

实用Common Lisp 编程

Practical Common Lisp

实用Common Lisp 编程

[美] Peter Seibel 著
田春 译

[美] Peter Seibel 著

人民邮电出版社

- 第16届 Jolt 生产效率奖图书
- 了解黑客青睐的编程语言
- 带你进入 Common Lisp 的精彩世界

“本书展示了Lisp的威力，不仅表现在传统领域上，例如仅使用短短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的领域上，诸如解析二进制MP3文件、构建浏览歌曲集的Web应用、在Web上传播音频流等。读过本书，你将体会到Lisp具有Python等脚本语言的简洁性、C++的高效性，以及在设计语言扩展时无与伦比的灵活性。”

——Peter Norvig
Google公司研究中心主任
《人工智能：现代方法》的作者



Peter Seibel

从作家演变成程序员，又从程序员演变成作家，其职业生涯可谓一波三折。他在获得英语专业学士学位后做过一段时间的记者工作，后来被Web所吸引。在20世纪90年代早期，他用Perl建立了*Mother Jones*杂志和Organic Online网站。他作为WebLogic的早期雇员参与了Java革命，随后又在加州大学伯克利分校教授Java编程。他也是第二代Lisp程序员之一，并曾经是Symbolics的早期股东。2003年他辞去技术工作，潜心研究Lisp，并凭借本书获得Jolt生产效率大奖。2009年他出版了名噪一时的访谈录《编程人生》(*Coders at Work*)。

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 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

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
4.7 宏	41
4.8 真、假和等价	42
4.9 格式化 Lisp 代码	43
第 5 章 函数	46
5.1 定义新函数	46
5.2 函数形参列表	47
5.3 可选形参	48
5.4 剩余形参	49
5.5 关键字形参	50
5.6 混合不同的形参类型	51
5.7 函数返回值	52
5.8 作为数据的函数——高阶函数	53
5.9 匿名函数	55
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		
19.3 状况处理器	205		
19.4 再启动	207		
19.5 提供多个再启动	210		
19.6 状况的其他用法	211		
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 计数型循环	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		
30.7 基本求值规则	385		
30.8 下一步是什么	389		
第 31 章 实践：HTML 生成库，编译器部分	390	第 32 章 结论：下一步是什么	404
31.1 编译器	390	32.1 查找 Lisp 库	404
31.2 FOO 特殊操作符	395	32.2 与其他语言接口	406
		32.3 让它工作，让它正确，让它更快	406
		32.4 交付应用程序	413
		32.5 何去何从	415

书评，作者和序言等

对《Practical Common Lisp》的评价（封面）

“Peter Seibel 的《Practical Common Lisp》名副其实：对那些想要学习并开始将 Common Lisp 用于实际工作的人来说是一本极佳的介绍性教材。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。

“与其花大量时间来抽象地讨论 Lisp 在编程语言界的地位，Seibel 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专家也要查询手册才知道的 Common Lisp 语言特性的角落。Seibel 这一示例驱动的思路的结果就是让读者准确地感受到了 Common Lisp 在以最少的努力构建复杂和革命性的软件系统过程中所体验出的强大威力。

“在 Common Lisp 领域已经有许多好的书籍采用了更为抽象和比较性的编写思路，但一本‘这就是你怎样完成它——以及为什么要这样做’风格的好书绝对是很有价值的贡献，无论对当前的 Common Lisp 用户和其他潜在的用户来说都是这样。”

——Scott E. Fahlman, Carnegie Mellon 大学计算机科学研究教授

“要是这本书在我当初最开始学 Lisp 的时候已经存在就好了。并不是说当时没有其他关于 Common Lisp 的好书了，但它们都不像这本书那样具有务实和与时俱进的思想。并且让我们不要忘了 Peter 涵盖了诸如路径名、状况和再启动等主题，这些内容在其他 Lisp 著作里完全被忽略了。

“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这本书吧。一旦你阅读并学会了它，然后接下来你就可以继续阅读 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 正则表达式库的作者。

“有经验的程序员可以从示例中学到最有用的知识，而 Lisp 最终被用在 Seibel 这本示例丰富的入门教材中是件令人高兴的事。而尤其令人高兴的是，这本书包含了如此多的涵盖当今程序员日常可能用到的诸多问题领域，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

“《Practical Common Lisp》是一本涵盖了 Common Lisp 广泛内容的优秀书籍，其中还演示通过一些读者可以运行和扩展的来子现实世界的应用来阐述 Common Lisp 的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. 的资深科学家

“Maxima 项目经常得到一些来自那些想要学习 Common Lisp 的潜在贡献者的垂询。我很高兴最终可以有一本书让我可以毫无保留地推荐给他们了。Peter Seibel 那干净直接的风格允许读者快速地领略 Common Lisp 的威力。他包含在书中的许多示例集中在当代的编程问题上，充分说明了 Lisp 绝不仅仅是一门学院派编程语言。《Practical Common Lisp》是该领域的一本受欢迎的新著作。”

——James Amundson, Maxima 项目负责人

“我喜欢那些分散在书中的描述‘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，将字符串和数字组装成树和图这件事可以在 Lisp 里轻松地做到。”

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

“学习一门编程语言的最重要的部分之一就是学习它的正确编程风格。这很难教授，但通过阅读《Practical Common Lisp》可以无痛地吸收。仅是阅读那写实践性的示例就可以让我成为任何语言的更好的程序员了。”

——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一个程序员的日常工作中是有用的。”

——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来学习它，那么《Practical Common Lisp》将是一个极佳的入口点。”

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

对《Practical Common Lisp》的评价

“终于有了一本为我们其他人所写的 Lisp 书了。如果你想要学习如何编写一个阶乘函数，那么这本书不适合你。Seibel 的书是为实用程序员所写的，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语言的威力。

“在大多数章节里，阅读时的感觉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，我有感于如此一个简单的例子是如何让我真正‘领会’它们的。书中的叙事性内容是极其有用的，而采用这种写法的技术书籍是非常少的。恭喜你！”——Keith Irwin, Lisp 程序员

“尽管在学习 Lisp 的过程中，当人们不知道一个特定函数的用途时会去查询 CL HyperSpec，但我发现只是通过阅读 HyperSpec 通常很难‘领会’其含义。当我遇到这类问题时，我每次都会打开《Practical Common Lisp》——它是目前在教你如何编程方面最具可读性的来源，绝不仅仅是平铺直叙。”——**Philip Haddad, Lisp 程序员**

“在急速发展的 IT 形势下，专业人员需要最强大的工具。这就是为什么 Common Lisp——最强大、灵活和稳定的编程语言——正获得广泛的关注。《Practical Common Lisp》是一本期待依旧的书，它将帮助你驾驭 Common Lisp 的威力以应对当今各种复杂的现实问题。”——**Marc Battyani, CL-PDF、CL-TYPESETTING 和 mod_lisp 的作者**

“请不要假设 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，那是因为这些东西碰巧在《Practical Common Lisp》一书中被实现了。”——**Tobias C. Rittweiler, Lisp 程序员**

“当我遇到 Peter 时，他正在写这本书。我问我自己（而不是问他），‘为什么在已经有了许多很好的介绍性书籍以后还要写又一本关于 Common Lisp 的书？’一年以后，我发现了这本新书的一份草稿，同时意识到我最初的想法错了。这本书不是‘又一本’书。作者将注意力集中在实践方面而非语言的技术细节上。当我最初通过阅读一本介绍性书籍来学习 Lisp 时，我觉得我理解了这门语言，但我立即有了这样的感觉：‘那又怎样？’——这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解释了最基本的语言概念以后就顺序转向了一个‘实用性’章节。然后读者将在跟随这些‘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相当规模的产品。在读完本书以后，读者们将会感到他们已经是 Common Lisp 程序员了，因为已经‘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来构建了一次对 Common Lisp 的有趣的介绍。”——**Taiichi Yuasa 教授，京都大学计算机科学与通信学院**。

关于作者

Peter Seibel 既是一个从作家变成的程序员也是一个从程序员变成的作家。在获得英语专业学士学位并简短地作为记者工作一段时间以后，他被 Web 所引诱了。在 1990 年代早期他用 Perl 建立了 Mother Jones 杂志和 Organic Online 的网站。他作为 WebLogic 的早期雇员参与了 Java 革命并随后在加州大学伯克利分校教授 Java 编程。他也是这个星球上的第二代 Lisp 程序员之一，并曾经是 Symbolics 的早期股东。他和他的妻子 Lily 以及他们的狗 Mahlanie 一起生活在奥克兰。

关于技术审稿人

Barry Margolin 在 1970 年代晚期上高中的时候自学了计算机编程，首先在 DEC PDP-8 时间共享系统上，随后在 Radio Shack TRS-80 个人计算机上，并且他通过反向工程这些系统学会了操作系统设计。他去了 M. I. T.，在那里他师从 Bernie Greenberg、David Moon 和 Alan Bawden 等人学习 Lisp 编程。Bernie Greenberg 是 Multics MacLisp 编译器和 Multics Emacs（第一个用 Lisp 写的 Emacs 克隆）的作者；David Moon 是 ITS MacLisp 的实现者之一以及 Symbolics 的

创立者之一；Alan Bawden 则可能是最好的 Lisp 宏逻辑学家之一。在他获得了计算机科学学位以后，他去了 Honeywell Multics 开发组，维护 Emacs。在 Honeywell 终止了 Multics 的开发以后，他去了 Thinking Machines 公司，参与维护他们的 Lisp 机开发环境。自从那时起，他辗转工作于 Bolt、Beranek 和 Newman——后来成为了 BBN Planet，然后是 GTE Internetworking，然后是 Genuity，直到被 Level(3) 收购——为他们的 Internet 服务提供技术支持。他现在为 Symantec 工作，为他们的企业级防火墙产品提供二级客户技术支持。

致谢

如果不是因为一些愉快的巧合的话，本书不会被写出来，至少作者不是我。因此，我必须从感谢 Franz 的 Steven Haflich 开始。当我们在 Bay Area Lispniks 见面会上相遇时，他邀请我和 Franz 的一些销售人员一起吃午饭，席间我们讨论了对一本 Lisp 新书的需求。然后我必须感谢 Steve Sears，那天午饭上的一名销售人员，他随后将我引荐给 Franz 的总裁 Fritz Kunze，后者曾经提到他正在寻找某人来写一本 Lisp 书。当然，也非常感谢 Fritz 说服 Apress 出版社发布一本新的 Lisp 书，决定由我来写，并在整个过程中提供激励和帮助。也感谢 Franz 的 Sheng-Chuang Wu，诸多帮助的协调人。

我在撰写本书时最不可或缺的资源是新闻组 comp.lang.lisp。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，一个技术资料的宝库。因此，感谢 Google 提供了这一服务，也感谢所有 comp.lang.lisp 的参与者们一直以来提供各种内容。特别地，我想要指出两个长期的 comp.lang.lisp 贡献者——Barry Margolin 在我整个阅读该组期间一直在提供各种琐碎的关于 Lisp 史和他的个人经验的资料；以及 Kent Pitman，他除了作为语言标准的首席技术编辑之一和 Common Lisp HyperSpec 的作者以外，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。

另一个在撰写本书时不可或缺的资源是用于 PDF 生成和排版的 Common Lisp 库，由 Marc Battyani 所编写的 CL-PDF 和 CL-TYPESETTING。我使用 CL-TYPESETTING 来生成用于我个人编辑工作的有用的 PDF 文件，并将 CL-PDF 作为我用来生成出现在本书中的插图的 Common Lisp 程序的基础。

我还想感谢许多在 Web 上阅读了本书部分章节的草稿并通过 e-mail 指出其中的笔误、提出问题，或简单地给出建议的人们。尽管没有办法提及所有这些名字，但其中的一些由于他们特别有价值的反馈而值得在这里明确指出来：J. P. Massar (Bay Area Lispnik 的成员，他在几次披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 Gabour。我的几个非 Lisp 相关的朋友也参与了审阅某些章节：感谢 Marc Hedlund、Jolly Chen、Steve Harris、Sam Pullara、Sriram Srinivasan 和 William Grosso 以及他们的反馈。另外也感谢 Scott Whitten 允许我使用出现在图 26-1 中的那张照片。

我的技术审稿人 Steven Haflich、Mikel Evins 和 Barry Margolin，以及我的编辑 Kim Wimpsett，他们以数不清的方式改进了本书。当然，如果书中还有任何错误的话，都是我自己的责任。同时也感谢 Apress 出版社的其他所有参与本书出版工作的人们。

最后，也是最重要的，我想要感谢我的家庭。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 Lily，我的妻子。

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、类等事物的名字，要么是我正在介绍的，要么是我将要介绍的。由语言标准所定义的名字被写成这样：`DEFUN`。更大块的示例代码如下所示：

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

由于 Common Lisp 的语法以其正规和简洁性著称，因此我采用简单的模版来描述一些 Lisp 形式的语法。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*)
```

这些模版中以斜体书写的名称意味着需要填入我将在正文中描述的特定名字或形式。一个后跟星号 (*) 的斜体名字代表该名字的零次或更多次的出现，而一个位于方括号 [] 内的名字代表可选的元素。偶尔也会出现以竖线 (|) 分隔的替代内容。模版中的其他所有东西——通常只是一些名字和括号——都是将会出现在形式中的字面文本。

最后，由于你跟 Common Lisp 之间的许多交互都发生在交互性的“读-求值-打印循环”或 REPL 中，我将经常像下面这样显示一个在 REPL 中求值 Lisp 形式的结果：

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

其中的 `CL-USER>` 是 Lisp 的提示符，并且总是后跟需要求值的表达式，本例中是 `(+ 1 2)`。求值的结果和其他生成的输出都被显示在接下来的几行里。我有时也会通过在表达式后面跟上一个 → 来表示求值的结果，就像下面这样：

```
(+ 1 2) → 3
```

偶尔我还会使用一个等价符号 (≡) 来说明两个表达式是等价的，就像这样：

```
(+ 1 2 3) ≡ (+ (+ 1 2) 3)
```

书评（封底）

“本书显示 Lisp 威力的方式不只是通过传统上已经提到的领域——例如使用短短 26 行代码开发一个完整的单元测试框架——而且还使用了一些全新的领域，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 上传播音频流。许多读者将对 Lisp 做到所有这些事的方式感到惊奇：类似于 Python 等脚本语言的简洁性，类似于 C++ 的效率，以及在设计你自己的语言扩展时无与伦比的灵活性。”

——Peter Norvig, Google 搜索质量组负责人;《Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp》的作者。

作者注 (封底)

亲爱的读者,

实用 Common Lisp……这难道不是反语吗? 如果你和大多数程序员一样, 那么你可能知道一点儿 Lisp——要么来自大学里的计算机科学课程, 要么来自学习了足够多的 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 这门历史上最伟大的语言。但你可能从未想过你会在同一本书的标题里看到“实用”和“Lisp”。

但你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 将使你成为任何语言的一名更好的程序员。或者可能你只是想要知道那些 Lisp 狂热者们每天究竟在喊些什么。或者可能你已经学了一些 Lisp 但还没有越过障碍去用它编写感兴趣的软件。

如果任何一种上述情形是真的, 那么本书就是为你而写的。通过使用 Common Lisp, 一种 ANSI 标准化的工业级别的 Lisp 方言, 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的编辑器定制的真实软件。并且我将说明 Lisp 即便在其许多特性已被其他语言所采纳以后还仍然尤其独到之处。

但是和许多 Lisp 书籍不同的是, 这本书不会只是刚刚触及到一些 Lisp 的伟大特性就留给你自己来思考如何使用它们。我涵盖了你将在编写实际程序中用到的所有语言特性, 并且我将本书超过三分之一的篇幅用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, 以及一个带有完整的在线 MP3 数据库和 Web 接口的通过网络流式传输 MP3 的服务器。

因此, 把这本书翻过来, 打开它, 看看这门曾经被发明的最伟大的语言在你的手里能有多么实用吧。

此致,

Peter Seibel

第1章 緒言：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完成许多事，那么 Common Lisp 编程可以说是用电脑所能做到的最有趣的事了。相比多数其他计算机语言，它可以让你更快地完成更多工作。

这是个大胆的主张。可我要怎样证明呢？当然不是用本章里的只言片语了，而是接下来的整本书——你必须先学习一些 Lisp 知识才能体会到这一点。不过眼下，让我们先从一些轶事开始，即我本人的 Lisp 之路。然后，在下一节里我将说明通过学习 Common Lisp 你可以得到些什么。

我是极少数的所谓第二代 Lisp 黑客之一。我父亲计算机生涯的开始是用汇编语言给他用来搜集物理学论文数据的机器编写操作系统。在成功帮助了几个物理实验室运行计算机系统之后，从 1980 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设备中生产过程的软件——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，已经烧掉了该项目一半的预算和几乎全部的时间，却没有交出任何成果。那是在 1980 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 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。然后他又说服了老板让他的团队接手那个失败的项目并用 Lisp 来重写。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。我父亲将其团队的成功归因于他们使用 Lisp 的决定。

当然，仅此一例还不足以说明问题。而且也有可能我父亲在其成功的原因上认识有误。或者也许 Lisp 仅仅在那个年代才可以跟其他语言相比较。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。我真的可以说 Lisp 在今天也具有跟 1980 年代我父亲那个时候一样的优势吗？请继续往下读。

尽管我父亲做了大量努力，但我直到高中也没有学过一点儿 Lisp。我在大学时代也没怎么用任何语言来写程序，是后来出现的 Web 让我重新回到了计算机的怀抱里。我首先使用的是 Perl，先是为 Mother Jones 杂志的网站构建在线论坛，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 Nike 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，我对编程语言的广泛兴趣使我充分领略了诸如 C、C++ 和 Python 这些主流语言，以及像 Smalltalk、Eiffel 和 Beta 这些小

众语言。

所以我对两种语言了如指掌，同时还熟悉其他不下六种。不过最终我还是意识到，我对编程语言的兴趣其实来源于那些根植在我头脑之中的我父亲的 Lisp 经历——语言和语言之间真的是天壤之别，而且尽管所有的编程语言在形式上都是图灵等价的，但一些语言真的可以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。不过讽刺的是，我从来没有花太多时间在 Lisp 上。于是我开始利用业余时间研究 Lisp。但无论我做什么，最令我兴奋的始终是我可以多快地将我的思路变成可用的代码。

举个例子，在一次假期里我花了差不多一个星期的时间在 Lisp 上，目标是实现一个基于遗传算法的程序系统来下围棋——我以前做 Java 程序员的时候已经写过了。虽然受限于我那时极其有限的 Common Lisp 知识，并且不时要去查询基本函数的用法，但我还是可以感觉到比用 Java 重写同样的程序来得顺手——尽管自从编写该程序的最初版本以来我又多了几年的 Java 经验。

类似的经历导致了我将在第 24 章里讨论的一个库的诞生。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。这个库可以正常使用，但是代码有一点儿乱，并且难以修改或扩展。几年来我曾多次重写这个库，并期望以我日益提高的 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。但是当我改用 Common Lisp 来做这件事时，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，还有一个通用的库可用于解析任何二进制文件。我们将在第 24 章里讨论这个库的工作原理，然后在第 25 章里用它写一个 MP3 文件的内嵌 ID3 标签的解析器。

1.1 为什么是 Lisp?

很难用序章的寥寥数页来说清楚为什么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个语言。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，你在开始学习 Lisp 之前仍然对投入产出心存疑虑也是可以理解的。

对某些语言来说，回报是相对明显的。举个例子，如果你打算编写 Unix 上的底层代码，那你应该去学 C。或者如果你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，那你就应该去学 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 拥护者们经常说 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它们的格言所说：“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，除非你刚好拥有一台 Lisp 机）。使用 Lisp 所获得的好处在很大程度上取决于使用经验。本书的其余部分将向读者展示这些经验究竟是什么。眼下，我只想让你先

^① Perl 作为“the duct tape of the Internet”也同样值得学习。

对 Lisp 哲学有个大致的感受。

Common Lisp 中最接近格言的，是一句类似回文的描述：“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追求一种哲学——凡是语言的设计者可以做到的，语言的使用者也可以。这就意味着，当你用 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你的程序更容易写的特性，因为——这也是你将在本书中看到的——你可以为语言添加任何你想要的特性。

因此，Common Lisp 程序倾向于把你关于程序如何工作的思想更清楚地映射到你实际所写的代码上。你的思想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。这将使你的代码更加易于维护，因为你不必在每次修改之前都先回顾大量的相关代码。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做相对少量的修改来实现。这也意味着你将可以更快速地开发；需要写的代码量变少了，你也不必再花时间在语言的限制里寻求更干净的方式来表达你的思想了。^①

Common Lisp 也是一门适合做探索性编程的优秀语言——如果你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帮助你增量和交互地开发你的程序代码。

对于初学者来说，这个将在下一章里介绍的交互式“读-求值-打印”循环可以让你在开发过程中持续地与你的程序交互。编写一个新函数，测试它，修改它，尝试不同的实现方法。你的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，你可以花较少的时间让编译器把你的代码跑起来，然后把更多的时间放在如何实际运行和改进代码上^③，而后者甚至可以让你交互式地开发你的错误处理代码。

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

^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 Lisp 在程序员和程序效率的组合上相比 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籍《人件：富有成果的项目和团队》（Peopleware: Productive Projects and Teams，作者 Dorset Hourse，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冒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，也是一个应该了解它的理由。

^③ 这个观点很可能有争议，至少对某些人来说。静态和动态类型的信仰之争在编程领域由来已久。如果你来自 C++ 和 Java（或者是诸如 Haskell 和 ML 这样的静态类型函数式编程语言）并且拒绝生活在没有类型检查的环境里，你可能会就此合上书本了。不过，在此之前，你最好先听听“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 的作者 Bruce Eckel（《Thinking in C++》[Prentice Hall, 1995 年]和《Thinking in Java》[Prentice Hall, 1998 年]的作者）在他们的网络博客（<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，来自 SmallTalk、Python、Perl 或者 Ruby 的人们应该会对 Common Lisp 的这方面感觉良好。

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 Common Lisp 写成的库。而在这个库在其被吸收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的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做任何修改，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 很好地支持了一组常用的现代数据类型；将在第 19 章里介绍的状态系统提供了 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？地球上怎么会进化出如此装备精良的语言来？

这么说吧，McCarthy 曾经是（现在也是）一个人工智能（AI）研究者，他在该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 1980 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困难问题。这些问题都需要大量难于编写的软件；为了处理它们，AI 程序员们需要一门强大的语言，而他们将 Lisp 发展成了这样一门语言。另外，冷战也起了积极的作用——五角大楼向国防部高等研究规划局（DARPA）投入了大量资金，其中的相当一部分给了那些研究诸如大尺度战场模拟、自动规划以及自然语言接口等问题的人。这些人也在使用 Lisp 并且持续地推进它以满足自身的需求。

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吃掉大量的计算资源，而如果你按照摩尔定律倒推 20 年，你就可以想象 80 年代的计算资源是何等的贫乏了。Lisp 工作者们不得不想尽办法从他们的实现中挤出更多的性能来。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的语言里也能获得可接受的性能了，性能对于

^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 的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) 那难以想象的灵活性。为了实现 AspectJ，Kiczales 不得不在事实上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 (M.I.T. Press, 1962 年)

Common Lisp 来说再也不是问题了。不过，读者在第 32 章里仍然可以看到，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 编译器的成果相媲美。

1980 年代也是 Lisp 机的年代，当时好几家公司（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 1980 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，旨在集既有 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被记录在 Guy Steele 的《Common Lisp: the Language》(CLtL) 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，美国国家标准组织 (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 和状态系统。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了一定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 接口、多线程控制和 TCP/IP 套接口等。今天的 Common Lisp 则进化得更像其他的开源语言——用户编写他们所需要的库并开放给其他人。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，其建构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和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个 Lisp 方言，以及他们是否是在远古时代学到这些东西的。^②

但是我曾经学过 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，并且取决于你在何时何地认识了 Lisp，你很有可能学的是某种其他方言。

^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 这些语言那样使用广泛，但把一个继续被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，后者在 Yahoo 买下了他的公司以后成为了 Yahoo Store；ITA Software 的航空票务系统 QPX 被在线机票销售商 Orbitz 等使用着；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 的领域相关 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。

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 M. I. T. 被设计出来，然后很快被用作本科计算机科学课程的教学语言，它总是被定位成一种比 Common Lisp 更加优美但又不同的语言。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。但是，这样做带来的问题是它缺失了许多在 Common Lisp 里已经标准化了的特性。个别的 Scheme 实现者们可能以实现相关的方式提供了这些特性，但它们在标准中的缺失使得编写可移植的 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比 Common Lisp 使用更多的递归。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的话，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。

如果你学过 Scheme，你也应该当心 Scheme 和 Common Lisp 之间的许多细微差别可能会绊你一跤。这些差别也是 Common Lisp 和 Scheme 社区的狂热份子之间的一些长期宗教战争的导火索。日后随着我们讨论的深入我将指出其中最重要的那些差别。

另外两种仍然广泛使用的 Lisp 方言是 Elisp——Emacs 编辑器的扩展语言，以及 Autolisp——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 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，无论你是否已经确定要使用它或者只是想一窥其门径，那么本书就是为你而准备的。

如果你已经学会了一些 Lisp 但在跨越从学术训练到真实程序之间的鸿沟上遇到困难，那么本书刚好可以帮你走上正途。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，那么本书应该可以给你一些思路。或者你根本就没打算使用 Lisp——可能你已经确信 Lisp 并不比你知道的其他语言更好但却因为因为不熟悉它而无法反驳那些 Lisp 程序员。如果是这样的话，本书将给你一个对 Common Lisp 的直截了当的介绍。如果在你读完本书以后你仍然认为 Lisp 赶不上你当前喜爱的其他语言，那么你将有充分的理由来说明你的观点了。

我不但涉及到了该语言的语法和语义，而且还包括如何使用它来编写有用的软件。在本书的第一部分里，我将谈及语言本身，同时混入一些“实践性”章节，在那里我将展示如何编写真实的代码。接着，在我覆盖了该语言的绝大部分以后——包括某些在其他书里往往留给你自己去考

查的内容，本书的其余部分包括了九个更实用的章节，在那里我将帮助你编写一些中等尺度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 分类服务器提供一个 Web 接口。

当你读完本书以后，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，到那时你已经用 Common Lisp 写出了一些非平凡的程序，并且你将可以凭借自身力量继续探索该语言的其他部分了。尽管每个人的 Lisp 之路都是不同的，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

在本章里，你将学会设置你自己的编程环境并编写你的第一个 Common Lisp 程序。我们将使用由 Matthew Danish 和 Mikel Evans 开发的安装便利的 Lisp in a Box 环境，它打包了一个 Common Lisp 实现外加 Emacs——强大且 Lisp 友好的文本编辑器，以及 SLIME^①——构建在 Emacs 之上的 Common Lisp 开发环境。

上述组合提供了一个完整的 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，我将全程使用这个 Lisp in a Box 环境；而那些想要使用其他开发环境的人，无论是使用某些商业 Lisp 提供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什么大的困难。^②

2.1 选择一个 Lisp 实现

你要做的第一件事是选择一个 Lisp 实现。这对那些曾经使用诸如 Perl、Python、Visual Basic (VB)、C# 和 Java 这类语言的人们来说看起来可能有些奇怪。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 那样由仁慈的独裁者所控制的单一实现，也不像 VB、C# 和 Java 那样由单一公司控制的认可的实现。任何打算阅读标准并实现该语言的人都可以自由地去做。更进一步，对标准的改动必须在标准化组织美国国家标准化协会 (ANSI) 所控制的程序下进行。这一设计可以避免任何个体——例如某个厂商——任意修改标准。^③这样，Common Lisp 标准就是 Common Lisp 提供商和 Common Lisp 程序员之间的一份协议。

^① Superior Lisp Interaction Mode for Emacs

^② 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，使用一个可能基本理解 Lisp 的编辑器来编写 Lisp 程序将是件有趣的事情。至少，你需要一个可以帮你自动匹配括号并且知道如何自动缩进 Lisp 代码的编辑器。因为 Emacs 本身大部分是用一种叫做 Elisp 的 Lisp 方言写成的，所以它对编辑 Lisp 代码有着相当程度的支持。Emacs 也在很大程度上嵌入到了 Lisp 的历史和 Lisp 黑客的文化中：最初的 Emacs 和它的直接后裔，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，其命名方式跟随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。后来的版本使用了一个 ZWEI 的后裔，其使用了一个缺乏想象力的名字 ZMACS。

^③ 实际上来讲，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清理掉的误点，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清理掉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 (de facto) 来进行，就像是 Perl 和 Python

这个协议告诉你如果你写了一个程序，其中按照标准里描述的方式使用了某些语言特性，那么你就可以指望你的程序在任何符合标准的实现里产生同样的行为。

另一方面，标准可能并没有覆盖你的程序所涉及到的各个方面——某些方面故意没有定义以允许实现者在这些领域里继续探索：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，有时选择一种带有你所需要的额外特性的特定实现也是合理的。另一方面，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。对于编写那些大部分可移植但需要用到标准中没有定义的功能的代码，Common Lisp 提供了灵活的方式让你可以编写“条件依赖于”特定实现中的某些特性的代码。你将在第 15 章里看到这样的代码，在那里我们开发一个简单的库来平滑处理不同的 Lisp 实现在对待文件名上的区别。

尽管如此，目前一个实现最重要的特征是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评估版产品，能够运行在 Linux、Windows 和 OS X 上。试图寻找开源实现的人们有几种选择。SBCL^①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 OS X。SBCL 来源于 CMUCL^②——最早由卡内基梅隆大学开发的一种 Common Lisp——并且像 CMUCL 那样，大部分源代码都是公共域（public domain）的，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。^③对于 OS X 用户来说，OpenMCL^④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 OS X 的 Carbon 和 Cocoa 工具箱很好地集成。还有其他的开源和商业实现。参见第 32 章以获取更多的相关资源的信息。

本书中的所有 Lisp 代码除非特别说明否则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平滑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。

2.2 安装和运行 Lisp in a Box

由于 Lisp in a Box 软件包被设计成可以让新的 Lisp 程序员以最少的痛苦在一一流的 Lisp 开发环境上起步，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 Practical Common Lisp 的 Web 站点 <http://www.gigamonkeys.com/book/> 上获取对应的安装包，然后按照安装说明的提示来操作即可。

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事情的应用程序接口（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平滑那些语言没有定义的特性在各种实现之间的区别。

^① Steel Bank Common Lisp。

^② CMU Common Lisp。

^③ 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。但这个代码分叉现在已经是友好的了；bug 修复经常在两个项目之间传递，并且有讨论说某一天他们将会合并在一起。

^④ 译者注：OpenMCL 项目自从移植到 Mac 之外的平台后，更名为 Clozure CL。

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。也许最好的 Emacs 起步方法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，Emacs 用户使用一种记号来描述组合键以避免经常性地书写诸如“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——所谓的和弦——被写在一起然后用连接号（-）分隔。顺序按下的键或者和弦用空格分隔。在一个和弦里，C 代表 Ctrl 键而 M 代表 Meta 键（也就是 Alt）。这样，我们可以将刚才描述的那个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 还提供有大量在线文档，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，可以通过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，全部都绑定到了以 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。除了向导以外的两个最有用的帮助命令是 C-h k，可以让我们输入任何组合键然后告诉我们它所对应的命令是什么，还有 C-h w，可以让我们输入一个命令的名字然后告诉我们它对应着什么组合键。

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) 的概念。当你使用 Emacs 的时候，你所编辑的每个文件都将被表示成一个不同的缓冲区，其中任何给定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——无论你是在打字还是调用任何命令。缓冲区也被用来表示与诸如 Common Lisp 这类程序的交互。因此，你将用到的一个常见操作就是“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你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，绑定到了组合键 C-x b 上，使用时将在 Emacs 框架的底部提示你输入一个缓冲区的名字。当输入一个缓冲区的名字时，按 Tab 键将在你输入的字符基础上对其进行补全或者显示一个所有可能补全方式的列表。该提示同时建议了一个缺省缓冲区，你可以直接敲击回车 (Return) 来选择它。你也可以通过在 Buffers 菜单里选择一个缓冲区来切换缓冲区。

在特定的上下文环境下，其他组合键也可能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 可以切换到你跟 Lisp 进行交互的那个缓冲区。

2.3 放开思想：交互式编程

当你启动 Lisp in a Box 时，你应该可以看到一个带有类似下面的提示符的缓冲区：

