写一大堆长相差不多每个字段写一个的选择器函数生成器。那么为什么不写一个通用的选择器函数生成器呢？让它根据传递给它的参数，生成用于不同字段甚至字段组合的选择器函数。完全可以写出这样一个函数来，不过首先需要快速学习一下关键字形参(keyword parameter)的有关内容。

目前写过的函数使用的都是一个简单的形参列表，随后被绑定到函数调用中对应的实参上。例如，下列函数：

```
(defun foo (a b c) (list a b c))
```

有3个形参，a、b和c，并且必须用3个实参来调用。但有时可能想要编写一个可以用任何数量的实参来调用的函数，关键字形参就是其中一种实现方式。使用关键字形参的foo版本可能看起来

^① 一个引用了其封闭作用域中变量的函数，称为闭包——因为函数“封闭包装”了变量。我将在第6章里讨论闭包的更多细节。

是这样的：

```
(defun foo (&key a b c) (list a b c))
```

它与前者唯一的区别在于形参列表的开始处有一个`&key`。但是，对这个新`foo`的调用方法将是截然不同的。下面这些调用都是合法的，同时在→的右边给出了相应结果。

```
(foo :a 1 :b 2 :c 3) → (1 2 3)
(foo :c 3 :b 2 :a 1) → (1 2 3)
(foo :a 1 :c 3) → (1 NIL 3)
(foo) → (NIL NIL NIL)
```

这些示例显示，变量`a`、`b`和`c`的值被绑定到了跟在相应关键字后面的值上。并且如果一个特定的关键字在调用中没有指定，那么对应的变量将被设置成`NIL`。关于关键字形参如何指定以及它们与其他类型形参的关系等诸多细节在此不予赘述，不过你还需要知道其中一点。

正常情况下，如果所调用的函数没有为特定关键字形参传递实参，该形参的值将为`NIL`。但有时你可能想要区分作为实参显式传递给关键字形参的`NIL`和作为默认值的`NIL`。为此，在指定一个关键字形参时，可以将那个简单的名称替换成一个包括形参名、默认值和另一个称为`supplied-p`形参的列表。这个`supplied-p`形参可被设置成真或假，具体取决于实参在特定的函数调用里是否真的被传入相应的关键字形参中。下面是一个使用了该特性的`foo`版本：

```
(defun foo (&key a (b 20) (c 30 c-p)) (list a b c c-p))
```

前面给出同样的调用将产生下面的结果：

```
(foo :a 1 :b 2 :c 3) → (1 2 3 T)
(foo :c 3 :b 2 :a 1) → (1 2 3 T)
(foo :a 1 :c 3) → (1 20 3 T)
(foo) → (NIL 20 30 NIL)
```

通用的选择器函数生成器`where`是一个函数，如果你熟悉SQL数据库的话，就会逐渐明白为什么叫它`where`了。它接受对应于我们的CD记录字段的四个关键字形参，然后生成一个选择器函数，后者可以选出任何匹配`where`子句的CD。例如，它可以让你写出这样的语句来：

```
(select (where :artist "Dixie Chicks"))
```

或是这样：

```
(select (where :rating 10 :ripped nil))
```

该函数看起来是这样的：

```
(defun where (&key title artist rating (ripped nil ripped-p))
  #'(lambda (cd)
    (and
      (if title (equal (getf cd :title) title) t)
      (if artist (equal (getf cd :artist) artist) t)
      (if rating (equal (getf cd :rating) rating) t)
      (if ripped-p (equal (getf cd :ripped) ripped) t))))
```

这个函数返回一个匿名函数，后者返回一个逻辑AND，而其中每个子句分别来自我们CD记录中的一个字段。每个子句会检查相应的参数是否被传递进来，然后要么将其跟CD记录中对应字

段的值相比较，要么在参数没有传进来时返回`t`，也就是Lisp版本的逻辑真。这样，选择器函数将只在CD记录匹配所有传递给`where`的参数时才返回真。^①注意到需要使用三元素列表来指定关键字形参`ripped`，因为你需要知道调用者是否实际传递了：`:ripped nil`，意思是“选择那些`ripped`字段为`nil`的CD”，或者是否它们将`:ripped`整个扔下不管了，意思是“我不在乎那个`ripped`字段的值”。

3.7 更新已有的记录——WHERE 再战江湖

有了完美通用的`select`和`where`函数，是时候开始编写下一个所有数据库都需要的特性了——更新特定记录的方法。在SQL中，`update`命令被用于更新一组匹配特定`where`子句的记录。这听起来像是个很好的模型，尤其是当已经有了一个`where`子句生成器时。事实上，`update`函数只是你已经见过的一些思路的再应用：使用一个通过参数传递的选择器函数来选取需要更新的记录，再使用关键字形参来指定需要改变的值。这里主要出现的新内容是对`MAPCAR`函数的使用，其映射在一个列表上（这里是`*db*`），然后返回一个新的列表，其中含有在原来列表的每个元素上调用一个函数所得到的结果。

```
(defun update (selector-fn &key title artist rating (ripped nil ripped-p))
  (setf *db*
    (mapcar
      #'(lambda (row)
        (when (funcall selector-fn row)
          (if title (setf (getf row :title) title))
          (if artist (setf (getf row :artist) artist))
          (if rating (setf (getf row :rating) rating))
          (if ripped-p (setf (getf row :ripped) ripped)))
        row) *db*)))
```

这里的另一个新内容是`SETF`用在了诸如`(getf row :title)`这样的复杂形式上。第6章将详细讨论`SETF`，目前只需知道它是一个通用的赋值操作符，可用于对各种“位置”而不只是对变量进行赋值即可。（`SETF`和`GETF`具有相似的名字，但这纯属巧合，两者之间并没有特别的关系。）眼下知道执行`(setf (getf row :title) title)`以后的结果就可以了：由`row`所引用的plist将具有紧跟着属性名`:title`后面的那项变量`title`的值。有了这个`update`函数，如果你觉得自己真的很喜欢Dixie Chicks，并且他们的所有专辑的评级应该升到11，那么可以对下列形式求值：

```
CL-USER> (update (where :artist "Dixie Chicks") :rating 11)
NIL
```

^① 注意，在Lisp中，`IF`形式和其他所有东西一样，是一个带有返回值的表达式。它事实上更像是Perl、Java和C语言中的三目运算符`(?:)`，其中这样的写法是合法的：

```
some_var = some_boolean ? value1 : value2;
这样则是不合法的：
some_var = if (some_boolean) value1; else value2;
因为在这些语言里，if是一个语句，而不是一个表达式。
```

这样就可以了。

```
CL-USER> (select (where :artist "Dixie Chicks"))
(:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T)
(:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T))
```

甚至可以更容易地添加一个函数来从数据库里删除记录。

```
(defun delete-rows (selector-fn)
  (setf *db* (remove-if selector-fn *db*)))
```

函数`REMOVE-IF`的功能跟`REMOVE-IF-NOT`正好相反，在它所返回的列表中，所有确实匹配谓词的元素都被删掉。和`REMOVE-IF-NOT`一样，它实际不会影响传入的那个列表，但是通过将结果重新保存到`*db*`中，`delete-rows`^①事实上改变了数据库的内容。^②

3.8 消除重复，获益良多

目前所有的数据库代码，支持插入、选择、更新，更不用说还有用来添加新记录和导出内容的命令行接口，只有50行多点儿。总共就这些。^③

不过这里仍然有一些讨厌的代码重复，看来可以在消除重复的同时使代码更为灵活。我所考虑的重复出现在`where`函数里。`where`的函数体是一堆像这样的子句，每字段一个：

```
(if title (equal (getf cd :title) title) t)
```

眼下情况还不算太坏，但它的开销和所有的代码重复是一样的：如果想要改变它的行为，就不得不改动它的多个副本。并且如果改变了CD的字段，就必须添加或删除`where`的子句。而`update`也需要承担同样的重复。最令人讨厌的一点在于，`where`函数的本意是动态生成一点儿代码来检查你所关心的那些值，但为什么它非要在运行期来检查`title`参数是否被传递进来了呢？

想象一下你正在试图优化这段代码，并且已经发现了它花费太多的时间检查`title`和`where`的其他关键字形参是否被设置了。^④如果真的想要移除所有这些运行期检查，则可以通过一个程序将所有这些调用`where`的位置以及究竟传递了哪些参数都找出来。然后就可以替换每一个对`where`的调用，使用一个只做必要比较的匿名函数。举个例子，如果发现这段代码：

```
(select (where :title "Give Us a Break" :ripped t))
```

- ① 你需要使用`delete-rows`这个名字而不是更明显的`delete`，因为已经有一个Common Lisp函数叫做`DELETE`了。Lisp包系统可以提供你处理这类名字冲突的途径，因此如果你真想要的话，还是可以得到一个叫做`delete`的函数的。但我还没准备来解释关于包的事情。
- ② 如果你担心这些代码产生了内存泄露，那么大可放心：Lisp就是发明垃圾收集（以及相关的堆分配）的那个语言。至少这段代码里没有。`*db*`的旧值所使用的内存将被自动回收，假设没有其他地方持有对它的引用的话。
- ③ 我的一个朋友某一次采访一个从事编程工作的工程师，并问了他一个典型的采访问题：“你怎样判断一个函数或者方法太大了？”“这个嘛，”被采访者说，“我不喜欢任何比我头还大的方法。”“你是说那些无法将所有细节都记到脑子里的方法？”“不，我是说我把我的头放在显示器上，然后那段代码不应该比我的头还大。”
- ④ 考虑到检测一个变量是否为`NIL`是非常省事的，因此很难说检查关键字参数是否被传递的开销对整体性能有可检测到的影响。另一方面，这些由`where`返回的函数刚好位于任何`select`、`update`或`delete-rows`调用的内循环之中，它们将在数据库每一项上都被调用一次。不管怎么说，出于阐述的目的，我们必须把它处理掉。

则可以将其改为：

```
(select
 #'(lambda (cd)
    (and (equal (getf cd :title) "Give Us a Break")
          (equal (getf cd :ripped) t))))
```

注意，这个匿名函数跟where所返回的那个是不同的，你并非在试图节省对where的调用，而是提供了一个更有效率的选择器函数。这个匿名函数只带有在这次调用里实际关心的字段所对应的子句，所以它不会像where可能返回的函数那样做任何额外的工作。

你可能会想象把所有的源代码都过一遍，并以这种方式修复所有对where的调用，但你也会想到这样做将是极其痛苦的。如果它们有足够多，足够重要，那么编写某种可以将where调用转化成你手写代码的预处理器就是非常值得的了。

使这件事变得极其简单的Lisp特性是它的宏（macro）系统。我必须反复强调，Common Lisp的宏和那些在C和C++里看到的基于文本的宏，从本质上讲，除了名字相似以外就再没有其他共同点了。C预处理器操作在文本替换层面上，对C和C++的结构几乎一无所知；而Lisp宏在本质上是一个由编译器自动为你运行的代码生成器。^①当一个Lisp表达式包含了对宏的调用时，Lisp编译器不再求值参数并将其传给函数，而是直接传递未经求值的参数给宏代码，后者返回一个新的Lisp表达式，在原先宏调用的位置上进行求值。

我将从一个简单而荒唐的例子开始，然后说明你应该怎样把where函数替换成一个where宏。在开始写这个示例宏之前，我需要快速介绍一个新函数：**REVERSE**，它接受一个列表作为参数并返回一个逆序的新列表。因此(`reverse '(1 2 3)`)的求值结果为`(3 2 1)`。现在让我们创建一个宏：

```
(defmacro backwards (expr) (reverse expr))
```

宏与函数的主要词法差异在于你需要用**DEFMACRO**而不是**DEFUN**来定义一个宏。除此之外，宏定义包括名字，就像函数那样，另外宏还有形参列表以及表达式体，这些也与函数一样。但宏却有着完全不同的效果。你可以像下面这样来使用这个宏：

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

它是怎么工作的？REPL开始求值这个backwards表达式时，它认识到backwards是一个宏名。因此它保持表达式`("hello, world" t format)`不被求值，这样正好，因为它不是一个合法的Lisp形式。REPL随后将这个列表传给backwards代码。backwards中的代码再将列表传给**REVERSE**，后者返回列表`(format t "hello, world")`。backwards再将这个值传回给REPL，然后对其求值以顶替最初表达式。

这样backwards宏就相当于定义了一个跟Lisp很像（只是反了一下）的新语言，你随时可以

^① 宏也可以由解释器来运行——但考虑编译的代码时能更容易理解宏的要点。和本章里的所有其他内容一样，相关细节将在后续各章里提及。

通过将一个逆序的Lisp表达式包装在一个对backwards宏的调用里来使用它。而且，在编译了的Lisp程序里，这种新语言的效率就跟正常Lisp一样高，因为所有的宏代码，即用来生成新表达式的代码，都是在编译期运行的。换句话说，编译器将产生完全相同的代码，无论你写成(backwards ("hello, world" t format))还是(format t "hello, world")。

那么这些东西又能对消除where里的代码重复有什么帮助呢？情况是这样的：可以写出一个宏，它在每个特定的where调用里只生成真正需要的代码。最佳方法还是自底向上构建我们的代码。在手工优化的选择器函数里，对于每个实际在最初的where调用中引用的字段来说，都有一个下列形式的表达式：

```
(equal (getf cd field) value)
```

那么让我们来编写一个给定字段名及值并返回表达式的函数。由于表达式本身只是列表，所以函数应写成下面这样。

```
(defun make-comparison-expr (field value) ; wrong
  (list equal (list getf cd field) value))
```

但这里还有一件麻烦事：你知道，当Lisp看到一个诸如field或value这样的简单名字不作为列表的第一个元素出现时，它会假设这是一个变量的名字并去查找它的值。这对于field和value来说是对的，这正是你想要的。但是它也会以同样的方式对待equal、getf以及cd，而这就不是你想要的了。尽管如此，你也知道如何防止Lisp去求值一个形式：在它前面加一个单引号。因此如果你将make-comparison-expr写成下面这样，它将如你所愿：

```
(defun make-comparison-expr (field value)
  (list 'equal (list 'getf 'cd field) value))
```

可以在REPL里测试它。

```
CL-USER> (make-comparison-expr :rating 10)
(EQUAL (GETF CD :RATING) 10)
CL-USER> (make-comparison-expr :title "Give Us a Break")
(EQUAL (GETF CD :TITLE) "Give Us a Break")
```

其实还有更好的办法。当你一般不对表达式求值，但又希望通过一些方法从中提取出确实想求值的少数表达式时，真正需要的是一种书写表达式的方式。当然，确实存在这样一种方法。位于表达式之前的反引号，可以像引号那样阻止表达式被求值。

```
CL-USER> `(1 2 3)
(1 2 3)
CL-USER> '(1 2 3)
(1 2 3)
```

不同的是，在一个反引用表达式里，任何以逗号开始的子表达式都是被求值的。请注意下面第二个表达式中逗号的影响。

```
`(1 2 (+ 1 2))      → (1 2 (+ 1 2))
`(1 2 ,(+ 1 2))    → (1 2 3)
```

有了反引号，就可以像下面这样书写make-comparison-expr了。

```
(defun make-comparison-expr (field value)
  `(equal (getf cd ,field) ,value))
```

现在如果回过头来看那个手工优化的选择器函数，就可以看到其函数体是由每字段/值对应于一个比较表达式组成的，它们全被封装在一个`AND`表达式里。假设现在想让`where`宏的所有实参排成一列传递进来，你将需要一个函数，可以从这样的列表中成对提取元素，并收集在每对参数上调用`make-comparison-expr`的结果。为了实现这个函数，就需要使用一点儿高级Lisp技巧——强有力的`LOOP`宏。

```
(defun make-comparisons-list (fields)
  (loop while fields
    collecting (make-comparison-expr (pop fields) (pop fields))))
```

关于`LOOP`的全面讨论放到了第22章，目前只需了解这个`LOOP`表达式刚好做了你想做的事：当`fields`列表有剩余元素时它会保持循环，一次弹出两个元素，将它们传递给`make-comparison-expr`，然后在循环结束时收集所有返回的结果。`POP`宏所执行的操作与往`*db*`中添加记录时所使用的`PUSH`宏的操作是相反的。

现在需要将`make-comparisons-list`所返回的列表封装在一个`AND`和一个匿名函数里，这可以由`where`宏本身来实现。使用一个反引号来生成一个模板，然后插入`make-comparisons-list`的值，很简单。

```
(defmacro where (&rest clauses)
  `#'(lambda (cd) (and ,@(make-comparisons-list clauses))))
```

这个宏在`make-comparisons-list`调用之前使用了“,”的变体“,`@`”。这个“,`@`”可以将接下来的表达式（必须求值成一个列表）的值嵌入到其外围的列表里。你可以通过下面两个表达式看出“,”和“,`@`”之间的区别：

```
`(and ,(list 1 2 3)) → (AND (1 2 3))
`(and ,@(list 1 2 3)) → (AND 1 2 3)
```

也可以使用“,`@`”在列表的中间插入东西。

```
`(and ,@(list 1 2 3) 4) → (AND 1 2 3 4)
```

`where`宏的另一个重要特性是在实参列表中使用`&rest`。和`&key`一样，`&rest`改变了解析参数的方式。当参数列表里带有`&rest`时，一个函数或宏可以接受任意数量的实参，它们将被收集到一个单一列表中，并成为那个跟在`&rest`后面的名字所对应的变量的值。因此如果像下面这样调用`where`的话。

```
(where :title "Give Us a Break" :ripped t)
```

那么变量`clauses`将包含这个列表。

```
(:title "Give Us a Break" :ripped t)
```

这个列表被传递给了`make-comparisons-list`，其返回一个由比较表达式所组成的列表。可以通过使用函数`MACROEXPAND-1`来精确地看到一个`where`调用将产生出哪些代码。

如果传给`MACROEXPAND-1`一个代表宏调用的形式，它将使用适当的参数来调用宏代码并返



回其展开式。因此可以像这样检查上一个where调用：

```
CL-USER> (macroexpand-1 '(where :title "Give Us a Break" :ripped t))
 #'(LAMBDA (CD)
  (AND (EQUAL (GETF CD :TITLE) "Give Us a Break")
       (EQUAL (GETF CD :RIPPED) T)))
T
```

看起来不错。现在让我们实际试一下。

```
CL-USER> (select (where :title "Give Us a Break" :ripped t))
((:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T))
```

它成功了。并且事实上，新的where宏加上它的两个助手函数还比老的where函数少了一行代码。并且新的代码更加通用，再也不需要理会我们CD记录中的特定字段了。

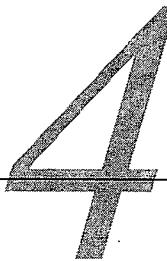
3.9 总结

现在，有趣的事情发生了。你不但去除了重复，而且还使得代码更有效且更通用了。这通常就是正确选用宏所达到的效果。这件事合乎逻辑，因为宏只不过是另一种创建抽象的手法——词法层面的抽象，以及按照定义通过更简明地表达底层一般性的方式所得到的抽象。现在这个微型数据库的代码中只有make-cd、prompt-for-cd以及add-cd函数是特定于CD及其字段的。事实上，新的where宏可以用在任何基于plist的数据库上。

尽管如此，它距离一个完整的数据库仍很遥远。你可能会想到还有大量需要增加的特性，包括支持多表或是更复杂的查询。第27章将建立一个具备这些特性的MP3数据库。

本章的要点在于快速介绍少量Lisp特性，展示如何用它们编写出比“hello, world”更有趣一点儿的代码。在下一章里，我们将对Lisp进行更加系统的概述。

语法和语义



在一阵旋风似的介绍之后，我们将坐下来用几章的篇幅对已经用到的那些特性来一次更加系统化的介绍。我将从对Lisp的基本语法和语义元素的概述开始，这意味着我必须先回答下面这个问题……

4.1 括号里都可以有什么

Lisp的语法和源自Algol的语言在语法上有很多不同。两者特征最明显的区别在于前者大量使用了括号和前缀表示法。说来也怪，大量的追随者都喜欢这样的语法。Lisp的反对者们总是将这种语法描述成“奇怪的”和“讨厌的”，他们说Lisp就是“大量不合理的多余括号”(Lots of Irritating Superfluous Parentheses)的简称；而Lisp的追随者则认为，Lisp的语法是它的最大优势。为什么两个团体之间会有如此对立的见解呢？

本章不可能完整地描述Lisp的语法，因为我还没有彻底地解释Lisp的宏，但我可以从一些宝贵的历史经验来说明保持开放的思想是值得的：John McCarthy首次发明Lisp时，曾想过实现一种类Algol的语法，他称之为M-表达式。尽管如此，他却从未实现这一点。他在自己的文章*History of Lisp*^①中对此加以解释。

这个精确定义M-表达式以及将其编译或至少转译成S-表达式的工程，既没有完成也没有明确放弃，它只不过是被无限期地推迟了。（相比S-表达式）更加偏爱类FORTRAN或类Algol表示法的新一代程序员也许会最终实现它。

换句话说，过去45年以来，实际使用Lisp的人们已经喜欢上了这种语法，并且发现它能使该语言变得更为强大。接下来的几章将告诉你为什么会这样。

4.2 打开黑箱

在介绍Lisp的语法和语义之前，值得花一点时间来看看它们的定义及其与许多其他语言的语法所存在的差异。

^① <http://www-formal.stanford.edu/jmc/history/lisp/node3.html>。

在大多数编程语言里，语言的处理器（无论是解释器还是编译器）的操作方式都类似于黑箱作业：一系列表示程序文本的字符被送进黑箱，然后它（取决于是解释器还是编译器）要么执行预想行为，要么产生一个编译版本的程序并在运行时执行这些行为。

当然，在黑箱的内部，语言的处理器通常划分成子系统，各自负责一部分将程序文本转换成具体行为或目标代码的任务。一个典型任务划分思路是将处理器分成三个阶段，每个阶段为下一个阶段提供内容：一个词法分析器将字符流分拆成语元并将其送进一个解析器，解析器再根据该语言的语法在程序中构建一个表达式的树形表示。这棵树被称为抽象语法树，它随即被送进一个求值器，求值器要么直接解释它，要么将其编译成某种其他语言（比如机器码）。由于语言处理器是一种黑箱，所以处理器所使用的包括语元和抽象语法树在内的数据结构，只对语言的实现者有用。

而在Common Lisp中，分工则有点不同，无论从实现者的角度还是从语言定义方式的角度上来说都是这样。与一个从文本到程序行为一步到位的单一黑箱有所不同的是，Common Lisp定义了两个黑箱，一个将文本转化成Lisp对象，而另一个则用这些对象来实现语言的语义。前一个箱子称为读取器，后一个称为求值器。^①

每个黑箱都定义了一个语法层面。读取器定义了字符串如何被转换为我们称之为S-表达式^②的Lisp对象。由于S-表达式语法可适用于由任意对象及其他列表所组成的列表，因此S-表达式可用来表达任意树形表达式，这跟由非Lisp语言的语法解析器所生成的抽象语法树非常相似。

求值器随后定义了一种构建在S-表达式之上的Lisp形式（form）的语法。并非所有的S-表达式都是合法的Lisp形式，更不用说所有字符序列都是合法的S-表达式了。举个例子，(foo 1 2)和("foo" 1 2)都是S-表达式，但只有前者才是一个Lisp形式，因为一个以字符串开始的列表对于Lisp形式来说是没有意义的。

这样的黑箱划分方法带来了一系列后果。其中之一是可将S-表达式（正如第3章那样）用作一种可暴露的数据格式来表达源代码之外的数据，用READ来读取它再用PRINT来打印它。^③另一个后果则在于，由于语言的语义是用对象树而非字符串定义而成的，因此很容易使用语言本身而非文本形式来生成代码。完全从手工生成代码的好处很有限——构造列表和构造字符串的工作量大致相同。尽管如此，真正的优势在于可通过处理现有数据来生成代码。这就是Lisp宏的本意，我将在后续章节详加论述。目前我将集中在Common Lisp所定义的两个层面上：读取器所理解的S-表达式语法以及求值器所理解的Lisp表达式语法。

^① 和任何语言的实现者一样，Lisp实现者们有许多方式可以实现一个求值器，从一个解释那些直接送进求值器的对象的“纯”解释器，到一个将对象转化成机器码并执行的编译器。在这两者之间，还有些实现将输入编译成类似虚拟机字节码这样的中间形式，然后解释执行字节码。近年来，多数Common Lisp实现都使用某种形式的编译，甚至在运行期求值代码的时候也是这样。

^② 术语“S-表达式”有时代表文本表示，而有时则代表从文本表示中读取到的对象。通常从上下文中可以清楚地判断它的含义，或者怎样理解都无所谓。

^③ 并非所有的Lisp对象都可以被写成一种可以被读回来的形式，但任何你可以用READ读取的东西都可以被PRINT打印成可读的形式。

4.3 S-表达式

S-表达式的基本元素是列表 (list) 和原子 (atom)。列表由括号所包围，并可包含任何数量的由空格所分隔的元素。原子是所有其他内容。^①列表元素本身也可以是S-表达式（换句话说，也就是原子或嵌套的列表）。注释从技术角度来讲不是S-表达式，它们以分号开始，直到一行的结尾，本质上将被当作空白来处理。

这就差不多了。列表在句法上十分简单，你需要知道的句法规则只有那些用来形成不同类型原子的规则了。在本节里我将描述几种常用原子类型的规则，这些原子包括：数字、字符串和名字。之后我将说明由这些元素所组成的S-表达式是如何作为Lisp形式求值的。

数字的表示方法很简单。任何数位的序列将被读取为一个数字，它们可能有一个前缀标识 (+ 或 -)，还可能会有一个十进制点 (.) 或者斜杠 (/)，或是以一个指数标记结尾。例如：

```

123      ; 整数一百二十三
3/7      ; 比值七分之三
1.0      ; 默认精度的浮点数一
1.0e0    ; 同一个浮点数的另一种写法
1.0d0    ; 双精度的浮点数一
1.0e-4   ; 等价于万分之一的浮点数
+42     ; 整数四十二
-42     ; 整数负四十二
-1/4    ; 比值负四分之一
-2/8    ; 负四分之一的另一种写法
246/2   ; 整数一百二十三的另一种写法

```

这些不同的形式代表着不同类型的数字：整数、比值和浮点数。Lisp也支持复数，但它们有自己的表示法，第10章将予以介绍。

从这些示例可见，你可以用多种方式来表示同一个数字。但无论怎样书写，所有的有理数（整数和比值）在内部都被表示成“简化”形式。换句话说，表示 $-2/8$ 或 $246/2$ 的对象跟表示 $-1/4$ 或123的对象并没有什么不同。与此相似，1.0和 $1.0e0$ 也只是同一个数字的不同写法而已。但另一方面，1.0、 $1.0d0$ 和1则可能会代表不同的对象，因为不同精度的浮点数和整数都是不同类型的对象。我将在第10章详细讨论不同类型数字的特征。

正如在前面章节所看到的那样，字符串是由双引号所包围着的。在字符串中，一个反斜杠会转义接下来的任意字符，使其被包含在字符串里。两个在字符串中必须被转义的字符是双引号和反斜杠本身。所有其他的字符无需转义即可被包含在一个字符串里，无论它们在字符串之外有何含义。下面是一些字符串的示例：

```

"foo"      ; 含有f、o和o的字符串
"fo\o"     ; 同一个字符串
"fo\\o"    ; 含有f、o、\和o的字符串
"fo\"o"   ; 含有f、o、"和o的字符串

```

Lisp中所使用的名字，诸如FORMAT、hello-world和*db*均由称为符号的对象所表示。读

^① 空列表即 ()，也可写成NIL，既是原子也是列表。

取器对于一个给定名的用途毫不知情——无论其究竟用作变量名、函数名还是其他什么东西。读取器只是读取字符序列并构造出此名所代表的对象。^①几乎任何字符都可以出现在一个名字里，不过空白字符不可以，因为列表的元素是用空格来分隔的。数位也可以出现在名字里，只要整个名字不被解释成一个数字。类似地，名字可以包含句点，但读取器无法读取一个只由句点组成的名字。有十个字符被用于其他句法目的而不能出现在名字里，它们是：开括号和闭括号、双引号和单引号、反引号、逗号、冒号、分号、反斜杠以及竖线。而就算是这些字符，如果你愿意的话，它们也可以成为名字的一部分，只需将它们用反斜杠进行转义，或是将含有需要转义的字符名字用竖线包起来。

这种读取器将名字转化成符号对象的方式有两个重要特征，一个是它如何处理名字中的字母大小写，另一个是它如何确保相同的名字总被读取成相同的符号。当读取名字时，读取器将所有名字中未转义的字符都转化成它们等价的大写形式。这样，读取器将把foo、Foo和FOO都读成同一个符号FOO。但\f\o\o和\f\oo\l将都被读成foo，这是和符号FOO不同的另一个对象。这就是为什么在REPL中定义一个函数时，它的名字会被打印成大写形式的原因。近年来，标准的编码风格是将代码全部写成小写形式，然后让读取器将名字转化成大写。^②

为了确保同一个文本名字总是被读取成相同的符号，读取器保留这个符号——在已读取名字并将其全部转化成大写形式以后，读取器在一个称为包（package）的表里查询带有相同名字的已有符号。如果无法找到，则将创建一个新符号并添加到表里。否则就将返回已在表中的那个符号。这样，无论在任何地方，同样的名字出现在任何S-表达式里，都会用同一个对象去表示它。^③

因为在Lisp中名字可以包含比源自Algol的语言更多的字符，故而命名约定在Lisp中也相应地有所不同，诸如可以使用像hello-world这类带有连字符的名字。另一个重要约定是全局变量名字在开始和结尾处带有“*”。类似地，常量名都以“+”开始和结尾。而某些程序员则将特别底层的函数名前加%甚至%%。语言标准所定义的名字只使用字母表字符（A-Z）外加*、+、-、/、1、2、<、=、>以及&。

只用列表、数字、字符串和符号就可以描述很大一部分的Lisp程序了。其他规则描述了字面向量、单个字符和数组的标识，我将在第10章和第11章里谈及相关的数据类型时再讲解它们。目前的关键是要理解怎样用数字、字符串和由符号借助括号所组成的列表来构建S-表达式，从而表示任意的树状对象。下面是一些简单的例子。

```
x          ; 符号x
()         ; 空列表
(1 2 3)    ; 三个数字所组成的列表
("foo" "bar") ; 两个字符串所组成的列表
(x y z)    ; 三个符号所组成的列表
```

4

^①事实上，正如你后面将要看到的，名字从本质上是一种独立的概念。根据不同的上下文，你可以使用相同的名字来同时引用一个变量或者函数，更不用说其他几种可能性了。

^②事实上，读取器的大小写转化行为是可以定制的，但是要想理解何时以及怎样改变它，则需要在相关的名字、符号和其他程序元素中，相对我已经涉及的内容做更加深入的讨论。

^③我将在第21章里讨论符号和包之间关系的更多细节。

```
(x 1 "foo") ; 由一个符号、一个数字和一个字符串所组成的列表
(+ (* 2 3) 4) ; 由一个符号、一个列表和一个数字所组成的列表
```

下面这种四元素列表就稍显复杂了，它含有两个符号、空列表以及另一个列表——其本身又含有两个符号和一个字符串：

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

4.4 作为 Lisp 形式的 S-表达式

在读取器把大量文本转化为S-表达式以后，这些S-表达式随后可以作为Lisp形式被求值。或者只有它们中的一些可以——并不是每个读取器可读的S-表达式都有必要作为Lisp形式来求值的，Common Lisp的求值规则定义了第二层的语法来检测哪种S-表达式可看作Lisp形式^①。这一层面的句法规则相当简单。任何原子（非列表或空列表）都是一个合法的Lisp形式，正如任何以符号为首先元素的列表那样。^②

当然，Lisp形式的有趣之处不在于其语法，而在于它们被求值的方式。为了便于讨论，你可以将求值器想象成一个函数，它接受一个语法良好定义的Lisp形式作为参数并返回一个值，我们称之为这个形式的值。当然，当求值器是一个编译器时，情况会更加简化一些——在那种情况下，求值器被给定一个表达式，然后生成在其运行时可以计算出相应值的代码。但是这种简化可以让我们从不同类型的Lisp形式如何被这个假想的函数求值的角度来描述Common Lisp的语义。

作为最简单的Lisp形式，原子可以被分成两个类别：符号和所有其他内容。符号在作为Lisp形式被求值时会被视为一个变量名，并且会被求值为该变量的当前值。^③第6章将讨论变量是如何得到这个值的。你也需要注意，某些特定的“变量”其实是编程领域的早期产物“常值变量”(constant variable)。例如，符号PI命名了一个常值变量，其值是最接近数学常量π的浮点数。

所有其他的原子，包括你已经见过的数字和字符串，都是自求值对象。这意味着当这样的表达式被传递给假想函数时，它会简单地直接返回自身。第2章在REPL里键入的10和"hello, world"实际上就是自求值对象的例子了。

把符号变成自求值对象也是可能的——它们所命名的变量可以被赋值成符号本身的值。两个以这种方式定义的常量是T和NIL，即所谓的真值和假值。我将在4.8节里讨论它们作为布尔值的角色。

另一类自求值符号是关键字符符号——以名字冒号开始的符号。当读取器保留这样一个名字时，它会自动定义一个以此命名的常值变量并以该符号作为其值。

^① 当然如同其他语言那样，其他层面的纠错也存在于Lisp中。例如，从读取(foo 1 2)得到的S-表达式在句法上是良好定义的，但是只有当foo是一个函数或宏的名字时，它才可以被求值。

^② 另一种很少用到的Lisp形式的类型是那种第一个元素是lambda表达式的列表。我将在第5章里讨论这类形式。

^③ 存在另一种可能性——可以定义出符号宏(symbol macro)，它可被稍有不同地求值。我们无需理会它们。

当我们开始考虑列表的求值方式时，事情变得更加有趣了。所有合法的列表形式均以一个符号开始，但是有三种类型的列表形式，它们会以三种相当不同的方式进行求值。为了确定一个给定的列表是哪种形式，求值器必须检测列表开始处的那个符号是一个函数、宏还是特殊操作符的名字。如果该符号尚未定义，比如说当你正在编译一段含有对尚未定义函数的引用的代码时，它会被假设成一个函数的名字。^①我将把这三种类型的形式称为函数调用形式（function call form）、宏形式（macro form）和特殊形式（special form）。

4.5 函数调用

函数调用形式的求值规则很简单，对以Lisp形式存在的列表其余元素进行求值并将结果传递到命名函数中。这一规则明显的有着一些附加的句法限制在函数调用形式上：除第一个以外，所有的列表元素它们自身必须是一个形态良好的Lisp形式。换句话说，函数调用形式的基本语法应如下所示，其中每个参数本身也是一个Lisp形式：

```
(function-name argument*)
```

这样下面这个表达式在求值时将首先求值1，再求值2，然后将得到的值传给+函数，再返回3：

```
(+ 1 2)
```

像下面这样更复杂的表达式也采用相似的求值方法，不过在求值参数(+ 1 2)和(- 3 4)时需要先对它们的参数求值，然后再对它们应用相应的函数：

```
(* (+ 1 2) (- 3 4))
```

最后，值3和-1被传递到*函数里，从而得到-3。

正如这些例子所显示的这样，许多在其他语言中需用特殊语法来处理的事务在Lisp中都可用函数来处理。Lisp的这种设计对于保持其语法正则化很有帮助。

4.6 特殊操作符

然而，并非所有的操作都可定义成函数。由于一个函数的所有参数在函数被调用之前都将被求值，因此无法写出一个类似第3章里用到的IF操作符那样的函数。为了说明这点，可以假设有下面这种形式：

```
(if x (format t "yes") (format t "no"))
```

如果IF是一个函数，那么求值器将从左到右依次对其参数表达式求值。符号x将被作为产生某个值的变量来求值，然后(format t "yes")将被当成一个函数调用来求值，在向标准输出打

^① 在Common Lisp中一个符号可以同时为操作符（函数、宏或特殊操作符）和变量命名。这是Common Lisp和Scheme的主要区别之一。这一区别有时被描述成Common Lisp是一种Lisp-2而Scheme则是Lisp-1——一个Lisp-2有两个命名空间，一个用于操作符而另一个用于变量，但一个Lisp-1仅使用单一的命名空间。两种选择都有其各自的优点，而拥护者们总是在无休止地争论哪种更好。

印“yes”以后得到NIL。接下来(format t "no")将被求值，打印出“no”同时也得到NIL。只有当所有三个表达式都被求值以后，它们的结果值才被传递给IF，而这时已经无法控制两个FORMAT表达式中的哪一个会被求值了。

为了解决这个问题，Common Lisp定义了一些特殊操作符，IF就是其中之一，它们可以做到函数无法做到的事情。它们总共有25个，但只有很少一部分直接用于日常编程。^①

当列表的第一个元素是一个由特殊操作符所命名的符号时，表达式的其余部分将按照该操作符的规则进行求值。

IF的规则相当简单：求值第一个表达式。如果得到非NIL，那么求值下一个表达式并返回它的值。否则，返回第三个表达式的求值，或者如果第三个表达式被省略的话，返回NIL。换句话说，一个IF表达式的基本形式是像下面这样：

```
(if test-form then-form [ else-form ])
```

其中test-form将总是被求值，然后要么是then-form要么是else-form。

一个更简单的特殊操作符是QUOTE，它接受一个单一表达式作为其“参数”并简单地返回它，不经求值。例如，下面的表达式求值得到列表(+ 1 2)，而不是值3：

```
(quote (+ 1 2))
```

这个列表没有什么特别的，你可以像用LIST函数所创建的任何列表那样处理它。^②

QUOTE被用得相当普遍，以至于读取器中内置了一个它的特别语法形式。除了能像下面这样写之外：

```
(quote (+ 1 2))
```

也可以这样写：

```
'(+ 1 2)
```

该语法是读取器所理解的S-表达式语法的小扩展。从求值器的观点来看，这两个表达式看起来是一样的：一个首元素为符号QUOTE并且次元素是列表(+ 1 2)的列表。^③

一般来说，特殊操作符所实现的语言特性需要求值器作出某些特殊处理。例如，有些的操作符修改了其他形式的求值环境。其中之一是LET，也是我将在第6章详细讨论的特殊操作符，它用来创建新的变量绑定。下面的形式求值得到10，因为在第二个x的求值环境中，它是由LET赋值为10的变量名：

```
(let ((x 10)) x)
```

^① 其他操作符提供了有用但有时晦涩难懂的特性。我将在它们所支持的特性里讨论它们。

^② 确实有一点区别——像引用列表这样的字面对象，也包括双引号里的字符串、字面数组和向量（以后你将看到它的语法），一定不能被修改。一般而言，任何你打算修改的列表都应该用LIST来创建。

^③ 该语法是读取宏(reader macro)的一个例子。读取宏可以修改读取器用来将文本转化成Lisp对象的语法。事实上，定义你自己的读取宏也是有可能的，但这是该语言的一种很少被用到的机制。多数Lisp程序员提到该语言的语法扩展时，他们指的是正规的宏，我将在稍后讨论它们。

4.7 宏

虽然特殊操作符以超越了函数调用所能表达的方式扩展了Common Lisp语法，但特殊操作符的数量在语言标准中是固定的。然而宏却能提供给语言用户一种语法扩展方式。如同在第3章里看到的那样，宏是一个以S-表达式为其参数的函数，并返回一个Lisp形式，然后对其求值并用该值取代宏形式。宏形式的求值过程包括两个阶段：首先，宏形式的元素不经求值即被传递到宏函数里；其次，由宏函数所返回的形式（称其为展开式（expansion））按照正常的求值规则进行求值。

重点在于要清醒地认识到一个宏形式求值的两个阶段。当你在REPL中输入表达式时很容易忘记这一点，因为两个阶段相继发生并且后一阶段的值被立即返回了。但是当Lisp代码被编译时，这两个阶段所发生时间却是完全不同的，因此关键在于对于何时发生什么要保持清醒。例如，当使用函数**COMPILE-FILE**来编译整个源代码文件时，文件中所有宏形式将被递归展开，直到代码中只含有函数调用形式和特殊形式。这些无宏的代码随后被编译成一个FASL文件——**LOAD**函数知道如何去加载它。但编译后的代码直到文件被加载时才会被执行。因为宏在编译期会生成其展开式，它们可以用相对大量的工作来生成其展开式，而无需在文件被加载时或是当文件中定义的函数被调用时再付出额外的代价。

由于求值器在将宏形式传递给宏函数之前并不对它们求值，因此它们不需要是格式良好的Lisp形式。每个宏都为其宏形式中的符号表达式指定了一种含义，用以指明宏将如何使用它们生成展开式。换句话说，每个宏都定义了它们自己的局部语法。例如，第3章的**backwards**宏就定义了一种语法，合法的**backwards**形式列表必须与合法的Lisp形式列表反序。

我会在本书中经常地提到宏。眼下最为重要的是认识到宏，虽然跟函数调用在句法上相似，但却有着相当不同的用途，并提供了一种嵌入编译器的钩子。^①

^① 对于缺少或没有Lisp宏使用经验的人们，以及那些甚至被C的预处理器所荼毒过的人们来说，当他们意识到宏调用跟正常函数调用一样时可能会紧张。但这在实践中却并不是一个问题，原因如下。一方面是因为宏形式通常在格式上与函数调用不同。例如，你会写成这样：

```
(dolist (x foo)
```

```
  (print x))
```

而不是这样：

```
(dolist (x foo) (print x))
```

也不会是这样：

```
(dolist (x foo)
```

```
  (print x))
```

后面两种形式使**DOLIST**显得像是一个函数。一个好的Lisp开发环境会自动正确地格式化宏调用，甚至对于用户定义的宏也是如此。

就算一个**DOLIST**形式被写在了单行里，也有几条线索可以说明它是一个宏。其一，表达式(x foo)只有在x是一个函数或宏的名字时本身才有意义。将这一现象和随后作为变量出现的x联系起来，就会很容易发现**DOLIST**是一个正在创建名为x的变量绑定的宏。命名约定也有帮助——通常作为宏的循环结构都带有一个以do开始的名字。

4.8 真、假和等价

最后两个需要了解的基本知识是Common Lisp对于真和假的表示法以及两个Lisp对象“等价”的含义。真和假的含义在这里是直截了当的：符号`NIL`是唯一的假值，其他所有的都是真值。符号`T`是标准的真值，可用于需要返回一个非`NIL`值却又没有其他值可用的情况。关于`NIL`，唯一麻烦的一点是，它是唯一一个既是原子又是列表的对象：除了用来表示假以外，它还用来表示空列表。^①这种`NIL`和空列表的等价性被内置在读取器之中：如果读取器看到了`()`，它将作为符号`NIL`读取它。它们是完全可以互换的。并且如同我前面提到的那样，因为`NIL`是一个以符号`NIL`作为其值的常值变量名，所以表达式`nil`、`()`、`'nil`以及`'()`求值结果是相同的——未引用形式将被看成是对值为符号`NIL`的常值变量的引用来进行求值，而在引用形式中，`QUOTE`特殊操作符将会直接求解出符号`NIL`。基于同样的理由，`t`和`'t`的求值结果也完全相同：符号`T`。

使用诸如“完全相同”这样的术语理所当然会引申出两个值“等价”的这个问题上。在后面的章节里将会看到，Common Lisp提供了许多特定于类型的等价谓词：`=`用来比较数字，`CHAR=`用来比较字符，依此类推。本节将讨论四个“通用”等价谓词——这些函数可以被传入任何两个Lisp对象，然后当它们等价时返回真，否则返回假。按照介绍的顺序，它们是`EQ`、`EQL`、`EQUAL`和`EQUALP`。

`EQ`用来测试“对象标识”，只有当两个对象相同时才是`EQ`等价的。不幸的是，数字和字符的对象标识取决于这些数据类型在特定Lisp平台上实现的方式。带有相同值的两个数字或字符可能会被`EQ`认为是等价的也可能会是不等价的。语言实现者有足够的空间将表达式`(eq x x)`合法地求值为真或假，这种情况更多会发生在`x`恰好为数字或字符时。

因此不该将`EQ`用于比较可能是数字或字符的值上。在个别实现的特定值上，它可能会以一种可预测的方式工作，但如果切换了语言实现，则它将不保证以相同的方式工作。切换实现可能意味着只是简单地把实现升级到一个新版本——而如果Lisp实现者改变了表示数字或字符的方式，那么`EQ`的行为也将很有可能发生改变。

因此，Common Lisp定义了`EQL`来获得与`EQ`相似的行为，除此之外，它也可以保证当相同类型的两个对象表示相同的数字或字符值时，它们是等价的。因此，`(eql 1 1)`能被确保是真。而`(eql 1 1.0)`则被确保是假，因为整数值1和浮点数1.0是不同类型的对象。

关于何时使用`EQ`以及何时使用`EQL`，这里有两种观点：“凡有可能就用`EQ`”阵营认为，当知道不存在比较数字或字符时就应该使用`EQ`，他们认为(a)这是一种说明你不在比较数字或字符的方式，以及(b)因为`EQ`不需要检查它的参数是否为数字或字符，所以它将会稍微更有效率。

但“总是使用`EQL`”阵营则认为永远不该使用`EQ`，因为(a)每次有人（包括你在内）阅读你的代码时，看到了一个`EQ`，就得停下来检查它是否被正确使用了（换句话说，它永远不该被用来比

^① 使用空列表作为假是Lisp作为列表处理语言所留下的遗产，就好比是C语言使用整数0作为假是其作为字节处理语言所留下的遗产。但并非所有的Lisp都以相同的方式处理布尔值。Common Lisp和Scheme的许多细微差异中的另一个就是Scheme使用一个单独的假值`#f`，`#f`无论跟符号`nil`还是空列表都是不同的值，并且即便是`nil`和空列表，两者也是不同的。



较数字或字符)，这样就会丢失潜在获得的代码清晰性，以及(b)EQ和EQL之间的效率差异相比于实际的性能瓶颈来说微不足道。

本书中的代码是以“总是使用EQL”风格写成的。^①

另外两个等价谓词EQUAL和EQUALP更为通用，因为它们可以操作在所有类型的对象上，但又不像EQ或EQL那样基础。它们每个都定义了相比EQL稍微宽松一些的等价性，允许认为不同的对象是等价的。除了它们曾经被过去的Lisp程序员认为是有用的之外，由这些函数所实现的特殊含义的等价性没有什么特别的。如果这些谓词不能满足需要，也可以自己定义谓词函数，以自己所需方式来比较不同类型对象。

EQUAL相比EQL的宽松之处在于，它将在递归上具有相同结构和内容的列表视为等价。EQUAL也认为含有相同字符的字符串是等价的，它对于位向量和路径名也定义了比EQL更加宽松的等价性，我将在未来的章节里讨论这两个数据类型。对于所有其他类型，它回退到EQL的水平上。

EQUALP甚至更加宽松之外，它和EQUAL是相似的。它在考察两个含有相同字符的字符串的等价性时忽略了大小写的区别。它还认为如果两个字符只在大小写上有区别，那么它们就是等价的。只要数字表示相同数学意义上的值，它们在EQUALP下面就是等价的。因此，(equalp 1 1.0)是真的。由EQUALP等价的元素所组成的列表也是EQUALP等价的。同样地，带有EQUALP元素的数组也是EQUALP等价的。和EQUAL一样，还有一些我尚未涉及的其他数据类型，EQUALP可认为两个对象是等价的，但EQL或EQUAL则不会。对于所有的其他数据类型，EQUALP回退到EQL的水平上。

4.9 格式化 Lisp 代码

严格说起来，代码格式化既不是句法层面也不是语法层面上的事情，好的格式化对于阅读和编写流利而又地道的代码而言非常重要。格式化Lisp代码的关键在于正确缩进它。这一缩进应当反映出代码结构，这样就不需要通过括号来查看代码究竟写到哪儿了。一般而言，每一个新的嵌套层次都需要多缩进一点儿，并且如果折行是必需的，位于同一个嵌套层次的项应当按行对齐。这样，一个需要跨越多行的函数调用可能会被写成这样：

```
(some-function arg-with-a-long-name
               another-arg-with-an-even-longer-name)
```

那些实现控制结构的宏和特殊形式在缩进上稍有不同：“主体”元素相对于整个形式的开括号缩进两个空格。就像这样：

```
(defun print-list (list)
  (dolist (i list)
    (format t "item: ~a~%" i)))
```

尽管如此，但也不要太担心这些规则，因为一个像SLIME这样的优秀Lisp环境将会帮你做

^① 甚至是语言标准，在关于EQ或EQL哪一个应当被使用方面也有一点歧义，对象标识（object identity）是由EQ定义的，但是标准在谈论对象时定义了术语“相同”来表达EQL，除非明确提到了另外的谓词。因此，如果你想要在技术上100%正确的话，你可以说(- 3 2)和(- 4 3)求值到“相同”(same)的对象而不是“同样”(identical)的对象。不得不承认，这个问题有些无聊。

到这点。事实上，Lisp正则语法的优势之一就在于，它可以让诸如编辑器这样的软件相对容易地知道应当如何缩进。由于缩进的本意是反映代码的结构，而结构是由括号来标记的，因此很容易让编辑器帮你缩进代码。

在SLIME中，在每行开始处按下Tab键将导致该行被适当地缩进，或者也可以通过将光标放置在一个开括号上并键入C-M-q来重新缩进整个表达式。或者还可以在函数内部的任何位置通过键入C-c M-q来重新缩进整个函数体。

的确，有经验的Lisp程序员们倾向于依赖编辑器来自动处理缩进，这样不但可以确保代码美观，还可以检测笔误：一旦熟悉了代码该如何缩进，那么一个错误放置的括号就将立即由于编辑器所给出的奇怪缩进而被发现。例如，假设要编写一个如下所示的函数：

```
(defun foo ()
  (if (test)
      (do-one-thing)
      (do-another-thing)))
```

而你不小心忘记了test后面的闭括号。如果不数括号，那么很可能就会在DEFUN形式的结尾处添加一个额外的括号，从而得到下面的代码：

```
(defun foo ()
  (if (test
        (do-one-thing)
        (do-another-thing))))
```

尽管如此，如果一直都在每行的开始处按Tab来缩进的话，就不会得到这样的代码。相反，将得到如下的代码：

```
(defun foo ()
  (if (test
        (do-one-thing)
        (do-another-thing))))
```

看到then和else子句被缩进到了条件语句的位置，而不是仅仅相对于IF稍微缩进了一点，你立即就能看出有错误发生。

另一个重要的格式化规则是，闭括号总是位于与它们所闭合的列表最后一个元素相同的行。也就是说，不要写成这样：

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)
  )
)
```

而一定要写成这样：

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

结尾处的)))可能看起来令人生畏，但是一旦代码缩进正确，那么括号的意义就不存在了，没有

必要通过将它们分散在多行来加以突出。

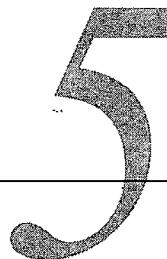
最后，注释应该根据其适用范围被前置一到四个分号，如同下面所说明的：

;;;; 四个分号用于文件头注释。

;;; 带有三个分号的注释将通常作为段落注释应用到接下来的一大段代码上。

```
(defun foo (x)
  (dotimes (i x)
    ;; 两个分号说明该注释应用于接下来的代码上。
    ;; 注意，该注释与其所应用的代码具有相同的缩进。
    (some-function-call)
    (another i)           ; 本注释仅用于此行
    (and-another)         ; 这个也是一样
    (baz)))
```

现在可以开始了解Lisp的主要程序构造的更多细节了：函数、变量和宏。下一章先来看看函数。



有了语法和语义规则以后，所有Lisp程序的三个最基本组成部分就是函数、变量和宏。在第3章里构建数据库时，这三个组件已经全部用到了，但是我没有详细提及它们是如何工作的，如何更好使用它们。接下来的几章将专门讲解这三个主题，先从函数开始。就跟其他语言里一样，函数提供了用于抽象和功能化的基本方法。

Lisp本身是由大量函数组成的。其语言标准中有超过四分之三的名字用于定义函数。所有内置的数据类型纯粹是用操作它们的函数来定义的。甚至连Lisp强大的对象系统也是构建在函数的概念性扩展——广义函数（generic function）之上的，第16章将会介绍它们。

而且，尽管宏对于Lisp风格有着重要的作用，但最终所有实际的功能还是由函数来提供的。宏运行在编译期，因此它们生成的代码，即当所有宏被展开后将实际构成程序的那些代码，将完全由对函数和特殊操作符的调用所构成。更不用说，宏本身也是函数了——尽管这种函数是用来生成代码，而不是用来完成实际的程序操作的。^①

5.1 定义新函数

函数一般使用DEFUN宏来定义。DEFUN的基本结构看起来像这样：

```
(defun name (parameter*)
  "Optional documentation string."
  body-form*)
```

任何符号都可用作函数名。^②通常函数名仅包含字典字符和连字符，但是在特定的命名约定里，其他字符也允许使用。例如，将值的一种类型转换成另一种的函数有时会在名字中使用→，一个将字符串转换成微件（widget）的函数可能叫做string->widget。最重要的一个命名约定

① 尽管函数对于Common Lisp很重要，但将其说成是函数型语言却并不是非常合适。尽管一些Common Lisp特性（例如其列表管理函数）被设计用于函数型编程风格，并且Lisp在函数型编程史上也有其突出的地位——McCarthy引进了许多被认为对函数型编程非常重要的思想，但Common Lisp其本意是被用于支持多种不同的编程风格的。在Lisp家族里，Scheme最接近“纯”函数型语言，而就算它也有一些特性，但相比于诸如Haskell和ML这类语言，其在纯粹程度上也不够格。

② 严格来讲是几乎任何符号。如果你使用了由语言标准所定义的名字作为你自己的函数的名字，其后果尚未可知。尽管如此，你在第21章会看到，Lisp的包系统允许你在不同的命名空间里创建名字，因此这实际上不是问题。

是在第2章里提到的那个，即要用连字符而不是下划线或内部大写来构造复合名称。因此，`frob-widget`比`frob_widget`或`frobWidget`更具有Lisp风格。一个函数的形参列表定义了一些变量，将用来保存函数在调用时所传递的实参。^①如果函数不带有实参，则该列表就是空的，写成`()`。不同种类的形参分别负责处理必要的、可选的、多重的以及关键字实参。我将在下一节里讨论相关细节。

如果一个字符串紧跟在形参列表之后，那么它应该是一个用来描述函数用途的文档字符串。当定义函数时，该文档字符串将被关联到函数名上，并且以后可以通过`DOCUMENTATION`函数来获取。^②

最后，一个`DEFUN`的主体可由任意数量的Lisp表达式所构成。它们将在函数被调用时依次求值，而最后一个表达式的值将被作为整个函数的值返回。另外`RETURN-FROM`特殊操作符可用于从函数的任何位置立即返回，我很快就会谈到它。

第2章里所写的`hello-world`函数，形式如下：

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

现在可以分析一下该程序的各个部分了。它的名字是`hello-world`，形参列表为空，因此不接受任何参数，它没有文档字符串，并且它的函数体由一个表达式所构成：

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

下面是一个更复杂一些的函数：

```
(defun verbose-sum (x y)
  "Sum any two numbers after printing a message."
  (format t "Summing ~d and ~d.~%" x y)
  (+ x y))
```

这个函数称为`verbose-sum`，它接受的两个实参分别与形参`x`和`y`一一对应并且带有一个文档字符串，以及一个由两个表达式所组成的主体。由“`+`”调用所返回的值将成为`verbose-sum`的返回值。

5.2 函数形参列表

关于函数名或文档字符串就没有更多可说的了，而本书其余部分将用很多篇幅来描述所有可在在一个函数体里做的事情，因此就只需讨论形参列表了。

很明显，一个形参列表的基本用途是为了声明一些变量，用来接收传递给函数的实参。当形参列表是一个由变量名所组成的简单列表时，如同在`verbose-sum`里那样，这些形参被称为必

^① 因为Lisp的函数表示法与lambda演算之间的历史关系，形参列表有时也称为lambda列表。

^② 例如：

```
(documentation 'foo 'function)
```

将返回函数`foo`的文档字符串。尽管如此，请注意文档字符串是用来给人看的，而没有任何程序意义上的用途。

一个Lisp实现不要求保存它们，实际上可在任何时候丢弃它们，因此可移植的程序不应该依赖于它们的存在。在某些实现里，一个由实现所定义的全局变量需要在使用文档字符串之前被设置成指定的值。

要形参。当函数被调用时，必须为它的每一个必要形参都提供一个实参。每一个形参被绑定到对应的实参上。如果一个函数以过少或过多的实参来调用的话，Lisp就会报错。

但是，Common Lisp的形参列表也给了你更灵活的方式将函数调用实参映射到函数形参。除了必要形参以外，一个函数还可以有可选形参，或者也可以用单一形参绑定到含有任意多个额外参数的列表上。最后，参数还可以通过关键字而不是位置来映射到形参上。这样，Common Lisp的形参列表对于几种常见的编码问题提供了一种便利的解决方案。

5.3 可选形参

虽然许多像*verbose-sum*这样的函数只有必要形参，但并非所有函数都如此简单。有时一个函数将带有一个只有特定调用者才会关心的形参，这可能是因为它有一个合理的默认值。例如一个可以创建按需增长的数据结构的函数。由于数据结构可以增长，那么从正确性角度来说，它的初始尺寸就无关紧要了。那些清楚知道自己打算在数据结构中放置多少个元素的调用者们，可以通过设置特定的初始尺寸来改进其程序的性能，而多数调用者只需让实现数据结构的代码自行选择一个好的通用值就可以了。在Common Lisp中，你可以使用可选形参，从而使两类调用者都满意。不在意的调用者们将得到一个合理的默认值，而其他调用者们有机会提供一个指定的值。^①

为了定义一个带有可选形参的函数，在必要形参的名字之后放置符号`&optional`，后接可选形参的名字。下面就是一个简单的例子：

```
(defun foo (a b &optional c d) (list a b c d))
```

当该函数被调用时，实参被首先绑定到必要形参上。在所有必要形参都被赋值以后，如果还有任何实参剩余，它们的值将被赋给可选形参。如果实参在所有可选形参被赋值之前用完了，那么其余的可选形参将自动绑定到值`NIL`上。这样，前面定义的函数会给出下面的结果：

```
(foo 1 2)      → (1 2 NIL NIL)
(foo 1 2 3)   → (1 2 3 NIL)
(foo 1 2 3 4) → (1 2 3 4)
```

Lisp仍然可以确保适当数量的实参被传递给函数——在本例中是2到4个。而如果函数用太少或太多的参数来调用的话，将会报错。

当然，你会经常想要一个不同于`NIL`的默认值。这时可以通过将形参名替换成一个含有名字跟一个表达式的列表来指定该默认值。只有在调用者没有传递足够的实参来为可选形参提供值的时候，这个表达式才会被求值。通常情况只是简单地提供一个值作为表达式：

^① 在那些不直接支持可选形参的语言里，程序员们通常可以找到模拟它们的方式。一种技术是使用可区分的“no-value”值供调用者传递，以说明它们想要一个给定形参的默认值。例如在C语言中，通常使用NULL作为这样的可区分值。尽管如此，这种在函数与其调用者之间的协议完全是自组织的——在某些函数或某些实参中，NULL可能是一个可区分值，而在另一些函数或实参中，这样的特殊值可能是-1或一些由#define所定义的常量。

在像Java这种支持用多个定义重载单个方法的语言里，可选形参也可以通过提供多个具有相同名称，但不同实参数的方法(method)来模拟，这时当一个方法使用较少的实参来调用时，会以默认值代替缺少的实参去调用“真实”的那个方法。

```
(defun foo (a &optional (b 10)) (list a b))
```

上述函数要求将一个实参绑定到形参a上。当存在第二个实参时，第二个形参b将使用其值，否则使用10。

```
(foo 1 2) → (1 2)
(foo 1) → (1 10)
```

不过有时可能需要更灵活地选择默认值。比如可能想要基于其他形参来计算默认值。默认值表达式可以引用早先出现在形参列表中的形参。如果要编写一个返回矩形的某种表示的函数，并且想要使它可以特别方便地产生正方形，那么可以使用一个像这样的形参列表：

```
(defun make-rectangle (width &optional (height width)) ...)
```

除非明确指定否则这将导致height形参带有和width形参相同的值。

5

有时，有必要去了解一个可选形参的值究竟是被调用者明确指定还是使用了默认值。除了通过代码来检查形参的值是否为默认值（假如调用者碰巧显式传递了默认值，那么这样做终归是无效的）以外，你还可以通过在形参标识符的默认值表达式之后添加另一个变量名来做到这点。该变量将在调用者实际为该形参提供了一个实参时被绑定到真值，否则为NIL。通常约定，这种变量的名字与对应的真实形参相同，但是带有一个-supplied-p后缀。例如：

```
(defun foo (a b &optional (c 3 c-supplied-p))
  (list a b c c-supplied-p))
```

这将给出类似下面的结果：

```
(foo 1 2) → (1 2 3 NIL)
(foo 1 2 3) → (1 2 3 T)
(foo 1 2 4) → (1 2 4 T)
```

5.4 剩余形参

可选形参仅适用于一些较为分散并且不能确定调用者是否会提供值的形参。但某些函数需要接收可变数量的实参，比如说前文已然出现过的一些内置函数。**FORMAT**有两个必要实参，即流和控制串。但在这两个之后，它还需要一组可变数量的实参，这取决于控制串需要插入多少个值。**+**函数也接受可变数量的实参——没有特别的理由限制它只能在两个数之间相加，它可以对任意数量的值做加法运算（它甚至可以没有实参，此时返回0——加法的底数）。下面这些都是这两个函数的合法调用：

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "x: ~d y: ~d" x y)
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
```

很明显，也可以通过简单地给它一些可选形参来写出接受可变数量实参的函数，但这样将会

非常麻烦，光是写形参列表就已经足够麻烦了，何况还要在函数体中处理所有这些形参。为了做好这件事，还不得不使用一个合法函数调用所能够传递的那么多的可选形参。这一具体数量与具体实现相关，但可以保证至少有50个。在当前所有实现中，它的最大值范围从4096到536 870 911。^①汗，这种绞尽脑汁的无聊事情绝对不是Lisp风格。

相反，Lisp允许在符号`&rest`之后包括一揽子形参。如果函数带有`&rest`形参，那么任何满足了必要和可选形参之后的其余所有实参就将被收集到一个列表里成为该`&rest`形参的值。这样，`FORMAT`和`+`的形参列表可能看起来会是这样：

```
(defun format (stream string &rest values) ...)
(defun + (&rest numbers) ...)
```

5.5 关键字形参

尽管可选形参和剩余形参带来了很大的灵活性，但两者都不能帮助应对下面的情形。假设有一个接受四个可选形参的函数，如果在多数的函数调用中，调用者只想为四个参数中的一个提供值，并且更进一步，不同的调用者甚至有可能将分别选择使用其中一个参数。

想为第一个形参提供值的调用者将会很方便——只需传递一个可选实参，然后忽略其他就好了。但是所有其他的调用者将不得不为所不关心的一到三个形参传递一些值。这不正是可选形参想来解决的问题吗？

当然是。问题在于可选形参仍然是位置相关的——如果调用者想要给第四个可选形参传递一个显式的值，就会导致前三个可选形参对于该调用者来说变成了必要形参。幸好我们有另一种形参类型，关键字形参，它可以允许调用者指定具体形参相应所使用的值。

为了使函数带有关键字形参，在任何必要的`&optional`和`&rest`形参之后，可以加上符号`&key`以及任意数量的关键字形参标识符，后者的格式类似于可选形参标识符。下面就是一个只有关键字形参的函数：

```
(defun foo (&key a b c) (list a b c))
```

当调用这个函数时，每一个关键字形参将被绑定到紧跟在同名键字后面的那个值上。如第4章所述，关键字是以冒号开始的名字，并且它们被自动定义为自求值常量。

如果一个给定的关键字没有出现在实参列表中，那么对应的形参将被赋予其默认值，如同可选形参那样。因为关键字实参带有标签，所以它们在必要实参之后可按任意顺序进行传递。例如`foo`可以用下列形式调用：

(foo)	→ (NIL NIL NIL)
(foo :a 1)	→ (1 NIL NIL)
(foo :b 1)	→ (NIL 1 NIL)
(foo :c 1)	→ (NIL NIL 1)
(foo :a 1 :c 3)	→ (1 NIL 3)
(foo :a 1 :b 2 :c 3)	→ (1 2 3)

^① 常量`CALL-ARGUMENTS-LIMIT`将告诉你这个与具体实现有关的数值。

```
(foo :a 1 :c 3 :b 2) → (1 2 3)
```

如同可选形参那样，关键字形参也可以提供一个默认值形式以及一个supplied-p变量名。在关键字形参和可选形参中，这个默认值形式都可以引用那些早先出现在形参列表中的形参。

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b)))
  (list a b c b-supplied-p))
(foo :a 1) → (1 0 1 NIL)
(foo :b 1) → (0 1 1 T)
(foo :b 1 :c 4) → (0 1 4 T)
(foo :a 2 :b 1 :c 4) → (2 1 4 T)
```

同样，如果出于某种原因想让调用者用来指定形参的关键字不同于实际形参名，那么可以将形参名替换成一个列表，令其含有调用函数时使用的关键字以及用作形参的名字。比如说下面这个foo的定义：

```
(defun foo (&key ((:apple a)) ((:box b) 0) ((:charlie c) 0 c-supplied-p))
  (list a b c c-supplied-p))
```

可以让调用者这样调用它：

```
(foo :apple 10 :box 20 :charlie 30) → (10 20 30 T)
```

这种风格在想要完全将函数的公共API与其内部细节相隔离时特别有用，通常是因为想要在内部使用短变量名，而不是API中的描述性关键字。不过该特性不常被用到。

5

5.6 混合不同的形参类型

在单一函数里使用所有四种类型形参的情况虽然罕见，但也是可能的。无论何时，当用到多种类型的形参时，它们必须以这样的顺序声明：首先是必要形参，其次是可选形参，再次是剩余形参，最后才是关键字形参。但在使用多种类型形参的函数中，一般情况是将必要形参和另外一种类型的形参组合使用，或者可能是组合`&optional`形参和`&rest`形参。其他两种组合方式，无论是`&optional`形参还是`&rest`形参，当与`&key`形参组合使用时，都可能导致某种奇怪的行为。

将`&optional`形参和`&key`形参组合使用时将产生非常奇怪的结果，因此也许应该避免将它们一起使用。问题出在如果调用者没有为所有可选形参提供值时，那么没有得到值的可选形参将吃掉原本用于关键字形参的关键字和值。例如，下面这个函数很不明智地混合了`&optional`形参和`&key`形参：

```
(defun foo (x &optional y &key z) (list x y z))
```

如果像这样调用的话，就没问题：

```
(foo 1 2 :z 3) → (1 2 3)
```

这样也可以：

```
(foo 1) → (1 nil nil)
```

但是这样的话将报错：

```
(foo 1 :z 3) → ERROR
```

这是因为关键字:`:z`被作为一个值填入到可选的`y`形参中了，只留下了参数`3`被处理。在这里，Lisp期待一个成对的关键字/值，或者什么也没有，否则就会报错。也许更坏的是，如果该函数带有两个`&optional`形参，上面最后一个调用将导致值:`:z`和`3`分别被绑定到两个`&optional`形参上，而`&key`形参`z`将得到默认值`NIL`，而不声明缺失了东西。

一般而言，如果正在编写一个同时使用`&optional`形参和`&key`形参的函数，可能就应该将它变成全部使用`&key`形参的形式——它们更灵活，并且总会可以在不破坏该函数的已有调用的情况下添加新的关键字形参。也可以移除关键字形参，只要没人再使用它们。^①一般而言，使用关键字形参将会使代码相对易于维护和拓展——如果需要为函数添加一些需要用到新参数的新行为，就可以直接添加关键字形参，而无需修改甚至重新编译任何调用该函数的已有代码。

虽然可以安全地组合使用`&rest`形参和`&key`形参，但其行为初看起来可能会有一点奇怪。正常地来讲，无论是`&rest`还是`&key`出现在形参列表中，都将导致所有出现在必要形参和`&optional`形参之后的那些值被特别处理——要么作为`&rest`形参被收集到一个形参列表中，要么基于关键字被分配到适当的`&key`形参中。如果`&rest`和`&key`同时出现在形参列表中，那么两件事都会发生——所有剩余的值，包括关键字本身，都将被收集到一个列表里，然后被绑定到`&rest`形参上；而适当的值，也会同时被绑定到`&key`形参上。因此，给定下列函数：

```
(defun foo (&rest rest &key a b c) (list rest a b c))
```

将得到如下结果：

```
(foo :a 1 :b 2 :c 3) → ((:A 1 :B 2 :C 3) 1 2 3)
```

5.7 函数返回值

目前写出的所有函数都使用了默认的返回值行为，即最后一个表达式的值被作为整个函数的返回值。这是从函数中返回值的最常见方式。

但某些时候，尤其是想要从嵌套的控制结构中脱身时，如果有办法从函数中间返回，那将是非常便利的。在这种情况下，你可以使用`RETURN-FROM`特殊操作符，它能够立即以任何值从函数中间返回。

在第20章将会看到，`RETURN-FROM`事实上不只用于函数，它还可以用来从一个由`BLOCK`特殊操作符所定义的代码块中返回。不过`DEFUN`会自动将其整个函数体包装在一个与其函数同名的代码块中。因此，对一个带有当前函数名和想要返回的值的`RETURN-FROM`进行求值将导致函数立即以该值退出。`RETURN-FROM`是一个特殊操作符，其第一个“参数”是它想要返回的代码块名。该名字不被求值，因此无需引用。

^① 有四个标准函数同时接受`&optional`参数和`&key`参数——`READ-FROM-STRING`、`PARSE-NAMESTRING`、`WRITE-LINE`和`WRITE-STRING`。在标准化的过程中，出于跟早期Lisp方言向后兼容的目的它们被原样保留了下来。`READ-FROM-STRING`应该是新的Lisp程序员最常用到的函数之一，一个诸如`(read-from-string s :start 10)`这样的调用看起来会忽略`:start`关键字形参，直接从索引位置0而非10处开始读取。这是因为`READ-FROM-STRING`还有两个`&optional`形参将参数`:start`和`10`覆盖了。

下面这个函数使用了嵌套循环来发现第一个数对——每个都小于10，并且其乘积大于函数的参数，它使用RETURN-FROM在发现之后立即返回该数对：

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

必须承认的是，不得不指定正在返回的函数名多少会有些不便——比如改变了函数的名字，就需要同时改变RETURN-FROM中所使用的名字。^①但在事实上，显式的RETURN-FROM调用在Lisp中出现的频率远小于return语句在源自C的语言里所出现的频率，因为所有的Lisp表达式，包括诸如循环和条件语句这样的控制结构，都会求值得到一个值。因此在实践中这不是什么问题。

5

5.8 作为数据的函数——高阶函数

使用函数的主要方式是通过名字来调用它们，但有时将函数作为数据看待也是很有用的。例如，可以将一个函数作为参数传给另一个函数，从而能写出一个通用的排序函数，允许调用者提供一个比较任意两元素的函数，这样同样的底层算法就可以跟许多不同的比较函数配合使用了。类似地，回调函数（callback）和钩子（hook）也需要能够保存代码引用便于以后运行。由于函数已经是一种对代码比特进行抽象的标准方式，因此允许把函数视为数据也是合理的。^②

在Lisp中，函数只是另一种类型的对象。在用DEFUN定义一个函数时，实际上做了两件事：创建一个新的函数对象以及赋予其一个名字。在第3章里我们看到，也可以使用LAMBDA表达式来创建一个函数而无需为其指定一个名字。一个函数对象的实际表示，无论是有名还是匿名的，都只是一些二进制数据——以原生编译的Lisp形式存在，可能大部分是由机器码构成。只需要知道如何保持它们以及需要时如何调用它们。

特殊操作符FUNCTION提供了用来获取一个函数对象的方法。它接受单一实参并返回与该参数同名的函数。这个名字是不被引用的。因此如果一个函数foo的定义如下。

```
CL-USER> (defun foo (x) (* 2 x))
FOO
```

就可以得到如下的函数对象。^③

```
CL-USER> (function foo)
#<Interpreted Function FOO>
```

事实上，你已经用过FUNCTION了，但它是以伪装的形式出现的。第3章里用到的#`语法就是

^① 另一个宏RETURN不要求使用一个名字。尽管如此，你不能用它来代替RETURN-FROM从而避免指定函数名，它是一个从称为NIL的块中返回的语法糖。我将在第20章里跟BLOCK以及RETURN-FROM一起讨论其细节。

^② 当然了，Lisp并不是唯一的将函数视为数据的语言。C使用函数指针；Perl使用子例程引用（subroutine reference）；Python使用与Lisp相似的模式；C#使用了代理（delegate），其本质上是带有类型的函数指针；而Java则使用了相当笨重的反射（reflection）和匿名类机制。

^③ 一个函数对象的确切打印格式将随着不同的实现而有所不同。

FUNCTION的语法糖，正如“`”是**QUOTE**的语法糖一样。^①因此也可以像这样得到**foo**的函数对象。

```
CL-USER> #'foo
#<Interpreted Function FOO>
```

一旦得到了函数对象，就只剩下一件事可做了——调用它。Common Lisp提供了两个函数用来通过函数对象调用函数：**FUNCALL**和**APPLY**。^②它们的区别仅在于如何获取传递给函数的实参。

FUNCALL用于在编写代码时确切知道传递给函数多少实参时。**FUNCALL**的第一个实参是被调用的函数对象，其余的实参被传递到该函数中。因此，下面两个表达式是等价的：

```
(foo 1 2 3) ≡ (funcall #'foo 1 2 3)
```

不过，用**FUNCALL**来调用一个写代码时名字已知的函数毫无意义。事实上，前面的两个表达式将很可能被编译成相同的机器指令。

下面这个函数演示了**FUNCALL**的另一个更有建设性的用法。它接受一个函数对象作为实参，并使用实参函数在min和max之间以step为步长的返回值来绘制一个简单的ASCII式柱状图：

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

FUNCALL表达式在每个*i*值上计算函数的值。内层LOOP循环使用计算得到的值来决定向标准输出打印多少星号。

请注意，不需要使用**FUNCTION**或`'来得到fn的函数值。因为它是作为函数对象的变量的值，所以你需要它被解释成一个变量。可以用任何接受单一数值实参的函数来调用plot，例如内置的函数**EXP**，它返回以e为底以其实参为指数的值。

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
***
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
NIL
```

然而，当实参列表只在运行期已知时，**FUNCALL**的表现不佳。例如，为了再次调用plot函数，假设你已有一个列表，其包括一个函数对象、一个最小值和一个最大值以及一个步长。换句

① 考虑**FUNCTION**的最佳方式是将其视为一个特殊类型的引用。**QUOTE**一个符号可以避免其被求值，从而得到该符号本身而不是由该符号所命名的变量的值。**FUNCTION**同样规避了正常的求值规则，但并非是避免其符号被求值，而是使其作为一个函数的名字来求值，就好像它作为函数调用表达式中的函数名那样。

② 实际上还有第三个，特殊操作符**MULTIPLE-VALUE-CALL**，但是我将保留到第20章中当我讨论到返回多值的表达式时再讨论它。

话说，这个列表包含了你想要作为实参传给plot的所有的值。假设这个列表保存在变量plot-data中，可以像这样用列表中的值来调用plot：

```
(plot (first plot-data) (second plot-data) (third plot-data) (fourth plot-data))
```

这样固然可以，但仅仅为了将实参传给plot而显式地将其解开，看起来相当讨厌。

这就是需要**APPLY**的原因。和**FUNCALL**一样，**APPLY**的第一个参数是一个函数对象。但在这个函数对象之后，它期待一个列表而非单独的实参。它将函数应用在列表中的值上，这就使你可以写出下面的替代版本：

```
(apply #'plot plot-data)
```

更方便的是，**APPLY**还接受“孤立”(loose)的实参，只要最后一个参数是个列表。因此，假如plot-data只含有最小、最大和步长值，那么你仍然可以像这样来使用**APPLY**在该范围内绘制**EXP**函数：

```
(apply #'plot #'exp plot-data)
```

APPLY并不关心所用的函数是否接受**&optional**、**&rest**或**&key**实参——由任何孤立实参和最后的列表所组合而成的实参列表必定是一个合法的实参列表，其对于该函数来说带有足够的实参用于所有必要形参和适当的关键字形参。

5.9 匿名函数

一旦开始编写或只是使用那些可以接受其他函数作为实参的函数，你就必然发现，有时不得不去定义和命名一个仅使用一次的函数，尤其是你可能从不用名字来调用它时，这会让人相当恼火。

觉得没必要用**DEFUN**来定义一个新函数时，可以使用一个**LAMBDA**表达式创建匿名的函数。第3章里讨论过，一个**LAMBDA**表达式形式如下：

```
(lambda (parameters) body)
```

可以将**LAMBDA**表达式视为一种特殊类型的函数名，其名字本身直接描述函数的用途。这就解释了为什么可以使用一个带有#’的**LAMBDA**表达式来代替一个函数名。

```
(funcall #'(lambda (x y) (+ x y)) 2 3) → 5
```

甚至还可以在一个函数调用表达式中将**LAMBDA**表达式用作函数名。由此一来，我们可以在需要时以更简洁方式来书写前面的**FUNCALL**表达式如下：

```
((lambda (x y) (+ x y)) 2 3) → 5
```

但几乎没人这样做。它唯一的用途是来强调将**LAMBDA**表达式用在任何一个正常函数名可以出现

^① 的场合都是合法的。

在需要传递一个作为参数的函数给另一个函数，并且需要传递的这个函数简单到可以内联表达时，匿名函数特别有用。例如，假设想要绘制函数 $2x$ ，你可以定义下面的函数：

```
(defun double (x) (* 2 x))
```

并随后将其传给plot：

```
CL-USER> (plot #'double 0 10 1)
***  
*****  
*****  
***** .  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
NTL:
```

但如果写成这样将会更简单和清晰：

LAMBDA表达式的另一项重要用途是制作闭包 (closure)，即捕捉了其创建时环境信息的函数。你在第3章里使用了一点儿闭包，但要深入了解闭包的工作原理及其用途，更多的还是要从变量而非函数的角度去考察，因此我将在下一章里讨论它们。

① 在Common Lisp里也可以不带前缀#来使用一个LAMBDA表达式作为FUNCALL的参数（或是其他一些接受函数参数的函数，诸如SORT和MAPCAR），像这样：

```
(funcall (lambda (x y) (+ x y)) 2 3)
```

(**LAMBDA** (*x y*) (+ *x y*)) *z* ;
这是合法的，并且出于一个诡异的理由，它跟带有#的版本等价。在历史上，**LAMBDA**表达式本身并非是可以求值的表达式。这就是说，**LAMBDA**并非是一个函数、宏或特殊操作符的名字。并且一个以符号**LAMBDA**开始的列表会被Lisp作为特殊的语法构造来识别成一种函数名。

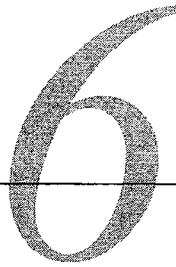
如果还是这样，那么 `(funcall (lambda (...) ...))` 就不合法，因为 **FUNCALL** 是一个函数，而一个函数调用的正常求值规则将要求该 **LAMBDA** 表达式被求值。不过，在 ANSI 标准化过程晚期，为了有可能实现 ISLISP（另一种同时被标准化的 Lisp 方言，将其作为用户层面的兼容层构建在 Common Lisp 之上），人们定义了一个 **LAMBDA** 宏，其展开成一个包装在 **LAMBDA** 表达式外围的 **FUNCTION** 调用。换句话说，下面的 **LAMBDA** 表达式：

(lambda () 42)

当其出现在一个会被求值的上下文由时，将展开成下面这样：

```
(function (lambda () 42)) : or #'(lambda () 42)
```

这使其可以合法地用在需要求值的位置上，例如作为`FUNCALL`的参数。换句话说，它纯粹是一个语法糖。多数人要么总是在值位置上的`LAMBDA`表达式前使用`#`，要么总是不用。在本书中，我将总是使用`#`。



我们需要了解的下一个基本程序构造单元是变量。Common Lisp支持两种类型的变量：词法（lexical）变量和动态（dynamic）变量^①。这两种变量类型分别对应于其他语言中的局部变量和全局变量，不过也只能说是大致相似。一方面，某些语言中的局部变量更像是Common Lisp的动态变量^②。另一方面，某些语言中的局部变量虽然是词法作用域的，但并没有提供由Common Lisp的词法变量所提供的所有功能，尤其是并非所有语言都提供了支持闭包的词法作用域变量。

许多含有变量的表达式都可以同时使用词法变量和动态变量，这样一来更令人困惑了。因此本章我先讨论同时涉及两种类型的Lisp变量的几个方面，然后再谈及词法变量和动态变量各自的特征，随后再讨论Common Lisp的通用赋值操作符`SETF`，它用于为变量和其他任何可以保存值的位置赋予新值。

6.1 变量的基础知识

和其他语言一样，Common Lisp中的变量是一些可以保存值的具名位置。但在Common Lisp中，变量并非像在Java和C++等语言中那样带有确定的类型，也就是说不需要为每一个变量声明其可以保存对象的类型。相反，一个变量可以保存任何类型的值，并且这些值带有可用于运行期类型检查的类型信息。因此，Common Lisp是动态类型的——类型错误会被动态地检测到。举个例子，假如将某个并非数字的对象传给了`+`函数，那么Common Lisp将会报类型错误。而另一方面，Common Lisp是一种强类型语言，因为所有的类型错误都将被检测到——无法将一个对象作为其不属于的类型的实例来对待。^③

① 出于某种你将在本章后面看到的原因，动态变量有时也称为特殊变量（special variable）。你需要注意这两个同义词，因为某些人（以及某些Lisp实现）使用其中一种术语，而其他人则习惯使用另外一种。

② 对于早期的Lisp而言，至少当其解释执行代码时，它倾向于使用动态变量作为局部变量。Elisp，用于Emacs的Lisp方言，在这方面有点守旧，它一直只支持动态变量。其他语言已经作出了从动态到词法的转换。例如Perl的`local`变量是动态的，而它由Perl 5所引入的`my`变量则是词法的。Python从来没有真正的动态变量，而只是在版本2.2以后引入了真正的词法作用域。（Python的词法变量相比Lisp而言仍在某些方面受限，因为语言语法中合并了赋值与绑定。）

③ 事实上，说所有类型错误将总能被检测并不是很正确——有可能使用可选的声明来告诉编译器，特定的变量将总是包含一个特定类型的对象，并且在一个特定的代码区域里关闭运行期类型检查。尽管如此，这类声明通常用于在开发和调试之后进行的代码优化，而不是在正常开发的期间。



至少从概念上来说，Common Lisp中所有的值都是对象的引用。^①因此，将一个变量赋予新值就会改变该变量所指向的对象，而对之前被引用的对象却没有影响。尽管如此，如果一个变量保存了对一个可变对象的引用，那么就可以用该引用来修改此对象，而这种改动将应用于任何带有相同对象引用的代码。

而另一种已经用到的引入新变量的方式是定义函数形参。正如前一章所示，在用DEFUN来定义函数时，形参列表定义了当函数被调用时用来保存实参的变量。例如，下列函数定义了三个变量x、y和z，用来保存其实参：

```
(defun foo (x y z) (+ x y z))
```

每当函数被调用时，Lisp就会创建新的绑定来保存由函数调用者所传递的实参。绑定代表了变量在运行期的存在。单个变量就是在程序源代码中所指出的那种东西。在程序运行过程中可以有多个不同的绑定，单个变量甚至可以同时带有多重绑定。例如，一个递归函数的形参会在每一次函数调用中被重新绑定。

和所有Common Lisp变量一样，函数形参也可以保存对象引用。^②因此，可以在函数体内为一个函数形参赋予新值，而这并不会影响到同样函数的另一个调用所创建的绑定。但如果改变了传递给函数的可变对象，则这些改动将会被调用者看到，因为无论调用者还是被调用者都在引用同一个对象。

引入新变量的另一种方式是使用LET特殊操作符。下面就是一个LET形式的结构：

```
(let (variable*)
      body-form*)
```

其中每一个variable都是一个变量初始化形式。每一个初始化形式要么是一个含有变量名和初值形式的列表，要么就是一个简单的变量名——作为将变量初始化到NIL的简略写法。例如，下面的LET形式会将三个变量x、y和z绑定到初始值10、20和NIL上：

```
(let ((x 10) (y 20) z)
      ...)
```

当这个LET形式被求值时，所有的初始值形式都将首先被求值，然后创建出新的绑定，并在形式体被执行之前这些绑定将初始化到适当的初始值上。在LET形式体中，变量名将引用新创建的绑定。在LET形式体执行结束后，这些变量名将重新引用在执行LET之前它们所引用的内容，如果有的话。

形式体中最后一个表达式的值将作为LET表达式的值返回。和函数形参一样，由LET所引入

^① 作为一种优化，特定类型的对象，诸如特定大小以下的整数与字符，可能会在内存中直接表示，而其他对象将被表示成一个实际对象的指针。但由于整数和字符都是不可修改的，因此在不同变量中是否存在同一个对象的多个副本也就无关紧要了，这就是在第4章里所讨论的EQ和EQUAL的根本区别。

^② 用编译器作者的话来说，Common Lisp函数是“传值的”，但被传递的值是对象的引用。这跟Java和Python的工作方式相似。

的变量将在每次进入LET时被重新绑定。^①

函数形参和LET变量的作用域（变量名可用来引用该绑定的程序区域）被限定在引入该变量的形式之内，该形式即函数定义或LET，被称为绑定形式。你很快将看到，词法变量和动态变量使用两种略有不同的作用域机制，但两者的作用域都被界定在绑定形式之内。

如果嵌套了引入同名变量的绑定形式，那么最内层的变量绑定将覆盖外层的绑定。例如，在调用下面的函数时，将创建一个形参x的绑定来保存函数的参数。第一个LET创建了一个带有初始值2的新绑定，而内层的LET创建了另外一个绑定，其初始值为3。右边的竖线标记出了每一个绑定的作用域。

```
(defun foo (x)
  (format t "Parameter: ~a~%" x)
  (let ((x 2))
    (format t "Outer LET: ~a~%" x)
    (let ((x 3))
      (format t "Inner LET: ~a~%" x)))
  (format t "Outer LET: ~a~%" x))
(format t "Parameter: ~a~%" x))
```

6

每一次对x的引用都将指向最小封闭作用域中的绑定。一旦程序控制离开了一个绑定形式的作用域，其最近的闭合作用域中的绑定就被解除覆盖，并且x将转而指向它。因此，调用foo将得到这样的输出：

```
CL-USER> (foo 1)
Parameter: 1
Outer LET: 2
Inner LET: 3
Outer LET: 2
Parameter: 1
NIL
```

后面的章节将讨论其他可作为绑定形式使用的程序构造，其特点在于所引入的新变量名只能用于该构造。

例如，第7章将遇到DOTIMES循环，这是一种基本的计数循环。它引入了一个变量用来保存每次通过循环时递增的计数器的值。例如下面这个可以打印从0-9数字的循环，它绑定了变量x：

```
(dotimes (x 10) (format t "~d " x))
```

另一个绑定形式是LET的变体：LET*。两者的区别在于，在一个LET中，被绑定的变量名只能用在LET的形式体之内——LET形式中变量列表之后的那部分；但在一个LET*中，每个变量的初始值形式，都可以引用到那些在变量列表中早先引入的变量。因此可以写成下面这样：

^① LET形式和函数形参中的变量是以完全相同的机制创建的。事实上，在某些Lisp方言中，尽管不是Common Lisp，LET只是一个展开到一个匿名函数调用的宏。也就是说，在那些方言里，

```
(let ((x 10)) (format t "~a" x))  
是一个展开到下列结果的宏：  
((lambda (x) (format t "~a" x)) 10)
```

```
(let* ((x 10)
      (y (+ x 10)))
  (list x y))
```

但不能这样写：

```
(let ((x 10)
      (y (+ x 10)))
  (list x y))
```

不过也可以通过嵌套的LET来达到相同的效果：

```
(let ((x 10))
  (let ((y (+ x 10)))
    (list x y)))
```

6.2 词法变量和闭包

默认情况下，Common Lisp中所有的绑定形式都将引入词法作用域变量。词法作用域的变量只能由那些在文本上位于绑定形式之内的代码所引用。那些曾经使用Java、C、Perl或者Python来编程的人们应该熟悉词法作用域，因为它们都提供词法作用域的局部变量。如此说来，Algol程序员们也该对其感到自然才是，因为Algol在20世纪60年代首先引入了词法作用域。

尽管如此，但Common Lisp的词法变量还是有一些变化的，至少和最初的Algol模型相比是这样。变化之处在于将词法作用域和嵌套函数一起使用时，按照词法作用域的规则，只有文本上位于绑定形式之内的代码可以指向一个词法变量。但是当一个匿名函数含有一个对来自封闭作用域之内词法变量的引用时，将会发生什么呢？例如，在下面的表达式中：

```
(let ((count 0)) #'(lambda () (setf count (1+ count))))
```

根据词法作用域规则，**LAMBDA**形式中对**count**的引用应该是合法的，而这个含有引用的匿名函数将被作为**LET**形式的值返回，并可能会通过**FUNCALL**被不在**LET**作用域之内的代码所调用。这样会发生什么呢？正如你将看到的那样，当**count**是一个词法变量时，情况一切正常。本例中，当控制流进入**LET**形式时所创建的**count**绑定将被尽可能地保留下来，只要某处保持了一个对**LET**形式所返回的函数对象的引用即可。这个匿名函数被称为一个闭包，因为它“封闭包装”了由**LET**创建的绑定。

理解闭包的关键在于，被捕捉的是绑定而不是变量的值。因此，一个闭包不仅可以访问它所闭合的变量的值，还可以对其赋予在闭包被调用时不断变化的新值。例如，可以像下面这样将前面的表达式所创建的闭包捕捉到一个全局变量里：

```
(defparameter *fn* (let ((count 0)) #'(lambda () (setf count (1+ count)))))
```

然后每当调用它时，**count**的值将被加1：

```
CL-USER> (funcall *fn*)
1
CL-USER> (funcall *fn*)
2
```

```
CL-USER> (funcall *fn*)
3
```

单一闭包可以简单地通过引用变量来闭合许多变量绑定，或是多个闭合可以捕捉相同的绑定。例如，下面的表达式返回由三个闭合所组成的列表，一个可以递增其所闭合的count绑定的值，另一个可以递减它，还有一个返回它的当前值。

```
(let ((count 0))
  (list
    #'(lambda () (incf count))
    #'(lambda () (decf count))
    #'(lambda () count)))
```

6.3 动态变量

词法作用域的绑定通过限制作用域（其中给定的名字只具有字面含义）使代码易于理解，这就是大多数现代语言将词法作用域用于局部变量的原因。尽管如此，有时的确需要全局变量——一种可以从程序的任何位置访问到的变量。尽管随意使用全局变量将使代码变得杂乱无章，就像毫无节制地使用`goto`那样，但全局变量确实有其合理的用途，并以某种形式存在于几乎每种编程语言里。^①正如你即将看到的，Lisp的全局变量和动态变量都更为有用并且更易于管理。

Common Lisp提供了两种创建全局变量的方式：`DEFVAR`和`DEFPARAMETER`。两种形式都接受一个变量名、一个初始值以及一个可选的文档字符串。在被`DEFVAR`和`DEFPARAMETER`定义以后，该名字可用于任何位置来指向全局变量的当前绑定。如前所述，全局变量习惯上被命名为以`*`开始和结尾的名字。你将在本节后面看到所述遵守该命名约定的重要性。`DEFVAR`和`DEFPARAMETER`的示例如下：

```
(defvar *count* 0
  "Count of widgets made so far.")

(deparameter *gap-tolerance* 0.001
  "Tolerance to be allowed in widget gaps.")
```

两种形式的区别在于，`DEFPARAMETER`总是将初始值赋给命名的变量，而`DEFVAR`只有当变量未定义时才这样做。`DEFVAR`形式也可以不带初始值来使用，从而在不给定其值的情况下定义一个全局变量。这样的变量称为未绑定的（*unbound*）。

从实践上来讲，应该使用`DEFVAR`来定义某些变量，这些变量所含数据是应持久存在的，即使使用到该变量的源码发生改变时也应如此。例如，假设前面定义的两个变量是一个用来控制部件工厂的应用程序的一部分，那么使用`DEFVAR`来定义`*count*`变量就比较合适，因为目前已生产的部件数量不会因为对部件生产的代码做了某些改变而就此作废。^②

^① Java将全局变量伪装成公共静态字段，C使用`extern`变量，而Python的模块级别变量和Perl的包级别变量同样可以从任何位置被访问到。

^② 如果你特定想要重设由`DEFVAR`定义的变量，那要么使用`SETF`直接设置它，要么使用`MAKUNBOUND`先将其变成未绑定的，再重新求值`DEFVAR`形式。

另一方面，假定变量`*gap-tolerance*`对于部件生产代码本身的行为具有影响。如果你决定使用一个或紧或松的容差值，并且改变了`DEFPARAMETER`形式中的值，那么就要在重新编译和加载文件时让这一改变产生效果。

在用`DEFVAR`和`DEFPARAMETER`定义了一个变量之后，就可以从任何一个地方引用它。例如，可以定义下面的函数来递增已生产部件的数量。

```
(defun increment-widget-count () (incf *count*))
```

全局变量的优势在于不必到处传递它们。多数语言将标准输入与输出流保存在全局变量里正是出于这个原因——你根本不知道什么时候会向标准输出流打印东西，并且也不想仅仅由于日后有人需要，就使每个函数都不得不接受并传递含有这些流的参数。

不过，一旦像标准输出流这样的值被保存在一个全局变量中，并且已经编写了引用那个全局变量的代码，那么试图通过更改变量值来临时改变代码行为的做法就颇为诱人了。

例如，假设正工作的一个程序中含有的一些底层日志函数会将输出打印到位于全局变量`*standard-output*`中的流上。现在假设在程序的某个部分里，想要将所有这些函数所生成的输出捕捉到一个文件里，那么可以打开一个文件并将得到的流赋予`*standard-output*`。现在底层函数们将把它们的输出发往该文件。

这样工作得很好，但假如完成工作时忘记将`*standard-output*`重新设置回最初的流上，那么程序中所有用到`*standard-output*`的其他代码也会将它们的输出发往该文件。^①

真正所需的代码包装方式似乎应如下所述：“在从这里以下的所有代码中——所有它调用的函数以及它们进一步调用的函数，诸如此类，直到最底层的函数全局变量`*standard-output*`都应使用该值。”然后当上层的函数返回时，`*standard-output*`应该自动恢复到其原来的值。

这看起来正像是Common Lisp的另一种变量，即动态变量所做的事。当绑定了一个动态变量时，例如通过一个`LET`变量或函数形参，在被绑定项上所创建的绑定替换了在绑定形式期间的对应全局绑定。与词法绑定——只能被绑定形式的词法作用域之内的代码所引用——所不同的是，动态绑定可以被任何在绑定形式执行期间所调用到的代码所引用。^②显然所有全局变量事实上都是动态变量。

因此，如果想要临时重定义`*standard-output*`，只需重新绑定它即可，比如说可以使用`LET`。

^① 这种临时重新赋值`* standard-output *`的策略也会在系统使用多线程时失效——如果有多个控制线程同时试图输出到不同的流上，它们将全都试图设置该全局变量到它们想要使用的流上，完全无视彼此的感受。你可以使用一个锁来控制对一个全局变量的访问，但那样就无法充分获得多重并发线程所带来的好处了。因为无论哪个线程正在输出时，即使它们想要输出到一个不同的流上，它将不得不锁住所有其他线程直到完成。

^② 一个绑定可被引用到的时间间隔，其技术术语称为生存期，这样作用域（scope）和生存期（extent）就是两个紧密相关的概念——作用域关注空间而生存期关注时间。词法变量具有词法作用域和不确定的（indefinite）生存期，意思是它们可以在不定长的间隔里保持存在，这取决于它们被需要多久。动态变量正好相反，它们具有不确定的作用域，因为它们可从任何位置访问却有动态的生存期。更加引起误会的是，不确定作用域和动态生存期的组合经常被错误地称为动态作用域（dynamic scope）。

```
(let ((*standard-output* *some-other-stream*))  
  (stuff))
```

在任何由于调用stuff而运行的代码中，对*standard-output*的引用将使用由LET所建立的绑定，并且当stuff返回并且程序控制离开LET时，这个对*standard-output*的新绑定将随之消失，接下来对*standard-output*的引用将看到LET之前的绑定。在任何给定时刻，最近建立的绑定会覆盖所有其他的绑定。从概念上讲，一个给定动态变量的每个新绑定都将被推到一个用于该变量的绑定栈中，而对该变量的引用总是使用最近的绑定。当绑定形式返回时，它们所创建的绑定会被从栈上弹出，从而暴露出前一个绑定。^①

一个简单的例子就能揭示其工作原理。

```
(defvar *x* 10)  
(defun foo () (format t "X: ~d~%" *x*))
```

上面的DEFVAR为变量*x*创建了一个到数值10的全局绑定。函数foo中，对*x*的引用将动态地查找其当前绑定。如果从最上层调用foo，由DEFVAR所创建的全局绑定就是唯一可用的绑定，因此它打印出10：

```
CL-USER> (foo)  
X: 10  
NIL
```

但也可以用LET创建一个新的绑定来临时覆盖全局绑定，这样foo将打印一个不同的值：

```
CL-USER> (let ((*x* 20)) (foo))  
X: 20  
NIL
```

现在不使用LET再次调用foo，它将再次看到全局绑定：

```
CL-USER> (foo)  
X: 10  
NIL
```

现在定义另一个函数：

```
(defun bar ()  
  (foo)  
  (let ((*x* 20)) (foo))  
  (foo))
```

注意，中间那个对foo的调用被包装在一个将*x*绑定到新值20的LET形式中。运行bar得到的结果如下所示：

```
CL-USER> (bar)  
X: 10  
X: 20  
X: 10  
NIL
```

^① 尽管标准并未指定如何在Common Lisp中使用多线程，但所有提供多线程的实践都遵循了由Lisp机所建立的原则，在每线程的基础上创建动态绑定，一个对全局变量的引用将查找当前线程中最近建立的绑定，或是全局绑定。

可以看到，第一次对foo的调用看到了全局绑定，其值为10。然而，中间的那个调用却看到了新的绑定，其值为20。但在LET之后，foo再次看到了全局绑定。

和词法绑定一样，赋予新值仅会影响当前绑定。为了理解这点，可以重定义foo来包含一个对*x*的赋值。

```
(defun foo ()
  (format t "Before assignment ~18tX: ~d~%" *x*)
  (setf *x* (+ 1 *x*))
  (format t "After assignment ~18tX: ~d~%" *x*))
```

现在foo打印*x*的值，对其递增，然后再次打印它。如果你只运行foo，将看到这样的结果：

```
CL-USER> (foo)
Before assignment X: 10
After assignment  X: 11
NIL
```

这看起来很正常，现在运行bar：

```
CL-USER> (bar)
Before assignment X: 11
After assignment  X: 12
Before assignment X: 20
After assignment  X: 21
Before assignment X: 12
After assignment  X: 13
NIL
```

注意*x*从11开始——之前的foo调用真的改变了全局的值。来自bar的第一次对foo的调用将全局绑定递增到12。中间的调用由于LET的关系没有看到全局绑定，然后最后一个调用再次看到了全局绑定，并将其从12递增到13。

那么它是怎样工作的呢？LET是怎样知道在它绑定*x*时打算创建的是动态绑定而不是词法绑定呢？这是因为该名字已经被声明为特殊的（special）。^①每一个由DEFVAR和DEFPARAMETER所定义的变量其名字都将被自动声明为全局特殊的。这意味着无论何时你在绑定形式中使用这样一个名字，无论是在LET中，或是作为一个函数形参，亦或是在任何创建新变量绑定的构造中，被创建的绑定将成为一个动态绑定。这就是为什么命名约定如此重要——如果你使用了一个变量，以为它是词法变量，而它却刚好是全局特殊的变量，这就很不好。一方面，你所调用的代码可能在你意想之外改变了绑定的值；另一方面，你可能会覆盖一个由栈的上一级代码所建立的绑定。如果总是按照*命名约定来命名全局变量，就不会在打算建立词法绑定时却意外使用了动态绑定。

也有可能将一个名字声明为局部特殊的，如果在一个绑定形式里将一个名字声明为特殊的，那么为该变量所创建的绑定将是动态的而不是词法的。其他代码可以局部地声明一个名字为特殊的，从而指向该动态绑定。尽管如此，局部特殊变量使用相对较少，所以你不需要担心它们。^②

^① 这就是动态变量有时也被称作特殊变量的原因。

^② 如果你一定想知道的话，你可以在HyperSpec上查找DECLARE、SPECIAL和LOCALLY。

动态绑定使全局变量更易于管理，但重要的是注意到它们将允许超距作用的存在。绑定一个全局变量具有两种超距效果——它可以改变下游代码的行为，并且它也开启了一种可能性，使得下游代码可以为栈的上一级所建立的绑定赋予一个新的值。你应该只有在需要利用这两个特征时才使用动态变量。

6.4 常量

我尚未提到的另一种类型的变量是常值变量 (constant variable)。所有的常量都是全局的，并且使用`DEFCONSTANT`定义，`DEFCONSTANT`的基本形式与`DEFPARAMETER`相似。

```
(defconstant name initial-value-form [ documentation-string ])
```

与`DEFVAR`和`DEFPARAMETER`相似，`DEFCONSTANT`在所使用的名字上产生了全局效果——从此该名字仅被用于指向常量，它不能被用作函数形参或是用任何其他的绑定形式进行重绑定。因此，许多Lisp程序员遵循了一个命名约定，用以+开始和结尾的名字来表示常量，这一约定在某种程度上不像全局特殊名字的*命名约定那样流行，但也不错。^①

关于`DEFCONSTANT`，需要注意的另一点是，尽管语言允许通过重新求值一个带有初始值形式的`DEFCONSTANT`来重定义一个常量，但在重定义之后究竟发生什么是没有定义的。在实践上，多数实现将要求对任何引用了该常量的代码进行求值以便它们能看到新值，因为老的值可能已经内联到代码中了。因此最好只用`DEFCONSTANT`来定义那些真正是常量的东西，例如 π 。而对于那些可能想改变的东西，则应转而使用`DEFPARAMETER`。

6.5 赋值

一旦创建了绑定，就可以对它做两件事：获取当前值以及为它设置新值。正如在第4章里所看到的，一个符号被求值为它所命名的变量的值，因此，可以简单地通过引用这个变量来得到它的当前值。而为绑定赋予新值则要使用`SETF`宏——Common Lisp的通用赋值操作符。下面是`SETF`的基本形式：

```
(setf place value)
```

因为`SETF`是宏，所以它可以检查它所赋值的`place`上的形式，并展开成适当的底层操作来修改那个位置。当该位置是变量时，它展开成一个对特殊操作符`SETQ`的调用，后者可以访问到词法和动态绑定。^②例如，为了将值10赋给变量`x`，可以写成这样：

```
(setf x 10)
```

正如早先所讨论的，为一个绑定赋予新值对该变量的任何其他绑定没有效果，并且它对赋值之前绑定上所保存的值也没有任何效果。因此，函数

^① 一些由语言本身所定义的关键常量并不遵循这一约定，包括但不限于`T`和`NIL`，这在偶尔有人想用`t`作为局部变量名时会很讨厌。另一个是`PI`，其含有最接近数学常量 π 的长浮点值。

^② 某些守旧的Lisp程序员喜欢使用`SETQ`对变量赋值，但现代风格倾向于将`SETF`用于所有的赋值操作。

```
(defun foo (x) (setf x 10))
```

中的`SETF`对于`foo`之外的任何值都没有效果。这个在`foo`被调用时所创建的绑定被设置到10，立即替换了作为参数传递的任何值，特别是在如下形式中：

```
(let ((y 20))
  (foo y)
  (print y))
```

将打印出20而不是10，因为传递给`foo`的`y`的值在该函数中变成了`x`的值，随后又被`SETF`设置成新值。

`SETF`也可用于依次对多个位置赋值。例如，与其像下面这样：

```
(setf x 1)
(setf y 2)
```

也可写成：

```
(setf x 1 y 2)
```

`SETF`返回最近被赋予的值，因此也可以像下面的表达式那样嵌套调用`SETF`，将`x`和`y`赋予同一个随机值：

```
(setf x (setf y (random 10)))
```

6.6 广义赋值

当然，变量绑定并不是唯一可以保留值的位置，Common Lisp还支持复合数据结构，包括数组、哈希表、列表以及由用户定义的数据结构，所有这些都含有多个可用来保存值的位置。

后续章节里将讨论那些数据结构，但就目前所讨论的赋值主题而言，你应该知道`SETF`可以为任何位置赋值。当描述不同的复合数据结构时，我将指出哪些函数可以作为`SETF`的“位置”来使用。总之，如果需要对位置赋值，那么几乎肯定要用到`SETF`。虽然在此不予介绍，但`SETF`经拓展后甚至可为由用户定义的位置赋值。^①

从这个角度来说，`SETF`和多数源自C的语言中的赋值操作符没有区别。在那些语言里，`=`操作符可以将新值赋给变量、数组元素和类的字段。在诸如Perl和Python这类支持哈希表作为内置数据类型的语言里，`=`也可以设置哈希表项的值。表6-1总结了`=`在那些语言里的不同用法。

表6-1 `=` 在其他语言中的用法

赋值对象	Java, C, C++	Perl	Python
变量	<code>x = 10;</code>	<code>\$x = 10;</code>	<code>x = 10</code>
数组元素	<code>a[0] = 10;</code>	<code>\$a[0] = 10;</code>	<code>a[0] = 10</code>
哈希表项		<code>\$hash{'key'} = 10;</code>	<code>hash['key'] = 10</code>
对象字段	<code>o.field = 10;</code>	<code>\$o->{'field'} = 10;</code>	<code>o.field = 10</code>

① 查看`DEFSETF`和`DEFINE-SETF-EXPANDER`以获取进一步的信息。

SETF以同样的方式工作——**SETF**的第一个参数用来保存值的位置，而第二个参数提供了值。和这些语言中的=操作符一样，你可以使用和正常获取其值相同的形式来表达位置。^①因此，表6-1中赋值语句的Lisp等价形式分别为：**AREF**是数组访问函数，**GETHASH**做哈希表查找，而**field**可能是一个访问某用户定义对象中名为**field**的成员的函数。如下所示：

```
Simple variable: (setf x 10)
Array:           (setf (aref a 0) 10)
Hash table:     (setf (gethash 'key hash) 10)
Slot named 'field': (setf (field o) 10)
```

注意，当用**SETF**对一个作为更大对象一部分的位置进行赋值时，与赋值一个变量具有相同的语义：被修改的位置对之前保存在该位置上的对象没有任何影响。再次说明，这跟=在Java、Perl和Python中的行为非常相似。^②

6.7 其他修改位置的方式

尽管所有的赋值都可以用**SETF**来表达，但有些固定模式（比如像基于当前值来赋予新值）由于经常使用，因此有它们自己的操作符。例如，尽管可以像这样使用**SETF**来递增一个数：

```
(setf x (+ x 1))
```

或是像这样来递减它：

```
(setf x (- x 1))
```

但跟C风格的`++x`和`--x`相比就显得很冗长了。相反，可以使用宏**INCF**和**DECF**，它们以默认为1的特定数量对一个位置的值进行递增和递减。

```
(incf x)    ≡ (setf x (+ x 1))
(decf x)   ≡ (setf x (- x 1))
(incf x 10) ≡ (setf x (+ x 10))
```

类似**INCF**和**DECF**这种宏称为修改宏（modify macro），修改宏是建立在**SETF**之上的宏，其基于作用位置上的当前值来赋予该位置一个新值。修改宏的主要好处是，它们比用**SETF**写出的同样的修改语句更加简洁。另外，修改宏所定义的方式使其可以安全地用于那些表达式必须只被求值一次的位置。在下面这个表达式中，**INCF**会递增一个数组中任意元素的值，这个例子确实很可笑：

```
(incf (aref *array* (random (length *array*))))
```

一个到**SETF**表达式的幼稚转换方法可能看起来像这样：

- ① 源自Algol：使用=左边的“位置”和右边的新值进行赋值的语法得到广泛使用，由此产生了术语“左值”（lvalue，left value的缩写），意思是可被赋值的某种东西，以及“右值”（rvalue），意思是某种可以提供值的东西。一个编译器黑客将会说，“SETF将其第一个参数视为左值”。
- ② C程序员可能将变量和其他位置看成是保存了实际对象的指针。对一个变量赋值简单地改变了其所指向的对象，而对一个复合对象中的一部分赋值，则类似于重定向通向实际对象的指针。C++程序员应该注意到在C++中当处理对象时，确切地说，在进行成员复制时，=的行为是相当特异的。

```
(setf (aref *array* (random (length *array*)))
      (1+ (aref *array* (random (length *array*)))))
```

但它不会正常工作，因为两次对**RANDOM**的调用不一定能返回相同的值——该表达式将很可能抓取数组中一个元素的值，将其递增，然后将其作为新值保存到另一个不同的数组元素上。与之相比，上面的**INCF**表达式却能产生正确的行为，因为它知道如何处理这个表达式：

```
(aref *array* (random (length *array*)))
```

取出其中可能带有副作用的部分，从而确保它们仅被求值一次。在本例中，经展开后，它差不多会等价于以下形式。

```
(let ((tmp (random (length *array*))))
  (setf (aref *array* tmp) (1+ (aref *array* tmp))))
```

一般而言，修改宏可以保证以从左到右的顺序，对它们的参数和位置形式的子形式每个只求值一次。

在第3章那个微型数据库中曾用来向*db*变量添加元素的宏**PUSH**则是另一个修改宏。第12章在讲到如何在Lisp中表示列表时会详细地介绍它及其对应的**POP**和**PUSHNEW**是如何工作的。

最后有两个稍微有些难懂但很有用的修改宏，它们是**ROTATEF**和**SHIFTF**。**ROTATEF**在位置之间轮换它们的值。如果有两个变量a和b，那么如下调用

```
(rotatef a b)
```

将交换两个变量的值并返回**NIL**。由于a和b是变量并且不需要担心副作用，因此前面的**ROTATEF**的表达式将等价于如下形式：

```
(let ((tmp a)) (setf a b b tmp) nil)
```

对于其他不同类型的位置，使用**SETF**的等价表达式将会更加复杂一些。

SHIFTF与之相似，除了它将值向左侧移动而不是轮换它们——最后一个参数提供的值移动到倒数第二个参数上，而其他的值将向左移动一个，第一个参数的最初值将被简单地返回。这样，

```
(shiftf a b 10)
```

将等价于如下形式。同样，不必担心副作用。

```
(let ((tmp a)) (setf a b b 10) tmp)
```

ROTATEF和**SHIFTF**都可被用于任意多个参数，并且和所有的修改宏一样，它们可以保证以从左到右的顺序对每个参数仅求值一次。

学完了Common Lisp函数和变量的基础知识以后，下面将开始介绍一个令Lisp始终区别于其他语言的特性：宏。

宏：标准控制构造

尽

管起源于Lisp的许多编程思想，从条件表达式到垃圾收集，都已经被吸取进其他语言，但Lisp的宏系统却始终使它保持了在语言风格上的独特性。遗憾的是，宏这个字虽然在计算领域可以描述很多东西，但和Common Lisp的宏相比，它们仅大致相似。当Lisp程序员们试图向非Lisp程序员解释宏这种特性的伟大之处时，这种相似性导致了无休止的误解。^①要想理解Lisp的宏，就真的需要重新看待它，不能带有任何基于其他碰巧也叫做宏的概念所带来的成见。现在先退一步，从观察各种语言支持扩展的不同方式讨论Lisp宏。

所有的程序员应该都熟知这么一种观点，语言的定义可能会包含一个借由“核心”语言实现的标准功能库——如果某些功能没有定义在标准库中，那么它们可能已经被程序员实现在语言中了。只要它还没有被定义成标准库的一部分，那么这些功能就可以被任何程序员在语言之上实现。例如，C的标准库就差不多可以完全用可移植的C来实现。类似地，Java的标准Java开发包（JDK）中所提供的不断改进的类和接口集合也是用“纯”Java编写的。

使用核心加上标准库的方式来定义语言的优势在于易于理解和实现。但真正的好处在于其可表达性——由于所认为的“语言”很大程度上其实是一个库，因此很容易对其进行扩展。如果C语言中不含有所需的用来做某件事的一个函数，那就可以写出这个函数，然后就得到了一个特性稍微丰富一点的C版本。类似地，在诸如Java或Smalltalk这类几乎所有有趣部分都是由类来定义的语言里，通过定义新的类就可以扩展该语言，使其更适用于编写你正试图编写的任何程序。

尽管Common Lisp支持所有这些扩展语言的方法，但宏还提供了另一种方式。如同第4章所述，每个宏都定义了自己的语法，它们能够决定那些被传递的S-表达式如何转换成Lisp形式。核心语言有了宏，就有可能构造出新的语法，诸如WHEN、DOLIST和LOOP这样的控制构造以及DEFUN和DEFPARAMETER这样的定义形式，从而使这些新语法可以作为“标准库”的一部分而不是将其硬编码到语言核心。这已经牵涉到语言本身是如何实现的，但作为一个Lisp程序员，你更关心的将是它所提供的另一种语言扩展方式，而这将使Common Lisp成为更好的用于表达特定编程问题

^① 想要了解这类误解，可以在相对长期的Usenet新闻组上，以macro为主题，在comp.lang.lisp和comp.lang.*之间交叉投递的讨论中搜索。一个大致的说法如下：

Lisp支持者：“Lisp是最强的，因为它有宏！”

其他语言支持者：“你认为Lisp好是因为它的宏吗？！但宏是可怕和有害的，因此Lisp也一定是可怕和有害的。”

解决方案的语言。

那么，利用另一种方式来拓展语言的好处似乎是显而易见的。但出于某些原因，对于大量没有实际使用过Lisp宏的人，他们可以为了解决编程问题而日复一日地去创建新的函数型抽象或定义类的层次体系，却被这种可以定义新的句法抽象的思想给吓到了。通常，这种宏恐惧症的原因多半是来自学习其他“宏”系统时的不良经历。简单地对未知事物的恐惧无疑也是其中一部分原因。为了避免触发任何宏恐惧症反应，我们将从Common Lisp所定义的几种标准控制构造宏开始讨论，既而缓慢进入该主题。它们都是那些如果Lisp没有宏，就必须构造在语言核心里的东西。使用它们时，你不必在意它们是作为宏实现的，尽管如此，它们的确可以很好地展示出宏的一些功用。^①下一章将说明如何定义你自己的宏。

7.1 WHEN 和 UNLESS

如前所述，最基本的条件执行形式是由`IF`特殊操作符提供的，其基本形式是：如果`x`成立，那么执行`y`；否则执行`z`。

```
(if condition then-form [else-form])
```

`condition`被求值，如果其值非`NIL`，那么`then-form`会被求值并返回其结果。否则，如果有`else-form`，它将被求值并返回其结果。如果`condition`是`NIL`并且没有`else-form`，那么`IF`返回`NIL`。

```
(if (> 2 3) "Yup" "Nope") → "Nope"
(if (> 2 3) "Yup") → NIL
(if (> 3 2) "Yup" "Nope") → "Yup"
```

尽管如此，`IF`事实上并不是什么伟大的句法构造，因为每个`then-form`和`else-form`都被限制必须是单一的Lisp形式。这意味着如果想在每个子句中执行一系列操作，则必须将其用其他一些语法形式进行封装。举个例子，假如在一个垃圾过滤程序中，当一个消息是垃圾时，你想要在将其标记为垃圾的同时更新垃圾数据库，那么你不能这样写：

```
(if (spam-p current-message)
  (file-in-spam-folder current-message)
  (update-spam-database current-message))
```

因为对`update-spam-database`的调用将被作为`else`子句来看待，而不是`then`子句的一部分。另一个特殊操作符`PROGN`可以按顺序执行任意数量的形式并返回最后一个形式的值。因此可以通过写成下面这样来得到预想的行为：

```
(if (spam-p current-message)
  (progn
    (file-in-spam-folder current-message)
    (update-spam-database current-message)))
```

^① 另一个重要的用宏来定义的语言构造类别是所有的定义性构造，诸如`DEFUN`、`DEFPARAMETER`以及`DEFVAR`。在第24章里你将定义自己的定义性宏，从而可以简洁地编写可以用来读写二进制数据的代码。

这样做并不算太坏。但假如不得不多次使用这样的写法，不难想象你将在一段时间以后开始厌倦它。你可能会自问：“为什么Lisp没有提供一种方式来做我真正想做的事，也就是说，‘当x为真时，做这个、那个以及其他一些事情’？”换句话说，你很快将注意到这种由IF加上PROGN所组成的模式，并且希望可以有一种方式来抽象所有细节而不是每次都写出来。

这正是宏所能够提供的功能。在这个案例中，Common Lisp提供了一个标准宏WHEN，可以让你写成这样：

```
(when (spam-p current-message)
  (file-in-spam-folder current-message)
  (update-spam-database current-message))
```

如果它没有被内置到标准库中，你也可以像下面这样用一个宏来自定义WHEN，这里用到了第3章中讨论过的反引号：^①

```
(defmacro when (condition &rest body)
  ` (if ,condition (progn ,@body)))
```

与WHEN宏同系列的另一个宏是UNLESS，它取相反的条件，只有当条件为假时才求值其形式体。换句话说：

```
(defmacro unless (condition &rest body)
  ` (if (not ,condition) (progn ,@body)))
```

必须承认，这些都是相当简单的宏。这里没有什么高深的道理，它们只是抽象掉了一些语言层面约定俗成的细节，从而允许你更加清晰地表达你的真实意图。它们的极度简单性产生了一个重要的观点：由于宏系统是直接构建在语言之中的，所以可以写出像WHEN和UNLESS这样简单的宏来获得小而重要的清晰性，并随后通过不断地使用而无限放大。第24、26和31章将展现宏是如何被更大规模地用于创建完整的特定领域的嵌入式语言。首先来介绍一下标准控制构造宏。

7

7.2 COND

当遇到多重分支的条件语句时，原始的IF表达式再一次变得丑陋不堪：如果a成立那么执行x，否则如果b成立那么执行y；否则执行z。只用IF来写这样的条件表达式链并没有逻辑问题，只是不太好看。

```
(if a
  (do-x)
  (if b
    (do-y)
    (do-z)))
```

如果需要在then子句中包括多个形式，那就需要用到PROGN，而那样事情就会变得更糟。因此毫不奇怪，Common Lisp提供了一个用于表达多重分支条件的宏COND。下面是它的基本结构：

^① 其实不能将这个定义输入到Lisp中，因为重定义WHEN所在的COMMON-LISP包中的名字是非法的。如果你真想写出这样一个宏，则需要改变名字，例如my-when。

```
(cond
  (test-1 form*)
  .
  .
  .
  (test-N form*))
```

主体中的每个元素都代表一个条件分支，并由一个列表所构成，列表中含有一个条件形式，以及零或多个当该分支被选择时将被求值的形式。这些条件形式按照分支在主体中出现的顺序被依次求值，直到它们中的一个求值为真。这时，该分支中的其余形式将被求值，且分支中最后一个形式的值将作为整个COND的返回值。如果该分支在条件形式之后不再含有其他形式，那么就将返回该条件形式的值。习惯上，那个用来表示if/else-if链中最后一个else子句的分支将被写成带有条件T。虽然任何非NIL的值都可以使用，但在阅读代码时，T标记确实有用。这样就可以像下面这样用COND来写出前面的嵌套IF表达式：

```
(cond (a (do-x))
      (b (do-y))
      (t (do-z)))
```

7.3 AND、OR 和 NOT

在使用IF、WHEN、UNLESS和COND形式编写条件语句时，经常用到的三个操作符是布尔逻辑操作符AND、OR和NOT。

严格来讲，NOT这个函数并不属于本章讨论范畴，但它跟AND和OR紧密相关。它接受单一参数并对其真值取反，当参数为NIL时返回T，否则返回NIL。

AND和OR则是宏。它们实现了对任意数量子表达式的逻辑合取和析取操作，并被定义成宏以便支持“短路”特性。也就是说，它们仅以从左到右的顺序对用于检测整体真值的必要数量的子表达式进行求值。这样，只要AND的一个子表达式求值为NIL，它就立即停止并返回NIL。如果有所有子表达式都求值到非NIL，那么它将返回最后一个子表达式的值。而对于OR来说，只要一个子表达式求值到非NIL，它就立即停止并返回当前子表达式的值。如果没有子表达式求值到真，OR返回NIL。下面是一些例子：

(not nil)	→ T
(not (= 1 1))	→ NIL
(and (= 1 2) (= 3 3))	→ NIL
(or (= 1 2) (= 3 3))	→ T

7.4 循环

循环构造是另外一类主要的控制构造。Common Lisp的循环机制，除了更加强大和灵活以外，还是一门关于宏所提供的“鱼和熊掌兼得”的编程风格的有趣课程。

初看起来，Lisp的25个特殊操作符中没有一个能够直接支持结构化循环，所有的Lisp循环控

制构造都是构建在一对提供原生goto机制的特殊操作符之上的宏。^①和许多好的抽象或句法一样，Lisp的循环宏构建在以那两个特殊操作符为基础的一组分层抽象之上。

底层（不考虑特殊操作符）是一个非常通用的循环构造DO。尽管非常强大，但DO和许多其他的通用抽象一样，在应用于简单情形时显得过于复杂。因此Lisp还提供了另外两个宏，DOLIST和DOTIMES。它们不像DO那样灵活，却提供了对于常见的在列表元素上循环和计数循环的便利支持。尽管一个实现可以用任何方式来实现这些宏，但它们被典型实现为展开到等价DO循环的宏。因此，在由Common Lisp特殊操作符所提供的底层原语之上，DO提供了一种基本的结构化循环构造，而DOLIST和DOTIMES则提供了两种易用却不那么通用的构造。并且如同在下一章将看到的那样，对于那些DOLIST和DOTIMES无法满足需要的情形，还可以在DO之上构建自定义的循环构造。

最后，LOOP宏提供了一种成熟的微型语言，它用一种非Lisp的类似英语（或至少类似Algol）的语言来表达循环构造。一些Lisp黑客热爱LOOP，其他人则讨厌它。LOOP爱好者们喜欢它是因为它用了一种简洁的方式来表达特定的常用循环构造。而贬低者们不喜欢它则是因为它不太像Lisp。无论你倾向于哪一方，LOOP本身都是为语言增加新构造的宏展示其强大威力的突出示例。

7

7.5 DOLIST 和 DOTIMES

先从易于使用的DOLIST和DOTIMES宏开始。

DOLIST在一个列表的元素上循环操作，使用一个依次持有列表中所有后继元素的变量来执行循环体。^②下面是其基本形式（去掉了一些比较难懂的选项）：

```
(dolist (var list-form)
  body-form*)
```

当循环开始时，list-form被求值一次以产生一个列表。然后循环体在列表的每一项上求值一次，同时用变量var保存当前项的值。例如：

```
CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
```

在这种方式下，DOLIST这种形式本身求值为NIL。

^① 为了满足你的好奇心，这两个特殊操作符是TAGBODY和GO。现在无需讨论它们，第20章会介绍它们。

^② DOLIST类似于Perl的foreach或Python的for。Java从Java 1.5开始作为JSR-201的一部分，增加了一个类似的增强型循环构造。注意到宏所带来的区别：一个注意到代码中常见模式的Lisp程序员可以写出一个宏来获得对该模式的源代码级抽象。一个注意到同样模式的Java程序员只能建议Sun说，这种特定的抽象值得添加到语言之中，然后Sun将会发布一个JSR并组织一个业界专家组来推敲其细节。按照Sun的说法，这一过程平均耗时18个月。在那之后，所有的编译器作者都将升级其编译器以支持新的特性，并且就算那个Java程序员所喜爱的编译器支持了这个新版本的Java，他们也仍然可能无法使用这个新特性，直到被允许打破与旧版本Java的源代码级兼容性。因此，一个Common Lisp程序员在五分钟里可以自行解决的麻烦问题却会困扰Java程序员几年时间。

如果想在列表结束之前中断一个**DOLIST**循环，则可以使用**RETURN**。

```
CL-USER> (dolist (x '(1 2 3)) (print x) (if (evenp x) (return)))
1
2
NIL
```

DOTIMES是用于循环计数的高级循环构造，其基本模板和**DOLIST**非常相似。

```
(dotimes (var count-form)
  body-form*)
```

其中的*count-form*必须要能求值为一个整数。通过每次循环，*var*所持有的整数依次为从0到比那个数小1的每一个后继整数。例如：

```
CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

和**DOLIST**一样，也可以使用**RETURN**来提前中断循环。

由于**DOLIST**和**DOTIMES**的循环体中可以包含任何类型的表达式，因此也可以使用嵌套循环。例如，为了打印出从 $1 \times 1 = 1$ 到 $20 \times 20 = 400$ 的乘法表，可以写出下面这对嵌套的**DOTIMES**循环：

```
(dotimes (x 20)
  (dotimes (y 20)
    (format t "~3d " (* (1+ x) (1+ y))))
  (format t "~%"))
```

7.6 DO

尽管**DOLIST**和**DOTIMES**方便且易于使用，却没有灵活到可用于所有循环。例如，如果想并行循环多个变量该怎样做？要是使用任意表达式来测试循环的末尾呢？如果**DOLIST**和**DOTIMES**都不能满足需求，那还可以用更通用的**DO**循环。

与**DOLIST**和**DOTIMES**只提供一个循环变量有所不同的是，**DO**允许绑定任意数量的变量，并且变量值在每次循环中的改变方式也是完全可控的也可以定义测试条件来决定何时终止循环，并可以提供一个形式，在循环结束时进行求值来为**DO**表达式整体生成一个返回值。基本模板如下所示。

```
(do (variable-definition*)
  (end-test-form result-form*)
  statement*)
```

每一个*variable-definition*引入了一个将存在于循环体作用域之内的变量。单一变量定义的完整形式是一个含有三个元素的列表。

```
(var init-form step-form)
```

上述*init-form*将在循环开始时被求值并将结果值绑定到变量*var*上。在循环的每一个后续

迭代开始之前，*step-form*将被求值并把新值分配给*var*。*step-form*是可选的，如果它没有给出，那么变量将在迭代过程中保持其值不变，除非在循环体中显式地为其赋予新值。和LET中的变量定义一样，如果*init-form*没有给出，那么变量将被绑定到NIL。另外和LET的情形一样的是，你可以将一个只含有名字的列表简化成一个简单的变量名来使用。

在每次迭代开始时以及所有循环变量都被指定新值后，*end-test-form*会被求值。只要其值为NIL，迭代过程就会继续，依次求值所有的*statement*。

当*end-test-form*求值为真时，*result-form*（结果形式）将被求值，且最后一个结果形式的值将被作为DO表达式的值返回。

在迭代的每一步里，所有变量的*step-form*（步长形式）将在分配任何值给变量之前被求值。这意味着可以在步长形式里引用其他任何循环变量。^①比如在下列循环中。

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    (= 10 n) cur))
```

其步长形式(1+ n)、next和(+ cur next)均使用n、cur和next的旧值来求值。只有当所有步长形式都被求值以后，这些变量才被指定其新的值。（有数学天赋的读者可能会注意到，这其实是一种计算第11个斐波那契数的特别有效的方式。）

这个例子还阐述了DO的另一种特征——由于可以同时推进多个变量，所以往往根本不需要一个循环体。其他时候，尤其在只是把循环用作控制构造时，则可能会省略结果形式。尽管如此，这种灵活性正是DO表达式有点儿晦涩难懂的原因。所有这些括号都该放在哪里？理解一个DO表达式的最佳方式是记住其基本模板：

```
(do (variable-definition*)
    (end-test-form result-form*)
    statement*)
```

该模板中的六个括号是DO结构本身所必需的。一对括号来围住变量声明，一对用来围住终止测试形式和结果形式，以及一对用来围住整个表达式。DO中的其他形式可能需要它们自己的括号，例如变量定义总是以列表形式存在，而测试形式则通常是一个函数调用。不过DO循环的框架将总是一致的。下面是一些框架用黑体表示的DO循环的例子。

```
(do ((i 0 (1+ i)))
    ((>= i 4))
    (print i))
```

注意，该例的结果形式被省略了。不过这种用法对DO来说没有特别意义，因为用DOTIMES来写这个循环会更简单。^②

```
(dotimes (i 4) (print i))
```

^① 一个DO的变体DO*，它会在求值后续变量的步长形式之前为每个变量赋值。关于它的更多细节，请查阅你喜爱的Common Lisp参考书。

^② 另一个推荐使用DOTIMES的理由是，其宏展开将可以包含允许编译器生成更有效代码的类型声明。

另一个例子是一个没有循环体的斐波那契数计算循环：

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

最后，下面循环演示了一个不绑定变量的DO循环。在当前时间小于一个全局变量值的时候，它保持循环，每分钟打印一个“Waiting”。注意，就算没有循环变量，仍然需要有那个空变量列表。

```
(do ()
  ((> (get-universal-time) *some-future-date*)
   (format t "Waiting~%")
   (sleep 60)))
```

7.7 强大的 LOOP

简单的情形可以使用DOLIST和DOTIMES。但如果它们不符合需要，就需要退而使用完全通用的DO。不然还能怎样？

然而，结果是有少量的循环用法一次又一次地产生出来，例如在多种数据结构上的循环：列表、向量、哈希表和包，或是在循环时以多种方式来聚集值：收集、计数、求和、最小化和最大化。如果需要用宏来做其中的一件事（或同时几件），那么LOOP宏可以提供一种更容易表达的方式。