```
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 的规则来求值它，然后打印结果。接着它再继续处理你输入的下一个表达式。这种无休止的读取、求值和打印的周期变化就是为什么它被称为读-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，你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。如果有必要的话，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可用的编程环境来。但如果追求真正的 Lisp 编程体验，你需要一个像 SLIME 这样的环境，它可以让你同时通过 REPL 和在编辑源文件时与 Lisp 进行交互。例如，你没有必要把一个函数定义从源文件里拷贝并粘贴到 REPL 里，也不必因为改变了一个函数就把整个文件重新加载；你的 Lisp 环境可以让我们求值或者编译单独的表达式和直接来自编辑器的整个文件。

2.4 体验 REPL

为了测试 REPL，你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，你可以输入 10 然后回车，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——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——REPL 中的 E——以后，它将求值到其自身。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 更有意义的东西那么事情就变得有意思一些了。比如说，你可以在 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 求值列表的时候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形成了该函数的参数。在本例里，符号 + 是加法函数的名字。2 和 3 求值到它们自身然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 也可能以其他方式求值列表，但我们现在还没有必要讨论它。让我们从 Hello World 开始。

2.5 Lisp 风格的“Hello, World”

没有“Hello, World”^①的编程书籍是不完整的。事实上，想让 REPL 打印出“hello, world”是极其简单的。

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

^① 声誉卓著的“Hello, World”在 Kernighan 和 Ritchie 的 C 语言书极大促进了它的流行以前就已经存在了。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》。（目前可以在线从 <http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html> 获得）

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求值到其自身的字符串对象，然后再以同样的语法打印出来。双引号本身不是字符串对象在内存中的一部分——它们只是语法，用来告诉读取器读入一个字符串。而打印机在打印字符串的时候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。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` 接收变长参数，但是只有两个必要的参数分别代表发送输出的位置以及一个字符串。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 ~，那么它就会被原样放出。如果你传递 `t` 作为第一个参数，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` 表达式看起来将会是这样：^①

```
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那个紧接着“hello, world”输出后面的那行里的 `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，一个 `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。但是 Lisp 中的每个表达式都会求值出某些结果。^②

尽管如此，在于你是否已经写出了一个真正的“程序”这点上恐怕仍有争议。不过你越来越接近了。而且你正在体会 REPL 所带来的自底向上的编程风格：你可以试验不同的方法然后从你已经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的表达式来做你想要的事，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，可以用类似下面这样的 `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-在内的在其他语言里非法的很多字符在 Common Lisp 里都是合法的。像 `hello-world` 这种用连字符而不是下划线 `hello_world` 或是内部大写 `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，在本例中为空因为函数不带参数。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

^① 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")
```

或是这个：

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

^② 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`。

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。不过，不像 FORMAT表达式，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。

一旦你定义了这个函数，你就可以像这样来调用它：

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

你将看到输出和你直接求值 FORMAT表达式的时候是一样的，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 保存你的工作

你可能会争辩说这就是一个完整的“hello, world”程序了。尽管如此，还有一个问题。如果你退出 Lisp 然后重启，函数定义将会丢失。写出了这么好的一个函数，你会想要保存你的工作。

很简单。你只需创建一个文件然后在里面把定义保存下来。在 Emacs 中你可以通过输入 C-x C-f 来创建一个新文件，然后根据 Emacs 的提示输入你要创建的文件的名字。你把文件放在哪里并不重要。Common Lisp 源文件习惯上带有.lisp 扩展名，尽管有些人用.cl 来代替。

一旦已创建了文件，你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，在你输入了开放的括号和单词 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你在使用哪个 Common Lisp 实现，但可能看起来像这样：

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

这个信息在你开始输入每一个新的列表元素时就会消失，但当你每次输入了空格以后又会重新出现。当你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。如果你输入回车然后按 Tab 键，SLIME 将自动把第二行缩进到合适的位置，就像这样：^②

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

SLIME 也会帮助匹配括号——当你输入了闭合的括号时，它将闪烁对应的开放括号。或者你也可以输入 C-c C-q 来调用命令 slime-close-parens-at-point，它将插入必要数量的闭合括号以匹配当前的所有开放括号。

现在你有几种方式将这个定义送给 Lisp 环境。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 做些修改，重新编译它然后回到 REPL，使用 C-c C-z 或者 C-x b，然后再次调用它。例如，你可以使其更加语法化。

^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。

^② 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。

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

接下来，用 C-c C-c 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
```

你将可能需要保存你的工作文件；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，你将需要退出并重启 Lisp 环境。退出的话你可以使用一个 SLIME 快捷键：在 REPL 中输入一个逗号。在 Emacs 窗口的底部，你将被提示输入一个命令。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。^①现在用 M-x slime 重启 SLIME。

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

天哪！发生了什么事？好吧，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——扔出一个异常并从栈上退回。而且它也绝对不会仅仅因为你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 把你扔进了调试器。

当身处调试器之中时，你仍然具有对 Lisp 的完全访问权限，所以你可以通过求值表达式来检查我们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>
```

^① SLIME 快捷键并不是 Common Lisp 的一部分——它们是 SLIME 的命令。

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，目前最重要的事情是知道你总是可以通过按 q 来退出它并回到 REPL 里。

回到 REPL 里你可以再试一次。问题在于 Lisp 不知道 `hello-world` 的定义。因此你需要让 Lisp 知道那个我们保存在 `hello.lisp` 文件中的定义。你有几种方式来做到这点。你可以切换回含有那个文件的缓冲区（使用 C-x b 然后在其提示时输入 `hello.lisp`）然后就像之前那样用 C-c C-c 重新编译那个定义。或者你可以加载整个文件，这对文件里含有大量定义时将是一个更加便利的方法，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 表示一切都成功加载了。^① 使用 `LOAD` 加载一个文件在本质上等价于以文件中出现的顺序逐个在 REPL 下输入每一个表达式，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时先用 `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，所以我们可以像下面这样进行编译和加载：

```
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` 还提供了不需要使用 REPL 来加载和编译文件的支持。当你在一个源代码缓冲区的时候，你可以使用 C-c C-1 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；你直接回车就可以了。或者你可以输入 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。当然我还没有覆盖到所有的技术和窍门，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在那里，增量地添加、重定义和测试它们的程序。

同样地，甚至当 Lisp 程序被部署以后，经常也仍然有一种方式可以进入 REPL。你将在第 26 章里看到如何使用 REPL 和 SLIME 来跟一个正在运行 Web 服务的 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——比如说

^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被带进调试器。如果发生了这样的事，多半可能是由于 Lisp 没有找到那个文件，而这可能是因为 Lisp 所认为的当前的工作目录和你文件所在的位置不一样。这种情况下，你可以用 q 退出调试器然后使用 SLIME 快捷命令 cd 来改变当前 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。

——像本地环境那样去调试一个远程服务器。

一个甚至更加令人印象深刻的案例是发生在 NASA 的 1998 年 Deep Space 1 号任务中的远程调试。在宇宙飞船升空一年以后，一点儿 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竟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 bug 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了。^①一个程序员用下面的话描述了这件事：

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。拥有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，被证明是无价的。

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>

第3章 实践：一个简单的数据库

很明显，在你可以用 Lisp 构建真实软件之前，你必须先学会这门语言。但是让我们面对现实——你可能在想，“‘Practical Common Lisp’ 难道不是反语吗？为什么你会被期望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呢？”于是我将从给你一个小型的可以用 Common Lisp 来做的例子开始。在本章里你将编写一个简单的数据库用来追踪 CD 光盘。在第 27 章里，当你为我们的流式 MP3 服务器构建一个 MP3 数据库时，你还会用到类似的技术。事实上，你可以把它视为 MP3 软件工程的一部分——毕竟为了有大量的 MP3 可听，跟踪你都拥有哪些 CD 以及哪些需要烧录将可能是有用的。

在本章里，我将刚好谈及足够你理解代码工作原理所需要的 Lisp 特性。但我将会解释许多细节。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覆盖这里用到的所有 Common Lisp 控制结构，以及更多。

一个术语方面的说明：我将在本章里讨论少量 Lisp 操作符。在第 4 章里你将学到 Common Lisp 提供了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，我将适时地在提及操作符时说成是函数或宏或特殊操作符，而不是简单地将细节隐藏在“操作符”这个词里。眼下你可以将函数、宏和特殊操作符看成是差不多等价的东西。^①

另外，请记住我不会为你这个最开始的后 “hello, world” 程序亮出所有最专业的 Common Lisp 技术来。本章的意图不在于表明你将如何用 Lisp 编写一个数据库；重点在于让你对 Lisp 编程有个大致的印象并且可以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

为了跟踪那些需要烧录成 MP3 的 CD，以及哪些 CD 应该先进行烧录，数据库里的每个记录将包含 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被烧录过的标记。因此，首先你将需要一种方式来表示一条单一的数据库记录（也就是一张 CD）。Common Lisp 给了你大量的数据结构可供选择——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。

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，

^① 尽管如此，在我正式开始之前，至关重要的一点是你必须忘记所有你知道的关于 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。

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)
```

你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。尽管如此，另一类列表——称为属性表（property list）或简称 plist——甚至更方便。属性表是一个这样的列表：从第一个元素开始的所有相间的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，称为关键字（keyword）符号。一个关键字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)
```

注意到你可以使用和你用来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。

真正令属性表成为表达数据库记录的便利方式的东西是函数 GETF，后者接受一个 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。Lisp 也有真正的哈希表，但 plist 足以满足你的当前需要并且在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
```

有了所有这些，你就可以轻易写出一个 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。函数的名字是 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：一个对 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。你需要一些更大的结构来保存记录。再一次，出于简化目的，使用列表看起来像是个好的选择。同样出于简化目的，你可以使用一个全局变量 *db*，

后者可以用 `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)
```

你可以使用 `PUSH` 宏为 `*db*` 添加新的项。但稍微做得抽象一些可能是个好主意，因此你应该定义一个函数 `add-record` 来给数据库增加一条记录。

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

现在你可以将 `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

无论何时你在 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))
```

尽管如此，这并不是一个令人满意的查看输出的方式。你可以写一个 `dump-db` 函数来将数据库转储成一个更加人类可读的格式，就像这样：

```
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
```

^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 章里，当你学会了更多的语言特性以后，你将可以构建一个更加灵活的数据库。在第 6 章里你也会看到，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。

该函数看起来像这样：

```
(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。对于每个 cd的值，你使用 FORMAT函数来打印它。

无可否认，这个 FORMAT调用多少有些晦涩。尽管如此，FORMAT并不比 C 或 Perl 的 printf 函数或者 Python 的 string-%操作符更复杂。在第 18 章里我将进一步讨论 FORMAT的细节。目前我们只需记住这个调用就可以了。正如你在第 2 章里看到的，FORMAT接受至少两个参数，第一个是它用来发送输出的流；t是流 *standard-output*的简称。

FORMAT的第二个参数是一个格式字符串，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接受大量的指令，每一个都有自己的选项集。^①尽管如此，目前我将只关注那些对你写 dump-db 有帮助的选项。

~a指令是美化指令；它的意图是消耗一个参数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，而字符串也不再有引用标记了。例如：

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

或是：

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

~t 指令用于格式化。~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 指令不消耗任何参数。

```
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，下一个被消耗的参数必须是一个列表。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消耗掉尽可能多的元素。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。~%指令并不消费任何参数而只是告诉 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 的数据之间产生一个空行。

从技术上来讲，你也可以使用 FORMAT在整个数据库本身上循环，从而将我们的 dump-db函

^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（为清楚起见做了折行处理）：

```
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
```

数变成只有一行。

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

这件事究竟算是酷还是可怕，完全取决于你的观点。

3.4 改进用户交互

尽管我们的 `add-record` 函数在添加记录方面工作得很好，但对于普通用户来说却显得过于 Lisp 化了。并且如果他们想要添加大量的记录，操作并不是非常方便。因此你可能想要写一个函数来提示用户输入一组 CD 的信息。这就意味着你将需要某种方式来提示用户输入一堆信息然后读取它们。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须的，为了确保 Lisp 在打印提示信息之前不会等待在一个换行上。

然后你可以使用名副其实的 `READ-LINE` 函数来读取单行的文本。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，调用 `READ-LINE` 所得到的值，也就是它所读取的字符串（不包括结尾的换行符）。

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 记录的函数。

```
(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，它们应该是一个数字和一个布尔值。取决于你想要一个多专业的用户接口，花在验证用户输入的数据上的努力可以是无止境的。目前让我们倾向于一个快速而肮脏的办法：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` 可以让其适当地容忍一些：

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

但这里还有一个问题：如果无法在所有垃圾里找出一个整数的话，PARSE-INTEGER将返回 NIL 而不是一个整数。为了保持这个快速而肮脏的思路，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，依次求值它们，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）。所以你可以使用下面这个：

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

来得到一个缺省值 0。

修复 Ripped 提示的代码就更容易了。你只需使用 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 会在你输入了没有以 y、Y、n，或者 N 开始的东西时重新提示你输入。

将所有这些东西放在一起你就可以得到一个相当健壮的 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，你可以通过将 prompt-for-cd 包装在一个不停循环直到用户完成的函数里来搞定这个“添加大量 CD”的接口。你可以使用 LOOP 宏的简单形式，其不断执行一个表达式体直到通过一个对 RETURN 的调用来退出。例如：

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

现在你可以使用 add-cds 来添加更多的 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
```

3.5 保存和加载数据库

有一个便利的方式来给数据库添加新记录是件好事。但如果用户不得不在每次退出并重启 Lisp 以后重新输入所有记录的话他们是绝对不会高兴的。幸运的是，借助你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以后重新加载。下面是一个 `save-db` 函数，其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，绑定流到一个变量，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求值其表达式体的时候出了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。它包括用来保存你将要在 `WITH-OPEN-FILE` 主体中写入的文件流的变量名，一个必须是文件名的值，以及一些控制文件如何打开的选项。这里你用 `:direction :output` 指定了你正在打开一个用于写入的文件，以及 `:if-exists :supersede` 说明了当存在同名的文件时你想要覆盖已存在的文件。

一旦已经打开了文件，所有你需要做的就是使用 `(print *db* out)` 将数据库的内容打印出来。跟 `FORMAT` 不同的是，`PRINT` 将 Lisp 对象打印成一种可以被 Lisp 读取器读回来的形式。宏 `WITH-STANDARD-IO-SYNTAX` 可以确保影响 `PRINT` 行为的特定一些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` 的参数应该是一个含有用户打算用来保存数据库的文件名的字符串。字符串的确切形式将取决于他们正在使用的操作系统。例如，在一个 Unix 系统上他们可能会这样调用 `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))
```

在 Windows 下，文件名可能是像 “c:/my-cds.db” 或 “c:\\my-cds.db” 一样的东西。^①

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

将数据加载回数据库的函数也是差不多的样子。

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

^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。这是个很方便的特性，因为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。

这次你不需要在 `WITH-OPEN-FILE` 的选项里指定 `:direction` 了，因为你要的是默认的 `:input`。并且与打印相反，你使用函数 `READ` 来从流中读入。这是与 REPL 所使用的相同的读取器，可以读取任何你可以在 REPL 提示里输入的 Lisp 表达式。尽管如此，在这种情况下，你将只是读取和保存表达式，并不会对它求值。再一次，`WITH-STANDARD-IO-SYNTAX` 宏可以确保 `READ` 使用和 `save-db` 在 `PRINT` 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 `*db*` 变量将含有从文件中读取的对象，也就是由 `save-db` 所写入的那些列表的列表。你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。因此如果你已经用 `add-record` 或者 `add-cds` 添加了尚未用 `save-db` 保存的记录，你将失去它们。

3.6 查询数据库

现在你有了保存和重载数据库的方法以及一个便利的用户接口用来添加新的记录，这样很快你就会有足够的记录以至于你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。你需要的是一种查询数据库的方式。例如，你可能会想要用类似这样的东西：

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

然后就可以得到所有艺术家为 Dixie Chicks 的记录的列表。再一次，看来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列表，然后返回一个仅包含原来列表中匹配该谓词的所有元素所组成的新列表。换句话说，它删除了所有不匹配该谓词的元素。尽管如此，`REMOVE-IF-NOT` 并没有真的删除任何东西——它会创建一个新列表，而不会去碰原来的列表。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且返回布尔值的函数——除了 `NIL` 表示假以外其余的都表示真。

举个例子，假如你需要从一个数字组成的列表里解出所有偶数来，你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)
```

这里的谓词是函数 `EVENP`，后者当其参数是偶数时返回真。那个有趣的 `#'` 记号是“得到接下来的名字所对应的函数”的简称。没有 `#'` 的话，Lisp 将把 `evenp` 作为一个变量的名字来对待并查找该变量的值，而不是函数。

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)))
```

它会检查其参数时候模 2 等于 0 (换句话说, 就是偶数)。如果你想要用匿名函数来解出所有的奇数, 你可以这样写:

```
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)
```

注意到 `lambda` 并不是函数的名字——它是一个表明你正在定义匿名函数的指示器。^① 尽管如此, 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, 然后再是函数体。

为了用 `REMOVE-IF-NOT` 从数据库里选出所有 Dixie Chicks 的专辑, 你将需要一个可以在一条记录的艺术家字段是 "Dixie Chicks" 时返回真的函数。请记住我们选择 `plist` 来表达数据库的记录是因为函数 `GETF` 可以从 `plist` 里解出给定名称的字段来。因此假设 `cd` 是一个保存着数据库单一记录的变量的名字, 那么你可以使用表达式 `(getf cd :artist)` 来解出艺术家的名字来。函数 `EQUAL` 当用于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"。所有你需要的就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))
```

现在假设你想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。你可以写成这样:

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

注意这个匿名函数的代码直到其被 `REMOVE-IF-NOT` 调用才会运行, 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, 它还让你写出了一个这样的函数: 其部分含义——`artist` 的值——继承自它所在的上下文环境。

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。你可 以 编 写 其 他 几 个 的 函 数 , 诸 如 `select-by-title`、`select-by-rating`、`select-by-title-and-artist`, 诸如此类。但他们之间除了匿名函数的内容以外就没有其他区别了。代替地, 你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参数。

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

但是 `#'` 哪里去了? 好吧, 这种情况下你并不希望 `REMOVE-IF-NOT` 去使用一个名为 `selector-fn` 的函数。你想要它使用的是一个作为 `select` 的参数传递到变量 `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")))
```

^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。

```
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

但是这样看起来相当杂乱无章。幸运的是，你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在 `artist-selector` 返回以后将不会有存在的变量——至少看起来是这样。^①它现在可能看起来有些奇怪，但它实际上确实可以按照你所想象的方式来工作——如果你带参数 "Dixie Chicks" 调用 `artist-selector`，那么你将得到一个可以匹配其 `:artist` 字段是否为 "Dixie Chicks" 的 CD 的匿名函数，而如果你用 "Lyle Lovett" 来调用它，你将得到一个匹配 `:artist` 字段是否为 "Lyle Lovett" 的不同的函数。所以现在你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))
```

现在你只需要更多的函数来生成选择器了。但正如你不想编写 `select-by-title`, `select-by-rating` 等等诸如此类的东西那样——因为它们非常相似——你也不会想去写一大堆长得差不多的选择器函数的生成器，每个字段写一个。为什么不写一个通用性的选择器函数生成器呢，一个根据你传递给它的参数可以生成用于不同字段甚至其组合的选择器函数？你可以写出这样一个函数来，不过首先你需要一个关于一种称为关键字行参(*keyword parameter*)的语言特性的速成课程。

在目前写过的函数里，你使用的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际参数上。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。但有时你可能想要编写一个可以用任何数量的参数来调用的函数。关键字形参就是其中一种实现方式。一个使用关键字形参的 `foo` 版本可能看起来是这样的：

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

唯一的区别在于参数列表的开始处有一个 `&key`。尽管如此，对这个新的 `foo` 的调用方法将是截然不同的。下面这些都是合法的调用，同时在 → 的右边给出了相应的结果。

```
(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)
```

正如这些示例所显示的那样，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`。我将跳过关于关键字参数如何指定以及它们与其他类型参数的关系等诸多细节，不过你还需要知道其中一点。

^① 一个引用了其外围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。

正常情况下如果一个函数在没有为特定关键字形参传递参数的情况下被调用时，该形参的值将为 `NIL`。尽管如此，有时你可能想要区分作为参数显式传递给关键字形参的 `NIL` 和作为缺省值的 `NIL`。为了实现这点，当你在指定一个关键字形参时你可以将那个简单的名称替换成一个包括参数名、缺省值和另一个称为 `supplied-p` 参数的列表。这个 `supplied-p` 参数将被设置成真或假，具体取决于一个参数在特定的函数调用里是否真的作为关键字形参被传递了。下面是一个使用了该特定的 `foo` 版本：

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

现在前面给出的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——如果你熟悉 SQL 数据库的话就会逐渐明白为什么叫它 `where` 了——是一个函数，其接受对应于我们的 CD 记录字段的四个参数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东西来：

```
(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，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 注意到你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”。

^① 注意到在 Lisp 中，一个 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。

3.7 更新已有的记录——WHERE 的再次使用

现在你有了完美通用化了的 `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——更新特定记录的方法了。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，尤其是当你已经有了一个 `where` 子句生成器时。事实上，`update` 函数只是一些你已经见过的一些思路的再次应用：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。主要的新东西是对 `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*)))
```

这里的另一个新东西是 `SETF` 用在了诸如 `(getf row :title)` 这样的复杂形式上。我将在第 6 章里讨论 `SETF` 更加深入的细节，但目前你只需知道它是一个通用的赋值操作符，可被用于在各种“位置”而不仅仅是变量上进行赋值。（`SETF` 和 `GETF` 具有相似的名字这点完全是巧合——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`，其紧跟着属性名 `:title` 后面的那项将具有变量 `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，那么你可以求值下列形式：

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

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))
```

你甚至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` 正相反；它返回一个所有确实匹配谓词的元素都被删掉的列表。和 `REMOVE-IF-NOT` 一样，它不会实际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

^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。但我还没准备好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

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。^①

不过这里仍然有一些讨厌的代码重复。并且其实你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：如果你想要改变它的行为，你不得不改动它多个拷贝。并且如果你改变了 CD 的字段，你将必须添加或删除 `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` 函数的本意是去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` 参数甚至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用来检查 `title` 和 `where` 的其他关键字形参甚至是否被设置了。^② 如果你真的想要移除所有这些运行期检查，你可以通过一个程序将所有这些你调用 `where` 的位置以及究竟传递了哪些参数都找出来。然后你可以替换每一个对 `where` 的调用，使用一个只做必要的比较的匿名函数。举个例子，如果你发现这段代码：

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

你可以将其改为：

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

注意到这个匿名函数跟 `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你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。

你可能会想象把你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。但你也可能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可能就是非常值得的了。

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

语言。被`*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——至少这段代码里没有。

^① 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。”“你是说你无法将所有细节都记到脑子里？”“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。”

^② 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，考虑到检测一个变量是否为 `NIL` 将是非常便宜的。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。