LOOP宏事实上有两大类——简化的和扩展的。简化的版本极其简单，就是一个不绑定任何变量的无限循环。其框架看起来像这样：

```
(loop
  body-form*)
```

主体形式在每次通过循环时都将被求值，整个循环将不停地迭代，直到使用RETURN来进行中止。例如，可以使用一个简化的LOOP来写出前面的DO循环：

```
(loop
  (when (> (get-universal-time) *some-future-date*)
    (return))
  (format t "Waiting ~%")
  (sleep 60)))
```

而扩展的LOOP则是完全不同的庞然大物。值得注意的是，并非所有的Lisp程序员都喜爱扩展的LOOP语言。至少一位Common Lisp的最初设计者就很讨厌它。LOOP的贬低者们抱怨它的语法是完全非Lisp化的（换句话说，没有足够的括号）。LOOP的爱好者们则反驳说，问题在于复杂的循环构造，如果不将它们用DO那晦涩语法包装起来，它们将难于被人理解。所以他们认为最好用一种稍显冗长的语法来提供某些逻辑线索。

例如，下面是一个地道的DO循环，它将把从1到10的数字收集到一个列表中：

```
(do ((nums nil) (i 1 (1+ i))))
```

```
((> i 10) (nreverse nums))
(push i nums)) → (1 2 3 4 5 6 7 8 9 10)
```

一个经验丰富的Lisp程序员将毫不费力地理解这些代码——只要理解一个DO循环的基本形式并且认识用于构建列表的PUSH/NREVERSE用法就可以了。但它并不是很直观。而它的LOOP版本理解起来就几乎可以像一个英语句子那样简单。

```
(loop for i from 1 to 10 collecting i) → (1 2 3 4 5 6 7 8 9 10)
```

接下来是一些关于LOOP简单用法的例子。下例可以对前十个平方数求和：

```
(loop for x from 1 to 10 summing (expt x 2)) → 385
```

这个用来统计一个字符串中元音字母的个数：

```
(loop for x across "the quick brown fox jumps over the lazy dog"
      counting (find x "aeiou")) → 11
```

下面这个例子用来计算第11个斐波那契数，它类似于前面使用DO循环的版本：

```
(loop for i below 10
      and a = 0 then b
      and b = 1 then (+ b a)
      finally (return a))
```

符号across、and、below、collecting、counting、finally、for、from、summing、then和to都是一些循环关键字，它们的存在表明当前正在使用扩展的LOOP。^①

第22章将介绍LOOP的细节，但目前值得注意的是，我们通过它可以再次看到，宏是如何被用于扩展基本语言的。尽管LOOP提供了它自己的语言用来表达循环构造，但它并没有抹杀Lisp的其他优势。虽然循环关键字是按照循环的语法来解析的，但一个LOOP中的其余代码都是正常的Lisp代码。

另外，值得再次指出的是，尽管LOOP宏相比诸如WHEN或者UNLESS这样的宏复杂了许多，但它也只是一个宏而已。如果它没有被包括在标准库之中，你也可以自己实现它或是借助一个第三方库来实现它。

以上就是我们对基本控制构造宏的介绍。现在可以进一步了解如何定义自己的宏了。

^① 循环关键字容易让人误解的一点在于它们不是关键字符。事实上，LOOP并不关心这些字符来自什么包。当LOOP宏解析其主体时，它将等价地考察任何适当命名的字符。如果你想的话，甚至可以使用诸如:for和:across这些真正的关键字，因为它们也有正确的名字。但多数人只用普通字符。由于循环关键字仅被用作句法标记，因此将它们用做其他目的，如作为函数或变量的名字，也是没有关系的。



第8章

如何自定义宏

8

现在可以开始编写自己的宏了。前一章里提及的标准宏暗示了可以用宏做到的某些事情，但这只是开始。

相比于C语言的函数可以让每个C程序员编写C标准库中的函数的简单变体，Common Lisp的宏也无非是可以让每个Lisp程序员创建他们自己的标准控制构造变体罢了。作为语言的一部分，宏能够用于在核心语言和标准库之上创建抽象，从而使你更直接地表达想表达的事物。

具有讽刺意义的是，也许对于宏的正确理解，最大的障碍是它们已经很好地集成到了语言里。在许多方面，它们看起来只是一些有趣的函数——它们用Lisp写成，接受参数并返回结果。同时，它们允许你抽象那些分散注意力的细节。尽管有这些相似性，但宏的操作层面与函数不同，而且它还有着完全不同类型的抽象。

一旦理解了宏与函数之间的区别，你就会发现这门语言中宏的紧密集成所带来的巨大优势。但同时它也是经常导致新程序员困惑的主要原因。下面来讲个故事，尽管从历史或技术意义上来说并不真实，然而通过这种方式，你倒是可以思考一下宏的工作方式，以此来缓解一下困惑。

8.1 Mac 的故事：只是一个故事

很久以前，有一个由Lisp程序员们所组成的公司。那个年代相当久远，所以Lisp还没有宏。每次，任何不能用函数来定义或是用特殊操作符来完成的事情都不得不完全通过手写来实现，这带来了很大的不便。不幸的是，这个公司的程序员们虽然杰出却非常懒惰。在他们的程序中，当需要编写大量单调乏味的代码时，他们往往会写下一个注释来描述想要在该位置上编写的代码。更不幸的是，由于很懒惰，他们也很讨厌回过头去实际编写那些注释所描述的代码。不久，这个公司就有了一大堆无法运行的程序，因为它们全都是代表着尚需编写代码的注释。

走投无路之下，老板雇用了一个初级程序员Mac。他的工作就是找到这些注释，编写所需的代码，然后再用其替换掉程序中的注释。Mac从未运行过这些程序——程序尚未完成，他当然运行不了。但就算这些程序完成了，Mac也不知道该用怎样的输入来运行它们。因此，他只是基于注释的内容来编写他的代码，再将其发还给最初的程序员。

在Mac的帮助下，不久之后，所有的程序都完成了，公司通过销售它们赚了很多钱，并用这些钱将其程序员团队扩大了一倍。但不知为何，没有人想到要雇用其他人来帮助Mac。很快他就

开始单枪匹马地同时协助几十个程序员了。为了避免将他所有的时间都花在搜索源代码的注释上，Mac对程序员们使用的编译器做了一个小小的更改。从那以后，只要编译器遇到一个注释，它就会将注释以电子邮件的形式发给他并等待他将替换的代码传送回来。然而，就算有了这个变化，Mac也很难跟上程序员的进度。他尽可能小心地工作，但有时，尤其是当注释不够清楚时，他会犯错误。

不过程序员们注意到了，他们将注释写得越精确，Mac就越有可能发回正确的代码。一天，一个花费大量时间用文字来描述他想要的代码的程序员，在他的注释里写入了一个可以生成他想要的代码的Lisp程序。这对Mac来说很简单；他只需运行这个程序并将结果发给编译器就好了。

接下来又出现了一种创新。有一个程序员在他程序的开始处写了一段备注，其中含有一个函数定义以及另一个注释，该注释为：“Mac，不要在这里写任何代码，但要把这个函数留给以后使用，我将在我的其他一些注释里用到它。”同一个程序里还有如下的注释：“Mac，将这个注释替换成用符号x和y作为参数来运行上面提到的那个函数所得到的结果。”

这项技术在几天里就迅速流行起来，多数程序都含有数十个注释，它们定义了那些只被其他注释中的代码所使用的函数。为了使Mac更容易地辨别那些只含有定义而不必立即回复的注释，程序员们用一个标准前缀来标记它们：“给Mac的定义，仅供阅读。”(Definition for Mac, Read Only.) 由于程序员们仍然很懒惰，这个写法很快简化成“DEF. MAC. R/O”，接着又被简化为“DEFMACRO”。

不久以后，这些给Mac的注释中再没有实际可读的英语了。Mac每天要做的事情就是阅读并反馈那些来自编译器的含有DEFMACRO注释的电子邮件，以及调用那些DEFMACRO里所定义的函数。由于注释中的Lisp程序做了所有实际的工作，跟上这些电子邮件的进度完全没有问题。Mac手上突然有了大量时间，可以坐在他的办公室里做那些关于白色沙滩、蓝色海水和鸡尾酒的白日梦了。

几个月以后，程序员们意识到已经很长时间没人见过Mac了。当他们去他的办公室时，发现所有东西上都积了薄薄的一层灰，一个桌子上还放着几本热带地区的旅行手册，而电脑则是关着的。但是编译器仍在正常工作——这怎么可能？看起来Mac对编译器做了最后一个修改：现在不需要用电子邮件将注释发给Mac了，编译器会将那些DEFMACRO中所定义的函数保存下来，并在其被其他注释调用时运行它们。程序员们觉得没有理由告诉老板Mac不再来办公室了。因此直到今天，Mac还领着薪水，并且时不时地会从某个热带地区给程序员们发一张明信片。

8.2 宏展开期和运行期

理解宏的关键在于必须清楚地知道那些生成代码的代码(宏)和那些最终构成程序的代码(所有其他内容)之间的区别。当编写宏时，你是在编写那些将被编译器用来生成代码并随后编译的程序。只有当所有的宏都被完全展开并且产生的代码被编译后，程序才可以实际运行。宏运行的时期被称为宏展开期(macro expansion time)，这和运行期(runtime)是不同的，后者是正常的代码(包括那些由宏生成的代码)实际运行的阶段。

牢记这一区别很重要，因为运行在宏展开期的代码与那些运行在运行期的代码相比，它们的运行环境完全不同。也就是说，在宏展开期无法访问那些仅存在于运行期的数据。正如Mac无法

运行他写的程序是因为不知道正确的输入那样，运行在宏展开期的代码也只能处理那些来自源代码本身的数据。例如，假设在程序的某个地方出现了下面这样的源代码：

```
(defun foo (x)
  (when (> x 10) (print 'big)))
```

正常情况下，你将x设为一个变量，用它保存传递给一个对foo调用的实参。但在宏展开期，比如当编译器正在运行WHEN宏的时候，唯一可用的数据就是源代码。由于程序尚未运行，没有对foo的调用，因此也没有值关联到x上。相反，编译器传递给WHEN的值只是代表源代码的Lisp列表，也即(> x 10)以及(print 'big)。假设WHEN确如前一章中所见的那样用类似下面的宏定义而成。

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

那么当foo中的代码被编译时，WHEN宏将以那两个形式作为实参来运行。形参condition会被绑定到形式(> x 10)上，而形式(print 'big)会被收集到一个列表中成为&rest body形参的值。那个反引用表达式将随后通过插入condition的值，并将body的值嵌入PROGN的主体来生成下面的代码：

```
(if (> x 10) (progn (print 'big)))
```

当Lisp被解释而非编译时，宏展开期和运行期之间的区别不甚明显，因为它们临时纠缠在一起。同样，语言标准并未规定解释器处理宏的具体方式——它可能在被解释的形式中展开所有的宏，然后解释执行那些宏所生成的代码，也可能是直接解释一个形式并在每次遇到宏的时候才展开。无论哪种情况，总是向宏传递那些代表宏形式中子形式的未经求值的Lisp对象，并且宏的作用仍然是生成做某些事情的代码，而非直接做任何事情。

8.3 DEFMACRO

如同你在第3章里所看到的那样，宏真的是用DEFMACRO来定义的。当然，它代表的是“定义宏”(DEFine MACRO)而不是“给Mac的定义”(Definition for Mac)。DEFMACRO的基本框架和DEFUN框架很相似。

```
(defmacro name (parameter*)
  "Optional documentation string."
  body-form*)
```

和函数一样，宏由名字、形参列表、可选文档字符串以及Lisp表达式体所构成。^①但如前所述，宏并不是直接做事，它只是用于生成以后工作所需的代码。

宏可以使用Lisp的所有功能来生成其展开式，这意味着本章只能初步说明宏的具体功用。不过我可以描述一个通用的宏编写过程，它适用于从最简单到最复杂的所有宏。

宏的工作是将宏形式（首元素为宏名的Lisp形式）转化成做特定事情的代码。有时是从想要

^① 和函数一样，宏也可以含有声明，但你现在不需要考虑它们。

编写的代码开始来编写宏的, 就是说从一个示例的宏形式开始。其他时候则是在连续几次编写了相同的代码模式并认识到通过抽象该模式可以使代码更清晰后, 才开始决定编写宏的。

无论从哪一端开始, 你都需要在开始编写宏之前搞清楚另一端: 既需要知道从哪里开始, 又要知道正在向何处去, 然后才能期待编写代码来自动地做到这点。因此编写宏的第一步是至少应去编写一个宏调用的示例以及该调用应当展开成的代码。

一旦有了示例调用及预想的展开式, 那么就可以开始第二步了: 编写实际的宏代码。对于简单的宏来说, 这将极其轻松——编写一个反引用模板并将宏参数插入到正确的位置上。复杂的宏则会是一个庞大的独立程序, 它将带有配套的助手函数和数据结构。

在已经编写了代码来完成从示例调用到适当的展开式的转换以后, 需要确保宏所提供的抽象没有“泄漏”其实现细节。有漏洞的宏抽象将只适用于特定参数上, 或会以预想之外的方式与调用环境中的代码进行交互。后面将会看到, 宏只能以很少的几种方式泄漏, 而所有这些都是可以轻易避免的, 只要知道如何检查它们就行。8.7节将讨论具体的方法。

总结起来, 编写宏的步骤如下所示:

- (1) 编写示例的宏调用以及它应当展开成的代码, 反之亦然;
- (2) 编写从示例调用的参数中生成手写展开式的代码;
- (3) 确保宏抽象不产生“泄漏”。

8.4 示例宏: do-primes

8

为了观察这三步过程是怎样发挥作用的, 下面将编写一个宏do-primes, 它提供了一个类似DOTIMES和DOLIST的循环构造, 只是它并非迭代在整数或者一个列表的元素上, 而是迭代在相继的素数上。这并不是一个特别有用的宏, 它只是在演示该过程。

首先你需要两个工具函数: 一个用来测试给定的数是否为素数, 另一个用来返回大于或等于其实参的下一个素数。这两种情况都可以使用简单而低效的暴力手法来解决。

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number) never (zerop (mod number fac)))))

(defun next-prime (number)
  (loop for n from number when (primep n) return n))
```

现在就可以写这个宏了。按照前面所概括的过程, 至少需要一个宏调用示例以及它应当展开成的代码。假设你开始时想通过如下代码来表示循环:

```
(do-primes (p 0 19)
  (format t "~d " p))
```

这个循环在每个大于等于0并小于等于19的素数上分别依次执行循环体, 并以变量p保存当前素数。仿照标准的DOLIST和DOTIMES宏来定义是合理的。按照已有宏的模式操作的宏比那些引入了无谓的新颖语法的宏更易于理解和使用。

如果没有do-primes宏, 你可以用DO (和前面定义的两个工具函数) 来写出下面这个

循环：

```
(do ((p (next-prime 0) (next-prime (1+ p))))
    ((> p 19))
    (format t "~d ~d" p))
```

现在就可以开始编写将前者转化成后者的代码了。

8.5 宏形参

由于传递给宏的实参是代表宏调用源代码的Lisp对象，因此任何宏的第一步工作都是提取出那些对象中用于计算展开式的部分。对于那些简单地将其实参直接插入到模板中的宏而言，这一步骤相当简单：只需定义正确的形参来保存不同的实参就可以了。

但是这一方法似乎并不适用于do-primes。do-primes调用的第一个参数是一个列表，其含有循环变量的名字p及其下界0和上界19。但如果查看展开式就会发现，该列表作为整体并没有出现在展开式中，三个元素被拆分开并分别放在不同的位置上。

可以用两个形参来定义do-primes，一个用来保存该列表，另一个`&rest`形参保存形式体，然后手工分拆该列表，类似下面这样：

```
(defmacro do-primes (var-and-range &rest body)
  (let ((var (first var-and-range))
        (start (second var-and-range))
        (end (third var-and-range)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
         ,@body)))
```

稍后，将解释上述宏形式体怎样生成正确的展开式。目前只需注意变量var、start和end都持有一个从var-and-range中提取出的值，它们随后被插入到反引用表达式中以生成do-primes的展开式。

尽管如此，但并不需要“手工”分拆var-and-range，因为宏形参列表是所谓的解构(destructuring)形参列表。顾名思义，“解构”涉及分拆一个结构体，在本例中是传递给一个宏的列表结构形式。

在解构形参列表中，简单的形参名将被替换成嵌套的形参列表。嵌套形参列表中的形参将从绑定到该形参列表的表达式的元素中获得其值。例如可以将var-and-range替换成一个列表(var start end)，然后这个列表的三个元素将被自动解构到三个形参上。

宏形参列表的另一个特性是可以使用`&body`作为`&rest`的同义词。`&body`和`&rest`在语义上是等价的，但许多开发环境根据一个`&body`形参的存在来修改它们缩进那些使用该宏的代码的方式。通常，`&body`被用来保存一个构成该宏主体的形式的列表。

因此你可以通过将do-primes定义成下面这样来完成其定义，并同时向读者和你的开发工具说明它的用途：

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
```

```
((> ,var ,end))
  ,@body))
```



除了更加简洁以外，解构形参列表还可以让你自动检查错误。通过以这种方式定义 do-primes，Lisp 可以检测到那些首参数不是三元素列表的调用，并提供有意义的错误信息，就好像你用太多或太少的参数调用了一个函数那样。同样，在诸如 SLIME 这样的开发环境中，只要输入一个函数或宏的名字就可以指示它所期待的参数。如果你使用了一个解构形参列表，那么环境将可以更明确地告诉你宏调用的语法。使用最初的定义，SLIME 将告诉你 do-primes 可以像这样来调用：

```
(do-primes var-and-range &rest body)
```

但在新定义下，它可以告诉你一个调用应当看起来像这样：

```
(do-primes (var start end) &body body)
```

解构形参列表可以含有 `&optional`、`&key` 和 `&rest` 形参，并且可以含有嵌套的解构列表。尽管如此，你在编写 do-primes 的过程中却不需要任何这些选项。

8.6 生成展开式

由于 do-primes 是一个相当简单的宏，在解构了参数以后，剩下的就是将它们插入到一个模板中来得到展开式。

对于像 do-primes 这样简单的宏，反引用语法刚好合适。回顾一下，反引用表达式与引用表达式很相似，除了可以“解引用”（unquote）特定的值表达式，即前面加上逗号，可能其后还会接着一个“@”符号。没有这个“@”符号，逗号会导致子表达式的值被原样包含。有了“@”符号，其值（必须是一个列表）可被“拼接”到其所在的列表中。

采用另一种方式也会有助于理解反引用语法，这就是将其视为编写生成列表的代码的一种特别简洁的方式。这种理解方式的优点是可以相当明确地看到其表象之下实际发生的事——当读取器读到一个反引用表达式时，它将其翻译成生成适当列表结构的代码。例如，` (, a b) 可以被读取成 (list a 'b)。语言标准并未明确指定读取器必须产生怎样的代码，只要它能生成正确的列表结构就可以了。

表 8-1 给出了一些反引用表达式的范例，同时带有与之等价的列表构造代码以及其中任意一种形式的求值结果。^①

表 8-1 反引用表达式的例子

反引用语法	等价的列表构造代码	结 果
`(a (+ 1 2) c)	(list 'a '(+ 1 2) 'c)	(a (+ 1 2) c)
`(a ,(+ 1 2) c)	(list 'a (+ 1 2) 'c)	(a 3 c)
`(a (list 1 2) c)	(list 'a '(list 1 2) 'c)	(a (list 1 2) c)
`(a ,(list 1 2) c)	(list 'a (list 1 2) 'c)	(a (1 2) c)
`(a ,@(list 1 2) c)	(append (list 'a) (list 1 2) (list 'c))	(a 1 2 c)

^① APPEND 函数尚未提及，它能够接受任意数量的列表实参并返回一个由它们拼接而成的单独列表。

重要的是要注意反引用只是一种便利措施。不过这确实相当便利。为了说明究竟怎么便利，我们可以将do-primes的反引用版本和下面的版本作比较，后者使用了显式的列表构造代码：

```
(defmacro do-primes-a ((var start end) &body body)
  (append '(do)
    (list (list (list var
      (list 'next-prime start)
      (list 'next-prime (list '1+ var))))))
    (list (list (list '> var end)))
    body))
```

稍后即将看到，do-primes的当前实现并不能正确地处理特定的临界情况，但首先应当确认它至少应能适用于最初的例子。可以用两种方式来测试它。可以简单地通过使用来间接地测试，也就是说，如果结果的行为是正确的，那么展开式很可能就是正确的。例如，可以将do-primes最初的用例键入到REPL中，你会看到它确实打印出了正确的素数序列。

```
CL-USER> (do-primes (p 0 19) (format t "~d " p))
2 3 5 7 11 13 17 19
NIL
```

或者也可以通过查看特定调用的展开式来直接检查该宏。函数MACROEXPAND-1接受任何Lisp表达式作为参数并返回做宏展开一层的结果。^①由于MACROEXPAND-1是一个函数，所以为了传给它一个字面的宏形式，就必须引用它。可以用它来查看前面调用的展开式。^②

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
  ((> P 19))
  (FORMAT T "~d " P))
T
```

或者，在SLIME中也可以更方便地检查一个宏的展开式：将光标放置在源代码中一个宏形式的开括号上，并输入C-c RET来调用Emacs函数slime-macroexpand-1，后者将把宏调用传递到MACROEXPAND-1上，并将结果“美化输出”到一个临时缓冲区上。

无论怎样得到展开式，你都可以看到宏展开的结果和最初的手写展开式是一样的，因此看起来do-primes是有效的。

8.7 堵住漏洞

Jeff Spolsky在他的随笔“The Law of Leaky Abstractions”里创造了术语“有漏洞的抽象”(leaky abstraction)，以此来描述一种抽象：“泄露”了本该抽象的细节。由于编写宏是一种创造抽象的

① 另一个函数MACROEXPAND将持续地展开结果，只要返回的展开式的第一个元素是宏的名字，它就会不断进行下去。但它通常会深入展示代码行为，其程度往往远超出你所预期。因为诸如DO这类基本控制构造也被实现为宏。换句话说，尽管它对看到宏最终可展开成怎样的代码具有一定教育意义，但这对于了解宏的作用并不是一个有用的视角。

② 如果所有宏展开被显示在一行里，这很有可能是因为变量*PRINT-PRETTY*为NIL。如果是这样，求值(setf *print-pretty* t)将使展开式更容易阅读。

方式，故此需要确保宏不产生不必要的泄露。^①

如同即将看到的，宏可能以三种方式泄露其内部工作细节。幸运的是，你可以相当容易地看出一个给定的宏是否存在任何一种泄露方式，并修复它。

当前的宏定义存在三种可能的宏泄露中的一种，确切地说，它会过多地对end子形式求值。假设没有使用诸如19这样的字面数字，而是用像(random 100)这样的表达式在end的位置上来调用do-primes：

```
(do-primes (p 0 (random 100))
  (format t "~d " p))
```

假设这里的意图是要在从0到由(random 100)所返回的任意随机数字的范围内循环查找素数。但MACROEXPAND-1的结果显示这并不是当前实现所做的事。

```
CL-USER> (macroexpand-1 '(do-primes (p 0 (random 100)) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
  ((> P (RANDOM 100)))
  (FORMAT T "~d " P))
```

T

当我们运行展开式代码时，RANDOM将在每次进行循环的终止测试时被求值一次。这样，循环将不会在p大于一个初始给定的随机数时终止，而是在循环刚好生成一个小于或等于当前p值的随机数时，循环才会终止。由于循环的整体次数仍然是随机的，因此它将产生一个与RANDOM所返回的统一分布相当不同的分布形式。

这就是一种抽象中的漏洞，因为为了正确使用该宏，调用者必须注意end形式被求值超过一次的情况。一种堵住漏洞的方式是简单地将其定义成do-primes的行为。但这并不非常令人满意，你在实现宏时应当试图遵守最少惊动原则（Principle of Least Astonishment）。而且通常情况下，程序员们希望他们传递给宏的形式除非必要将不会被多次求值。^②更进一步，由于do-primes是构建在标准宏DOTIMES和DOLIST之上的，而这两个宏都不会导致其循环体之外的形式被多次求值，所以多数程序员将期待do-primes具有相似的行为。

修复多重求值问题是相当容易的：只需生成代码来对end求值一次，并将其值保存在一个稍后将会用到的变量里。回想在DO循环中，用一个初始形式但没有步长形式来定义的变量并不会在迭代过程中改变其值。因此可以用下列定义来修复多重求值问题：

```
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
        (,var (next-prime ,start) (next-prime (1+ ,var))))
    ((> ,var ending-value))
    ,@body))
```

8

^① 该随笔出自Joel Spolsky的*Joel on Software*，你也可以查阅<http://www.joelonsoftware.com/articles/LeakAbstractions.html>来获取相关内容。Spolsky在随笔中的观点是，所有的抽象都在某种意义上存在泄露。也就是说，不存在完美的解决方案。但这也不意味着你可以容忍那些可以轻易堵上的漏洞。

^② 当然，特定形式其本意就是被多次求值，例如一个do-primes循环体中的形式。

然而不幸的是，这一修复却又给宏抽象引入了两个新漏洞。

其中一个新漏洞类似于刚修复的多重求值漏洞。因为在DO循环中，变量的初始形式是以变量被定义的顺序来求值的，当宏展开被求值时，传递给end的表达式将在传递给start的表达式之前求值，这与它们出现在宏调用中的顺序相反。并在start和end都是像0和19这样的字面值时，这一泄露，不会带来任何问题。但当它们是可以产生副作用的形式时，不同的求值顺序将使它们再次违反最少惊动原则。

通过交换两个变量的定义顺序就可轻易堵上该漏洞。

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (ending-value ,end))
       ((> ,var ending-value))
       ,@body))
```

最后一个需要堵上的漏洞是由于使用了变量名ending-value而产生的。问题在于这个名字（其应当完全属于宏实现内部的细节）它可以跟传递给宏的代码或是宏被调用的上下文产生交互。下面这个看似无辜的do-primes调用会由于这个漏洞而无法正常工作。

```
(do-primes (ending-value 0 10)
            (print ending-value))
```

这样也不可以。

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

MACROEXPAND-1再次向你展示问题所在。第一次调用展开成这样。

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
      (ending-value 10))
    ((> ending-value ending-value))
    (print ending-value))
```

某些Lisp可能因为ending-value作为变量名在同一个DO循环中被用了两次而拒绝上面的代码。如果没有被完全拒绝，上述代码也将无限循环下去，由于ending-value永远不会大于其自身。

第二个问题调用展开成下面的代码：

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
        (ending-value 10))
    ((> p ending-value))
    (incf ending-value p))
  ending-value)
```

在这种情况下生成的代码是完全合法的，但其行为完全不是你想要的那样。由于在循环之外由LET所建立的ending-value绑定被DO内部同名的变量所掩盖，形式(incf ending-valuep)

将递增循环变量`ending-value`而不是同名的外层变量，因此得到了另一个无限循环。^①

很明显，为了补上这个漏洞，需要一个永远不会在宏展开代码之外被用到的符号。可以尝试使用一个真正罕用的名字，但即便如此也不可能做到万无一失。也可以使用第21章里介绍的包(`package`)，从而在某种意义上起到保护作用。但还有一个更好的解决方案。

函数`GENSYM`在其每次被调用时返回唯一的符号。这是一个没有被Lisp读取器读过的符号并且永远不会被读到，因为它不会进入到任何包里。因而就可以在每次`do-primes`被展开时生成一个新的符号以替代像`ending-value`这样的字面名称。

```
(defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
         ,@body)))
```

注意调用`GENSYM`的代码并不是展开式的一部分，它作为宏展开器的一部分来运行从而在每次宏被展开时创建一个新符号。这初看起来有一点奇怪——`ending-value-name`是一个变量，其值是另一个变量名。但其实它和值为一个变量名的形参`var`并没有什么区别，区别在于`var`的值是由读取器在宏调用被读取时创建的，而`ending-value-name`的值则是在宏代码运行时由程序化生成的。

使用这个定义，前面两个有问题的形式现在就可以展开成按预想方式运作的代码了。第一个形式

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

展开成下面的代码：

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
      (#:g2141 10))
    ((> ending-value #:g2141))
    (print ending-value))
```

现在用来保存循环终值的变量是生成符号，`#:g2141`。该符号的名字`G2141`是由`GENSYM`所生成的，但这并不重要，重要的是这个符号的对象标识。生成符号是以未保留符号通常的语法形式打印出来的，带有前缀`#:`。

另一个之前有问题的形式：

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

^① 该循环在给定任意素数下的无限性并非是显而易见的。为了证明其确实是无限的，起始点是Bertrand公设：对任何 $n > 1$ 都存在一个素数 p ， $n < p < 2n$ 。由此你就可以证明，对于给定的任意素数， P 总是小于它之前的所有素数之和，而下一个素数 P' 也同样小于前面的这个和再加上 P 。

如果将do-primes形式替换成其展开式的话，以上形式将会变成这样：

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
        (#:g2140 10))
       ((> p #:g2140))
       (incf ending-value p))
  ending-value)
```

再一次，由于do-primes循环外围的LET所绑定的变量ending-value不再被任何由展开代码引入的变量所掩盖，因此再没有漏洞了。

并非宏展开式中用到的所有字面名称都会导致问题。等你对于多种绑定形式有了更多经验以后，将可以鉴别一个用在某个位置上的给定名字是否会导致在宏抽象中出现漏洞。但安全起见，使用一个符号生成的名字并没有什么坏处。

利用这些修复就可以堵上do-primes实现中的所有漏洞了。一旦积累了一点宏编写方面的经验以后，你将获得在预先堵上这几类漏洞的情况下编写宏的本领。事实上做到这点很容易，只须遵循下面所概括的这些规则即可。

- 除非有特殊理由，否则需要将展开式中的任何子形式放在一个位置上，使其求值顺序与宏调用的子形式相同。
- 除非有特殊理由，否则需要确保子形式仅被求值一次，方法是在展开式中创建变量来持有求值参数形式所得到的值，然后在展开式中所有需要用到该值的地方使用这个变量。
- 在宏展开期使用GENSYM来创建展开式中用到的变量名。

8.8 用于编写宏的宏

当然，没有理由表明只有在编写函数的时候才能利用宏的优势。宏的作用是将常见的句法模式抽象掉，而反复出现在宏的编写中的特定模式同样也可受益于其抽象能力。

事实上，你已经见过了这样一种模式。许多宏，例如最后版本的do-primes，它们都以一个LET形式开始，后者引入了一些变量用来保存宏展开过程中用到的生成符号。由于这也是一个常见模式，那为什么不用一个宏来将其抽象掉呢？

本节将编写一个宏with-gensyms，它刚好做到这点。换句话说，你将编写一个用来编写宏的宏：一个宏用来生成代码，其代码又生成另外的代码。尽管在你习惯于在头脑中牢记不同层次的代码之前，可能会对复杂的编写宏的宏感到有一点困惑，但with-gensyms是相当简单的，而且还可以当作一个有用但又不会过于浪费脑筋的练习。

所写的宏应类似于下面这种形式。

```
defmacro do-primes ((var start end) &body body)
  (with-gensyms (ending-value-name)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
         ,@body)))
```

并且还需要让其等价于之前版本的do-primes。换句话说，with-gensyms需要展开成一个LET，它会把每一个命名的变量（在本例中是ending-value-name）都绑定到一个生成符号上。很容易就可以写出一个简单的反引用模板。

```
(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect `',(n (gensym)))
    ,@body))
```

注意你是怎样用一个逗号来插入LOOP表达式的值的。这个循环生成了一个绑定形式的列表，其中每个绑定形式由一个含有with-gensyms中的一个给定名字和字面代码(gensym)的列表所构成。你可以通过将name替换成一个符号的列表，从而在REPL中测试LOOP表达式生成的代码。

```
CL-USER> (loop for n in '(a b c) collect `',(n (gensym)))
((A (GENSYM)) (B (GENSYM)) (C (GENSYM)))
```

在绑定形式的列表之后，with-gensyms的主体参数被嵌入到LET的主体之中。这样，被封装在一个with-gensyms中的代码将可以引用任何传递给with-gensyms的变量列表中所命名的变量。

如果在新的do-primes定义中对with-gensyms形式进行宏展开，就将看到下面这样的结果：

```
(let ((ending-value-name (gensym)))
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (,ending-value-name ,end))
       ((> ,var ,ending-value-name))
     ,@body))
```

看起来不错。尽管这个宏相对简单，但重要的是要清楚地了解不同的宏是分别在何时被展开的：当编译关于do-primes的DEFMACRO时，with-gensyms形式就被展开成刚刚看到的代码并被编译了。这样，do-primes的编译版本就已经跟你手写外层的LET时一样了。当编译一个使用了do-primes的函数时，由with-gensyms生成的代码将会运行用来生成do-primes的展开式，但with-gensyms宏本身在编译一个do-primes形式时并不会被用到，因为在do-primes被编译时，它早已经被展开了。

3

另一个经典的用于编写宏的宏：once-only

另一个经典的用于编写宏的宏是once-only，它用来生成以特定顺序仅求值特定宏参数一次的代码。使用once-only，你几乎可以跟最初的有漏洞版本一样简单地写出do-primes来，就像这样：

```
(defmacro do-primes ((var start end) &body body)
  (once-only (start end)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

尽管如此，但如果详加解释的话，once-only的实现将远远超出本章的内容，因为它依赖于多层的反引用和解引用。如果真想进一步提高宏技术的话，你可以尝试分析它的工作方式。如下所示：

```
(defmacro once-only ((&rest names) &body body)
  (let ((gensyms (loop for n in names collect (gensym))))
    `(let (,@(loop for g in gensyms collect `(,g ,gensym))))
      `(let (,,@(loop for g in gensyms for n in names collect `',(,g ,n)))
          ,(let (,@(loop for n in names for g in gensyms collect `',(n ,g)))
              ,@body))))
```

8.9 超越简单宏

当然，我可以说更多关于宏的事情。目前为止，所有你见到的宏都是相当简单的例子，它们帮助你减轻了一些写代码的工作量，但却并没有提供表达事物的根本性的新方式。在接下来的章节里你将看到一些宏的示例，它们允许你以一种假如没有宏就完全做不到的方式来表达事物。从下一章开始，你将构建一个简单而高效的单元测试框架。

实践：建立单元测试框架



在本章里，你将编写代码开发一个简单的Lisp单元测试框架。这将使你有机会在真实代码中使用从第3章起已学到的某些语言特性，包括宏和动态变量。

该测试框架的主要设计目标是使其可以尽可能简单地增加新测试，运行多个测试套件，以及跟踪测试的失败。目前，你将集中于设计一个可以在交互开发期间使用的框架。

一个自动测试框架的关键特性在于该框架应该能够告诉你是否所有的测试都通过了。当计算机可以处理得更快更精确时，你就不应该将时间花在埋头检查测试所输出的答案上。因此，每个测试用例必须是一个能产生布尔值的表达式——真或假，通过或失败。举个例子，如果正在为内置的“+”函数编写测试，那么下面这些可能是合理的测试用例：^①

```
(= (+ 1 2) 3)
(= (+ 1 2 3) 6)
(= (+ -1 -3) -4)
```

带有副作用的函数会以稍微不同的方式进行测试。你必须调用该函数，然后查找是否有证据表明存在着预期的副作用。^②但最终，每个测试用例都将归结为一个布尔表达式，要么真要么假。

9.1 两个最初的尝试

如果正在做测试，那么就可以在REPL中输入这些表达式并检查它们是否返回`T`。但你可能想要一个框架使其可以在需要时轻松地组织和运行这些测试用例。如果想先处理最简单的可行情况，那就可以只写一个函数，让它对所有的测试用例都予以求值并用`AND`将结果连在一起：

```
(defun test-+ ()
  (and
```

① 这仅仅是出于阐述目的。很明显，编写对于诸如“+”这样的内置函数的测试用例有点荒唐，因为如果连这么基本的东西都无法正常工作的话，那么测试过程按照你期待的方式运行的可能性也微乎其微。另一方面，多数Common Lisp平台在很大程度上是用Common Lisp本身实现的，因此不难想象可以用Common Lisp编写测试套件来测试标准库函数。

② 副作用也可以包括诸如报错这样的情况，我将在第19章里讨论Common Lisp的错误。你可以在读过那章以后再来考虑如何在测试中检测一个函数是否在特定情况下产生了一个特别的错误。

```
(= (+ 1 2) 3)
(= (+ 1 2 3) 6)
(= (+ -1 -3) -4)))
```

无论何时，当想要运行这组测试用例时，都可以调用`test-+`。

```
CL-USER> (test-+)
T
```

一旦它返回`T`，就可知测试用例通过了。这种组织测试的方式也很优美简洁——不需要编写大量的重复测试代码。然而一旦发现某个测试用例失败了，同样也会发现它的运行报告会遗漏一些有用的信息。当`test-+`返回`NIL`时，你会知道某些测试失败了，但却不会知道这究竟是哪一个测试用例。

因此，让我们来尝试另一种简单得甚至有些愚蠢的方法。为了找出每个测试用例的运行情况，你可以写成类似下面这样。

```
(defun test-+ ()
  (format t "~~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (format t "~~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (format t "~~:[FAIL~;pass~] ... ~a~%" (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

现在每个测试用例都将单独报告结果。`FORMAT`指令中的`~~:[FAIL~;pass~]`部分将导致`FORMAT`在其第一个格式实参为假时打印出`FAIL`，而在其他情况下为`pass`。^①然后会将测试表达式本身标记到结果上。现在运行`test-+`就可以明确显示出发生了什么事。

```
CL-USER> (test-+)
pass ...
pass ...
pass ...
NIL
```

这次的结果报告更像是你想要的，可是代码本身却一团糟。问题出在对`FORMAT`的重复调用以及测试表达式乏味的重复，这些急切需要被重构。其中测试表达式的重复尤其讨厌，因为如果发生了错误输入，测试结果就会被错误地标记。

另一个问题在于，你无法得到单一的关于所有测试是否都通过的指示。如果只有三个测试用例的话，很容易通过扫描输出并查找“`FAIL`”来看到这点。不过当有几百个测试用例时，这将非常困难。

9.2 重构

我们真正所需要的编程方式应该是可以写出像第一个`test-+`那样能够返回单一的`T`或`NIL`值的高效函数，但同时它还可以像第二个版本那样能够报告单独测试用例的结果。就功能而言，由于第二个版本更接近于预期结果，所以最好是看看能否可以将某些烦人的重复消除掉。

消除重复的`FORMAT`相似调用的最简单方法就是创建一个新函数。

^① 我将在第18章里讨论包括这个指令在内的`FORMAT`指令的更多细节。

```
(defun report-result (result form)
  (format t "~-:[FAIL~;pass~] ... ~a~%" result form))
```

现在就可以用report-result来代替FORMAT编写test-+了。这不是一个大的改进，但至少现在如果打算改变报告结果的方式，则只需要修改一处即可。

```
(defun test-+ ()
  (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

接下来需要摆脱的是测试用例表达式的重复以及由此带来的错误标记结果的风险。真正想要的应该是可以将表达式同时看作代码（为了获得结果）和数据（用来作为标签）。无论何时，若想将代码作为数据来对待，这就意味着肯定需要一个宏。或者从另外一个角度来看，你所需要的方法应该能够自动编写容易出错的report-result调用。代码可能要写成下面这样。

```
(check (= (+ 1 2) 3))
```

并要让其与下列形式含义等同。

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
```

很容易就可以写出一个宏来作这种转换。

```
(defmacro check (form)
  `(report-result ,form ',form))
```

现在就可以改变test-+来使用check了。

```
(defun test-+ ()
  (check (= (+ 1 2) 3))
  (check (= (+ 1 2 3) 6))
  (check (= (+ -1 -3) -4)))
```

既然不喜欢重复的代码，那为什么不将那些对check的重复调用也一并消除掉呢？你可以定义check来接受任意数量的形式并将它们中的每个都封装在一个对report-result的调用里。

```
(defmacro check (&body forms)
  `(progn
   ,@(loop for f in forms collect `(report-result ,f ',f))))
```

这个定义使用了一种常见的宏习惯用法，将一系列打算转化成单一形式的形式分装在一个PROGN之中。注意是怎样使用,@将反引用模板所生成的表达式列表嵌入到结果表达式之中的。

有了check的新版本就可以写出一个像下面这样新版本的test-+~：

```
(defun test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

其等价于下面的代码：

```
(defun test-+ ()
  (progn
```

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
(report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
(report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4))))
```

多亏有了check，这个版本才和test-+的第一个版本一样简洁，而其展开代码却与第二个版本有着相同的功能。并且现在若想对test-+的行为做出任何改变，也都可以通过改变check来做到。

9.3 修复返回值

接下来可以修复test-+以使其返回值可以指示所有测试用例是否都通过了。由于check负责生成最终用来运行测试用例的代码，所以只需改变它来生成可以同时跟踪结果的代码就可以了。

首先可以对report-result做一个小改变，以使其在报告时顺便返回测试用例结果。

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form)
  result)
```

现在report-result返回了它的测试用例结果，故而看起来只需将PROGN变成AND就可以组合结果了。不幸的是，由于AND存在短路行为，即一旦某个测试用例失败了就跳过其余的测试，AND在本例中并不能完成你想要的事。另一方面，如果有一个像AND那样运作的操作符，同时又没有短路行为，那么就可以用它来代替PROGN，从而事情也就完成了。虽然Common Lisp并不提供这样一种构造，但你没有理由不能使用它：自己编写提供这一功能的宏是极其简单的。

暂时把测试用例放在一边，所需要的宏应如下所示，我们称其为combine-results。

```
(combine-results
  (foo)
  (bar)
  (baz))
```

并且它应与下列形式等同：

```
(let ((result t))
  (unless (foo) (setf result nil))
  (unless (bar) (setf result nil))
  (unless (baz) (setf result nil)))
  result)
```

编写这个宏唯一麻烦之处在于，需要在展开式中引入一个变量，即前面代码中的result。但正如前所述，在宏展开式中使用一个变量的字面名称会导致宏抽象出现漏洞，因此需要创建唯一的名字，这就需要用到with-gensyms了。可以像下面这样来定义combine-results：

```
(defmacro combine-results (&body forms)
  (with-gensyms (result)
    `(let ((,result t))
       ,(loop for f in forms collect `(unless ,f (setf ,result nil)))
       ,result)))
```

现在可以通过简单地改变展开式用combine-results代替PROGN来修复check。

```
(defmacro check (&body forms)
  `(combine-results
    ,(loop for f in forms collect `(report-result ,f ',f))))
```

使用这个版本的check, test-+就可以输出它的三个测试表达式结果，并返回T以说明每一个测试都通过了。^①

```
CL-USER> (test-+
  pass ... (= (+ 1 2) 3)
  pass ... (= (+ 1 2 3) 6)
  pass ... (= (+ -1 -3) -4)
  T
```

如果改变了一个测试用例而导致其失败^②，最终的返回值也将变成NIL。

```
CL-USER> (test-+
  pass ... (= (+ 1 2) 3)
  pass ... (= (+ 1 2 3) 6)
  FAIL ... (= (+ -1 -3) -5)
  NIL
```

9.4 更好的结果输出

由于只有一个测试函数，所以当前的结果输出是相当清晰的。如果一个特定的测试用例失败了，那么只需在check形式中找到那个测试用例并找出其失败原因即可。但如果编写了大量测试，可能就要以某种方式将它们组织起来，而不是将它们全部塞进一个函数里。例如，假设想要对“*”函数添加一些测试用例，则可以写一个新测试函数。

```
defun test-* ()
  (check
   (= (* 2 2) 4)
   (= (* 3 5) 15)))
```

现在有了两个测试函数，你可能还想用另一个函数来运行所有测试，这也相当简单。

```
(defun test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

这个函数使用combine-results来代替check，因为test-+和test-*都将分别汇报它们自己的结果。运行test-arithmetic将得到下列结果：

^① 如果test-+已经被编译了，这在特定Lisp实现中可能会隐式地发生，你可能需要重新求值test-+的定义以使改变后的check定义影响test-+的行为。另一方面，解释执行的代码通常在每次代码被解释时重新展开宏，从而使宏的重定义的效果立竿见影。

^② 你不得不通过改变测试来使其失败，因为你不能改变“+”的行为。

```
CL-USER> (test-arithmetic)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
pass ... (= (* 2 2) 4)
pass ... (= (* 3 5) 15)
T
```

现在假设其中一个测试用例失败了并且需要跟踪该问题。在只有5个测试用例和2个测试函数的情况下，找出失败测试用例的代码并不太困难。但假如有500个测试用例分散在20个函数里，如果测试结果可以显示每个测试用例来自什么函数就非常好了。

由于打印结果的代码集中在`report-result`函数里，所以需用一种方式来将当前所在测试函数的信息传递给`report-result`。可以为`report-result`添加一个形参来传递这一信息，但生成`report-result`调用的`check`却并不知道它是从什么函数被调用的，这就意味着还需要改变调用`check`的方式，向其传递一个参数使其随后传给`report-result`。

设计动态变量就是用于解决这类问题的。如果创建一个动态变量使得每个测试函数在调用`check`之前将其函数名绑定于其上，那么`report-result`就可以无需理会`check`来使用它了。

第一步是在最上层声明这个变量。

```
(defvar *test-name* nil)
```

现在需要对`report-result`稍微改动一下，使其在`FORMAT`输出中包括`*test-name*`。

```
(format t "~-:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
```

有了这些改变，测试函数将仍然可以工作但将产生下面的输出，因为`*test-name*`从未被重新绑定：

```
CL-USER> (test-arithmetic)
pass ... NIL: (= (+ 1 2) 3)
pass ... NIL: (= (+ 1 2 3) 6)
pass ... NIL: (= (+ -1 -3) -4)
pass ... NIL: (= (* 2 2) 4)
pass ... NIL: (= (* 3 5) 15)
T
```

为了正确报告其名字，需要改变两个测试函数。

```
(defun test-+ ()
  (let ((*test-name* 'test-+))
    (check
      (= (+ 1 2) 3)
      (= (+ 1 2 3) 6)
      (= (+ -1 -3) -4)))))

(defun test-* ()
  (let ((*test-name* 'test-*))
    (check
      (= (* 2 2) 4)
      (= (* 3 5) 15))))
```

现在结果被正确地打上了标签。

```
CL-USER> (test-arithmetic)
pass ... TEST-+: (= (+ 1 2) 3)
pass ... TEST-+: (= (+ 1 2 3) 6)
pass ... TEST-+: (= (+ -1 -3) -4)
pass ... TEST-*: (= (* 2 2) 4)
pass ... TEST-*: (= (* 3 5) 15)
T
```

9.5 抽象诞生

在修复测试函数的过程中，你又引入了一点儿新的重复。不但每个函数都需要包含其函数名两次——一次作为`DEFUN`中的名字，另一次是在`*test-name*`绑定里，而且同样的三行代码模式被重复使用在两个函数中。你可以在认定所有的重复都有害这一思路的指导下继续消除这些重复。但如果更进一步地调查一下导致代码重复的根本原因，你就可以学到关于如何使用宏的重要一课。

这两个函数的定义都以相同的方式开始，原因在于它们都是测试函数。导致重复是因为此时测试函数只做了一半抽象。这种抽象存在于你的头脑中，但在代码里没有办法表达“这是一个测试函数”，除非按照特定的模式来写代码。

不幸的是，部分抽象对于构建软件来说是很糟糕的，因为一个半成品的抽象在代码中就是通过模式来表现的，因此必然会得到大量的重复代码，它们将带有一切影响程序可维护性的不良后果。更糟糕的是，因为这种抽象仅存在于程序员的思路之中，所以实际上无法保证不同的程序员（或者甚至是工作在不同时期的同一个程序员）会以同样的方式来理解这种抽象。为了得到一个完整的抽象，你需要用一种方法来表达“这是一个测试函数”，并且这种方法要能将模式所需的全部代码都生成出来。换句话说，你需要一个宏。

由于试图捕捉的模式是一个`DEFUN`加上一些样板代码，所以需要写一个宏使其展开成`DEFUN`。然后使用该宏（而不是用一个简单的`DEFUN`）去定义测试函数，因此可以将其称为`deftest`。

```
(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
    (let ((*test-name* ',name))
      ,@body)))
```

使用该宏，你可以像下面这样重写`test-+`:

```
(deftest test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

9.6 测试层次体系

由于所建立的测试函数，所以可能会产生一些问题，`test-arithmetic`应该是一个测试函数吗？事实证明，这件事无关紧要——如果确实用`deftest`来定义它，那么它对`*test-name*`

的绑定将在任何结果被汇报之前被`test-+`和`test-*`中的绑定所覆盖。

但是现在，想象你有上千个测试用例需要组织在一起。组织的第一层是由诸如`test-+`和`test-*`这些能够直接调用`check`的测试函数所建立起来的，但在有数千个测试用例的情况下，你将需要其他层面的组织方式。诸如`test-arithmetic`这样的函数可以将相关的测试函数组成测试套件。现在假设某些底层测试函数会被多个测试套件所调用。测试用例很有可能在一个上下文中可以通过而在另一个中失败。如果发生了这种事，你想知道的很可能就不仅仅是哪一个底层测试函数含有这个测试用例那么简单了。

如果用`deftest`来定义诸如`test-arithmetic`这样的测试套件函数，并且对其中的`*test-name*`作一个小改变，就可以用测试用例的“全称”路径来报告结果，就像下面这样：

```
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
```

因为已经抽象了定义测试函数的过程，所以就无需修改测试函数的代码从而改变相关的细节。^①为了使`*test-name*`保存一个测试函数名的列表而不只是最近进入的测试函数的名字，你需要将绑定形式

```
(let ((*test-name* ',name))
```

变成

```
(let ((*test-name* (append *test-name* (list ',name))))
```

由于`APPEND`返回一个由其实参元素所构成的新列表，这个版本将把`*test-name*`绑定到一个含有追加其新的名字到结尾处的`*test-name*`的旧内容的列表。^②当每一个测试函数返回时，`*test-name*`原有的值将被恢复。

现在你可以用`deftest`代替`DEFUN`来重新定义`test-arithmetic`。

```
(deftest test-arithmetic ()
  (combine-results
    (test-+)
    (test-*)))
```

现在的结果明确地显示了你是怎样到达每一个测试表达式的。

```
CL-USER> (test-arithmetic)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

随着测试套件的增长，你可以添加新的测试函数层次。只要它们用`deftest`来定义，结果就会正确地输出。例如定义

^① 再强调一次，如果测试函数已经被编译了，那么在改变宏以后你将需要重新编译它们。

^② 你将在第12章里看到，用`APPEND`在列表结尾处追加元素并不是构造一个列表的最有效方式。但目前这种方法是有效的，只要测试的层次不是很深就可以了。并且如果这成为问题，所有你需要做的就是改变`deftest`的定义。

```
(deftest test-math ()
  (test-arithmetic))
```

将产生这样的结果：

```
CL-USER> (test-math)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

9.7 总结

你可以继续为这个测试框架添加更多特性。但作为一个以最小成本编写测试并可以在REPL轻松运行的框架来说，这已经是一个很好的开始了。这里给出完整的代码，全部只有26行：

```
(defvar *test-name* nil)

(defmacro deftest (name parameters &body body)
  "Define a test function. Within a test function we can call
  other test functions or use 'check' to run individual test
  cases."
  `(defun ,name ,parameters
    (let ((*test-name* (append *test-name* (list ',name))))
      ,@body)))

(defmacro check (&body forms)
  "Run each expression in 'forms' as a test case."
  `,(combine-results
    ,(loop for f in forms collect `(report-result ,f ',f)))))

(defmacro combine-results (&body forms)
  "Combine the results (as booleans) of evaluating 'forms' in order."
  (with-gensyms (result)
    `(let ((,result t))
       ,(loop for f in forms collect `(unless ,f (setf ,result nil)))
       ,result)))

(defun report-result (result form)
  "Report the results of a single test case. Called by 'check'."
  (format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
  result)
```

9

值得回顾的是，你能走到这一步是因为它显示了Lisp编程的一般方式。

你从定义一个解决问题的简单版本开始——怎样求值一些布尔表达式并找出它们是否全部返回真。将它们用AND连在一起可以工作并且在句法上很简洁，却无法满足以更好的结果输出的需要。因此你写了一些真正简单的代码，其中充满了代码重复以及在用你想要的方式报告结果时容易出错的用法。

下一步是查看你是否可以将第二个版本重构得跟前面版本一样简洁。你从一个标准的重构技术开始，将某些代码放进函数`report-result`中。不幸的是，你发现使用`report-result`会产生冗长和易错的代码，由于你不得不将测试表达式传递两次，一次作为值而另一次作为引用的数据。因此，你写了`check`宏来自动地正确调用`report-result`的细节。

在编写`check`的时候，你认识到在生成代码的同时，也可以让对`check`的单一调用生成对`report-result`的多个调用，从而得到了一个和最初**AND**版本一样简洁的`test-+`函数。

在那一点上你调整了`check`的API，从而你可以开始看到`check`内部的工作方式。下一个任务是修正`check`，使其生成的代码可以返回一个用来指示所有测试用例是否均已通过的布尔值。接下来你没有立即继续玩弄`check`，而是停下来沉迷于一个设计巧妙的微型语言。你梦想假如有一个非短路的**AND**构造就好了，这样修复`check`就会非常简单了。回到现实以后，你认识到不存在这样构造，但你可以用几行程序写一个出来。在写出了`combine-results`以后，对`check`的修复确实简单多了。

在那一点上唯一剩下的就是对你报告测试结果的方式做一些进一步的改进。一旦你开始修改测试函数，你就会认识到这些函数代表了特殊的一类值得有其自己的抽象方式的函数。因此你写出了`deftest`来抽象代码模式，使一个正常函数变成了一个测试函数。

借助`deftest`所提供的在测试定义和底层机制之间的抽象障碍，你可以无需修改测试函数而改进汇报结果的方式。

至此，你已经掌握了函数、变量和宏的基础知识，有了一点儿使用它们的实践经验，可以开始探索由函数和数据类型组成的Common Lisp的丰富的标准库了。

数字、字符和字符串



尽管函数、变量、宏和25个特殊操作符组成了语言本身的基本构造单元，但程序的构造单元则是你所使用的数据结构。正如Fred Brooks在《人月神话》里提到的，“数据的表现形式是编程的根本。”^①

Common Lisp为现代语言中常见的大多数数据类型都提供了内置支持：数字（整数、浮点数和复数）、字符、字符串、数组（包括多维数组）、列表、哈希表、输入和输出流以及一种可移植地表示文件名的抽象。函数在Lisp中也是第一类（first-class）数据类型。它们可以被保存在变量中，可以作为实参传递，也可以作为返回值返回以及在运行期创建。

而这些内置类型仅仅是开始。它们被定义在语言标准中，因此程序员们可以依赖于它们的存在，并且也因为它们可以跟语言的其余部分紧密集成，从而使其可以更容易地高效实现。但正如你将在后续章节里看到的那样，Common Lisp另外还提供了几种定义新的数据类型的方式，能定义对其的操作，并能将它们与内置数据类型集成起来。

但目前将先从内置数据类型开始讲起。因为Lisp是一种高阶语言，不同的数据类型的具体实现细节在很大程度上是隐藏的。从语言用户的角度来看，内置数据类型是由操作它们的函数所定义的。因此为了学习一个数据类型，你只需学会那些与之一起使用的函数就行了。另外，多数内置数据类型都具有Lisp读取器所理解并且Lisp打印机可使用的特殊语法。所以你才能将字符串写成"foo"，将数字写成123、1/23和1.23，以及把列表写成(a b c)。我将在描述操作它们的函数时具体描述不同对象的语法。

本章将介绍内置的“标量”数据类型：数字、字符和字符串。从技术上来讲，字符串并不是真正的标量。字符串是字符的序列，你可以访问单独的字符并使用一个操作在序列上的函数来处理该字符串。但我在这里讨论字符串则是因为多数字符串的相关函数会将它们作为单一值来处理，同时也是因为某些字符串函数与它们的字符组成部分之间有着紧密的关系。

10.1 数字

正如Barbie所说，数学很难。^②虽说Common Lisp并不能使其数学部分变得简单一些，但它确

① Fred Brooks，《人月神话》(Boston:Addison-Wesley,1995)，p.103。

② Mattel's Teen Talk Barbie。

实可以比其他编程语言在这方面简单不少，考虑到它的数学传统这并不奇怪。Lisp最初是由一位数学家设计而成的，用作研究数学函数的工具。并且MIT的MAC项目的主要项目之一——Macsyma符号代数系统也是由Maclisp（一种Common Lisp的前身）写成的。此外，Lisp还一直在MIT这类院校用作教学语言，它能很好地支持精确比值，省得计算机科学教授们要给学生们解释为什么 $10/4=2$ 。Lisp还曾经多次在高性能数值计算领域与FORTRAN竞争。

Lisp是一门用于数学的良好语言，其原因之一是它的数字更加接近于真正的数学数字，而不是易于在有穷计算机硬件上实现的近似数字。例如，Common Lisp中的整数几乎可以是任意大而不是限制在一个机器字的大小上。^①而两个整数相除将得到一个确切的比值而非截断的值。并且比值是由成对的任意大小的整数表示的，所以比值可以表示任意精度的分数。^②

另一方面，对于高性能数值编程，你可能想要用有理数的精度来换取使用硬件的底层浮点操作所得到的速度。因此Common Lisp也提供了几种浮点数，它们可以映射到适当的硬件支持浮点表达的实现上。^③浮点数也被用于表示其真正数学值为无理数的计算结果。

最后，Common Lisp支持复数——通过在负数上获取平方根和对数所得到的结果。Common Lisp标准甚至还指定了复域上无理和超越函数的主值和分支切断。

10.2 字面数值

可以用多种方式来书写字面数值，第4章已经介绍了一些例子。但你需要牢记Lisp读取器和Lisp求值器之间的分工——读取器负责将文本转化成Lisp对象，而Lisp求值器只处理这些对象。对于一个给定类型的数字来说，它可以有多种不同的字面表示方式，所有这些都将被Lisp读取器转化成相同的对象表示。例如，你可以将整数10写成10、20/2、#xA或是其他形式的任何数字，但读取器将把所有这些转化成同一个对象。当数字被打印回来时，比如在REPL中，它们将以一种可能与输入该数字时不同的规范化文本语法被打印出来。例如：

```
CL-USER> 10
10
CL-USER> 20/2
10
```

^① 很明显，一个具有有限内存的计算机可以表示的数字大小在事实上仍然是有限的。更进一步，特定Common Lisp实现中使用的大数的实际表示在其所能表达的数字大小上可能有另外的限制，但这些限制通常会超出“天文数字”般大的数字。例如，整个宇宙中原子的数量预计少于 2^{269} ，而当前的Common Lisp实现可以轻易处理最大或超出 2^{262144} 的数字。

^② 那些对于使用Common Lisp密集的数值计算感兴趣的人们应该注意到，如果单纯比较数值代码的性能，那么和诸如C或FORTRAN这样的语言比起来，Common Lisp可能会更慢。这是因为在Common Lisp中即便是 $(+ a b)$ 这样简单的表达式也比其他语言中看起来等价的 $a+b$ 做更多的事。由于Lisp中动态类型的机制以及对诸如任意精度有理数和复数的支持，一个看来简单的加法操作比两个已知表达为机器字的数字相加做更多的事。尽管如此，你可以使用声明来告诉Common Lisp关于你所使用的数字类型的信息，从而使其生成与C或FORTRAN编译器做相同工作的代码。为这类优化调节数字代码超出了本书的范围，但这确实是可能的。

^③ 尽管标准并未要求，但许多Common Lisp实现都支持IEEE浮点算术标准，*IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985* (Institute of Electrical and Electronics Engineers, 1985年)。

```
CL-USER> #xa
10
```

整数的语法是可选的符号（+或-）后接一个或多个数字。比值的写法则依次组合了一个可选符号、一个代表分子的数位序列、一个斜杠 (/) 以及另一个代表分母的数位序列。所有的有理数在读取后都被“规范化”，这就是10和20/2都被读成同一个数字的原因，3/4和6/8也是这样，有理数以“简化”形式打印，整数值以整数语法来打印，而比值被打印成分值和分母约分到最简的形式。

用十进制以外的进制来书写有理数也是可能的。如果前缀是#B或#b，一个字面有理数将以二进制来读取，其中0和1是唯一的合法数字。前缀是#0或#o代表一个八进制数（合法数字0-7），而#X或#x则代表十六进制数（合法数字0-F或0-f）。你可以使用#nR以2到36的其他进制书写有理数，其中n代表进制数（一定要以十进制书写）。超过9的附加“数字”从字母A-Z或a-z中获取。注意，这些进制指示符将应用到整个有理数上——不可能以一种进制来书写比值的分值，而用另一种进制来书写分母。另外，你可以将整数而非比值写成以十进制小数点结尾的十进制数。^①下面是一些有理数的例子，带有它们对应的规范化十进制表示：

123	→ 123
+123	→ 123
-123	→ -123
123.	→ 123
2/3	→ 2/3
-2/3	→ -2/3
4/6	→ 2/3
6/3	→ 2
#b10101	→ 21
#b1010/1011	→ 10/11
#o777	→ 511
#xDADA	→ 56026
#36rABCDEFGHIJKLMNOPQRSTUVWXYZ	→ 8337503854730415241050377135811259267835

也可以用多种方式来书写浮点数。和有理数不同，表示浮点数的语法可以影响数字被读取的实际类型。Common Lisp定义了四种浮点数子类型：短型、单精度、双精度和长型。每一个子类型在其表示中可以使用不同数量的比特，这意味着每个子类型可以表示跨越不同范围和精度的值。更多的比特可以获得更宽的范围和更高的精度。^②

浮点数的基本格式是一个可选符号后跟一个非空的十进制数字序列，同时可能带有一个嵌入的小数点。这个序列可能后接一个代表“计算机科学计数法”^③的指数标记。指数标记由单个字

10

- ① 通过改变全局变量*READ-BASE*，也有可能实现无需使用特别的进制标记即可改变读取器在数字上使用的默认基数。不过这样可能会导致严重的混乱。
- ② 由于浮点数的目的是为了有效使用浮点硬件，因此每个Lisp实现都允许将这四种子类型映射到适当的原生浮点类型上。如果硬件支持少于四种区别的表示方法，这些类型中的一种或几种可能是等价的。
- ③ “计算机科学计数法”加引号是因为，尽管其自从FORTRAN时就被广范用在计算机语言里，但它实际上和真正的科学计数法很不相同，确切地说，像1.0e4这样的数字代表10000.0，而在真正的科学计数法中它表示为 1.0×10^4 。而进一步产生混淆的是，在真正的科学计数法中字母e代表自然对数的底。因此， $1.0 \times e^4$ 从表面上看类似于1.0e4，但其却是一个完全不同的值，约等于54.6。

母后跟一个可选符号和一个数字序列组成，其代表10的指数用来跟指数标记前的数字相乘。该字母有两重作用：它标记了指数的开始并且指出该数字应当使用的浮点表示方式。指数标记s、f、d、l（以及它们等价的大写形式）分别代表短型、单精度、双精度以及长型浮点数。字母e代表默认表示方式（单浮点数）。

没有指数标记的数字以默认表示来读取，并且必须含有一个小数点后面还至少有一个数字，以区别于整数。浮点数中的数字总是以十进制数字来表示——#B、#X、#O和#R语法只用在有理数上。下面是一些浮点数的例子，带有它们的规范表示形式：

1.0	→ 1.0
1e0	→ 1.0
1d0	→ 1.0d0
123.0	→ 123.0
123e0	→ 123.0
0.123	→ 0.123
.123	→ 0.123
123e-3	→ 0.123
123E-3	→ 0.123
0.123e20	→ 1.23e+19
123d23	→ 1.23d+25

最后，复数有它们自己的语法，也就是#c或#c跟上一个由两个实数所组成的列表，分别代表复数的实部和虚部。事实上因为实部和虚部必须同为有理数或相同类型的浮点数，所以共有五种类型的复数。

不过你可以随意书写它们。如果复数被写成由有理数和浮点数组成，该有理数将被转化成一个适当表示的浮点数。类似地，如果实部和虚部是不同表示法的浮点数，使用较小表示法的那个将被提高。

尽管如此，没有复数可以具有一个有理的实部和一个零的虚部，因为这样的值从数学上讲是有理的，所以它们将用对应的有理数值来表示。同样的数学论据对于由浮点数所组成的复数也是成立的，但其中那些带有零虚部的复数总是一个与代表实部的浮点数不同的对象。下面是一些以复数语法写成的数字的例子：

#c(2 1)	→ #c(2 1)
#c(2/3 3/4)	→ #c(2/3 3/4)
#c(2 1.0)	→ #c(2.0 1.0)
#c(2.0 1.0d0)	→ #c(2.0d0 1.0d0)
#c(1/2 1.0)	→ #c(0.5 1.0)
#c(3 0)	→ 3
#c(3.0 0.0)	→ #c(3.0 0.0)
#c(1/2 0)	→ 1/2
#c(-6/3 0)	→ -2

10.3 初等数学

基本的算术操作即加法、减法、乘法和除法，通过函数+、-、*、/支持所有不同类型的Lisp数字。使用超过两个参数来调用这其中的任何一个函数，这种作法将等价于在前两个参数上调用

相同的函数而后再在所得结果和其余参数上再次调用。例如， $(+ 1 2 3)$ 等价于 $(+ (+ 1 2) 3)$ 。当只有一个参数时， $+$ 和 $*$ 直接返回其值， $-$ 返回其相反值，而 $/$ 返回其倒数。^①

$(+ 1 2)$	$\rightarrow 3$
$(+ 1 2 3)$	$\rightarrow 6$
$(+ 10.0 3.0)$	$\rightarrow 13.0$
$(+ #c(1 2) #c(3 4))$	$\rightarrow #c(4 6)$
$(- 5 4)$	$\rightarrow 1$
$(- 2)$	$\rightarrow -2$
$(- 10 3 5)$	$\rightarrow 2$
$(* 2 3)$	$\rightarrow 6$
$(* 2 3 4)$	$\rightarrow 24$
$(/ 10 5)$	$\rightarrow 2$
$(/ 10 5 2)$	$\rightarrow 1$
$(/ 2 3)$	$\rightarrow 2/3$
$(/ 4)$	$\rightarrow 1/4$

如果所有实参都是相同类型的数（有理数、浮点数或复数），则结果也将是同类型的，除非带有有理部分的复数操作的结果产生了一个零虚部的数，此时结果将是一个有理数。尽管如此，浮点数和复数是有传播性的。如果所有实参都是实数但其中有一个或更多是浮点数，那么其他实参将被转化成以实际浮点实参的“最大”浮点表示而成的最接近浮点值。那些“较小”表示的浮点数也将被转化成更大的表示。同样，如果实参中的任何一个是复数，则任何实参数会被转化成等价的复数。

$(+ 1 2.0)$	$\rightarrow 3.0$
$(/ 2 3.0)$	$\rightarrow 0.6666667$
$(+ #c(1 2) 3)$	$\rightarrow #c(4 2)$
$(+ #c(1 2) 3/2)$	$\rightarrow #c(5/2 2)$
$(+ #c(1 1) #c(2 -1))$	$\rightarrow 3$

因为/不作截断处理，所以Common Lisp提供了4种类型的截断和舍入用于将一个实数（有理数或浮点数）转化成整数：**FLOOR**向负无穷方向截断，返回小于或等于实参的最大整数；**CEILING**向正无穷方向截断，返回大于或等于参数的最小整数；**TRUNCATE**向零截断，对于正实参而言，它等价于**FLOOR**，而对于负实参则等价于**CEILING**；而**ROUND**舍入到最接近的整数上，如果参数刚好位于两个整数之间，它舍入到最接近的偶数上。

两个相关的函数是**MOD**和**REM**，它返回两个实数截断相除得到的模和余数。这两个函数与**FLOOR**和**TRUNCATE**函数之间的关系如下所示：