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，然后说明你可以怎样把 `where` 函数替换成一个 `where` 宏。在我开始写这个示例宏之前，我需要快速介绍一个新函数：`REVERSE` 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))` 求值到 `(3 2 1)`。现在让我们创建一个宏。

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

一个函数与一个宏的主要词法差异在于你需要用 `DEFMACRO` 而不是 `DEFUN` 来定义一个宏。然后一个宏定义包括一个名字，就像函数那样，一个形参列表，以及一个表达式体，这两个也像函数。尽管如此，一个宏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？当 REPL 开始求值这个 `backwards` 表达式时，它认识到 `backwards` 是一个宏的名字。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 随后将这个列表转给 `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` 然后将这个值传回给 REPL，然后在最初表达式的位置上再被求值。

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——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")`。

那么这些东西对于 `where` 里的代码重复又有什么帮助呢？情况是这样：你可以写出一个宏，它在每个特定的 `where` 调用里只生成你确切需要的代码。再一次，最佳的思路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` 调用中引用的字段你都有一个下列形式的表达式：

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

那么让我们来写一个函数，给定一个字段的名字和一个值，返回这样一个表达式。由于一个表达式本身只是一个列表，你可能认为你可以写成类似这样：

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

尽管如此，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，它假设这是一个变量的名字并去查找它的值。这对

^① 宏也可以由解释器来运行——尽管如此，当你考虑编译的代码时更容易理解宏的要点。和本章里的所有其他内容一样，我将在未来的章节里提及关于这点的更多细节。

于 `field` 和 `value` 来说是对的；这正是你想要的。但是它也会同样对待 `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，它将做你想要的事：

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

你可以在 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来做这件事。你真正需要的是一种方式来书写表达式，其中的大部分都是不需要求值的，然后可以有某种方式提取出的其中的你确实想求值的少数表达式来。然后，当然了，确实存在这样一种方法。一个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)
```

尽管如此，在一个反引用表达式里，任何以逗号开始的子表达式是被求值的。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，你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))
```

现在如果你回过头来看那个手工优化的选择器函数，你可以看到函数体是由每字段/值对一个比较表达式组成的，全部包装在一个 `AND` 表达式里。假设现在你想让 `where` 宏的所有参数排成一列传递进来。你将需要一个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` 的结果的函数。为了实现这个函数，你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))))
```

完整的关于 `LOOP` 的讨论将不得不等到第 22 章；目前只需了解这个 `LOOP` 表达式刚好做了你想要的事：当 `fields` 列表的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` 宏做了与你往 `*db*` 中添加记录时所使用的 `PUSH` 宏相反的操作。

现在你需要将 `make-comparisons-list` 所返回的列表包装在一个 `AND` 和一个匿名函数里，这件事可以由 `where` 宏本身做到。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“,”的变种“,@”。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)
```

你也可以使用“,@”在一个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参数列表中对 `&rest` 的使用。和 `&key` 一样，`&rest` 改变了参数被解析的方式。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参数，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。因此如果你像这样调用 `where`：

```
(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`，其返回一个比较表达式所组成的列表。你可以通过使用函数 `MACROEXPAND-1` 来精确地看到一个 `where` 调用将产生出哪些代码。

如果你传给 `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。因此你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))
```

它可以工作了。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，有趣的事情已经发生了。你不但去除了重复还使得代码同时更有效和更通用了。这通常就是正确选用宏所达到的效果。这件事说得过去是因为宏只不过是另一种创建抽象的手法——词法层面的抽象，以及按照定义用通过更简明地表达底层一般性的方式所得到抽象。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` 与其字段的。事实上，我们新的 `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。你可能会想到大量需要增加的特性，包括支持多表或是更复杂的查询。在第 27 章里我们将建立起一个部分吸收这些特性的 MP3 数据库。

本章的要点在于给你一个对少量 Lisp 特性的快速介绍以及它们如何用来编写出比 “hello, world” 更有趣一点儿的代码。在下一章里我们将开始一个更加系统的 Lisp 概述。

第4章 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内容对你已经用到的那些特性来一次更加系统化的考察。我将从对 Lisp 的基本语法和语义元素的概述开始，它们是——当然了，我必须先要说清楚这些东西是什么……

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。两者最明显的特征区别在于前者对括号和前缀表示法的大量使用。出于无论什么原因，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；另一方面，Lisp 的追随者们认为，Lisp 的语法是它的最大优势。为什么在两个团体之间会有如此不同的见解呢？

我不可能在本章里完整描述 Lisp 的语法，除非我更彻底地解释了 Lisp 的宏，但是我可以从一些宝贵的历史经验开始——保持开放的思想是值得的：当 John McCarthy 首次发明 Lisp 时，他倾向于实现一种类 Algol 的语法，后者他称之为 M-表达式。尽管如此他从未实现这一点。他在他的文章《History of Lisp》^① 中解释了这一点：

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，它只不过是被推迟到无限遥远的未来罢了。新一代的[相比 S-表达式]更加偏爱类 FORTRAN 或类 Algol 表示法的程序员也许会最终实现它。

换句话说，在过去 45 年来实际使用 Lisp 的人们，已经喜欢上了这个语法并且发现它使该语言变得更强大。在接下来的几章里，你将开始看到为什么会这样。

4.2 打开黑箱

在我们观察 Lisp 的语法和语义之前，值得花一点时间看看它们是如何定义的以及它和许多其他语言的语法有怎样的不同。

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——都是按照黑箱来操作的：你将一系列表示程序文本的字符送进黑箱，然后它——取决于是解释器还是编译器——要么

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

执行预想的行为，要么产生一个编译版本的程序并在它运行的时候执行这些行为。

当然，在黑箱的内部，语言的处理器通常划分成子系统，各自负责将程序文本转换成具体行为或目标代码这项任务的一部分。一个典型的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，后者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——称为抽象语法树——随即被送进一个求值器，后者要么直接解释它要么将其编译成包括机器码在内的某种其他语言。由于语言处理器是一个黑箱，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。

在 Common Lisp 中，分工有点不一样，无论是从实现者的角度还是从语言如何定义的角度都是这样。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，另一个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

每一个黑箱定义了一个语法层面。读取器定义了字符串如何被翻译成称为 S-表达式^② 的 Lisp 对象。由于 S-表达式语法中包括任意对象以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 表达式，更不用说所有字符的序列都是合法的 S-表达式了。举个例子，(`foo 1 2`) 和 (`("foo" 1 2)`) 都是 S-表达式，但只有前者可以是一个 Lisp 形式，因为一个以字符串开始的列表作为 Lisp 形式没有意义。

这样的黑箱划分方法带来了一系列后果。其中之一是你可以使用 S-表达式，正如第 3 章里你所看到的那样，作为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 另一个后果是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，真正的优势在于你可以生成用来处理已有数据的代码。这就是 Lisp 宏的本意，我将在后续章节里讨论更多细节。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。

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

^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用来执行的编译器。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所读取到的对象。通常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。

的由空格所分隔的元素。原子是所有其他的东西。^①列表元素本身也可以是 S-表达式（换句话说，原子或嵌套的列表）。注释——从技术来讲它们不是 S-表达式——以分号开始，扩展到一行的结尾，本质上将被当作空白处理。

这就差不多了。由于列表在句法上如此简单，其余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。在本节里我将描述大多数常用类型原子的规则：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 形式被求值的。

数字的表示是相当直接的：任何数位的序列——可能前缀一个标识 (+或-)，还有一个十进制点或者斜杠，或是以一个指数标记结尾——将为读取为一个数字。例如：

```
123 ; 整数一百二十三
3/7 ; 比值七分之三
1.0 ; 缺省精度的浮点数一
1.0e0 ; 同一个浮点数另一种写法
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不同类型的数字：整数、比值和浮点数。Lisp 也支持复数，它们有其自己的表示方法，我将在第 10 章里讨论它们。

正如这些示例所显示的那样，你可以用多种方式来表示同一个数字。但无论你怎样书写它们，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，代表 -2/8 或 246/2 的对象跟代表 -1/4 或 123 的对象并没有什么不同。相似地，1.0 和 1.0e0 只是同一个数字的不同写法。另一方面，1.0、1.0d0 和 1 可能代表不同的对象，因为不同精度的浮点数和整数都是不同的类型。我将把关于不同类型的数字的特征的细节留给第 10 章。

字符串，正如你在前面章节看到的那样，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字符，使其无论是什么都被包含在字符串里。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。所有其他的字符都可以无需转义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。读取器对于一个名字的用途毫不知情——无论其究竟是一个变量或是函数的名字或者其他什么东西。读取器只是读取一个字符的序列并构造出代表这个名字的对象。^②几乎任何字符都可以出现在名字里。不过空白字符不可以，因为列表的元素是用空格来分隔的。数位可以出现在名字里，

^① 空列表，()，也可写成 NIL，既是原子也是列表。

^② 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。

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所组成的名字。有十个字符被用于其他的句法目的，因此不能出现在名字里：开放和闭合的小括号、双引号和单引号、反引号、逗号、冒号、分号、反斜杠以及竖线。而就算这些字符如果你愿意转义它们的话也可以成为名字的一部分，只需将它们用反斜杠转义起来或是将含有需要转义的字符的名字用竖线包起来。

这种读取器将名字转化成对象的方式有两个重要的特征，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` 读取成同一个符号 `FOO`。尽管如此，`\f\o\o` 和 `|foo|` 将都被读成 `foo`，这是和符号 `FOO` 不同的另一个对象。这就是为什么当你在 REPL 中定义一个函数的时候，它会打印出被转化成大写形式的这个函数的名字。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。^①

为了确保同一个文本名字总是被读取成相同的符号，读取器将 `intern` 这个符号——在它读取名字并将其全部转化成大写以后，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。如果它无法找到一个，它将创建一个新符号并添加到表里。否则，它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，同一个对象将被用来表示它。^②

因为在 Lisp 中名字可以包含比源自 Algol 的语言更多的字符，命名约定在 Lisp 中也相应地有所不同，诸如像 `hello-world` 这类带有连字符的名字的使用。另一个重要的约定是全局变量的名字在开始和结尾处带有“*”。类似地，常量的名字以加号“+”开始和结尾。而某些程序员将特别底层的函数命名为以%甚至%%开始的名字。语言标准所定义的名字只使用字典字符（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提及它们。目前的关键是要理解怎样用数字、字符串和符号借助括号所组成的列表来构建 S-表达式从而表达任意的树状对象。一些简单的例子看起来像这样：

```
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
(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
```

一个稍微更复杂的例子是下面这个四元素列表，其含有两个符号、空列表、另一个列表——其本身含有两个符号和两个字符串：

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

^① 事实上读取器的大小写转化行为可以定制的，但是要想理解何时以及怎样改变它需要在相关的名字、符号和其他程序元素中相对我已经涉及到的内容做更加深入的讨论。

^② 我将在第 21 章里讨论符号和包之间的关系的进一步细节。

4.4 作为 Lisp 形式的 S-表达式

在读取器转换了大量文本成为 S-表达式以后，这些 S-表达式随后可以作为 Lisp 代码被求值。或者只有它们中的一些——不是每一个读取器可以读的 S-表达式都有必要作为 Lisp 表达式来求值，Common Lisp 的求值规则定义了第二层的语法规则来检测哪个 S-表达式可以作为 Lisp 形式^① 来对待。这一层面的语法规则相当简单。任何原子——非列表或空列表——都是一个合法的 Lisp 形式，正如任何首元素为符号的列表那样。^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。出于讨论的目的你可以将求值器想象成一个函数，其接受一个句法良好定义的 Lisp 形式作为参数并返回一个值，后者我们称其为这个形式的值。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：符号和所有其他的东西。一个符号在作为形式被求值时被视为一个变量的名字并且求值到该变量的当前值上。^③ 我将在第 6 章里讨论变量是如何得到它们的值的。你也应该注意到特定的“变量”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。这意味着当这样一个表达式被传递给假想的函数时，它会简单地直接返回自身。第 2 章里，当你在 REPL 里键入 `10` 和 `"hello, world"` 时你已经见到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价值。两个以这种方式定义的常量是 `T` 和 `NIL`，所谓的真值和假值。我将在“真相、谎言和等价”那一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——名字以冒号开始的符号。当读取器 `intern` 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。当我们开始考虑列表的求值方式时，事情变得更加有趣起来了。所有合法的列表形式均以一个符号开始，但是三种类型的列表形式以三种相当不同的方式进行求值。为了检测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否是一个函数、宏或者特殊操作符的名字。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——它被假设成一个函数的名字。^④

^① 当然其他层面的纠错也存在于 Lisp 中，如同其他语言那样。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。

^④ 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，而拥护者们总是无休止的争论哪种更好。

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

函数调用的求值规则很简单，求值作为 Lisp 形式的列表的其余元素并将结果传递到命名函数中。这一规则明显的放置了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，一个函数调用形式基本语法是像下面这样的，其中每个参数其本身也是一个 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，除了在求值参数 (+ 1 2) 和 (- 3 4) 时需要先求值它们的参数再应用相应的函数到它们身上：

```
(* (+ 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，用于各类事务的函数在其他语言里往往有着特别的语法。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

就是说，并非所有的操作都可以被定义成函数。由于一个函数的所有参数在函数被调用之前都将被求值，因此没有办法写出一个类似你在第 3 章里用到的 IF 操作符那样的函数。为了看到这点，考虑下面这个形式：

```
(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求值其参数表达式。符号 x 将被作为产生某个值的变量来求值；然后 (format t "yes") 将被求值成一个函数调用，在向标准输出打印“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结果才被传递给 IF，而这对于想要控制两个 FORMAT 表达式中的哪一个被求值来说已经太晚了。

为了解决这个问题，Common Lisp 定义了一些称为特别操作符的东西，IF 就是其中之一，它们可以做到函数所无法做到的事情。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别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。

^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。

`IF`的规则相当简单：求值第一个表达式。如果它求值到非 `NIL`，那么求值下一个表达式并返回它的值。否则，返回求值第三个表达式的值或者 `NIL`如果第三个表达式被省略的话，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，然后是 `then-form` 和 `else-form` 中的一个或另一个。

一个更简单的特殊操作符是 `QUOTE`，其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，下面的东西求值到列表 `(+ 1 2)`，而不是值 3：

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

这个列表没有什么特别的；你可以像任何你用 `LISP` 函数所创建的列表那样处理它。^①

`QUOTE` 被用得相当普遍，以致于它的一个特别的语法被内置到读取器之中。除了像下面这样的写法：

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

你也可以这样写：

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

该语法是读取器所理解的 S-表达式语法的稍稍扩展。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。^②

一般来说，特殊操作符所实现的语言特性需要求值器做出某些特殊处理。例如，有些特殊操作符修改了其他形式将被求值的环境。其中之一，也是我将在第 6 章详细讨论的，是 `LET`，其用来创建新的变量绑定。下面的形式将求值到 10，因为第二个 `x` 被求值于一个环境，其中带有该名字的变量被 `LET` 以值 10 所建立：

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

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。另一方面，宏给了语言的用户一种扩展其语法的方式。如同你在第 3 章里看到的那样，宏是一个接受 S-表达式作为其参数的函数，并返回一个 Lisp 形式然后在宏形式的位置上进行求值。一个宏形式的求值过程包括两个阶段，首先，该宏形式的元素不经求值而传递到宏函数里。其次，由宏函数所返回的形式——称为其展开式（expansion）——按照正常的求值规则进行求值。

在你头脑中清醒地认识到一个宏形式求值的两个阶段将尤为重要。当你在 REPL 中输入表达

^① 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 `LISP` 来创建。

^② 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。当大多数程序员都在谈论该语言的语法扩展时，他们则在谈论正规的宏，我将在稍后讨论它们。

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，这两个阶段发生在完全不同的时间，因此重要的是对于何时发生什么保持清醒。例如，当你使用函数 `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地展开，直到代码中不再包含除宏调用形式和特殊形式之外的东西。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。尽管如此，编译后的代码直到文件被加载才会被执行。因为宏在编译期生成其展开式，它们可以做相对大量的工作来生成其展开式而无需在文件被加载或是文件中定义的函数被调用时付出额外的代价。

由于求值器在传递宏形式给宏函数之前并不求值它们，因此它们不需要是形态良好的 Lisp 形式。每一个宏都为其宏形式中的符号表达式指定了一种含义，包括它将如何使用它们生成展开式。换句话说，每一个宏都定义了它们自己的局部语法。例如，来自第 3 章的 `backwards` 宏定义了一种语法，其中一个表达式是合法的 `backwards` 形式，当且仅当其作为一个列表的逆是一个合法的 Lisp 形式。

我将在本书中相当多地提到宏，眼下对你来说重要的是认识到宏——尽管跟函数调用在句法上相似——用于相当不同的目的，提供了一种嵌入编译器的钩子。^①

4.8 真、假和等价

最后两个你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接了当的：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，可被用于需要返回一个非 `NIL` 值却没有其他东西可用时。关于 `NIL` 唯一麻烦的一点是它唯一一个既是原子又是列表的对象：除了用来表示假以外，它还被用来表示空列表。^② 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。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的名字，所以表达式 `nil`、`()`、`'nil` 以及 `'()` 全部求值到同样的东西——未经求值的形式将被求值到对值为符号 `NIL` 的常值变量的引用，而在引用的形式里 `QUOTE` 特殊操作符将直接求值

^① 缺少或没有 Lisp 宏使用经验的人们以及更有甚者被 C 的预处理器所荼毒过的人们，当他们意识到宏调用跟正常函数调用一样时可能会紧张。出于某些原因这在实践中并不是一个问题。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))
```

或是这样：

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

就好像 `DOLIST` 是一个函数。一个好的 Lisp 环境将自动的正确的格式化宏调用，甚至对于用于定义的宏。

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` 组合考虑，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。

^② 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`，后者无论跟符号 `nil` 还是空列表都是不同的值，并且即便后两者也是不同的。

到那个符号。出于同样的理由，`t` 和 `'t` 也将求值到同样的东西：符号 `T`。

使用诸如“同一种东西”这样的术语理所当然地会引申到关于两个值“等价”的这个问题的含义上。在后面的章节里你将看到，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。在本节里我将讨论四个“通用”等价谓词——可以被传递任何两个 Lisp 对象然后当它们等价时返回真否则返回假的函数。按照介绍的顺序，它们是 `EQ`、`EQL`、`EQUAL` 和 `EQUALP`。

`EQ` 用来测试“对象标识”，当两个对象是同样的东西时才是 `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。因此，`EQ` 可能认为带有相同值的两个数字和两个字符是等价的，或者不等价。具体实现有足够的余地将表达式 `(eq 3 3)` 合法地求值为真或假。进一步说，`(eq x x)` 可以被求值为真或假，当 `x` 的值是一个数字或一个字符。

因此，你不该将 `EQ` 用于比较可能是数字或者字符的值上。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，但是如果你切换语言实现它将不保证以相同的方式工作。而切换实现可能意味着简单地把你的实现升级到一个新版本——如果你的 Lisp 实现者改变了它们表达数字或字符的方式，那么 `EQ` 的行为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，除了它也可以保证当相同类型的两个对象代表相同的数字或字符值的时候将是等价的。因此，`(eq 1 1)` 被确保是真的。而 `(eq 1 1.0)` 被确保是假的，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：“有可能时就使用 `EQ`” 阵营认为当你知道你不在比较数字或字符时就应该使用 `EQ`，因为 (a) 这是一种说明你不在比较数字或字符的方式，以及 (b) 它将会稍微更有效率，因为 `EQ` 不需要检查它的参数是否为数字或字符。

“总是使用 `EQL`” 阵营认为你永远不该使用 `EQ`，因为 (a) 潜在获得的代码清晰性丢失了，因为每次有人阅读你的代码时——包括你在内——看到了一个 `EQ`，他们将停下来检查是否它被正确使用了（换句话说，它永远不该被用来比较数字或字符），以及 (b) `EQ` 和 `EQL` 之间的效率差异相比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，因为它们操作在所有类型的对象上，但是它们不像 `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，允许不同的对象被认为是等价的，由这些函数所实现的特殊含义的等价性没有什么特别的，除了它们曾经被过去的 Lisp 程序员认为是有用的。如果这些谓词不满足你的需要，你总是可以定义你自己的谓词函数，以你所需要的方式来比较不同类型的对象。

`EQUAL` 相比 `EQL` 的宽松之处在于它将具有递归相同结构和内容的列表视为等价的。`EQUAL` 也认为含有相同字符串是等价的，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谓词类型。对于所有的其他类型，它回退到 `EQL` 的水平上。

^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` 除非另外的谓词被明确提到了。因此，如果你要在技术上 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。

`EQUALP` 和 `EQUAL` 相似，除了它甚至更加宽松。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。它还认为两个字符是等价的如果它们只在大小写上有区别。数字在 `EQUALP` 下面是等价的，只要它们表达相同的数学意义上的值。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，带有 `EQUALP` 元素的数组也是 `EQUALP` 等价的。和 `EQUAL` 一样，有一些为尚未涉及到的数据类型可被 `EQUALP` 视为等价但不会被 `EQL` 或 `EQUAL` 通过。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的结构，这样你就不需要通过数据括号来查看你究竟写到哪儿了。一般而言，每一个新的嵌套层次都需要多缩进一点儿，并且如果折行是必须的，位于同一个嵌套层次的项应当按行对齐。这样，一个需要跨过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放括号缩进两个空格。就像这样：

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

尽管如此，你不需要太担心这些规则，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可以相对容易的知道应当如何缩进。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，在每行的开始处按 Tab 将导致该行被适当的缩进，或者你可以通过将光标放置在一个开放的括号上并键入 C-M-q 来重新缩进整个表达式。或者你也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。

确实，有经验的 Lisp 程序员们倾向于依赖于他们的编辑器来自动处理缩进，这样不但可以确保他们的代码美观，还可以检测笔误：一旦你熟悉了代码应该如何被缩进，那么一个错误放置的括号将立即由于你编辑器所给出的奇怪缩进而被发现。例如，假设你在编写一个应该看起来像这样的函数：

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

现在假设你不小心忘记了 `test` 后面的闭合括号。因为你不会去数据括号，你很可能会在 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，如果你一直都在每行的开始处按 Tab 来缩进的话，你将不会得到那样的代码。相反你将得到如下的代码：

```
(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，你将立即看出有错误发生。

另一个重要的格式化规则是闭合的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。这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，但是一旦你的代码被正确缩进，那么括号的意义就不存在了——没有必要通过将他们分散在多行来使其得到特别的关注。

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。

;;; 一个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。
    ;; 注意到该注释处在与其所应用的代码的相同的缩进上。
    (some-function-call)
    (another i)                      ; 本注释仅用于此行
    (and-another)                    ; 这个也是一样
    (baz)))
```

现在你可以开始了解 Lisp 的主要程序构造的进一步细节了：函数、变量和宏。下一章：函数。

第5章 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是函数、变量和宏。在第 3 章里构建数据库时，你已经用到了所有这三个组件，但是我跳过了大量关于它们如何工作以及如何更好使用它们的细节。我将把接下来几章献给这三个主题，从函数开始——就像在其他语言里那样，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。语言标准中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。我将在第 16 章里介绍它们。

并且，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——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。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*)
```

任何符号都可以被用作函数名。^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，其他字符也被允许使用。例如，一个将字符串转换成物件 (widget) 的函数可能被叫做 `string->widget`。最重要的一个命名约定是在第 2 章里提到的那个，就是说你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` 是比 `frob_widget` 或 `frobWidget` 更好的 Lisp 风格的名字。一个函数的形参列表定义了那些将被用来保存函数被调用时所传递的

^① 尽管函数对于 Common Lisp 很重要，但将其描述成函数型语言并不真的合适。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——McCarthy 引进了许多被认为对函数型编程非常重要的思想——也有其突出的地位，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，使其相比诸如 Haskell 和 ML 这类语言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一个函数的名字，其后果将是未定义。尽管如此，正如你将在第 21 章所看到的那样，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。

参数的变量。^①如果函数不带有参数，则该列表就是空的，写成`()`。不同类型的形参分别处理必要的、可选的、多重的，以及关键字参数。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，它应该是一个用来描述函数用途的文档字符串。当函数被定义时，该文档字符串将被关联到函数的名字上，并且以后可以通过 DOCUMENTATION 函数来获取。^②

最后，一个 DEFUN 的主体由任意数量的 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 特殊操作符可被用于在函数的任何位置立即返回，我将很快开始讨论它。

在第 2 章里我们写了一个 hello-world 函数，它看起来像这样：

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

现在你可以分析该程序的各部分了。它的名字是 `hello-world`；它的形参列表为空，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")
```

下面是一个稍微更复杂一些的函数：

```
(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`，其接受两个参数并绑定到形参 `x` 和 `y` 上，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

关于函数名称或文档字符串就没有更多的可说了，而本书的其余部分将用来描述所有你可以在一个函数体里做的事情，因此就只剩下形参列表需要讨论了。

很明显，一个形参列表的基本用途是为了声明那些即将用来接收传递给函数的参数的变量。当一个形参列表是个由变量名所组成的简单列表时——如同在 `verbose-sum` 里那样——这些形参被称为必要形参。当一个函数被调用时，它的每一个必要形参都必须被提供一个参数。每一个形参被绑定到对应的参数上。如果一个函数以过少或过多的参数来调用的话，Lisp 将报错。

尽管如此，Common Lisp 的形参列表也给了你更灵活的将函数调用参数映射到函数形参的方式。除了必要形参以外，一个函数还可以有可选形参。或是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

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

^② 例如，下面的：

```
(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，并被允许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。

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缺省值。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，它的初始尺寸就无关紧要了——如果从一个正确的观点来看的话。但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，可能会通过设置特定的初始尺寸来改进其程序的性能。尽管如此，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；不关心它的调用者们将得到一个合理的缺省值，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，在必要参数的名字之后放置符号`&optional`，后接可选形参的名字。一个简单的例子看起来像这样：

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

当该函数被调用时，参数被首先绑定到必要形参上。在所有必要形参都被赋值以后，如果还有任何参数剩余，它们的值将被赋值给可选形参。如果参数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参数被传递给函数——在本例中介于二到四之间——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` 的缺省值。你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缺省值。这个表达式将只有在调用者没有传递足够的参数来为可选形参提供值的时候才会被求值。通常情况是简单的提供一个值作为表达式：

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

上述函数要求一个参数来绑定到形参 `a` 上。第二个形参 `b` 将在有第二个参数时使用其值，否则使用 10。

```
(foo 1 2) -> (1 2)
(foo 1)   -> (1 10)
```

尽管如此，有时你可能需要更灵活地选择缺省值。你可能想要基于其他形参来计算缺省值。

^① 在那些不直接支持形参的语言里，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，这种在函数和其调用者之间的协议完全是自组织的——在某些函数或某些参数中 `NULL` 可能是一个可区分值，而在其他函数或参数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，可选参数也可以通过提供多个具有相同名称但不同参数个数的方法(`method`)来模拟，并且当一个方法使用较少的形参来调用时，会以缺省值代替缺少的参数去调用“真实”的那个方法。

而你真的可以——缺省值表达式可以引用早先出现在形参列表中的形参。如果你正在编写一个返回矩形的某种表示的函数并且你想要使它可以特别方便地产生正方形，你可以使用一个像这样的参数列表：

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

这将导致 `height` 形参除非明确指定否则将带有和 `width` 形参相同的值。

有时知道一个可选参数的值究竟被调用者明确指定还是使用了缺省值将是有用的。除了通过代码来检查形参的值是否为缺省值(这样做有时无效，假如调用者碰巧显式传递了缺省值)以外，你还可以通过在形参标识符的缺省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参数时被绑定到真值，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

可选形参仅用于当你有一些离散的调用者可能想也可能不想提供值的形参的场合。但是某些函数需要接收可变数量的参数，一些你已经见过的内置函数就是以这种方式工作的。`FORMAT`有两个必要参数，流和控制串。但在这两个之后，它还需要一组可变数量的参数，取决于控制串需要插入多少个值。“`+`”函数也接受可变数量的参数——没有特别的理由限制它只能在两个数之间相加；它将做任意个数的值的加法（它甚至可在零个参数上工作，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，你可以通过简单的给它一些可选形参来写出一个接受可变数量参数的函数。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，你将不得不使用相当于一个合法的函数调用可以传递的参数数量那么多的可选形参。这一数量是具体实现相关的但可以保证至少有 50 个。在当前的所有实现中，它的范围从 4096 到 536870911。^①汗，这种歪曲头脑的乏味事情绝对不是 Lisp 风格。

相反，Lisp 允许你在符号 `&rest` 之后包括一个一揽子形参。如果一个函数带有一个 `&rest` 形参，任何满足了必要和可选形参之后的其余所有参数将被收集到一个列表里成为该 `&rest` 形参

^① 常量 `CALL-ARGUMENTS-LIMIT` 将告诉你这个实现相关的值。

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

尽管可选和剩余形参给了你很多灵活性, 但两者都不能帮助你应对下面的情形: 假设你有一个接受四个可选参数的函数。现在假设在函数被调用的多数场合理, 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——它们只需传递一个可选参数, 然后忽略其他就好了。但是所有其他的调用者将不得不为它们所不关心的一到三个参数传递一些值。难道这真的是可选形参被设计用于解决的问题吗?