```
(+ (* (floor (/ x y)) y) (mod x y)) ≡ x
(+ (* (truncate (/ x y)) y) (rem x y)) ≡ x
```

因此，对于正的商它们是等价的，而对于负的商它们产生不同的结果。^②

10

① 出于数学一致性的考虑， $+$ 和 $*$ 也可以不带参数被调用。这种情况下，它们将返回适当的值： $+$ 返回0，而 $*$ 返回1。

② 严格来讲，**MOD**等价于Perl和Python中的%操作符，而**REM**等价于C和Java中的%。（从技术上来讲，%在C中的行为直到C99标准时才明确指定。）

函数`1+`和`1-`提供了表示从一个数字增加或减少一个的简化方式。注意它们与宏`INCF`和`DECF`有所不同。`1+`和`1-`只是返回一个新值的函数，而`INCF`和`DECF`会修改一个位置。下面的恒等式显示了`INCF/DECF`、`1+/1-`和`+/-`之间的关系：

```
(incf x)    ≡ (setf x (+ x)) ≡ (setf x (+ x 1))
(decf x)    ≡ (setf x (- x)) ≡ (setf x (- x 1))
(incf x 10) ≡ (setf x (+ x 10))
(decf x 10) ≡ (setf x (- x 10))
```

10.4 数值比较

函数`=`是数值等价谓词。它用数学意义上的值来比较数字，而忽略类型上的区别。这样，`=`将把不同类型在数学意义上等价的值视为等价，而通用等价谓词`EQL`将由于其类型差异而视其不等价。（但通用等价谓词`EQUALP`使用`=`来比较数字。）如果它以超过两个参数被调用，它将只有当所有参数具有相同值时才返回真。如下所示：

```
(= 1 1)          → T
(= 10 20/2)      → T
(= 1 1.0 #c(1.0 0.0) #c(1 0)) → T
```

相反，只有当函数`/=`的全部实参都是不同值时才返回真。

```
(/= 1 1)        → NIL
(/= 1 2)        → T
(/= 1 2 3)      → T
(/= 1 2 3 1)    → NIL
(/= 1 2 3 1.0) → NIL
```

函数`<、>、<=和>=`检查有理数和浮点数（也就是实数）的次序。跟`=`和`/=`相似，这些函数也可以用超过两个参数来调用，这时每个参数都跟其右边的那个参数相比较。

```
(< 2 3)        → T
(> 2 3)        → NIL
(> 3 2)        → T
(< 2 3 4)      → T
(< 2 3 3)      → NIL
(<= 2 3 3)     → T
(<= 2 3 3 4)   → T
(<= 2 3 4 3)   → NIL
```

要想选出几个数字中最小或最大的那个，你可以使用函数`MIN`或`MAX`，其接受任意数量的实数参数并返回最小或最大值。

```
(max 10 11)   → 11
(min -12 -10) → -12
(max -1 2 -3) → 2
```

其他一些常用函数包括`ZEROP`、`MINUSP`和`PLUSP`，用来测试单一实数是否等于、小于或大于零。另外两个谓词`EVENP`和`ODDP`，测试单一整数参数是否是偶数或奇数。这些函数名称中的`P`后缀是一种谓词函数的标准命名约定，这些函数能够测试某些条件并返回一个布尔值。

10.5 高等数学

目前为止，你所看到的函数只是初级的内置数学函数。Lisp也支持对数函数LOG，指数函数EXP和EXPT，基本三角函数SIN、COS和TAN及其逆函数ASIN、ACOS和ATAN，双曲函数SINH、COSH和TANH及其逆函数ASINH、ACOSH和ATANH。它还提供了函数用来获取一个整数中单独的位，以及取出一个比值或一个复数中的部分。完整的函数列表参见任何Common Lisp参考。

10.6 字符

Common Lisp字符和数字是不同类型的对象。本该如此——字符不是数字，而将其同等对待的语言当字符编码改变时（比如说从8位ASCII到21位Unicode^①）可能会出现问题。由于Common Lisp标志并未规定字符的内部表示方法，当今几种Lisp实现都使用Unicode作为其“原生”字符编码，尽管从标准化组织的观点来看，Unicode在Common Lisp自身的标准化成型时期只是昙花一现。

字符的读取语法很简单：#\后跟想要的字符。这样，#\x就是字符x。任何字符都可以用在#\之后，包括“”、“（”和空格这样的特殊字符。但以这种方式来写空格字符对我们来说可读性不高，特定字符的替代语法是#\后跟该字符的名字。具体支持的名字取决于字符集和所在的Lisp实现，但所有实现都支持名字Space和Newline。这样就应该写成用#\Space来代替“#\”，尽管后者在技术上是合法的。其他半标准化的名字（如果字符集包含相应的字符实现就必须采用的名字）是Tab、Page、Rubout、Linefeed、Return和Backspace。

10.7 字符比较

可以对字符做的主要操作，除了将它们放进字符串（我将在本章后面讨论这点）之外，还可将它们与其他字符相比较。由于字符不是数字，所以不能使用诸如<和>这样的数值比较函数。作为替代，有两类函数提供了数值比较符的特定于字符的相似物：一类是大小写相关的，而另一类是大小写无关的。

数值=的大小写相关相似物是函数CHAR=。像=那样，CHAR=可以接受任意数量的实参并只在它们全是相同字符时才返回真。大小写无关版本是CHAR-EQUAL。

其余的字符比较符遵循了相同的命名模式：大小写相关的比较符通过在其对应的数值比较符前面加上CHAR来命名；大小写无关的版本拼出比较符的名字，前面加上CHAR和一个连字符。不过，<=和>=被拼写成了其逻辑等价形式NOT-GREATERP和NOT-LESSP，而不是更确切的LESSP-OR-EQUALP和GREATERP-OR-EQUALP。和它们的数值等价物一样，所有这些函数都接受一个或更多参数。表10-1总结了数值和字符比较函数之间的关系。

^① 甚至像Java也会产生问题，它基于Unicode注定将成为未来主流字符编码这一理论，而从一开始就被设计使用Unicode字符。产生问题是因为Java字符被定义为16位值，而Unicode3.1标准将Unicode字符集范围扩展到了需要21位表示。太惨了。

表10-1 字符比较函数

数值相似物	大小写相关	大小写无关
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL
<	CHAR<	CHAR-LESSP
>	CHAR>	CHAR-GREATERP
<=	CHAR<=	CHAR-NOT-GREATERP
>=	CHAR>=	CHAR-NOT-LESSP

其他处理字符的函数包括测试一个给定字符是否是字母或者数字字符，测试一个字符的大小写，获取不同大小写的对应字符，以及在代表字符编码的数值和实际字符对象之间转化。对于完整的细节，请参见你喜爱的Common Lisp参考。

10.8 字符串

如前所述，Common Lisp中的字符串其实是一种复合数据类型，即一维字符数组。因此，我将在下一章讨论用来处理序列的许多函数时谈及许多字符串应用，因为字符串只不过是一种序列。但字符串也有其自己的字面语法和一个用来进行字符串特定操作的函数库。本章将讨论字符串的这些方面，其余部分留到第11章再作介绍。

正如你所看到的那样，字面字符串写在闭合的双引号里。你可以在一个字面字符串中包括任何字符集支持的字符，除了双引号 ("") 和反斜杠 (\)。而如果你将它们用一个反斜杠转义的话也可以包括这两个字符。事实上，反斜杠总是转义其下一个字符，无论它是什么，尽管这对于除了 “\” 和 “\” 本身之外的其他字符并不必要。表10-2显示了不同的字面字符串是如何被Lisp读取器读取的。

表10-2 字面字符串

字面字符串	内 容	说 明
"foobar"	foobar	纯字符串
"foo\"bar"	foo" bar	反斜杠转义引号
"foo\\bar"	foo\bar	第一个反斜杠转义第二个反斜杠
"\\ foobar\\ \"	"foobar"	反斜杠转义引号
"foo\\bar"	foobar	反斜杠“转义” b

注意，REPL将以可读的形式原样打印字符串，并带有外围的引号和任何必要的转义反斜杠。因此，如果想要看到一个字符串的实际内容，就要使用诸如FORMAT这种原本就是设计用于打印出可读性良好输出的函数。例如，下面是在REPL中输入一个含有内嵌引号的字符串时所看到的输出。

```
CL-USER> "foo\"bar"
"foo\"bar"
```

另一方面, **FORMAT**将显示出实际的字符串内容:^①

```
CL-USER> (format t "foo\"bar")
foo"bar
NIL
```

10.9 字符串比较

你可以使用一组遵循了和字符比较函数相同的命名约定的函数来比较字符串, 只不过要将前缀的CHAR换成STRING(参见表10-3)。

表10-3 字符串比较函数

数值相似物	大小写相关	大小写无关
=	STRING=	STRING-EQUAL
/=	STRING/=	STRING-NOT-EQUAL
<	STRING<	STRING-LESSP
>	STRING>	STRING-GREATERP
<=	STRING<=	STRING-NOT-GREATERP
>=	STRING>=	STRING-NOT-LESSP

但跟字符和数字比较符不同的是, 字符串比较符只能比较两个字符串。这是因为它们还带有关键字参数, 从而允许你将比较限制在每个或两个字符串的子字符串上。这些参数:`:start1`、`:end1`、`:start2`和`:end2`指定了起始和结束参数字符串中子字符串的起始和终止位置(左闭右开区间)。因此请看下面这个表达式。

```
(string= "foobarbaz" "quuxbarfoo" :start1 3 :end1 6 :start2 4 :end2 7)
```

在两个参数中比较子字符串"bar"并返回真。参数`:end1`和`:end2`可以为NIL(或者整个关键字参数被省略)指示相应的子字符串扩展到字符串的结尾。

当参数不同时返回真的比较符, 也即**STRING=**和**STRING-EQUAL**之外的所有操作符, 将返回第一个字符串中首次检测到不匹配的索引。

```
(string/= "lisp" "lissome") → 3
```

如果第一个字符串是第二个字符串的前缀, 返回值是第一个字符串的长度, 也就是一个大于字符串中最大有效索引的值。

```
(string< "lisp" "lisper") → 4
```

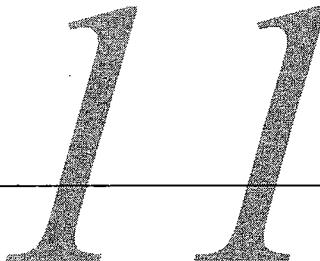
当比较子字符串时, 返回值仍然是该字符串作为整体的索引, 例如, 下面的调用比较子字符串"bar"和"baz"; 但返回了5, 因为它是"r"在第一个字符串中的索引:

```
(string< "foobar" "abaz" :start1 3 :start2 1) → 5 ; N.B. not 2
```

10

^① 注意: 尽管如此, 并非所有的字面字符串都可以通过将其作为**FORMAT**的第二个参数传递来打印, 因为特定的字符序列对于**FORMAT**有特殊的含义。为了安全地通过**FORMAT**打印一个任意字符串, 比如说, 一个变量s的值, 你应当写为(`format t "~a" s`)。

其他字符串函数允许你转化字符串的大小写以及从一个字符串的一端或两端修剪字符。而且如同我前面提到的，由于字符串实际上是一种序列，我将在下一章讨论的所有序列函数都可用于字符串。例如，你可以用**LENGTH**函数来检查字符串的长度并获取和设定字符串中的个别字符，使用通用序列元素访问函数**ELT**或者使用通用数组元素访问函数**AREF**。你还可以使用特定于字符串的访问函数**CHAR**。这些以及其他一些函数都是下一章的主题，让我们继续吧。



和多数编程语言一样，Common Lisp也提供了能将多个值收集到单一对象的标准数据类型。每一种语言在处理集合问题上都稍有不同，但基本的集合类型通常都可归结为一个整数索引的数组类型，以及一个可将或多或少的任意关键字映射到值上的表类型。前者分别称为数组 (array)、列表 (list) 或元组 (tuple)，后者命名为哈希表 (hash table)、关联数组 (associative array)、映射表 (map) 和字典 (dictionary)。

当然，Lisp以其列表数据结构闻名于世，而多数遵循了语言用法的进化重演 (ontogeny-recapitulates-phylogeny) 原则的Lisp教材也都从讨论基于列表的Lisp集合开始。尽管如此，这一观点通常导致读者错误地推论出列表是Lisp的唯一集合类型。更糟糕的是，因为Lisp的列表是如此灵活的数据结构，它可用于许多其他语言使用数组和哈希表的场合。但是将注意力过于集中在列表上是错误的。尽管它是一种将Lisp代码作为Lisp数据来表达的关键数据结构，但在许多场合其他数据结构更合适。

为了避免让列表过于出风头，在本章里我将集中在Common Lisp的其他集合类型上：向量和哈希表。^①尽管如此，向量和列表共享了许多特征，因此Common Lisp将它们都视为更一般抽象的子类型（即序列）。因此，你可以将本章我所讨论的许多函数同时用在向量和列表上。

11.1 向量

向量是Common Lisp基本的整数索引集合，它们分为两大类。定长向量与Java等语言里的数组非常相似：一块数据头以及一段保存向量元素的连续内存区域。^②另一方面，变长向量更像是Perl或Ruby中的数组、Python中的列表以及Java中的ArrayList类：它们抽象了实际存储，允许向量随着元素的增加和移除而增大和减小。

你可以用函数VECTOR来生成含有特定值的定长向量，该函数接受任意数量的参数并返回一个新分配的含有那些参数的定长向量。

① 一旦你熟悉了Common Lisp提供的所有数据类型，会发现列表可以作为原型数据结构来使用，并且以后可以替换为其他更高效的东西，只要你清楚了使用数据的确切方式。

② 向量被称为向量，而不是像其他语言里那样被称为数组，是因为Common Lisp支持真正的多维数组。将它称为一维数组应该更加确切，但过于冗繁。

```
(vector)      → #()
(vector 1)    → #(1)
(vector 1 2)  → #(1 2)
```

语法#(...)是Lisp打印机和读取器使用的向量的字面表示形式，该语法可使你用PRINT打印并用READ读取，以此来保存并恢复向量。可以使用#(...)语法在代码中添加字面向量，但修改字面对象的后果并不明确，因此应当总是使用VECTOR或更为通用的函数MAKE-ARRAY来创建打算修改的向量。

MAKE-ARRAY比**VECTOR**更加通用，因为它可以用来创建任何维度的数组以及定长和变长向量。**MAKE-ARRAY**的一个必要参数是一个含有数组维数的列表。由于向量是一维数组，所以该列表将含有一个数字，也就是向量的大小。出于方便的考量，**MAKE-ARRAY**也会用一个简单的数字来代替只含有一项的列表。如果没有其他参数，**MAKE-ARRAY**就将创建一个带有未初始化元素的向量，它们必须在被访问之前设置其值。^①为了创建所有元素都设置到一个特定值上的向量，你可以传递一个:initial-element参数。因此，为了生成一个元素初始化到NIL的五元素向量，你可以写成下面这样：

```
(make-array 5 :initial-element nil) → #(NIL NIL NIL NIL NIL)
```

MAKE-ARRAY也是用来创建变长向量的函数。变长向量是比定长向量稍微复杂的向量。除了跟踪其用来保存元素的内存和可访问的槽位数量，变长向量还要跟踪实际存储在向量中的元素数量。这个数字存放在向量的填充指针里，这样称呼是因为它是当为向量添加一个元素时下一个被填充位置的索引。

为了创建带有填充指针的向量，你可以向**MAKE-ARRAY**传递一个:fill-pointer实参。例如，下面的**MAKE-ARRAY**调用生成了一个带有五元素空间的向量，它看起来是空的，因为填充指针是零：

```
(make-array 5 :fill-pointer 0) → #()
```

为了向可变向量的尾部添加一个元素，你可以使用函数**VECTOR-PUSH**。它在填充指针的当前值上添加一个元素并将填充指针递增一次，并返回新元素被添加位置的索引。函数**VECTOR-POP**返回最近推入的项，并在该过程中递减填充指针。

```
(defparameter *x* (make-array 5 :fill-pointer 0))

(vector-push 'a *x*) → 0
*x*                → #(A)
(vector-push 'b *x*) → 1
*x*                → #(A B)
(vector-push 'c *x*) → 2
*x*                → #(A B C)
(vector-pop *x*)   → C
*x*                → #(A B)
(vector-pop *x*)   → B
*x*                → #(A)
```

^① 数组元素在其被访问前“必须”被赋值，如果不这样做，其行为将是未定义的。Lisp不一定会报错。

```
(vector-pop *x*)      → A
*x*                   → #()
```

尽管如此，甚至一个带有填充指针的向量也不是完全变长的。向量*x*只能保存最多五个元素。为了创建一个可任意变长的向量，你需要向**MAKE-ARRAY**传递另外一个关键字参数:**:adjustable**。

```
(make-array 5 :fill-pointer 0 :adjustable t) → #()
```

这个调用生成了一个可调整的向量，其底层内存可以按需调整大小。为了向一个可调整向量添加元素，你可以使用**VECTOR-PUSH-EXTEND**，它就像**VECTOR-PUSH**那样工作，只是在你试图向一个已满的向量（其填充指针等于底层存储的大小）中推入元素时，它能自动扩展该数组。^①

11.2 向量的子类型

目前为止，你处理的所有向量都是可以保存任意类型对象的通用向量。你也可以创建特化的向量使其仅限于保存特定类型的元素。使用特化向量的理由之一是，它们可以更加紧凑地存储，并且可以比通用向量提供对其元素更快速的访问。不过目前我们将集中介绍几类特化向量，它们本身就是重要的数据类型。

其中一类你已经见过了，就是字符串，它是特定用来保存字符的向量。字符串特别重要，以至于它们有自己的读写语法（双引号）和一组特定于字符串的函数，前一章已讨论过。但因为它们也是向量，所有接下来几节里所讨论的接受向量实参的函数也可以用在字符串上。这些函数将为字符串函数库带来新的功能，例如用一个子串来搜索字符串，查找一个字符在字符串中出现的次数，等等。

诸如“foo”这样的字面字符串，和那些用#()语法写成的字面向量一样，其大小都是固定的，并且根本不能修改它们。但你可以用**MAKE-ARRAY**通过添加另一个关键字参数:**:element-type**来创建变长字符串。该参数接受一个类型描述符。我将不会介绍你可以在这里使用的所有可能的类型描述符，目前只需知道，你可以通过传递符号**CHARACTER**作为:**:element-type**来创建字符串。注意，你需要引用该符号以避免它被视为一个变量名。例如，创建一个初始为空但却变长的字符串，如下所示：

```
(make-array 5 :fill-pointer 0 :adjustable t :element-type 'character) → ""
```

位向量是元素全部由0或1所组成的向量，它也得到一些特殊对待。它们有一个特别的读/写语法，看起来像#*00001111，另外还有一个相对巨大的函数库。这些函数可用于按位操作，例如将两个位数组“与”在一起（不会介绍）。用来创建一个位向量传递给:**:element-type**的类型描述符是符号**BIT**。

^① 尽管经常一起使用，但:**:file-pointer**和:**:adjustable**是无关的——你可以生成一个不带有填充指针的可调整数组。不过，你只能在带有填充指针的向量上使用**VECTOR-PUSH**和**VECTOR-POP**，并只能在带有填充指针且可调整的向量上使用**VECTOR-PUSH-EXTEND**。你还可以使用函数**ADJUST-ARRAY**以超出扩展向量长度的多种方式来修改可调整数组。

11.3 作为序列的向量

正如早先所提到的，向量和列表是抽象类型序列的两种具体子类型。接下来几节里讨论的所有函数都是序列函数：除了可以应用于向量（无论是通用还是特化的）之外，它们还可应用于列表。

两个最基本的序列函数是`LENGTH`，其返回一个序列的长度；`ELT`，其允许通过一个整数索引来访问个别元素。`LENGTH`接受序列作为其唯一的参数并返回它含有的元素数量。对于带有填充指针的向量，这些是填充指针的值。`ELT`是元素（element）的简称，它接受序列和从0到序列长度（左闭右开区间）的整数索引，并返回对应的元素。`ELT`将在索引超出边界时报错。和`LENGTH`一样，`ELT`也将把一个带有填充指针的向量视为其具有该填充指针所指定的长度。

```
(defparameter *x* (vector 1 2 3))

(length *x*) → 3
(elt *x* 0) → 1
(elt *x* 1) → 2
(elt *x* 2) → 3
(elt *x* 3) → error
```

`ELT`也是一个支持`SETF`的位置，因此可以像这样来设置一个特定元素的值：

```
(setf (elt *x* 0) 10)

*x* → #(10 2 3)
```

11.4 序列迭代函数

尽管理论上所有的序列操作都可归结于`LENGTH`、`ELT`和`ELT`的`SETF`操作的某种组合，但Common Lisp还是提供了一个庞大的序列函数库。

一组序列函数允许你无需编写显式循环就可以表达一些特定的序列操作，比如说查找或过滤指定元素等。表11-1总结如下。

表11-1 基本序列函数

名 称	所需参数	返 回
COUNT	项和序列	序列中出现某项的次数
FIND	项和序列	项或NIL
POSITION	项和序列	序列中的索引NIL
REMOVE	项和序列	项的实例被移除后的序列
SUBSTITUTE	新项、项和序列	项的实项被新项替换后的序列

下面是一些关于如何使用这些函数的简单例子：

```
(count 1 #(1 2 1 2 3 1 2 3 4)) → 3
(remove 1 #(1 2 1 2 3 1 2 3 4)) → #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4)) → (2 2 3 2 3 4)
(remove #\a "foobarbaz") → "foobrbz"
```

```
(substitute 10 1 #(1 2 1 2 3 1 2 3 4)) → #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4)) → (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz") → "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4)) → 1
(find 10 #(1 2 1 2 3 1 2 3 4)) → NIL
(position 1 #(1 2 1 2 3 1 2 3 4)) → 0
```

注意，**REMOVE**和**SUBSTITUTE**总是返回与其序列实参相同类型的序列。

可以使用关键字参数以多种方式修改这五个函数的行为。例如，在默认情况下，这些函数会查看序列中与其项参数相同的对象。你可以用两种方式改变这一行为。首先，你可以使用:**:test**关键字来传递一个接受两个参数并返回一个布尔值的函数。如果有了这一函数，它将使用该函数代替默认的对象等价性测试**EQL**来比较序列中的每个元素。^①其次，使用:**:key**关键字可以传递单参数函数，其被调用在序列的每个元素上以抽取出一个关键值，该值随后会和替代元素自身的项进行比对。但请注意，诸如**FIND**这类返回序列元素的函数将仍然返回实际的元素而不只是被抽取出的关键值。

```
(count "foo" #("foo" "bar" "baz") :test #'string=) → 1
(find 'c #((a 10) (b 20) (c 30) (d 40)) :key #'first) → (C 30)
```

为了将这些函数的效果限制在序列实参的特定子序列上，你可以用:**:start**和**:end**参数提供边界指示。为:**:end**传递**NIL**或是省略它与指定该序列的长度具有相同的效果。^②

如果你使用非**NIL**的:**:from-end**参数，那些序列的元素将以相反的顺序被检查。**:from-end**单独使用只能影响**FIND**和**POSITION**的结果。例如：

```
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first) → (A 10)
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first :from-end t) → (A 30)
```

而:**:from-end**参数和另一个关键字参数:**:count**用于指定有多少个元素被移除或替换，这两个参数一起使用时可能影响**REMOVE**和**SUBSTITUTE**的行为。如果指定了一个低于匹配元素数量的:**:count**，那么从哪一端开始显然至关重要：

```
(remove #\a "foobarbaz" :count 1) → "foorbaz"
(remove #\a "foobarbaz" :count 1 :from-end t) → "foobarbz"
```

尽管:**:from-end**无法改变**COUNT**函数的结果，但它确实可以影响传递给任何:**:test**和**:key**函数的元素的顺序，这些函数可能带有副作用。例如：

```
CL-USER> (defparameter *v* #((a 10) (b 20) (a 30) (b 40)))
*V*
```

^① 另一个形参:**:test-not**指定了一个两参数谓词，它可以像:**:test**参数那样使用除了带有逻辑上相反的布尔结果。这个参数已经过时，而目前推荐使用**COMPLEMENT**函数。**COMPLEMENT**接受一个函数参数，然后返回一个带有相同数量参数的函数，其返回与原先函数逻辑上相反的结果。因此你可以而且也应该写成这样：

```
(count x sequence :test (complement #'some-test))
```

而不是这样：

```
(count x sequence :test-not #'some-test)
```

^② 注意，尽管如此，**:start**和**:end**在**REMOVE**和**SUBSTITUTE**的效果仅限于它们所考虑移除或替换的元素，在:**:start**之前和:**:end**之后的元素将原封不动地传递。

```

CL-USER> (defun verbose-first (x) (format t "Looking at ~s~%" x) (first x))
VERBOSE-FIRST
CL-USER> (count 'a *v* :key #'verbose-first)
Looking at (A 10)
Looking at (B 20)
Looking at (A 30)
Looking at (B 40)
2
CL-USER> (count 'a *v* :key #'verbose-first :from-end t)
Looking at (B 40)
Looking at (A 30)
Looking at (B 20)
Looking at (A 10)
2

```

表11-2总结了这些参数。

表11-2 标准序列函数关键字参数

参数	含义	默认值
:test	两参数函数用来比较元素（或由:key函数解出的值）和项	EQL
:key	单参数函数用来从实际的序列元素中解出用于比较的关键字值 NIL表示原样采用序列元素	NIL
:start	子序列的起始索引（含）	0
:end	子序列的终止索引（不含）。NIL表示到序列的结尾	NIL
:from-end	如果为真，序列将以相反的顺序遍历，从尾到头	NIL
:count	数字代表需要移除或替换的元素个数，NIL代表全部。（仅用于 REMOVE和SUBSTITUTE）	NIL

11.5 高阶函数变体

对于每个刚刚讨论过的函数，Common Lisp都提供了两种高阶函数变体，它们接受一个将在每个序列元素上调用的函数，以此来代替项参数。一组变体被命名为与基本函数相同的名字并带有一个追加的-IF。这些函数用于计数、查找、移除以及替换序列中那些函数参数返回真的元素。另一类变体以-IF-NOT后缀命名并计数、查找、移除以及替换函数参数不返回真的元素。

```

(count-if #'evenp #(1 2 3 4 5))          → 2
(count-if-not #'evenp #(1 2 3 4 5))       → 3
(position-if #'digit-char-p "abcd0001")    → 4
(remove-if-not #'(lambda (x) (char= (elt x 0) #\f))
  #("foo" "bar" "baz" "foom")) → #("foo" "foom")

```

根据语言标准，这些-IF-NOT变体已经过时了。但这种过时通常被认为是由于标准本身欠考虑。不过，如果再次修订标准，更有可能被去掉的是-IF而非-IF-NOT系列。比如说，有个叫

REMOVE-IF-NOT的变体就比**REMOVE-IF**更经常使用。尽管它有一个听起来具有否定意义的名字，但**REMOVE-IF-NOT**实际上是一个具有肯定意义的变体——它返回满足谓词的那些元素。^①

除了:**:test**，这些-**IF**和-**IF-NOT**变体都接受和它们的原始版本相同的关键字参数，**:test**不再被需要是因为主参数已经是一个函数了。^②通过使用**:key**参数，由**:key**函数所抽出的值将代替实际元素传递给该函数。

```
(count-if #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) → 2
(count-if-not #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) → 3
(remove-if-not #'alpha-char-p
  #("foo" "bar" "baz") :key #'(lambda (x) (elt x 0))) → #("foo" "bar")
```

REMOVE函数家族还支持第四个变体**REMOVE-DUPLICATES**，它接受序列作为仅需的必要参数，并将其中每个重复的元素移除到只剩下一个实例。除**:count**外，它与**REMOVE**有相同的关键字参数，因为它总是移除所有重复的元素。

```
(remove-duplicates #(1 2 1 2 3 1 2 3 4)) → #(1 2 3 4)
```

11.6 整个序列上的操作

有一些函数每次在整个序列（或多个序列）上进行操作。这些函数比目前已描述的其他函数简单一些。例如，**COPY-SEQ**和**REVERSE**都接受单一的序列参数并返回一个相同类型的新序列。**COPY-SEQ**返回的序列包含与其参数相同的元素，而**REVERSE**返回的序列则含有顺序相反的相同元素。注意，这两个函数都不会复制元素本身，只有返回的序列是一个新对象。

函数**CONCATENATE**创建一个将任意数量序列连接在一起的新序列。不过，跟**REVERSE**和**COPY-SEQ**简单地返回与其单一参数相同类型序列有所不同的是，**CONCATENATE**必须被显式指定产生何种类型的序列，因为其参数可能是不同类型的。它的第一个参数是类型描述符，就像是**MAKE-ARRAY**的**:element-type**参数。这里最常用到的类型描述符是符号**VECTOR**、**LIST**和**STRING**。^③例如：

```
(concatenate 'vector #(1 2 3) '(4 5 6)) → #(1 2 3 4 5 6)
(concatenate 'list #(1 2 3) '(4 5 6)) → (1 2 3 4 5 6)
(concatenate 'string "abc" '(\d \e \f)) → "abcdef"
```

11

①同样的功能由Perl中的grep和Python中的filter所实现。

②作为**:test**参数传递的谓词与作为函数参数传递给-**IF**和-**IF-NOT**函数的谓词之间的区别在于：**:test**谓词是用来将序列元素与特定项相比较的两参数谓词，而-**IF**和-**IF-NOT**谓词是简单测试序列元素的单参数函数。如果原始变体不存在，你可以通过将一个特定的项嵌入到测试函数中，从而用-**IF**版本来实现它们。

```
(count char string ≡)
  (count-if #'(lambda (c) (eql char c)) string)
  (count char string :test #'CHAR-EQUAL) ≡
    (count-if #'(lambda (c) (char-equal char c)) string)
```

③如果让**CONCATENATE**返回一个特化的向量，例如一个字符串，那么参数序列的所有元素都必须是该向量元素类型的实例。

11.7 排序与合并

函数**SORT**和**STABLE-SORT**提供了两种序列排序方式。它们都接受一个序列和一个由两个实参组成的谓词，返回该序列排序后的版本。

```
(sort (vector "foo" "bar" "baz") #'string<) → #("bar" "baz" "foo")
```

它们的区别在于，**STABLE-SORT**可以保证不会重排任何被该谓词视为等价的元素，而**SORT**只保证结果是已排序的并可能重排等价元素。

这两个函数都是所谓的破坏性（destructive）函数。通常出于效率的原因，破坏性函数都会或多或少地修改它们的参数。这有两层含义：第一，你应该总是对这些函数的返回值做一些事情（比如给它赋值一个变量或将它传递给另一个函数）；第二，除非你不再需要传给破坏性函数的那个对象，否则应该传递一个副本。下一章里将讨论更多有关破坏性函数的内容。

在排序以后，你通常不会再关心那个序列的未排序版本，因此在排序的过程中，允许**SORT**和**STABLE-SORT**破坏序列是合理的。但这意味着需要记得要这样来写：^①

```
(setf my-sequence (sort my-sequence #'string<))
```

而不只是这样：

```
(sort my-sequence #'string<)
```

这两个函数也接受关键字参数:**:key**，它和其他序列函数的:**:key**参数一样，应当是一个将被用来从序列元素中抽取出传给排序谓词的值的函数。被抽出的关键字仅用于确定元素顺序，返回的序列将含有参数序列的实际元素。

函数**MERGE**接受两个序列和一个谓词，并返回按照该谓词合并这两个序列所产生的序列。它和两个排序函数之间的关系在于，如果每个序列已经被同样的谓词排序过了，那么由**MERGE**返回的序列也将是有序的。和排序函数一样，**MERGE**也接受一个:**:key**参数。和**CONCATENATE**一样，出于同样原因，**MERGE**的第一个参数必须是用来指定所生成序列类型的类型描述符。

```
(merge 'vector #(1 3 5) #(2 4 6) #'<) → #(1 2 3 4 5 6)
(merge 'list #(1 3 5) #(2 4 6) #'<) → (1 2 3 4 5 6)
```

11.8 子序列操作

另一类函数允许你对已有序列的子序列进行操作，其中最基本的是**SUBSEQ**，它解出序列中从一个特定索引开始并延续到一个特定终止索引或结尾处的子序列。例如：

```
(subseq "foobarbaz" 3) → "barbaz"
(subseq "foobarbaz" 3 6) → "bar"
```

SUBSEQ也支持**SETF**，但不会扩大或缩小一个序列。如果新的值和将被替换的子序列具有不

^① 当传递给排序函数的序列是一个向量时，其破坏性实际上可以确保进行元素的就地交换，因此你可以无需保存返回值而得到正确的效果。尽管如此，总是对返回值做一些事情是好的编程风格，因为排序函数可以以更灵活的方式来修改列表。

同的长度，那么两者中较短的那个将决定有多少个字符被实际改变。

```
(defparameter *x* (copy-seq "foobarbaz"))

(setf (subseq *x* 3 6) "xxx") ; 子序列和新值具有相同的长度。
*x* → "fooxxbaz"

(setf (subseq *x* 3 6) "abcd") ; 新值太长，其他字符被忽略。
*x* → "fooabcbaz"

(setf (subseq *x* 3 6) "xx")    ; 新值太短，只有两个字符被替换。
*x* → "fooxxcbaz"
```

你可以使用**FILL**函数来将一个序列的多个元素设置到单个值上。所需的参数是一个序列以及所填充的值。在默认情况下，该序列的每个元素都设置到该值上。**:start**和**:end**关键字参数可以将效果限制在一个给定的子序列上。

如果你需要在一个序列中查找一个子序列，**SEARCH**函数可以像**POSITION**那样工作，不过第一个参数是一个序列而不是一个单独的项。

```
(position #\b "foobarbaz") → 3
(search "bar" "foobarbaz") → 3
```

另一方面，为了找出两个带有相同前缀的序列首次分岔的位置，可以使用**MISMATCH**函数。它接受两个序列并返回第一对不相匹配的元素的索引。

```
(mismatch "foobarbaz" "foom") → 3
```

如果字符串匹配，它将返回**NIL**。**MISMATCH**也接受许多标准关键字参数：**:key**参数可以指定一个函数用来抽取出被比较的值；**:test**参数用于指定比较函数；而**:start1**、**:end1**、**:start2**和**:end2**参数则用来指定两个序列中的子序列。另外，一个设置为**T**的**:from-end**参数可以指定以相反的顺序搜索序列，从而导致**MISMATCH**返回索引值，这个索引值表示两个序列的相同后缀在第一个序列中的开始位置。

```
(mismatch "foobar" "bar" :from-end t) → 3
```

11

11.9 序列谓词

另外四个常见的函数是**EVERY**、**SOME**、**NOTANY**和**NOTEVERY**，它们在序列上迭代并测试一个布尔谓词。所有这些函数的第一参数是谓词，其余的参数都是序列。这个谓词应当接受与所传递序列相同数量的参数。序列的元素传递给该谓词，每个序列中各取出一个元素，直到某个序列用完所有的元素或满足了整体终止测试条件。**EVERY**在谓词失败时返回假；如果谓词总被满足，它返回真。**SOME**返回由谓词所返回的第一个非**NIL**值，或者在谓词永远得不到满足时返回假。**NOTANY**将在谓词满足时返回假，或者在从未满足时返回真。而**NOTEVERY**在谓词失败时返回真，或是在谓词总是满足时返回假。下面是一些仅在一个序列上测试的例子：

```
(every #'evenp #(1 2 3 4 5)) → NIL
(some #'evenp #(1 2 3 4 5)) → T
```

```
(notany #'evenp #(1 2 3 4 5)) → NIL
(notevery #'evenp #(1 2 3 4 5)) → T
```

下面的调用比较成对的两个序列中的元素:

```
(every #'> #(1 2 3 4) #(5 4 3 2)) → NIL
(some #'> #(1 2 3 4) #(5 4 3 2)) → T
(notany #'> #(1 2 3 4) #(5 4 3 2)) → NIL
(notevery #'> #(1 2 3 4) #(5 4 3 2)) → T
```

11.10 序列映射函数

最后的序列函数是通用映射函数。**MAP**和序列谓词函数一样，接受一个n-参数函数和n个序列。**MAP**不返回布尔值，而是返回一个新序列，它由那些将函数应用在序列的相继元素上所得到的结果组成。与**CONCATENATE**和**MERGE**相似，**MAP**需要被告知其所创建序列的类型。

```
(map 'vector #'* #(1 2 3 4 5) #(10 9 8 7 6)) → #(10 18 24 28 30)
```

MAP-INTO和**MAP**相似，但它并不产生给定类型的新序列，而是将结果放置在一个作为第一个参数传递的序列中。这个序列可以是为函数提供值的序列中的一个。例如，为了将几个向量a、b和c相加到其中一个向量里，可以写成这样：

```
(map-into a #'+ a b c)
```

如果这些序列的长度不同，那么**MAP-INTO**将只影响与最短序列元素数量相当的那些元素，其中也包括那个将被映射到的序列。不过，如果序列被映射到一个带有填充指针的向量里，受影响元素的数量将不限于填充指针而是该向量的实际大小。在对**MAP-INTO**的调用之后，填充指针将被设置成被映射元素的数量。尽管如此，**MAP-INTO**将不会扩展一个可调整大小的向量。

最后一个序列函数是**REDUCE**，它可以做另一种类型的映射：映射在单个序列上，先将一个两参数函数应用到序列的最初两个元素上，再将函数返回值和序列后续元素继续用于该函数。这样，下面的表达式将对从1到10的整数求和：

```
(reduce #'+ #(1 2 3 4 5 6 7 8 9 10)) → 55
```

REDUCE函数非常有用，无论何时，当需要将一个序列提炼成一个单独的值时，你都有机会用**REDUCE**来写它，而这通常是一种相当简洁的表达意图的方法。例如，为了找出一个数字序列中的最大值，可以写成(**reduce #'max numbers**)。**REDUCE**也接受完整的关键字参数(**:key**、**:from-end**、**:start**和**:end**)以及一个**REDUCE**专用的**:initial-value**。后者可以指定一个值，在逻辑上被放置在序列的第一个元素之前（或者，如果你同时指定了一个为真的**:from-end**参数，那么该值被放置在序列的最后一个元素之后）。

11.11 哈希表

Common Lisp提供的另一个通用集合类型是哈希表。与提供整数索引的数据结构的向量有所不同的是，哈希表允许你使用任意对象作为索引或是键(key)。当向哈希表添加值时，可以把它

保存在一个特定的键下。以后就可以使用相同的键来获取该值，或者可以将同一个键关联到一个新值上——每个键映射到单一值上。

不带参数的**MAKE-HASH-TABLE**将创建一个哈希表，其认定两个键等价，当且仅当它们在**EQL**的意义上是相同的对象。这是一个好的默认值，除非你想要使用字符串作为键，因为两个带有相同内容的字符串不一定是**EQL**等价的。在这种情况下，你需要一个所谓的**EQUAL**哈希表，它可以通过将符号**EQUAL**作为:**:test**关键字参数传递给**MAKE-HASH-TABLE**来获得。**:test**参数的另外两个可能的值是符号**EQ**和**EQUALP**。这些都是第4章里讨论过的标准对象比较函数的名字。不过，和传递给序列函数的**:test**参数不同的是，**MAKE-HASH-TABLE**的**:test**不能用来指定一个任意函数——只能是值**EQ**、**EQL**、**EQUAL**和**EQUALP**。这是因为哈希表实际上需要两个函数：一个等价性函数；一个以一种和等价函数最终比较两个键时相兼容的方式，用来从键中计算出一个数值的哈希码的哈希函数。不过，尽管语言标准仅提供了使用标准等价函数的哈希表，但多数实现都提供了一些自定义哈希表的方法。

函数**GETHASH**提供了对哈希表元素的访问。它接受两个参数，即键和哈希表，并返回保存在哈希表中相应键下的值（如果有的话）或是**NIL**。^①例如：

```
(defparameter *h* (make-hash-table))

(gethash 'foo *h*) → NIL

(setf (gethash 'foo *h*) 'quux)

(gethash 'foo *h*) → QUUX
```

由于当键在表中不存在时**GETHASH**返回**NIL**，所以无法从返回值中看出，究竟是键在哈希表中不存在还是键在表中存在却带有值**NIL**。**GETHASH**用一个我尚未讨论到的特性解决了这一问题，即通过多重返回值。**GETHASH**实际上返回两个值：主值是保存在给定键下的值或**NIL**；从值是一个布尔值，用来指示该键在哈希表中是否存在。由于多重返回值的工作方式，除非调用者用一个可以看见多值的形式显式地处理它，否则额外的返回值将被偷偷地丢掉。

我将在第20章里讨论更多关于多重返回值的细节，但目前我将概要地介绍一下如何使用**MULTIPLE-VALUE-BIND**宏来利用**GETHASH**额外返回值。**MULTIPLE-VALUE-BIND**创建类似于**LET**所做的变量绑定，并用一个形式返回的多个值来填充它们。

下面的函数显示了怎样使用**MULTIPLE-VALUE-BIND**，它绑定的变量是**value**和**present**：

```
(defun show-value (key hash-table)
  (multiple-value-bind (value present) (gethash key hash-table)
    (if present
        (format nil "Value ~a actually present." value)
        (format nil "Value ~a because key not found." value)))))

(setf (gethash 'bar *h*) nil) ; provide an explicit value of NIL
```

^① 由于历史上的意外，**GETHASH**的参数顺序与**ELT**相反。**ELT**将集合作为第一个参数，然后是索引，而**GETHASH**将键作为第一个参数，然后是集合。

```
(show-value 'foo *h*) → "Value QUUX actually present."
(show-value 'bar *h*) → "Value NIL actually present."
(show-value 'baz *h*) → "Value NIL because key not found."
```

由于将一个键下面的值设置成NIL会造成把键留在表中，因而你需要另一个函数来完全移除一个键值对。**REMHASH**接受和**GETHASH**相同的参数并移除指定的项。也可以使用**CLRHASH**来完全清除哈希表中的所有键值对。

11.12 哈希表迭代

Common Lisp提供了几种在哈希表项上迭代的方式，其中最简单的方式是通过函数**MAPHASH**。和**MAP**函数相似，**MAPHASH**接受一个两参数函数和一个哈希表，并在哈希表的每一个键值对上调用一次该函数。例如，为了打印哈希表中所有的键值对，可以像这样来使用**MAPHASH**：

```
(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) *h*)
```

在迭代一个哈希表的过程中，向其中添加或移除元素的后果没有被指定（并且可能会很坏），但有两个例外：可以将**SETF**与**GETHASH**一起使用来改变当前项的值，并且可以使用**REMHASH**来移除当前项。例如，为了移除所有其值小于10的项，可以写成下面这样：

```
(maphash #'(lambda (k v) (when (< v 10) (remhash k *h*))) *h*)
```

另一种在哈希表上迭代的方式是使用扩展的**LOOP**宏，我将在第22章里讨论它。^①第一个**MAPHASH**表达式的等价**LOOP**形式如下所示：

```
(loop for k being the hash-keys in *h* using (hash-value v)
      do (format t "~a => ~a~%" k v))
```

关于Common Lisp所支持的非列表集合，我还可以讲更多的内容，例如多维数组以及处理位数组的函数库。但本章中涉及的内容已能满足多数通用编程场合的需要。现在，可以介绍列表这个让Lisp因此得名的数据结构了。

^① **LOOP**的哈希表迭代通常是由更基本的形式**WITH-HASH-TABLE-ITERATOR**来实现的，你不需要担心这点。它被添加到语言里特定用来支持实现诸如**LOOP**这样的东西，并且除非你需要编写迭代在哈希表之上的全新控制构造，否则几乎不会用到它。

LISP名字的由来：列表处理

无 论是出于历史还是实践，列表在Lisp中都扮演着重要的角色。在历史上，列表曾经是Lisp最初的复合数据类型，尽管很多年以来它是这方面唯一的数据类型。现在，Lisp程序员可能会使用向量、哈希表、用户自定义的类或者结构体来代替列表。

从实践上来讲，由于列表对特定问题提供了极佳的解决方案，因此它们仍然能留在语言之中。比如有这样一个问题：如何将代码表示成数据，从而支持代码转换和生成代码的宏。它就是特定于Lisp的，这就可以解释为什么其他语言没有因缺少Lisp式列表所带来的不便。更一般地讲，列表是用于表达任何异构和层次数据的极佳数据结构。另外，它们相当轻量并且支持函数式的编程风格，而这种编程风格也是Lisp传统的另一个重要方面。

因此，你需要更加深入地理解列表。一旦对列表的工作方式有了更加深刻的理解，你将会对如何适时地使用它们有更好的认识。

12.1 “没有列表”

Spoon Boy：不要试图弯曲列表，那是不可能的。你要试着看清真相。

Neo：什么真相？

Spoon Boy：没有列表。

Neo：没有列表？

Spoon Boy：你会发现，弯曲的不是列表，而只是你自己。^①

理解列表的关键在于，要理解它们在很大程度上是一种构建在更基本数据类型实例对象之上的描述。那些更简单的对象是称为点对单元（cons cell）的成对的值，使用函数CONS可以创建它们。

CONS接受两个实参并返回一个含有两个值的新点对单元。^②这些值可以是对任何类型对象的引用。除非第二个值是NIL或是另一个点对单元，否则点对都将打印成在括号中并用一个点分隔两个值的形式，即所谓的“点对”。

12

① 改编自《黑客帝国》(<http://us.imdb.com/Quotes?0133093>)。

② CONS最初是动词construct（构造）的简称。

```
(cons 1 2) → (1 . 2)
```

点对单元中的两个值分别称为**CAR**和**CDR**，它们同时也是用来访问这两个值的函数名。在它们刚出现的年代，这些名字是有意义的，至少对于那些在IBM 704计算机上最早实现Lisp的人们来说是这样的。但即便在那时，它们也只被看作是用来实现这些操作的汇编助记符。然而，这些名字稍显缺乏意义也并不是件很坏的事情。当考虑单独的点对单元时，最好将它们想象成是简单的没有任何特别语义的任意数对。因此：

```
(car (cons 1 2)) → 1
(cdr (cons 1 2)) → 2
```

CAR和**CDR**也都能够支持**SETF**的位置，即给定一个已有的点对单元，有可能将新的值赋给它的任何一个值。^①

```
(defparameter *cons* (cons 1 2))
*cons*           → (1 . 2)
(setf (car *cons*) 10) → 10
*cons*           → (10 . 2)
(setf (cdr *cons*) 20) → 20
*cons*           → (10 . 20)
```

由于点对中的值可以是对任何类型对象的引用，因此可以通过将点对连接在一起，用它们构造出更大型的结构。列表是通过将点对以链状连接在一起而构成的。列表的元素被保存在点对的**CAR**中，而对后续点对的链接则被保存在**CDR**中。链上最后一个单元的**CDR**为**NIL**，正如第4章所提到的那样，它同时代表空列表和布尔值**false**。

这一安排毫无疑问是Lisp所独有的，它被称为单链表。不过，很少有Lisp家族之外的语言会对这种低微的数据类型提供如此广泛的支持。

因此当我讲一个特定的值是一个列表时，其实真正的意思是说它要么是**NIL**要么是对一个点对单元的引用。该点对单元的**CAR**就是该列表的第一个元素，而**CDR**则包含着其余元素，它引用着其他的列表，这有可能是另一个点对单元或**NIL**。Lisp打印机器可以理解这种约定并能将这种链状的点对单元打印成括号列表而不是用点分隔的数对。

```
(cons 1 nil)           → (1)
(cons 1 (cons 2 nil)) → (1 2)
(cons 1 (cons 2 (cons 3 nil))) → (1 2 3)
```

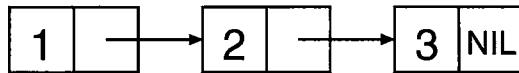
当谈论构建在点对单元之上的结构时，一些图例可以很好地帮助我们理解它们。方框和箭头所组成的图例可以像下面这样将点对单元表示成一对方框。



左边的方框代表**CAR**，而右边的则代表**CDR**。保存在一个特定点对单元中的值要么画在适当

① 当给定**SETF**的位置是**CAR**或**CDR**时，它将展开成一个对函数**RPLACA**或**RPLACD**的调用。和那些仍然使用**SETQ**的一样，一些守旧的Lisp程序员仍然直接使用**RPLACA**和**RPLACD**，但现代风格是使用**CAR**或**CDR**的**SETF**。

的方框之内，要么通过一个从方框指向其所引用值的箭头来表示。^①例如，列表(1 2 3)是由三个点对单元通过它们的CDR链接在一起所构成的，如下所示：



尽管如此，一般在使用列表时并不需要处理单独的点对单元——创建和管理列表的函数将为你做这些事。例如，**LIST**函数可以在背后为你构建一些点对单元并将它们链接在一起。下面的**LIST**表达式等价于前面的**CONS**表达式：

```
(list 1)      → (1)
(list 1 2)    → (1 2)
(list 1 2 3) → (1 2 3)
```

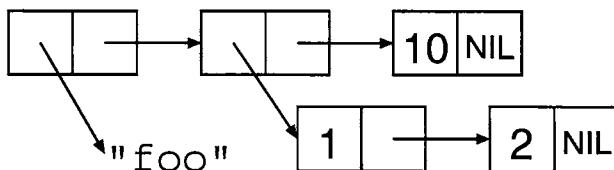
类似地，当从列表的角度考虑问题时，并不需要使用没有意义的名字**CAR**和**CDR**，**FIRST**和**REST**分别是**CAR**和**CDR**的同义词，当处理作为列表的点对时应该使用它们。

```
(defparameter *list* (list 1 2 3 4))
(first *list*)           → 1
(rest *list*)            → (2 3 4)
(first (rest *list*))   → 2
```

因为点对单元可以保存任何类型的值，所以它也可以保存列表。并且单一列表可以保存不同类型的对象。

```
(list "foo" (list 1 2) 10) → ("foo" (1 2) 10)
```

该列表的结构如下所示：



由于列表可以将其他列表作为元素，因此可以用它们来表示任意深度与复杂度的树。由此它们可以成为任何异构和层次数据的极佳表示方式。例如，基于Lisp的XML处理器通常在内部将XML文档表示成列表。另一个明显的树型结构数据的例子就是Lisp代码本身。第30章和第31章将编写一个HTML生成库，其中使用列表的列表来表示被生成的HTML。第13章将介绍如何用点对来表示其他数据结构。

Common Lisp为处理列表提供了一个相当大的函数库。在12.5节和12.6节里将介绍一些更重要的这类函数。但利用取自函数式编程的一些观点来考虑，这些函数将更容易被理解。

^① 在一般情况下，诸如数字这类简单对象画在相应方框的内部，而更复杂的对象画在方框的外部并带有一个来自方框的箭头以指示该引用。这实际上很好地反映了许多Common Lisp实现的工作方式。从概念上来讲，尽管所有对象都是按引用保存的，但特定的简单不可修改的对象可以被直接保存在点对单元里。

12.2 函数式编程和列表

函数式编程的本质在于，程序完全由没有副作用的函数组成，也就是说，函数完全基于其参数的值来计算结果。函数式风格的好处在于它使得程序更易于理解。在消除副作用的同时也消除了所有超距作用的可能。并且由于函数的结果仅取决于其参数的值，因此它的行为更容易被理解和测试。例如，当看到像 $(+ 3 4)$ 这样的表达式时，你知道其结果完全取决于“ $+$ ”函数的定义以及值3和4。你不需要担心程序执行以前发生的事，因为没有什么可以改变该表达式的求值结果。

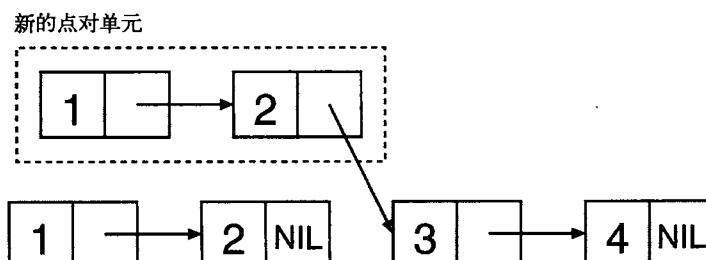
处理数字的函数天生就是函数式的，因为数字都是不可改变的对象。另一方面，如同刚刚看到的那样，通过`SETF`构成本点对单元的`CAR`和`CDR`，列表是可改变的。但列表可以当作函数式数据类型来对待，只要将其值视为是由它们包含的元素所决定的即可。这样，形式 $(1 2 3 4)$ 表示的任何列表在函数式意义上就将等价于任何其他含有这四个值的列表，无论实际表示该列表的是什么点对单元。并且，任何接受一个列表作为实参且其返回值完全依赖于列表内容的函数，也同样可以认为是函数式的。例如，当给定列表 $(1 2 3 4)$ 时，序列函数`REVERSE`总是返回列表 $(4 3 2 1)$ 。但由函数式等价的列表作为实参的不同`REVERSE`调用将返回函数式等价的结果列表。我将在12.6节里讨论的函数式编程的另一个方面即是对高阶函数的使用：函数将其他函数作为数据来对待，接受它们作为实参或是返回它们作为结果。

多数Common Lisp的列表操作函数都以函数式风格写成的。我将在后面讨论如何将函数式风格和其他编码风格混合在一起使用，但首先应当理解函数式风格应用在列表上的一些微妙之处。

多数列表函数之所以会用函数式编写，是因为这能使它们返回与其实参共享点对单元的结果。举一个具体的例子，函数`APPEND`可接受任意数量的列表实参并返回一个含有其参数列表的所有元素的新列表。例如：

```
(append (list 1 2) (list 3 4)) → (1 2 3 4)
```

从函数式观点来看，`APPEND`的工作是返回列表 $(1 2 3 4)$ 而无需修改列表 $(1 2)$ 和 $(3 4)$ 中的任何点对单元。显然，为了实现该目标，可以创建由四个新的点对单元组成的新列表。但这样做并无必要。相反，`APPEND`实际上只用两个新的点对单元来持有值1或2，然后将它们连接在一起，并把第2个点对单元的`CDR`指向最后一个实参——列表 $(3 4)$ ——的头部。然后它返回含有1的那个新生成的点对单元。原先的点对单元都未被修改过，并且结果确实是列表 $(1 2 3 4)$ 。唯一美中不足的是，`APPEND`返回的列表与列表 $(3 4)$ 共享了一些点对单元。产生的结构如下所示：



一般而言，**APPEND**必须复制除最后一个实参以外的所有其他实参，但它的返回结果却总是会与其最后一个实参共享结构。

其他一些函数也相似地利用了列表共享结构的能力。一些像**APPEND**这样的函数被指定总是返回以特定方式共享结构的结果。其他函数则被简单地允许根据具体实现来返回共享的结构。

12.3 “破坏性”操作

如果Common Lisp只是一门纯函数式语言，那么故事就应该到此为止了。不过，因为在一个点对单元被创建之后有可能通过对CAR或CDR进行`SETF`操作来修改它，所以你需要想一想副作用是如何跟结构共享混合的。

由于Lisp的函数式传统，修改已有对象的操作被称作是破坏性的（destructive）。在函数式编程中，改变一个对象的状态相当于“破坏”了它，因为它不再代表相同那个值了。尽管如此，使用同样的术语来描述所有的状态修改操作会在一定程度上产生误解，因为存在两种相当不同破坏性操作，即副作用性（for-side-effect）操作和回收性（recycling）操作。^①

副作用性操作是那些专门利用其副作用的操作。就此而言，所有对`SETF`的使用都是破坏性的，此外还包括诸如`VECTOR-PUSH`或`VECTOR-POP`这类在底层使用`SETF`来修改已有对象状态的函数。但是将这些操作描述成是破坏性的有一点不公平——它们没打算被用于以函数式风格编写的代码中，因此你不该用函数式术语来描述它们。但如果将非函数式的副作用性操作和那些返回结构共享结果的函数混合使用，那么就需要小心不要疏忽地修改了共享的结构。例如，考虑下面三个定义：

```
(defparameter *list-1* (list 1 2))
(defparameter *list-2* (list 3 4))
(defparameter *list-3* (append *list-1* *list-2*))
```

在对这些形式求值之后，你有了三个列表，但是`*list-3*`和`*list-2*`就像前面的图示中的列表那样共享了一些结构。

<code>*list-1*</code>	→ (1 2)
<code>*list-2*</code>	→ (3 4)
<code>*list-3*</code>	→ (1 2 3 4)

现在看看当修改了`*list-2*`时会发生什么：

<code>(setf (first *list-2*) 0)</code>	→ 0
<code>*list-2*</code>	→ (0 4) ; 如你所愿
<code>*list-3*</code>	→ (1 2 0 4) ; 你可能并不想要这种结果

在共享的结构中，由于`*list-2*`中的第一个点对单元也是`*list-3*`中的第三个点对单元，对`*list-2*`的改变也改变了`*list-3*`，对`*list-2*`的`FIRST`进行`SETF`改变了该点对单元中`CAR`部分的值，从而影响了两个列表。

而另一种破坏性操作，即回收性操作，其本来就是用于函数式代码中的。它们的副作用仅是

12

^① for-side-effect是被语言标准所采用的短语，而recycling则是我自己的发明。多数Lisp著作简单地将术语“破坏性”统用在这两类操作上，从而产生了我正试图消除的误解。



一种优化手段。特别地，它们在构造结果时会重用来自它们实参的特定点对单元。尽管如此，和诸如APPEND这种在返回列表中包含未经修改的点对单元的函数有所不同的是，回收性函数将点对单元作为原材料来重用，如有必要它将修改其CAR和CDR来构造想要的结果。这样，只有当调用回收性函数之后不再需要原先列表的情况下，回收性函数才可以被安全地使用。

为了观察回收性函数是怎样工作的，让我们将REVERSE，即返回一个序列的逆序版本的非破坏性函数，与它的回收性版本NREVERSE进行比较。由于REVERSE不修改其参数，它必须为将要逆序的列表的每个元素分配一个新的点对单元。但假如写出了类似下面的代码：

```
(setf *list* (reverse *list*))
```

通过将REVERSE的结果赋值回*list*，你就删除了对*list*原先的值的引用。假设原先列表中的点对单元不被任何其他位置引用，它们现在可以被作为垃圾收集了。不过，在许多Lisp实现中，立即重用已有的点对单元会比分配新的点对单元并让老的变成垃圾更加高效。

NREVERSE就能让你这么做。函数名字中N的含义是non-consing，意思是它不需要分配任何新的点对单元。虽然故意没有说明NREVERSE的明确的副作用（它可以修改列表中任何点对单元的任何CAR或CDR），但典型的实现可能会沿着列表依次改变每个点对单元的CDR，使其指向前一个点对单元。最终返回的点对单元曾经是旧列表的最后一个点对单元，而现在则成为逆序后的列表的头节点。不需要分配新的点对单元，也没有产生垃圾。

像NREVERSE这样的回收性函数大多都带有对应的能产生相同结果但没有破坏性的同伴函数。一般来说，回收性函数和它们的非破坏性同伴带有相同的名字，除了一个起始的字母N。尽管如此，并非所有函数都是这样，其中包括几个更常用的回收性函数。例如NCONC，它是APPEND的回收性版本；而DELETE、DELETE-IF、DELETE-IF-NOT和DELETE-DUPLICATES则是序列函数REMOVE家族的回收性版本。

总之，你可以用与回收性函数的非破坏性同伴相同的方式来使用它们，但只有当你知道参数在函数返回之后不再被使用时，才能安全地使用它们。多数回收性函数的副作用说明并未严格到足以信赖的程度。

但有一组回收性函数带有可靠的明确指定的副作用，这使得事情变得更复杂了。它们是NCONC，即APPEND的回收性版本，以及NSUBSTITUTE和它的-IF和-IF-NOT变体，这些是序列函数SUBSTITUTE及其变体的回收性版本。

和APPEND一样，NCONC返回其列表实参的连结体，但是它以下面的方式构造其结果：对于传递给它的每一个非空列表，NCONC会将该列表的最后一个点对单元的CDR设置成指向下一个非空列表的第一个点对单元。然后它返回第一个列表，后者现在是拼接在一起的结果的开始部分。结果如下所示：

```
(defparameter *x* (list 1 2 3))
(nconc *x* (list 4 5 6)) → (1 2 3 4 5 6)
*x* → (1 2 3 4 5 6)
```

NSUBSTITUTE及其变体可靠地沿着列表实参的列表结构向下遍历，将任何带有旧值的点对单

元的CAR部分SETF到新的值上，否则保持列表原封不动。然后它返回最初的列表，其带有与SUBSTITUTE计算得到的结果相同的值。^①

关于NCONC和NSUBSTITUTE，关键是需要记住，它们是不能依赖于回收性函数的副作用这一规则的例外。忽视它们副作用的可靠性，而像任何其他回收性函数一样来使用它们，只用来产生返回值，这种做法不但完全可以接受，甚至还是一种好的编程风格。

12.4 组合回收性函数和共享结构

尽管可以在函数实参在函数调用之后不会被使用的情况下使用回收性函数，但值得注意的是，如果不小心将一个回收性函数用在了以后会用到的参数上，你肯定会遭遇搬起石头砸自己的脚的境遇。

使事情变得更糟的是，共享结构和回收性函数会用于不同的目的。非破坏性列表函数在点对单元永远不会被修改的假设下返回带有共享结构的列表，但是回收性函数却通过违反这一假设得以正常工作。或者换另一种说法，使用共享结构是基于不在乎究竟由哪些点对单元构成列表这一前提的，而使用回收性函数则要求精确地知道哪些点对单元会在哪里被引用到。

在实践中，回收性函数会有一些习惯用法。其中最常见的一种是构造一个列表，它是由一个在列表前端不断做点对分配操作的函数返回，通常是将元素PUSH进一个保存在局部变量中的列表里，然后返回对其NREVERSE的结果。^②

这是一种构造列表的有效方式，因为每次PUSH都只创建一个点对单元并修改一个局部变量，而NREVERSE只需穿过列表并重新赋值每个元素的CDR。由于列表完全是在函数之内创建，所以完全不存在任何函数之外的代码会引用列表的任何点对单元的风险。下面是一个函数使用该习惯用法来构造一个由从0开始的前n个数字组成的列表：^③

```
(defun upto (max)
  (let ((result nil))
    (dotimes (i max)
      (push i result))
    (nreverse result)))

(upto 10) → (0 1 2 3 4 5 6 7 8 9)
```

12

① 字符串函数NSTRING-CAPITALIZE、NSTRING-DOWNCASE和NSTRING-UPCASE也具有相似的行为——它们返回与其不带N的同伴相同的结果，但被指定在原位修改其字符串参数。

② 例如，在一次对Common Lisp Open Code Collection (CLOCC，即一个由许多开发者所写的功能丰富的库集合) 中所有回收性函数的使用情况进行评测中，PUSH/NREVERSE习惯用法在所有的回收性函数使用中占据了将近一半。

③ 当然，还有其他方法来做到相同的事。例如，扩展的LOOP宏尤其方便做到这一点，并且很可能会生成比PUSH/NREVERSE版本更高效的代码。

```
(defun upto (max)
  (loop for i below max collect i))
```

无论如何，重要的是能够识别PUSH/NREVERSE习惯用法，因为其相当普遍。

还有一个最常见的回收性习惯用法，^①是将回收性函数的返回值立即重新赋值到含有可能会被回收的值的位置上。例如，你经常看到像下面这样的表达式，它使用了**DELETE**，即**REMOVE**的回收性版本：

```
(setf foo (delete nil foo))
```

这将**foo**的值设置到了它的旧值上，只是所有的**NIL**都被移除了。不过，即便是这种习惯用法，你在使用时也需小心一些。如果**foo**和在其他位置上引用的列表共享了一些结构，那么使用**DELETE**来代替**REMOVE**可能会破坏其他那些列表的结构。例如早先那两个共享了它们最后两个点对单元的列表*list-2*和*list-3*：

```
*list-2* → (0 4)
*list-3* → (1 2 0 4)
```

可以像下面这样将4从*list-3*中删除：

```
(setf *list-3* (delete 4 *list-3*)) → (1 2 0)
```

不过，**DELETE**将很可能进行必要的删除，通过将第三个点对单元的**CDR**设置为**NIL**，从而从列表中断开了第四个保存了数字4的点对单元。由于*list-3*的第三个点对单元同时也是*list-2*的第一个点对单元，所以上述操作也改变了*list-2*：

```
*list-2* → (0)
```

如果使用**REMOVE**来代替**DELETE**，它将会构造一个含有值1、2和0的列表，在必要时创建新的点对单元而不会修改*list-3*中的任何点对单元。在这种情况下，*list-2*将不会受到影响。

PUSH/NREVERSE和**SETF/DELETE**的习惯用法很可能占据了80%的回收性函数使用。其他的使用是可能的，但需要小心地跟踪哪些函数返回共享的结构而哪些没有。

总之，当操作列表时，最好是以函数式风格来编写自己的代码——函数应当只依赖于它们的列表实参的内容而不应该修改它们。当然，按照这样的规则将会排除对任何破坏性函数的使用，无论是回收性的还是其他。一旦运行了代码，如果性能评估显示需要进行优化，你可以将非破坏性列表操作替换成相应的回收性操作，但只有当你确定其他任何位置不会引用实参列表时才可以这样做。

最后需要注意的是，当第11章里提到的排序函数**SORT**、**STABLE-SORT**和**MERGE**应用于列表时，它们也是回收性函数。^②不过，这些函数并没有非破坏性的同伴，因此当需要对列表排序而又不破坏它时，你需要传给排序函数一个由**COPY-LIST**生成的列表副本。无论哪种情况，你都需要确保可以保存排序函数的结果，因为原先的实参很可能已经一团糟了。例如：

```
CL-USER> (defparameter *list* (list 4 3 2 1))
*LIST*
CL-USER> (sort *list* #'<)
(1 2 3 4)                                     ; looks good
CL-USER> *list*
(4)                                         ; whoops!
```

① 这一习惯用法在CLOCC代码库里占据了30%的回收性使用。

② **SORT**和**STABLE-SORT**在向量上可被用作副作用性操作，但由于它们仍然返回排序了的向量，你应当忽略这一事实，并出于一致性的目的仅使用它们的返回值。

12.5 列表处理函数

有了前面这些背景知识，现在就可以开始学习Common Lisp为处理列表而提供的函数库了。

前面已经介绍了获取列表中元素的基本函数：**FIRST**和**REST**。尽管可以通过将足够多的**REST**调用（用于深入列表）和一个**FIRST**调用（用于抽取元素）组合起来，以获得一个列表中的任意元素，但这样可能有点冗长。因此，Common Lisp提供了以从**SECOND**到**TENTH**的由其他序数命名的函数来返回相应的元素。而函数**NTH**则更为普遍，它接受两个参数，一个索引和一个列表，并返回列表中第n个（从0开始）元素。类似地，**NTHCDR**接受一个索引和一个列表，并返回调用**CDR**n次的结果。（这样，(**nthcdr** 0 ...)简单地返回最初的列表，而(**nthcdr** 1 ...)等价于**REST**。）但要注意的是，就计算机完成的工作而言，这些函数都不会比等价的**FIRST**和**REST**组合更高效，因此无法在没有跟随n个**CDR**引用的情况下得到一个列表的第n个元素。^①

28个复合**CAR/CDR**函数则是另一个不时会用到的函数家族。每个函数都是通过将由最多四个A和D组成的序列放在C和R之间来命名的，其中每个A代表对**CAR**的调用而每个D代表对**CDR**的调用。因此我们可以得到：

```
(caar list) ≡ (car (car list))
(cadr list) ≡ (car (cdr list))
(cadadr list) ≡ (car (cdr (car (cdr list))))
```

但要注意，这其中许多函数仅当应用于含有其他列表的列表时才有意义。例如，**CAAR**抽取出给定列表的**CAR**的**CAR**，因此传递给它的列表必须含有另一个列表，并将该列表用作其第一个元素。换句话说，这些函数其实是用于树而不是列表的：

```
(caar (list 1 2 3))           → error
(caar (list (list 1 2) 3))    → 1
(cadr (list (list 1 2) (list 3 4))) → (3 4)
(caadr (list (list 1 2) (list 3 4))) → 3
```

现在这些函数不像以前那样常用了，并且即便是最顽固的守旧Lisp黑客也倾向于避免使用过长的组合。尽管如此，它们还是被用在很多古老的Lisp代码上，因此至少应当去理解它们的工作方式。^②

12

^① **NTH**基本上等价于序列函数**ELT**，但只工作在列表上。另外容易使人困惑的是，**NTH**接受其索引作为第一个参数，而**ELT**正相反。另一个区别是，如果你试图在一个大于或等于列表长度的索引上访问一个元素，**ELT**将报错，而**NTH**将返回**NIL**。

^② 特别地，在发明解构参数列表之前，它们通常被用来解出传递给宏的表达式的不同部分。例如，你可以将下列表达式：

```
(when (> x 10) (print x))
```

像下面这样拆开：

```
; the condition
(cadr '(when (> x 10) (print x))) → (> x 10)
; the body, as a list
(cddr '(when (> x 10) (print x))) → ((PRINT X))
```

如果你正在非函数式地使用列表，这些FIRST-TENTH和CAR、CADR等函数也可用作SETF的位置。

表12-1 其他列表处理函数描述

函数	描述
LAST	返回列表的最后一个点对单元。带有一个整数参数时，返回最后n个点对单元
BUTLAST	返回列表的一个副本，最后一个点对单元除外。带有一个整数参数时，排除最后n个单元
NBUTLAST	BUTLAST的回收性版本。可能修改并返回其参数列表但缺少可靠的副作用
LDIFF	返回列表直到某个给定点对单元的副本
TAILP	返回真，如果给定对象是作为列表一部分的点对单元
LIST*	构造一个列表来保存除最后一个参数外的所有参数，然后让最后一个参数成为这个列表最后一个节点的CDR。换句话说，它组合了 LIST和 APPEND
MAKE-LIST	构造一个n项的列表。该列表的初始元素是NIL或者通过:initial-element关键字参数所指定的值
REVAPPEND	REVERSE和APPEND的组合。像REVERSE那样求逆第一个参数，再将其追加到第二个参数上
NRECONC	REVAPPEND的回收性版本。像NREVERSE那样求逆第一个参数，再将其追加到第二个参数上。没有可靠的副作用
CONSP	用来测试一个对象是否为点对单元的谓词
ATOM	用来测试一个对象是否不是点对单元的谓词
LISTP	用来测试一个对象是否为点对单元或NIL的谓词
NULL	用来测试一个对象是否为NIL的谓词。功能上等价于NOT但在测试空列表而非布尔假时文体上推荐使用

12.6 映射

函数式风格的另一个重要方面是对高阶函数的使用，即那些接受其他函数作为参数或将函数作为返回值的函数。前面章节里出现过几个高阶函数，例如MAP。尽管MAP可被同时用于列表和向量（也就是说，任何类型的序列），但Common Lisp另外还提供了6个特定用于列表的映射函数。这6个函数之间的区别在于它们构造结果的方式，以及它们究竟会将函数应用到列表的元素还是列表结构的点对单元上。

MAPCAR是最接近MAP的函数。因为它总是返回一个列表，所以它并不要求MAP所要求的结果类型实参。作为替代，它的第一个参数是想要应用的函数，而后续参数是其元素将为该函数提供实参的列表。除此之外，它和MAP的行为相同：函数被应用在列表实参的相继元素上，每次函数的应用会从每个列表中各接受一个元素。每次函数调用的结果都被收集到一个新列表中。例如：

```
(mapcar #'(lambda (x) (* 2 x)) (list 1 2 3)) → (2 4 6)
(mapcar #'+ (list 1 2 3) (list 10 20 30)) → (11 22 33)
```

MAPLIST也和**MAPCAR**较为相似，它们之间的区别在于**MAPLIST**传递给函数的不是列表元素而是实际的点对单元。^①这样，该函数不仅可以访问到列表中每个元素的值（通过点对单元的**CAR**），还可以访问到列表的其余部分（通过**CDR**）。

除构造结果的方式不同，**MAPCAN**和**MAPCON**与**MAPCAR**和**MAPLIST**的工作方式很相似。**MAPCAR**和**MAPLIST**会构造一个全新的列表来保存函数调用的结果，而**MAPCAN**和**MAPCON**则通过将结果（必须是列表）用**NCONC**拼接在一起产生它们的结果。这样，每次函数调用都可以向结果中提供任意数量的元素。^②**MAPCAN**像**MAPCAR**那样把列表的元素传递到映射函数中，而**MAPCON**则像**MAPLIST**那样来传递点对单元。

最后，函数**MAPC**和**MAPL**是伪装成函数的控制构造，它们只返回第一个列表实参，因此只有当映射函数的副作用有用时，它们才是有用的。**MAPC**是**MAPCAR**和**MAPCAN**的近亲，而**MAPL**属于**MAPLIST/MAPCON**家族。

12.7 其他结构

尽管点对单元和列表通常被视作同义词，但这并不很准确。正如我早先提到的，你可以使用列表的列表来表示树。正如本章所讨论的函数允许你将构建在点对单元上的结构视为列表那样，其他函数也允许你使用点对单元来表示树、集合以及两类键/值映射表。我将在第13章讨论一些这类函数。

① 因此，**MAPLIST**是两个函数中更基本的。如果你只有**MAPLIST**，你可以在它的基础上构造出**MAPCAR**，但你不可能在**MAPCAR**的基础上构造**MAPLIST**。

② 在没有像**REMOVE**这样的过滤函数的Lisp方言中，过滤一个列表的惯用方法是使用**MAPCAN**：

```
(mapcan #'(lambda (x) (if (= x 10) nil (list x))) list) ≡ (remove 10 list)
```

超越列表：点对单元的其他用法



如 同你在前面章节里看到的，列表数据类型是由一组操作点对单元的函数描述的。另外，Common Lisp还提供了一些函数，它们允许你把点对单元构建出的数据结构看作树、集合及查询表。本章将简要介绍这其中的一些数据结构及其处理函数。和列表处理函数一样，在开始编写更复杂的宏以及需要将Lisp代码作为数据处理时，这其中有很多函数会很有用。

13.1 树

由点对单元构建的数据结构既然可看作成列表，自然也可看成是树。毕竟，换另一种思考方式，树不就是一种列表的列表吗？将一组点对单元作为列表来看待的函数与将同样的点对单元作为树来看待的函数，其区别就在于函数将到哪些点对单元里去寻找该列表或树的值。由一个列表函数所查找的点对单元称为列表结构，其查找方式是以第一个点对单元开始，然后跟着CDR引用直到遇到NIL。列表元素就是由列表结构点对单元的CAR所引用的对象。如果列表结构中的一个点对单元带有一个引用到其他点对单元的CAR，那么被引用的点对单元将被视为作为外部列表元素的一个列表的头部。^①而另一方面，树结构则是同时跟随CAR和CDR引用，只要它们指向其他点对单元。因此，树中的值就是该树结构中所有点对单元引用的非点对单元的值。

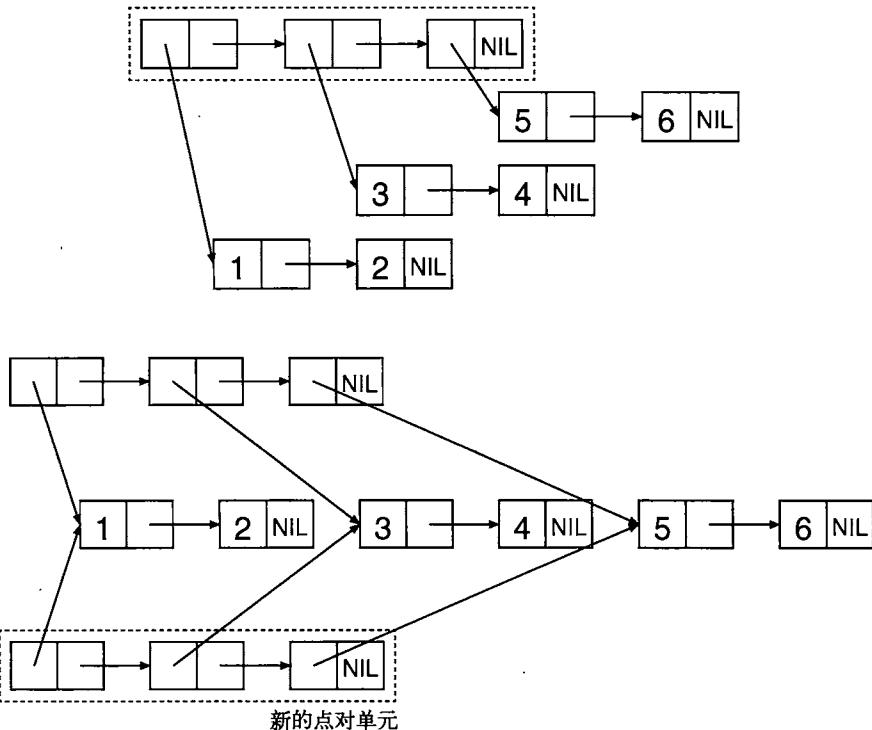
例如，下面的方框和箭头图例显示了构成列表的列表((1 2) (3 4) (5 6))的点对单元。列表结构仅包括虚线框之内的三个点对单元，而树结构则包含全部的点对单元。

为了搞清列表函数和树函数之间的区别，你可以考查一下函数COPY-LIST和COPY-TREE复制这些点对单元的方式。作为一个列表函数COPY-LIST只复制那些构成列表结构的点对单元。也就是说，它根据虚线框之内的每个点对单元生成一个对应的新点对单元。每一个这些新点对单元的CAR均指向与原来列表结构中的点对单元的CAR相同的对象。这样，COPY-LIST就不会复制子

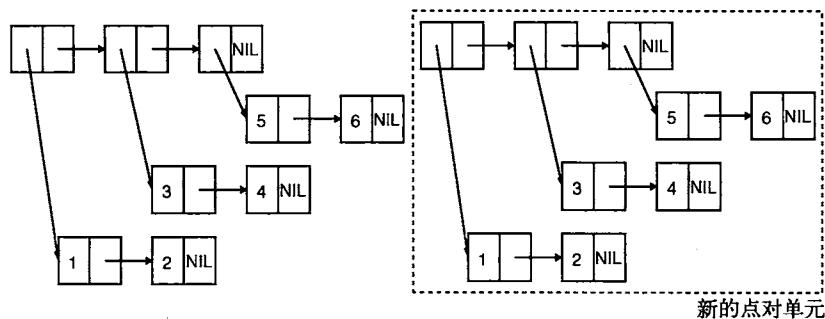
^① 有可能构造出一串点对单元，其中最后一个点对单元的CDR不为NIL而是一些其他的原子。这被称为点列表，因为其最后一个元素前带有一个点。

(cons 1 (cons 2 (cons 3 4))) → (1 2 3 . 4)
没有点的列表（其最后一个CDR为NIL）称为正则列表。

列表(1 2)、(3 4)或(5 6)，如下图所示：



另一方面，**COPY-TREE**将会为图中的每个点对单元都生成一个新的点对单元，并将它们以相同的结构连接在一起，如下图所示：



当原先的点对单元中引用了一个原子值时，复制中的相应点对单元也将指向相同的值。这样，由原先的树和**COPY-TREE**所产生的复制共同引用的唯一对象就是数字1~6以及符号NIL。

另一个在树的点对单元的**CAR**和**CDR**上都进行遍历的函数是**TREE-EQUAL**。它会比较两棵树，当这两棵树具有相同的形状以及它们对应的叶子是**EQL**等价时（或者如果它们满足带有:**:test**关键字参数的测试），函数就认为它们相等。

其他一些以树为中心处理对象的函数包括SUBSTITUTE和NSUBSTITUTE这两个序列函数用于树的类似版本及其-IF和-IF-NOT变体。函数SUBST会像SUBSTITUTE一样接受一个新项、一个旧项和一棵树（跟序列的情况刚好相反），以及:key和:test关键字参数，然后返回一棵与原先的树具有相同形状的新树，只不过其中所有旧项的实例都被替换成新项。例如：

```
CL-USER> (subst 10 1 '(1 2 (3 2 1) ((1 1) (2 2))))
(10 2 (3 2 10) ((10 10) (2 2)))
```

SUBST-IF与**SUBSTITUTE-IF**相似。它接受一个单参数函数而不是一个旧项，该函数在树的每一个原子值上都会被调用，并且当它返回真时，新树中的对应位置将被填充成新值。**SUBST-IF-NOT**也是一样，只不过那些测试返回NIL的值才会被替换。**NSUBST**、**NSUBST-IF**和**NSUBST-IF-NOT**是**SUBST**系列函数的回收性版本。和其他大多数回收性函数一样，只有在明确知道不存在修改共享结构的危险时，才可以将这些函数作为它们非破坏性同伴的原位替代品来使用。特别的是，你必须总是保存这些函数的返回值，因为无法保证其结果与原先的树是EQ等价的。^①

13.2 集合

集合也可以用点对单元来实现。事实上，你可以将任何列表都看作是集合，Common Lisp提供的几个函数可用于对列表进行集合论意义上的操作。但你应当在头脑中牢记，由于列表结构的组织方式，当集合变得更大时，这些操作将会越来越低效。

也就是说，使用内置的集合函数可以轻松地写出集合操作的代码，并且对于小型的集合而言，它们可能会比其他替代实现更为高效。如果性能评估显示这些函数成为代码的性能瓶颈，那么你也总能将列表替换成构建在哈希表或位向量之上的集合。

可以使用函数ADJOIN来构造集合。**ADJOIN**接受一个项和一个代表集合的列表并返回另一个代表集合的列表，其中含有该项和原先集合中的所有项。为了检测该项是否存在，它必须扫描该列表。如果该项没有被找到，那么**ADJOIN**就会创建一个保存该项的新点对单元，并让其指向原先的列表并返回它。否则，它返回原先的列表。

ADJOIN也接受:key和:test关键字参数，它们被用于检测该项是否存在于原先的列表中。和CONS一样，**ADJOIN**不会影响原先的列表——如果打算修改一个特定的列表，则需要将**ADJOIN**返回的值赋值到该列表所来自的位置上。**PUSHNEW**修改宏可以自动做到这点。

```
CL-USER> (defparameter *set* ())
*SET*
CL-USER> (adjoin 1 *set*)
(1)
CL-USER> *set*
NIL
CL-USER> (setf *set* (adjoin 1 *set*))
(1)
```

^① **NSUBST**家族的函数确实可以就地修改树。不过，这里有一种边界情况：当被传递的“树”事实上是一个原子时，它不可能被就地修改，因此**NSUBST**的结果是将与其参数不同的对象：(ns subst 'x 'y 'y) -> x。

```
CL-USER> (pushnew 2 *set*)
(2 1)
CL-USER> *set*
(2 1)
CL-USER> (pushnew 2 *set*)
(2 1)
```

你可以使用`MEMBER`和相关的函数`MEMBER-IF`以及`MEMBER-IF-NOT`来测试一个给定项是否在一个集合中。这些函数与序列函数`FIND`、`FIND-IF`以及`FIND-IF-NOT`相似，不过它们只能用于列表。当指定项存在时，它们并不返回该项，而是返回含有该项的那个点对单元，即以指定项开始的子列表。当指定项不在列表中时，所有三个函数均返回`NIL`。

其余的集合论函数提供了批量操作：`INTERSECTION`、`UNION`、`SET-DIFFERENCE`以及`SET-EXCLUSIVE-OR`。这些函数中的每一个都接受两个列表以及`:key`和`:test`关键字参数，并返回一个新列表，其代表了在两个列表上进行适当的集合论操作所得到的结果：`INTERSECTION`返回一个由两个参数中可找到的所有元素组成的列表。`UNION`返回一个列表，其含有来自两个参数的每个唯一元素的一个实例。^①`SET-DIFFERENCE`返回一个列表，其含有来自第一个参数但并不出现在第二个参数中的所有元素。而`SET-EXCLUSIVE-OR`则返回一个列表，其含有仅来自两个参数列表中的一个而不是两者的那些元素。这些函数中的每一个也都还有一个相应的回收性函数，唯一区别在于后者的名字带有一个前缀`N`。

最后，函数`SUBSETP`接受两个列表以及通常的`:key`和`:test`关键字参数，并在第一个列表是第二个列表的一个子集时返回真，也就是说，第一个列表中的每一个元素也都存在于第二个列表中。列表中元素的顺序无关紧要。

```
CL-USER> (subsetp '(3 2 1) '(1 2 3 4))
T
CL-USER> (subsetp '(1 2 3 4) '(3 2 1))
NIL
```

13.3 查询表：alist 和 plist

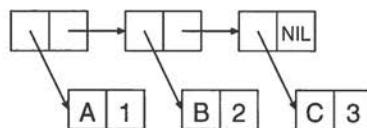
除了树和集合以外，你还可以用点对单元来构建表将键映射到值上。有两类基于点对的查询表会经常用到，这两者都是前面章节里提到过的。它们是关联表，也称为`alist`、属性表以及`plist`。尽管`alist`与`plist`都不能用于大型表（那种情况下你可以使用哈希表），但是值得去了解其使用方式，这既是因为对于小型的表而言，它们可以比哈希表更加高效，同时也是因为它们的一些专有属性十分有用。

`alist`是一种数据结构，它能将一些键映射到值上，同时也支持反向查询，并且当给定一个值时，它还能找出一个对应的键。`alist`也支持添加键/值映射来掩盖已有的映射，并且当这个映射以后被移除时原先的映射还可以再次暴露出来。

从底层来看，`alist`本质上是一个列表，其每一个元素本身都是一个点对单元。每个元素可以

^① `UNION`从每个列表中只接受一个元素，但如果任何一个列表含有重复的元素，那么结果可能也含有重复的元素。

被想象成是一个键值对，其中键保存在点对单元的CAR中而值保存在CDR中。例如，下面是一个将符号A映射到数字1、B映射到2，C映射到3的alist的方框和箭头图例：



除非CDR中的值是一个列表，否则代表键值对的点对单元在表示成S-表达式时将是一个点对(dotted pair)。例如上图所表示的alist将被打印成下面的样子：

```
((A . 1) (B . 2) (C . 3))
```

alist的主查询函数是**ASSOC**，其接受一个键和一个alist并返回第一个CAR匹配该键的点对单元，或是在没有找到匹配时返回**NIL**。

```

CL-USER> (assoc 'a '((a . 1) (b . 2) (c . 3)))
(A . 1)
CL-USER> (assoc 'c '((a . 1) (b . 2) (c . 3)))
(C . 3)
CL-USER> (assoc 'd '((a . 1) (b . 2) (c . 3)))
NIL
  
```

为了得到一个给定键的对应值，可以简单地将**ASSOC**的结果传给**CDR**。

```

CL-USER> (cdr (assoc 'a '((a . 1) (b . 2) (c . 3))))
1
  
```

在默认情况下，指定的键使用**EQL**与alist中的键进行比较，但你可以通过使用:**:key**和:**:test**关键字参数的标准组合来改变这一行为。例如，如果想要用字符串的键，则可以这样写。

```

CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)) :test #'string=)
("a" . 1)
  
```

如果没有指定:**:test**为**STRING=**，**ASSOC**将可能返回**NIL**，因为带有相同内容的两个字符串不一定**EQL**等价。

```

CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)))
NIL
  
```

由于**ASSOC**搜索列表时会从列表的前面开始扫描，因此alist中的一个键值对可以遮盖列表中后面带有相同键的其他键值对。

```

CL-USER> (assoc 'a '((a . 10) (a . 1) (b . 2) (c . 3)))
(A . 10)
  
```

可以像下面这样使用**CONS**向一个alist的前面添加键值对。

```
(cons (cons 'new-key 'new-value) alist)
```

但为方便起见，Common Lisp提供了函数**ACONS**，它可以让你这样写：

```
(acons 'new-key 'new-value alist)
```

和CONS一样，ACONS是一个函数，因此它不能修改用来保存所传递的alist的位置。如果你想要修改alist，你需要这样写成。

```
(setf alist (acons 'new-key 'new-value alist))
```

或

```
(push (cons 'new-key 'new-value) alist)
```

很明显，使用ASSOC搜索一个alist所花的时间是当匹配对被发现时当前列表深度的函数。在最坏情况下，检测到没有匹配的对将需要ASSOC扫描alist的每一个元素。但由于alist的基本机制是如此轻量，故而对于小型的表来说，alist可以在性能上超过哈希表。另外，alist在如何做查询方面也提供了更大的灵活性。我已经提到了ASSOC接受:key和:test关键字参数。当这些还不能满足你的需要时，可以使用ASSOC-IF和ASSOC-IF-NOT函数，其返回CAR部分满足（或不满足，在ASSOC-IF-NOT的情况下）传递到指定项上的测试函数的第一个键值对。并且还有另外3个函数，即RASSOC、RASSOC-IF和RASSOC-IF-NOT，和对应的ASSOC系列函数相似，只是它们使用每个元素的CDR中的值作为键，从而进行反向查询。

函数COPY-ALIST与COPY-TREE相似，除了代替复制整个树结构，它只复制那些构成列表结构的点对单元，外加那些单元的CAR部分直接引用的点对单元。换句话说，原先的alist和它的副本将同时含有相同的对象作为键和值，哪怕这些键或值刚好也由点对单元构成也是如此。

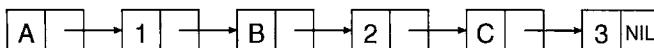
最后，你可以从两个分开的键和值的列表中用函数PAIRLIS构造出一个alist。返回的alist可能含有与原先列表相同或相反顺序的键值对。例如，你可能得到下面这样的结果：

```
CL-USER> (pairlis '(a b c) '(1 2 3))
((C . 3) (B . 2) (A . 1))
```

或者你也可能刚好得到下面这样的效果：

```
CL-USER> (pairlis '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))
```

另一类查询表是属性表或plist，你曾经在第3章里用它来表示数据库中的行。从结构上来讲，plist只是一个正常的列表，其中带有交替出现的键和值作为列表中的值。例如，一个将A、B和C分别映射到1、2和3的plist就是一个简单的列表(A 1 B 2 C 3)。用方框和箭头的形式来表示，它看起来像这样：



不过，plist不像alist那样灵活。事实上，plist仅支持一种基本查询操作，即函数GETF，其接受一个plist和一个键，返回所关联的值或是在键没有被找到时返回NIL。GETF也接受一个可选的第三个参数，它将在键没有被找到时代替NIL作为返回值。

与ASSOC不同，其使用EQ作为默认测试并允许通过:test参数提供一个不同的测试函数，GETF总是使用EQ来测试所提供的键是否匹配plist中的键。因此，你一定不能用数字和字符作为plist中的键。正如你在第4章里看到的那样，EQ对于这些类型的行为在本质上是未定义的。从实

实际上讲，一个plist中的键差不多总是符号，这是合理的，因为plist最初被发明用于实现符号“属性”，即名字和值之间的任意映射。

你可以将`SETF`与`GETF`一起使用来设置与给定键关联的值。`SETF`也会稍微特别地对待`GETF`，`GETF`的第一个参数被视为将要修改的位置。这样，你可以使用`GETF`的`SETF`来向一个已有的plist中添加新的键值对。

```
CL-USER> (defparameter *plist* ())
*PLIST*
CL-USER> *plist*
NIL
CL-USER> (setf (getf *plist* :a) 1)
1
CL-USER> *plist*
(:A 1)
CL-USER> (setf (getf *plist* :a) 2)
2
CL-USER> *plist*
(:A 2)
```

为了从plist中移除一个键/值对，你可以使用宏`REMFF`，它将作为其第一个参数给定的位置设置成含有除了指定的那个以外的所有键值对的plist。当给定的键被实际找到时，它返回真。

```
CL-USER> (remff *plist* :a)
T
CL-USER> *plist*
NIL
```

和`GETF`一样，`REMFF`总是使用`EQ`来比较给定的键和plist中的键。

由于plist经常被用于想从同一个plist中抽取出几个属性的场合，所以Common Lisp还提供了一个函数`GET-PROPERTIES`，它能更高效地从单一plist中抽取出多个值。它接受一个plist和一个需要被搜索的键的列表，并返回多个值：第一个被找到的键、其对应的值，以及一个以被找到的键开始的列表的头部。这可以允许你处理一个属性表，抽取出想要的属性，而无需持续地从列表的开始处重新扫描。例如，下面的函数使用假想的函数`process-property`有效地处理用于指定键列表的Plist中的所有键/值对。

```
(defun process-properties (plist keys)
  (loop while plist do
    (multiple-value-bind (key value tail) (get-properties plist keys)
      (when key (process-property key value))
      (setf plist (cddr tail)))))
```

关于plist，最后特别要指出的是它们与符号之间的关系：每一个符号对象都有一个相关联的plist，以便用来保存关于该符号的信息。这个plist可以通过函数`SYMBOL-PLIST`获取到。但你很少需要关心整个plist，更常见的情况是使用函数`GET`，其接受一个符号和一个键，功能相当于在符号的`SYMBOL-PLIST`上对同一个键使用`GETF`。

```
(get 'symbol 'key) ≡ (getf (symbol-plist 'symbol) 'key)
```

和GETF一样，GET也可以用SETF来操作，因此你可以像下面这样将任意信息附加到一个符号上：

```
(setf (get 'some-symbol 'my-key) "information")
```

为了从一个符号的plist中移除属性，你可以使用SYMBOL-PLIST上的REMF或是更便捷的函数REMPROP。^①

```
(remprop 'symbol 'key) ≡ (remf (symbol-plist 'symbol key))
```

向名字中附加任意信息对于任何类型的符号编程来说都是很有用的。例如，第24章将编写一个宏，它将向名字中附加信息，以便同一个宏的其他实例能将其抽取出并用于生成它们的展开式。

13.4 DESTRUCTURING-BIND

最后一个我需要介绍的，同时也是你将在后续章节里使用的一个用于拆分列表的工具是DESTRUCTURING-BIND宏。这个宏提供了一种解构(destructure)任意列表的方式，这类似于宏形参列表分拆它们的参数列表的方式。DESTRUCTURING-BIND的基本骨架如下所示：

```
(destructuring-bind (parameter*) list
  body-form*)
```

该参数列表可以包含宏参数列表中支持的任何参数类型，比如`&optional`、`&rest`和`&key`参数。^②并且，如同在宏参数列表中一样，任何参数都可以被替换成一个嵌套的解构参数列表，从而将一个原本绑定在单个参数上的列表拆开。其中的list形式被求值一次并且应当返回一个列表，其随后被解构并且适当的值会被绑定到形参列表的对应变量中，然后那些body-form将在这些绑定的作用下被求值。一些简单的例子如下所示：

```
(destructuring-bind (x y z) (list 1 2 3)
  (list :x x :y y :z z)) → (:X 1 :Y 2 :Z 3)

(destructuring-bind (x y z) (list 1 (list 2 20) 3)
  (list :x x :y y :z z)) → (:X 1 :Y (2 20) :Z 3)

(destructuring-bind (x (y1 y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) → (:X 1 :Y1 2 :Y2 20 :Z 3)

(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) → (:X 1 :Y1 2 :Y2 20 :Z 3)

(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) → (:X 1 :Y1 2 :Y2 NIL :Z 3)
```

13

^① 直接SETF SYMBOL-PLIST也是有可能的。不过这是一个坏主意，因为不同的代码可能出于不同的原因添加了不同的属性到符号的plist上。如果一段代码清除了该符号的整个plist，它可能干扰其他向plist中添加自己的属性的代码。

^② 宏参数列表确实支持一种参数类型，即`&environment`参数，而DESTRUCTURING-BIND不支持。不过，我没有在第8章里讨论这种参数类型，并且你现在也不需要考虑它。

```
(destructuring-bind (&key x y z) (list :x 1 :y 2 :z 3)
  (list :x x :y y :z z)) → (:X 1 :Y 2 :Z 3)
```

```
(destructuring-bind (&key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z)) → (:X 3 :Y 2 :Z 1)
```

另外还有一种参数（尽管第8章并未介绍），它既可以用在DESTRUCTURING-BIND中，也可以用在宏参数列表中，这就是`&whole`。如果被指定，它必须是参数列表中的第一个参数，并且它会绑定到整个列表形式上。^①在一个`&whole`参数之后，其他参数可以像通常那样出现并且将像没有`&whole`参数存在那样抽出列表中的指定部分。一个将`&whole`与DESTRUCTURING-BIND一起使用的例子如下所示：

```
(destructuring-bind (&whole whole &key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z :whole whole))
→ (:X 3 :Y 2 :Z 1 :WHOLE (:Z 1 :Y 2 :X 3))
```

你将在一个宏里使用`&whole`参数，它是将在第31章里开发的HTML生成库的一部分。不过，在那之前，我还要谈及更多的主题。在关于点对单元的两章相当Lisp化的主题之后，下面将介绍如何处理文件和文件名这种相对乏味的问题。

^① 当一个`&whole`参数被用在宏参数列表中时，它所绑定的形式是整个宏形式，包括该宏的名字。

Common Lisp提供了一个功能丰富的用于处理文件的函数库。在本章里，我将把重点放在少数基本的文件相关的任务上：读写文件以及列出文件系统中的文件。对这些基本任务，Common Lisp 的I/O机制与其他语言相似。Common Lisp为读写数据提供了一个流的抽象和一个称为路径名（pathname）的抽象，它们以一种与操作系统无关的方式来管理文件名。另外，Common Lisp还提供了其他一些只有Lisp才有的功能，比如读写S-表达式。

14.1 读取文件数据

最基本的文件I/O任务是读取文件的内容。可以通过**OPEN**函数获得一个流并从中读取文件的内容。默认情况下，**OPEN**返回一个基于字符的输入流，你可以将它传给许多函数以便读取文本中的一个或多个字符：**READ-CHAR**读取单个字符；**READ-LINE**读取一行文本，去掉行结束字符后作为一个字符串返回；而**READ**读取单一的S-表达式并返回一个Lisp对象。当完成了对流的操作后，你可以使用**CLOSE**函数来关闭它。

OPEN的唯一必要参数是需要读取的文件名。如同你将会在14.6节里看到的那样，Common Lisp提供了许多表示文件名的方式，但最简单的方式是使用一个含有以本地文件命名语法表示的文件名的字符串。因此，假设`/some/file/name.txt`是一个文件，那么可以像下面这样打开它：

```
(open "/some/file/name.txt")
```

你可以把返回对象作为任何读取函数的第一个参数。例如，为了打印文件的第一行，你可以组合使用**OPEN**、**READ-LINE**和**CLOSE**，如下所示：

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

当然，在试图打开和读取一个文件时可能会出现一些错误。该文件可能不存在或者可能在读取时无意中遇到了文件结尾。默认情况下，**OPEN**和**READ-***系列函数将在出现这些情况时报错。在第19章里，我将讨论如何从这类错误中恢复。不过眼下有一个更轻量级的解决方案：每个这样的函数都可接受参数来修改这些异常情况下的行为。

如果你想打开一个可能不存在的文件而又不想让**OPEN**报错，那么可以使用关键字参数：`:if-does-not-exist`来指定不同的行为。三个可能的值是：`:error`，报错（默认

值)；`:create`，继续进行并创建该文件，然后就像它已经存在那样进行处理；`NIL`，让它返回`NIL`来代替一个流。这样，你就可以改变前面的示例来处理文件可能不存在的情况。

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))
```

读取函数，即`READ-CHAR`、`READ-LINE`和`READ`，都接受一个可选的参数，其默认值为真并指定当函数在文件结尾处被调用时是否应该报错。如果该参数为`NIL`，它们在遇到文件结尾时将返回它们第三个参数的值，默认为`NIL`。因此，可以像下面这样打印一个文件的所有行：

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
          while line do (format t "~a~%" line))
    (close in)))
```

在这三个文本读取函数中，`READ`是Lisp独有的。这跟提供了REPL中R部分的函数是同一个，它用于读取Lisp源代码。当每次被调用时，它会读取单一的S-表达式，跳过空格和注释，然后返回由S-表达式代表的Lisp对象。例如，假设`/some/file/name.txt`带有下列内容：

```
(1 2 3)
456
"a string" ; this is a comment
((a b)
 (c d))
```

换句话说，它含有四个S-表达式：一个数字列表、一个数字、一个字符串和一个列表的列表。你可以像下面这样读取这些表达式：

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"a string"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

如同第3章所述，你可以使用`PRINT`以“可读的”形式打印Lisp对象。这样，每当你需要在文件中保存一点数据时，`PRINT`和`READ`就提供了一个做这件事的简单途径，而无须设计一套数据格式或编写一个解析器。它们甚至可以让你自由地添加注释，如同前面的示例所演示的那样。并且由于S-表达式被设计成是供人编辑的，所以它也是用于诸如配置文件等事务的良好格式。^①

^① 尽管如此，注意Lisp读取器知道如何跳过注释，它会完全跳过它们。这样，如果你使用`READ`来读取一个含有注释的配置文件，然后使用`PRINT`来保存对数据的修改，那么你将失去那些注释。

14.2 读取二进制数据

默认情况下，**OPEN**返回字符流，它根据特定的字符编码方案将底层字节转化成字符。^①为了读取原始字节，你需要向**OPEN**传递一个值为'(*unsigned-byte 8*)'的:**:element-type**参数。^②你可以将得到的流传给**READ-BYTE**，它将在每次被调用时返回0~255的整数。与字符读取函数一样，**READ-BYTE**也支持可选的参数以便指定当其被调用在文件结尾时是否应该报错，以及在遇到结尾时返回何值。第24章将构建一个库，它允许使用**READ-BYTE**来便利地读取结构化的二进制数据。^③

14.3 批量读取

最后一个读取函数**READ-SEQUENCE**可同时工作在字符和二进制流上。你传递给它一个序列（通常是一个向量）和一个流，然后它会尝试用来自流的数据填充该序列。它返回序列中第一个没有被填充的元素的索引，或是在完全填充的情况下返回该序列的长度。你也可以传递:**:start**和:**:end**关键字参数来指定一个应当被代替填充的子序列。该序列参数的元素类型必须足以保存带有该流元素类型的元素。由于多数操作系统支持某种形式的块I/O，**READ-SEQUENCE**通常比重复调用**READ-BYTE**或**READ-CHAR**来填充一个序列更加高效。

14.4 文件输出

为了向一个文件中写数据，你需要一个输出流，你可以通过在调用**OPEN**时使用一个值为:**:output**的:**:direction**关键字参数来获取它。当你打开一个用于输出的文件时，**OPEN**会假设该文件不该存在并会在文件存在时报错。但你可以使用:**:if-exists**关键字参数来改变该行为。传递值:**:supersede**可以告诉**OPEN**来替换已有文件。传递:**:append**将导致**OPEN**打开已有的文件并保证新数据被写到文件结尾处，而:**:overwrite**返回一个从文件开始处开始的流从而覆盖已有数据。而传递**:NIL**将导致**OPEN**在文件已存在时返回**:NIL**而不是流。一个典型的使用**OPEN**来输出的例子如下所示：

```
(open "/some/file/name.txt" :direction :output :if-exists :supersede)
```

Common Lisp也提供了几个用于写数据的函数：**WRITE-CHAR**会向流中写入一个单一字符；**WRITE-LINE**写一个字符串并紧跟一个换行，其将被输出成用于当前平台的适当行结束字符或字

^① 默认情况下，**OPEN**使用当前操作系统的默认字符编码，但它也接受一个关键字参数:**:external-format**，它可以传递由实现定义的值来指定一个不同的编码。字符流也会转换平台相关的行结束序列到单一字符#\Newline上。

^② 类型(*unsigned-byte 8*)代表8位字节。Common Lisp的“字节”类型并不是固定大小的，由于Lisp曾经运行在不同时期的体系结构上，其字节长度为6~9位。更不用说还有PDP-10计算机，其带有可独立寻址的长度为1~36比特的变长字节域。

^③ 一般情况下，一个流要么是字符流要么是二进制流，因此你不能混和调用**READ-BYTE**、**READ-CHAR**或者其他基于字符的函数。不过某些实现，例如Allegro，支持所谓的二义流(bivalent stream)，其同时支持字符和二进制I/O。

符序列。另一个函数`WRITE-STRING`写一个字符串而不会添加任何行结束符。有两个不同的函数可以只打印一个换行：`TERPRI`是“终止打印”(terminate print)的简称，即无条件地打印一个换行字符；`FRESH-LINE`打印一个换行字符，除非该流已经在一行的开始处。在想要避免由按顺序调用的不同函数所生成的文本输出中的额外换行时，`FRESH-LINE`很有用。例如，假设一个函数在其生成输出时总是带有一个换行，而另一个函数应当每次从一个新行开始输出。但假设这两个函数被依次调用，而你又不希望在两个输出操作之间产生一个空行，如果你在第二个函数开始处使用`FRESH-LINE`，那么它的输出将总是从一个新行开始。但如果它刚好在前一个函数之后调用，则它将不会产生一个额外换行。

一些函数会将Lisp数据输出成S-表达式：`PRINT`打印一个S-表达式，前缀一个换行及一个空格；`PRIN1`只打印S-表达式；而函数`PPRINT`会像`PRINT`和`PRIN1`那样打印S-表达式，但使用的是美化打印器(pretty printer)，它会试图将输出打印得赏心悦目。

但并非所有对象都能以一种`READ`可理解的形式打印出来。当试图使用`PRINT`、`PRIN1`或`PPRINT`来打印这样一种对象时，变量`*PRINT-READABLY*`将会予以控制。当它是`NIL`时，这些函数将以一种导致`READ`在试图读取时肯定会报错的特殊语法来打印该对象；否则它们将直接报错而不打印该对象。

另一个函数`PRINC`也会打印Lisp对象，但其工作方式很适合人们使用。例如，`PRINC`在打印字符串时不带有引号。你可以使用极其灵活但有时略显神秘的`FORMAT`函数来生成更加复杂的文本输出。我将在第18章里讨论一些关于`FORMAT`的更重要的细节，它从本质上定义了一种用于产生格式化输出的微型语言。

为了向一个文件中写入二进制数据，你需要在使用`OPEN`打开文件时带有与读取该文件时相同的`:element-type`实参，其值为`'(unsigned-byte 8)`，然后就可以使用`WRITE-BYTE`向流中写入单独的字节。

批量输出函数`WRITE-SEQUENCE`可同时接受二进制和字符流，只要序列中的所有元素都是用于该流的适当类型即可，无论其是字符还是字节。和`READ-SEQUENCE`一样，该函数会比每次输出一个序列元素更为高效。

14.5 关闭文件

任何编写过处理大量文件代码的人都知道，当处理完文件之后，关闭它们是多么重要。因为文件句柄往往是稀缺资源，如果打开一些文件却不关闭它们，你很快会发现不能再打开更多文件了。^①确保每一个`OPEN`都有一个匹配的`CLOSE`可能是非常显而易见的。例如，完全可以像下面这样来组织文件使用代码：

^① 有些人可能认为这在诸如Lisp的垃圾收集型语言里不是个问题。在多数Lisp实现中，当一个流变成垃圾之后都将会自动关闭。不过这种行为不能被依赖——问题在于垃圾收集器通常只在内存变少时才运行，它们不认识诸如文件句柄这样的稀缺资源。如果有大量的可用内存，就可以轻易地在垃圾收集器运行之前用光所有的文件句柄。

```
(let ((stream (open "/some/file/name.txt")))
  ;; do stuff with stream
  (close stream))
```

但这一方法还是有两方面的问题。一是容易出现错误——如果忘记使用CLOSE，那么代码将在每次运行时泄漏一个文件句柄。而更重要的一点是，该代码并不保证你能够到达CLOSE那里。例如，如果CLOSE之前的代码含有一个RETURN或RETURN-FROM，那就会在没有关闭流的情况下离开LET语句块。或者如同第19章里将要介绍的，如果任何CLOSE之前的代码产生了一个错误，那么控制流可能就会跳出LET语句块而转到一个错误处理器中，然后不再回来关闭那个流。

Common Lisp对于如何确保一直运行特定代码这一问题提供了一个通用的解决方案：特殊操作符UNWIND-PROTECT，第20章将予以讨论。不过因为这种打开文件，对产生的流做一些事情，然后再关闭流的模式是如此普遍，Common Lisp就提供了一个构建在UNWIND-PROTECT之上的宏WITH-OPEN-FILE来封装这一模式。下面是它的基本形式：

```
(with-open-file (stream-var open-argument*)
  body-form*)
```

其中body-form中的形式将在stream-var被绑定到一个文件流的情况下进行求值，该流由一个以open-argument为实参的OPEN调用而打开。WITH-OPEN-FILE会确保stream-var中的流在WITH-OPEN-FILE返回之前被关闭。因此，从一个文件中读取一行的代码应如下所示。

```
(with-open-file (stream "/some/file/name.txt")
  (format t "~a%" (read-line stream)))
```

为了创建一个新文件，你可以这样写。

```
(with-open-file (stream "/some/file/name.txt" :direction :output)
  (format stream "Some text."))
```

在你所使用的90%~99%的文件I/O中都可能会用到WITH-OPEN-FILE——需要使用原始OPEN和CLOSE调用的唯一情况是，当需要在一个函数中打开一个文件并在函数返回之后仍然保持所产生的流时。在那种情况下必须注意一点，最终要由你自己来关闭这个流，否则你将会泄漏文件描述符并可能最终无法打开更多文件。

14.6 文件名

到目前为止，文件名都是用字符串表示的。但使用字符串作为文件名会将代码捆绑在特定操作系统和文件系统上。而且如果按照一个特定的文件命名方案的规则（比如说，使用“/”来分隔目录）用程序来构造文件名，那么你就会将代码捆绑到特定的文件系统上。

为了避免这种不可移植性，Common Lisp提供了另一种文件名的表示方式：路径名(pathname)对象。路径名以一种结构化的方式来表示文件名，这种方式使得它们易于管理而无须捆绑在特定的文件名语法上。而在以本地语法写成的字符串，即名字字符串(namestring)，和路径名之间进行来回转换的责任则被放在了Lisp实现身上。

不幸的是，如同许多被设计用于隐藏本质上不同的底层系统细节的抽象那样，路径名抽象也



引入了它们自己的复杂性。当路径名最初被设计时，通常使用的文件系统集合比今天所使用的更加丰富多彩。这带来的结果是，在你只关心如何表示Unix或Windows文件名时，路径名抽象的某些细微之处就没有什么意义了。不过，一旦你理解了路径名抽象中的哪些部分可以作为路径名发展史中的遗留产物而忽略时，你就会发现它们确实提供了一种管理文件名的便捷方式。^①

在多数使用文件名的调用场合里，你都可以同时使用名字字符串或是路径名。具体使用哪个在很大程度上取决于该名字的来源。由用户提供的文件名（例如作为参数或是配置文件中的值）通常是名字字符串，因为用户只知道它们所运行在的文件系统，而不关心Lisp如何表示文件名的细节。但通过编程方法产生的文件名是路径名，因为你能移植地创建它们。一个由OPEN返回的流也代表文件名，也就是那个当初用来打开该流的文件名。这三种类型的文件名被总称为路径名描述符(pathname designator)。所有内置的以文件名作为参数的函数都能接受所有这三种路径名描述符。例如，前面章节里所有的用字符串来表示文件名的位置都同样可以传入路径名对象或流。

进化历程

存在于20世纪七八十年代的文件系统的历史多样性很容易被遗忘。Kent Pitman，Common Lisp标准的主要技术编辑之一，有一次在comp.lang.lisp (Message-ID: sfwzo74np6w.fsf@world.std.com) 新闻组上描述了如下情形。

在Common Lisp的设计完成时期，处于支配地位的文件系统是：TOPS-10、TENEX、TOPS-20、VAX VMS、AT&T Unix、MIT Multics、MIT ITS，更不用说还有许多大型机操作系统了。它们中的一些只支持大写字母，一些是大小写混合的，另一些则是大小写敏感但却能自动作大小写转换（就像Common Lisp）。它们中的一些将目录视为文件，而另一些则不会。一些对于特殊的文件字符带有引用字符，另一些不会。一些带有通配符，而另一些没有。一些在相对路径名中使用:up，另一些不这样做。一些带有可命名的根目录，而另一些没有。还存在没有目录的文件系统，使用非层次目录结构的文件系统，不支持文件类型的文件系统，没有版本的文件系统以及没有设备的文件系统，等等。

如果从任何单一文件系统的观点上观察路径名抽象，那么它看起来显得过于复杂。不过，如果考察两种像Windows和Unix这样相似的文件系统，你可能已经开始注意路径名系统可能帮你抽象掉的一些区别了。例如，Windows文件名含有一个驱动器字母，而Unix文件名却没有。使用这种用来处理过去存在的广泛的文件系统的路径名抽象带来的另一种好处是，它有可能可以处理将来可能存在的文件系统。比如说，如果版本文件系统重新流行起来的话，Common Lisp就已经准备好了。

① 路径名系统从某种意义上被认为结构复杂的另一个原因是其对逻辑路径名(logical pathname)的支持。不过，你可以完美地使用路径名系统的其余部分而无需了解任何关于逻辑路径名更多的东西，从而安全地忽略它们。简洁地说，逻辑路径名允许Common Lisp程序含有对路径名的引用而无须命名特定的文件。逻辑路径名随后可以在一个实际的文件系统中被映射到特定的位置上，前提是当程序被安装时，通过定义“逻辑路径名转换”(logical pathname translation)来将匹配特定通配符的文件路径名转化成代表文件系统中文件的路径名，也就是所谓的物理路径名。它们在特定场合下有其自己的用途，但是你可以足够远离且无须担心它们。

14.7 路径名如何表示文件名

路径名是一种使用6个组件来表示文件名的结构化对象：主机（host）、设备（device）、目录（directory）、名称（name）、类型（type）以及版本（version）。这些组件的多数都接受原子值，通常是字符串。只有目录组件有其进一步的结构，含有一个目录名（作为字符串）的列表，其中带有关键字：`:absolute`或`:relative`作为前缀。但并非所有路径名组件在所有平台上都是必需的——这也是路径名让许多初级Lisp程序员感到无端复杂的原因之一。另一方面，你真的不必担心哪个组件在特定文件系统上是否可被用来表示文件名，除非你需要手工地从头创建一个新路径名对象。相反，通常将通过让具体实现来把一个文件系统相关的名字字符串解析到一个路径名对象，或是通过从一个已有路径名中取得其多数组件来创建新路径名，从而得到路径名对象。

例如，为了将名字字符串转化成路径名，你可以使用`PATHNAME`函数。它接受路径名描述符并返回等价的路径名对象。当该描述符已经是一个路径名时，它就会被简单地返回。当它是一个流时，最初的文件名就会被抽取出然后返回。不过当描述符是一个名字字符串时，它将根据本地文件名语法来解析。作为一个平台中立的文档，语言标准没有指定任何从名字字符串到路径名的特定映射，但是多数实现遵守了与其所在操作系统相同的约定。

在Unix文件系统上，只有目录、名称和类型组件通常会被用到。在Windows上，还有一个组件（通常是设备或主机）保存了驱动器字母。在这些平台上，一个名字字符串在解析时首先会被路径分隔符——在Unix上是一个斜杠而在Windows上是一个斜杠或反斜杠——分拆成基本元素。在Windows上，驱动器字母要么被放置在设备中，要么就是在主机组件中。其他名字元素除最后一个之外都被放置在一个以`:absolute`或`:relative`开始的列表中，具体取决于该名字是否（如果有的话，忽略驱动器字母）以一个路径分隔符开始。这个列表将成为路径名的目录组件。如果有的话，最后一个元素将在最右边的点处被分拆开，然后得到的两部分将被放进路径名的名称和类型组件中。^①

你可以使用函数`PATHNAME-DIRECTORY`、`PATHNAME-NAME`和`PATHNAME-TYPE`来检查一个路径名中的单独组件。

```
(pathname-directory (pathname "/foo/bar/baz.txt")) → (:ABSOLUTE "foo" "bar")
(pathname-name (pathname "/foo/bar/baz.txt"))      → "baz"
(pathname-type (pathname "/foo/bar/baz.txt"))       → "txt"
```

其他三个函数`PATHNAME-HOST`、`PATHNAME-DEVICE`和`PATHNAME-VERSION`允许你访问其他三个路径名组件，尽管它们在Unix上不太可能带有感兴趣的值。在Windows上，`PATHNAME-HOST`和`PATHNAME-DEVICE`两者之一将返回驱动器字母。

14

^①许多基于Unix的实现特别地对待那些最后一个元素以点开始并且不含有任何其他点的文件名，将整个元素包括点在内放置在名称组件中，并且保留类型组件为`NIL`。

```
(pathname-name (pathname "/foo/.emacs")) → ".emacs"
(pathname-type (pathname "/foo/.emacs")) → NIL
```

尽管如此，并非所有实现都遵守这一约定。一些实现创建一个以空字符串作为名称同时将“`emacs`”作为类型的路径名。

和许多其他内置对象一样，路径名也有其自身的读取语法：#p后接一个双引号字符串。这允许你打印并且读回含有路径名对象的S-表达式，但由于其语法取决于名字字符串解析算法，这些数据在操作系统之间不一定可移植。

```
(pathname "/foo/bar/baz.txt") → #p"/foo/bar/baz.txt"
```

为了将一个路径名转化回一个名字字符串，例如，为了呈现给用户，你可以使用函数NAMESTRING，其接受一个路径名描述符并返回一个名字字符串。其他两个函数DIRECTORY-NAMESTRING和FILE-NAMESTRING返回一个部分名字字符串。DIRECTORY-NAMESTRING将目录组件的元素组合成一个本地目录名，而FILE-NAMESTRING则组合名字和类型组件。^①

```
(namestring #p"/foo/bar/baz.txt") → "/foo/bar/baz.txt"
(directory-namestring #p"/foo/bar/baz.txt") → "/foo/bar/"
(file-namestring #p"/foo/bar/baz.txt") → "baz.txt"
```

14.8 构造新路径名

你可以使用MAKE-PATHNAME函数来构造任意路径名。它对每个路径名组件都接受一个关键字参数并返回一个路径名，任何提供了的组件都被填入其中而其余的为NIL。^②

```
(make-pathname
  :directory '(:absolute "foo" "bar")
  :name "baz"
  :type "txt") → #p"/foo/bar/baz.txt"
```

不过，如果你希望程序是可移植的，你不会想要完全用手工生成路径名：就算路径名抽象可以保护你免于使用不可移植的文件名语法，文件名也可能以其他方式不可移植。例如，文件名/home/peter/foo.txt对于Mac OS X来说就不是一个好的文件名，因为在那裡/home/被称为/Users/。

不推荐完全用手工生成路径名的另一个原因是，不同的实现使用路径名组件的方式略有差异。例如，前面已经提到过，某些基于Windows的Lisp实现会将驱动器字母保存在设备组件中，而其他一些实现则将它保存在主机组件中。如果你将代码写成这样

```
(make-pathname :device "c" :directory '(:absolute "foo" "bar") :name "baz") .
```

那么它在一些实现里将是正确的而在其他实现里则不是。

与其用手工生成路径名，不如基于一个已有的路径名使用MAKE-PATHNAME的关键字参数:defaults来构造一个新路径名。你可以为该参数提供一个路径名描述符，它将提供没有被其他参数指定的任何组件的值。例如，下面的表达式创建了一个带有.html扩展名的路径名，同时所有其他组件都与变量input-file中的路径名相同：

```
(make-pathname :type "html" :defaults input-file)
```

^① 由FILE-NAMESTRING返回的名字在支持版本的文件系统上也含有版本组件。

^② 主机组件可能不是默认为NIL，但如果不是的话，它将是一个不透明的由实现定义的值。

假设`input-file`中的值是一个用户提供的名字，这一代码对于操作系统和实现的区别来说是健壮的，无论文件名是否带有驱动器字母或是它被保存在路径名的那个组件上。^①

你可以使用相同的技术来创建一个带有不同目录组件的路径名：

```
(make-pathname :directory '(:relative "backups") :defaults input-file)
```

不过，这会创建出一种整个目录组件是相对目录`backups/`的路径名，而不管`input-file`是否可能会有任何目录组件。例如：

```
(make-pathname :directory '(:relative "backups")
:defaults #p"/foo/bar/baz.txt") → #p"backups/baz.txt"
```

但有时你会想要通过合并两个路径名的目录组件来组合两个路径名，其中至少一个带有相对的目录组件。例如，假设有一个诸如`#p"foo/bar.html"`的相对路径名，你想将它与一个诸如`#p"/www/html/`这样的绝对路径名组合起来得到`#p"/www/html/foo/bar.html"`。在这种情况下，**MAKE-PATHNAME**将无法处理。相反，你需要的是**MERGE-PATHNAMES**。

MERGE-PATHNAMES接受两个路径名并合并它们，用来自第二个路径名的对应值填充第一个路径名中的任何**NIL**组件，这和**MAKE-PATHNAME**使用来自`:defaults`参数的组件来填充任何未指定的组件非常相似。不过，**MERGE-PATHNAMES**会特别对待目录组件：如果第一个组件名的目录是相对的，那么生成的路径名的目录组件将是第一个路径名的目录相对于第二个路径名的目录。这样：

```
(merge-pathnames #p"foo/bar.html" #p"/www/html/") → #p"/www/html/foo/bar.html"
```

第二个路径名也可以是相对的，在这种情况下得到的路径名也将是相对的：

```
(merge-pathnames #p"foo/bar.html" #p"html/") → #p"html/foo/bar.html"
```

为了反转这一过程以便获得一个相对于特定根目录的文件名，你可以使用函数**ENOUGH-NAMESTRING**，这很有用。

```
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/") → "html/foo/bar.html"
```

随后可以组合**ENOUGH-NAMESTRING**和**MERGE-PATHNAMES**来创建一个表达相同名字但却在不同根目录中的路径名。

```
(merge-pathnames
(enough-namestring #p"/www/html/foo/bar/baz.html" #p"/www/")
#p"/www-backups/") → #p"/www-backups/html/foo/bar/baz.html"
```

MERGE-PATHNAMES也被用来实际访问文件系统中标准函数内部用于填充不完全的路径名的文件。例如，假设有一个只有名称和类型的路径名：

① 对于完全最大化的可移植性，你真的应该写成这样：

```
(make-pathname :type "html" :version :newest :defaults input-file)
```

没有`:version`参数的话，在一个带有内置版本支持的文件系统上，输出的路径名将继承来自输入文件的版本号，这可能不是正确的行为——如果输入文件已经被保存了许多次，它将带有一个比生成的HTML文件大的版本号。在没有文件版本的实现上，`:version`参数会被忽略。如果你比较在意可移植性，最好加上它。

```
(make-pathname :name "foo" :type "txt") → #p"foo.txt"
```

如果想用这个路径名作为OPEN的一个参数，那么缺失的组件（诸如目录）就必须在Lisp可以将路径名转化成一个实际文件名之前被填充进去。Common Lisp将通过合并给定路径名与变量*DEFAULT-PATHNAME-DEFAULTS*中的值来获得缺失组件的值。该变量的初始值由具体实现决定，通常是一个路径名，其目录组件表示Lisp启动时所在的目录，如果需要，主机和设备组件也带有适当的值。如果只有一个参数被调用，MERGE-PATHNAMES将把该参数与*DEFAULT-PATHNAME-DEFAULTS*的值进行合并。例如，如果*DEFAULT-PATHNAME-DEFAULTS*是#p"/home/peter/"，那么将得到下面的结果：

```
(merge-pathnames #p"foo.txt") → #p"/home/peter/foo.txt"
```

14.9 目录名的两种表示方法

当处理命名目录的路径名时，你需要注意一点。路径名将目录和名称组件区分开，但Unix和Windows却将目录视为另一种类型的文件。这样，在这些系统里，每一个目录都有两种不同的路径名表示方法。

一种表示方法，我将它称为文件形式(file form)，将目录当成像其他任何文件一样来对待，将名字字符串中的最后一个元素放在名称和类型组件中。另一种表示方法，目录形式(directory form)将名字中的所有元素都放在目录组件中，而留下名称和类型组件为NIL。如果/foo/bar/是一个目录，那么下面两个路径名都可以命名它：

```
(make-pathname :directory '(:absolute "foo") :name "bar") ; file form
(make-pathname :directory '(:absolute "foo" "bar")) ; directory form
```

用MAKE-PATHNAME创建路径名时，你可以控制得到的形式，但需要在处理名字字符串时多加小心。当前所有实现都创建文件形式的路径名，除非名字字符串以一个路径分隔符结尾。但你不能依赖于用户提供的名字字符串必须是以一种或另一种形式。例如，假设提示用户输入一个保存文件的目录而他们输入了"/home/peter"。如果你将该值作为MAKE-PATHNAME的:defaults参数，像这样

```
(make-pathname :name "foo" :type "txt" :defaults user-supplied-name)
```

来传递，那么最后你将把文件保存成/home/foo.txt而不是你想要的/home/peter/foo.txt，因为当user-supplied-name被转化成一个路径名时，名字字符串中的"peter"将被放在名称组件中。在下一章我将讨论的路径名移植库中，你将编写一个称为pathname-as-directory的函数，它将一个路径名转化成目录形式。使用该函数，你可以放心地在用户给出的目录里保存文件。

```
(make-pathname
  :name "foo" :type "txt" :defaults (pathname-as-directory user-supplied-name))
```



14.10 与文件系统交互

通常，与文件系统的多数交互可能是用**OPEN**打开文件来读写。你偶尔也需要测试一个文件是否存在，列出一个目录的内容，删除和重命名文件，创建目录以及获取一个文件的信息，诸如谁拥有它、它在何时被最后修改以及它的长度。这就是由路径名抽象所带来的通用性开始造成痛苦的地方：因为语言标准并没有指定那些与文件系统交互的函数是如何映射到任何特定的文件系统上的，从而给实现者们留下了充分的余地。

这就是说，多数与文件系统进行交互的函数仍然是相当直接的。我将在这里讨论标准函数，并且指出其中哪些是在实现之间存在不可移植性的。在下一章里，你将开发一个路径名可移植库来消除那些不可移植因素中的一部分。

为了测试一个对应于某个路径名描述符（路径名、名字字符串或文件流）的文件是否存在于文件系统中，你可以使用函数**PROBE-FILE**。如果由路径名描述符命名的文件存在，那么**PROBE-FILE**将返回该文件的真实名称（*truename*），一个进行了诸如解析符号链接这类文件系统层面转换的路径名。否则它返回**NIL**。不过，并非所有实现都支持使用该函数来测试一个目录是否存在。同样，Common Lisp也不支持用一种可移植的方式来测试一个给定文件是否作为一个正规文件或目录而存在。在下一章里，你将把**PROBE-FILE**包装在一个新函数**file-exists-p**中，它不但可以测试目录是否存在，还可以告诉你一个给定的名字究竟是文件名还是目录名。

类似地，用于列出文件系统中文件的标准函数**DIRECTORY**对于简单的情形工作得很好，但实现之间的区别却使得它难以可移植地使用。在下一章里，你将定义一个**list-directory**函数来消除这些区别。

DELETE-FILE和**RENAME-FILE**的功能恰如其名。**DELETE-FILE**接受一个路径名描述符并删除所命名的文件，当其成功时返回真。否则它产生一个**FILE-ERROR**报错。^①

RENAME-FILE接受两个路径名描述符，并将第一个名字命名的文件重命名为第二个名字。

你可以使用函数**ENSURE-DIRECTORIES-EXIST**来创建目录。它接受一个路径名描述符并确保目录组件中的所有元素存在并且是目录，如果必要的话它会创建它们。它返回被传递的路径名，这使得它易于内联使用。