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, 这将导致前三个可选形参对于该调用者来说变成了必要形参。幸运的是, 另一种形参类型, 关键字形参, 可以允许调用者指定具体哪个形参使用哪个值。

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。这里是一个只有关键字形参的函数:

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

当这个函数被调用时, 每一个关键字形参将被绑定到紧跟在同名的关键字后面的那个值上。回顾第 4 章里提到的, 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参数列表中, 那么对应的形参将被赋予其默认值, 如同可选形参那样。因为关键字参数是标签化的, 所以它们在必要参数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)
(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缺省的值形式以及一个 `supplied-p` 变量名。在关键字和可选形参中, 这个缺省的值形式都可以引用那些更早出现在形参列表中的形参。

同样, 如果出于某种原因你想要关键字的调用者指定一个与实际形参名不同的形参, 你可以将形参名替换成一个列表, 其含有当调用函数时所使用的关键字以及用作形参的名字。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)
```

这种风格在你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，通常是因为你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。

5.6 混合不同的形参类型

虽然罕见，但在单一函数里使用所有四种类型的形参也是可能的。无论何时，当超过一种类型的形参被用到时，它们必须以这样的顺序被声明：首先是必要形参，其次是可选形参，再次是剩余形参，最后是关键字形参。尽管如此，在使用多种类型形参的函数中，典型情况是你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` 和 `&rest` 形参。其他两种组合方式，无论 `&optional` 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` 和 `&key` 形参组合使用时将产生非常奇怪的结果，因此你也许应该避免将其一起使用。问题出在如果一个调用者没有为所有可选形参提供值时，那么没有得到值的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` 和 `&key` 形参：

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

如果像这样被调用的话，它将工作得很好：

```
(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。在那一点上，Lisp 将期待要么一个成对的关键字/值，要么什么也没有，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` 将在毫无指示的情况下得到缺省值 `NIL`。

一般而言，如果你发现自己正在编写一个同时使用 `&optional` 和 `&key` 形参的函数，你可能应该将它变成全部使用 `&key` 形参的形式——它们更灵活，并且你总是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。你也可以移除关键字形参，只要没人正在使用它们。^① 一般而言，使用关键字形参将会使代码相对易于维护和发展——如果你需要为函数添加一些需要

^① 有四个标准函数同时接受 `&optional` 和 `&key` 参数——`READ-FROM-STRING`、`PARSE-NAMESTRING`、`WRITE-LINE` 和 `WRITE-STRING`。它们在标准化的过程中出于跟早期 Lisp 方言向后兼容的目的被原样保留了下来。`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`)这样的调用看起来将会忽略该`: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。

用到新参数的新行为，你可以直接添加关键字形参而无需修改或甚至重新编译任何调用该函数的已有代码。

你可以安全地组合使用 `&rest` 和 `&key` 形参，但其行为初看起来可能会有一点奇怪。正常来讲，无论 `&rest` 还是 `&key` 出现在形参列表中都将导致所有出现在必要和 `&optional` 形参之后的那些值被特别处理——要么被作为 `&rest` 形参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))
```

你将得到如下结果：

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

5.7 函数返回值

目前你写出的所有函数都使用了默认的返回值行为，即最后一个被求值的表达式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。

尽管如此，有办法从函数中间返回，尤其是当你想要从嵌套的控制结构中脱身时，某些时候将是非常便利的。在这种情况下，你可以使用 `RETURN-FROM` 特别操作符立即以任何值从函数中间返回。

你将在第 20 章里看到 `RETURN-FROM` 事实上不只用于函数；它被用来从一个由 `BLOCK` 特别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，求值一个带有当前函数名和你想要返回的值的 `RETURN-FROM` 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”是其想要返回的代码块名。该名字不被求值，因此无需引用。

下面这个函数使用了嵌套的循环来发现第一个数对——每个都小于 10，并且其成绩大于函数的参数，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，不得不指定你正在返回的函数名多少有些不便——其中之一就是如果你改变了函数的名字，你将需要同时改变 `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，都会求值到一个值。因此在实践中这不是什么问题。

^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。

5.8 作为数据的函数——高阶函数

尽管你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是很有用的。例如，如果你可以将一个函数作为参数传给另一个函数的话，你将可以写出一个通用的排序函数，其允许调用者提供一个用来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）也需要依赖保存代码引用从而以后运行的能力。由于函数已经是一种对代码比特进行抽象的标准方式，因此允许函数被视为数据也是合理的。^①

在 Lisp 中，函数只是另一种类型的对象。当你用 `DEFUN` 定义一个函数时，你实际上做了两件事：创建一个新的函数对象以及赋予其一个名字。如同你在第 3 章里看到的，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，无论是有名的还是匿名的，都只是一些二进制数据——以原生编译的 Lisp 形式存在，可能大部分由机器码所构成的。你唯一需要知道的就是如何保持它们以及当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。它接受单一参数并返回带有该名字的函数。这个名字是不被引用的。因此如果你像这样来定义一个函数 `foo`：

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

你可以像这样得到对应的函数对象：^②

```
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 `FUNCTION` 的语法糖，正如 “`,`” 是 `QUOTE` 的语法糖一样。^③ 因此你也可以像这样得到 `foo` 的函数对象：

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

一旦你得到了函数对象，你就只剩下一件事可做了一——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`。^④ 它们的区别仅在于如何获取传递给函数的参数。

`FUNCALL` 用于在你编写代码时确切知道你打算传递给函数的参数数量的场合。`FUNCALL` 的第一个参数是被调用的函数对象，其余的参数被传递到该函数中。因此，下面两个表达式是等价的：

^① 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。

^② 一个函数对象的确切打印表示形式将随着不同的实现而有所不同。

^③ 思考 `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。

^④ 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。

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

不过，用 `FUNCALL` 来调用一个当你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` 的一个更有建设性的用法。它接受一个函数对象作为参数，并使用参数函数在 `min` 和 `max` 之间以 `step` 为步长的返回值来绘制一个简单的 ASCII 艺术条形图：

```
(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个星号。

注意到你不需要使用 `FUNCTION` 或 `#'` 来得到 `fn` 的函数值；你需要其被解释成一个变量，因为它是作为函数对象的变量的值。你可以用任何接受单一数值参数的函数来调用 `plot`，例如内置的函数 `EXP`，其返回以 `e` 为底的其参数的指数值。

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

然而，当参数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，其含包括一个函数对象，一个最小和最大值以及一个步长。换句话说，这个列表包含了你想要作为参数传给 `plot` 的所有的值。假设这个列表保存在变量 `plot-data` 中。你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))
```

这样可以用，但仅仅为了将参数传给 `plot` 而显式将其解开，看起来相当讨厌。

这就是为什么需要 `APPLY`。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参数。它将函数作用在列表中的值上。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”的参数跟作为列表的最后一个参数一起使用。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` 并不关心其所应用的函数是否接受 `&optional`、`&rest` 或是 `&key` 参数——由任何孤立

参数和最后的列表所组合而成的参数列表必须是一个合法的参数列表，其对于该函数来说必须带有足够的参数用于所有必要的形参和适当的关键字形参。

5.9 匿名函数

一旦你开始编写或是简单地使用那些可以接受其他函数作为参数的函数，你将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是相当恼人的，尤其是当你从不用名字来调用它时。

当用 `DEFUN` 来定义一个新函数看起来不必要时，你可以使用一个 `LAMBDA` 表达式来创建“匿名”的函数。如同第 3 章里讨论的那样，一个 `LAMBDA` 表达式看起来像这样：

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

一种考虑 `LAMBDA` 表达式的方式，是将其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。这解释了为什么你可以在一个函数名的位置上使用一个带有`#'`的 `LAMBDA` 表达式。

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

你甚至可以在一个函数调用中使用 `LAMBDA` 表达式作为函数的名字。如果你想要的话，你可以更简洁地书写前面的 `FUNCALL` 表达式：

```
((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的场合是合法的。^①

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

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

并随后将其传给 `plot`：

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

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

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` 表达式本身并非可以求值的表达式。这就是说，`LAMBDA` 并非一个函数、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被识别成一种函数名。

但如果这仍然是真的，那么`(funcall (lambda (...) ...))`将是不合法的，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。尽管如此，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，一个 `LAMBDA` 宏被定义出来，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中时，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`#'`。

```
CL-USER> (plot #'double 0 10 1)
**
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
NIL
```

但如果写成这样将会更简单和清晰：

```
CL-USER> (plot #'(lambda (x) (* 2 x)) 0 10 1)
**
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
NIL
```

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。

第6章 变量

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）^①变量。这两种变量类型分别对应于其他语言中的“局部”和“全局”变量，不过只能说是大致相似。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 Common Lisp 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。

许多含有有变量的表达式都可以同时使用词法和动态变量，这使得事情变得有些令人困惑。因此我将从讨论同时涉及到两种类型的 Lisp 变量开始。然后再谈及词法和动态变量各自的特征。随后我将讨论 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地方赋予新值。

6.1 变量基础

和其他语言一样，Common Lisp 中的变量是一些可以保存值的命名位置。尽管如此，在 Common Lisp 中，变量并非像 Java 和 C++ 等语言那样带有确定的类型，这就是说你不需要为每一个变量声明其可以保存的对象的类型。相反，一个变量可以保存任何类型的值，并且变量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，假如你传递了某个并非数字的对象给了“+”函数，那么 Common Lisp 将会报类型错误。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——没有办法将一个对象作为其不属于的类型的实例来对待。^③

在 Common Lisp 中所有的值从概念上讲都是对象的引用。^④因此，将一个变量赋予新值会改

^① 动态变量出于某种你将在本章后面看到的原因，有时也被称为特殊变量（special variable）。你需要注意这两个同义词，因为某些人（包括 Lisp 实现本身）使用其中一种术语而其他人使用另外一种。

^② 早期的 Lisp 倾向于使用动态变量作为局部变量，至少当其解释执行代码时。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，而它的 my 变量——由 Perl 5 所引入——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，因为语言语法规则合并了赋值与绑定）

^③ 事实上说所有类型错误将总是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变量，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，这类声明通常被用于在代码被开发和调试之后进行优化，而不是在正常开发的期间。

^④ 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。而其他对象将被表达成一个实际对象的指针。尽管如此，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拷贝也就无关紧要了，这就是在第 4 章里所讨论的 `EQ` 和 `EQL` 的根本区别。

变该变量所指向的对象，而对之前被引用的对象没有影响。尽管如此，如果一个变量保存了对一个可变对象的引用，那么你可以使用该引用来修改此对象，而改动将被任何带有相同对象的引用的代码所看到。

一种你已经用到的引入新变量的方式是定义函数形参。正如前一章里你所看到的那样，当你用 `DEFUN` 来定义一个函数时，形参列表定义了当函数被调用时用来保存形参的变量，例如下面这个函数定义了三个变量——`x`、`y` 和 `z`——用来保存其参数：

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

每当一个函数被调用时，Lisp 就会创建新的绑定来保存由函数调用者所传递的参数。绑定代表了一个变量在运行期的存在。单个变量——你可以在程序的源代码中所指出的那种东西——在程序运行的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个函数调用中都被重新绑定。

和所有 Common Lisp 变量一样，函数形参也可以保存对象引用。^①因此，在函数体内你可以为一个函数形参赋予新值，而这将不会影响到同样函数的另一个调用所创建的绑定。但如果传递给函数的对象是可变的，而你在函数中改变了它，那么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形式是使用 `LET` 特殊操作符。一个 `LET` 形式的结构看起来像这样：

```
(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` 形式将三个变量 `x`、`y`、`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，所有的初值形式将首先被求值，然后新的绑定被创建，并在形式体被执行之前初始化到适当的初值上。在 `LET` 形式体中，变量名将引用到新创建的绑定。在 `LET` 形式结束后，这些名字将引用 `LET` 之前它们所引用的无论什么东西，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` 所引入的变量将在每次进入 `LET` 时都被重新绑定^②。

函数形参和 `LET` 变量的作用域——变量名可被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——函数定义或 `LET`——被称为绑定形式。正如你很快将看到的，两种类型的变量——词法的和动态的——使用两种略有不同的作用域机制，但对于两者来说，其

^① 用编译器作者的话来说，Common Lisp 函数是“传值的”，尽管如此，被传递的值是对象的引用。这跟 Java 和 Python 的工作方式相似。

^② `LET` 形式和函数形参中的变量是以完全相同的手法被创建。事实上在某些 Lisp——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，下面的：

```
(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)
```

作用域都被界定在绑定形式之内。

如果你嵌套绑定了引用同名变量的形式，那么最内层的变量绑定将覆盖外层的绑定。例如，当下面的函数被调用时，一个形参 `x` 的绑定将被创建用来保存函数的参数。然后第一个 `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，这次带有初始值 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))
```

每一个对 `x` 的引用将指向最小的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将被解除覆盖，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
```

在后面的章节里，我将讨论其他可作为绑定形式使用的程序构造——任何引入了一个新的变量名并只在该构造中使用的程序构造都是一个绑定形式。

例如，在第 7 章里你将遇到 `DOTIMES` 循环，一个基本的计数循环。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` 的变种：`LET*`。两者的区别在于，在一个 `LET` 中，被绑定的变量名只能被用在 `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。因此你可以写成下面这样：

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

但却不能这样写：

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

尽管如此，你也可以通过嵌套的 `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。词法作用域应该被那些曾经使用 Java、C、Perl 或者 Python 来编程的人们所熟悉，因为他们都提供词法作用域的“局部”变量。这样说起来，Algol 程序员们也该对其感到自然，因为 Algol 在 1960 年代首先引入了词法作用域。

尽管如此，Common Lisp 的词法变量是带有一些变化的词法变量，至少和最初的 Algol 模型相比是这样的。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来自其所封闭作用域之内的词法变量的引用时，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))))
```

LAMBDA 形式中对 count 的引用根据词法作用域规则应该是合法的，而那个含有引用的匿名函数将被作为 LET 形式的值返回，并通过 FUNCALL 被不在 LET 作用域之内的代码所调用。这样就会发生什么呢？正如你将看到的那样，当 count 是一个词法变量时，它可以正常工作。当控制流进入 LET 形式时，所创建的 count 绑定将被按需保留。在本例中只要某处保持了一个对 LET 形式的所返回的一个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，还可以对其赋予可在闭包调用之间持久化的新值。例如，你可以像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)))))
```

然后每次当你调用它时，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（特殊）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，这就是为什么大多数现代语言将词法作用域用于局部变量。尽管如此，有时你真的想要一个全局变量——一个你可以从程序的任何位置访问到的变量。尽管随意地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一种编程语言里。^①正如你即将看到的，Lisp 的全局变量和动态变量都更加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，一个初始值和一个可选的文档字符串。在被 `DEFVAR` 和 `DEFPARAMETER` 定义以后，该名字可被用于任何位置来指向全局变量的当前绑定。如同你在前面章节里所看到的，全局变量习惯上被命名为以“*”开始和结尾的名字。你将在本节的后面看到遵守该命名约定的重要性。`DEFVAR` 和 `DEFPARAMETER` 的示例像这样：

```
(defvar *count* 0
  "Count of widgets made so far.")
(defparameter *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。一个 `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。这样一个变量被称为未绑定的（*unbound*）。

从实践上来讲，你应该使用 `DEFVAR` 来定义某些变量，其含有你想要保持并且甚至当你改变了使用该变量的源代码时也要保持的数据，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。使用 `DEFVAR` 来定义 `*count*` 变量是合适的，因为目前已生产的部件数量不会因为你对部件生产的代码做了某些改变而就此作废。^②

另一方面，假设变量 `*gap-tolerance*` 对于部件生成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许值，并且改变了 `DEFPARAMETER` 形式中的值，你将需要在你重新编译和加载文件的时候让这一改变产生效果。

在用 `DEFVAR` 和 `DEFPARAMETER` 定义了一个变量之后，你可以从任何一个地方引用它。例如，你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你不必到处传递它们，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——你永远不知道什么时候你会想要向标准输出打印东西，并且你不会想要每一个函数都不得不接受并传递含有那些流的参数只为了日后有人可能会用到它们。

^① Java 将全局变量伪装成公共静态字段，C 使用 `extern` 变量，而 Python 的模块及变量和 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又 `DEFVAR` 定义的变量，那你要么使用 `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。

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你已经编写了引用那个全局变量的代码。那么如果想要临时修改这些代码的行为就只能去改变那个变量的值了。

例如,假设你正工作在一个程序上,其中含有某些底层日志函数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,你想要将所有这些函数所生成的输出捕捉到一个文件里。你可以打开一个文件并且得到的流赋予 `*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,直到当你完成工作时忘记了将 `*standard-output*` 设置回最初的流上。如果你忘记了重设 `*standard-output*`,那么程序中所有用到 `*standard-output*` 的其他代码也会将把它们的输出发往该文件。^①

看起来你真正想要的是一种类似于下面所描述的包装一片代码的方式:“从这里以下的所有代码——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——应该为全局变量 `*standard-output*` 使用这个值”。然后当上层的函数返回时 `*standard-output*` 的旧值应该被自动恢复。

这看起来正像是 Common Lisp 的另一种变量——动态变量——所做的事。当你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。与一个词法绑定——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,一个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 并且看起来事实上所有全局变量都是动态变量。

这样如果你想要临时重定义 `*standard-output*` 做法就是简单地重新绑定它,比如说使用一个 `LET`:

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

在任何因为上述 `stuff` 调用而运行的代码中,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*` 绑定将随之消失,并且接下来对 `*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。概念上讲,一个给定的动态变量的新绑定,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。^③

一个简单的例子显示了它是怎样工作的:

^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打印到不同的流上,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,但那样你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打印时它将不得不锁住所有其他线程直到完成,甚至当它们想要打印到一个不同的流上。

^② 一个绑定可被引用到的时间间隔,其技术语称为其生存期,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,但却有动态的生存期。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)。

^③ 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。

```
(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，如果你从最上层调用 `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
```

但你也可以用 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))
```

注意到中间那个对 `foo` 调用被包装在一个将 `*x*` 绑定到新值 20 的 LET 形式中。当你运行 `bar` 时，你将得到这样的结果：

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

正如你所看到的，第一个对 `foo` 的调用看到了全局绑定带有其值 10。然而，中间的那个调用却看到了新的绑定带有其值 20。但是在 LET 之后 `foo` 再次看到了全局绑定。

和词法绑定一样，赋予新值仅影响当前绑定。为了看到这点，你可以重定义 `foo` 来包含一个对 `*x*` 的赋值。

```
(defun foo ()
(format t "Before assignment~18tX: ~d~%" *x*)
(setq *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`，你将看到这样的结果：

```
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
```

注意到 `*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` 是怎样知道当它绑定 `*x*` 时它打算创建一个动态绑定而不是一个词法绑定呢？它知道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或是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——如果你使用了一个你以为它是词法变量而它却刚好是全局特殊的变量的话就很不好了。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，如果你总是按照“*”命名约定来命名全局变量，你将不会在打算建立词法绑定时意外使用了一个动态绑定。

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，你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。其他代码可以局部的声明一个名字为特殊的，从而指向该动态绑定。尽管如此，局部特殊变量相对较少使用，所以你不需要担心它们。^②

动态绑定使全局变量更加易于管理，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，并且它也打开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。你应该只有在你需要利用这两个特征之时才使用动态变量。

6.4 常量

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

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

与 `DEFVAR` 和 `DEFPARAMETER` 相似，`DEFCONSTANT` 在其使用的名字上产生了一个全局效果——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+开始或结尾的名字来表示常量，这一约定在

^① 这就是为什么动态变量有时也被称为特殊变量（special variable）。

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

某种程度上不像全局特殊名字的“*”命名约定那样流行，但出于某些原因是个好主意。^①

关于 DEFCONSTANT 另一点需要注意的是，尽管语言允许你通过重新求值一个带有一个初始值形式的 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，多数实现将要求你重新求值任何引用了该常量的代码以便看到新的值，因为老的值可能已经被内连到代码中了。因此最好只用 DEFCONSTANT 来定义那些真正是常量的东西，例如 π 的值。对于那些你可能想改变的东西，你应当代替使用 DEFPARAMETER。

6.5 赋值

一旦你创建了一个绑定，你可以对它做两件事：获取当前值以及为它设置新值。正如你在第 4 章里所看到的，一个符号求值到它所命名的变量的值上，因此，你可以简单地通过引用一个变量来得到它的当前值。为了给一个绑定赋予新值，你要使用 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 是一个宏，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，当位置上是一个变量时，它展开成一个对特殊操作符 SETQ 的调用，后者作为一个特殊操作符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*，你可以写成这样：

```
(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，下面函数中的 SETF：

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

对于 *foo* 之外的任何值都没有效果，这个当 *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无论什么值。特别地，一个类似下面的形式：

```
(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)
```

你也可写成：

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

^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 来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。

SETF 返回最近被赋予的值，因此你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，变量绑定不是唯一可以保留值的位置，Common Lisp 支持复合数据结构，包括数组、哈希表和列表以及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。

我将在未来的章节里讨论那些数据结构，但在目前我们所讨论的赋值主题下，你应该知道 SETF 可以为任何位置赋值。当我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。简短来说，如果你需要对一个位置赋值，SETF 差不多就是你要使用的工具。甚至有可能扩展 SETF 以使其可以为用户定义的位置赋值，尽管我将不会涉及这些内容。^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。

<i>Table 6-1. Assignment with = in Other Languages</i>			
Assigning to...	Java, C, C++	Perl	Python
...variable	<u>x = 10;</u>	<u>\$x = 10;</u>	<u>x = 10</u>
...array element	<u>a[0] = 10;</u>	<u>\$a[0] = 10;</u>	<u>a[0] = 10</u>
...hash table entry		<u>\$hash{'key'} = 10;</u>	<u>hash['key'] = 10</u>
...field in object	<u>o.field = 10;</u>	<u>\$o->{'field'} = 10;</u>	<u>o.field = 10</u>

SETF 以同样的方式工作——SETF 的第一个“参数”是用来保存值的位置，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。^② 因此，表 6-1 中赋值语句的 Lisp 等价形式——给定 `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)
```

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

^② 源自 Algol 的使用“=”左边的“位置”和右边的新值进行赋值的语法，其广泛使用产生了术语“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”。

注意到用 `SETF` 赋值一个作为更大的对象一部分的位置，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。再一次，这跟 “=” 在 Java、Perl 和 Python 中的行为非常相似。^①

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，但特定的模式当需要基于当前值来赋予新值时，由于经常使用因此有它们自己的操作符。例如，尽管你可以像这样使用 `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。相反你可以使用宏 `INCF` 和 `DECREF`，其以缺省为 1 的特定数量递增和递减一个位置。

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

`INCF` 和 `DECREF` 是一类称为修改宏（modify macro）的宏的例子，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` 写出的同样修改语句更加简洁。更进一步，修改宏所定义的方式使其可以安全地使用那些表达式必须只被求值一次的位置。一个很傻的例子是下面这个表达式，其递增一个数组中任意元素的值：

```
(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：

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

尽管如此，它不会正常工作，因为两次对 `RANDOM` 调用不一定返回相同值——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一个不同的元素上。尽管如此，上面的 `INCF` 表达式将产生正确的行为，因为它知道如何处理这个表达式：

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

来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，它将很可能展开成差不多等价于下面这样的某种东西：

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

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

^① 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，类似于重定向的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，成员拷贝——“=” 的行为是相当特异的。

右的顺序。

宏 `PUSH`——在微型数据库中你曾用来向 `*db*` 变量添加元素——是另一个修改宏。你将在第 12 章里当我谈到列表在 Lisp 中如何表示时，进一步地观察它和它的伙伴 `POP` 和 `PUSHNEW` 是如何工作的。

最后，两个稍微有些难懂但很有用的修改宏是 `ROTATEF` 和 `SHIFTF`。`ROTATEF` 在位置之前旋转它们的值。例如，如果你有两个变量 `a` 和 `b`。下面的调用：

```
(rotatef a b)
```

将交换两个变量的值并返回 `NIL`。由于 `a` 和 `b` 是变量并且你不需要担心副作用，因此前面的 `ROTATEF` 的表达式等价于下面这个：

```
(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旋转它们——最后一个参数提供了一个值用来移动到最数第二个参数上，而其他的值将向左移动一个，第一个参数的最初的值将被简单地返回。这样，下面的：

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

将等价于——再一次，假设你不关心副作用——下面这个：

```
(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。

学会了 Common Lisp 函数和变量的基础以后，现在你将开始学习一个令 Lisp 始终区别于其它语言的特性：宏。

第7章 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——已经被吸收进其他语言，但 Lisp 的宏系统却始终使它得以保持其语言风格的特殊性。不幸的是，“宏”这个字在计算领域可以描述很多东西，和 Common Lisp 的宏相比仅具有模糊和大致的相似性。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为了理解 Lisp 的宏，你真的需要重新看待它，不能带有任何基于其他碰巧叫做宏的东西所带来的成见。现在让我们退一步，从观察各种语言支持扩展的不同方式来开始对 Lisp 宏的讨论。

所有的程序员应该都持有一种观点，一种语言的定义可以包括带有各种功能的标准库，后者是用“核心”语言来实现的——各种功能可以被任何程序员在语言之上实现，只要它还没有被定义成标准库的一部分。例如 C 的标准库就差不多可以完全移植 C 来实现。类似地，Java 的标准 Java 开发包（JDK）中所提供的日益增长的类和接口集合，也是用“纯” Java 所编写的。

使用一个核心加上一个标准库的方式来定义语言的一个优势是使其易于理解和实现。但真正的好处在于其可表达性——由于你所认为的“该语言”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 语言中不含有你所需要的用来做某件事的一个函数，那你就写出这个函数，然后你就得到了一个特性更加丰富一点的 C 版本。类似地，在诸如 Java 或 Smalltalk 这类几乎所有有趣部分都由类所定义的语言里，通过定义新的类你就可以扩展该语言，使其更适合用于编写你正试图编写的无论什么程序。

尽管 Common Lisp 支持所有这些扩展语言的方法，宏还提供了另一种方式。如同我在第 4 章里简要讨论的，每一个宏定义了其自己的语法，决定那些被传递的 S-表达式如何被转换成 Lisp 形式。有了宏作为核心语言的一部分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诸如 DEFUN 和 DEFPARAMETER 这样的定义形式——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给了你另一种扩展语言的方式，使其成为用于表达你特定编程问题的解决方案的更好语言。