```
(with-open-file (out (ensure-directories-exist name) :direction :output)
  ...)
```

注意如果你传给**ENSURE-DIRECTORIES-EXIST**一个目录名，它应该是目录形式的，否则目录的最后一级子目录将不会被创建。

函数**FILE-WRITE-DATE**和**FILE-AUTHOR**都接受一个路径名描述符。**FILE-WRITE-DATE**返回文件上次被写入的时间，表示形式是自从格林尼治标准时间（GMT）1900年1月1日午夜起的

^① 更多关于错误处理的内容请参见第19章。

秒数，而`FILE-AUTHOR`在Unix和Windows上返回该文件的拥有者。^①

为了知道一个文件的长度，你可以使用函数`FILE-LENGTH`。出于历史原因，`FILE-LENGTH`接受一个流而不是一个路径名作为参数。在理论上，这允许`FILE-LENGTH`返回在该流的元素类型意义上的长度。尽管如此，由于在当今大多数操作系统上关于一个文件的长度唯一可以得到的信息（除了实际读取整个文件来测量它以外）只有以字节为单位的长度，这也是多数实现所返回的，甚至当`FILE-LENGTH`被传递了一个字符流时情况也是如此。不过，标准并没有强制要求这一行为，因此为了得到可预测的结果，获得一个文件长度的最佳方式是使用一个二进制流。^②

```
(with-open-file (in filename :element-type '(unsigned-byte 8))
  (file-length in))
```

一个同样接受打开的文件流作为参数的相关函数是`FILE-POSITION`。当只用一个流来调用它时，该函数返回文件中的当前位置，即已经被读取或写入该流的元素的数量。当以两个参数（流和位置描述符）来调用该函数时，它将该流的位置设置到所描述的位置上。这个位置描述符必须是关键字`:start`、`:end`或者非负的整数。两个关键字可以将流的位置设置到文件的开始或结尾处，而一个整数将使流的位置移动到文件中指定的位置上。对于二进制流来说，这个位置就是文件中的字节偏移量。尽管如此，因为字符编码因素的存在，对于字符流来说事情变得有一点儿复杂。当你需要在一个文本数据的文件中做跳转时，最可靠的方法就是只为两参数版本的`FILE-POSITION`的第二个参数传递一个由单参数版本`FILE-POSITION`同样的流参数下曾经返回的一个值。

14.11 其他 I/O 类型

除了文件流以外，Common Lisp还支持其他类型的流，它们也可被用于各种读、写和打印I/O函数。例如，你可以使用`STRING-STREAM`从一个字符串中读取或写入数据，你可以使用函数`MAKE-STRING-INPUT-STREAM`和`MAKE-STRING-OUTPUT-STREAM`来创建`STRING-STREAM`。

`MAKE-STRING-INPUT-STREAM`接受一个字符串以及可选的开始和结尾指示符来鉴定字符串中数据应被读取的区域，然后返回一个可被传递到任何诸如`READ-CHAR`、`READ-LINE`或`READ`这

① 对于需要访问特定操作系统或文件系统上其他文件属性的应用来说，第三方库提供了对底层C系统调用的绑定。

<http://common-lisp.net/project/osicat/>上的Osicat库提供了一个构建在Universal Foreign Function Interface (UFFI) 之上的简单API，该库应当可以运行在POSIX操作系统的多数Common Lisp上。

② 就算在你没有使用多字节字符编码时，一个文件中字节和字符的数量也可能是不同的。因为字符流也会将平台相关的行结束符转化成单一的#\Newline字符，在Windows上（其中使用CRLF作为行结束符），字符的数量通常小于字节的数量。如果你真想知道文件中字符的数量，你不得不亲自数一下并书写类似下面这样的代码：

```
(with-open-file (in filename)
  (loop while (read-char in nil) count t))
```

或者可能是如下更高效的代码：

```
(with-open-file (in filename)
  (let ((scratch (make-string 4096)))
    (loop for read = (read-sequence scratch in)
          while (plusp read) sum read)))
```

些基于字符的输入函数中的字符流。例如，如果你有一个含有Common Lisp语法的字面浮点数的字符串，那么你可以像下面这样将它转化成一个浮点数：

```
(let ((s (make-string-input-stream "1.23")))
  (unwind-protect (read s)
    (close s)))
```

类似地，**MAKE-STRING-OUTPUT-STREAM**创建一个流，其可被用于**FORMAT**、**PRINT**、**WRITE-CHAR**以及**WRITE-LINE**等。它不接受参数。无论你写了什么，字符串输出流都将被累积到字符串中，你随后可以通过函数**GET-OUTPUT-STREAM-STRING**来获取该字符串。每次当你调用**GET-OUTPUT-STREAM-STRING**时，该流的内部字符串会被清空，因此就可以重用一个已有的字符串输出流。

不过你将很少直接使用这些函数，因为宏**WITH-INPUT-FROM-STRING**和**WITH-OUTPUT-TO-STRING**提供了一个更加便利的接口。**WITH-INPUT-FROM-STRING**和**WITH-OPEN-FILE**相似，它从给定字符串中创建字符串输入流，并在该流绑定到你提供的变量的情况下执行它的主体形式。例如，与其使用**LET**形式并带有显式的**UNWIND-PROTECT**，你可以这样来写。

```
(with-input-from-string (s "1.23")
  (read s))
```

宏**WITH-OUTPUT-TO-STRING**与之相似：它把新创建的字符串输出流绑定到你所命名的变量上，然后执行它的主体。在所有主体形式都被执行以后，**WITH-OUTPUT-TO-STRING**返回由**GET-OUTPUT-STREAM-STRING**返回的值。

```
CL-USER> (with-output-to-string (out)
  (format out "hello, world ")
  (format out "~s" (list 1 2 3)))
"hello, world (1 2 3)"
```

语言标准中定义的其他流提供了多种形式的流拼接技术，它允许你以几乎任何配置将流拼接在一起。**BROADCAST-STREAM**是一个输出流，它将向其写入的任何数据发送到一组输出流上，这些流是它的构造函数**MAKE-BROADCAST-STREAM**的参数。^①与之相反的，**CONCATENATED-STREAM**是一个输入流，它从一组输入流中接收其输入，在每个流的结尾处它从一个流移动到另一个。你可以使用函数**MAKE-CONCATENATED-STREAM**来构造**CONCATENATED-STREAM**，其接受任何数量的输入流作为参数。

两种可以将流以多种方式拼接在一起的双向流是**TWO-WAY-STREAM**和**ECHO-STREAM**。它们的构造函数**MAKE-TWO-WAY-STREAM**和**MAKE-ECHO-STREAM**都接受两个参数，一个输入流和一个输出流，并返回一个适当类型的可同时用于输入和输出函数的流。

在**TWO-WAY-STREAM**中，你所做的每一次读取都会返回从底层输入流中读取的数据，而每次写入将把数据发送到底层的输出流上。除了所有从底层输入流中读取的数据也被回显到输出流之外，**ECHO-STREAM**本质上以相同的方式工作。这样，**ECHO-STREAM**中的输出流将含有会话双方的一个副本。

^① 通过不带参数地调用它，**MAKE-BROADCAST-STREAM**可以生成一个数据黑洞。

使用这五种流，你可以构造出几乎任何你想要的流拼接拓扑结构。

最后，尽管Common Lisp标准并没有涉及有关网络API的内容，但多数实现都支持socket编程并且通常将socket实现成另一种类型的流，因此你可以使用正规I/O函数来操作它们。^①

现在，你已准备好开始构建一个库来消除不同Common Lisp实现在基本路径名函数行为上的一些区别了。

① Common Lisp的标准I/O机制最缺失的是一种允许用户定义新的流类（stream class）的方式。不过，存在两种用户自定义流的事实标准。在Common Lisp标准化期间，德州仪器的David Gray编写了一份API草案，其中允许用户定义新的流类。然而不幸的是，当时没有时间解决由这份草案产生的所有问题，从而将其包含到语言标准中。尽管如此，许多实现都支持某种形式的所谓Gray Streams，它们的API都是基于Gray的草案。另一种更新的API称为Simple Streams，它由Franz开发并包括在Allegro Common Lisp中。它被设计用来改进用户自定义流相对于Gray Streams的性能，并且已经被某些开源Common Lisp实现所采用。

实践：可移植路径名库

如 同我在前面章节里讨论的那样，Common Lisp提供了一种称为路径名的抽象，这样一来，你就不用顾忌不同操作系统和文件系统在文件命名方式上的差异。路径名提供了一个有用的API来管理作为路径的名字，但是当它涉及实际与文件系统交互的函数时，事情就会变得有些复杂。

如同我提到的，问题的根源在于，路径名抽象被设计用来表示比当今常用的文件系统更加广泛的系统上的文件名。不幸的是，为了让路径名足够抽象从而可以应用于广泛的文件系统，Common Lisp的设计者们留给了实现者们大量的选择空间，来决定究竟如何将路径名抽象映射到任何特定文件系统上。这样带来的结果是，不同的实现者虽然在相同的文件系统上实现路径名抽象，但在一些关键点上却做出了不同的选择，从而导致遵循标准的实现在一些主要的路径名相关函数上不可避免地提供了不同的行为。

然而，所有的实现都以这样或那样的方式提供了相同的基本功能，因此你可以写一个库，对多数跨越不同实现的常见操作提供一致的接口。这就是你在本章中的任务。编写这个库，你不但可以获得后续几章将会用到的几个有用的函数，还可以有机会学习如何编写处理不同Lisp实现间区别的代码。

15.1 API

该库支持的基本操作是获取目录中的文件列表，并检测给定名字的文件或目录是否存在。你也将编写函数用于递归遍历目录层次，并在目录树的每个路径名上调用给定的函数。

从理论上来讲，这些列目录和测试文件存在性的操作已经由标准函数 DIRECTORY 和 PROBE-FILE 提供了。不过正如你将看到的那样，会有许多不同的方式来实现这些函数——所有这些均属于语言标准的有效解释，因此你希望编写新的函数，以在不同实现间提供一致的行为。

15.2 *FEATURES* 和读取期条件化

在能够实现这个可在多个Common Lisp实现上正确运行的库的API之前，我需要首先介绍编写实现相关代码的手法。

“在任何符合标准的Common Lisp实现上正确运行的代码都将产生相同的行为”，从这个意义



上说，尽管你所编写的多数代码都是“可移植的”，但你可能偶尔需要依赖于实现相关的功能，或者为不同实现编写稍有差别的代码。为了使你在不完全破坏代码可移植性的情况下做到这点，Common Lisp提供了一种称为读取期条件化的机制，从而允许你有条件地包含基于当前运行的实现等各种特性的代码。

该机制由一个变量`*FEATURES*`和两个被Lisp读取器理解的附加语法构成。`*FEATURES*`是一个符号的列表，每个符号代表存在于当前实现或底层平台的一个“特性”。这些符号随后用在特性表达式中，根据表达式中的符号是否存在于`*FEATURES*`求值为真或假。最简单的特性表达式是单个符号，当符号在`*FEATURES*`中时该表达式为真，否则为假。其他的特性表达式是构造在`NOT`、`AND`和`OR`操作符上的布尔表达式。例如，如果要条件化某些代码使其只有当特性`foo`和`bar`存在时才被包含，那么可以将特性表达式写成`(and foo bar)`。

读取器将特性表达式与两个语法标记`#+`和`#-`配合使用。当读取器看到任何一个这样的语法时，它首先读取特性表达式并按照我刚刚描述的方式求值。当跟在`#+`之后的特性表达式为真时，读取器会正常读取下一个表达式；否则它会跳过下一个表达式，将它作为空白对待。`#-`以相同的方式工作，只是它在特性表达式为假时才读取后面的形式，而在特性表达式为真时跳过它。

`*FEATURES*`的初始值是实现相关的，并且任何给定符号的存在所代表的功能也同样是由实现定义的。尽管如此，所有的实现都包含至少一个符号来指示当前是什么实现。例如，Allegro Common Lisp含有符号:`:allegro`，CLISP含有`:clisp`，SBCL含有`:sbcl`，而CMUCL含有`:cmu`。为了避免依赖于在不同实践中可能不存在的包，`*FEATURES*`中的符号通常是关键字，并且读取器在读取特性表达式时将`*PACKAGE*`绑定到`KEYWORD`包上。这样，不带包限定符的名字将被读取成关键字符。因此，你可以像下面这样编写一个在前面提到的每个实现中行为稍有不同的函数：

```
(defun foo ()
  #+allegro (do-one-thing)
  #+sbcl (do-another-thing)
  #+clisp (something-else)
  #+cmu (yet-another-version)
  #- (or allegro sbcl clisp cmu) (error "Not implemented"))
```

在Allegro中读取上述代码，就好像代码原本就写成下面这样：

```
(defun foo ()
  (do-one-thing))
```

在SBCL中读取器将读到下面的内容：

```
(defun foo ()
  (do-another-thing))
```

而在一个不属于上述特定条件化实现的平台上，它将被读取成下面这样：

```
(defun foo ()
  (error "Not implemented"))
```

因为条件化过程发生在读取器中，编译器根本无法看到被跳过的表达式，^①这意味着你不会

^① 这种读取器条件化工作方式所带来的一个稍为麻烦的后果是，无法简单地编写fall-through case。例如，如果通过在`foo`中增加另一个`#+`前缀的表达式来为其添加对另一种实现的支持，那么你需要记得也要在`#-`之后的`or`特性表达式中添加同样的特性，否则`ERROR`形式将会在新代码运行以后被求值。

为不同实现的不同版本付出任何运行时代价。另外，当读取器跳过条件化的表达式时，它不会保留其中的符号，因此被跳过的表达式可以安全地包含在其他实现中可能不存在的包中的符号。

对库打包

从包的角度讲，如果你下载了该库的完整代码，会看到它被定义在一个新的包`com.gigamonkeys.pathname`s中。我将在第21章讨论定义以及使用包的细节。目前你应当注意，某些实现提供了它们自己的包，其中含有一些函数与你将在本章中定义的一些函数有相同的名字，并且这些名字可在CL-USER包中访问。这样，如果你试图在CL-USER包中定义该库中的某些函数，可能会得到关于破坏了已有定义的错误或警告。为了避免发生这种情况，你可以创建一个称为`packages.lisp`的文件，其中带有下面的内容：

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.pathname
  (:use :common-lisp)
  (:export
   :list-directory
   :file-exists-p
   :directory-pathname-p
   :file-pathname-p
   :pathname-as-directory
   :pathname-as-file
   :walk-directory
   :directory-p
   :file-p))
```

接着加载它，然后在REPL中或者在你输入定义的文件顶端，输入下列表达式：

```
(in-package :com.gigamonkeys.pathname)
```

将库以这种方式打包，除了可以避免与那些已存在于CL-USER包中的符号产生冲突以外，还可以使其更容易被其他代码使用，你在后续几章中将看到这一点。

15.3 列目录

你可以将用于列举单独目录的函数`list-directory`，实现成标准函数`DIRECTORY`外围的一个包装层。`DIRECTORY`接受一种特殊类型的路径名，称为通配路径名，其带有一个或多个含有特殊值`:wild`的组件，然后返回一个路径名的列表，用来表示文件系统中匹配该通配路径名的文件。^①匹配算法和多数在Lisp与一个特定文件系统之间的交互一样，它没有被语言标准定义，但Unix与Windows上的多数实现遵循了相同的基本模式。

^① 另一个特殊值`:wild-inferiors`可以作为一个通配路径名的目录组件的一部分出现，但在本章里你不需要它们。

DIRECTORY函数有两个需要在list-directory中解决的问题。其中主要的一个是，即便在相同的操作系统上，其行为的特定方面在不同的Common Lisp的实现间也具有相当大的区别。另一个问题在于，尽管**DIRECTORY**提供了一个强大的用于列举文件的接口，但要想正确地使用它需要对路径名抽象有相当细致的理解。在这些不同细微之处和不同实现的特征影响下，实际编写可移植代码来使用**DIRECTORY**完成一些像列出单个目录中所有文件和子目录这样简单的事情，都可能令人沮丧。你可以通过编写list-directory来一次性地处理所有这些细节和特征，并从此忘记它们。

我在第14章里讨论过的一个细节是，将一个目录的名字表示成路径名有两种方式，即目录形式和文件形式。

为了让**DIRECTORY**返回`/home/peter/`中的文件列表，需要传给它一个通配路径名，其目录组件是你想要列出的目录，其名称和类型组件是`:wild`。因此，为了列出`/home/peter/`中的文件，需要这样来写：

```
(directory (make-pathname :name :wild :type :wild :defaults home-dir))
```

其中的`home-dir`是代表`/home/peter/`的路径名。如果`home-dir`是以目录形式表示的，那么上述写法是有效的。但如果它以文件形式表示，例如，它通过解析名字字符串`"/home /peter"`来创建，那么同样的表达式将列出`/home`中的所有文件，因为名字组件`"peter"`将被替换成`:wild`。

为了避免两种形式间的显式转换，可以定义list-directory来接受任何形式的非通配路径名，随后它将被转化成适当的通配路径名。

为此，应当定义一些助手函数。其中一个是component-present-p，它将测试一个路径名的给定组件是否“存在”，也就是说该组件既不是NIL也不是特殊值`:unspecified`。^①另一个函数是directory-pathname-p，用来测试一个路径名是否已经是目录形式，而第三个函数pathname-as-directory可以将任何路径名转换成目录形式的路径名。

```
(defun component-present-p (value)
  (and value (not (eql value :unspecified))))  
  

(defun directory-pathname-p (p)
  (and
    (not (component-present-p (pathname-name p)))
    (not (component-present-p (pathname-type p)))
    p))  
  

(defun pathname-as-directory (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (not (directory-pathname-p name))
        (make-pathname
```

^① 具体实现返回`:unspecified`来代替NIL，作为某些特定情况下路径名组件的值，例如当该组件没有被该实现使用时。

```
:directory (append (or (pathname-directory pathname) (list :relative))
                    (list (file-namestring pathname)))
  :name      nil
  :type      nil
  :defaults pathname)
  pathname)))
```

现在看起来似乎可以通过在由 pathname-as-directory 返回的目录形式名字上调用 **MAKE-PATHNAME**, 来生成一个通配路径名并传给 **DIRECTORY**。不幸的是, 事情没有那么简单, 多亏了CLISP的 **DIRECTORY**实现中的一个怪癖。在CLISP中, 除非通配符中的类型组件是 **NIL**而非 :wild, **DIRECTORY** 将不会返回那些没有扩展名的文件。因此, 你可以定义一个函数 **directory-wildcard**, 其接受一个目录形式或文件形式的路径名, 并返回一个给定实现下的适当的通配符, 它通过使用读取期条件化在除CLISP之外的所有实现里生成一个带有 :wild 类型组件的路径名, 而在CLISP中该类型组件为 **NIL**。

```
(defun directory-wildcard (dirname)
  (make-pathname
    :name :wild
    :type #-clisp :wild #+clisp nil
    :defaults (pathname-as-directory dirname)))
```

注意每一个读取期条件是怎样在单个表达式层面上操作的。在 #-clisp 之后, 表达式 :wild 要么被读取要么被跳过; 同样, 在 #+clisp 之后, NIL 要么被读取要么被跳过。

现在你可以首次看到 **list-directory** 函数了:

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (directory (directory-wildcard dirname)))
```

使用上述定义, 该函数在SBCL、CMUCL和LispWorks中正常工作。不幸的是, 你还需要谨慎处理另外一些实现间的区别。其中一点是, 并非所有实现都返回给定目录中的子目录。Allegro、SBCL、CMUCL和LispWorks可以返回子目录。OpenMCL默认不会这样做, 但如果为 **DIRECTORY** 传递一个实现相关的、值为真的关键字参数 :directories, 那么它将返回子目录。对于CLISP的 **DIRECTORY**, 只有当传递给它一个以 :wild 作为目录组件的最后一个元素且名字和类型组件为 **NIL** 的通配路径名时, 才可以返回子目录。而且, 在这种情况下, 它只返回子目录, 因此需要使用不同的通配符, 调用 **DIRECTORY** 两次并组合结果。

一旦所有实现都返回目录了, 你会发现它们返回的目录名有些是目录形式的, 有些是文件形式的。你想要 **list-directory** 总是返回目录形式的目录名, 以便可以只通过名字来区分子目录和正规文件。除Allegro之外, 所有实现都支持做到这点。Allegro要求为 **DIRECTORY** 传递一个实现相关的、值为 **NIL** 的关键字参数 :directories-are-files, 从而使其以目录形式返回目录。

一旦你知道如何使每一个实现做到你想要的事, 那么实际编写 **list-directory** 就只是简单地将不同版本用读取期条件组合起来了。

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
```

```

(error "Can only list concrete directory names."))
(let ((wildcard (directory-wildcard dirname)))

#+(or sbcl cmu lispworks)
(directory wildcard)

#+openmcl
(directory wildcard :directories t)

#+allegro
(directory wildcard :directories-are-files nil)

#+clisp
(nconc
 (directory wildcard)
 (directory (clisp-subdirectories-wildcard wildcard)))

#-(or sbcl cmu lispworks openmcl allegro clisp)
(error "list-directory not implemented"))

```

函数`clisp-subdirectories-wildcard`事实上并非是CLISP相关的，由于任何其他实现不需要它，因而可以将其定义放在一个读取期条件之后。在这种情况下，由于跟在`#+`后面的表达式是整个`DEFUN`，因此整个函数定义是否被包含将取决于`clisp`是否存在`*FEATURES*`中。

```

#+clisp
(defun clisp-subdirectories-wildcard (wildcard)
  (make-pathname
   :directory (append (pathname-directory wildcard) (list :wild))
   :name nil
   :type nil
   :defaults wildcard))

```

15.4 测试文件的存在

为了替换`PROBE-FILE`，你可以定义一个称为`file-exists-p`的函数。它应当接受一个路径名，并在其代表的文件存在时返回一个等价的路径名，否则返回`NIL`。它应当可以接受目录形式或文件形式的目录名，但如果该文件存在并且是一个目录，那么它应当总是返回目录形式的路径名。这将允许你使用`file-exists-p`和`directory-pathname-p`来测试任意一个名字是文件名还是目录名。

从理论上讲，`file-exists-p`和标准函数`PROBE-FILE`非常相似。确实，在一些实现，即SBCL、LispWorks和OpenMCL里，`PROBE-FILE`已经提供了`file-exists-p`的行为，但并非所有实现的`PROBE-FILE`都具有相同的行为。

Allegro和CMUCL的`PROBE-FILE`函数接近于你想要的行为——接受任何形式的目录名但不会返回目录形式的路径名，而只是简单地返回传给它的参数。幸运的是，如果以目录形式传递给它一个非目录的名字，它会返回`NIL`。因此对于这些实现，为了得到想要的行为，你可以首先以目录形式将名字传给`PROBE-FILE`。如果文件存在并且是一个目录，它将返回目录形式的名字。

如果该调用返回NIL，那么你可以用文件形式的名字再试一次。

另一方面，CLISP再一次有其自己的做事方式。它的PROBE-FILE将在传递目录形式的名字时立即报错，无论该名字所代表的文件或目录是否存在。它也会在以文件形式传递一个名字且该名字实际上是一个目录的名字时报错。为了测试一个目录是否存在，CLISP提供了它自己的函数probe-directory（在ext包中）。这几乎就是PROBE-FILE的镜像：它将在传递文件形式的名字，或者目录形式的名字而刚好该名字是一个文件时报错。唯一的区别在于，当命名的目录存在时，它返回T而不是路径名。

就算在CLISP中，你也可以通过将对PROBE-FILE和probe-directory的调用包装在IGNORE-ERRORS中来实现想要的语义。^①

```
(defun file-exists-p (pathname)
  #+(or sbcl lispworks openmcl)
  (probe-file pathname)

  #+(or allegro cmu)
  (or (probe-file (pathname-as-directory pathname))
      (probe-file pathname))

  #+clisp
  (or (ignore-errors
        (probe-file (pathname-as-file pathname)))
      (ignore-errors
        (let ((directory-form (pathname-as-directory pathname)))
          (when (ext:probe-directory directory-form)
            directory-form)))

  #- (or sbcl cmu lispworks openmcl allegro clisp)
  (error "list-directory not implemented")))
```

CLISP版本的file-exists-p用到的函数pathname-as-file，是前面定义的pathname-as-directory的逆函数，它返回等价于其参数的文件形式的路径名。尽管该函数只有CLISP用到，但它通常很有用，因此我们为所有实现定义它并使其成为该库的一部分。

```
(defun pathname-as-file (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (directory-pathname-p name)
        (let* ((directory (pathname-directory pathname))
               (name-and-type (pathname (first (last directory))))))
          (make-pathname
            :directory (butlast directory)
            :name (pathname-name name-and-type)
            :type (pathname-type name-and-type)))
```

^① 这个方法稍微有一点问题，例如，PROBE-FILE可能因为其他原因报错，这时代码将错误地解释它。不幸的是，CLISP文档并未指定PROBE-FILE和probe-directory可能报错的类型，并且从经验来看，在多数出错情况下它们将会报出simple-file-error。

```
:defaults pathname))
pathname)))
```

15.5 遍历目录树

最后，为了完成这个库，你可以实现一个称为walk-directory的函数。与前面定义的那些函数不同，这个函数不需要做任何事情来消除实现间的区别，它只需要用到你已经定义的那些函数。尽管如此，该函数很有用，你将在后续几章里多次用到它。它接受一个目录的名字和一个函数，并在该目录下所有文件的路径名上递归地调用该函数。它还接受两个关键字参数：`:directories` 和 `:test`。当`:directories`为真时，它将在所有目录的路径名和正规文件上调用该函数。如果有`:test`参数，它指定另一个函数，在调用主函数之前在每一个路径名上调用该函数，主函数只有当测试参数返回真时才会被调用。`:test`参数的默认值是一个总是返回真的函数，它是通过调用标准函数CONSTANTLY而生成的。

```
(defun walk-directory (dirname fn &key directories (test (constantly t)))
  (labels
    ((walk (name)
       (cond
         ((directory-pathname-p name)
          (when (and directories (funcall test name))
            (funcall fn name))
          (dolist (x (list-directory name)) (walk x)))
          ((funcall test name) (funcall fn name))))))
    (walk (pathname-as-directory dirname))))
```

现在你有了一个用于处理路径名的有用的函数库。正如我提到的那样，这些函数在后面的章节里将会很有用，尤其是第23章和第27章，在那里你将使用walk-directory在含有垃圾信息和MP3文件的目录树中将会艰难前行，但在我们到达那里之前，我还需要在接下来的两章中谈论一下面向对象。

重新审视面向对象： 广义函数

16

Lisp的发明比面向对象编程的兴起早了几十年^①，新的Lisp程序员们有时会惊奇地发现，原来Common Lisp竟是一门非常彻底的面向对象语言。Common Lisp之前的几个Lisp语言开发于面向对象还是一个崭新思想的年代，而那时有许多实验在探索将面向对象的思想（尤其是Smalltalk中所展现的）合并到Lisp中的方式。作为Common Lisp标准化过程的一部分，这些实验中的一些被合成在一起，以Common Lisp Object System（即CLOS）的名义出现。ANSI标准将CLOS合并到了语言之中，因此单独提及CLOS就不再有任何实际意义了。

CLOS为Common Lisp贡献的特性既有那些必不可少的，也有那些相对难懂的Lisp的“语言作为语言的构造工具”这一哲学的具体表现。本书无法对所有这些特性全部加以介绍，但在本章和下一章里，我将描述其中最常用的特性，并给出关于Common Lisp对象的概述。

你应当从一开始就注意到，Common Lisp的对象系统体现了与许多其他语言中相当不同的面向对象的原则。如果你能够深刻理解面向对象背后的基本思想，那么将会感谢Common Lisp在实现这些思想时所采用的强大和通用的方式。另一方面，如果你的面向对象经历很大程度上来自单一语言，那么你可能会发现Common Lisp的观点多少有些另类。你应当试图避免假设只存在一种方式令一门语言支持面向对象。^②如果你几乎没有面向对象编程经验，那么也应当不难理解这里

① 现在，Simula通常被认为是第一个面向对象的语言，其发明于20世纪60年代早期，只比McCarthy的第一个Lisp晚了几年。尽管如此，直到20世纪80年代Smalltalk的第一个广泛使用的版本发布以后，面向对象才真正起飞，几年以后C++才得以发布。Smalltalk从Lisp那里获得了许多灵感，并将它与来自Simula的思想组合起来，产生出一种动态的面向对象语言，C++则组合了Simula和C——另一种相当静态的语言，从而得到了一个静态的面向对象语言。这种早期的分道扬镳导致了许多关于面向对象的定义的困惑。来自C++阵营的人们倾向于认为C++的特定方面，例如严格的数据封装是面向对象的关键特征。不过来自Smalltalk阵营的人们则认为C++的许多特性只是C++的特性而已，并不属于面向对象的核心内容。事实上，据说Smalltalk的发明者Alan Kay就曾说过：“我发明了术语面向对象（object oriented），而我可以告诉你C++并不是我头脑里所想的东西。”

② 有些人反对将Common Lisp作为面向对象语言。特别是那些将严格数据封装视为面向对象关键特征的人们，通常是诸如C++、Eiffel或Java这类相对静态语言的拥护者，他们不认为Common Lisp是真正面向对象的。当然，如果按照那样的定义，就算是Smalltalk这种无可争议的最早的和最纯粹的面向对象语言也不再是面向对象的了。另外，那些将消息传递视为面向对象关键特征的人们也不会很高兴，因为Common Lisp在声称自己是面向对象的同时，其面向广义函数的设计提供了纯消息传递所无法提供的自由度。

的解释，不过文中偶尔比较其他语言做同样事情的方式的内容，你就只好跳过不看了。

16.1 广义函数和类

面向对象的基本思想在于一种组织程序的强大方式：定义数据类型并将操作关联在那些数据类型上。特别是，你希望产生一种操作并让其确切行为取决于该操作所涉及的一个或多个对象的类型。所有关于面向对象的介绍中使用的经典例子，是可应用于代表各种几何图形的对象的draw操作。draw操作的不同实现可用于绘制圆、三角形和矩形，而对draw的调用将实际绘制出圆、三角形或矩形，具体取决于draw操作所应用到的对象类型。draw的不同实现被分别定义，并且新的版本可以被定义来绘制其他图形，而无需修改调用方或是任何其他draw实现的代码。这一面向对象风格称为“多义性”，源自希腊语polymorphism，意思是“多种形式”，因为单一的概念性操作，诸如绘制一个对象，可以带有许多不同的具体形式。

Common Lisp和今天的多数面向对象语言一样都是基于类的，所有对象都是某个特定类的实例。^①一个对象的类决定了它的表示，诸如NUMBER和STRING这样的内置类带有不透明的表示，只能通过管理这些类型的标准函数来访问；而用户自定义类的实例，如同你将在下一章里看到的，由称为槽（slot）的命名部分组成。

类通过层次结构组织在一起，形成了所有对象的分类系统。一个类可以定义成另一个类的子类（subclass），后者称为它的基类（superclass）。一个类从它的基类中继承（inherit）其定义的一部分，而一个类的实例也被认为是其基类的实例。在Common Lisp中，类的层次关系带有一个单根，即类T，它是其他类的所有直接或间接基类。这样，Common Lisp中的每一个数据都是T的一个实例。^②Common Lisp也支持多继承（multiple inheritance），即单一的类可以拥有多个直接基类。

在Lisp家族之外，几乎所有的面向对象语言都遵循了由Simula建立的基本模式：类所关联的行为由属于一个特定类的方法（method）或成员函数（member function）定义。在这些语言里，在一个特定对象上调用一个方法，然后该对象所属的类决定运行什么代码。这种方法调用的模型称为消息传递（message passing），这是来自Smalltalk的术语。从概念上来讲，在一个消息传递系统中，方法调用开始于向被调用方法所操作的对象发送一个消息，其中含有需要运行的方法名和任何参数。该对象随后使用其类来查找与该消息中的名字所关联的方法并运行它。由于每个类对于一个给定名字都有它自己的方法，因此发送相同的消息到不同的对象可以调用不同的方法。

早期的Lisp对象系统以类似的方式工作，提供了一个特殊函数SEND，用于向特定对象发送消息。尽管如此，这种方式并不完全令人满意，因为它使得方法调用不同于正常的函数调用。句法意义上的方法调用应写成

```
(send object 'foo)
```

① 基于原型的语言是另一种面向对象的语言类型。在这些语言里，JavaScript可能是最流行的例子，它的对象通过克隆一个原型对象来创建。该克隆可以随后被修改并用作其他对象的原型。

② 作为常量的T和作为类的T，除了刚好具有相同的名字以外没有特别的关系。作为值的T是类SYMBOL的一个直接实例，并且只是间接地成为作为类的T的一个实例。

而不是

```
(foo object)
```

更重要的是，由于方法不是函数，它们无法作为参数传递给像MAPCAR这样的高阶函数。如果一个人想要使用MAPCAR在一个列表的所有元素上调用方法，他不得不写成

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

而不是

```
(mapcar #'foo objects)
```

最终，工作在Lisp对象系统上的人们通过创建一种新的称为广义函数 (generic function) 的函数类型而将方法和函数统一在一起。广义函数不但解决了上面描述的问题，它还为对象系统开放了新的可能性，包括许多在消息传递对象系统中基本无法实现的特性。

广义函数是Common Lisp对象系统的核心，也是本章其余部分的主题。虽然我不可能在不提到类的情况下谈论广义函数，但目前我将把注意力集中在如何定义和使用广义函数上。在下一章里，我将向你展示如何定义你自己的类。

16.2 广义函数和方法

广义函数定义了抽象操作，指定了其名字和一个参数列表，但不提供实现。例如，下面就是你可能定义广义函数draw的方式，它将用来在屏幕上绘制不同的形状：

```
(defgeneric draw (shape)
  (:documentation "Draw the given shape on the screen."))

```

我将在下一节里讨论`DEFGENERIC`的语法，目前只需注意该定义并不含有任何实际代码。

广义函数的广义性至少在理论上体现在，它可以接受任何对象作为参数。^①不过，广义函数本身并不能做任何事。如果你只是定义广义函数，那么无论用什么参数来调用它，它都将会报错。广义函数的实际实现是由方法 (method) 提供的。每一个方法提供了广义函数用于特定参数类的实现。也许在一个基于广义函数的系统和一个消息传递系统之间最大的区别在于方法并不属于类，它们属于广义函数，其负责在一个特定调用中检测哪个或哪些方法将被运行。

方法通过特化那些由广义函数所定义的必要参数，来表达它们可以处理的参数类型。例如，在广义函数draw中，你可以定义一个方法来特化shape参数，使其用于circle类的实例对象；而另一个方法则将shape特化成triangle类的实例对象。去掉实际的绘图代码以后，它们如下所示：

```
(defmethod draw ((shape circle))
  ...)
(defmethod draw ((shape triangle))
  ...)
```

^① 这里和其他地方一样，对象意味着任何Lisp数据——Common Lisp并不像一些语言里那样区分对象和“基本”数据类型。Common Lisp中的所有数据都是对象，并且任何对象都是某个类的实例。

当一个广义函数被调用时，它将那些被传递的实际参数与它的每个方法的特化符进行比较，找出可应用 (applicable) 的方法，即那些特化符与实际参数相兼容的方法。如果你调用 `draw` 并传递一个 `circle` 的实例，那么在 `circle` 类上特化了 `shape` 的方法将是可应用的；而如果你传递了一个 `triangle` 实例，那么在 `triangle` 上特化了 `shape` 的方法将被应用。在简单的情况下，只有一个方法是可应用的，并且它将处理该调用。在复杂的情况下，可能有多个方法均可应用，它们随后将被组合起来——我将在 16.5 节里进行讨论，成为一个有效的 (effective) 方法来处理该调用。

你可以用两种方式来特化参数。通常你将指定一个类，其参数必须是该类的实例。由于一个类的实例也被视为该类的所有基类的实例，因此一个带有特化了某个特定类的参数的方法可以被应用在对应参数无论是该特定类的直接实例或是该类的任何子类的实例上。另一种类型的特化符是所谓的 `EQL` 特化符，其指定了方法所应用的特定对象。

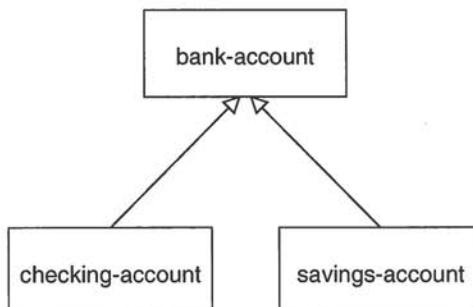
当一个广义函数只具有特化在单一参数上的方法并且所有特化符都是类特化符时，调用广义参数的结果跟在一个消息传递系统下调用方法的结果非常相似——操作的名字与调用时对象的类的组合决定了哪个方法被运行。

尽管如此，相反的方法查找顺序带来了消息传递系统所没有的可能性。广义函数支持特化在多个参数上的方法，提供了一个使多继承更具有可管理性的框架，并且允许你使用声明性的构造来控制方法如何组合成有效方法，从而在无须使用大量模板代码的情况下直接支持几种常用的设计模式。我将很快讨论到这些主题。但首先你需要了解两个用来定义广义函数的宏 `DEFGENERIC` 和 `DEFMETHOD` 的一些基础。

16.3 DEFGENERIC

为了给你一个关于这些宏和它们所支持的不同功能的大致印象，我将向你展示一些可能作为一个银行应用，或者说，一个相当幼稚的银行应用的一部分来编写的代码。重点在于观察一些语言特性而不是学习如何实际编写银行软件。例如，这些代码甚至并不打算处理像多种货币、审查跟踪以及事务集成这样的问题。

由于我不准备在下一章之前讨论如何定义新的类，因此目前你可以假设特定的类已经存在了。假设你已有一个 `bank-account` 类和它的两个子类 `checking-account` 以及 `savings-account`。类层次关系如下所示：



第一个广义函数将是withdraw，它将账户余额减少指定数量。如果余额小于提款量，它将报错并保持余额不变。你可以从通过`DEFGENERIC`定义该广义函数开始。

除了缺少函数体之外，`DEFGENERIC`的基本形式与`DEFUN`相似。`DEFGENERIC`的形参列表指定了那些定义在该广义函数上的所有方法都必须接受的参数。在函数体的位置上，`DEFGENERIC`可能含有不同的选项。一个你应当总是带有的选项是`:documentation`，它提供了一个用来描述该广义函数用途的字符串。由于广义函数是纯抽象的，让用户和实现者了解它的用途将是重要的。因此，你可以像下面这样定义withdraw：

```
(defgeneric withdraw (account amount)
  (:documentation "Withdraw the specified amount from the account.
Signal an error if the current balance is less than amount."))

```

16.4 DEFMETHOD

现在你开始使用`DEFMETHOD`来定义实现了withdraw的方法。^①

方法的形参列表必须与它的广义函数保持一致。在本例中，这意味着所有定义在withdraw上的方法都必须刚好有两个必要参数。在更一般的情况下，方法必须带有由广义函数指定的相同数量的必要和可选参数，并且必须可以接受对应于任何`&rest`或`&key`形参的参数。^②

由于提款的基本操作对于所有账户都是相同的，因此你可以定义一个方法，其在`bank-account`类上特化了`account`参数。你可以假设函数`balance`返回当前账户的余额并且可被用于同`SETF`（因此也包括`DECF`）一起来设置余额。函数`ERROR`是一个用于报错的标准函数，我将在第19章里讨论其进一步的细节。使用这两个函数，你可以像下面这样定义出一个基本的withdraw方法：

```
(defmethod withdraw ((account bank-account) amount)
  (when (< (balance account) amount)
    (error "Account overdrawn."))
  (decf (balance account)))

```

如同这段代码显示的，`DEFMETHOD`的形式比`DEFGENERIC`更像是一个`DEFUN`形式。唯一的区别在于必要形参可以通过将形参名替换成两元素列表来进行特化。其中第一个元素是形参名，而第二个元素是特化符，其要么是类的名字要么是`EQL`特化符，其形式我将很快讨论到。形参名可

^① 从技术上来讲，你可以完全跳过`DEFGENERIC`。如果你用`DEFMETHOD`定义了一个方法而没有定义相关的广义函数，那么广义函数将被自动创建。但是显式地定义广义函数是好的形式，因为它给你一个好的位置来文档化你想要的行为。

^② 一个方法“接受”由其广义函数定义的`&key`和`&rest`参数的方式可以是使用`&rest`形参，使用相同的`&key`形参或是将`&allow-other-keys`与`&key`一起指定。方法也可以指定广义函数的形参列表中所没有的`&key`形参，当广义函数被调用时，任何由广义函数指定的`&key`参数或任何可应用的方法将被接受。

这种一致性规则带来的一个后果是，同一个广义函数上的所有方法将同样带有一致的形参列表。Common Lisp不支持诸如C++和Java那样的某些静态类型语言里支持的方法重载（method overloading），在那里相同的名字可被用于带有不同形参列表的方法。

以是任何东西——它不需要匹配广义函数中使用的名字，尽管经常是使用相同的名字。

该方法在每当withdraw的第一个参数是bank-account的实例时被应用。第二个形参amount被隐式特化到T上，而由于所有对象都是T的实例，它不会影响该方法的可应用性。

现在假设所有现金账户都带有透支保护。这就是说，每个现金账户都与另一个银行账户相关联，该账户将在现金账户的余额本身无法满足提款需求时被提款。你可以假设函数overdraft-account接受checking-account对象并返回代表了关联账户的bank-account对象。

这样，相比从标准的bank-account对象中提款，从checking-account对象中提款需要一些额外的步骤。你必须首先检查提款金额是否大于该账户的当前余额，如果大于，就将差额转给透支账户。然后你可以像处理标准的bank-account对象那样进行处理。

因此，你要做的是在withdraw上定义一个特化在checking-account上的方法来处理该传递过程，然后再让特化在bank-account上的方法接手。这样一个方法如下所示：

```
(defmethod withdraw ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft)))
    (call-next-method)))
```

函数**CALL-NEXT-METHOD**是广义函数机制的一部分，用于组合可应用的方法。它指示控制应当从该方法传递到特定于bank-account的方法上。^①当它不带参数被调用时，就像这里的这样，下一个方法将以最初传递给广义函数的参数被调用。它也可以带参数被调用，这些参数随后被传递给下一个方法。

你不必在每一个方法中调用**CALL-NEXT-METHOD**。尽管如此，如果你不这样做的话，新的方法将负责完全实现你想要的广义函数行为。假如你有一个bank-account的子类proxy-account，它并不实际跟踪自己的余额而是将提款请求代理到其他账户，那么你可以写一个如下的方法（假设有一个函数proxied-account的方法可以返回代理的账户）：

```
(defmethod withdraw ((proxy proxy-account) amount)
  (withdraw (proxied-account proxy) amount))
```

最后，**DEFMETHOD**还允许你通过使用**EQL**特化符来创建特化在一个特定对象上的方法。例如，假设该银行应用被部署在某个腐败银行上。假设变量*account-of-bank-president*保存了一个特定银行账户的引用，如同其名字显示的，该账户属于该银行的总裁。进一步假设变量*bank*代表该银行整体，而函数embezzle可以从银行中偷钱。银行总裁可能会让你“修复”withdraw来特别处理他的账户。

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
```

^① **CALL-NEXT-METHOD**大致相当于Java中在super上调用一个方法，或是在Python或C++中使用一个显式的类限定方法或函数名。

```
(when (plusp overdraft)
  (incf (balance account) (embezzle *bank* overdraft)))
  (call-next-method))
```

不过需要注意的是，**EQL**特化符中提供了特化对象的形式，在本例中是变量`*account-of-bank-president*`，其只在**DEFMETHOD**被求值时求值一次。在定义方法时该方法将特化`*account-of-bank-president*`的值。随后，改变该变量将不会改变该方法。

16.5 方法组合

在一个方法体之外，**CALL-NEXT-METHOD**没有任何意义。在一个方法之内，它被广义函数机制定义，用来在每次广义函数使用的所有应用于特定调用的方法被调用时构造一个有效方法。这种通过组合可应用的方法来构造有效方法的概念是广义函数概念的核心，并且是让广义函数可以支持消息传递系统里所没有的机制的关键。因此，值得更进一步地观察究竟发生了什么。那些在他们的意识中带有根深蒂固的消息传递模型思想的人们应当尤其注意这点，因为广义函数相比消息传递完全颠覆了方法的调度过程，使得广义函数而不是类成为了主要推动者。

从概念上讲，有效方法由三步构造而成：首先，广义函数基于被传递的实际参数构造一个可应用的方法列表。其次，这个可应用方法的列表按照它们的参数特化符中的特化程度(specificity)排序。最后，根据排序后列表中的顺序来取出这些方法并将它们的代码组合起来以产生有效方法。^①

为了找出可应用的方法，广义函数将实际参数与它的每一个方法中的对应参数特化符进行比较。当且仅当所有特化符均和对应的参数兼容，一个方法便是可应用的。

当特化符是一个类的名字时，如果该名字是参数的实际类名或是它的一个基类的名字，那么该特化符将是兼容的。（再次强调，不带有显式特化符的形参将隐式特化到类T上从而与任何参数兼容。）一个**EQL**特化符当且仅当参数和特化符中所指定的对象是同一个时才是兼容的。

由于所有参数都将在对应的特化符中被检查，它们都会影响一个方法是否是可应用的。显式地特化了超过一个形参的方法被称为多重方法(multimethod)。我将在16.8节中讨论它们。

在可应用的方法被找到以后，广义函数机制需要在将它们组合成一个有效方法之前对它们进行排序。为了确定两个可应用方法的顺序，广义函数从左到右比较它们的参数特化符，^②并且两个方法中第一个不同的特化符将决定它们的顺序，其中带有更加特定的特化符的方法排在前面。

由于只有可应用的方法在排序，你可以看出所有由类特化符命名的类对应的参数实际上都是它们的实例。在典型情况下，如果两个类特化符不同，那么其中一个将是另一个的子类。在那种情况下，命名了子类的特化符将被认为是更加相关的。这就是为什么在`checking-account`上特化了`account`的方法被认为比在`bank-account`上特化它的方法是更加相关的。

^① 尽管构造有效方法的过程听起来很费时，但在开发快速的Common Lisp实现过程中有相当多的努力被用于使上述过程更有效率，一种策略是缓存有效方法以便未来在相同参数类型上的调用可以被直接处理。

^② 事实上，比较特化符的顺序可以通过**DEFGENERIC**的选项:`argument-precedence-order`来定制，尽管该选项很少被用到。

多重继承稍微复杂化了特化性的概念，因为实际参数可能是两个类的实例，而两者都不是对方的子类。如果这样的类被用于参数特化符，那么广义函数就无法只通过子类比它们的基类更加相关这一规则来决定它们的顺序。在下一章里，我将讨论特化性的概念如何被扩展用于处理多重继承。目前我要说明的只是存在一个确定的算法来决定类特化符的顺序。

最后，**EQL**特化符总是比任何类特化符更加相关，并且由于只有可应用的方法被考虑，如果对于一个特定形参多于一个方法带有**EQL**特化符，那么它们一定全部带有相同的**EQL**特化符。这样对这些方法的比较将取决于其他参数。

16.6 标准方法组合

现在，你理解了找出可应用的方法并对它们进行排序的方式，你可以更进一步来观察最后一步——排序的方法列表是如何被组合成单一有效方法的。默认情况下，广义函数使用一种称为标准方法组合（standard method combination）的机制。标准方法组合将方法组合在一起，从而使**CALL-NEXT-METHOD**像你看到的那样工作——最相关的方法首先运行，然后每个方法可以通过**CALL-NEXT-METHOD**将控制传递给下一个最相关的方法。

不过，这里面还有更多的细节。到目前为止，我所讨论过的所有方法都称为主方法（primary method）。如同其名字所显示的，主方法被用于提供一个广义函数的主要实现。标准方法组合也支持三种类型的辅助方法：**:before**、**:after**和**:around**。附加方法定义是用**DEFMETHOD**像一个主方法那样写成的，但是它带有一个方法限定符（method qualifier），其命名了方法的类型，介于方法名和形参列表之间。例如，一个在类**bank-account**上特化了**account**形参的**withdraw**的**:before**方法以下面的定义开始：

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

每种类型的附加方法以不同的方式组合到有效方法之中。所有可应用的**:before**方法（不只是最相关的）都将作为有效方法的一部分来运行。如同其名字所显示的，这些**:before**方法将在最相关的主方法之前以最相关者优先的顺序来运行。这样，**:before**方法可用来做任何需要确保主方法可以运行的准备工作。例如，你可以使用特化在**checking-account**上的**:before**方法像下面这样来实现对现金账户的透支保护：

```
(defmethod withdraw :before ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft))))
```

这个**:before**方法相比于一个主方法有三个优点。其中之一是它使得该方法改变**withdraw**整体行为的方式变得非常直观，但它并不打算影响主要的行为或是改变返回的结果。

下一个优点在于，一个特化在比**checking-account**更相关类上的主方法将不会影响该**:before**方法，从而使得**checking-account**子类的作者可以更容易地扩展**withdraw**的行为而同时保存它的一部分老的行为。

最后，由于`:before`方法不需要调用`CALL-NEXT-METHOD`来将控制传递给其余的方法，所以就不可能因为忘记这点而引入bug。

其他的附加方法同样以它们的名字所建议的方式融合进有效方法。所有的`:after`方法都在主方法之后以最相关者最不优先的顺序运行，也就是说与`:before`方法相反。这样，`:before`和`:after`方法组合在一起创建了一系列嵌套包装在由主方法所提供的核心功能周边的环境。每一个更相关的`:before`方法将有机会设置环境以便不太相关的`:before`方法和主方法得以成功运行，而每一个更相关的`:after`方法将有机会在所有主方法和更不相关的`:after`之后进行清理工作。

最后，除了它们运行在所有其他方法的外围，`:around`将以非常类似主方法的方式被组合。这就是说，来自最相关`:around`方法的代码将在其他任何代码之前运行。在`:around`方法的主体中，`CALL-NEXT-METHOD`将指向下一个最相关的`:around`方法的代码，或是在最不相关的`:around`的方法中指向由`:before`方法、主方法和`:after`方法组成的复合体。几乎所有的`:around`方法都会含有一个对`CALL-NEXT-METHOD`的调用，因为如果不这样做的话，`:around`的方法就会完全劫持广义函数中除了最相关`:around`方法之外的所有方法的实现。

这种类型的方法劫持偶尔也会被用到，但是典型的`:around`方法通常被用于建立一些其他方法得以运行的动态上下文。例如绑定一个动态变量，或是建立一个错误处理器（我将在第19章里讨论这一点）。差不多一个`:around`方法不去调用`CALL-NEXT-METHOD`的唯一场合就是当它返回一个缓存自之前对`CALL-NEXT-METHOD`的调用时。不管怎么说，一个没有调用`CALL-NEXT-METHOD`的`:around`的方法有责任正确实现广义函数在方法可能应用到的所有类型参数下的语义，包括未来定义的子类。

附加方法只是一种更简洁和具体地表达特定常用模式的便利方式。它们并不能让你做到任何通过将带有额外努力的主方法与一些代码约定和额外输入相组合所不能做到的事情。也许它们最大的好处在于它们提供了一个扩展广义函数的统一框架。通常库将定义一个广义函数并提供默认的主方法，然后允许该库的用户通过定义适当的附加方法来定制它的行为。

16.7 其他方法组合

在标准方法组合之外，该语言还指定了九种其他的内置方法组合，也称为简单内置方法组合。你还可以自定义方法组合，尽管这是一个相对难懂的特性并且超出了本书的范围。我将简要介绍简单内置组合的功能。

所有的简单组合都遵循了相同的模式：和调用最相关主方法并让它通过`CALL-NEXT-METHOD`来调用次相关主方法的方式有所不同，简单方法组合通过将所有主方法的代码一个接一个地全部包装在一个由方法组合的名字所给出的函数、宏或特殊操作符的调用中来产生一个有效方法。⁹ 9种组合分别以下列操作符来命名：`+`、`AND`、`OR`、`LIST`、`APPEND`、`NCONC`、`MIN`、`MAX`和`PROGN`。另外简单组合只支持两种方法：按照刚刚描述的方式进行组合的主方法，以及`:around`方法，它和标准方法组合中的`:around`方法具有相似的工作方式。

例如，一个使用“+”方法组合的广义函数将返回其有主方法返回的结果之和。注意到由于这些宏的短路行为，**AND**和**OR**方法组合不一定会运行所有主方法——使用**AND**组合的广义函数将在一个方法返回**NIL**时立即返回，否则将返回最后一个方法的值。类似地，**OR**组合将返回第一个由任何方法返回的非**NIL**的值。

为了定义一个使用特定方法组合的广义函数，你可以在**DEFGENERIC**形式中包含一个:**:method-combination**选项。连同该选项所提供的值是你想要使用的方法组合的名字。例如，为了定义一个广义函数**priority**，其使用“+”方法组合返回所有单独方法的返回值之和，你可以写成下面这样：

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination +))
```

默认情况下，所有这些方法组合以最相关者优先的顺序组合主方法。尽管如此，你可以通过在**DEFGENERIC**形式中的方法组合名之后包含关键字:**:most-specific-last**来逆转这一顺序。该顺序在你使用“+”组合的时候可能无关紧要，除非方法带有副作用，但是出于演示的目的，你可以像下面这样使用最相关者最不优先的顺序来改变**priority**：

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination + :most-specific-last))
```

定义在使用这些组合之一的广义函数上的主方法必须被限定在该方法组合的名字上。这样，一个特定于**priority**的主方法可能看起来像这样：

```
(defmethod priority + ((job express-job)) 10)
```

这可以使你清楚地看到一个属于特定类型广义函数的方法定义。

所有简单内置方法组合也支持:**:around**方法，其工作方式与标准方法组合中的:**:around**方法类似：最相关的:**:around**方法在任何其他方法之前运行，而**CALL-NEXT-METHOD**被用于将控制传递给越来越不相关的:**:around**方法，直到到达组合的主方法。**:most-specific-last**选项并不影响:**:around**方法的顺序。并且如同我前面提到的，内置方法组合不支持:**:before**或:**:after**方法。

和标准方法组合一样，这些方法组合不允许你做到任何你不能手工做到的事情。但是它们确实可以允许你表达你所想要的事情，并且让语言来帮助你将所有东西组织在一起从而使你的代码更加简洁且更富有表达性。

这就是说，很可能在99%的时间里标准方法组合是你需要的东西。在其中剩下的1%的事件里，可能99%的情况将被一个简单内置方法组合处理。如果你遇到了1%中的1%的情况，其中没有内置组合可以满足需要，那么你可以在你喜爱的Common Lisp参考中查询**DEFINE-METHOD-COMBINATION**。

16.8 多重方法

显式地特化了超过一个广义函数的必要形参的方法称为多重方法。多重方法是广义函数和消息传递真正有区别的地方。多重方法无法存在于消息传递语言之中是因为它们不属于一个特定的

类；相反，每一个多重方法都定义了一个给定广义函数的部分实现，并且当广义函数以匹配所有该方法的特化参数被调用时采用。

多重方法与方法重载

曾经使用过诸如Java和C++这些静态类型消息传递语言的程序员们可能认为，多重方法听起来类似于这些语言中一种称为方法重载（method overloading）的特性。不过事实上这两种语言特性相当不同，因为重载的方法是在编译期被选择的，其所基于的是编译期的参数类型而不是运行期。为了观察方法重载的工作方式，考虑下面的两个Java类：

```
public class A {
    public void foo(A a) { System.out.println("A/A"); }
    public void foo(B b) { System.out.println("A/B"); }
}

public class B extends A {
    public void foo(A a) { System.out.println("B/A"); }
    public void foo(B b) { System.out.println("B/B"); }
}
```

现在考虑当你从下面的类中运行main方法时将会发生什么。

```
public class Main {
    public static void main(String[] argv) {
        A obj = argv[0].equals("A") ? new A() : new B();
        obj.foo(obj);
    }
}
```

当你告诉Main来实例化A时，它像你可能期待的那样打印出“A/A”。

```
bash$ java com.gigamonkeys.Main A
A/A
```

不过，如果你告诉Main来实例化B，那么obj的真正类型将只有一半被实际分发。

```
bash$ java com.gigamonkeys.Main B
B/A
```

如果重载的方法像Common Lisp的多重方法那样工作，那么上面将打印出“B/B”。在消息传递语言里有可能手工实现多重分发，但这将与消息传递模型背道而驰，因为多重分发的方法中的代码并不属于任何一个类。

多重方法对于所有这些情形都很完美，而在一个消息传递语言里你将很难决定一个特定行为应该属于哪个类。一个鼓在用鼓棒敲它的时候产生的声音究竟是由鼓的类型还是棒的类型决定的？当然，两者都是。为了在Common Lisp中对这种情况建模，你可以简单地定义一个接受两个参数的广义函数beat。

```
(defgeneric beat (drum stick)
  (:documentation
   "Produce a sound by hitting the given drum with the given stick.))
```

然后，你可以定义不同的多重方法来实现用于你所关心的不同组合的beat。例如：

```
(defmethod beat ((drum snare-drum) (stick wooden-drumstick)) ...)  
(defmethod beat ((drum snare-drum) (stick brush)) ...)  
(defmethod beat ((drum snare-drum) (stick soft-mallet)) ...)  
(defmethod beat ((drum tom-tom) (stick wooden-drumstick)) ...)  
(defmethod beat ((drum tom-tom) (stick brush)) ...)  
(defmethod beat ((drum tom-tom) (stick soft-mallet)) ...)
```

多重方法不能帮助处理组合爆炸。如果你需要建模五种类型的鼓和六种类型的鼓棒，并且每一种组合都产生不同的声音，那么不存在更好的办法。无论是否使用多重方法你都需要三十种不同的方法来实现所有的组合。多重方法使你免于手工编写大量用于分发的代码，而让你使用与处理特化在单一参数上的方法相同的内置多态分发技术。^①

多重方法还可以使你免于将一组类互相关联在一起。在鼓/棒示例中，鼓类的实现不需要知道任何关于不同类型鼓棒的信息，而鼓棒类也不需要知道任何关于不同类型鼓的信息。多重方法将完全无关的类联系在一起描述它们的组合行为，而不要求这些类彼此之间的任何互操作。

16.9 未完待续……

我已经介绍了广义函数的基础，并且还介绍了超出范围之外的内容，即Common Lisp对象系统中的动词。在下一章里，我将向你展示如何定义你自己的类。

① 在没有多重方法的语言里，你必须手工编写分发代码来实现依赖于超过一个对象的类的行为。流行的Visitor的设计模式的目的就是结构化一系列单一分发的方法调用从而提供多重分发。尽管如此，它要求有一种彼此知道的类。Visitor模式在被用作分发超过两个对象时还会快速地陷入组合爆炸。

重新审视面向对象：类

如 果说广义函数是对象系统的动词，那么类就是名词。如同我在前面一章里提到的，Common Lisp程序中所有的值都是某个类的实例。更进一步，所有的类都被组织成以类为根的单一层次体系。

类层次的体系由两个主要的类家族构成，即内置的类和用户定义的类。到目前为止，你已经学过的代表数据类型的类，诸如`INTEGER`、`STRING`和`LIST`这样的类，都是内置的。它们生活在类层次体系中它们自己的区域里，按照适当的子类和基类关系组织在一起，并且由那些我在本书中到目前为止讨论过的函数所管理。你不能创建这些类的子类，但是正如你在前一章里看到的，你可以定义特化在它们之上的方法，从而有效地扩展那些类的行为。^①

但是当你想要创建新的名词时，例如前一章里用来表示银行帐户的那些类，你就需要定义你自己的类。这就是本章的主题。

17.1 DEFCLASS

你可以使用`DEFCLASS`宏来创建用户定义的类。由于一个类关联的行为是通过定义广义函数和特化在该类上的方法决定的，`DEFCLASS`的责任仅仅是将类定义为一种数据类型。

一个类作为数据类型的三个方面是它的名字、它与其他类的关系以及构成该类实例的那些槽(slot)的名字。^②一个`DEFCLASS`的基本形式很简单。

```
(defclass name (direct-superclass-name*)
  (slot-specifier*))
```

什么是“用户定义的类”

术语“用户定义的类”不是来自语言标准的术语。从技术上来讲，当我说“用户定义的类”时，我指的是那些属于`STANDARD-OBJECT`的子类并且其元类(meta-class)是`STANDARD-CLASS`的类。但由于我不打算谈论你可以定义不是`STANDARD-OBJECT`的子类并且其元类不是

^① 为一个已有的类定义新的方法，对于那些曾经使用诸如C++和Java这样的静态类型语言的人们来说可能听起来有些奇怪。在这些语言里，类的所有方法必须被定义为该类定义的一部分。但是具有使用诸如Smalltalk和Objective-C这类动态类型面向对象语言经验的程序员们则不觉得为已有类添加新行为有任何奇怪之处。

^② 在其他面向对象语言里，“槽”可能被称为字段(field)、成员变量(member variable)或属性(attribute)。



STANDARD-CLASS的那些类的方式，你根本不需要关心这点。“用户定义的”并不是一个用来描述这些类的完美术语，因为实现可能以相同的方式定义了特定的类。不过，将它们称为标准类可能会带来更多的困惑，因为诸如`INTEGER`和`STRING`这样的内置类也是标准的，它们是由语言标准定义的但却没有扩展`STANDARD-OBJECT`。更复杂的事情在于，用户也有可能定义不是`STANDARD-OBJECT`子类的新类。特别的是，宏`DEFSTRUCT`同样定义了新的类，但那在很大程度上是为了向后兼容——`DEFSTRUCT`出现在CLOS之前，并且当CLOS被集成进语言时曾被改进用于定义类。因此在本章里，我将只谈论那些由`DEFCLASS`定义的使用默认的`STANDARD-CLASS`作为元类的类，并且由于缺少一个更好的术语，我将把它们称为“用户定义的”。

与函数和变量一样，你可以使用任何符号作为一个新类的名字。^①类的名字与函数和变量的名字处在独立的名字空间里，因此你可以让类、函数和变量全部带有相同的名字。你将使用类名作为`MAKE-INSTANCE`的参数，该函数用来创建用户定义类的新实例。

那些`direct-superclass-name`指定了该新类将成为其子类的那些类。如果没有基类被列出，那么新类将直接成为`STANDARD-OBJECT`的子类。任何列出的类必须是其他用户定义的类，这确保了每一个新类都将最终追溯到`STANDARD-OBJECT`。`STANDARD-OBJECT`又是`T`的子类，因此，所有用户定义的类都是同样含有全部内置类的单一类层次体系的一部分。

在暂时省略槽描述符的情况下，前一章里你用到的某些类的`DEFCLASS`形式可能看起来像这样：

```
(defclass bank-account () ...)

(defclass checking-account (bank-account) ...)

(defclass savings-account (bank-account) ...)
```

我将在17.8节里讨论在`direct-superclass-name`中列出多于一个直接基类的含义。

17.2 槽描述符

`DEFCLASS`形式的大部分是由槽描述符的列表组成的。每个槽描述符定义的槽都属于该类的每个实例。实例中的每个槽都是一个可以保存值的位置，该位置可以通过函数`SLOT-VALUE`来访问。`SLOT-VALUE`接受一个对象和一个槽的名字作为参数并返回给定对象中该命名槽的值。它可以和`SETF`一起使用来设置对象中某个槽的值。

一个类也从它的所有基类中继承槽描述符，因此，实际存在于任何对象中的槽的集合是一个类的`DEFCLASS`形式中指定的所有槽和它的全部基类中指定的槽的并集。

在最小情况下，一个槽描述符可以只是一个名字。例如，你可以将`bank-account`类定义为带有两个槽`customer-name`和`balance`，如下所示：

^① 跟为函数和变量命名一样，你可以使用任何符号作为类名的这个说法并不是很正确，你不能使用由语言标准所定义的名字。你将在第21章里看到如何避免这样的名字冲突。

```
(defclass bank-account ()
  (customer-name
   balance))
```

该类的每个实例都含有两个槽，一个用来保存该账户所属客户的名字而另一个用来保存当前余额。借助该定义，你可以用`MAKE-INSTANCE`来创建新的`bank-account`对象。

```
(make-instance 'bank-account) → #<BANK-ACCOUNT @ #x724b93ba>
```

`MAKE-INSTANCE`的参数是想要实例化的类的名字，而返回的值就是新的对象。^①一个对象的打印形式取决于广义函数`PRINT-OBJECT`。在本例中，可应用的方法是由实现提供的特化在`STANDARD-OBJECT`上的方法。每一个对象都可以被打印成随后可被读回的形式，因此`STANDARD-OBJECT`打印方法使用了#<>语法，这将导致读取器在它试图读取该对象时报错。打印表示的其余部分是由实现定义的，但通常是一些类似于上面所显示的输出，其中包括该类的名字和一些诸如该对象的内存地址这样的可区别值。在第23章里，你将看到一个关于如何定义`PRINT-OBJECT`方法的例子，它使得一个特定类的对象可以被打印成更具说明性的形式。

使用刚刚给出的`bank-account`定义，创建出的新对象将带有未绑定（`unbound`）的槽。任何尝试获取未绑定槽的值的操作都将会报错，因此你必须在读取一个槽之前先设置它：

```
(defparameter *account* (make-instance 'bank-account)) → *ACCOUNT*
(setf (slot-value *account* 'customer-name) "John Doe") → "John Doe"
(setf (slot-value *account* 'balance) 1000) → 1000
```

现在你可以访问这些槽的值了：

```
(slot-value *account* 'customer-name) → "John Doe"
(slot-value *account* 'balance) → 1000
```

17.3 对象初始化

由于你不能对一个带有未绑定槽的对象做太多事，因此如果可以创建带有预先初始化槽的对象就非常好。Common Lisp提供了三种方式来控制槽的初始值。前面两种是通过在`DEFCLASS`形式中向槽描述符添加选项来实现的：通过`:initarg`选项，你可以指定一个随后作为`MAKE-INSTANCE`的关键字形参的名字并使该参数的值保存在槽中。另一个选项：`:initform`可以让你指定一个Lisp表达式在没有`:initarg`参数传递给`MAKE-INSTANCE`时为该槽计算一个值。最后，为了完全控制初始化过程，你可以在广义函数`INITIALIZE-INSTANCE`上定义一个方法，它将被`MAKE-INSTANCE`调用。^②

① `MAKE-INSTANCE`的参数实际上既可以是一个类的名字，也可以是一个由函数`CLASS-OF`或`FIND-CLASS`返回的类对象。

② 另一种影响槽的初始值的方式是通过`DEFCLASS`的`:default-initargs`选项。当一个特定的`MAKE-INSTANCE`调用没有给定该值时，该选项用来指定将被求值的形式以及提供特定初始化形参的参数。目前你不需要担心`:default-initargs`。

包含诸如`:initarg`或`:initform`等选项的槽描述符被写成以槽的名字开始后跟选项的列表。例如，如果你想要修改`bank-account`的定义，从而允许`MAKE-INSTANCE`的调用者传递客户名和初始余额，并为余额提供一个零美元的默认值，你可以写成这样：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name)
   (balance
    :initarg :balance
    :initform 0)))
```

现在你可以创建一个账户并同时指定所有的槽值：

```
(defparameter *account*
  (make-instance 'bank-account :customer-name "John Doe" :balance 1000))

(slot-value *account* 'customer-name) → "John Doe"
(slot-value *account* 'balance) → 1000
```

如果你没有提供`:balance`参数给`MAKE-INSTANCE`，那么`balance`的`SLOT-VALUE`将通过求值由`:initform`选项指定的形式而得到。但如果你没有指定`:customer-name`参数，那么`customer-name`槽将是未绑定的，并且在你设置它之前尝试读取它的操作将会报错。

```
(slot-value (make-instance 'bank-account) 'balance) → 0
(slot-value (make-instance 'bank-account) 'customer-name) → error
```

如果你想要确保在创建账户的同时也提供客户名，那么你可以在初始化形式中产生一个错误，因为它只在没有提供初始化参数时被求值一次。你还可以使用初始化形式在每次它们被求值时生成一个不同的值——初始化形式对于每个对象都被重新求值。为了体会这些技术，你可以修改`customer-name`槽描述符并添加一个新的槽`account-number`，它被初始化为一个永远递增的计数器的值。

```
(defvar *account-numbers* 0)

(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))))
```

多数时候，`:initarg`和`:initform`选项的组合可以很好地初始化一个对象。不过，尽管初始化形式可以是任何Lisp表达式，但它却无法访问正在初始化的对象，因此它不能基于一个槽的值来初始化另一个槽。对于这种情况你需要在广义函数`INITIALIZE-INSTANCE`上定义一个方法。

基于它们的`:initarg`和`:initform`选项，在`STANDARD-OBJECT`上特化的`INITIALIZE-INSTANCE`主方法负责槽的初始化工作。由于你不想干扰这些工作，最常见的添加定制初始化代码

的方式是定义一个特化在你的类上的`:after`方法。^①例如，假设你想要添加一个`account-type`槽并需要根据该账户的初始余额将其设置成`:gold`、`:silver`或`:bronze`这些值中的一个。你可以将你的类定义改成下面这样，其中添加了一个没有选项的`account-type`槽：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))
   account-type))
```

然后你可以为`INITIALIZE-INSTANCE`定义一个`:after`方法，根据保存在`balance`槽中的值来设置`account-type`槽。^②

```
(defmethod initialize-instance :after ((account bank-account) &key)
  (let ((balance (slot-value account 'balance)))
    (setf (slot-value account 'account-type)
      (cond
        ((>= balance 100000) :gold)
        ((>= balance 50000) :silver)
        (t :bronze)))))
```

为了保持该方法的形参列表与广义函数一致，形参列表中的`&key`是必不可少的。广义函数`INITIALIZE-INSTANCE`指定的形参列表包含了`&key`，从而允许个别方法可以指定它们自己的关键字参数，但其对特定的关键字参数却没有要求。这样，每一个方法都必须指定`&key`，哪怕它们没有指定任何`&key`参数。

另一方面，如果特化在某个特定类上的`INITIALIZE-INSTANCE`方法指定了一个`&key`参数，那么在创建该类的实例时，该参数就成为了一个`MAKE-INSTANCE`的合法参数。例如，有时在你开户时银行会支付一定比例的初始余额作为奖励，那么你可以像下面这样使用一个接受关键字参数来指定奖励百分比的`INITIALIZE-INSTANCE`方法来实现这一点：

```
(defmethod initialize-instance :after ((account bank-account)
                                         &key opening-bonus-percentage)
  (when opening-bonus-percentage
    (incf (slot-value account 'balance)
```

^① 在Common Lisp中向`INITIALIZE-INSTANCE`添加一个`:after`方法，相当于在Java或C++中定义一个构造函数或是在Python中定义一个`__init__`方法。

^② 在未习惯使用附加方法之前，你可能会犯的错误是，在`INITIALIZE-INSTANCE`上定义了一个方法而没有使用`:after`限定符。如果你这样做了，你将得到一个覆盖了默认方法的新的主方法。你可以使用函数`REMOVE-METHOD`和`FIND-METHOD`来移除不想要的主方法。某些开发环境可能提供图形用户接口来实现同样的事情。

```
(remove-method #'initialize-instance
  (find-method #'initialize-instance () (list (find-class 'bank-account))))
```

```
(* (slot-value account 'balance) (/ opening-bonus-percentage 100))))
```

通过定义这个**INITIALIZE-INSTANCE**方法，你使:`:opening-bonus-percentage`在创建`bank-account`时成为了**MAKE-INSTANCE**的合法参数。

```
CL-USER> (defparameter *acct* (make-instance
                                     'bank-account
                                     :customer-name "Sally Sue"
                                     :balance 1000
                                     :opening-bonus-percentage 5))
*ACCT*
CL-USER> (slot-value *acct* 'balance)
1050
```

17.4 访问函数

从**MAKE-INSTANCE**到**SLOT-VALUE**，你有了用于创建和管理你的类实例的所有工具。你想要做的其他任何事都可以用这两个函数来实现。不过，每一位了解优秀的面向对象编程实践原则的人都知道，直接访问一个对象的槽（或字段或成员变量）可能导致脆弱的代码。问题在于直接访问槽会将你的代码过于紧密地绑定到你的类的具体结构上。例如，假设你打算改变`bank-account`的定义，不再保存数值形式的当前余额，而是保存一个带有时间戳的提款和存款列表。在你改变了类定义来移除该槽或是保存新的列表到旧的槽时，直接访问`balance`槽的代码将很可能被打断。另一方面，如果你定义了一个用来访问该槽的`balance`函数，那么随后你可以重定义它，在类的内部表示改变的情况下保留其行为。并且使用这样一个函数的代码将无需修改而继续工作。

另一个使用访问函数而不是直接通过**SLOT-VALUE**来访问槽的优点在于，它可以让你限制外部代码修改槽的方式。^①对于`bank-account`类的用户来说能够得到当前余额就可以了，但你可能想让余额的所有修改通过你将提供的其他函数访问到，例如`deposit`和`withdraw`。如果客户知道他们被假定只通过已发布的函数型API来管理对象，那么你可以提供一个`balance`函数但不使它成为可`SETF`的，如果你想让余额是只读的。

最后，使用访问函数可以使你的代码更整齐，因为它帮助你在大量情况下都不必使用相当繁琐的**SLOT-VALUE**。

很容易定义一个函数来读取`balance`槽的值。

```
(defun balance (account)
  (slot-value account 'balance))
```

不过，如果你知道你打算定义的`bank-account`的子类，那么将`balance`定义成一个广义函数可能是个好主意。通过这种方式，你可以在`balance`上为这些子类提供不同的方法或使用附加

^①当然，提供一个访问函数并不能真的产生任何限制，因为其他代码仍然可以使用**SLOT-VALUE**来直接访问槽。Common Lisp并没有提供C++和Java这些语言里所提供的严格对象封装。不过，如果一个类的设计者提供了访问函数而你忽略了它们仍然使用**SLOT-VALUE**，那么你最好知道你在做什么。你也可以使用我将在第21章里讨论的包系统，其更清楚地说明了某些槽不用于直接访问，方法就是不导出这些槽的名字。

方法来扩展其定义。因此你可能写出下面的定义：

```
(defgeneric balance (account))

(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

正如我已讨论过的，你不希望调用者直接设置余额。但对于其他槽，诸如customer-name，你可能也想提供一个函数来设置它们。定义这样的函数最简洁的方式是将其定义为SETF函数。

SETF函数是一种扩展SETF的方式，其定义了一种新的位置类型使其知道如何设置它。SETF函数的名字是一个两元素列表，其第一个元素是符号setf而第二个元素是一个符号，通常是一个用来访问该SETF函数将要设置的位置的函数名。SETF函数可以接受任何数量的参数，但第一个参数总是赋值到位置上的值。^①例如，你可以定义SETF函数像下面这样设置bank-account中的customer-name槽：

```
(defun (setf customer-name) (name account)
  (setf (slot-value account 'customer-name) name))
```

在求值该定义之后，一个类似

```
(setf (customer-name my-account) "Sally Sue")
```

的表达式将被编绎成一个对你刚刚定义SETF函数的调用，其中"Sally Sue"作为第一个参数而my-account的值作为第二个参数。

当然，和读取函数一样，你希望你的SETF函数是广义的，因此你将实际像下面这样定义它：

```
(defgeneric (setf customer-name) (value account))

(defmethod (setf customer-name) (value (account bank-account))
  (setf (slot-value account 'customer-name) value))
```

并且你当然也想为customer-name定义一个读取函数。

```
(defgeneric customer-name (account))

(defmethod customer-name ((account bank-account))
  (slot-value account 'customer-name))
```

这允许你写出下面的表达式：

```
(setf (customer-name *account*) "Sally Sue") → "Sally Sue"

(customer-name *account*) → "Sally Sue"
```

编写这些访问函数没有什么困难的，但是完全手工编写它们就跟Lisp风格不太吻合了。因此，DEFCLASS提供了三个槽选项，从而允许你为一个特定的槽自动创建读取和写入函数。

^① 定义一个SETF函数，比如说(setf foo)，其带来的一种后果是，如果你还定义了对应的访问函数，在这种情况下是foo，那么你将可以在这个新的位置类型上使用构建在SETF之上的所有修改宏，比如INCF、DECREF、PUSH和POP。

:read选项指定广义函数的名字，该函数只接受一个对象参数。当DEFCLASS被求值时，如果广义函数不存在则创建它。然后，为它添加一个方法，此方法基于新类特化一个参数并返回该槽的值。该名字可以是任意的，但通常将其命名成与槽本身相同的名字。这样，代替了前面给出的那些显式编写的balance广义函数的方法，你可以将bank-account中的balance槽的槽描述符修改成下面这样：

```
(balance
  :initarg :balance
  :initform 0
  :reader balance)
```

:write选项用来创建设置一个槽的值的广义函数和方法。该函数和方法按照SETF函数的要求创建，接受新值作为其第一个参数并把它作为结果返回，因此你可以通过提供一个诸如(setf customer-name)这样的名字来定义SETF函数。例如，你可以将槽描述符改变成下面的样子来为customer-name提供等价于前面所写的读取和写入方法：

```
(customer-name
  :initarg :customer-name
  :initform (error "Must supply a customer name.")
  :reader customer-name
  :writer (setf customer-name))
```

由于经常同时需要用于读取和写入的函数，因此DEFCLASS还提供了一个选项:accessor来同时创建读取函数和对应的SETF函数。取代刚刚给出的槽描述符，一般情况下还可以写成这样：

```
(customer-name
  :initarg :customer-name
  :initform (error "Must supply a customer name.")
  :accessor customer-name)
```

最后，还有一个你应当知道的槽选项是:documentation选项，使用它可以提供一个字符串来记录一个槽的用途。将所有这些放在一起，并为account-number和account-type槽添加了读取方法，现在bank-account类的DEFCLASS形式看起来像下面这样：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name.")
    :accessor customer-name
    :documentation "Customer's name")
   (balance
    :initarg :balance
    :initform 0
    :reader balance
    :documentation "Current account balance")
   (account-number
    :initform (incf *account-numbers*)
    :reader account-number
    :documentation "Account number, unique within a bank.")
   (account-type
    :reader account-type
    :documentation "Type of account, one of :gold, :silver, or :bronze.")))
```

17.5 WITH-SLOTS 和 WITH-ACCESSORS

尽管使用访问函数将使代码更易于维护，但使用它们仍然有些繁琐。当编写那些实现一个类底层行为的方法时，情况将会更严重，这时你可能特别想直接访问槽来设置那些没有写入函数的槽，或是得到一些槽的值而无需为其定义读取函数。

这就是SLOT-VALUE适用的场合，不过它仍然很繁琐。更糟糕的是，一个多次访问同一个槽的函数或方法可能会产生大量对访问函数和SLOT-VALUE的调用。例如，就算是下面这个相当简单的方法也充满了对balance和SLOT-VALUE的调用，该方法在bank-account账户余额低于某个最小值时对其科以罚款：

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (balance account) *minimum-balance*)
    (decf (slot-value account 'balance) (* (balance account) .01))))
```

而如果你打算直接访问槽值以避免运行附加的方法，它会变得更加混乱。

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (slot-value account 'balance) *minimum-balance*)
    (decf (slot-value account 'balance) (* (slot-value account 'balance) .01))))
```

两个标准宏WITH-SLOTS和WITH-ACCESSORS可以减轻这种混乱情况。两个宏都创建了一个代码块，在其中，简单的变量名可用于访问一个特定对象的槽。WITH-SLOTS提供了对槽的直接访问，就像SLOT-VALUE那样，而WITH-ACCESSORS提供了一个访问方法的简称。

WITH-SLOTS的基本形式如下所示：

```
(with-slots (slot*) instance-form
  body-form*)
```

每一个slot元素可以是一个槽的名字，它也用作一个变量名；或是一个两元素列表，第一个元素是一个用作变量的名字，第二个元素则是对应槽的名字。instance-form被求值一次来产生将要访问其槽的对象。在代码体内，这些变量名的每一次出现都被翻译成一个对SLOT-VALUE的调用，该对象和适当的槽名作为其参数。^①这样，你可以像下面这样编写access-low-balance-penalty：

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots (balance) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

或者使用两元素列表形式，像这样：

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots ((bal balance)) account
```

^① 由WHITH-SLOTS和WITH-ACCESSORS提供的“变量”名并不是真正的变量，它们是使用一种特殊类型的宏来实现的，这种宏称为符号宏 (symbol macro)，它允许一个简单的名字被展开成任意代码。在语言中引入符号宏主要是为了支持WITH-SLOTS和WITH-ACCESSORS，但你也可以将它们用于自己的目的。我将在第20章讨论它们的更多细节。

```
(when (< bal *minimum-balance*)
  (decf bal (* bal .01)))))
```

如果你已经用一个`:accessor`而不只是`:reader`定义了`balance`，那么还可以使用`WITH-ACCESSORS`。`WITH-ACCESSORS`形式和`WITH-SLOTS`相同，除了槽列表的每一项都必须是包含一个变量名和一个访问函数名字的两元素列表。在`WITH-ACCESSORS`的主体中，对一个变量的引用等价为对相应访问函数的调用。如果访问函数是可以`SETF`的，那么该变量也可以。

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-accessors ((balance balance)) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

上面的代码中第一个`balance`是变量的名字，第二个是访问函数的名字，它们不必相同。例如，你可以编写一个方法来合并两个账户，其中使用两个`WITH-ACCESSORS`调用，每个账户一个。

```
(defmethod merge-accounts ((account1 bank-account) (account2 bank-account))
  (with-accessors ((balance1 balance)) account1
    (with-accessors ((balance2 balance)) account2
      (incf balance1 balance2)
      (setf balance2 0))))
```

是使用`WITH-SLOTS`还是使用`WITH-ACCESSORS`，与在`SLOT-VALUE`和一个访问函数之间进行选择是一样的：提供类基本功能的底层代码可以使用`SLOT-VALUE`和`WITH-SLOTS`直接修改那些不被访问函数支持的槽，或是为了显式地避免那些可能定义在访问函数上的附加方法所带来的影响。但你通常都应当使用访问函数或`WITH-ACCESSORS`，除非有特定的理由不这样做。

17.6 分配在类上的槽

你需要知道的最后一个槽选项是`:allocation`。`:allocation`的值可以是`:instance`或`:class`，如果没有指定则默认为`:instance`。当一个槽带有`:class`分配选项时，该槽只有单一值存储在类中并且被所有实例所共享。

尽管如此，`:class`槽和`:instance`槽的访问方法相同——通过`SLOT-VALUE`或访问函数来访问，这意味着你只能通过该类的一个实例来访问该槽的值，尽管它实际并没有保存在实例中。`:initform`和`:initarg`选项本质上也具有相同的效果，只是`:initform`将在类定义时而不是每次创建实例时求值。另一方面，传递`:initarg`给`MAKE-INSTANCE`会设置该值，从而影响该类的所有实例。

由于不能在没有该类实例的情况下访问一个类分配的槽，类分配的槽事实上并不等价于诸如Java、C++和Python这些语言里的静态字段或类字段。^①而且，类分配的槽主要用来节省空间。

^① 元对象协议（Meta Object Protocol, MOP）其不是语言标准的一部分，但被多数Common Lisp实现支持，它提供了一个函数`class-prototype`，该函数可以返回一个类的实例用来访问类槽。如果你正在使用一个支持MOP的实现，并且刚好在翻译一些来自其他语言的带有大量对静态字段或类字段使用的代码，那么该函数将给你一种简化翻译过程的方式。但这并不是惯用的做法。

如果你打算创建一个类的许多实例且所有实例都打算带有对同一个对象的引用，比如一个共享的资源池，那么可以通过使该槽成为类分配的槽而节省由每个实例都带有它们自己的引用所产生的开销。

17.7 槽和继承

正如我在前面章节里讨论的，类通过广义函数机制继承了来自其基类的行为，即一个在类A上进行特化的方法不仅可以应用在A的直接实例上，还可以应用在A的子类的实例上。类也可以从它的基类中继承槽，但手法上稍有不同。

在Common Lisp中，一个给定对象只能拥有一个带有给定名字的槽。尽管如此，一个给定类的继承层次关系中可能有多个类指定了具有同一个特定名字的槽。这既可能是因为一个子类包含了与其父类所指定的槽具有相同名字的槽描述符，也可能是多个基类指定了具有相同名字的槽。

Common Lisp处理这些情形的方式是，将来自新类和所有其基类的同名描述符合并在一起，并为每个唯一的槽名创建单一的描述符。当合并描述符时，不同的槽选项有不同的处理方式。例如，由于一个槽只能有单一的默认值，那么如果多个类指定了:initform，新类将使用来自最相关类的那个。这允许子类可以指定一个与它本该继承的不同的默认值。

另一方面，:initargs不需要是互斥的，槽描述符中的每个:initarg选项都将创建一个可用来初始化该槽的关键字形参。多个参数不会产生冲突，因此新的槽描述符将含有所有的:initargs。**MAKE-INSTANCE**的调用者可以使用任何一个:initargs来初始化该槽。如果调用者传递了多个关键字参数来初始化同一个槽，那么会使用**MAKE-INSTANCE**调用中最左边的参数。

继承得到的:reader、:writer和:accessor选项不会包含在合并了的槽描述符中，因为由基类的**DEFCLASS**创建的这些方法已经可以在新类上。不过，新类可以通过提供它自己的:reader、:writer或:accessor选项来定义其自己的访问函数。

最后，:allocation选项和:initform一样，由指定该槽的最相关的类决定。这样，有可能一个类的所有实例共享了一个:class槽，而它的子类的每个实例可能带有相同名字的自己的:instance槽。随后一个子类的子类可能将其重定义回:class槽，从而使该类的所有实例再次共享单一的槽。在后面的这种情况下，由子类的子类的实例共享的槽和由最初的基类共享的槽是不同的。

例如你有下面的类：

```
(defclass foo ()
  ((a :initarg :a :initform "A" :accessor a)
   (b :initarg :b :initform "B" :accessor b)))

(defclass bar (foo)
  ((a :initform (error "Must supply a value for a")
   (b :initarg :the-b :accessor the-b :allocation :class)))
```

当实例化类bar时，你可以使用继承了的初始化参数:a来为槽a指定值，事实上必须这样做才能避免出错，因为由bar提供的:initform覆盖了继承自foo的那个。为了初始化b槽，可以使

用继承的初始化参数:`b`或者新的初始化参数:`the-b`。不过，由于`bar`中的`b`槽带有`:allocation`选项，指定的值将保存在由`bar`的所有实例共享的槽中。同样的槽既可以使用在`foo`上特化的广义函数`b`的方法来访问，也可以使用直接在`bar`上特化的广义函数`the-b`的新方法来访问。为了访问`foo`或`bar`上的`a`槽，你将继续使用广义函数`a`。

通常，合并槽定义可以工作得很好。尽管如此，你需要关注当使用多重继承时，两个碰巧带有相同名字的无关的槽会被合并成新类中的单一的槽。这样，当在不同类上特化的方法应用在一个扩展了这些类的类上时，它们可能最终操作在同一个槽上。这实际上并不是太大的问题，因为你可以使用即将在第21章里学到的包（package）系统，来避免互不相关的代码中的名字冲突。

17.8 多重继承

到目前为止，你看到的所有类都只有单一的直接基类。Common Lisp也支持多重继承——一个类可以有多个直接基类，从所有这些类中继承可应用的方法和槽描述符。

多重继承并没有在本质上改变任何目前为止我所谈及的继承机制——每个用户定义的类已经带有多个基类，因为它们全部扩展至`STANDARD-OBJECT`，而后者扩展了`T`，所以它们至少有两个基类。多重继承带来的问题是一个类可以有超过一个的直接基类。这使得类的特化性概念在用于构造一个广义函数的有效方法以及合并继承的槽描述符时，会变得更加复杂。

这就是说，如果每个类都只有一个直接基类，那么决定类的特化性将极其简单。一个类及其所有基类可以排序成一条直线，从该类开始，后接它的直接基类，然后是后者的直接基类，最后一直上溯到`T`。但是当一个类有多个直接基类时，这些基类通常是彼此互不相关的。确实，如果一个类是另一个类的子类，那么你不会同时需要它们的直接子类。在这种情况下，子类比其基类更加相关这一规则不足以排序所有的基类。因此，Common Lisp提供了第二条规则，根据`DEFCLASS`的直接基类列表中列出的顺序来排序不相关的基类，更早出现在列表中的类被认为比列表中后面的类更相关。这条规则被认为有些随意，但确实可以允许每个类都有一个线性的类优先级列表（class precedence list），它可被用于检测某个基类是否比其他基类更相关。尽管如此，要注意并不存在所有类的全序。每个类都有其自己的类优先级列表，而同一个类可能以不同的顺序出现在不同类的类优先级列表中。

为了了解它是如何工作的，我们向银行应用中添加一个类`money-market-account`。一个货币市场账户组合了来自支票账户和储蓄账户的特征：客户可以填写支票，也可以挣得利息。你可以像下面这样定义它：

```
(defclass money-market-account (checking-account savings-account) ())
```

`money-market-account`的类优先级列表如下所示：

```
(money-market-account
  checking-account
  savings-account
  bank-account
  standard-object
  t)
```

注意该列表是怎样同时满足两条规则的：每个类都出现在它所有基类之前，并且*checking-account*和*savings-account*按照`DEFCLASS`中指定的顺序出现。

该类没有定义自己的槽，但是它会继承来自其两个直接基类的槽，包括两个直接基类继承自其基类的槽。同样，任何应用在类优先级列表中的任何类上的方法也将应用在*money-market-account*对象上。由于同一个槽的所有槽描述符都被合并了，因此*money-market-account*从*bank-account*中两次继承相同的槽描述符是不会有什么问题的。^①

当不同的基类提供了完全无关的槽和行为时，多继承最容易理解。例如，*money-market-account*将从*checking-account*中继承用于处理支票的槽和行为，而从*savings-account*中继承用于计算利息的槽和行为。你不需要为只从一个或另一个基类继承的方法和槽担心类优先级列表。

尽管如此，也有可能从不同的基类中继承同一广义函数的不同方法。在这种情况下，类优先级列表将发挥其作用。例如，假设银行应用定义了一个广义函数*print-statement*来生成月对账单。假设已经有了在*checking-account*和*savings-account*上特化的*print-statement*方法。这两个方法对于*money-market-account*实例来说都是可应用的，但在*checking-account*上特化的方法被认为比在*savings-account*上特化的方法更加相关，因为*checking-account*在*money-market-account*的类优先级列表中出现在*savings-account*之前。

假设继承到的方法都是主方法并且你还没有定义任何其他方法，那么如果你在*money-market-account*上调用*print-statement*，则在*checking-account*上特化的方法将被使用。不过，这并不一定可以给你想要的行为，因为你可能希望货币市场账户的对账单中同时含有来自支票账户和储蓄账户对账单的元素。

可以用几种方式来修改用于*money-market-account*的*print-statement*的行为。一种直接的方式是定义一个在*money-market-account*上特化的新的主方法。这可以让你更好地控制新行为，但可能需要比我即将讨论的其他选项要求更多的新代码。问题在于，当你可以使用`CALL-NEXT-METHOD`来“向上”调用下一个最相关方法时，也就是在*checking-account*上特化的那个方法，就没有办法来调用一个特定的不太相关的方法，例如在*savings-account*上特化的方法。因此，如果你想要重用那些打印对账单中*savings-account*部分的代码，就需要将那些代码分解成单独的函数，它随后可同时被*money-market-account*和*savings-account*的*print-statement*方法直接调用。

另一种可能性是，编写所有三个类的主方法去调用`CALL-NEXT-METHOD`，然后在*money-market-account*上特化的方法将使用`CALL-NEXT-METHOD`来调用在*checking-account*上特化的方法。当后者再调用`CALL-NEXT-METHOD`时，它将会运行*savings-account*上的方法，因为根据*money-market-account*的类优先级列表，它将是下一个最相关的方法。

^① 换句话说，Common Lisp不会遇到像C++那样的宝石继承（diamond inheritance）问题。在C++中，当一个类子类了两个同时从公共基类继承了一个成员变量的类时，底下的类继承了成员变量两次，这导致了大量的混乱。

当然，如果你打算依赖一种编码约定（即所有方法都调用`CALL-NEXT-METHOD`）来确保所有可应用的方法都能在某一点处运行，那么应该考虑使用附加方法。在这种情况下，除了为`checking-account`和`savings-account`定义`print-statement`之上的主方法，还可以将这些方法定义成`:after`方法，同时在`bank-account`上定义单一的主方法。然后，调用在`money-market-account`上的`print-statement`将首先打印出由在`bank-account`上特化的主方法输出的基本账户对账单，接着是由在`savings-account`和`checking-account`上特化的`:after`方法输出的细节。如果你想要添加特定于`money-market-account`的细节，那么可以定义一个在`money-market-account`上特化的`:after`方法，它将在最后运行。

使用附加方法的优点在于，它使得哪个方法对于实现广义函数负主要责任，哪个方法只贡献附加的一点儿功能变得非常清楚了。其缺点在于，你无法良好地控制这些附加方法的运行顺序。如果你想要对账单中的`checking-account`部分在`savings-account`部分之前打印，那么就必须改变`money-market-account`，子类化这些类的顺序。但这种改变相当大，可能会影响其他方法和继承的槽。一般而言，如果你发现自己纠结于把直接基类列表的顺序作为调节特定方法行为的手段，那么你可能需要重新考虑你的设计。

另一方面，如果你并不关心具体的顺序，而只想让它在多个广义函数中是一致的，那么使用附加方法可能刚好合适。例如，如果在`print-statement`之外还有一个`print-detailed-statement`广义函数，你可以将这两个函数都实现为`bank-account`的不同子类上的`:after`方法，这样它们中正规和细化的对账单部分的顺序将是相同的。

17.9 好的面向对象设计

以上就是Common Lisp对象系统的主要特性。如果你有丰富的面向对象编程经验，那么就能看到Common Lisp的特性是怎样来实现好的面向对象设计的。不过，如果你缺乏面向对象经验，那么就可能需要花费一些时间来吸取面向对象的思想。遗憾的是，这个主题相当大，已经超出了本书的讨论范围。或者正如Perl的对象系统手册页中所写：“现在你只需买一本面向对象设计方法学的书，并在接下来的六个月里埋头苦读就好了。”或者你可以阅读本书后面的一些实用章节，在那里你将看到这些特性在实践中的使用方法的一些示例。不过，目前你需要暂且放下所有这些面向对象理论，转而学习另一个相当不同的主题：如何更好地使用Common Lisp强大但有时晦涩难懂的`FORMAT`函数。

一些FORMAT秘诀

18

Common Lisp的**FORMAT**函数和扩展的**LOOP**宏，是Common Lisp在许多用户中引起强烈反响的两个特性。对于**FORMAT**函数，有些人喜欢它，而另一些人讨厌它。^①

FORMAT的爱好者们因为它的强大威力和简洁而喜欢它，它的反对者们则由于其潜在的误用和不透明性而讨厌它。复杂的**FORMAT**控制字符串有时就像是一行乱码，但**FORMAT**仍然受到一些Common Lisp程序员们的欢迎，他们希望能够生成少许人类可读的输出而无需手工编写大量的输出生成代码。尽管**FORMAT**的控制字符串可能是晦涩难懂的，但至少单一的**FORMAT**表达式还不致使事情变得太糟。例如，假设你想要将一个列表中的值以逗号分隔打印出来，可以写成下面这样：

```
(loop for cons on list
      do (format t "~a" (car cons))
      when (cdr cons) do (format t ", "))
```

这还不算太糟，但任何读到这些代码的人不得不在大脑里解析它，然后发现它所做的无非是向标准输出打印list的内容。另一方面，你可以立即说出下面的表达式正在以某种形式向标准输出打印list：

```
(format t "~{~a~^, ~}" list)
```

如果你关心该输出的具体形式，那么需要仔细分析控制字符串；但如果你只是想要第一时间估计出这段代码的用途，那么这是立即可以做到的。

不管怎么说，你应当至少可以读懂**FORMAT**，并且在你加入支持或反对**FORMAT**的阵营之前，有必要先知道它究竟能干什么。理解**FORMAT**的基础也是重要的，因为其他标准函数，诸如下一章将讨论的用来抛出各种状况的函数，都使用**FORMAT**风格的控制字符串来生成输出。

进一步说，**FORMAT**支持三种相当不同类型的格式化：打印表中的数据，美化输出S-表达式，以及使用插入的值生成人类可读的消息。现在将表格中的数据作为文本打印已经有些过时了，它是Lisp几乎和FORTRAN一样老的象征之一。事实上，一些可以用来在定长字段中打印浮点值的

^①当然，多数人认识到不值得在一门编程语言里将它实现出来，并且可以没有障碍地使用或不使用它。另一方面，有趣的是Common Lisp所实现的这两种特性，本质上是使用了不基于S-表达式语法的领域相关语法。**FORMAT**控制字符串的语法是基于字符的，而扩展的**LOOP**宏采用了由**LOOP**关键字所描述的语法。对于**FORMAT**和**LOOP**“不够Lisp化”这一常见批评恰恰反映了Lisp程序员们真的很喜欢S-表达式语法。

指令相当直接地来源于FORTRAN的编辑描述符，它们在FORTRAN中用来读取和打印组织成定长字段的数据列。不过，将Common Lisp作为FORTRAN的替代品来使用超出了本书的范围，因此我不会讨论FORMAT的这些方面。

美化输出同样超出了本书的范围——并不是因为它们过时，而只是因为这是一个太大的主题。简单地说，Common Lisp精美打印机是一个可定制的系统，用来打印包括但不限于S-表达式的块结构数据，其中需要变长的缩进和动态添加的断行。它在需要的时候是非常有用的东西，但在日常编程中并不常用。^①

因此，我将聚焦在FORMAT中可以使用插入的值来生成人类可读字符串的那部分内容。即便以这种方式限定范围，仍然谈及大量内容。你不必要求自己记住本章中所描述的每一个细节。只使用少量FORMAT用法通常就够了。我将首先描述FORMAT最重要的特性，究竟对它理解到何种程度完全取决于你自己。

18.1 FORMAT 函数

如同你在前面章节里看到的，FORMAT函数接受两个必要的参数：一个是用于输出的目的地，另一个是含有字面文本和嵌入指令的控制字符串。任何附加的参数都提供了用于控制字符串中指令并插入到输出中的值。我把这些参数称为格式化参数（format argument）。

FORMAT的第一个参数，用于输出的目的地，它可以是T、NIL、一个流或一个带有填充指针的字符串。T是流*STANDARD-OUTPUT*的简称，而NIL会导致FORMAT将输出生成到一个字符串中并随后返回。^②如果目的地是一个流，那么输出将写到该流中。而如果目的地是一个带有填充指针的字符串，那么格式化的输出将被追加到字符串的结尾，并且填充指针也会作适当调整。除了FORMAT在目的地是NIL时返回一个字符串以外，其他情况下FORMAT均返回NIL。

第二个参数，控制字符串，在本质上是一段用FORMAT语言写成的程序。FORMAT语言完全不是Lisp风格的——其基本语法是基于字符而不是S-表达式的，并且它是为简洁性而非易于理解而优化的。这就是为什么一个复杂的FORMAT控制字符串可以最终看起来像是一行乱码。

多数FORMAT指令简单地以一种或另一种形式将参数插入到输出中。某些指令，诸如~%，可以导致FORMAT产生一个换行而不会使用任何参数。而其他的指令，如同你将要看到的，可以使用超过一个参数。有个指令甚至允许你在参数列表中跳动从而多次处理同一个参数，或是在特定情况下跳过特定参数。在讨论特定指令之前，我们首先了解一下指令的一般语法。

^① 对于精美打印机感兴趣的读者可以阅读Richard Waters的论文“XP: A Common Lisp Pretty Printing System”。它是一个对后来合并到Common Lisp中的精美打印机的描述。你可以从<ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-1102a.pdf>下载它。

^② 稍有混淆的一点是，许多其他的I/O函数也接受T和NIL作为流标识符，但带有不同的语义：作为流标识符，T代表双向流*TERMINAL-IO*，而NIL在作为输出流时代表*STANDARD-OUTPUT*，作为输入流时代表*STANDARD-INPUT*。



18.2 FORMAT 指令

所有的指令都以一个波浪线 (~) 开始并终止于单个字符。字符可以使用大写或小写来书写。某些指令带有前置参数 (prefix parameter)，它们紧跟在波浪线后面，由逗号分隔，用来控制诸如在打印浮点数时小数点后打印多少位之类的事情。例如~\$指令，它是用来打印浮点值的指令，默认情况下在小数点之后打印两位数字。

```
CL-USER> (format t "~$" pi)
3.14
NIL
```

不过，通过使用前置参数，可以指定它打印出的小数位数，比如五位小数，像下面这样：

```
CL-USER> (format t "~5$" pi)
3.14159
NIL
```

前置参数的值既可以是写成十进制的数字，也可以是字符，字符的书写形式是一个单引号后接想要的字符。前置参数的值还可以通过两种方式从格式化参数中获得：前置参数v导致FORMAT 使用一个格式化参数并将其值用作前置参数，而前置参数#将被求值为剩余的格式化参数的个数。例如：

```
CL-USER> (format t "~v$" 3 pi)
3.142
NIL
CL-USER> (format t "~# $" pi)
3.1
NIL
```

我将在18.7节中给出一些如何使用#参数的实际例子。

你也可以完全省略前置参数。不过，如果你想要指定一个参数但不指定它前面的那个，你必须为每个未指定的参数加上一个逗号。例如~F指令，它是另一个用来打印浮点值的指令，也接受一个参数来控制需要打印的数的十进制位，但这是第二个参数而不是第一个。如果你想要使用~F 将数字打印为五个十进制位，那么可以写成这样：

```
CL-USER> (format t "~,5f" pi)
3.14159
NIL
```

你还可以使用冒号和@修饰符 (modifier) 来调整某些指令的行为，它放在前置参数之后、指令的标识字符之前。这些修饰符可以细微地改变指令的行为。例如，使用冒号修饰符，以十进制输出整数的~D 指令将在输出数字时每三位用逗号分隔，而“@”修饰符可以使~D在数字为正时带上加号。

```
CL-USER> (format t "~d" 1000000)
1000000
NIL
CL-USER> (format t "~:@d" 1000000)
1,000,000
NIL
```

```
CL-USER> (format t "~@d" 1000000)
+1000000
NIL
```

在合理的情况下，可以组合使用冒号和@修饰符来同时得到两种调整。

```
CL-USER> (format t "~:@d" 1000000)
+1,000,000
NIL
```

在两个修改行为不能有意义地组合在一起的那些指令中，同时使用两个修饰符要么是未定义的，要么给出第三个含义。

18.3 基本格式化

现在来看特定指令。我将从几个最常用的指令开始，包括一些前面章节里已经提到的。

最通用的指令是~A，它使用一个任何类型的格式化参数，并将其输出成美化（人类可读）形式。例如，字符串输出成没有引用标记或转义字符的形式，而数字输出成正常的方式。如果你只是想产生一个人能看懂的值，那么这个指令是最合适的。

```
(format nil "The value is: ~a" 10)           → "The value is: 10"
(format nil "The value is: ~a" "foo")         → "The value is: foo"
(format nil "The value is: ~a" (list 1 2 3)) → "The value is: (1 2 3)"
```

一个紧密相关的指令是~S，它同样使用一个任何类型的格式化参数并输出它。不过，~S试图将输出生成为可被READ读回来的形式。这样，字符串将被包围在引用标记中，而符号在必要的时候是包限定的，等等。那些不带有可读表示的对象将被打印成不可读对象语法#<>。使用一个冒号修饰符，~A和~S指令都可以将NIL输出成()而不是NIL。~A和~S指令也都接受最多四个前置参数，它们用于控制是否在值的后面（或者当使用@修饰符时在值的前面）添加占位符，但这些参数只在生成表格数据时才是真正有用的。

其他两个最常用的指令是用来产生换行的~%，以及用来产生新行的~&。两者的区别在于，~%总是产生换行，而~&只在当前没有位于一行开始处时才产生换行。这对于编写松散耦合的函数特别有用，其中每个函数都生成一块输出，然后这些输出需要以不同方式组合在一起。例如，如果一个函数生成了以换行(~%)结尾的输出，而另一个函数生成了一些以新行(~&)开始的输出，那么当依次调用它们的时候就不必担心会产生一个额外的空行。这两个指令都可以接受单个前置参数来指定想要产生的换行的个数。~%指令会简单地输出那些换行符，而~&指令会输出n-1或n个换行，具体取决于它是否在一一行的开始处输出。

不太常用的相关指令是~~，它导致FORMAT产生一个字面波浪线。与~%和~&相同，它可以通过一个数字来参数化地控制产生多少个波浪线。

18.4 字符和整数指令

除通用指令~A和~S外，FORMAT还支持一些指令用来以特定方式输出指定类型的值。这些指

令中最简单的是~C指令，它用来输出字符。它不接受前置参数，但可用冒号和@修饰符进行修改。在不进行修改的情况下，除了它只能工作在字符上以外，其行为和~A没有区别。修改后的版本更加有用。使用冒号修饰符，~:C可以将诸如空格、制表符和换行符这些不可打印的字符按它们的名字输出。当你想要向用户输出关于某些字符的信息时，这个指令是非常有用的。例如，下面的形式

```
(format t "Syntax error. Unexpected character: ~:c" char)
```

可以产生下面的信息

```
Syntax error. Unexpected character: a
```

但也可以像下面这样：

```
Syntax error. Unexpected character: Space
```

使用@修饰符，~@C将按Lisp的字面字符串语法输出字符。

```
CL-USER> (format t "~@c~%" #\a)
#\a
NIL
```

同时使用冒号和@修饰符，~C指令可以打印出额外的信息：如果该字符要求特殊的按键组合，那么在键盘上输入该字符的方式也将打印出来。例如，在Macintosh上，在特定应用中可以通过按下Control键，然后输入@来键入一个空字符（在ASCII或ISO-8859-1和Unicode等ASCII超集中字符编码为0的字符）。在OpenMCL中，如果使用~:@C指令来打印空字符，那么它将告诉你下面的信息：

```
(format nil "~:@c" (code-char 0)) → "^\@ (Control @)"
```

尽管如此，并非所有的Lisp都实现了~C指令的这个方面。而且就算它们实现了，结果也可能不是精确的。例如，如果在SLIME中运行OpenMCL，那么C-@键组合将被Emacs劫持，并调用set-mark-command。^①

那些致力于输出数字的格式化指令构成了另一个重要的分类。尽管你可以使用~A和~S来输出数字，但如果想要更好地控制它们被打印的形式，那么就需要使用特定于字符的指令了。这些数值指令可以分成两个子类别：用来格式化整数值的指令以及用来格式化浮点值的指令。

有五个紧密相关的指令可以格式化整数值：~D、~X、~O、~B和~R。最常用的是~D指令，它以十进制输出整数。

```
(format nil "~d" 1000000) → "1000000"
```

如同我前面提到的，使用冒号修饰符会在输出中添加逗号。

```
(format nil "~:d" 1000000) → "1,000,000"
```

而使用@修饰符，它总是打印一个正负符号。

```
(format nil "~@d" 1000000) → "+1000000"
```

^① ~C指令的这个变体在像Lisp Machine这样的平台上更有意义，其中键击事件是由Lisp字符所表示的。

并且这两个修饰符可以组合使用。

```
(format nil "~:@d" 1000000) → "+1,000,000"
```

第一个前置参数可以指定输出的最小宽度，而第二个参数可以指定一个用作占位符的字符。默认的占位符是空格，而占位符总是插入在数字之前。

```
(format nil "~12d" 1000000) → "      1000000"
(format nil "~12,'0d" 1000000) → "000001000000"
```

这些参数在格式化日期这样的固定宽度格式时是很有用的。

```
(format nil "~4,'0d-~2,'0d-~2,'0d" 2005 6 10) → "2005-06-10"
```

第三和第四个参数是与冒号修饰符配合使用的：第三个参数指定了用作数位组之间分隔符的字符，而第四个参数指定了每组中数位的数量。这些参数默认为逗号和数字3。这样，你可以使用不带参数的~:D指令来将大整数输出成用于美国的标准格式，但也可以使用~,~,4D将逗号改成句点并将分组从3调整到4。

```
(format nil "~:d" 100000000) → "100,000,000"
(format nil "~,,,'4:d" 100000000) → "1.0000.0000"
```

注意，你必须使用逗号来保留未指定的宽度和占位符参数的位置，从而允许它们保持各自的默认值。

除了将数字分别输出成十六进制、八进制和二进制之外，~X、~O和~B指令与~D指令的工作方式相同。

```
(format nil "~x" 1000000) → "f4240"
(format nil "~o" 1000000) → "3641100"
(format nil "~b" 1000000) → "11110100001001000000"
```

最后，~R指令是通用的进制输出指令。它的第一个参数是一个介于2和36（包括2和36）之间的数字，用来指示所使用的进制。其余的参数与~D、~X、~O和~B指令所接受的四个参数一样，并且冒号和@修饰符也以相同的方式修改其行为。当不使用任何前置参数时，~R指令还有一些特殊行为。我将在18.6节里讨论它。

18.5 浮点指令

有四个格式化浮点值的指令：~F、~E、~G和~\$，其中前三个是基于FORTRAN的编辑描述符的指令。它们多数用于以表格形式格式化浮点值，所以我将跳过这些指令的多数细节。尽管如此，你可以使用~F、~E和~\$指令将浮点值插入到文本中。通用浮点指令~G组合了~F和~E指令的特性，从而使得其只在生成表格输出时才真正有意义。

~F指令以十进制格式输出其参数（该参数应当是一个数字^①），并可以控制十进制小数点之后的数位数量。此外，~F指令在数字特别大或特别小时允许使用科学计数法。而~E指令在输出数

^① 技术上讲，如果该参数不是一个实数，那么~F应当像使用~D指令那样来格式化它，而如果该参数根本不是一个数字，其行为应当像~A指令那样，但并非所有实现都很好地遵守了这一约定。

字时总是使用科学计数法。这两个指令都接受一些前置参数，但你需要关注的只有第二个参数，它控制在十进制小数点之后打印的位数。

```
(format nil "~f" pi)    → "3.141592653589793d0"
(format nil "~,4f" pi) → "3.1416"
(format nil "~e" pi)   → "3.141592653589793d+0"
(format nil "~,4e" pi) → "3.1416d+0"
```

18

`~$`指令和`~F`指令相似，但更简单一些。如同其名字所示，它用于输出货币单位。不带有参数时，它基本上等价于`~,2F`。为了修改十进制小数点之后打印的位数，你可以使用第一个参数，而第二个参数用来控制十进制小数点之前所打印的最小位数。

```
(format nil "~$" pi)    → "3.14"
(format nil "~,4$" pi) → "0003.14"
```

`~F`、`~E`和`~$`三个指令都可以通过使用`@`修饰符来使其总是打印一个正负号。^①

18.6 英语指令

用来生成人类可读消息的最有用的一些`FORMAT`指令，是那些用来产生英文文本的指令。这些指令允许将数字输出成英语单词，基于格式化参数的值来输出复数标识，并且为`FORMAT`的输出分段应用大小写转换。

我在18.4节中所讨论的`~R`指令，当不指定输出进制来使用时，它将数字打印成英语单词或罗马数字。当不带前置参数和修饰符使用时，它将数字输出成基数词。

```
(format nil "~r" 1234) → "one thousand two hundred thirty-four"
```

使用冒号修饰符，它将数字输出成序数。

```
(format nil "~:r" 1234) → "one thousand two hundred thirty-fourth"
```

而当使用`@`修饰符时，它将数字输出成罗马数字。同时使用`@`和冒号时，它产生“旧式风格”的罗马数字，其中4和9被写成`III`和`VIII`而不是`IV`和`IX`。

```
(format nil "~@r" 1234) → "MCCXXXIV"
(format nil "~:@r" 1234) → "MCCXXXIIII"
```

对于那些以给定形式无法表示的过大数字，`~R`将回退到与`~D`相同的行为。

为了生成带有正确复数化单词的消息，`FORMAT`提供了`~P`指令。如果某对应的参数不是1，它就简单地输出一个`s`。

```
(format nil "file~p" 1) → "file"
(format nil "file~p" 10) → "files"
(format nil "file~p" 0) → "files"
```

不过，一般情况下你将使用带有冒号修饰符的`~P`，这会使它重新处理前一个格式化参数。

^① 这只是语言标准里所说的。出于一些原因，可能是源自于同一份古老的基础代码，一些Common Lisp实现并没有正确地实现`~F`指令的这个方面。

```
(format nil "~r file~:p" 1) → "one file"
(format nil "~r file~:p" 10) → "ten files"
(format nil "~r file~:p" 0) → "zero files"
```

使用@修饰符与冒号修饰符组合使用，~P将输出y或ies。

```
(format nil "~r famil~:@p" 1) → "one family"
(format nil "~r famil~:@p" 10) → "ten families"
(format nil "~r famil~:@p" 0) → "zero families"
```

很明显，~P不能解决所有复数化问题，并且对于生成其他语言的消息也没有帮助，但对于那些它可以处理的情况是很有用的。而我将很快讨论的~[指令可以提供更灵活的方式来有条件地输出FORMAT中的某些部分。

最后一个用来输出英语文本的指令是~(，它允许你控制输出文本中的大小写。每一个~(都要与一个~)成对使用，由控制字符串中两个标记之间的部分所生成的输出将被全部转化成小写。

```
(format nil "~(~a~)" "FOO") → "foo"
(format nil "~(~@r~)" 124) → "cxxiv"
```

可以使用@符号来修改~(的行为，将一段文本中第一个词的首字母变成大写；使用冒号可以将所有单词首字母大写；而同时使用两个修饰符将使全部文本转化成大写形式。（这里所说的单词，指的是以字母和数字组成的字符序列，各单词间由既非字母又非数字的字符分隔。）

```
(format nil "~(~a~)" "tHe Quick BROWN foX") → "the quick brown fox"
(format nil "~@(~a~)" "tHe Quick BROWN foX") → "The quick brown fox"
(format nil "~:(~a~)" "tHe Quick BROWN foX") → "The Quick Brown Fox"
(format nil "~:@(~a~)" "tHe Quick BROWN foX") → "THE QUICK BROWN FOX"
```

18.7 条件格式化

除了那些插入参数和修改其他输出的指令以外，FORMAT还提供了一些指令用来实现控制字符串之中的简单控制构造。其中之一是你在第9章里曾经用过的条件指令~[，该指令闭合于一个对应的~]，在它们之间是一组由~;所分隔的子句。~[指令的任务是选取一个子句，随后由FORMAT处理。在没有修饰符或参数的情况下，该子句用数值索引选择。~[指令使用一个格式化参数，它应当是一个数字，该指令取出第n个（从0开始的）子句，其中n是该参数的值。

```
(format nil "~[cero~;uno~;dos~]" 0) → "cero"
(format nil "~[cero~;uno~;dos~]" 1) → "uno"
(format nil "~[cero~;uno~;dos~]" 2) → "dos"
```

如果该参数的值大于子句的数量，那么不打印任何东西。

```
(format nil "~[cero~;uno~;dos~]" 3) → ""
```

如果最后一个子句分隔符是~:；而不是~;，那么最后一个子句将作为默认子句提供。

```
(format nil "~[cero~;uno~;dos~:;mucho~]" 3) → "mucho"
(format nil "~[cero~;uno~;dos~:;mucho~]" 100) → "mucho"
```

也可以使用一个前置参数来指定被选择的子句。尽管使用控制字符串中的字面值是没有意义的，但回顾一下作为前置参数的#代表需要处理的剩余参数的个数。这样，你可以定义一个像下面这样的格式字符串：

```
(defparameter *list-etc*
  "~~#[NONE~;~a~;~a and ~a~:;~a, ~a~]~#[~; and ~a~:;, ~a, etc~].")
```

然后像这样使用它：

```
(format nil *list-etc*)          → "NONE."
(format nil *list-etc* 'a)        → "A."
(format nil *list-etc* 'a 'b)     → "A and B."
(format nil *list-etc* 'a 'b 'c)   → "A, B and C."
(format nil *list-etc* 'a 'b 'c 'd) → "A, B, C, etc."
(format nil *list-etc* 'a 'b 'c 'd 'e) → "A, B, C, etc."
```

注意，上述控制字符串实际包含了两个“~~[~]”指令，两个指令都使用了#来选择要使用的子句。第一个指令使用零到两个参数，第二个指令在需要的时候会再使用一个参数。**FORMAT**将默默忽略在处理控制字符串时没有被使用的任何参数。

如果使用冒号修饰符，那么~~[将只含有两个子句。该指令使用单个参数，并在该参数为NIL时处理第一个子句，而在其他情况下处理第二个子句。在第9章里，你曾经使用该~~[变体来生成“通过或失败”消息，像下面这样：

```
(format t "~~:[FAIL~;pass~]" test-result)
```

注意，任何一个子句都可以是空的，但指令中必须含有~;作为分隔。

最后，借助@修饰符，指令~~[可以只带一个子句。该指令使用一个参数，并且当它是非空时可以回过头来再次使用该参数，然后再处理其子句。

```
(format nil "~~@[x = ~a ~]~~@[y = ~a~]" 10 20) → "x = 10 y = 20"
(format nil "~~@[x = ~a ~]~~@[y = ~a~]" 10 nil) → "x = 10 "
(format nil "~~@[x = ~a ~]~~@[y = ~a~]" nil 20) → "y = 20"
(format nil "~~@[x = ~a ~]~~@[y = ~a~]" nil nil) → ""
```

18.8 迭代

另一个你已经见过的**FORMAT**指令是迭代指令~~{。该指令可以让**FORMAT**在列表的元素或者隐式的**FORMAT**参数列表上进行迭代。

不带修饰符时，~~{使用一个格式化参数，它必须是一个列表。和~~[指令必须与一个~~]}指令配对使用一样，~~{指令也必须和一个闭合的~~}成对使用。两个标记间的文本将作为一个控制字符串来处理，它们从~~{指令所使用的列表中取得其参数。只要被迭代的列表尚有元素剩余，**FORMAT**将重复处理该控制字符串。在下面的示例中，~~{使用单个格式化参数，列表(1 2 3)，然后处理控制字符串“~a， ”，重复操作直到该列表的所有元素都已使用。

```
(format nil "~~{~a, ~}" (list 1 2 3)) → "1, 2, 3, "
```

可是在输出中，列表最后一个元素后面跟了一个逗号和一个空格，这显得很让人讨厌。你可

以使用`~^`指令来修复这点。在一个`~{}`指令体内，当列表中没有元素剩余时，`~^`将令迭代立即停止且无须处理其余的控制字符串。这样，为了避免在列表的最后元素之后打印出逗号和空格，你可以在它们前面添加一个`~^`。

```
(format nil "~{~a~^, ~}" (list 1 2 3)) → "1, 2, 3"
```

在迭代过程的前两次里，处理`~^`时，列表中尚有未处理的元素。到了第三次的时候，在`~a`指令处理了3之后，`~^`将令FORMAT跳出迭代而不会打印出逗号和空格。

使用`@`修饰符，`~{}`将把其余的格式化参数作为列表来处理。

```
(format nil "~@{~a~^, ~}" 1 2 3) → "1, 2, 3"
```

在一个`~{...~}`的主体中，特殊前置参数`#`代表列表中需要被处理的剩余项的个数，而不是剩余格式化参数的个数。你可以将`#`和`~[`指令一起使用，来打印用逗号分隔、并在最后一个列表项之前带有`and`的列表，就像下面这样：

```
(format nil "~{~a~#[~, and ~::, ~]~}" (list 1 2 3)) → "1, 2, and 3"
```

不过，当列表是两个元素长度时，由于它添加了一个额外的逗号，上述表达式的输出并不正常。

```
(format nil "~{~a~#[~, and ~::, ~]~}" (list 1 2)) → "1, and 2"
```

你可以通过多种方式来修复这个问题。下面的方法利用了`~@{}`嵌入到另一个`~{}`或`~@{}`指令中时所具有的行为——它迭代由外层`~{}`迭代的列表中的剩余元素。你可以将它与`~#[`指令组合使用，从而使得下面的控制字符串按照英语语法来格式化列表：

```
(defparameter *english-list*
  "~{~#[~,~a~,~a and ~a~::,~@{~a~#[~, and ~::, ~]~}~}~")"

(format nil *english-list* '()) → ""
(format nil *english-list* '(1)) → "1"
(format nil *english-list* '(1 2)) → "1 and 2"
(format nil *english-list* '(1 2 3)) → "1, 2, and 3"
(format nil *english-list* '(1 2 3 4)) → "1, 2, 3, and 4"
```

尽管这个控制字符串已经接近于只能写不能读的程度了，但如果你能花一点时间还是不难理解它的。外层的`~{...~}`将使用并迭代在一个列表上。整个迭代体随后由一个`~#[...~]`构成。这样每次通过迭代所产生的输出将取决于列表中待处理项的个数。通过使用`~;`子句分隔符来分拆`~#[...~]`指令，你可以看到它由四个子句所组成，而且因为它前置了一个`~::`而不是普通的`~,`，所以最后一个默认子句。第一个子句用于当还有零个元素需要处理时，其为空是合理的——如果没有元素需要处理了，那么迭代就该停止了。第二个子句处理只有一个元素的情况，它带有一个简单的`~a`指令。两元素的情况由`"~a and ~a"`所处理。而默认子句用来处理三个或更多元素的情形，它由另一个迭代子句所构成，这一次使用`~@{}`来迭代由外层`~{}`处理的列表的剩余元素。该迭代的主体可以正确处理三个或更多元素列表的控制字符串，在这种情况下是正确的。因为该`~@{}`循环将消耗掉所有剩余的列表元素，而外层循环只迭代一次。

如果你想要在列表为空时，打印出诸如`<empty>`这样的特殊标识，那么有几种方式可以做到这点。最简单的一种可能是把你想要的文本放在外层`~[`的第一个（确切地说是第零个）子句里，并在外层迭代的闭合`]~`上添加一个冒号修饰符——该冒号会强制迭代至少运行一次，就算列表是空的，在这种情况下`FORMAT`将处理条件子句中的第零个子句。

```
(defparameter *english-list*
  "~~{~#[<empty>~;~a~;~a and ~a~:;~@(~a~#[~, and ~:;, ~]~)~]~:}~")"

(format nil *english-list* '()) → "<empty>"
```

令人惊奇的是，`~`指令还通过不同的前置参数和修饰符组合提供了更多的变体。我不会详细讨论它们，简单而言，你可以使用一个整数前置参数来限制迭代的最大数量，以及通过一个冒号修饰符，列表（无论是一个实际列表还是由`@{`指令所构造出的列表）中的每个元素其本身都必须是一个列表，并且后者的元素将用作`~:{...~}`指令中控制字符串的参数。

18.9 跳，跳，跳

一个更简单的指令是`~*`指令，它允许你在格式化参数列表中跳跃。在它的基本形式中，没有修饰符的情况下，它简单地跳过下一个参数，使用它而不输出任何东西。不过更常见的情况是，它和一个冒号修饰符一起使用，这使它可以向前移动，从而允许同一个参数被再次使用。例如，你可以使用`~:*`将一个数值参数按单词打印一次，再按数值打印一次：

```
(format nil "~r ~:*(~d)" 1) → "one (1)"
```

或者，你也可以组合`~:*`与`~[`来实现一个类似于`~:P`的不规则复数形式的指令。

```
(format nil "I saw ~r el~:*[ves~;f~:;ves~]." 0) → "I saw zero elves."
(format nil "I saw ~r el~:*[ves~;f~:;ves~]." 1) → "I saw one elf."
(format nil "I saw ~r el~:*[ves~;f~:;ves~]." 2) → "I saw two elves."
```

在这个控制字符串中，`~R`将格式化参数打印成一个基数。然后`~:*`指令回过头来使该数字再用作`~[`指令的参数，并在该数字是0、1或其他任何值时分别选择不同的子句。^①

在一个`~{`指令中，`~*`可以跳过或恢复列表中的项。例如，可以像下面这样只打印一个plist中的键：

```
(format nil "~(~s~*~^ ~)" '(:a 10 :b 20)) → ":A :B"
```

还可以给`~*`指令一个前置参数。当没有修饰符或使用冒号修饰符时，该参数指定了向前或向后移动的参数个数，默认为1。当使用`@`修饰符时，该前置参数指定了一个用来跳跃的、绝对的、

^① 如果你觉得“`I saw zero elves`”这种说法有点奇怪，那么可以使用一个更加精巧的格式字符串，其使用了`~:*`的另一种用途：

```
(format nil "I saw ~[no~;~:~*~r~] el~:*[ves~;f~:;ves~]." 0) → "I saw no elves."
(format nil "I saw ~[no~;~:~*~r~] el~:*[ves~;f~:;ves~]." 1) → "I saw one elf."
(format nil "I saw ~[no~;~:~*~r~] el~:*[ves~;f~:;ves~]." 2) → "I saw two elves."
```

以零开始的参数索引，默认为0。在你想要使用不同的控制字符串来为同一组参数生成不同的消息，并且不同的消息需要使用不同顺序的这些参数时，~*的@变体可能是有用的。^①

18.10 还有更多……

还有更多的内容。我还没有提到~?指令，它可以从格式化参数中获取控制字符串；还有~/指令，它允许你调用任意函数来处理下一个格式化参数。还有所有用于生成表格和优美打印输出的全部指令。但在本章中所讨论的这些指令对于目前来说应当足够使用了。

在下一章里，你将接触Common Lisp的状况系统，类似于其他语言的异常和错误处理系统。

^① 这类问题可能在试图本地化一个应用程序并将人类可读消息翻译成不同语言时出现。`FORMAT`可以对这些问题中的一些提供帮助，但绝不意味着它是一个全功能的本地化系统。

超越异常处理：状况和再启动

Lisp的状况系统（condition system）是它最伟大的特性之一。它与Java、Python和C++中的异常处理系统有着相同的目标但更加灵活。事实上，它的灵活性扩展到了错误处理之外——“状况”相比“异常”更具一般性，因为状况可以代表程序执行过程中的任何事件，程序调用栈中不同层次的代码都可能对这些事件感兴趣。例如，在19.6节里你将看到，状况可用来输出警告而不会中断产生警告的那些代码的执行，并允许调用栈中更高层的代码来控制是否打印警告信息。不过，一开始我还是集中讨论错误处理。

状况系统相比异常系统的更灵活之处在于，它没有明确划分产生错误^①的代码和处理错误^②的代码，而是将责任分拆成三个部分：产生（signaling）状况，处理（handling）它以及再启动（restarting）。在本章里，我将描述你在假想的分析日志文件应用程序中如何使用状况系统。你将看到如何使用状况系统，允许底层函数在解析日志文件时检测问题并产生一个错误，允许中层代码提供几种可能的方式从这样一个错误中恢复，以及允许该应用程序的最上层代码制定一种方针来选择所使用的恢复策略。

一开始，我将介绍一些术语。错误（error），我采用墨菲法则定义该术语。如果某件事可能出错，那么它终将出错：一个程序需要读取的文件会丢失，一个需要写入的磁盘会满，正在连接的服务器会崩溃，或者网络会断开。如果发生了任何这些情况，它可能使一些正在做你所想的事情的代码停止工作。但这里没有bug，代码中没有可以修复的地方，以使那些不存在的文件存在或是令磁盘不再是满的。尽管如此，如果程序的其余部分正在依赖于这些正打算进行的操作，那么你最好以某种方式处理这些错误，否则就会引入bug。因此，错误并不是由bug导致的，但忽视处理错误就总是可能作一个bug。

那么，处理错误究竟意味着什么呢？在一个编写良好的程序中，每个函数都是一个隐藏了其内部工作的黑箱。程序随后由分层的函数构建而成：高层次的函数是构建在相对低层次的函数之上的，诸如此类。功能的层次关系在运行期以调用栈的形式显示其自身：如果high调用了medium，medium调用了low，那么当控制流在low中时，它也仍然在medium和high中，也就是

① 在Java/Python的术语中称为抛出（throw）或提升（raise）一个异常。

② 在Java/Python的术语中称为捕捉（catch）该异常。

说它们仍然在调用栈中。

由于每个函数都是一个黑箱，函数边界刚好是处理错误的最佳场所。每个函数（比如说`low`）都有一个需要完成的任务。其直接调用者（这里是`medium`）的工作是统计它的调用次数。不过，一个使它不能正常工作的错误将给它的所有调用者带来风险：`medium`调用`low`是因为它需要的工作`low`能够完成，如果`low`不能完成该工作，那么`medium`就有问题了。这意味着`medium`的调用者`high`也有问题了，诸如此类，一直沿着调用栈到达程序的最顶端。另一方面，由于每个函数都是一个黑箱，如果调用栈中的任何函数可能以某种方式完成其工作而无需关注底层错误，那么在它之上的程序就没有必要知道曾经出现的问题——那些函数所关心的只是它们所调用的函数无论如何都要完成期待的工作。

在多数语言里，错误的处理方式都是从一个失败的函数返回并给调用者一个机会，要么修复该错误要么让其自身失败。某些语言使用了正常的函数返回机制，而带有异常的语言可以通过抛出一个异常来返回控制。使用异常比使用正常函数返回是巨大的进步，但两种模式有一个共同的缺点：在搜索一个可恢复的函数时，栈被展开了，这意味着可以恢复错误的代码无法在错误实际发生时底层代码所在的上下文中进行操作。