现在，拥有另一种扩展语言的方式，其好处可能被认为是显而易见的。但出于某些原因大量没有实际使用过 Lisp 宏的人——那些日复一日不加思考地将时间花在创建新的函数型抽象或是定义类的层次体系来解决他们编程问题的人们——被这种可以定义新的句法抽象的思想给吓到

^① 想要看到这类误解的大致情况，可以在相对长期的 Usenet 新闻组上以 macro 为主题搜索任何在 comp.lang.lisp 和 comp.lang.* 之间交叉投递的线索。一个大致的说法是像下面这样：

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 好是因为它的宏么？！但宏是可怕和有害的；因此 Lisp 也一定是可怕和有害的。”

了。这种宏恐惧症的通常原因看起来多半是由于来自其他“宏”系统的不良经历。简单地对未知事物的恐惧无疑也扮演了其中一个角色。为了避免触发任何宏恐惧症反应，我将从讨论由 Common Lisp 所定义的几种标准控制构造宏开始缓慢进入该主题。这些东西都是那些如果 Lisp 没有宏就将不得不构造在语言核心里的东西。当你使用它们的时候，你不必关心它们被定义为宏，但是它们提供了一个你可以用宏来做某些事的极好示例。^①在下一章里我将说明如何定义你自己的宏。

7.1 WHEN 和 UNLESS

正如你所见到的，条件执行的最基本形式——如果 *x*，那么 *y*；否则 *z*——是由 IF 特殊操作符提供的，其基本形式是：

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

condition 被求值，如果其值是非 NIL，那么 *then-form* 会被求值并返回其结果。否则，如果有 *else-form* 的话，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 *then-form* 和 *else-form* 的每个被限制在必须是单一 Lisp 形式上。这意味着如果你想在每个子句中执行一系列操作，你必须将其包装在其他一些语法里。举个例子，假如在一个垃圾过滤程序中，你想要在一个消息是垃圾时同时将其标记为垃圾并更新垃圾数据库，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。因此你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)))
```

这样并不算太坏。但假如你将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。你可能会问你自己：“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，很快你将注意到这种由一个 IF 加上一个 PROGN 所组成的模式，并且希望可以有一种方式来抽象掉所有细节而不是每次都得将它们写出来。

这就是为什么要有宏。在这个案例中，Common Lisp 提供了一个标准宏 WHEN，可以让你写成

^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、DEFVAR 等等。在第 24 章里你将定义你自己的定义性宏，从而允许你可以简洁地编写用来读写二进制数据的代码。

这样：

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

但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，这里用到了我在第 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是意图。但是它们的极度简单性却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，你可以写出像 WHEN 和 UNLESS 这样简单的宏来获得虽小但却重要的清晰性，并随后通过不断地使用而无限放大。在第 24、26 和 31 章里，你将看到宏是如何被更大尺度地用于创建完整的领域相关嵌入式语言。但首先让我们完成对标准控制构造宏的讨论。

7.2 COND

当你具有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 那么 x，否则如果 b 那么 y；否则 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)))
```

并且，如果你需要在 *then* 子句中包括多个形式时——需要用到 PROGN——事情会变得更糟。因此毫不奇怪地，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：

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

主体中的每个元素代表一个条件分支，并由一个列表所构成，后者含有一个条件形式，以及零或多个当该分支被选择时将被执行的形式。这些条件分支一起在主体中出现的顺序依次被求

^① 你不能实际将这个定义输入到 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，你需要将其名字改变成其他东西，例如 my-when。

值，直到它们中的一个求值为真。在那一点上，该分支的其余形式将被求值，且分支中最后一个形式的值将被作为整个 `COND` 的返回值。如果该分支在条件之后不含有其他形式，那么该条件的值将被返回。按照约定，那个用来表达 `if/else-if` 链中最后一个 `else` 子句的分支将被写成带有条件 `T`。任何非 `NIL` 的值都可以，但 `T` 可以在阅读代码时作为有用的标记。这样你可以像下面这样使用 `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` 是一个不属于本章的函数，但它跟 `AND` 和 `OR` 紧密相关。它接受单一参数并取反其真值，当参数为 `NIL` 时返回 `T`，否则返回 `NIL`。

而 `AND` 和 `OR` 是宏。它们实现了对任意数量子形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。就是说，它们仅以从左到右的顺序求值用于检测整体真值的必要数量的子形式。这样，只要 `AND` 的一个子表达式求值到 `NIL` 它就立即停止并返回 `NIL`。如果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。另一方面，只要 `OR` 的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：

<code>(not nil)</code>	$\rightarrow T$
<code>(not (= 1 1))</code>	$\rightarrow NIL$
<code>(and (= 1 2) (= 3 3))</code>	$\rightarrow NIL$
<code>(or (= 1 2) (= 3 3))</code>	$\rightarrow T$

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——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直接支持结构化的循环，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 和许多好的抽象一样，无论句法的还是其他什么，Lisp 的循环宏构建以那两个特殊操作符为基础的一组分层抽象之上。

在最底层（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，应用于简单情形时显得过于复杂。因此 Lisp 也提供了另外两个宏，`DOLIST` 和 `DOTIMES`，它们不像 `DO` 那样灵活，但却提供了对于常见的在列表元素上循环和计

^① 如果你想知道的话，这两个特殊操作符是 `TAGBODY` 和 `GO`。现在无需讨论它们，但我将在第 20 章里谈及它们。

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，`DO` 在 Common Lisp 特殊操作符所提供的底层原语之上，提供了一个基本的结构化循环构造，而 `DOLIST` 和 `DOTIMES` 则提供了两个易于使用却不那么通用的构造。并且如同你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，你可以在 `DO` 之上构建你自己的循环构造。

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`；其他的则痛恨它。`LOOP` 爱好者们喜欢它是因为它提供了简洁的方式来表达特定的常用循环构造。它的贬低者们不喜欢它是因为它不太像 Lisp。但无论你倾向于哪一方，`LOOP` 本身都是一个为语言增加新构造的宏的强大威力的令人印象深刻的示例。

7.5 DOLIST 和 DOTIMES

我将从易于使用的 `DOLIST` 和 `DOTIMES` 宏开始。

`DOLIST` 在一个列表的元素上循环操作，使用一个持有列表中所有后继元素的变量来执行循环体。^① 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
```

使用这种方式，`DOLIST` 这种形式本身求值到 `NIL`。

如果你想在列表结束之前中断一个 `DOLIST` 循环，你可以使用一个 `RETURN`。

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

`DOTIMES` 是用于计数循环的高层次循环构造。其基本模板和 `DOLIST` 非常相似。

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

^① `DOLIST` 类似于 Perl 的 `foreach` 或 Python 的 `for`。Java 从版本 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 说这种特定的抽象只得添加到语言之中，然后 Sun 将会发布一个 JSR 并组织一个工业范围内的专家组来推敲其细节。这一过程——按照 Sun 的说法——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，他们可能也仍然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可以为祸 Java 程序员几年时间。

```
body-form*)
```

其中的 *count-form* 必须求值到一个整数上。每次通过循环时 *var* 持有从 0 到比那个数小一个的每一个后继整数。例如：

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

和 DOLIST一样，你可以使用 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，因此你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，你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 是方便和易于使用的，但它们没有灵活到可用于所有循环。例如，如果你想要并行循环多个变量该怎样做？或是使用任意表达式来测试循环的结束呢？如果无论 DOLIST 还是 DOTIMES 都不能满足你的需求，那你还可以使用更通用的 DO 循环。

与 DOLIST 和 DOTIMES 只提供一个循环变量有所不同，DO 允许你绑定任意数量的变量并让你完全控制它们在每次通过循环时将如何改变。你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 表达式整体得到一个返回值。基本的模板看起来像这样：

```
(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)
```

上面的 *init-form* 将在循环的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的迭代开始之前，*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，除非你在循环体中显式为其赋予新值。和 LET 中的变量定义一样，如果 *init-form* 没有给出，那么变量将绑定到 NIL。和 LET 的情形一样，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。

在每次迭代的开始以及所有循环变量都被指定了新值以后，*end-test-form* 会被求值。只要

它求值到 `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` 将被求值，且最后一个形式的值将被作为 `DO` 表达式的值返回。

在循环的每一步里，所有变量的 `step-form` 将在其值赋给任何变量之前被求值。这意味着你可以在步长形式里引用任何其他的循环变量。^① 就是说，在一个循环里像这样：

```
(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 Fibonacci 数的特别有效的方式。）

这个例子还阐述了 `DO` 的另一种特征——由于你可以同时推进多个变量，你经常根本不需要一个循环体。其他时候，你可能省略结果形式，尤其是当你只是使用循环作为控制构造时。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` 本身所唯一需要的。你需要一对括号来围住变量声明，一对用来围住终止测试和结果形式，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。不过一个 `DO` 循环的框架将总是相同的。下面是一些 `DO` 循环的例子，带有黑体表示的框架：

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

注意到结果形式被省略了。不过这并不是一个对 `DO` 特别有意义的使用，因为这个循环可以使用 `DOTIMES` 更简单地写出来。^②

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

另一个例子是一个没有循环体的 Fibonacci 计算循环：

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

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

^① 一个 `DO` 的变种，`DO*`，在求值后续变量的步长形式前为每个变量赋值。关于它的更多细节，参阅你喜爱的 Common Lisp 参考。

^② `DOTIMES` 的另一个被推荐使用的原因是其宏展开将可以包含类型声明从而允许编译器生成更有效的代码。

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

7.7 强大的 LOOP

对于简单的情形，你使用 `DOLIST` 和 `DOTIMES`。但如果它们不符合你的需要，你可以退而使用完整的通用 `DO`。你还能怎样？

看起来大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集聚值：收集、计数、求和、最小值和最大值。如果你需要一个宏来做这些事情中的一件（或同时几件），那么 `LOOP` 宏可以给你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*)
```

主体中的形式在每次通过循环时被求值，整个循环将不停地迭代直到你使用 `RETURN` 来中止它。例如，你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))
```

扩展的 `LOOP` 是一个完全不同的庞然大物。它的区别在于使用特定的循环关键字来实现一种用于表达循环用法的通用性语言。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` 的爱好者们反对说问题在于复杂的循环构造，就算不将它们包装在 `DO` 的晦涩语法里就已经足够难于理解了。他们说最好可以有一种稍微更加详尽的语法来给你关于正在做的事情的某些线索。

例如，下面是一个地道的 `DO` 循环将从 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)
```

一个富有经验的 Lisp 程序员将毫不费力地理解这些代码——只要知道一个 `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。另一方面它的 `LOOP` 版本几乎可以像一个英语句子那样轻松理解。

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

接下来是更多的一些关于 `LOOP` 简单用法的例子。下面这个可以求和前十个平方数：

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

这个用来统计一个字符串中原因字母的个数：

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

下面这个用来计算第 11 个 Fibonacci 数，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

我将把 LOOP 的细节留给第 22 章，但这里值得注意的是，它是另一个宏可被用来扩展基本语言的例子。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 LOOP 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，但它也只是另外一个宏而已。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使用一个提供它的第三方库。

以上就是我们对基本控制构造宏的介绍。现在你可以进一步考察如何定义你自己的宏了。

^① 循环关键字令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符号。如果你想要的话，甚至可以使用真正的关键字——`:for`、`:across`，诸如此类——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——作为函数或变量的名字——也是没有关系的。

第8章 定义你自己的宏

现在是时候开始编写你自己的宏了。前一章里提及的标准宏暗示了你可以用宏做到的某些事情，但这只是开始。相比 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。宏作为语言的一部分可以允许你在核心语言或标准库之上创建抽象来使你更直接地表达你想表达的事物。

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，它们允许你将那些分散注意力的细节抽象掉。而尽管有这些相似性，宏却操作在一个与函数不同的层面上，并创建了完全不同类型的抽象。

一旦你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带来了巨大的好处。但同时它也是经常导致新程序员困惑的主要原因。下面这个故事，尽管从历史或技术意义上并不是真的，但却试图通过给你一种思考宏如何工作的方式来缓解你的困惑。

8.1 Mac 的故事：只是一个故事

很久以前，有一个由 Lisp 程序员所组成的公司。那个时候 Lisp 还没有宏。任何不能用一个函数来定义或是用一个特殊操作符来完成的事情都将不得不每次完全手写，这样带来了很大的不便。不幸的是，这个公司里的程序员们虽然杰出但却非常懒惰。经常在他们的程序中间，当需要编写大量单调乏味的代码时，他们会代替地写下一个注释来描述他们想要在程序的那个位置上编写的代码。更不幸的是，由于他们很懒惰，这些程序员们也很讨厌回过头去实际编写那些由注释所描述的代码。不久，这个公司就有了一大堆无人可以运行的程序，因为它们完全由代表尚需编写的代码的注释所组成。

在走投无路的情况下，老板雇佣了一个初级程序员 Mac。他的工作就是找到这些注释，编写需要的代码，然后再将其插入到程序中注释的位置上。Mac 从未运行过这些程序——因为程序尚未完成，所以他当然运行不了。但就算这些程序被完成了，Mac 也不知道将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，并将其发还给最初的程序员。

在 Mac 的帮助下，所有的程序不久都被完成了，并且公司通过销售它们赚了很多钱——以至于用这些钱公司可以将其程序员团队扩大一倍。但出于某种原因，没有人想到雇佣任何人来帮助 Mac；很快他就在单枪匹马地同时协助几十个程序员。为了避免将他的所有时间花在搜索源代码

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 email 给他并等待他将替换的代码 email 回来。不幸的是，就算有了这个变化，Mac 也很难跟上程序员的脚步。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未遂的程序员，在他的一个注释里包含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。

下一个创新的到来是有一个程序员在他程序的开始处放了一个注释，其中含有一个函数定义以及一个注释说，“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。”同一个程序里的其他注释描述了类似这样的东西：“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快速流行起来，多数程序都含有数十个注释用来定义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)”这个写法——由于程序员们仍然很懒惰——很快被简化成“DEF. MAC. R/O”，然后是“DEFMACRO”。

不久以后这些给 Mac 的注释中再没有实际的英语了。Mac 每天做的事情就是阅读并响应那些来自编译器的含有 DEFMACRO 注释的 email 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 email 完全没有问题。Mac 突然手头上有了大量时间并且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认识到已经很长时间没人见过 Mac 了。当它们去他的办公室时，他们发现所有东西上都积了薄薄的一层灰，一个桌子上放着几个热带地区的旅行册，而电脑是关着的。但是编译器仍在正常工作——这怎么可能？看起来 Mac 对编译器做了一个最后的修改：与其将注释 email 发给 Mac，现在编译器将那些 DEFMACRO 中所定义的函数保存下来并再其被其他注释调用时运行它们。程序员们决定没有理由告诉大老板 Mac 不再来办公室了。因此直到今天 Mac 还领着薪水并且不时地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(所有其他东西)之间的区别。当你编写宏时，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以后程序才可以实际被运行。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所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，和那些运行在运行期的代码，是运行在完全不同的环境下的。也就是说，在宏展开期没有办法访问那些仅存在于运行期的数据。正如 Mac 无法运行他正在写的程序是因为他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，假设下面的源代码出现在一个程序的某个地方：

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

正常情况下，你将 `x` 设为一个变量，它将保存传递给一个对 `foo` 调用的参数。但在宏展开期，比如说当编译器正在运行 `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` 的值是代表源代码的 Lisp 列表，即 `(> x 10)` 以及 `(print 'big)`。假设 `WHEN` 是像前一章里你所看到那样用类似下面的宏所定义而成的：

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

当 `foo` 中的代码被编译时，`WHEN` 宏将以那两个形式作为参数来运行。形参 `condition` 将被绑定到形式 `(> x 10)` 上，而形式 `(print 'big)` 将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无甚明显，因为它们临时纠缠在了一起。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，宏总是被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产生做某些事情的代码而不是直接做任何事情。

8.3 DEFMACRO

如同你在第 3 章里所看到的，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)”。一个 `DEFMACRO` 的基本框架和一个 `DEFUN` 框架很相似：

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

和函数一样，一个宏由一个名字、一个形参列表、一个可选文档字符串，以及一个 Lisp 表达式体所构成的。^① 尽管如此，正如我刚刚所讨论的，一个宏的任务并不是直接做任何事——它的工作是生成以后可以做你想要的事的代码。

宏可以使用 Lisp 的所有能力来生成其展开式，这意味着在本章里我只能初步说明你可以用宏做到的事情。不过我却可以描述一个通用的编写宏的过程，它可以工作在从最简单到最复杂的所有宏上。

一个宏的工作是将一个宏形式——也就是一个首元素为宏的名字的 Lisp 形式——转化成做

^① 和函数一样，宏也可以含有声明，但你现在不需要担心它们。

特定事情的代码。有时你是从想要编写的代码开始来编写宏的，就是说是从一个示例的宏形式开始的。其他时候你是在连续几次书写了相同的代码模式并认识到通过抽象该模式可以使你的代码更清晰以后才开始决定编写一个宏的。

无论你从哪一端开始，你都需要在开始编写一个宏之前搞清楚另一端：你需要同时知道你从哪里开始并且正在向何处去然后才能希望编写代码来自动地做到这点。因此编写一个宏的第一步是去书写至少一个宏调用的示例以及该调用应当展开成的代码。

一旦你有了一个示例调用和预想的展开式，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，这将是一件极其简单的事——编写一个反引用模板并将宏参数插入到正确的位置上。复杂的宏将是一个独立的程序，带有配套的助手函数和专门的数据结构。

在你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，你需要确保宏所提供的抽象没有“泄漏”其实现的细节。有漏洞的宏抽象将只在特定参数上可以正常工作，或是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，而所有这些都可以轻易避免，只要你知道如何检查它们就好了。我将在“堵住漏洞”那一章里讨论具体的方法。

总结起来，编写一个宏的步骤如下所示：

- 1、编写一个示例的宏调用以及它应当展开成的代码，或者以相反的顺序。
- 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 一个示例宏：do-primes

为了看到这三步过程是怎样工作的，你将编写一个宏 `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。这并非意味着它是一个特别有用的宏——只是一种演示该过程的手段。

首先你将需要两个工具函数，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参数的下一个素数。对于这两种情况你都可以使用两个简单而低效的暴力手法。

```
(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))
```

现在你可以写这个宏了。按照前面所概括的过程，你需要至少一个宏调用的示例以及它应当展开成的代码。假设你从你想要写出下面代码的思路开始：

```
(do-primes (p 0 19)
  (format t "~d " p))
```

来表达一个循环，在每个大于等于 0 并小于等于 19 的素数上分别执行依次循环体，并以变

量 p 保存当前素数。仿照标准 DOLIST 和 DOTIMES 宏来定义该宏是合理的。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 ~ p)))
```

现在你可以开始编写将前者转化成后者的代码了。

8.5 宏参数

由于传递给一个宏的参数是代表宏调用源代码的 Lisp 对象，因此任何宏的第一步就是去解出那些对象中用于计算展开式的部分。对于那些简单的将其参数直接插入到一个模板中的宏，这已步骤是相当简单的：只需定义正确的形参来保存不同的参数就可以了。

但是这一方法并不适用于 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，下界 0 和上界 19。但如果你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；三个元素被拆分开放在不同的位置上。

你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 形参来保存形式体，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)))
```

很快我将解释上述宏形式体怎样生成正确的展开式；目前你只需注意到变量 var、start 和 end 每个都持有一个从 var-and-range 中解出的值，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，你不需要“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。“解构”正如其名字所显示的那样，涉及到分拆一个结构体——在本例中是传递给一个宏的列表结构形式。

在一个解构形参列表中，一个简单的形参名将被替换成一个嵌套的形参列表。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。例如你可以将 var-and-range 替换为一个列表 (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殊特性是你可以使用 &body 作为 &rest 的同义词。&body 和 &rest 在语义上是等价的，但许多开发环境将根据一个 &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人类读者和你的开发

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给你自动错误检查——通过以这种方式定义 `do-primes`，Lisp 将可以检测到那些首参数不是三元素列表的调用并给你一个有意义的错误信息，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，在你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，特别的反引用语法刚好合适。回顾一下，一个反引用表达式与一个引用表达式相似，除了你可以“解引用”（`unquote`）特定的值表达式——在其前面加上逗号，可能后接一个“`@`”符号。没有这个“`@`”符号，逗号会导致子表达式的值被原样包含。有了这个“`@`”符号，其值——必须是一个列表——将被“拼接”到其所在的列表中。

另一个理解反引用语法的有用方式是将其视为编写生成列表的代码的一种特别简洁的方式。这种理解方法的优点是可以相当明确地看到其表面之下实际发生的事——当读取器读到一个反引用表示式时，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例子，同时带有等价的列表构造代码以及如果你求值无论反引用表达式或者其等价代码将得到的结果。^①

Table 8-1. Backquote Examples

Backquote Syntax	Equivalent List-Building Code	Result
------------------	-------------------------------	--------

^① 我还没有讨论到的 APPEND 是一个函数，其接受任意数量的列表参数并返回将它们拼接在一起成为一个单独的列表。

<code>`(a (+ 1 2) c)</code>	<code>(list 'a '(+ 1 2) 'c)</code>	<code>(a (+ 1 2) c)</code>
<code>`(a ,(+ 1 2) c)</code>	<code>(list 'a (+ 1 2) 'c)</code>	<code>(a 3 c)</code>
<code>`(a (list 1 2) c)</code>	<code>(list 'a '(list 1 2) 'c)</code>	<code>(a (list 1 2) c)</code>
<code>`(a ,(list 1 2) c)</code>	<code>(list 'a (list 1 2) 'c)</code>	<code>(a (1 2) c)</code>
<code>`(a ,@(list 1 2) c)</code>	<code>(append (list 'a) (list 1 2) (list 'c))</code>	<code>(a 1 2 c)</code>

重要的是注意到反引用只是一种便利设施。只不过它带来了极大的便利而已。为了说明究竟有多大，我们可以将 `do-primes` 的反引用版本和下面的版本相比较，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))
```

如同你即将看到的，`do-primes` 的当前实现不能正确地处理特定的边界情况。但首先你应当确认它可以至少工作在最初的例子上。你可以用两种方式来测试它。你可以通过简单的使用它来间接测试它——也就是说如果结果的行为是正确的那么展开式就是正确的。例如你可以将 `do-primes` 最初的用例输入到 REPL 中并看到它确实打印了正确的素数序列。

```
CL-USER> (do-primes (p 0 19) (format t "~d " p))
2 3 5 7 11 13 17 19
NIL
```

或者你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 表达式作为一个参数并返回做一层宏展开的结果。^① 由于 `MACROEXPAND-1` 是一个函数，为里传给它一个字面的宏形式，你必须引用它。你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 中你可以更方便地检查一个宏的展开式：将光标放置在你源代码中一个宏形式的开放括号上并输入 C-c RET 来调用 Emacs 函数 `slime-macroexpand-1`，后者将传递宏调用到 `MACROEXPAND-1` 上并“美化输出”结果到一个临时缓冲区上。

无论你怎样得到展开式，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` 可以工作。

^① 另一个函数，`MACROEXPAND`，将持续展开结果，只要返回的展开式的第一个元素是宏的名字就会不断进行下去。尽管如此，这个函数经常会显示出关于代码行为的比你想知道的更加底层的视角，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你的宏最终可以展开成怎样的代码具有一定教育意义，但这对你自己的宏正在做什么并不是一个有用的视图。

^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。如果是这样的话，求值 (`(setf *print-pretty* t)`) 将使展开式更易于阅读。

8.7 堵上漏洞

Joel 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) 来描述一种抽象，其“泄露”了其本该抽象掉的细节。由于编写宏是一种创造抽象的方式，你需要确保你的宏不产生不必要的泄露。^①

如同即将看到的，一个宏可以以三种方式泄露其内部工作细节。幸运的是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。

当前的定义存在三种可能宏泄露中的一种：确切的说，它会多次求值 `end` 子形式。假设你没有使用诸如 `19` 这样的字面数字而是像 `(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循环在从 `0` 到由 `(random 100)` 所返回的无论什么随机数字上。尽管如此，`MACROEXPAND-1` 的结果显示这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
```

当展开式代码被运行时，`RANDOM` 将在每次进行循环的终止测试时被求值一次。这样，循环将不会终止在 `p` 大于一个初始给定地随机数上，而是循环将迭代到刚好产生出一个小于或等于当前 `p` 的值的随机数上才会终止。由于迭代的整体次数将仍然是随机的，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小心 `end` 形式被求值超过一次的情况。一种堵上漏洞的方式将是简单的将其定义成 `do-primes` 的行为。但这并不是非常令人满意的——你在实现宏时应当试图遵守最少惊动原则 (Principle of Least Astonishment)。并且程序员们通常希望它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，后两者都不会导致其循环体之外的形式被多次求值，多数程序员将期待 `do-primes` 具有相似的行为。

你可以相当容易地修复多重求值问题；你只需生成代码来求值 `end` 一次并将其值保存在一个稍后用到的变量里。回想在一个 `DO` 循环中，用一个初始形式并且没有步长形式来定义的变量不会在迭代过程中改变其值。因此你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)
```

^① 出自 Joel Spolsky 的《Joel on Software》，也可在 <http://www.joelonsoftware.com/articles/LeakAbstractions.html> 获取到。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你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。

```
(,var (next-prime ,start) (next-prime (1+ ,var))))
((> ,var ending-value))
,@body))
```

不幸的是，这一修复给宏抽象引入了两个新的漏洞。

一个新的漏洞类似于你刚修复的多重求值漏洞。因为在一个 `DO` 循环中变量的初始形式是以变量被定义来求值，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。这一泄露在 `start` 和 `end` 都是像 0 和 19 这样的字面值时不会带来任何问题。但当它们是可以产生副作用的形式时，以不同的顺序求值他们将再一次违反最少惊动原则。

这一漏洞可以通过交换两个变量定义的顺序来轻易堵上。

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (ending-value ,end))
       ((> ,var ending-value))
       ,@body))
```

最后一个你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——可以跟传递给宏的代码或是宏被调用的上下文产生交互。下面这个看起来无辜的 `do-primes` 调用由于这个漏洞将无法正常工作：

```
(do-primes (ending-value 0 10)
           (print ending-value))
```

这个也不可以：

```
(let ((ending-value 0))
  (do-primes (p 0 10)
             (incf ending-value p))
  ending-value)
```

再一次，`MACROEXPAND-1` 可以向你展示问题所在。第一个调用展开成这样：

```
(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` 将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-value)`

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，你所需要的是一个永远不会在宏展开代码之外被用到的符号。你可以尝试使用一个真正罕用的名字，但即便如此也做不到万无一失。你也可以通过使用第 21 章里将要介绍的包（package）在某种意义上起到保护作用。但有更好的解决方案。

函数 `GENSYM` 在其每次被调用时返回一个唯一的符号。这是一个没有被 Lisp 读取器所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。这样代替使用一个像 `ending-value` 这样的字面名称，你可以在每次 `DOTIMES` 被展开时生成一个新的符号。

```
(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)))
```

注意到调用 `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，其值是另一个变量的名字。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在读取时所创建的，而 `ending-value-name` 的值则是在宏代码运行时程序化生成的。

使用这个定义，前面两个有问题的形式现在可以展开成按你预想方式工作的代码了。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，但它并不重要；重要的在于这个符号的对象标识。生成符号被打印成未进入符号的正常语法，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)
```

如果你将 `do-primes` 形式替换成其展开式的话将看起来像这样：

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p))))
```

^① 该循环在给定任意素数下的无限性并非显而易见的。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 。

```
(#: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所引入的变量所掩盖，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里更多的经验以后，你将可以鉴别一个用在某个位置上的给定名字是否会导致宏抽象中的漏洞。但出于安全起见，使用一个符号生成的名字并没有什么坏处。

有了这些修复，现在你已堵上 `do-primes` 实现中的所有漏洞了。一旦你学得了一点宏编写方面的经验以后，你将学会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，只要你遵循下面所概括的这些规则就好了：

- 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，没有理由认为你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，你已经见过了一个这样的模式——许多宏，例如最后版本的 `do-primes`，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的模式，为什么不用一个宏来将其抽象掉呢？

在本节中你将编写一个宏，`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，代码生成另外的代码。尽管复杂的编写宏的宏在你习惯于在头脑中牢记不同层次的代码之前可能会有一点困惑，但 `with-gensyms` 是相当直接的并且可以作为一个有用但又不会过于浪费脑筋的练习来对待。

你想要写出类似下面这种用法的宏：

```
(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`，其绑定每一个命名的变量——在本例中是 `ending-value-name`——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))
```

注意到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替换为一个符号的列表，从而在 REPL 中测试 LOOP表达式所生成的代码。

```
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，被你封装在一个 with-gensyms中的代码将可以引用到任何传递给 with-gensyms的变量列表中所命名的变量。

如果你在新的 do-primes定义中对 with-gensyms形式进行宏展开，你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。尽管这个宏是相对平凡的，但重要的是要清楚地看到不同的宏分别在何时被展开的：当你编译关于 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。当你编译一个使用了 do-primes的函数时，由 with-gensyms所生成的代码将被运行用来生成 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，因为它在 do-primes被编译时早已经被展开了。

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，你可以几乎跟最初的有漏洞版本一样简单地写出 do-primes来，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，once-only的实现如果详加解释的话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。如果你真想进一步提高你的宏技术的话，你可以试图分析它的工作方式。它看起来像这样：

```
(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。所有你目前为止见到的宏都是相当简单的例子，它们帮助你节省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，它们允许你一种假如没有宏就完全做不到的方式来表达事物。你将从下一章开始，在那里你将构建一个简单而高效的单元测试框架。

第9章 实践：建立一个单元测试框架

在本章里，你将回来继续写代码，为 Lisp 开发一个简单的单元测试框架。这将给你一个机会在真实代码中使用你从第 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有能力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并且更精确时，你没有必要将时间花在埋头检查测试所输出的答案上。因此，每一个测试用例必须是一个产生布尔值的表达式——真或假，通过或失败。举个例子，如果你正在为内置的“+”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)
```

带有副作用的函数将以稍微不同的方式进行测试——你将需要调用该函数然后检查所期待的副作用的证据。^②但最终，每一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

如果你正在做自主测试，那么你可以在 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在你想要的时候可以轻松地组织和运行这些测试用例。如果你想要以可能可以工作的最简单情况开始，那你可以只写一个函数，它求值所有测试用例并用 `AND` 将结果连在一起：

```
(defun test-+ ()
  (and
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

无论何时当你想要运行这组测试用例，你可以调用 `test-+`。

^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，因为要是这么基本的东西都无法工作的话，那么测试过程可以按照你期待方式运行的机会也微乎其微。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 来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东西；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。

```
CL-USER> (test-+)
T
```

一旦它返回 `T`，你就知道测试用例通过了。这种组织测试的方式也是优美简洁的——你不需要编写大量的重复测试代码。尽管如此，一旦你发现一个测试用例失败了，它的运行报告就会留下一些想要的东西。当 `test-+` 返回 `NIL` 时，你将知道某些测试失败了，但你将不会知道究竟是哪一个测试用例。

因此，让我们来尝试另一个简单的——甚至有些蠢的——方法。为了找出每一个测试用例所发生的情况，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)))
```

现在每一个测试用例将单独报告结果。`FORMAT` 指令中的 `~:[FAIL~;pass~]` 部分导致 `FORMAT` 在其第一个格式参数为假时打印出“FAIL”，而在其他情况下为“pass”。^① 然后你将测试表达式本身标记到结果上。现在运行 `test-+` 可以明确显示发生了什么事。

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
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。对 `FORMAT` 的重复调用以及测试表达式乏味的重复急切需要被重构。测试表达式的重复尤其讨厌，因为如果你错误输入了它，测试结果将被错误地标记。

另一个问题在于你无法得到单一的关于是否所有测试都通过的指示。对于只有三个测试用例来说，很容易通过扫描输出并查找“FAIL”来看到这点；不过当你有几百个测试用例时，这将会非常困难。

9.2 重构

你真正想要的是一种可以写出像第一个 `test-+` 那样返回单一的 `T` 或 `NIL` 值的流线型函数的方式，但同时还可以像第二个版本那样报告单独测试用例的结果。由于第二个版本更接近于从功能性角度上你所想要的，你最好的办法就是看是否可以将某些恼人的重复消除掉。

消除重复的相似 `FORMAT` 调用的最简单方法是创建一个新函数。

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form))
```

现在你可以使用 `report-result` 代替 `FORMAT` 来写 `test-+` 了。这不是一个大的改进，但至少现在如果你打算改变报告结果的方式，只有一处是你需要修改的。

```
(defun test-+ ()
```

^① 我将在第 18 章里讨论包括这个在内的 `FORMAT` 指令的更多细节。

```
(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)))
```

现在你需要摆脱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。你真正想要的是可以将表达式同时作为代码（为了获得结果）和数据（用来作为标签）来对待。无论何时当你想要将代码作为数据来对待，这就意味着你肯定需要一个宏。或者从另外一个角度来看，你所需要的是一种自动编写可能出错的 `report-result` 调用的方式。你可能会想要写出类似这样的东西：

```
(check (= (+ 1 2) 3))
```

并让其具有下面的含义：

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
```

很容易写出一个宏来做这种转换。

```
(defmacro check (form)
  `(report-result ,form ',form))
```

现在你可以改变 `test-+` 来使用 `check`。

```
(defun test-+ ()
  (check (= (+ 1 2) 3))
  (check (= (+ 1 2 3) 6))
  (check (= (+ -1 -3) -4)))
```

由于你不喜欢重复的代码，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每个包装在一个对 `report-result` 的调用里。

```
(defmacro check (&body forms)
  `(#(loop for f in forms collect `(report-result ,f ',f))))
```

这个定义使用了一个常见的宏习惯用法，将一系列打算转化成单一形式的形式分装在一个 `PROGN` 之中。注意到你是怎样使用 `,@` 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。

通过使用 `check` 的新版本，你可以写出一个像下面这样的 `test-+` 的新版本：

```
(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))))
```

感谢 `check`，这个版本和 `test-+` 的第一个版本一样简洁，但却可以展开成和第二个版本做相同事情的代码。并且现在任何你对 `test-+` 的行为想要做出的改变，都可以通过改变 `check` 来

做到。

9.3 修复返回值

接下来你开始修复 `test-+`使其返回值可以指示是否所有测试用例都通过了。由于 `check`负责生成最终用来运行测试用例的代码，你只需改变它来生成可以同时跟踪结果的代码就可以了。

作为第一步，你可以对 `report-result`做一个小的改变使其在报告时顺便返回测试用例的结果。

```
(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的结果，看起来你只需将 `PROGN`改变成一个 `AND`就可以组合结果了。不幸的是，由于其短路行为的存在，`AND`在本例中并不能完成你想要的事：一旦一个测试用例失败了，`AND`将跳过其余的测试。另一方面，如果你有一个像 `AND`那样工作的操作符，同时却没有短路行为，那么你就可以将它用在 `PROGN`的位置上并且事情也就做完了。Common Lisp 并不提供这样的一个构造，但没有理由你不能使用它：你自己来写一个宏提供这一功能是件极其简单的事。

暂时把测试用例放在一遍，你所需要的是一个宏——让我们称其为 `combine-results`——它可以让你写成这样：

```
(combine-results
  (foo)
  (bar)
  (baz))
```

并且意味着下面这些东西：

```
(let ((result t))
  (unless (foo) (setf result nil))
  (unless (bar) (setf result nil))
  (unless (baz) (setf result nil)))
  result)
```

编写这个宏，唯一麻烦的一点是，你需要在展开式中引入一个变量——前面代码中的 `result`。如同你从前面章节所看到的，在宏展开式中使用一个变量的字面名称可以导致你宏抽象中的一个漏洞，因此你将需要创建一个唯一的名字。这就是 `with-gensyms`的工作。你可以像这样来定义 `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)))
```

现在你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-+`可以输出它的三个测试表达式的结果并返回 `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
```

并且如果你改变里一个测试用例使其失败^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

由于你只有一个测试函数, 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, 所有你需要做的就是在 `check` 形式中找到那个测试用例然后找出其失败的原因。但如果你编写里大量测试, 你将可能想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, 加入你想要对 “`*`” 函数添加一些测试用例。你可以写一个新的测试函数。

```
(defun test-* ()
  (check
    (= (* 2 2) 4)
    (= (* 3 5) 15)))
```

现在你有两个测试函数, 你可能想要另一个函数来运行所有的测试, 这是相当简单的。

```
(defun test-arithmetic ()
  (combine-results
    (test-+)
    (test-*)))
```

在这个函数中, 你使用 `combine-results` 来代替 `check`, 因为 `test-+` 和 `test-*` 都将分别汇报它们自己的结果。当你运行 `test-arithmetic` 时, 你将得到下面的结果:

```
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你需要跟踪该问题。在只有五个测试用例和两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。但假如你有 500 个测试用例分散在 20 个函数里, 如果测试结果可以告诉你每一个测试用例来自什么函数就非常好了。

^① 如果 `test-+` 已经被编译了——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, 从而允许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。

由于打印结果的代码集中在 `report-result` 函数里，你需要一种方式来传递你当前所在测试函数的信息给 `report-result`。你可以为 `report-result` 增加一个形参来传递这一信息，但生成 `report-result` 调用的 `check` 并不知道它是从什么函数被调用的，这就以为着你也需要改变你调用 `check` 的方式，向其传递一个参数使其随后传给 `report-result`。

这正是动态变量被设计用于解决的那类问题。如果你创建里一个动态变量使得每一个测试函数在调用 `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)
```

现在你需要对 `report-result` 做出另一个细微的改变使其在 `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，你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标签了。

```
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。但如果你更进一步查看导致代码重复的根源，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来表达“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一个劣质的工具，因为一个半成品的抽象在代码中就是通过模式来表现的，因此你将必然得到大量的重复代码，其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，没有办法可以保证不同的程序员（或者甚至同一个程序员工作在不同时期）实际上以同样的方式来理解这种抽象。为了得到一个完整的抽象，你需要一种方式来表达“这是一个测试函数”并且让所有模式所需要的代码为你生成出来。换句话说，你需要一个宏。

由于你试图捕捉的模式是一个 `DEFUN` 加上一些样板代码，你需要写一个宏使其展开成一个 `DEFUN`。然后你使用该宏来代替一个简单的 `DEFUN` 去定义测试函数，因此将其称为 `deftest` 将是合理的。

```
(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
    (let ((*test-name* ',name)) ,@body)))
```

使用该宏你可以向下面这样重写 `test-+`:

```
(deftest test-+ ()
  (check
   (= (+ 1 2) 3) (= (+ 1 2 3) 6) (= (+ -1 -3) -4)))
```

9.6 测试层次体系

现在你建立了作为一等公民的测试函数，问题可能产生了，`test-arithmetic` 应该是一个测试函数吗？事实证明，这件事无关紧要——如果你确实用 `deftest` 来定义它，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。

但是现在想像你有上千个测试用例需要被组织在一起。组织的第一层由诸如 `test-+` 和 `test-*` 这些直接调用 `check` 的测试函数所提供，但在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。很有可能一个测试用例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，你很可能不仅仅想要知道哪一个底层测试函数含有这个测试用例。

如果你用 `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*` 做一个小的改变，你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)
```

因为你已经抽象了定义测试函数的过程，你可以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))
```

改变成下面的：

```
(let ((*test-name* (append *test-name* (list ',name))))
```

由于 `APPEND` 返回一个由其参数作为元素所构成的新列表，这个版本将绑定 `*test-name*` 到一个含有 `*test-name*` 的旧内容以及新的名字追加到结尾处的列表。^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
```

随着你测试套件的增长，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。例如下面的定义：

```
(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
```

^① 再一次，如果测试函数已经被编译了，在改变了宏以后你将需要重新编译它们。

^② 你在第 12 章里将会看到，用 `APPEND` 在列表结尾处追加元素并不是构造一个列表的最高效的方式。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。并且如果这成为一个问题，所有你需要做的就是改变 `deftest` 的定义。

9.7 总结

你可以继续为这个测试框架增加更多特性。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里。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)
```

值得回顾的是，你能走到这一步是因为它显示了 Lisp 编程的通常方式。

你从定义一个解决你问题的简单版本开始——怎样求值一些布尔表达式并找出是否它们全部返回真。将它们用 AND 连在一起可以工作并且在句法上很干净但却无法满足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干净。你从一个标准的重构技术开始，将某些代码放进一个函数 `report-result`。不幸的是你发现使用 `report-result` 会导致冗长和易错的代码，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。因此你写里 `check` 宏来自动化正确调用 `report-result` 的细节。

在编写 `check` 的时候，你认识到你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。

在那一点处你调整了 `check` 的 API，从而允许你开始看到 `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布尔值用来指示是否所有测试用例均已通过。接下来你没有立即继续玩弄 `check`，而是停下来沉溺于一个设计精美的微型语言。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

实很简单了。

在那一点处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。一旦你开始对测试函数做出修改，你就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结果汇报的方式。

现在学会了函数、变量和宏的基础知识，以及一点使用它们的实践经验，你可以开始探索 Common Lisp 由函数和数据类型所组成的丰富的标准库了。

第10章 数字、字符和字符串

尽管函数、变量、宏和 25 个特殊操作符提供了语言本身的基本构造单元，但程序的构造单元将是你所使用的数据结构。正如 Fred Brooks 在《The Mythical Man-Month》里所提到的，“编程的本质在于表示。”^①

Common Lisp 提供了在现在语言中通常可以见到的大多数数据类型的内置支持：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达对文件名的抽象。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，作为参数传递，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，并且因为可以跟语言的其余部分紧密集成，它们可以更容易地高效实现。但正如你在后续章节里将要看到的，Common Lisp 也为你提供了几种方式来定义新的数据类型及其之上的操作，以及将它们与内置数据类型相集成。

尽管如此，目前你将从内置数据类型开始。因为 Lisp 是一种高阶语言，关于不同的数据类型具体实现的细节在很大程序上是隐藏的。从你作为一个语言用户的角度来看，内置数据类型是由操作在它们之上的函数所定义的。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 打印器可使用的特殊语法。这就是为什么你可以将字符串写成 "foo"，将数字写成 123, 1/23 和 1.23，以及列表写成 (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。

在本章中，我将谈及内置的“标量”数据类型：数字、字符和字符串。技术上来讲字符串并不是真正的标量——一个字符串是一个字符的序列，并且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是因为多数字符串相关的函数将它们用为单一值来处理，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的紧密关系。

10.1 数字

正如 Barbie 所说，数学是困难的。^②Common Lisp 不能使其数学部分变得简单一些，但它确实可以比其它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 最初是作为

^① Fred Brooks, *The Mythical Man-Month*, 20th Anniversary Edition (Boston: Addison-Wesley, 1995), p. 103. Emphasis in original.

^② Mattel's Teen Talk Barbie

用来研究数学函数的工具——由一个数学家所设计而成的。并且 MIT 的 MAC 项目的主要项目之一，Macsyma 符号代数系统，是由 Maclisp，一种 Common Lisp 的前身所写成的。此外，Lisp 还曾被用作 MIT 这类院校的教学语言，在那里即便计算机科学教授们也不愿意告诉他们的学生 $10/4=2$ ，这导致了 Lisp 对精确比值的支持。Lisp 还曾经多次在高性能数值计算领域与 FORTRAN 竞争。

Lisp 作为一门用于数学的良好语言的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 中的整数可以是几乎任意大而不是被限制在一个机器字的尺寸上。^①而两个整数相除将得到一个确切的比值而非一个截断的值。并且由于比值是由成对的两个任意尺寸的整数所表示的，因此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类型，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

你可以用多种方式来书写字面数值；在第 4 章里你已经看到了一些例子。尽管如此，你需要牢记 Lisp 读取器 Lisp 求值之间的分工——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给定数字来说，它可以有多种不同的字面表示方式，所有这些都将被 Lisp 读取成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——比如说在 REPL 中——它们将以一种可能与输入该数字时所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
CL-USER> #xa
10
```

整数的语法是一个可选的符号 (+ 或 -) 后接一个或多个数位。比值被写成一个可选符号、

^① 很明显，一个具有有限内存的计算机可以表达的数字的尺寸在事实上仍然是有限的；更进一步，特定 Common Lisp 实现中所使用的大数的实际表示在其所能表达的数字尺寸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 实现可以轻易处理大至或超出 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 ($+ a b$) 这样简单的东西也将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将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。从而允许其生成与 C 或 FORTRAN 编译器做相同工作的代码。为这类优化而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，许多 Common Lisp 实现 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)。

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”——这就是为什么 10 和 20/2 都被读成同一个数字，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 10 以外的进制来书写有理数也是可能的。如果前缀#B 或#b，一个字面有理数将被作为二进制读取，其中 0 和 1 是唯一的合法数位。一个 #o 或 #o 代表一个八进制数(合法数位 0-7)，而 #x 或 #x 代表十六进制数(合法数位 0-F 或 o-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(总是以十进制书写)。超过 9 的附加“数位”从字母 A-Z 或 a-z 中获取。注意到这些进制指示符应用到整个有理数上——不可能以一种进制来书写一个比值的分值，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一个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
#36rABCDEFGHIJKLMNPQRSTUVWXYZ	→ 8337503854730415241050377135811259267835

你也可以用多种方式来书写浮点数。和有理数不同，用来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达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的符号后跟一个非空的十进制数位序列，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。指数标记由一个单一字母后跟一个可选符号和一个数位序列所组成，其代表 10 的指数用来跟指数标记的数字相乘。该字母由两重作用：它标记了指数的开始并且指示了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 代表缺省表示方式(默认为单浮点数)应当被使用。

^① 通过改变全局变量 *READ-BASE* 也有可能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相区别的表示方法，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”被加上引号是因为，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 这样的东西代表 10000.0，而在真正的科学计数法中将被写成 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，因此像 $1.0 \times e^4$ 这样的东西尽管表面上看类似于 1.0e4，但却是一个完全不同的值，约等于 54.6。

没有指数标记的数字以缺省表示来读取并且必须含有一个小数点后接至少一个数位以区别于整数。一个浮点数中的数位总是以十进制数位来对待——语法 #B、#X、#O 和 #R 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	~ARR 1.0
1e0	~ARR 1.0
1d0	~ARR 1.0d0
123.0	~ARR 123.0
123e0	~ARR 123.0
0.123	~ARR 0.123
.123	~ARR 0.123
123e-3	~ARR 0.123
123E-3	~ARR 0.123
0.123e20	~ARR 1.23e+19
123d23	~ARR 1.23d+25

最后，浮点数以它们自己的语法写成，也就是 #C 或 #c 跟上一个由两个实数所组成的列表，分别代表实数的实部和虚部。事实上有五种类型的复数因为实部和虚部必须同为有理数或同是相同类型的浮点数。

不过你可以随意书写它们——如果一个复数被写成由一个有理数和一个浮点数所组成，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方法的浮点数，使用较小的表示方法的那个将被升级。

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——由于这样的值从数学上讲是有理的，它们将用对应的有理数值来表示。同样的数学理由对于由浮点数所组成的复数也成立，但其中那些带有零虚部的复数将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所写成的数字的例子：

#c(2 1)	~ARR #c(2 1)
#c(2/3 3/4)	~ARR #c(2/3 3/4)
#c(2 1.0)	~ARR #c(2.0 1.0)
#c(2.0 1.0d0)	~ARR #c(2.0d0 1.0d0)
#c(1/2 1.0)	~ARR #c(0.5 1.0)
#c(3 0)	~ARR 3
#c(3.0 0.0)	~ARR #c(3.0 0.0)
#c(1/2 0)	~ARR 1/2
#c(-6/3 0)	~ARR -2

10.3 初等数学

基本的算术操作——加法、减法、乘法和除法——通过函数 “+”、“-”、“*”、“/” 支持在所有不同类型的 Lisp 数字上。使用超过两个参数来调用这些函数中的任何一个等价于在前两个参数上调用相同的函数然后在得到的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)	~ARR 3
---------	--------

^① 出于数学一致性考虑，“+” 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。

```
(+ 1 2 3)           ~ARR 6
(+ 10.0 3.0)        ~ARR 13.0
(+ #c(1 2) #c(3 4)) ~ARR #c(4 6)
(- 5 4)             ~ARR 1
(- 2)               ~ARR -2
(- 10 3 5)          ~ARR 2
(* 2 3)             ~ARR 6
(* 2 3 4)           ~ARR 24
(/ 10 5)            ~ARR 2
(/ 10 5 2)          ~ARR 1
(/ 2 3)             ~ARR 2/3
(/ 4)               ~ARR 1/4
```

如果所有参数都是相同类型的数（有理数、浮点数或复数），结果将是相同类型的，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——如果所有参数都是实数但其中有一个或更多是浮点数，那么其他参数将被转化成以实际浮点参数的“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。类似地，如果参数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)           ~ARR 3.0
(/ 2 3.0)           ~ARR 0.6666667
(+ #c(1 2) 3)       ~ARR #c(4 2)
(+ #c(1 2) 3/2)    ~ARR #c(5/2 2)
(+ #c(1 1) #c(2 -1)) ~ARR 3
```

因为“/”不做截断处理，所以Common Lisp提供了4种类型的截断和舍入用于将一个实数（有理或浮点）转化成整数：`FLOOR`向负无穷方向截断，返回小于或等于参数的最大整数。`CEILING`向正无穷方式截断，返回大于或等于参数的最小整数。`TRUNCATE`向零截断，使其对于正的参数等价于`FLOOR`而对于负的参数等价于`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`，它们返两个实数截断相除得到的模和余数。这两个函数与`FLOOR`与`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。^①

函数`1+`和`1-`提供了表达从一个数字增加或减少一个的简化方式。注意到它们和宏`INCF`和`DECREF`有所不同。`1+`和`1-`只是返回一个新值的函数，而`INCF`和`DECREF`会修改一个位置。下面的恒等式显示了`INCF/DECREF`、`1+/1-`，和`+/-`之间的关系：

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

^①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%。（技术上来讲，%在C中的行为直到C99标准时才明确指定）

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，“=”将把不同类型的数学意义上等价的值视为等价的，而通用等价谓词 EQUAL 将由于其类型差异而视其不等价。(尽管如此，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。这样：

```
(= 1 1)           ~ARR T
(= 10 20/2)       ~ARR T
(= 1 1.0 #c(1.0 0.0) #c(1 0)) ~ARR T
```

相反，函数 /= 只有当它的全部参数都是不同值才返回真。

```
(/= 1 1)      ~ARR NIL
(/= 1 2)      ~ARR T
(/= 1 2 3)    ~ARR T
(/= 1 2 3 1)  ~ARR NIL
(/= 1 2 3 1.0) ~ARR 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)      ~ARR T
(> 2 3)      ~ARR NIL
(> 3 2)      ~ARR T
(< 2 3 4)    ~ARR T
(< 2 3 3)    ~ARR NIL
(<= 2 3 3)   ~ARR T
(<= 2 3 3 4) ~ARR T
(<= 2 3 4 3) ~ARR 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 用来测试是否单一实数等于、小于或大于零。另外两个谓词，EVENP 和 ODDP，测试是否单一整数参数是偶数或奇数。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内置数学函数的开始。Lisp 也支持对数函数：LOG；指数函数：EXP、EXPT；基本三角函数：SIN、COS 和 TAN；它们的逆：ASIN、ACOS 和 ATAN；双曲：SINH、COSH 和 TANH；以及它们的逆：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。这是因为其本该如此——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 Unicode 在 Common Lisp 自身的标准化时期从标准化组织的观点来看只是昙花一现。

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，包括那些诸如“”、“（”和空格这样的特殊字符。尽管如此，以这种方式来写空格字符不是非常（人类）可读的；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。这样你应该写成 #\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（实现必须采用，如果字符集包含相应的字符）是 Tab、Page、Rubout、Linefeed、Return 和 Backspace。

10.7 字符比较

你可以对字符所做的主要事情，除了将它们放进字符串以外（我将在本章后面讨论这点），就是将它们与其他字符相比较。由于字符不是数字，你不能使用诸如“<”和“>”这样的数值比较函数。代替地，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=可以接受任一数量的参数并只在它们全是相同字符时返回真。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，注意到 <= 和 >= 被拼写成其逻辑等价形式 NOT-GREATERP 和 NOT-LESSP 而不是更确切的 LESSP-OR-EQUALP 和 GREATERP-OR-EQUALP。和它们的数值伙伴一样，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：

Table 10-1. 字符比较函数		
Numeric Analog	Case-Sensitive	Case-Insensitive
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL

^① 甚至像 Java 这种基于 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 字符的语言也会产生问题，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要求 21 位的表示。太惨了。

<	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了测试是否一个给定字符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。再一次，对于完整的细节，参见你所喜爱的 Common Lisp 参考。

10.8 字符串

如同早些所提到的，Common Lisp 中的字符串其实是一个复合数据类型，换句话说，一个字符的一维数组。因此，我将在下一章讨论到用来处理序列的许多函数时谈及许多你可以用字符串来做的事情，因为字符串只不过是一种序列。但是字符串也有其自己的字面语法和一个函数库用来进行特定与字符串的操作。我将在本章讨论字符串的这些方面并将其余的留给第 11 章。

正如你所看到的，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所支持的字符，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将如何被 Lisp 读取器读取。

Table 10-2 字面字符串		
Literal	Contents	Comment
"foobar"	foobar	Plain string.
"foo\"bar"	foo"bar	The backslash escapes quote.
"foo\\bar"	foo\bar	The first backslash escapes second backslash.
"\\\"foobar\\\""	“foobar”	The backslashes escape quotes.
"foo\bar"	foobar	The backslash “escapes” b.

注意到 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。因此你想要看到一个字符串的实际内容你需要使用诸如 `FORMAT` 这样设计用于打印人类可读输出的函数。例如，下面是当你在 REPL 中输入一个含有内嵌引号的字符串时所看到的：

```
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命名约定除了将前缀的 CHAR换成 STRING的函数来比较字符串。（见表 10-3）

Table 10-3. String Comparison Functions

Numeric Analog	Case-Sensitive	Case-Insensitive
=	STRING=	STRING-EQUAL
/=	STRING/=	STRING-NOT-EQUAL
<	STRING<	STRING-LESSP
>	STRING>	STRING-GREATERP
<=	STRING<=	STRING-NOT-GREATERP
>=	STRING>=	STRING-NOT-LESSP

尽管如此，跟字符和数字比较符所不同的是，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子字符串上。这些参数——:start1、:end1、:start2和:end2——指定了第一个和第二个参数字符串中子字符串的起始和终止位置（左闭右开区间）。这样，下面的

```
(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）来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——也就是说，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") ~ARR 3
```

如果第一个字符串是第二个字符串的前缀，返回值将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") ~ARR 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

^① 注意到，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——你应当写成(format t "~a" s)。

串 "bar" 和 "baz" 但却返回了 5，因为它是 "r" 在第一个字符串中的索引：

```
(string< "foobar" "abaz" :start1 3 :start2 1) ~ARR 5 ; N.B. not 2
```