想想函数`high`、`medium`和`low`的假想调用链。如果`low`失败而`medium`无法恢复，那么决定权将在`high`手中。在`high`处理错误时，它必须要么在无须得到任何来自`medium`的帮助的情况下完成其工作，要么以某种方式改变一些，使对`medium`的调用能够发挥作用，然后再次调用它。前一个选项在理论上是清晰的，但可能导致大量的额外代码，那将需要一个完整的对于`medium`的工作的额外实现。并且当栈进一步展开时，还有更多工作需要重做。后一个选项（修补环境并重试）比较棘手，这要求`high`必须改变当前环境的状况，使得下一个对`medium`的调用不会导致`low`中的错误，这需要同时对`medium`和`low`的内部工作原理有所了解，这并不恰当，与每个函数都是一个黑箱的理念相违背。

19.1 Lisp 的处理方式

Common Lisp的错误处理系统提供了一种跳出这一难题的方式，它将实际从错误中恢复的代码和决定如何恢复的代码分开。这样，你可以将恢复代码放在底层函数中而无须决定实际使用何种特定的恢复策略，将决策留给高层函数中的代码。

为了了解其工作原理，我们假设你正在编写一个读取某种文本日志文件的应用程序，例如读取一个Web服务器的日志。在应用程序的某处，你将有一个函数用来解析单独的日志项。我们假设你将编写一个函数`parse-log-entry`，它将接受一个含有单个日志项文本的字符串，并假设可以返回代表该项的一个`log-entry`对象。该函数将从函数`parse-log-file`中调用，后者读取一个完整的日志文件并返回代表该文件中所有日志项的对象列表。

为了保持简单性，`parse-log-entry`函数不需要解析不正确的格式化项。不过，它可以检测到有问题的输入。当它检测到不对的输入后应当做什么呢？在C中会返回一个特殊值来指示这里出了问题，在Java或Python中会抛出一个异常。而在Common Lisp中会产生一个状况。

19.2 状况

一个状况 (condition) 是一个对象，它所属的类代表了该状况的一般性质，它的实例数据则带有导致该状况产生的特定情形细节的信息。^①在这个假想的日志分析程序里，你可以定义一个状况类malformed-log-entry-error，函数parse-log-entry将在给定数据无法解析时产生该状况。

状况类使用**DEFINE-CONDITION**宏来定义，它本质上就是**DEFCLASS**，除了使用**DEFINE-CONDITION**所定义的类的默认基类是**CONDITION**而非**STANDARD-OBJECT**。槽以相同的方式来指定，状况类可以单一和多重地继承自其他源自**CONDITION**的类。但出于历史原因，状况类不要求是**STANDARD-OBJECT**的实例，因此一些你和**DEFCLASS**所定义类一起使用的函数不要求可以工作在状况对象上。特别地，一个状况的槽不能使用**SLOT-VALUE**来访问，必须为任何你打算使用其值的槽分别指定:**:reader**选项或:**:accessor**选项。同样，新的状况对象使用**MAKE-CONDITION**而非**MAKE-INSTANCE**来创建。**MAKE-CONDITION**基于其被传递的:**:initargs**来初始化新状况的槽，但没有办法以等价于**INITIALIZE-INSTANCE**的方式更进一步地定制一个状况的初始化过程。^②

当把状况系统用于错误处理时，你应当将状况定义成**ERROR**的子类，后者是**CONDITION**的一个子类。这样，你可以像下面这样定义malformed-log-entry-error，其中带有一个槽用来保存传递给parse-log-entry的参数：

```
(define-condition malformed-log-entry-error (error)
  ((text :initarg :text :reader text)))
```

19.3 状况处理器

在parse-log-entry中，当无法解析日志项时将产生一个malformed-log-entry-error。函数**ERROR**用来报错，它将调用底层函数**SIGNAL**并在状况未处理时进入调试器。有两种方式来调用**ERROR**：传给它一个已经实例化的状况对象，或者传给它该状况类的名字以及需要用来构造新状况的初始化参数，然后它将实例化该状况。前者对于重新产生一个已有的状况对象偶尔会有用，而后者更普遍。因此，你可以像下面这样编写parse-log-entry，其中隐藏了实际解析日志项的细节：

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (error 'malformed-log-entry-error :text text)))
```

^① 在这个意义上，除了并非所有状况都代表一个错误或异常的 (exceptional) 情形，一个状况和Java或Python中的异常很像。

^② 在一些Common Lisp实现中，状况被定义为**STANDARD-OBJECT**的子类。在这种情况下，**SLOT-VALUE**、**MAKE-INSTANCE**和**INITIALIZE-INSTANCE**将可以工作，但依赖于它们将是不可移植的。

当产生错误时，实际发生的事情取决于调用栈中parse-log-entry之上的代码。为了避免进入调试器，必须在导致调用parse-log-entry的某个函数中建立一个状况处理器（condition handler）。当产生状况时，信号机制会查看活跃状况处理器的列表，并基于状况的类来寻找可以处理所产生的状况的处理器。每个状况处理器由一个代表它所能处理的状况类型的类型说明符和一个接受单个状况参数的函数所构成。在任何给定时刻可能有多个活跃的状况处理器建立在调用栈的不同层次上。当一个状况产生时，信号机制会查找最新建立的类型说明符与当前所产生状况相兼容的处理器并调用它的函数，同时传递状况对象给该函数。

处理器函数随后可以选择是否处理该状况。该函数可以通过简单的正常返回来放弃处理该状况。在这种情况下，控制将返回到SIGNAL函数，SIGNAL函数随后继续搜索下一个带有兼容类型说明符的最新建立的处理器。为了处理该状况，该函数必须通过一个非本地退出（nonlocal exit）将控制传递到SIGNAL之外。在下一节里，你将看到一个处理器是怎样选择传递控制的位置的。尽管如此，许多状况处理器简单地想要将栈展开到它们被建立的位置上并随后运行一些代码。宏HANDLER-CASE可以建立这种类型的状况处理器。一个HANDLER-CASE的基本形式如下所示：

```
(handler-case expression
  error-clause*)
```

其中每一个error-clause均为下列形式：

```
(condition-type ([var]) code)
```

如果expression正常返回，那么其值将被HANDLER-CASE返回。HANDLER-CASE的主体必须是单一表达式，你可以使用PROGN将几个表达式组合成单一形式。不过，如果该表达式产生了一个状况，并且其实例属于任何错误子句中指定的状况类型之一，那么将执行相应错误子句中的代码，且其值将由HANDLER-CASE返回。如果形参var被包含，在执行处理器代码时，它将成为保存状况对象的变量名。如果代码不需要访问状况对象，则可以省略这个变量名。

例如，一种在parse-log-entry的调用者parse-log-file中处理前者所产生的malformed-log-entry-error的方式，将是跳过有问题的项。在下面的函数中，HANDLER-CASE表达式要么返回由parse-log-entry返回的值，要么在产生malformed-log-entry-error时返回NIL。（LOOP子句collect it中的it是另一个LOOP关键字，它指向最新求值条件测试的值，在这种情况下是entry的值。）

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
          for entry = (handler-case (parse-log-entry text)
                        (malformed-log-entry-error () nil))
          when entry collect it)))
```

当parse-log-entry正常返回时，它的值将赋值给变量entry中并被LOOP收集。但如果parse-log-entry产生了一个malformed-log-entry-error，那么错误子句将返回NIL，从而不会被收集。

JAVA风格的异常处理

HANDLER-CASE是Common Lisp中最接近Java或Python风格的异常处理。在Java中可能写成这样：

```
try {
    doStuff();
    doMoreStuff();
} catch (SomeException se) {
    recover(se);
}
```

或在Python中写成这样：

```
try:
    doStuff()
    doMoreStuff()
except SomeException, se:
    recover(se)
```

而在Common Lisp中需要写成这样：

```
(handler-case
  (progn
    (do-stuff)
    (do-more-stuff))
  (some-exception (se) (recover se)))
```

这个版本的parse-log-file有一个严重的缺陷：它做得太多了。顾名思义，parse-log-file的任务是解析文件并产生log-entry对象的列表。如果它做不到这一点，决定采用何种替代方案就不是它的职责所在。假如你想在一个想要告诉用户日志文件被破坏了的应用中使用parse-log-file，或是想要从有问题的项中通过修复并重新解析它们来恢复，又该怎样做呢？或者一个应用对于跳过它们是没问题的，但只有当发现特定数量的受损日志项后才需要特别处理。

你可以通过将**HANDLER-CASE**移动到一个更高层的函数中来修复该问题。不过，这样你就没有办法实现当前跳过个别项的策略了——当错误发生时，栈将被一路展开到更高层的函数上，连同日志文件的解析本身一并丢弃了。你所需要的是一种方式，能提供当前的恢复策略而不是总要被用到。

19.4 再启动

状况系统可以让你将错误处理代码拆分成两部分。将那些实际从错误中恢复的代码放在再启动(restart)中，状况处理器随后会通过调用一个适当的再启动来处理状况。可以将再启动代码放在中层或底层的函数里，例如parse-log-file或parse-log-entry，而将状况处理器移到应用程序的更上层。



为了改变parse-log-entry，从而使其建立一个再启动而非状况处理器，可以将HANDLER-CASE改成RESTART-CASE。除了再启动的名字可以只是普通的名字而无需是状况类型的名字外，RESTART-CASE的形式与HANDLER-CASE很相似。一般而言，一个再启动名应当描述了再启动所产生的行为。在parse-log-file中，你可以根据其行为将这个再启动称为skip-log-entry。该函数的新版本如下所示：

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
          for entry = (restart-case (parse-log-entry text)
                               (skip-log-entry () nil))
          when entry collect it)))
```

如果在一个含有受损日志项的日志文件上调用该版本的parse-log-file，它将不会直接处理错误，最终会转到调试器中。不过，在调试器所提供的几个再启动选项中会有一个称为skip-log-entry的选项，如果你选择了它，将导致parse-log-file继续其之前的操作。为了避免进入调试器，你可以建立一个状况处理器来自动地调用skip-log-entry再启动。

建立一个再启动而不是让parse-log-file直接处理错误的好处在于，它使得parse-log-file可以用在更多情形里了。调用parse-log-file的更高层代码不需要调用skip-log-entry再启动。它可以选择在更高层进行错误处理。或者，如同我将在下一节讲述的那样，你可以为parse-log-entry添加再启动来提供其他的恢复策略，随后状况处理器可以选择它们想要使用的策略。

在我开始谈论这些之前，你需要知道如何设置一个会调用skip-log-entry再启动的状况处理器。你可以在通向parse-log-file的函数调用链的任何位置上设置这个处理器。这可能是应用程序中很上层的某个位置，不一定在parse-log-file的直接调用者中。例如，假设应用程序的主入口点是函数log-analyzer，它查找一组日志并使用函数analyze-log来分析它们，后者最终导致了对parse-log-file的调用。在没有任何错误处理的情况下，它可能像这样：

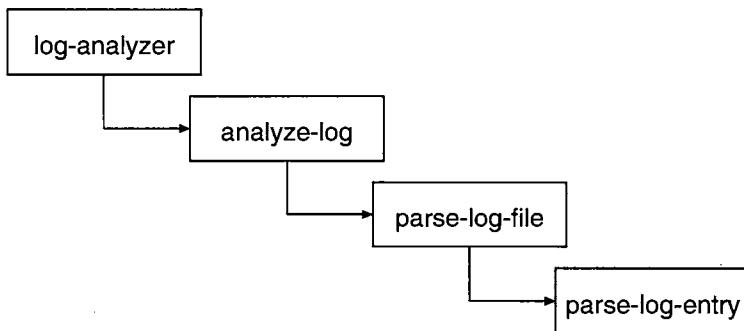
```
(defun log-analyzer ()
  (dolist (log (find-all-logs))
    (analyze-log log)))
```

analyze-log的任务是直接或间接地调用parse-log-file，然后再对返回的日志项列表做一些事情。它的一个极其简化的版本可能像这样：

```
(defun analyze-log (log)
  (dolist (entry (parse-log-file log))
    (analyze-entry entry)))
```

根据推测，其中函数analyze-entry将会实际取出那些你想要从每个日志项中得到的信息，并存储在其他地方。

这样，从最上层的函数log-analyzer到实际产生错误的parse-log-entry的路径将如下所示。



假设你总是想要跳过有问题的日志项，那么可以改变这个函数来建立一个状况处理器，为你调用`skip-log-entry`。不过，你不能使用`HANDLER-CASE`来建立这个状况处理器，因为那样的话栈会回退到`HANDLER-CASE`所在的函数里。你需要使用更底层的宏`HANDLER-BIND`。`HANDLER-BIND`的基本形式如下所示：

```
(handler-bind (binding*) form*)
```

其中的每个绑定都是由状况类型和一个单参数处理函数所组成的列表。在处理器绑定之后，`HANDLER-BIND`的主体可以包含任意数量的形式。与`HANDLER-CASE`的处理器代码有所不同的是，这里的处理器代码必须是一个函数对象，并且它必须只接受单一参数。`HANDLER-BIND`和`HANDLER-CASE`之间的一个更大的区别在于，由`HANDLER-BIND`所绑定的处理器函数必须在不回退栈的情况下运行——当调用该函数时，控制流仍然在对`parse-log-entry`的调用中。对`INVOKE-RESTART`的调用将查找并调用最近绑定的带有给定名字的再启动。因此你可以像下面这样给`log-analyzer`添加一个处理器，使其可以调用由`parse-log-file`所建立的再启动：^①

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (invoke-restart 'skip-log-entry)))
                (dolist (log (find-all-logs))
                  (analyze-log log))))
```

在这个`HANDLER-BIND`中，处理器函数是一个调用了`skip-log-entry`再启动的匿名函数。你也可以定义一个命名函数来做相同的事并绑定该函数。事实上，在定义再启动时有个常用的实践技巧是定义一个函数，它与再启动具有相同的名字并接受单一状况参数来调用对应的再启动。这样的函数称为再启动函数（restart function）。你可以像下面这样为`skip-log-entry`定义一个再启动函数：

```
(defun skip-log-entry (c)
  (invoke-restart 'skip-log-entry))
```

然后将`log-analyzer`的定义修改成下面这样：

^① 编译器可能会抱怨函数的参数从未被使用。你可以通过添加一个声明(`declare (ignore c)`)作为`LAMBDA`形体体中的第一个表达式来消除这类警告。

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error #'skip-log-entry))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

如同代码中所写，`skip-log-entry`再启动函数假设已经建立了`skip-log-entry`再启动。如果`malformed-log-entry-error`被来自`log-analyzer`的代码抛出，却没有建立`skip-log-entry`再启动，那么对`INVOKE-RESTART`的调用将在无法找到`skip-log-entry`再启动时产生`CONTROL-ERROR`报错。如果你想要允许`malformed-log-entry-error`可以被没有建立`skip-log-entry`再启动的代码抛出，那么可以将`skip-log-entry`函数修改成下面这样：

```
(defun skip-log-entry (c)
  (let ((restart (find-restart 'skip-log-entry)))
    (when restart (invoke-restart restart))))
```

`FIND-RESTART`查找一个给定名字的再启动，并在找到时返回一个代表该再启动的对象，否则返回`NIL`。你可以通过将再启动对象传递给`INVOKE-RESTART`来调用该再启动。这样，当`skip-log-entry`被`HANDLER-BIND`绑定时，它将在存在一个再启动时通过调用`skip-log-entry`再启动来处理该状况，而在其他情况下正常返回，从而给那些绑定在栈的更高层的其他状况处理器提供机会来处理该状况。

19.5 提供多个再启动

由于再启动必须显式调用才有效果，因此你可以定义多个再启动，让它们中的每一个分别提供不同的恢复策略。如同我早先提到的，并非所有的日志解析应用都需要跳过那些有问题的项。一些应用可能想要`parse-log-file`包含一个特殊类型的对象来表示`log-entry`对象列表中有问题的日志项；其他应用可能有一些方式来修复有问题的项，并可能需要一种方式来将修复后的项传递回`parse-log-entry`。

为了允许这些更复杂的恢复机制，再启动可以接受任意参数，它们被传递给对`INVOKE-RESTART`的调用。通过为`parse-log-entry`增加两个再启动，其中每个都只接受单个参数，你可以同时支持我刚刚提到的两种恢复策略。一个简单地返回传递给它的`parse-log-entry`的返回值，而另一个则试图在最初的日志项上就地解析其参数。

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (restart-case (error 'malformed-log-entry-error :text text)
      (use-value (value) value)
      (reparse-entry (fixed-text) (parse-log-entry fixed-text)))))
```

`USE-VALUE`是这类再启动的标准名字。Common Lisp为`USE-VALUE`定义了一个再启动函数，类似于你之前为`skip-log-entry`定义的再启动函数。因此，如果你想要改变处理有问题项的策略，在遇到这类问题项时创建一个`malformed-log-entry`的实例，那么就可以像下面这样修改`log-analyzer`（假设存在一个带有`:text`初始化参数的`malformed-log-entry`类）：

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (use-value
                        (make-instance 'malformed-log-entry :text (text c))))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

你还可以将这些新的再启动放在`parse-log-file`而不是`parse-log-entry`中。不过，你通常会想把再启动放在尽可能最底层的代码里。尽管如此，将`skip-log-entry`再启动移到`parse-log-entry`中并不合适，因为这会导致`parse-log-entry`有时正常返回一个`NIL`，而这是你极力予以避免的。同时，基于状况处理器可以通过在`NIL`上调用`use-value`再启动而获得同样效果这一理论，直接去掉`skip-log-entry`再启动也不是个好主意，因为这要求状况处理器理解`parse-log-file`的工作原理。正如其名字所显示的，`skip-log-entry`是一个正确抽象了的日志解析API的组成部分。

19.6 状况的其他用法

虽然状况系统主要用于错误处理，但它们还可以用于其他目的，你可以使用状况、状况处理器和再启动在底层和上层代码之间构建多种协议。理解状况的潜在用途的关键在于，要理解仅仅抛出一个状况并不会改变程序的控制流。

基本的信号函数`SIGNAL`实现了搜索适用的状况处理器并调用其处理器函数的机制。一个处理器通过正常返回来拒绝处理状况的原因在于，对处理器函数的调用只是一个正规函数调用——当处理器返回时，控制被传递回`SIGNAL`，`SIGNAL`随后继续查询另一个较近绑定的可以处理该状况的处理器。如果`SIGNAL`在状况处理之前找不到其他的状况处理器，那么它也会正常返回。

你曾经使用的`ERROR`函数会调用`SIGNAL`。如果错误被一个通过`HANDLER-CASE`传递控制的状况处理器或通过调用一个再启动所处理，那么这个对`SIGNAL`的调用将不再返回。但如果`SIGNAL`返回了，那么`ERROR`将通过调用保存在`*DEBUGGER-HOOK*`中的函数来启动调试器。这样，一个对`ERROR`的调用永远不会正常返回。状况必须要么被一个状况处理器所处理，要么在调试器中被处理。

另一个状况信号函数`WARN`提供了构建在状况系统之上的不同类型协议的示例。和`ERROR`一样，`WARN`调用`SIGNAL`来产生一个状况。但是如果`SIGNAL`返回了，`WARN`并不会调用调试器——它将状况打印到`*ERROR-OUTPUT*`中并返回`NIL`，从而交给它的调用者来处理。`WARN`也会在`SIGNAL`调用的外围建立一个再启动`MUFFLE-WARNING`，从而允许一个状况处理器令`WARN`直接返回而不打印任何东西。再启动函数`MUFFLE-WARNING`可以查找并调用与其同名的再启动，并在不存在这样的再启动时产生一个`CONTROL-ERROR`。当然，一个通过`WARN`产生的状况也可以用其他方式来处理——一个状况处理器可以像处理真正的错误一样来处理它，从而将一个警告“提升”为一个错误。

举个例子，在日志分析应用里，如果存在某些情况使得一个日志项稍不正常但仍可解析，那

么你可以编写parse-log-entry来使其继续解析这些稍有问题的日志项，但同时用**WARN**产生一个状况。然后更大的应用可以选择打印这些警告、隐藏这些警告，或是将这些警告当作错误来处理，并使用与来自malformed-log-entry-error相同的方式进行恢复。

第三个报错函数**CERROR**提供了另一种协议。和**ERROR**一样，**CERROR**将在状况没被处理时就会转到调试器。但和**WARN**一样，它在产生状况之前会建立一个再启动。这个再启动是**CONTINUE**，它可以使**CERROR**正常返回——如果再启动由一个状况处理器所调用的话，它将确保始终不转到调试器。否则，可以在转到调试器以后使用再启动，立即恢复到**CERROR**调用之后的计算状态。函数**CONTINUE**查找并在**CONTINUE**再启动可用时调用它，否则返回**NIL**。

你也可以在**SIGNAL**之上构建自己的协议——无论底层代码需要何种方式来与调用栈中的上层代码沟通信息，状况机制都可以合理使用。但对于多数目标来说，标准的错误和警告协议应该足够了。

你将在后续的实践章节里用到状况系统，既可以用于正常的错误处理，也可以像第25章那样帮助处理ID3文件解析过程中一些棘手的边界情况。遗憾的是，编程教材总是过于轻视错误处理。正确的错误处理，或者在这方面的欠缺，往往是阐述性代码和坚不可摧的产品级代码之间最大的区别。后者的难点在于，需要进行大量的关于软件本身而不是任何特定编程语言构造细节的思考。这就是说，如果你的目标是编写一个那样的软件，那么你将发现Common Lisp状况系统是用于编写健壮代码的极佳工具，并且它可以完美地融合到Common Lisp的增量式开发风格中。

编写健壮的软件

关于编写健壮的软件方面的信息，你可以从查阅由Glenford J.Meyers编写的*Software Reliability* (John Wiley & Sons, 1976) 开始。Bertrand Meyer关于Design By Contract的著作也提供了一种思考软件正确性的有用方式，可参见他的*Object-Oriented Software Construction* (Prentice Hall, 1997) 一书的第11章和第12章。不过要记住，Bertrand Meyer是Eiffel的发明者，Eiffel是一种静态类型的受约束的Algol/Ada系语言。尽管他在面向对象和软件可靠性方面有许多聪明的见解，但在他的编程观点和Lisp编程方式之间仍然存在着一条鸿沟。最后，关于围绕构建失效容忍系统的更大问题的一个绝佳综述，可以参见Jim Gray和Andreas Reuter所编写的经典的*Transaction Processing: Concepts and Techniques* (Morgan Kaufmann, 1993) 一书的第3章。

在下一章里，我将简单概述一下你尚未有机会使用或者说至少还没有直接用到的25个特殊操作符。

特殊操作符

20

20

从某种意义上说，对于前一章里所描述的状况系统，其最令人印象深刻的方面在于，如果它还不是语言的一部分，那么它完全可以被实现成用户层面的库。这种可能性体现在，尽管没有哪个Common Lisp的特殊操作符是直接用于产生或处理状况的，但它们提供了通向语言底层机制的足够的权限，从而能够做到诸如控制栈的回退这样的事情。

我在前面的章节里已经讨论了大多数常用的特殊操作符，但有两个理由使我们有必要熟悉其他的特殊操作符。首先，一些不太常用的特殊操作符之所以不太常用，只是因为需要用它们来处理的情况也很少发生。你有必要熟悉这些特殊操作符，这样当有一天需要它们时，你至少可以想到它们。其次，因为25个特殊操作符（连同求值函数调用的基本规则以及内置数据类型）提供了语言其余部分的基础，因此对它们有整体的了解将有助于你理解该语言的工作方式。

在本章里，我将讨论所有的特殊操作符，其中的一些只是简要介绍，另一些则会详细叙述，你会看到它们是如何在一起工作的。我将指出它们中的哪些是可以直接用在你自己的代码中的，哪些是用来作为你一直使用的其他控制构造的基础的，以及哪些是你将很少直接使用但在宏生成的代码中却是相当有用的。

20.1 控制求值

第一类特殊操作符包括三个对形式求值提供基本控制的操作符，它们是QUOTE、IF和PROGN，我们已经全部讨论过了。尽管如此，还是该注意这些操作符是如何分别对一个或多个形式求值提供基本控制类型的。QUOTE完全避免求值，从而允许你得到作为数据的S-表达式。IF提供了基本的布尔选择操作符，从而构造出其他所有的条件执行构造。^①PROGN则提供了序列化一组形式的能力。

20.2 维护词法环境

特殊操作符中最大的一类由那些维护和访问词法环境 (lexical environment) 的操作符所组成。

^①当然，如果IF不是一个特殊操作符而其他某个条件形式（例如COND）是的话，那么你也可以将IF构造成一个宏。事实上，在许多Lisp方言里，从McCarthy最初的Lisp开始，COND都是那个最基本的条件求值操作符。

前面已经讨论的`LET`和`LET*`就是用于维护词法环境的特殊操作符，因为它们可以引入新的词法变量绑定。任何诸如`DO`或`DOTIMES`这类绑定了词法变量的构造都将被展开成一个`LET`或`LET*`^①。`SETQ`特殊操作符是一种用来访问词法环境的特殊操作符，因为可以用它设置那些由`LET`和`LET*`所创建的绑定。

不过，变量并不是唯一可以在词法作用域里命名的东西。虽然大多数函数都是通过`DEFUN`全局定义的，但也可能通过特殊操作符`FLET`和`LABELS`创建局部函数，通过`MACROLET`创建局部宏，以及通过`SYMBOL-MACROLET`创建一种特殊类型的宏，称为符号宏 (symbol macro)。

`LET`允许你引入一个词法变量，其作用域是`LET`形式的主体；同样，`FLET`和`LABELS`可以让你定义一个函数，使其只能在`FLET`或`LABELS`形式的作用域内被访问。在你需要一个比内联定义的`LAMBDA`表达式更复杂的局部函数，或是将多次使用的局部函数时，这些特殊操作符将会非常有用。两者具有相同的基本形式，看起来像下面这样：

```
(flet (function-definition*)
  body-form*)
```

以及

```
(labels (function-definition*)
  body-form*)
```

其中每个`function-definition`具有下面的形式：

```
(name (parameter*) form*)
```

`FLET`与`LABELS`之间的区别在于，由`FLET`所定义的函数名只能在`FLET`的主体中使用，而由`LABELS`所引入的名字却可以立即使用，包括`LABELS`所定义的函数本身。这样，`LABELS`可以用来定义递归函数，而`FLET`就不行。`FLET`不能用来定义递归函数虽然看起来是一种限制，但Common Lisp同时提供了`FLET`和`LABELS`，这是因为有时能够编写出一个调用另一个同名函数的局部函数也是有用的，被调用的同名函数可能是一个全局定义的函数或是来自外围作用域的另一个局部函数。

在一个`FLET`或`LABELS`的主体内，你可以像任何其他函数那样使用这些局部函数的名字，包括使用`FUNCTION`特殊操作符。由于可以使用`FUNCTION`来获得代表`FLET`或`LABELS`所定义函数的函数对象，并且因为一个`FLET`或`LABELS`可以定义在其他诸如`LET`这样的绑定形式的作用域内，所以这些函数可能是闭包。

因为局部函数可以引用来自其外围作用域的变量，所以它们通常可以书写成比等价的辅助函数带有更少参数的形式。这在你需要传递一个只接受单一参数作为函数参数的函数时尤为方便。例如，在下面的函数中（你在第25章里还会再次看到它），`FLET`所定义的函数`count-version`接受单一参数，这是`walk-directory`所要求的，但该函数还用到了由外围`LET`所引入的变量`versions`：

^① 从技术上来讲，这些构造也可以展开成一个`LAMBDA`表达式，正如我在第6章里所提到的，因为`LET`可以被定义成一个展开成对匿名函数调用的宏，而在一些早期的Lisp实现里确实就是这样做的。

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
             (incf (cdr (assoc (major-version (read-id3 file)) versions))))
           (walk-directory dir #'count-version :test #'mp3-p)))
      (walk-directory dir #'count-version :test #'mp3-p)
      versions)))
```

这个函数也可以写成在**FLET**定义的**count-version**位置上使用一个匿名函数，给这个函数一个有意义的名字更易于阅读它。

另外，当一个辅助函数需要进行递归时，使用匿名函数不可能做到这点。^①当你不想把一个递归的辅助函数定义成全局函数时，可以使用**LABELS**。例如，下面的函数**collect-leaves**使用递归的辅助函数**walk**来遍历一棵树，并把树中所有的原子收集到一个列表里，该列表（在求逆以后）由**collect-leaves**返回：

```
(defun collect-leaves (tree)
  (let ((leaves ()))
    (labels ((walk (tree)
                  (cond
                    ((null tree))
                    ((atom tree) (push tree leaves))
                    (t (walk (car tree))
                        (walk (cdr tree)))))))
      (walk tree))
    (nreverse leaves)))
```

要再次注意，在**walk**函数里可以引用变量**leaves**，它是由外围的**LET**引入的。

FLET和**LABELS**在用于宏展开时也是相当有用的——一个宏的展开代码里可以含有一个**FLET**或**LABELS**，用来创建可在宏主体中使用的函数。这个技术既可用来引入宏的用户要调用的函数，也可以只作为一种组织宏所生成的代码的方式。举个例子，这就是为什么能够定义出像**CALL-NEXT-METHOD**这种只能在一个方法的定义内使用的函数的原因。

一个与**FLET**和**LABELS**紧密相关的特殊操作符是**MACROLET**，它可以用来定义局部宏。局部宏的工作方式与**DEFMACRO**定义的全局宏一样，只是并不作用在全局名字空间上。求值一个**MACROLET**宏时，主体形式在局部宏定义生效的情况下被求值，其中的局部宏定义可能会覆盖全局的函数和宏定义，或是来自外围形式的局部定义。与**FLET**和**LABELS**一样，**MACROLET**可以被直接使用，也适用于宏生成的代码——通过将一些用户提供的代码包装进一个**MACROLET**，一个宏可以提供只用于这些代码中的构造，或是覆盖一个全局定义的宏。你将在第31章里看到**MACROLET**的后一种用法。

最后，还有一个定义宏的特殊操作符**SYMBOL-MACROLET**，它定义了一种名副其实的称为符号宏（symbol macro）的特殊类型的宏。符号宏和常规的宏相似，只是不能接受任何参数，并且只能用单个符号而非列表的形式来引用它。换句话说，当定义了一个特定名字的符号宏以后，在值的位置上对该符号的任何使用将被展开，由此产生的形式将在该位置上进行求值。这就是诸如

^① 听起来可能令人惊讶，但确实有可能使匿名函数成为递归的。不过，你必须使用一种称为“Y组合器”（Y combinator）的古怪手法。Y组合器属于一种有趣的理论结果，并非实用的编程工具，因此完全在本书的讨论范围之外。

WITH-SLOTS和**WITH-ACCESSORS**是如何定义“变量”用来在特定范围内访问某个特定对象的状态的。例如，下面的**WITH-SLOTS**形式：

```
(with-slots (x y z) foo (list x y z))
```

可以展开成使用**SYMBOL-MACROLET**的下列代码：

```
(let ((#:g149 foo))
  (symbol-macrolet
    ((x (slot-value #:g149 'x))
     (y (slot-value #:g149 'y))
     (z (slot-value #:g149 'z)))
    (list x y z)))
```

当表达式(`list x y z`)被求值时，符号`x`、`y`和`z`将被替换成它们的展开式，例如(`slot-value #:g149 'x`)。^①

符号宏通常都是局部的，由**SYMBOL-MACROLET**定义，Common Lisp也提供了一个宏**DEFINE-SYMBOL-MACRO**来定义全局的符号宏。一个由**SYMBOL-MACROLET**定义的符号宏将覆盖由**DEFINE-SYMBOL-MACRO**或外围**SYMBOL-MACROLET**形式所定义的其他同名符号宏。

20.3 局部控制流

接下来将讨论的四个特殊操作符也会在词法环境中创建并使用名字，目的是为了调整控制流而非定义新的函数和宏。我曾经提到过这四个特殊操作符，因为它们提供了其他语言特性用到的底层机制。它们是**BLOCK**、**RETURN-FROM**、**TAGBODY**和**GO**。前两个操作符**BLOCK**和**RETURN-FROM**一起使用时，可以立即从一段代码中返回我在第5章里讨论过的将**RETURN-FROM**作为一种从函数中立即返回的方式，但它还有更一般的用途。另外的**TAGBODY**和**GO**提供了一种相当底层的goto结构，这种结构是目前你所见到的所有更上层循环结构的基础。

BLOCK形式的基本结构如下所示：

```
(block name
      form*)
```

其中的`name`是一个符号，而`form`是一些Lisp形式。这些形式按顺序进行求值，最后那个形式的值作为整个**BLOCK**的值返回，除非有一个**RETURN-FROM**用来从块结构中提前返回。如同你在第5章里看到的，一个**RETURN-FROM**形式由打算返回到该形式的块名称，以及一个可选的提供了返

① **WITH-SLOTS**不一定非要用**SYMBOL-MACROLET**来实现。在某些实现里，**WITH-SLOTS**可能会遍历提供给它的代码，并生成一个带有`x`、`y`和`z`的，已经被替换成对应的SLOT-VALUE形式的展开式。你可以通过求值下面的形式来查看你所用的实现是怎样做的：

```
(macroexpand-1 '(with-slots (x y z) obj (list x y z)))
```

不过，遍历形式体这件事由Lisp实现来做比用户代码更容易一些。要想让`x`、`y`和`z`仅在作为值出现时才被替换掉，这需要一个代码遍历器能够理解所有的特殊操作符，并且可以递归地展开所有宏形式来检测其展开式里是否含有值位置上的那些符号。Lisp实现本身显然带有它自己的一个代码遍历器，但这是Lisp中没有暴露给语言用户的少数部分之一。

回值的形式所组成。当一个`RETURN-FROM`被求值时，它会导致该命名的`BLOCK`立即返回。如果调用`RETURN-FROM`时带有返回值，那么`BLOCK`将返回该值；否则整个`BLOCK`将求值为`NIL`。

一个块的名字可以是任何符号，包括`NIL`在内。许多标准控制构造宏，诸如`DO`、`DOTIMES`和`DOLIST`，都会生成一个含有名为`NIL`的`BLOCK`的扩展。这允许你使用`RETURN`宏来从这些循环中跳出，该宏是`(return-from nil ...)`的语法糖。这样，下面的循环将打印出至多10个随机数，并在首次遇到大于50的数字时立即停下来：

```
(dotimes (i 10)
  (let ((answer (random 100)))
    (print answer)
    (if (> answer 50) (return))))
```

另一方面，诸如`DEFUN`、`FLET`和`LABELS`这类可以定义函数的宏，会将它们的函数体封装在一个与该函数同名的`BLOCK`中。这就是你可以用`RETURN-FROM`来从一个函数中返回的原因。

`TAGBODY`和`GO`之间的关系类似于`BLOCK`和`RETURN-FROM`的关系：一个`TAGBODY`形式定义了一个上下文，其中定义的名字可被`GO`使用。一个`TAGBODY`形式的模板如下所示：

```
(tagbody
  tag-or-compound-form*)
```

其中每个`tag-or-compound-form`要么是一个称为标记（tag）的符号，要么是一个非空的列表形式。这些列表形式按顺序进行求值，而那些标记将被忽略，除了我即将讨论的一种情况。当`TAGBODY`的最后一个形式被求值以后，整个`TAGBODY`返回`NIL`。在`TAGBODY`的词法作用域的任何位置，你可以使用`GO`特殊操作符立即跳转到任何标记上，而求值过程将从紧跟着该标记的那个形式开始继续进行。例如，你可以像下面这样使用`TAGBODY`和`GO`编写一个简单的无限循环：

```
(tagbody
  top
  (print 'hello)
  (go top))
```

注意，尽管标记名必须出现在`TAGBODY`的最顶层，而不能内嵌到其他形式中，但`GO`特殊操作符却可以出现在`TAGBODY`作用域的任何位置上。这意味着可以像下面这样编写一个随机次数的循环：

```
(tagbody
  top
  (print 'hello)
  (when (plusp (random 10)) (go top)))
```

还有一个更无聊的`TAGBODY`示例，它表明你可以在单个`TAGBODY`里使用多个标记，看起来像这样：

```
(tagbody
  a (print 'a) (if (zerop (random 2)) (go c))
  b (print 'b) (if (zerop (random 2)) (go a))
  c (print 'c) (if (zerop (random 2)) (go b)))
```

上面这个形式将不断作随机跳转并顺便打印出一些a、b和c，直到最后一个RANDOM表达式偶然返回了1并且控制落到了TAGBODY的结尾。

很少直接使用TAGBODY，因为使用已有的循环宏来编写迭代控制构造往往更方便。不过，它在将来自其他语言的算法转译成Common Lisp时会很有用，无论是自动的还是手工的。一个自动转译工具的例子是从FORTRAN到Common Lisp的转译器f2cl，它将FORTRAN源代码转译成Common Lisp，从而允许Common Lisp程序员得以使用各种各样的FORTRAN库。由于许多FORTRAN库写于结构化编程革命以前，所以代码里有很多跳转(goto)语句。f2cl编译器可以简单地将那些跳转语句转译成带有适当TAGBODY的GO语句。^①

类似地，TAGBODY和GO在转译那些以文字或框图描述的算法时也很有用。例如，在Donald Knuth不朽的经典系列著作《计算机程序设计艺术》中，他使用了一种“菜谱”式的格式来描述算法：第一步，做这个；第二步，做那个；第三步，回到第二步；诸如此类。比如，在《计算机程序设计艺术，卷2：半数值算法》第3版(Addison-Wesley, 1998)的第142页，他以下面的形式描述了算法S，该算法将在第27章里用到：

算法 S (选择取样技术)：从有 N 个记录的集合里随机选出 n 个记录，其中 $0 < n \leq N$ 。

S1. [初始化] 设 $t \leftarrow 0$, $m \leftarrow 0$ 。(在本算法中， m 表示已选出的记录数， t 表示已经处理的输入记录的总数。)

S2. [生成 U] 生成一个随机数 U ，它平均分布在0和1之间。

S3. [测试] 如果 $(N-t)U \geq n-m$ ，那么转向步骤S5。

S4. [选择] 选择下一个记录，并将 m 和 t 递增1。如果 $m < n$ ，那么转向步骤S2；否则取样过程结束，算法终止。

S5. [跳过] 跳过下一个记录(不将它选作样本)，将 t 递增1，然后回到步骤S2。

这些描述可以轻易转译成一个Common Lisp函数，在重命名了一些变量以后，如下所示：

```
(defun algorithm-s (n max) ; max is N in Knuth's algorithm
  (let (seen                      ; t in Knuth's algorithm
        selected                  ; m in Knuth's algorithm
        u                         ; U in Knuth's algorithm
        (records ())              ; the list where we save the records selected
        (tagbody
         s1
         (setf seen 0)
         (setf selected 0)
         s2
         (setf u (random 1.0)))
         s3
```

^① 某个版本的f2cl现在是Common Lisp Open Code Collection (CLOCC) 的一部分，请查阅<http://clocc.sourceforge.net/>。相比之下，看看f2j (FORTRAN到Java的转译器) 的作者被迫采取的方法吧。尽管Java虚拟机 (JVM) 支持一个跳转指令，但它没有直接暴露给Java。因此为了编译FORTRAN的跳转语句，他们首先将FORTRAN代码编译成带有那些对代表标签和跳转的空类的调用的合法Java源代码，然后对产生的字节码进行后期处理，把那些空调用再转译成JVM层面的字节码。这是很聪明的做法，但是太折磨人了。

```

  (when (>= (* (- max seen) u) (- n selected)) (go s5))
s4
  (push seen records)
  (incf selected)
  (incf seen)
  (if (< selected n)
      (go s2)
      (return-from algorithm-s (nreverse records)))
s5
  (incf seen)
  (go s2)))

```

20

这不算是精美的代码，但很容易验证它是Knuth算法的一个忠实转译。这些代码和Knuth的文字描述的不同之处在于，它是可以运行和测试的。然后你可以开始着手重构它，确保该函数在每次变更之后仍然可以工作。^①

在经过一番优化以后，你可能最终会得到类似下面的代码：

```

(defun algorithm-s (n max)
  (loop for seen from 0
        when (< (* (- max seen) (random 1.0)) n)
        collect seen and do (decf n)
        until (zerop n)))

```

尽管一眼看不出来这些代码是否正确实现了算法S，但如果它是一系列与最初的Knuth算法描述的字面转译具有相同行为的函数，那么你有理由相信它是正确的。

20.4 从栈上回退

从语言的另一方面讲，特殊操作符还可以让你控制调用栈的行为。例如，尽管可以正常使用**BLOCK**和**TAGBODY**来管理单一函数内的控制流，但也可以将它们与闭包一起使用，可以强制从栈底部的函数立即非本地返回。这是因为**BLOCK**名字和**TAGBODY**标记可以被**BLOCK**或**TAGBODY**词法作用域之内的任何代码所闭合。例如，考虑下面这个函数：

```

(defun foo ()
  (format t "Entering foo~%")
  (block a
    (format t " Entering BLOCK~%")
    (bar #'(lambda () (return-from a)))
    (format t " Leaving BLOCK~%"))
  (format t " Leaving foo~%"))

```

传递给**bar**的匿名函数使用**RETURN-FROM**从**BLOCK**中返回。但**RETURN-FROM**要直到匿名函数被**FUNCALL**或**APPLY**调用时才会求值。现在假设**bar**函数看起来像这样：

^① 由于这个算法取决于**RANDOM**所返回的值。你也许想要使用一致的随机数种子来测试它，这可以通过在每一次对algorithm-s的调用中将***RANDOM-STATE***绑定到(**make-random-state nil**)的值上来实现。例如，你可以通过求值下面的形式来对algorithm-s做一次基本的健全性检查：

```
(let ((*random-state* (make-random-state nil))) (algorithm-s 10 200))
如果你的重构都是合法的，那么这个表达式应当每次求值都得到同样的列表。
```

```
(defun bar (fn)
  (format t " Entering bar~%")
  (baz fn)
  · (format t " Leaving bar~%"))
```

匿名函数仍然不会被调用。现在再看baz：

```
(defun baz (fn)
  (format t " Entering baz~%")
  (funcall fn)
  (format t " Leaving baz~%"))
```

最终，函数被调用了。但是一个位于栈的上方数层的BLOCK对于RETURN-FROM来说算什么呢？看起来一切工作正常——栈被回退到BLOCK最初建立的地方，且控制则从BLOCK返回。函数foo、bar和baz中的FORMAT表达式显示了这点：

```
CL-USER> (foo)
Entering foo
Entering BLOCK
Entering bar
Entering baz
Leaving foo
NIL
```

注意，唯一打印出的“Leaving ...”信息是foo函数中出现在BLOCK之后的那个。

由于块的名字是词法作用域的，一个RETURN-FROM总是从它所在的词法环境中最小的外围BLOCK上返回，即使RETURN-FROM是在不同的动态上下文中执行的。例如，bar也可以含有一个名为a的BLOCK，像这样：

```
(defun bar (fn)
  (format t " Entering bar~%")
  (block a (baz fn))
  (format t " Leaving bar~%"))
```

这个额外的BLOCK根本不会改变foo的行为——名字a是词法解析的，并且是在编译期而非动态地解析的，因此这个插入的块对于RETURN-FROM的行为没有影响。反过来说，只有当RETURN-FROM出现在该BLOCK的词法作用域内部时，才可以使用一个BLOCK的名字。没有办法让块外的语句从该块上返回，除非通过调用一个在BLOCK的词法作用域内部封装的闭包。

TAGBODY和GO在这一点上与BLOCK和RETURN-FROM的工作方式相同。当调用一个含有GO形式的闭包时，如果这个GO被求值，那么栈将回退到适当的TAGBODY，然后跳转到特定的标记上。

不过，BLOCK名字和TAGBODY标记在某个重要方面与词法变量绑定不同。如同我在第6章讨论的，词法绑定具有无限时效（indefinite extent），这意味着即便在绑定形式返回后绑定也可以保持效果。另一方面，BLOCK和TAGBODY具有动态时效（dynamic extent）——只有当BLOCK和TAGBODY在栈上时，你才能通过RETURN-FROM回到一个BLOCK，或者通过GO回到一个TAGBODY标记上。换句话说，一个捕捉了块名或是TAGBODY标记的闭包只能向栈的下方传递从而留到稍后再调用，但它不能向栈的上方传递。如果你调用了一个闭包，它试图在某个BLOCK本身返回以后再通过RETURN-FROM回到这个BLOCK上，那么你将得到一个错误。同样，试图通过GO回到一个不存在

的TAGBODY上也将导致错误发生。^①

你不太可能亲自使用BLOCK和TAGBODY来实现这种栈回退。但无论你何时使用状况系统，恐怕都是在间接地使用它们，因此理解其工作方式有助于你更好地理解它们，比如，当调用一个再启动时究竟发生了什么。^②

CATCH和THROW是另一对可以强制回退栈的特殊操作符。相比到目前为止提到的其他相关操作符，这些操作符使用得更少，它们是早期没有Common Lisp状况系统的Lisp方言所留下的东西。绝对不能把它们跟诸如Java和Python这些语言中的try/catch和try/except结构相混淆。

CATCH和THROW是BLOCK和RETURN-FROM的动态版本。就是说，你用CATCH包装了一个代码体，然后用THROW使CATCH形式立即从一个特定值返回。区别在于，CATCH和THROW之间的关联是动态建立的——相对一个词法作用域的名字来说，一个CATCH的标签是对象，称为捕捉标记(catch tag)，而任何在CATCH的动态时效中求值的THROW在抛出该对象时，将导致栈回退到CATCH形式上，然后它会立即返回。这样，你可以编写另一个版本的foo、bar和baz函数，像下面这样使用CATCH和THROW来代替BLOCK和RETURN-FROM：

```
(defparameter *obj* (cons nil nil)) ; i.e. some arbitrary object

(defun foo ()
  (format t "Entering foo~%")
  (catch *obj*
    (format t " Entering CATCH~%")
    (bar)
    (format t " Leaving CATCH~%"))
  (format t "Leaving foo~%"))

(defun bar ()
  (format t " Entering bar~%")
  (baz)
  (format t " Leaving bar~%"))

(defun baz ()
  (format t " Entering baz~%")
  (throw *obj* nil)
  (format t " Leaving baz~%"))
```

注意，没有必要向下传递闭包，baz可以直接调用THROW。结果和之前的版本很相似。

```
CL-USER> (foo)
Entering foo
Entering CATCH
Entering bar
Entering baz
Leaving foo
NIL
```

^① 这是一个相当合理的限制——从一个已经返回了的形式中返回的意义并不是完全清楚的——当然，除非你是一个Scheme程序员。Scheme支持续延(continuation)，一个允许从相同的函数调用中多次返回的语言构造。但出于多种原因，很少有Scheme之外的语言支持这类续延特性。

^② 如果你是那种凡事都要刨根问底的人，那么思考一下怎样才能通过BLOCK、TAGBODY、闭包和动态变量来实现状况系统的那些宏，这可能是相当有意义的。

不过，**CATCH**和**THROW**过于动态了。在**CATCH**和**THROW**中，标记形式都会被求值，这意味着它们的值都是在运行期检测的。这样，如果在bar中的某些代码重新赋值或绑定了`*obj*`，那么baz中的**THROW**将不会抛出同样的**CATCH**。这使得代码中的**CATCH**和**THROW**比**BLOCK**和**RETURN-FROM**更难理解。使用了**CATCH**和**THROW**的foo、bar和baz的演示代码的唯一优势就是，不再需要向下传递一个闭包以便底层代码可以从一个**CATCH**中返回——任何在一个**CATCH**的动态时效内运行的代码都可以通过抛出正确的对象来返回。

在那些没有任何类似Common Lisp状况系统机制的古老Lisp方言里，**CATCH**和**THROW**用于错误处理。不过，为了确保它们的可管理性，捕捉标记通常只是一些引用了的符号，因此你只需观察代码就可以看出**CATCH**和**THROW**是否会在运行期关联在一起。在Common Lisp中，你很少有机会使用**CATCH**和**THROW**，因为使用状况系统会更加灵活。

最后一个跟栈控制有关的特殊操作符是我之前提到过的操作符**UNWIND-PROTECT**。**UNWIND-PROTECT**让你能够控制在栈被回退时所发生的事——确保特定代码在控制离开**UNWIND-PROTECT**作用域的任何情况下总可以运行，无论是通过一个被调用的再启动正常返回，还是采用任何在本章中所讨论的方式。^①**UNWIND-PROTECT**的基本结构看起来像这样：

```
(unwind-protect protected-form
  cleanup-form*)
```

单一的*protected-form*被求值，随后无论它是否返回，*cleanup-form*都会被求值。如果*protected-form*正常返回了，那么它所返回的值将在执行这些用于清理的形式后被**UNWIND-PROTECT**返回。这些清理形式在与**UNWIND-PROTECT**相同的动态环境中被求值，因此，那些在进入**UNWIND-PROTECT**之前相同的动态变量绑定、再启动和状况处理器将对清理形式中的代码是可见的。

你偶尔会直接使用**UNWIND-PROTECT**。不过更常见的情况是将它作为**WITH**-风格宏的基础，类似于**WITH-OPEN-FILE**，它在上下文中求值任意数量的形式，其中它们所访问的某些资源需要在它们结束访问后被清理干净，无论它们是正常返回的、通过再启动返回的，还是其他的非本地退出。举个例子，如果你正在编写一个数据库，其中定义了函数**open-connection**和**close-connection**，你可能会写一个像下面这样的宏：^②

```
(defmacro with-database-connection ((var &rest open-args) &body body)
  ` (let ((,var (open-connection ,@open-args)))
    (unwind-protect (progn ,@body)
      (close-connection ,var))))
```

它可以让你写出类似下面这样的代码：

```
(with-database-connection (conn :host "foo" :user "scott" :password "tiger")
  (do-stuff conn)
  (do-more-stuff conn))
```

^① **UNWIND-PROTECT**本质上与Java和Python中的try/finally结构等价。

^② 事实上，CLSQL是跨Lisp平台和数据库的SQL接口库，它确实提供了一个称为**with-database**的类似的宏。CLSQL的主页是<http://clsql.b9.com>。



你不必担心数据库的关闭，因为UNWIND-PROTECT会确保数据库被关闭，不论with-database-connection形式的主体中发生了什么。

20.5 多值

Common Lisp的另一个特性是我在第11章里讨论GETHASH时提到过的，即单一形式可以返回多个值的能力。现在我将进一步讨论它的细节。不过，把这些内容放在关于特殊操作符的章节里并不太合适，因为多重返回值并不仅仅是由一两个特殊操作符提供的，而是紧密集成到了整个语言之中。在处理多值时，最常使用的操作符是宏和函数，而非特殊操作符，但最后获取多重返回值的基本功能是由特殊操作符MULTIPLE-VALUE-CALL提供的，而更常用的MULTIPLE-VALUE-BIND宏则构建于其上。

理解多重返回值的关键在于，返回多个值与返回一个列表是有本质不同的——如果一个形式返回了多个值，除非你做了一些特别的事情来捕捉多个值，那么除了主值（primary value）以外的其他值都将被悄悄地丢掉。为了理解这一区别，来看看函数GETHASH，它返回两个值：哈希表中找到的值和一个布尔值——在没有找到值时为NIL。如果它将这两个值返回到一个列表中，那么每次你调用GETHASH时都必须解析这个列表来取得实际的值，无论你是否关心第二个返回值。假设你有一个哈希表*h*，它含有一些数值。如果GETHASH返回一个列表，那么你就不能写出类似下面的形式了：

```
(+ (gethash 'a *h*) (gethash 'b *h*))
```

因为“+”期待其参数是数字而非列表。但由于多重返回值机制悄悄地丢弃了那个不需要的第二个返回值，从而使这个形式可以正常工作。

使用多重返回值包括两个方面——返回多个值以及获取那些返回多值的形式所返回的非主值。返回多值的开始点是函数VALUES和VALUES-LIST。这些都是正规函数而非特殊操作符，因此它们的参数将以正常方式传递。VALUES接受可变数量的参数并将它们作为多值返回，VALUES-LIST接受单个列表并将它的元素作为多值返回。换句话说：

```
(values-list x) ≡ (apply #'values x)
```

多值返回的机制和向函数传递参数的机制一样，都是具体实现相关的。几乎所有可以返回一些子形式的值的语言构造都会“传递”多值，并返回由其子形式返回的所有值。这样，一个返回了调用VALUES或VALUES-LIST的结果的函数将返回多值，并且其结果来自对这个函数的调用的另一个函数也会返回多值，以此类推。^①

但是当一个形式被放在值的位置上求值时，只有主值会被使用，这也就是之前的加法形式能

^① 少量有用的宏并不会传递它们所求值形式的其他返回值。特别的是PROG1宏，它像PROGN那样求值一组形式并返回第一个形式的值，只返回该形式的主值。同样，PROG2返回其第二个子形式的值，也只返回主值。特殊操作符MULTIPLE-VALUE-PROG1是PROG1的一个变体，它可以返回第一个形式的所有值。PROG1不具有MULTIPLE-VALUE-PROG1那样的行为，这多少算是一个缺点，但这两个宏其实都不太常用。OR和COND宏也并不总是对多值透明的，在特定的子形式上只返回其主值。

够以你期待的方式运行的原因。特殊操作符MULTIPLE-VALUE-CALL提供了访问一个形式返回的多个值的机制。MULTIPLE-VALUE-CALL和FUNCALL相似，除了FUNCALL是个正规函数并且因此只能看到并传递那些传给它的主值，而MULTIPLE-VALUE-CALL则为它第一个子形式返回的函数传递其余子形式返回的所有值。

```
(funcall #'+ (values 1 2) (values 3 4)) → 4
(multiple-value-call #'+ (values 1 2) (values 3 4)) → 10
```

不过，你一般不需要简单地将一个函数返回的所有值都传给另一个函数。更常见的用法是，你希望多个值分别保存在不同的变量里，然后再对这些变量作处理。第11章里提到的MULTIPLE-VALUE-BIND宏就是最常用的用于接收多重返回值的操作符。它的模板看起来像这样：

```
(multiple-value-bind (variable*) values-form
  body-form*)
```

其中*values-form*被求值，它返回的多个值被绑定到那些变量上。然后那些*body-form*在绑定的作用下被求值。这样：

```
(multiple-value-bind (x y) (values 1 2)
  (+ x y)) → 3
```

另一个宏MULTIPLE-VALUE-LIST甚至更简单——它接受单一的形式，求值它，然后将得到的多个值收集到一个列表中。换句话说，它是VALUES-LIST的逆操作。

```
CL-USER> (multiple-value-list (values 1 2))
(1 2)
CL-USER> (values-list (multiple-value-list (values 1 2)))
1
2
```

不过，如果你发现自己使用了很多的MULTIPLE-VALUE-LIST，这也许是一个信号，表明某些函数应当开始返回一个列表而不是多值了。

最后，如果你想要将一个形式返回的多个值一次性赋值到已有变量上，那么可以将VALUES作为可SETF的位置来使用。例如：

```
CL-USER> (defparameter *x* nil)
*x*
CL-USER> (defparameter *y* nil)
*y*
CL-USER> (setf (values *x* *y*) (floor (/ 57 34)))
1
23/34
CL-USER> *x*
1
CL-USER> *y*
23/34
```

20.6 EVAL-WHEN

为了写出某些特定类型的宏，你必须理解EVAL-WHEN操作符。出于一些原因，Lisp书籍通常将EVAL-WHEN视为巫师级别的话题。但其实理解EVAL-WHEN的唯一前提，只是理解两个函数LOAD

和`COMPILE-FILE`是如何交互的。理解`EVAL-WHEN`对于你开始编写特定类型的更加专业的宏非常重要，例如你将在第24章和第31章里编写的一些宏。

我在前面的章节已经简要提过`LOAD`和`COMPILE-FILE`之间的关系，这里有必要再说一次。`LOAD`的任务是加载一个文件并求值它包括的所有顶层形式。`COMPILE-FILE`的任务则是将一个源代码文件编译成FASL文件，后者随后可以被`LOAD`加载，因此(`(load "foo.lisp")`)和(`(load "foo.fasl")`)在本质上是等价的。

由于`LOAD`在读取每一个形式以后立即求值，因而求值文件中靠前面的形式就会影响读取和求值后续形式的行为。例如，求值一个`IN-PACKAGE`形式将改变`*PACKAGE*`的值，这将影响读取后续形式的方式。^①类似地，一个较早出现在文件中的`DEFMACRO`形式可以定义一个可被文件中后续代码使用的宏。^②

另一方面，`COMPILE-FILE`通常在编译时不求值文件中的形式。只有当FASL文件被加载时，这些形式或它们的编译后等价物才会被求值。尽管如此，`COMPILE-FILE`必须求值一些诸如`IN-PACKAGE`和`DEFMACRO`等形式，以保持(`(load "foo.lisp")`)和(`(load "foo.fasl")`)具有一致的行为。

那么诸如`IN-PACKAGE`和`DEFMACRO`这样的宏在被`COMPILE-FILE`处理时是怎样工作的呢？在Common Lisp标准之前的Lisp版本里，文件编译器简单地在编译后进一步求值某些形式。Common Lisp从MacLisp中借鉴了`EVAL-WHEN`，从而避免了类似异常情况的发生。顾名思义，这个操作符允许你控制特定的代码在何时被求值。`EVAL-WHEN`形式的模板看起来像这样：

```
(eval-when (situation*)
  body-form*)
```

存在三种可能的情形：`:compile-toplevel`、`:load-toplevel`和`:execute`，并且你指定的那些情形将控制所有`body-form`的求值时间。一个带有多重情形的`EVAL-WHEN`等价于分开的几个`EVAL-WHEN`形式，同样的代码每种情形各一个。为了解释三种情形的含义，我需要解释一下`COMPILE-FILE`，它也称为文件编译器，用来编译一个文件。

为了解释`COMPILE-FILE`是如何编译`EVAL-WHEN`形式的，先要介绍编译顶层形式和非顶层形式的区别。简单地说，一个顶层形式就是一些编译之后可以在FASL文件加载时运行的代码。这样，所有直接出现在一个源代码文件顶层的形式都将作为顶层形式来编译。类似地，任何直接出现在一个顶层`PROGN`中的形式也将作为顶层形式来编译，因为`PROGN`本身并不做任何事——它只

^① 加载一个带有`IN-PACKAGE`形式的文件，在`LOAD`返回以后看不到`*PACKAGE*`值的改变，这是因为`LOAD`在对该变量做任何改变之前绑定了它的当前值。换句话说，一些等价于下面这个`LET`形式的结构封装了`LOAD`中其余的代码：

```
(let ((*package* *package*)) ...)
```

任何对`*PACKAGE*`的赋值都将是新的绑定，而旧的绑定将在`LOAD`返回时恢复。它还以同样方式绑定了变量`*READTABLE*`，该变量我尚未谈及。

^② 在某些实现中，你或许可以正确求值一个在函数体中使用了未定义宏的`DEFUN`定义，只要在函数实际被调用之前定义好该宏即可。但仅当从源代码加载这些定义时，它们才可以正常工作，而在通过`COMPILE-FILE`编译时是不行的。因此一般来说，宏定义必须在它们被使用前被求值。

是把其子形式组织在一起，然后它们会在FASL被加载时运行。^①类似地，直接出现在一个MACROLET或SYMBOL-MACROLET中的形式将作为顶层形式来编译，因为在编译器展开了这些局部宏或符号宏之后，编译后的代码中就不再有MACROLET或SYMBOL-MACROLET了。最后，一个顶层宏形式的展开式将作为顶层形式来编译。

这样，一个出现在源代码文件顶层的DEFUN就是一个顶层形式，那些定义了函数并且将其与函数名相关联的代码将会在加载FASL时运行——但是函数体中的形式不是顶层形式，它们要等到函数被调用时才会运行。大多数形式在作为顶层和非顶层形式来编译时产生的结果都是相同的，但一个EVAL-WHEN的语义取决于它是作为顶层形式还是非顶层形式来编译的，或是简单地被求值，所有这些条件将按照列在其情形列表中的情形组合来决定。

当一个EVAL-WHEN作为顶层形式来编译时，情形：compile-toplevel和：load-toplevel控制其含义。当存在：compile-toplevel时，文件编译器将在编译期求值其子形式。当存在：load-toplevel时，它将把那些子形式作为顶层形式来编译。如果这两个情形都不在顶层EVAL-WHEN的情形列表中，那么编译器将会忽略它。

当一个EVAL-WHEN作为非顶层形式来编译时，它要么在：execute情形被指定时像PROGN那样被编译，要么被忽略。类似地，一个被求值的EVAL-WHEN（包括那些被LOAD加载的源代码文件中的顶层EVAL-WHEN，以及出现在带有：compile-toplevel情形的顶层EVAL-WHEN的子形式中的在编译期被求值的EVAL-WHEN），要么在：execute存在时被当作PROGN来对待，要么被忽略。

这样，一个诸如IN-PACKAGE这样的宏可以通过展开成类似下面这样的形式，来确保同时在编译期和源代码加载期都产生效果：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf *package* (find-package "PACKAGE-NAME")))
```

*PACKAGE*在编译期设置是因为有：compile-toplevel情形，在加载FASL时设置是因为有：load-toplevel，而在加载源代码时设置是因为有：execute。

你会经常在两种情况下使用EVAL-WHEN。一种是你想编写一个需要在编译期保存一些信息的宏时，这些信息将被同一个文件中的其他宏形式的展开式所用到。通常这些都是定义性的宏：一个文件开始处的定义可以影响同一个文件内其他定义所生成的代码。你将在第24章里编写这种类型的宏。

另一个你可能需要EVAL-WHEN的场合是，你想要把一个宏和它的辅助函数与使用该宏的代码放在同一个文件里。DEFMACRO已经在其展开式里包含了一个EVAL-WHEN，因此宏定义可以立即被文件的后续部分所使用。但是DEFUN正常情况下并不会在编译期产生函数定义。但如果你在定义了一个宏的文件中使用这个宏，就需要确保被用到的宏和该宏所用到的函数都有定义。如果你把该宏所用到的任何函数的DEFUN都封装在一个带有：compile-toplevel的EVAL-WHEN里，那么在宏的展开函数运行时这些定义就可以使用了。你很可能想再加上：load-toplevel和：execute，因为

^① 相反，一个顶层LET中的子形式并不会作为顶层形式来编译，因为它们不会在FASL加载时直接运行。它们将会运行，但是在由LET所建立的绑定和运行期上下文中运行的。理论上来说，一个不绑定任何变量的LET将会被当作PROGN来对待，但其实不是这样的，出现在LET中的子形式不会作为顶层形式来对待。

在文件被编译加载或者从源代码不编译而直接加载以后，该宏也需要这些函数。

20.7 其他特殊操作符

其余的4个特殊操作符分别是**LOCALLY**、**THE**、**LOAD-TIME-VALUE**和**PROGV**，它们都允许你访问以其他任何方式都无法访问到的语言的底层部分。**LOCALLY**和**THE**是Common Lisp声明系统的一部分，它们用来与编译器沟通而对代码的含义没有影响，但可能有助于编译器生成更好的代码，比如更快更清晰的错误信息。^①我将在第32章里简要地讨论有关声明的内容。

另外两个**LOAD-TIME-VALUE**和**PROGV**都很少会用到，而且解释为什么会想要使用它们，比解释它们能干什么还费劲。因此我只告诉你它们能干什么，让你知道它们的存在。日后当你偶尔遇到刚好可以用上它们的场合时，就知道该怎么用了。

顾名思义，**LOAD-TIME-VALUE**用来创建一个在加载期决定的值。当文件编译器编译含有**LOAD-TIME-VALUE**形式的代码时，它会安排在加载FASL时只求值其第一个子形式一次，然后让含有**LOAD-TIME-VALUE**形式的代码指向该值。换句话说，下面的写法：

```
(defvar *loaded-at* (get-universal-time))

(defun when-loaded () *loaded-at*)
```

可以简化成这样：

```
(defun when-loaded () (load-time-value (get-universal-time)))
```

在没有被**COMPILE-FILE**处理过的代码中，**LOAD-TIME-VALUE**仅在代码被编译时求值一次，这可能是在显式使用**COMPILE**编译一个函数，或是在求值代码的过程中具体实现进行了隐式的编译时发生的。在未编译的代码中，**LOAD-TIME-VALUE**在其每次被求值时都会求值其子形式。

最后，**PROGV**可以为变量创建其名字在运行期才确定的新的动态绑定。这对于支持动态作用域变量的语言实现嵌入式解释器尤其有用。其基本框架如下所示：

```
(progv symbols-list values-list
      body-form*)
```

其中**symbols-list**是一个形式，它求值到一个符号的列表上，而**values-list**是一个求值到值列表的形式。每个符号被动态地绑定到对应的值上，然后求值那些**body-form**。**PROGV**和**LET**的区别在于，**symbols-list**是在运行期求值的，被绑定的变量名可以动态地确定。要我说，这并不是你需要经常干的事情。

这就是特殊操作符的所有内容。在下一章里，我将回到更加实用的话题，展示如何使用Common Lisp的包系统来控制名字空间，这样你才可以写出彼此并存而没有名字冲突的库和应用程序。

^① 唯一一个对程序语义有影响的声明是第6章里提到的**SPECIAL**声明。