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减字符。并且如同我前面所提到的由于字符串实际上是一种序列，因此我将在下章所讨论的所有序列函数都可用于字符串。例如，你可以用 LENGTH 函数来检查一个字符串的长度并获取和设定一个字符串中的个别字符，使用通用序列元素访问函数 ELT 或者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。但这些函数以及其他一些都是下一章的主题，那么让我们继续。

第11章 集合

和多数编程语言一样，Common Lisp 提供了用来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。前者被分别称为数组 (array)、列表 (list) 或元组 (tuple)；后者被命名为哈希表 (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 教材也都从对基于列表的 Lisp 集合的讨论开始。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，它可被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 将它们都视为一个更通用的抽象——序列——的子类型。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 的基本整数索引集合，它们分为两大类。定长向量与诸如 Java 这样的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，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。

```
(vector)    ~ARR #()
(vector 1)  ~ARR #(1)
(vector 1 2) ~ARR #(1 2)
```

^① 一旦你熟悉了 Common Lisp 提供的所有数据类型，你将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成其他更高效的东西，一旦你清楚了数据被使用的确切方式。

^② 向量被称为向量，而不是像其他语言里那样称为数组，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却过于笨重。

语法 `#(...)` 是 List 打印器和读取器所使用的向量字面表示形式，该语法允许你通过用 `PRINT` 打印它们并用 `READ` 读取它们来保存并恢复向量。你可以使用 `#(...)` 语法在你的代码中包含字面向量，但修改字面对象的效果是未定义的，因此你应当总是使用 `VECTOR` 或更通用的函数 `MAKE-ARRAY` 来创建你打算修改的向量。

`MAKE-ARRAY` 比 `VECTOR` 更加通用，因为你可以用它来创建任何维度的数组以及定长和变长向量。`MAKE-ARRAY` 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，该列表将含有一个数字，也就是向量的大小。出于方便，`MAKE-ARRAY` 也在单元数列表的位置上接受一个简单数字。如果没有其他参数，`MAKE-ARRAY` 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 为了创建一个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。这样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) ~ARR #(NIL NIL NIL NIL NIL)
```

`MAKE-ARRAY` 也是用来创建变长向量的函数。一个变长向量是一个比定长向量稍为更复杂些的向量；为了跟踪其用来保存元素的内存和可访问的槽位数量。一个变长向量还要跟踪实际存储在向量元素中的数量。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你为向量添加一个元素时下一个被填充位置的索引。

为了创建一个带有填充指针的向量，你可以向 `MAKE-ARRAY` 传递一个 `:fill-pointer` 参数。例如，下面的 `MAKE-ARRAY` 调用生成了一个带有五元素空间的向量；但它看起来是空的因为填充是零：

```
(make-array 5 :fill-pointer 0) ~ARR #()
```

为了向一个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*) ~ARR 0
*x* ~ARR #(A)
(vector-push 'b *x*) ~ARR 1
*x* ~ARR #(A B)
(vector-push 'c *x*) ~ARR 2
*x* ~ARR #(A B C)
(vector-pop *x*) ~ARR C
*x* ~ARR #(A B)
(vector-pop *x*) ~ARR B
*x* ~ARR #(A)
(vector-pop *x*) ~ARR A
*x* ~ARR #()
```

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` 传递另外一个关键参数 `:adjustable`。

```
(make-array 5 :fill-pointer 0 :adjustable t) ~ARR #()
```

^① 数组元素在其被访问前“必须”被赋值，是因为如果不这样做其行为将是未定义的；Lisp 不一定会报错。

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`，后者就像 `VECTOR-PUSH` 那样工作，只是它将在你试图向一个已满的向量——其填充指针等于底中存储的大小——中推入元素时自动扩展该数组。^①

11.2 向量的子类型

目前为止，你所处理的所有向量都是可以保存任意类型对象的通用向量。也有可能创建特化的向量使其限于保存特定类型的元素。一种使用特化向量的理由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地访问。不过目前让我们将注意力集中在几类特化向量上，它们本身就是重要的数据类型。

这些类型中你已经见过的一种——字符串是特定用来保存字符的向量。字符串特别重要，以至于它们到有自己的读写语法（双引号）和前一章我讨论过的一组特定于字符串的函数。但因为它们也是向量，所有我将在接下来几节里讨论的接受向量参数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，诸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，其尺寸是固定的，并且它们一定不能被修改。尽管如此，你可以使用 `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。我将不会讨论你可以在这里使用的所有可能的类型描述符；目前只要知道你可以通过传递符号 `CHARACTER` 作为 `:element-type` 来创建字符串就够了。注意到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，你可以写成这样：

```
(make-array 5 :fill-pointer 0 :adjustable t :element-type 'character) ~ARR ""
```

位向量——元素全部由 0 或 1 所组成的向量——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`，以及一个相对巨大的函数库。这些函数用来进行按位操作，例如将两个位数组“与”在一起，我将不会讨论它们。用来创建一个位向量所传递给 `:element-type` 类型描述符是符号 `BIT`。

11.3 作为序列的向量

正如早些提到的，向量和列表是抽象类型序列的两种具体子类型。我将在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——无论通用还是特化的——它们还可以应用于列表。

^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并且可调整的向量上使用 `VECTOR-PUSH-EXTEND`。你还可以使用函数 `ADJUST-ARRAY` 来以超出扩展向量长度的多种方式来修改可调整数组。

两个最基本的序列函数是：`LENGTH`，其返回一个序列的长度，以及`ELT`，其允许你通过一个整数索引来访问个别元素。`LENGTH`接受一个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`，元素（element）的简称，接受一个序列和一个从 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`将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*) ~ARR 3
(elt *x* 0) ~ARR 1
(elt *x* 1) ~ARR 2
(elt *x* 2) ~ARR 3
(elt *x* 3) ~ARR ERROR
```

`ELT`也是一个支持`SETF`的位置，因此你可以像这样设置一个特定元素的值：

```
(setf (elt *x* 0) 10)

*x* ~ARR #(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 还是提供了一个巨大的序列函数库。

一组序列函数可以允许你无需编写显式循环就可以表达诸如查找或过滤指定元素这类序列上的特定操作。表 11-1 总结了它们：

Table 11-1. Basic Sequence Functions

Name	Required Arguments	Returns
COUNT	Item and sequence	Number of times item appears in sequence
FIND	Item and sequence	Item or <code>NIL</code>
POSITION	Item and sequence	Index into sequence or <code>NIL</code>
REMOVE	Item and sequence	Sequence with instances of item removed
SUBSTITUTE	New item, item, and sequence	Sequence with instances of item replaced with new item

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)) ~ARR 3
(remove 1 #(1 2 1 2 3 1 2 3 4)) ~ARR #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4)) ~ARR (2 2 3 2 3 4)
```

```
(remove #\a "foobarbaz") ~ARR "foobrbz"
(substitute 10 1 #(1 2 1 2 3 1 2 3 4)) ~ARR #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4)) ~ARR (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz") ~ARR "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4)) ~ARR 1
(find 10 #(1 2 1 2 3 1 2 3 4)) ~ARR NIL
(position 1 #(1 2 1 2 3 1 2 3 4)) ~ARR 0
```

注意到 REMOVE 和 SUBSTITUTE 总是返回与其序列参数相同类型的序列。你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，这些函数在缺省情况下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。如果提供了这一函数，它将使用该函数代替缺省的对象等价性测试 EQL 来比较序列中的每个元素。^① 其次，使用 :key 关键字你可以传递一个单参数函数，其被调用在序列的每个元素上以解出一个关键字值，其随后被置于元素本身的位置上用于和其他项比较。注意到，尽管如此，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解出的关键字。

```
(count "foo" #("foo" "bar" "baz") :test #'string=) ~ARR 1
(find 'c #((a 10) (b 20) (c 30) (d 40)) :key #'first) ~ARR (C 30)
```

为了将这些函数的效果限制在序列参数的特定子序列上，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

如果非 NIL 的 :from-end 参数被提供，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) ~ARR (A 10)
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first :from-end t) ~ARR (A 30)
```

尽管如此，:from-end 参数和另一个关键字参数 :count——用于指定多少个元素被移除或替换——一起使用时可能影响 REMOVE 和 SUBSTITUTE 的行为。如果你指定了一个低于匹配元素数量的 :count，那么你从哪一端开始显然至关重要：

```
(remove #\a "foobarbaz" :count 1) ~ARR "foobrbaz"
(remove #\a "foobarbaz" :count 1 :from-end t) ~ARR "foobarbz"
```

并且尽管 :from-end 无法改变 COUNT 函数的结果，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*
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)
```

^① 另一个形参 :test-not 指定了一个两参数谓词，其可以像 :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。因此你可以并且也应该写成这样：

```
(count x sequence :test (complement #'some-test))
```

而不是下面这样：

```
(count x sequence :test-not #'some-test)
```

^② 注意到，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。

```

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。

<i>Table 11-2. Standard Sequence Function Keyword Arguments</i>		
Argument	Meaning	Default
: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

对于每一个刚刚讨论过的函数，Common Lisp 提供了两个高阶函数变体，在项参数的位置上，接受一个函数用来在序列的每个元素上被调用。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))           ~ARR 2
(count-if-not #'evenp #(1 2 3 4 5))        ~ARR 3
(position-if #'digit-char-p "abcd0001") ~ARR 4

```

```
(remove-if-not #'(lambda (x) (char= (elt x 0) #\\f))
#("foo" "bar" "baz" "foom")) ~ARR #("foo" "foom")
```

根据语言标准，这些`-IF-NOT`变体已经过时了。尽管如此，这种过时通常被认为是标准本身的欠考虑。不过，如果标准被再次修订的话，更有可能被去掉的是`-IF`而非`-IF-NOT`系列。比如收，有个叫`REMOVE-IF-NOT`变体就比`REMOVE-IF`更经常被使用。尽管它有一个听起来否定意义的名字，但`REMOVE-IF-NOT`实际上是一个肯定意义的变体——它返回满足谓词的那些元素。^①

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，除了`:test`，后者不再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`函数所解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) ~ARR 2
(count-if-not #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) ~ARR 3
(remove-if-not #'alpha-char-p
#("foo" "bar" "1baz") :key #'(lambda (x) (elt x 0))) ~ARR #("foo" "bar")
```

`REMOVE`函数家族还支持第四个变体`REMOVE-DUPLICATES`，其接受作为序列的单个必要参数，并将其中每个重复的元素移除到只剩下一个实例。

```
(remove-duplicates #(1 2 1 2 3 1 2 3 4)) ~ARR #(1 2 3 4)
```

11.6 整个序列的管理

有一类函数可以一次在整个序列（或多个序列）上进行操作。这些函数比我目前为止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相同类型的一个新序列。`COPY-SEQ`所返回的序列包含与其参数相同的元素，而`REVERSE`所返回的序列则含有相反顺序的相同元素。注意到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`创建一个含有任意数量序列连接在一起的新序列。不过，跟`REVERSE`和`COPY-SEQ`简单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。它的第一个参数是一个类型描述符，就像是`MAKE-ARRAY`的`:element-type`参数那样。这里你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)) ~ARR #(1 2 3 4 5 6)
```

^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，它们之间的区别在于：`: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) =
(concat-if #'(lambda (c) (eql char c)) string)
(count char string :test #'CHAR-EQUAL) =
(concat-if #'(lambda (c) (char-equal char c)) string)
```

^③如果你让`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。

```
(concatenate 'list #(1 2 3) '(4 5 6))      ~ARR (1 2 3 4 5 6)
(concatenate 'string "abc" '#(\d #\e #\f)) ~ARR "abcdef"
```

11.7 排序与合并

函数 `SORT` 和 `STABLE-SORT` 提供了两种对序列排序的方式。它们都接受一个序列和一个两参数谓词并返回该序列排序后的版本。

```
(sort (vector "foo" "bar" "baz") #'string<) ~ARR #("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的顺序，而 `SORT` 只保证结果是排序了的并可能重排等价的元素。

这两个函数都是所谓的破坏性 (destructive) 函数的例子。破坏性函数被允许——通常出于效率的原因——以或多或少的方式修改它们的参数。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（诸如将它赋给一个变量或将它传递给另一个函数；第二，除非你不再需要你传给破坏性函数的那个对象，否则你应该代替传递一个复本。我将在下一章里提及关于破坏性函数的更多内容。

通常你在排序以后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。但是这意味着你需要记得像下面这样书写：^①

```
(setf my-sequence (sort my-sequence #'string<))
```

而不只是这样：

```
(sort my-sequence #'string<)
```

这两个函数也接受一个关键字参数 `: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解出传给排序谓词的值的函数。被解出的关键字仅用于元素的顺序；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。并且出于同样原因，和 `CONCATENATE` 一样，`MERGE` 的第一个参数必须是一个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) #'<) ~ARR #(1 2 3 4 5 6)
(merge 'list  #(1 3 5) #(2 4 6) #'<)   ~ARR (1 2 3 4 5 6)
```

11.8 子序列管理

另一类函数允许你已有序列的子序列进行操作。其中最基本的是 `SUBSEQ`，其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：

^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。

```
(subseq "foobarbaz" 3) ~ARR "barbaz"
(subseq "foobarbaz" 3 6) ~ARR "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同的长度，那么两者中较短的那个将决定多少个字符被实际改变。

```
(defparameter *x* (copy-seq "foobarbaz"))

(setf (subseq *x* 3 6) "xxx") ; subsequence and new value are same length
*x* ~ARR "fooxxbaz"

(setf (subseq *x* 3 6) "abcd") ; new value too long, extra character ignored.
*x* ~ARR "fooabcbaz"

(setf (subseq *x* 3 6) "xx") ; new value too short, only two characters changed
*x* ~ARR "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。默认情况下该序列的每个元素都被设置到该值上；: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那样工作除了其第一个参数是一个序列而不是一个单独的项。

```
(position #\b "foobarbaz") ~ARR 3
(search "bar" "foobarbaz") ~ARR 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，你可以使用 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") ~ARR 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：一个 :key参数可以指定一个函数用来解出被比较的值；一个 :test 参数用于指定比较函数；而 :start1、:end1、:start2和 :end2参数可以指定两个序列中的子序列。另外一个设置为 T 的 :from-end参数可以指定序列以相反的顺序被搜索，从而导致 MISMATCH返回两个序列的相同后缀在第一个序列中开始位置的索引。

```
(mismatch "foobar" "bar" :from-end t) ~ARR 3
```

11.9 序列谓词

另外四个常见的函数是 EVERY、SOME、NOTANY和 NOTEVERY，其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一个谓词，其余的参数都是序列。这个谓词应当接受与所传递序列数量一样多的参数。序列的元素被传递给该谓词——每个序列中各取出一个元素——直到某个序列用光里所有的元素或是整体终止测试条件被满足：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。下面是一些仅测试在一个序列上的例子：

```
(every #'evenp #(1 2 3 4 5)) ~ARR NIL
```

```
(some #'evenp #(1 2 3 4 5)) ~ARR T
(notany #'evenp #(1 2 3 4 5)) ~ARR NIL
(notevery #'evenp #(1 2 3 4 5)) ~ARR 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)) ~ARR NIL
(some #'> #(1 2 3 4) #(5 4 3 2)) ~ARR T
(notany #'> #(1 2 3 4) #(5 4 3 2)) ~ARR NIL
(notevery #'> #(1 2 3 4) #(5 4 3 2)) ~ARR T
```

11.10 序列映射函数

最终, 序列函数的最后是通用映射函数。MAP和序列谓词函数一样, 接受一个 n-参数函数和 n 个序列。但并非返回布尔值, MAP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)) ~ARR #(10 18 24 28 30)
```

MAP-INTO 和 MAP 相似, 除了它并不产生给定类型的新序列, 而是将结果放置在作为第一个参数传递的一个序列中。这个序列可以和为函数提供值的序列中的一个相同。例如, 为了将几个向量 a、b 和 c 相加到其中一个向量里, 你可以写成这样:

```
(map-into a #'+ a b c)
```

如果这些序列的长度不同, 那么 MAP-INTO 将只影响相当于最短序列中元素数量的那些元素, 其中也包括将被映射到的那个序列。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尺寸。在一个对 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尺寸的向量。

最后一个序列函数是 REDUCE, 它可以做另一种类型的映射: 它映射在一个单一序列上, 先将一个两参数函数应用到序列的最初两个元素上, 再将函数返回的值和序列的后续元素继续用于该函数。这样下面的表达式将求和从 1 到 10 的整数:

```
(reduce #'+ #(1 2 3 4 5 6 7 8 9 10)) ~ARR 55
```

REDUCE 是一个非常有用的函数, 无论何时当你需要将一个序列提炼成一个单独的值, 你都有机会用 REDUCE 来写它, 而这通常都是表达你所想要的事情的一种相当简洁的方法。例如, 为了找出一个数字序列中的最大值, 你可以写成 (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, 在逻辑上放置在序列的第一个元素之前 (或是最后一个元素之后, 如果你同时指定了一个为真的 :from-end 参数)。

11.11 哈希表

Common Lisp 提供的另一个通用集合类型是哈希表。与提供里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)。当你向一个哈希表添加一个值时, 你在一个特定的键下保存它。以后你可以使用相同的键来获取该值。或者你可以将同一个

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` 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。这是一个好的缺省值，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，你将想要一个所谓的 `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` 参数所不同的是，`MAKE-HASH-TABLE` 的 `:test` 不能被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，一个等价性函数以及一个哈希函数用来从键中以一种和等价函数最终比较两个键时相兼容的方式计算出一个数值的哈希码。不过，尽管语言标准仅提供了使用标准等价函数的哈希表，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——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*) ~ARR NIL

(setf (gethash 'foo *h*) 'quux)

(gethash 'foo *h*) ~ARR QUUX
```

由于 `GETHASH` 当键在表中不存在时返回 `NIL`，没有办法从返回值中看出究竟是一个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——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，用来指示是否该键在哈希表中存在。由于多重返回值的工作方式，额外的返回值除非调用者用一个可以看见多值的形式显式地处理它，否则将被偷偷地丢掉。

我将在第 20 章里讨论多重返回值的进一步细节，但目前我将给你一个关于如何使用 `MULTIPLE-VALUE-BIND` 宏来利用 `GETHASH` 额外返回值的直观预览。`MULTIPLE-VALUE-BIND` 创建类似 `LET` 所做的变量绑定，并用一个形式所返回的多个值来填充它们。

下面的函数显示了你可以怎样使用 `MULTIPLE-VALUE-BIND`；它所绑定的变量是 `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

(show-value 'foo *h*) ~ARR "Value QUUX actually present."
(show-value 'bar *h*) ~ARR "Value NIL actually present."
(show-value 'baz *h*) ~ARR "Value NIL because key not found."
```

^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，然后才是索引，而 `GETHASH` 将键作为第一个参数，然后才是集合。

由于将一个键下面的值设置成 `NIL` 会把键留在表中，你将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。你也可以使用 `CLRHASH` 来完全清除一个哈希表中的所有键值对。

11.12 哈希表迭代

Common Lisp 提供了几种方式在哈希表项上迭代。其中最简单的方法是通过函数 `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，为了打印一个哈希表中所有的键值对，你可以像这样使用 `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指定（并且可能会很坏）但却有两个例外：你可以将 `SETF` 与 `GETHASH` 一起使用来改变当前项的值，并且你可以使用 `REMHASH` 来移除当前项。例如，为了移除所有其值小于 10 的项，你可以写成下面这样：

```
(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，后者将在第 22 章里讨论。^① 第一个 `MAPHASH` 表达式的等价 `LOOP` 形式将看起来像这样：

```
(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说得更多。例如，我还没有讨论到多维数组以及处理位数组的函数库。尽管如此，我在本章中涉及到的内容将满足多数通用编程场合的需要。现在终于到了查看列表这个让 Lisp 因此得名的数据结构的时候了。

^① `LOOP` 的哈希表迭代通常是用更基本的形式 `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。

第12章 LISP 名字的由来：列表处理

列表在 Lisp 中扮演重要的角色——无论是出于历史还是实践的理由。在历史上，列表曾经是 Lisp 最初的复合数据类型，尽管在很多年来它是这方面唯一的数据类型。现在一个 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。

现实来讲，列表仍然留在语言之中因为它们对特定问题提供了极佳的解决方案。其中一个问题是——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——是特定于 Lisp 的，这就可以解释为什么其他语言没有感觉到缺少 Lisp 风格列表所带来的不便。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。它们也是相当轻量的并且支持一种函数式的编程风格，后者是 Lisp 传统的另一个重要方面。

因此，你需要更加深入的理解列表；一旦你对列表的工作方式有了更加深刻地理解，你将会对何时应该或不应该使用它们具有更好的认识。

12.1 “没有列表”

Spoon Boy：不要试图弯曲列表。那是不可能的。相反... 只要试图认识真相就好。

Neo：什么真相？

Spoon Boy：没有列表。

Neo：没有列表？

Spoon Boy：然后你将看到弯曲的不是列表；而只是你自己。^①

理解列表的关键是认识到它们在很大程度上是一种构建在更基本数据类型的实例对象之上的观念。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` 接受两个参数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，否则一个点对将被打印成两个值在括号中用一个点所分隔的形式，一个所谓的“点对”。

`(cons 1 2) -> (1 . 2)`

^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。

一个点对单元中的两个值分别被称为 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。但即便那时它们也只被看作是用来实现这些操作的汇编助记符。尽管如此，这些名字缺乏意义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简单的任意数对而没有任何特别的语义。因此：

```
(car (cons 1 2)) -> 1
(cdr (cons 1 2)) -> 2
```

CAR 和 CDR 也都是支持 SETF 的位置——给定一个已有的点对单元，有可能将一个新的值赋值到它的任何一个值上。^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)
```

由于一个点对中的值可以是对任何类型对象的引用，因此你可以通过将点对连接在一起而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被保存在 CDR 中。链上最后一个单元的 CDR 为 NIL，后者——正如我在第 4 章所提到——同时代表空列表和布尔值 false。

这一安排毫无疑问是 Lisp 独有的；它被称为一个单链表。不过，很少有 Lisp 家族之外的语言对这种低微的数据类型提供如此广泛的支持。

因此当我说一个特定的值是一个列表时，我真正的意思是它要么是 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 则是对另一个列表的引用，后者可能是另一个点对单元或是 NIL，其中含有其余的元素。Lisp 打印器可以理解这种约定并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，一些图例可以起到很大帮助。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：

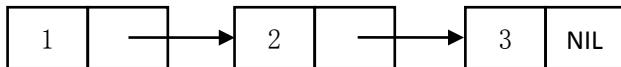


左边的方框代表 CAR，而右边的则代表 CDR。保存在一个特定点对单元中的值要么被画在适当的方框之内要么通过一个从方框指向别处的箭头来代表一个引用的值。^②例如，列表 (1 2 3)，

^① 当给定 SETF 的位置是一个 CAR 或 CDR 时，它将展开成一个对函数 RPLACA 或 RPLACD 的调用；一些守旧的 Lisp 程序员——和那些仍然使用 SETQ 的一样——仍然直接使用 RPLACA 和 RPLACD，但现代风格是使用 CAR 或 CDR 的 SETF。

^② 典型情况下，诸如数字这类简单对象被画在相应方框的内部，而更复杂的对象被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——尽管所有对象从概念上来讲都是按引用保存的，但特定的简单不可修改对象可以被直接保存在点对单元里。

由三个点对单元通过它们的 CDR链接在一起所构成的，将被图示成下面这样：



尽管如此，你使用列表的多数时候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，`LISP`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 `LISP`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)
```

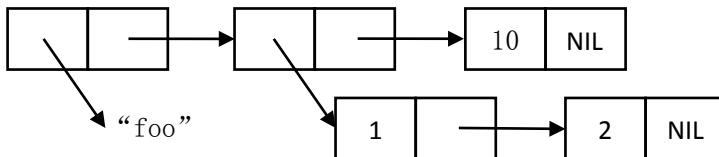
类似地，当你从列表的角度考虑问题时，你不需要使用没有意义的名字 `CAR` 和 `CDR`；`FIRST` 和 `REST` 分别是 `CAR` 和 `CDR` 的同义词，当你处理作为列表的点对时应该使用它们。

```
(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，因此也可以保存列表。并且一个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)
```

该列表的结构将看起来像这样：



由于列表可以将其他列表作为元素，因此你可以用它们来表示任意深度与复杂度的树。用这样的方法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的数据的例子是 `Lisp` 代码本身。在第 30 和第 31 章里你将编写一个 `HTML` 生成库，其中使用列表的列表来表示被生成的 `HTML`。我将在下一章里谈到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“列表操作函数”和“映射”两节里你将看到一些更重要的这类函数。尽管如此，它们在来自函数式编程的一些思想的上下文里将更容易被理解。

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所组成，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。并且由于一个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，当你看到一个像 `(+ 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所发生的事，因为没有什么东西可以改变求值该表达式的结果。

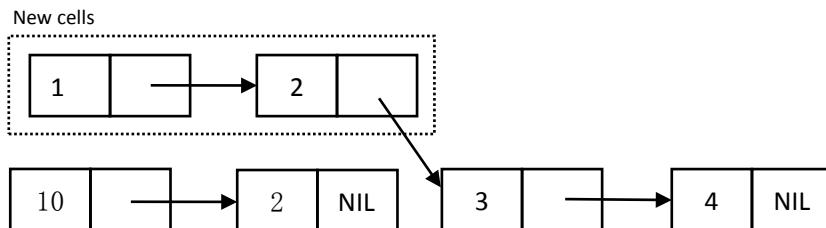
处理数字的函数是天生函数式的，因为数字都是不可改变的对象。另一方面，列表如同你刚刚看到的那样——通过 `SETF` 构成点对单元的 `CAR` 和 `CDR`——是可改变的。尽管如此，列表可以被当作函数式数据类型来对待，只要你将其值视为由它们包含的元素所决定的即可。这样，形式 `(1 2 3 4)` 所表示的任何列表在函数式意义上将等价于任何其他含有这四个值的列表，无论实际用来表示该列表的是什么点对单元。并且任何接受一个列表作为参数并返回完全依赖于列表内容的值的函数也可以同样被认为是函数式的。例如，序列函数 `REVERSE` 当给定列表 `(1 2 3 4)` 时总是返回一个列表 `(4 3 2 1)`。但由函数式等价的列表作为参数的不同 `REVERSE` 调用将返回函数式等价的结果列表。我将在“映射”那节里讨论的函数式编程的另一个方面是对高阶函数的使用：函数将其他函数作为数据对待，接受它们作为参数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。我将在后面讨论如何将函数式风格和其他编码风格在一起混用，但首先你应当理解函数式风格应用在列表上的一些微妙之处。

多数列表函数以函数式编写是因为它允许它们返回与其参数共享点对单元的结果。举一个具体的例子，函数 `APPEND` 接受任意数量的列表参数并返回一个含有所有其参数列表的元素的新列表。例如：

```
(append (list 1 2) (list 3 4)) → (1 2 3 4)
```

从一个函数式观点来看，`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。一个达到该目标的明显方式是创建一个由四个新的点对单元所组成的新列表。尽管如此，这样做了一些不必要的工作。相反，`APPEND` 实际上只生成两个新的点对单元作为后一个参数——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。没有原先的点对单元被修改过，并且结果确实是列表 `(1 2 3 4)`。唯一的美中不足是 `APPEND` 所返回的列表与列表 `(3 4)` 共享了一些点对单元。产生的结构看起来像这样：



一般而言，`APPEND` 必须复制除最后一个参数以外的所有其他参数，但它总是会返回一个与其最后一个参数共享结构的结果。

其他一些函数也相似地利用了列表共享结构的能力。像 `APPEND` 这样的一些函数被指定总是返回以特定方式共享结构的结果。其他函数被简单地允许根据具体实现的判断可以返回共享的结构。

12.3 “破坏性”操作

如果Common Lisp是一个纯函数式语言，那么故事就应该到这里为止了。不过，因为有可能在一个点对单元被创建之后通过对其CAR或CDR进行SETF操作来修改它，你需要想一想副作用是如何跟结构共享混合使用的。

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将产生大量的误解，因为存在两种相当不同的类型破坏性操作，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特定用来产生副作用的操作。所有对SETF的使用都在这种意义上是破坏性的，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所编写的代码中，因此它们不该用函数式术语来描述。尽管如此，如果你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，那么你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*))
```

在求值这些形式之后，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。

list-1	→ (1 2)
list-2	→ (3 4)
list-3	→ (1 2 3 4)

现在考察当你修改了*list-2*时会发生什么：

```
(setf (first *list-2*) 0) → 0
*list-2* → (0 4) ; as expected
*list-3* → (1 2 0 4) ; maybe not what you wanted
```

对*list-2*的改变由于共享的结构也改变了*list-3*：*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。

另一方面，另一种类型的破坏性操作，回收性操作，本意被用在函数式代码中。它们仅把副作用用作一种优化。特别地，它们在构造结果时会重用来自它们参数的特定点对单元。尽管如此，和诸如APPEND这种通过在返回的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始材料来重用，如有必要将修改其CAR和CDR来构造想要的结果。这样，回收性函数只有当原先的列表在对回收性函数的调用之后不再需要的情况下才可以被安全地使用。

^①习惯用语for-side-effect被用在语言标准中，而recycling则是我自己的发明；多数Lisp著作简单地将术语“破坏性”用在两类操作上，从而产生了我正试图消除的误解。

为了看到一个回收性函数是怎样工作的，让我们将 `REVERSE`——返回一个序列的逆序版本非破坏性函数——与它的回收性版本 `NREVERSE` 进行比较。由于 `REVERSE` 不修改其参数，所以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。但假如你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所引用，它们现在可以被垃圾收集了。不过，在许多 Lisp 实现中立即重用已有的点对单元而不是分配新的点对单元并让老的变成垃圾会更加高效。

`NREVERSE` 刚好允许你做到这点。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。`NREVERSE` 的确切副作用被有意地没有指定——它被允许修改列表中任何点对单元的任何 `CAR` 或 `CDR`——但一个典型的实现可能会沿着列表依次走上去并改变每一个点对单元的 `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。没有新的点对单元需要被分配，也没有垃圾被产生出来。

多数像 `NREVERSE` 这样的回收性函数都带有相应的产生相同结果但却没有破坏性的伙伴函数。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` 则是序列函数的 `REMOVE` 家族的回收性版本。

一般而言，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，除了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没有以可以足够信赖的方式被指定。

尽管如此，有一组回收性函数带有可以依赖的明确指定的副作用，这使得事情变得进一步复杂了。它们是 `NCONC`、`APPEND` 的回收性版本，以及 `NSUBSTITUTE` 和它们的 `-IF` 和 `-IF-NOT` 变种，序列函数 `SUBSTITUTE` 及其变体的回收性版本。

和 `APPEND` 一样，`NCONC` 返回其列表参数的连结体，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` 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。这样：

```
(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` 及其变种可信赖地沿着列表参数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。它然后返回最初的列表，其带有与 `SUBSTITUTE` 所计算得到的相同结果。^①

关于 `NCONC` 和 `NSUBSTITUTE` 需要记住的关键一点是，它们是你不能依赖于回收性函数的副作用这一规则的例外。忽略它们副作用的可信赖性，而像任何其他回收性函数一样来使用它们，只

^① 字符串函数 `NSString-CAPITALIZE`、`NSString-DOWNCASE` 和 `NSString-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原位修改其字符串参数。

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

尽管你可以在无论何时函数的参数在函数调用之后不会被使用的情况下使用回收性函数，值得注意的是每个回收性函数都是一把指向脚面的装了子弹的枪：如果你不小心将一个回收性函数用在了以后被用到的参数上，你肯定会失去一些脚趾。

使事情变得更糟的是，共享结构和回收性函数倾向于用于不同的目的。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，但是回收性却通过违反这一假设得以正常工作。或者，换另一种说法，共享结构基于你不在乎究竟由哪些点对单元构成列表这一前提，而使用回收性函数要求你精确地知道哪些点对单元会在哪里被引用到。

在实践中，回收性函数倾向于以一些惯用的方式来使用。其中最常见的回收性习惯用法是构造一个列表，它由一个在列表前端不断做点对分配操作的函数所返回，通常是像一个保存在局部变量中的列表里 `PUSH` 元素然后返回对其 `NREVERSE` 的结果。^①

这是一种构造列表的有效方式，因为每一次 `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`。由于列表完全在函数之内创建，完全不存在任何函数之外的代码带有对其任何点对单元的引用的风险。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 个数字所组成的列表：^②

```
(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)`

下一个最常见的回收性习惯用法^③ 是将回收性函数的返回值立即重新赋值到含有潜在被回收的值的位置上。例如，你将经常看到像下面这样的表达式，其使用了 `DELETE`——`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。尽管如此，即便这种习惯用法也需要在某种程度上小心地使用——如果 `foo` 和其他位置上所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，考查早先那两个共享了

^① 例如，在一次对 Common Lisp Open Code Collection (CLOCC)——一个由许多作者所写成的功能丰富的库集合——中所有回收性函数的使用情况的检查中，`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。

^② 当然，还有其他方法来做到相同的事。例如，扩展的 `LOOP` 宏可以尤其方便地做到这一点，并且很可能会生成比 `PUSH/NREVERSE` 版本甚至更高效的代码。

```
(defun upto (max)
  (loop for i below max collect i))
```

无论如何，重要的是要能够识别 `PUSH/NREVERSE` 习惯用法因为其相当普遍。

^③ 这一习惯用法在 CLOCC 代码库里占据了 30% 的回收性使用。

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)
```

你可以像下面这样将 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, 上述操作也改变了 *list-2*:

```
*list-2* → (0)
```

如果你使用 `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, 但需要小心的跟踪哪些函数返回共享的结构而哪些不是。

一般而言, 当操作列表时, 最好是以函数式风格来编写你自己的代码——你的函数应当只依赖于它们的列表参数的内容并且不应该修改它们。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。一旦你让你的代码工作起来了, 如果性能评估显示你需要进行优化, 你可以将非破坏性列表操作替换成它们的回收性同伴, 但只有当你确定参数列表不会从其他任何位置被引用时才这样做。

最后需要看到的一点是, 第 11 章里提到的排序函数 `SORT`、`STABLE-SORT` 和 `MERGE` 当应用于列表时也是回收性函数。^① 不过, 这些函数并没有非破坏性的同伴, 因此你需要排序一个列表而不破坏它, 那么你需要传给排序函数一个由 `COPY-LIST` 所生成的拷贝。无论哪种情况你都需要确保可以保存排序函数的结果, 因此原先的参数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!
```

12.5 列表处理函数

有了前面的这些背景知识, 现在你可以开始观察 Common Lisp 为处理列表所提供的函数库了。

你已经见过了获取列表中元素的基本函数: `FIRST` 和 `REST`。尽管你可以通过组合足够多的 `REST` 调用 (来深入列表) 和一个 `FIRST` 调用 (来解出元素) 来获得一个列表中的任意元素, 但这样可能有一点冗长。因此 Common Lisp 提供了以从 `SECOND` 到 `TENTH` 其他序数所命名的函数

^① `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。

来返回相应的元素。更一般地，函数 `NTH` 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`）尽管如此，注意到这些函数中没有哪个在计算机所完成的工作意义上比等价的 `FIRST` 和 `REST` 组合更高效——没有办法在无需跟随 `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` 函数是另一个你将不时用到的函数家族。每一个函数通过将由最多四个 `A` 和 `D` 所组成的序列放在 `c` 和 `r` 之间所命名，其中每个 `A` 代表一个对 `CAR` 的调用而每个 `D` 代表对 `CDR` 的调用。这样：

```
(caar list) ≡ (car (car list))
(cadr list) ≡ (car (cdr list))
(cadadr list) ≡ (car (cdr (car (cdr list))))
```

尽管如此，注意到许多这些函数仅当应用于含有其他列表的列表时才有意义。例如，`CAAR` 解出给定列表的 `CAR` 的 `CAR`；这样，传递给它的列表必须含有另一个列表作为其第一个元素。换句话说，这些函数其实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
```

这些函数现在不像以前那样常用了。并且即便最顽固的守旧 Lisp 黑客也倾向于避免过长的组合。尽管如此，它们被用在很多古老的 Lisp 代码上，因此至少值得理解它们的工作方式。^②

这些 `FIRST-TENTH` 和 `CAR`、`CADR` 等函数也可被用作 `SETF` 的位置如果你正在非函数式地使用列表。

（表 12-1）

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返回函数作为值的函数。你在前面章节里见过了几个高阶函数的例子，例如 `MAP`。尽管 `MAP` 可以同时被用在列表和向量上（也就是说，任何类型的序列），Common Lisp 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将函数应用到列表的元素还

^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
(caddr '(when (> x 10) (print x))) -> ((PRINT X))
```

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，它不要求 MAP所要求的结果类型参数。代替地，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参数的列表。除此之外，它和 MAP的行为相同：函数被应用在列表参数的相继元素上，每次函数的应用从每个列表中各接受一个元素。每次函数调用的结果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和 MAPCAR相似，除了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）。

MAPCAN和 MAPCON与 MAPCAR和 MAPLIST的工作方式相似，除了它们构造结果的方式。MAPCAR 和 MAPLIST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，每次函数调用可以向结果中提供任意数量的元素。^② MAPCAN像 MAPCAR那样传递列表的元素到映射函数中，而 MAPCON则像 MAPLIST那样传递点对单元。

最后，函数 MAPC和 MAPL是伪装成函数的控制构造——它们简单地返回它们的第一个列表参数，因此它们只有当映射函数做了有用的副作用时才是有用的。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，但这并不很正确——正如我早先提到的，你可以使用列表的列表来表示树。正如本章里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，其他函数可以允许你使用点对单元来表示树、集合以及两类键/值映射表。我将在下一章里讨论某些这类函数。

^① 因此，MAPLIST 是两个函数中更基本的——如果你只有 MAPLIST，你可以在它的基本上构造出 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)
```

第13章 超越列表：点对单元的其他用法

如同你在前面章节里所看到的，列表数据类型是由一组操作点对单元的函数所描述的。Common Lisp 也提供了函数可以让你处理由点对单元所构造出的诸如树、集合和查询表这样的数据结构。在本章里，我将给你一个关于这些其他的数据结构和管理它们的函数的简要介绍。和列表操作函数一样，这些函数中有很多将在你开始编写更复杂的宏以及需要将 Lisp 代码作为数据处理时会很有用。

13.1 树

处理从点对单元中构造出来的树形结构，与将它们作为列表来看待一样自然。毕竟另一种思考树的方式不就是将他们看作一个列表的列表么？一个将一组点对单元作为列表来看待的函数，与一个将同样的点对单元作为树来看待的函数，其区别就在于该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——成为列表结构——其查找方式是以第一个点对单元开始然后跟着 CDR 引用直到遇到一个 NIL。列表中的元素是由列表结构中所有点对单元的 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，那么被引用的单元将被视为作为外部列表元素的一个列表的头。^①另一方面，树结构的便利方式是同时跟随 CAR 和 CDR 引用，只要它们指向其他点对单元。因此一个树中的值就是该树结构中所有点对单元所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。

(图)

为了看到一个列表函数和一个树函数之间的区别，你可以考查函数 COPY-LIST 和 COPY-TREE 复制这些点对单元的方式。COPY-LIST 作为一个列表函数只拷贝那些构成列表函数的点对单元。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 不会拷贝子列表 (1 2)、(3 4) 或 (5 6)，如下图所示：

^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。这成为一个点列表，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)
一个没有点的列表——其最后一个 CDR 为 NIL——称为一个正则列表。

(图)

另一方面，`COPY-TREE`将会为图中的每一个点对单元生成一个新的点对单元并将他们以相同的结构连接在一起，如下图所示：

(图)

当原先的点对单元中引用了一个原子值时，拷贝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` 所产生的拷贝共同引用的唯一对象就是数字 1-6 以及符号 `NIL`。

另一个在一棵树的点对单元的 `CAR` 和 `CDR` 上进行遍历的函数是 `TREE-EQUAL`，其比较两棵树，并在两棵树具有相同的形状以及它们对应的叶子是 `EQL` 等价时（或者如果它们满足由 `:test` 关键字参数所提供的测试）认为它们相等。

其他一些以树为中心的函数是 `SUBSTITUTE` 和 `NSUBSTITUTE` 序列函数及其 `-IF` 和 `-IF-NOT` 变种的用于树的类似版本。函数 `SUBST` 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，除了其中所有旧项的实例都被替换成新的项。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的项——该函数在树的每一个原子值上被调用，并且当它返回真时，新树中的对应位置将被填充成新的值。`SUBST-IF-NOT` 也是一样，除了那些测试返回 `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，你应当只有在明确知道不存在修改共享结构的危险时才将这些函数作为它们的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，你可以将任何列表当作集合来看待——Common Lisp 提供了几个函数用于在列表上进行集合论意义上的操作。尽管如此，你应当在头脑中牢记由于列表结构的组织方式，这些操作在集合变得更大时将会越来越低效。

这就是说，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它们可能会比其他替代实现更为高效。如果性能评估告诉你这些函数成为你代码中的性能瓶颈，那么你总是可以将列表替换成构建在哈希表或位向量之上的集合。

为了构造一个集合，你可以使用函数 `ADJOIN`。`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

^① 可能看起来 `NSUBST` 家族的函数可以并且确实就地修改树。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`。

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——如果你打算修改一个特定的列表，你需要将 `ADJOIN` 返回的值赋值到该列表所来自的位置上。`PUSHNEW` 修改宏可以为你自动做到这点。

```
CL-USER> (defparameter *set* ())
*SET*
CL-USER> (adjoin 1 *set*)
(1)
CL-USER> *set*
NIL
CL-USER> (setf *set* (adjoin 1 *set*))
(1)
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` 来测试是否一个给定项在一个集合中。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。并且代替了当指定项存在时返回该项，它们返回含有该项的那个点对单元——换句话说，以指定项开始的子列表。当指定的项不在列表中时，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` 返回一个含有两个参数中可找到的所有元素所组成的列表。`UNION` 返回一个列表，其含有来自含有两个函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具有一个回收性同伴，起名字与其相同除了带有一个前缀 `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——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。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
```

^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。

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。两类基于点对的查询表被经常使用，两者都是我在前面的章节里提到过的。它们是关联表，也称为 *alist*，以及属性表，也称为 *plist*。尽管你不能将无论 *alist* 还是 *plist* 用于大型的表上——那种情况下你可以使用哈希表——但还是值得了解怎样使用它们，既是因为对于小型的表它们可以比哈希表更加高效，也是因为它们有一些它们自己的有用属性。

Alist 是一个数据结构，其映射一些键到值上同时也支持反向查询，当给定一个值时找出一个对应的键。*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* 在本质上是一个列表，其每一个元素本身都是一个点对单元。每个元素可以被想象成一个键值对，其中键保存在点对单元的 *CAR* 中而值则在 *CDR* 中。例如，下面是一个将符号 *A* 映射到数字 1，*B* 到 2，以及 *C* 到 3 的 *alist* 的方框和箭头图例：

(图)

除非 *CDR* 中的值是一个列表，否则代表键值对的点对单元表示成 S-表达式时将是一个点对 (dotted pairs)。例如上图中所表示的 *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，你可以简单地将 *ASSOC* 的结果传给 *CDR*。

```
CL-USER> (cdr (assoc 'a '((a . 1) (b . 2) (c . 3))))
1
```

缺省情况下给定的键使用 *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，如果你想要使用字符串的键，你可以写成这样：

```
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是 *EQL* 等价的。

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)))
NIL
```

由于 *ASSOC* 搜索列表时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)
```

你可以像下面这样使用 `cons` 向一个 `alist` 的前面添加一个键值对：

```
(cons (cons 'new-key 'new-value) alist)
```

尽管如此，出于方便起见，Common Lisp 提供了函数 `ACONS`，它可以让你写成这样：

```
(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。如果你想要修改一个 `alist`，你需要写成要么这样：

```
(setf alist (acons 'new-key 'new-value alist))
```

或是这样：

```
(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的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。尽管如此，由于 `alist` 的基本机制是如此轻量，对于小型的表来说 `alist` 可以在性能上超过哈希表。另外，`alist` 在如何做查询方面给了你更多的灵活性。我已经提到了 `ASSOC` 接受 `:key` 和 `:test` 关键字参数。当这些还不满足你的需要时，你还可以使用 `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。并且还有另外三个函数——`RASSOC`、`RASSOC-IF` 和 `RASSOC-IF-NOT`——和对应的 `ASSOC` 系列函数相似，除了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，除了代替拷贝整个树结构，它只拷贝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` 和它的拷贝将同时含有相同的对象作为键和值，哪怕是这些键或值刚好也由点对单元所构成。

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：

```
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，它看起来像这样：

（图）

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，函数 `GETF`，其接受一个 `plist` 和一个键并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。

不像 `ASSOC` 那样，其使用 `EQL` 作为缺省测试并允许通过一个 `:test` 参数提供一个不同的测试函数，`GETF` 总是使用 `EQ` 来测试是否所提供的键匹配 `plist` 中的键。因此，你应该从不使用数字和字符作为 `plist` 中的键；正如你在第 4 章里看到的，`EQ` 对于这些类型的行为在本质上是未定义的。从实践上来讲，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”，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所关联的值。`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)
```

为了从一个 `plist` 中移除一个键值对，你可以使用宏 `REMF`，它将作为其第一个参数给定的位置设置成一个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> (remf *plist* :a)
T
CL-USER> *plist*
NIL
```

和 `GETF` 一样，`REMF` 总是使用 `EQ` 来比较给定的键和 `plist` 中的键。

由于 `plist` 经常被用于你想要从同一个 `plist` 中解出几个属性的场合，Common Lisp 提供了一个函数，`GET-PROPERTIES`，它使得从单一 `plist` 中解出多个值更加高效。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，解出想要的属性，而无需持续地从列表的开始处重新扫描。例如，下面的函数有效地处理——使用假想的函数 `process-property`——对于一个给定的键的列表在一个 `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` 的最后一件特别的事情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` 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。尽管如此，你很少需要关心整个 `plist`；更通常的情况是你将使用函数 `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) ~EQU (getf (symbol-plist 'symbol) 'key)
```

和 GETF一样，GET也可以用 SETF来操作，因此你可以向下面这样将任意信息附加到一个符号上：

```
(get 'symbol 'key) ~EQU (getf (symbol-plist 'symbol) 'key)
```

为了从一个符号的 plist 中移除一个属性，你可以要么使用 SYMBOL-PLIST的 REMF或是更便捷的函数 REMPROP。^①

```
(remprop 'symbol 'key) ~EQU (remf (symbol-plist 'symbol key))
```

可以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，一个你将在第 24 章里编写的宏将向名字中附加信息以便同一个宏的其他实例将其解出并用于生成它们的展开式。

13.4 DESTRUCTURING-BIND

最后一个由于你将在后续章节里使用因此我需要谈及的用于拆装列表的工具是 DESTRUCTURING-BIND宏。这个宏提供了一种解构 (destrucrcture) 任意列表的方式，类似于宏形参列表分拆它们的参数列表的方式。一个 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所支持的任何参数类型，诸如 &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)) ~ARR (:X 1 :Y 2 :Z 3)

(destructuring-bind (x y z) (list 1 (list 2 20) 3)
  (list :x x :y y :z z)) ~ARR (: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)) ~ARR (: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)) ~ARR (: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)) ~ARR (:X 1 :Y1 2 :Y2 NIL :Z 3)

(destructuring-bind (&key x y z) (list :x 1 :y 2 :z 3)
```

^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，&environment 参数，而 DESTRUCTURING-BIND 不支持。尽管如此，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担心它。

```
(list :x x :y y :z z)) ~ARR (: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)) ~ARR (:X 3 :Y 2 :Z 1)
```

一种你可以在 `DESTRUCTURING-BIND` 中使用，也可以在宏参数列表中使用的参数类型，尽管我没有在第 8 章里提到过，是 `&whole` 参数。如果被指定，它必须是参数列表中的第一个参数，并且它被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` 参数存在那样解出列表中的指定部分。一个将 `&whole` 与 `DESTRUCTURING-BIND` 一起使用的例子看起来像这样：

```
(destructuring-bind (&whole whole &key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z :whole whole))
~ARR (: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，它是你将在第 31 章里开发的 HTML 生成库的一部分。不过，在你到达那里之前，我还要谈及更多的一些主题。在关于点对单元的两章相当 Lisp 化的主题之后，现在你将进入到关于如何处理文件和文件名的相对乏味的事情上了。

^① 当一个 `&whole` 参数被用在一个宏参数列表中时，它所绑定的形式将是整个宏形式，包括该宏的名字。

第14章 文件和文件 I/O

Common Lisp 提供了一个用于处理文件的丰富的函数库。在本章里我将集中在少数基本的文件相关任务上：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 机制与其它语言中相似。Common Lisp 为读写数据提供了一个流的抽象和一个称为路径名（pathnames）的抽象，它们以一种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，诸如 S-表达式的能力。

14.1 读写文件数据

最基本的文件 I/O 任务是读取一个文件的内容。你可以通过 `OPEN` 函数获得一个流并从中读取文件的内容。缺省情况下，`OPEN` 返回一个基于字符的输入流，你可以将它传给许多函数来读取文本中的一个或多个字符：`READ-CHAR` 读取一个单一字符；`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。当你完成了对流的操作以后你可以使用 `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“文件名”那一节里所看到的那样，Common Lisp 提供了许多表示一个文件名的方式，在最简单的方式是使用一个含有以本地文件命名语法所表示的文件名的字符串。因此，假设 `/some/file/name.txt` 是一个文件，那么你可以像下面这样打开它：

```
(open "/some/file/name.txt")
```

你可以使用返回的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中可能会出现一些错误，该文件可能不存在或者你可能在读取时无意中遇到了文件结尾。缺省情况下，`OPEN` 和 `READ-*` 系列函数将在这些情况下报错。在第 19 章里，我将讨论如何从这类错误中恢复。不过眼下有一个更加轻量的解决方案：每一个这样的函数都接受参数用来修改这些异常情况下的行为。

如果你想要打开一个可能不存在的文件而不想让 `OPEN` 报错，那么你可以使用关键字参数 `:if-does-not-exist` 来指定一个不同的行为。三个可能的值是 `:error`，报错（缺省值）；`: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——READ-CHAR、READ-LINE 和 READ——都接受一个可选的参数，其缺省值为真，它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其第三个参数的值，缺省为 NIL。因此你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 中的 R 的函数是同一个，并且它被用于读取 Lisp 源代码。每次它被调用时，它读取单一的 S-表达式，跳过空格和注释，然后返回由 S-表达式所代表的 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：一个数字的列表，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
```

如同你在第 3 章里所看到的，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需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的添加注释——如同前面的示例所演示的那样。并且由于 S-表达式被设计成是人类可读的，它也适用于诸如配置文件等事务的良好格式。^①

^① 尽管如此，注意到 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

缺省情况下 OPEN返回字符流，它将底层字节根据特定的字符编码模式转化成字符。^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从 0 到 255 之间的整数。READ-BYTE和字符读取函数一样也支持可选的参数来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。在第 24 章里你将构建一个库，它允许你使用 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，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的元素类型的元素。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，它可以通过在调用 OPEN时使用一个值为 :output的 :direction关键字参数来获取。当打开一个用于输出的文件时，OPEN假设该文件不应该已存在并将在文件存在时报错。尽管如此，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来替换已有的文件。传递 :append将导致 OPEN打开已有的文件并保证新数据将被写到文件结尾处，而 :overwrite返回一个从文件开始处开始的流从而覆盖已有的数据。而传递 NIL将导致 OPEN在文件已存在时返回 NIL而不是一个流。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 向流中写一个单一字符；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字符。两个不同的

^① 缺省情况下 OPEN 使用当前操作系统的字符默认编码，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”类型并不是固定尺寸的，由于 Lisp 曾经运行在不同时期的体系结构上，其字节长度从 6 位到 9 位之间。更不用说还有 PDP-10 计算机，其带有可独立寻址的长度从 1 到 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 或 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。

函数可以只打印一个换行：`TERPRI`——“终止打印 (terminate print)”的简称——无条件地打印一个换行字符，而 `FRESH-LINE` 打印一个换行字符除非该流已经在一行的开始处。`FRESH-LINE` 在你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虚假换行时很有用。例如，假设你有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。但假设如果这两个函数被依次调用，你不希望在两个输出操作之间产生一个空行。如果你使用 `FRESH-LINE` 在第二个函数开始处，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，它将不会产生一个额外的换行。

一些函数将 Lisp 数据输出成 S-表达式：`PRINT` 打印一个 S-表达式，前缀一个换行以及一个空格；`PRIN1` 只打印 S-表达式；而函数 `PPRINT` 像 `PRINT` 和 `PRIN1` 那样打印 S-表达式，但使用的是“美化打印器 (pretty printer)”，后者试图将输出以一种赏心悦目的方式打印出来。

尽管如此，并非所有的对象都可以以一种 `READ` 可以理解的形式打印出来。变量 `*PRINT-READABLY*` 控制当你试图使用 `PRINT`、`PRIN1` 或 `PPRINT` 来打印这样的一个对象时将会发生什么。当它是 `NIL` 时，这些函数将以一种可以保证会导致 `READ` 在试图读取时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是打印该对象。

另一个函数 `PRINC` 也会打印 Lisp 对象，但却以一种适合人类使用的方式来工作。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` 打开文件时带有和你读取时相同的值为 `'(unsigned-byte 8)` 的 `:element-type` 参数。然后你可以使用 `WRITE-BYTE` 向流中写入单独的字节。

批量输出函数 `WRITE-SEQUENCE` 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，无论是字符还是字节。和 `READ-SEQUENCE` 一样，该函数相比每次输出序列中的一个元素会更加高效一些。

14.5 关闭文件

任何编写过处理大量文件的代码的人都知道，当你处理完文件之后关闭它们是非常重要的，因为文件句柄往往是稀缺资源，如果你打开一些文件却不关闭它们，你将很快发现你不能再打开更多的文件了。^① 确保每一个 `OPEN` 都有一个匹配的 `CLOSE` 可能是足够显而易见的。例如，你总是可以像下面这样组织你的文件使用代码：

```
(let ((stream (open "/some/file/name.txt")))
  ;; do stuff with stream
  (close stream))
```

^① 有些人可能认为这在诸如 Lisp 的垃圾收集型语言里将不是一个大问题。在多数 Lisp 实现中当一个流变成垃圾之后都将会被自动关闭。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；它们不识别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。

尽管如此，这一方法有两方面的问题。一是容易出现错误——如果你忘记使用 CLOSE，那么代码将在每次运行时泄漏一个文件句柄。更重要的一点是，问题在于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，那么你就会在没有关闭流的情况下离开 LET 语句块。或者，如同你在 19 章里将要看到的，如果任何 CLOSE 之前的代码产生了一个错误，那么控制流可能跳出 LET 语句块到一个错误处理器中然后不再回来关闭那个流。

Common Lisp 对于如何确保特定代码总是被运行这一问题提供了一个通用的解决方案：特殊操作符 UNWIND-PROTECT，我将在第 20 章里讨论它。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，因此，Common Lisp 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* 中的形式在 *stream-var* 被绑定到一个文件流的情况下被求值，该流由一个对 OPEN 的调用使用 *open-argument* 作为其参数而打开。WITH-OPEN-FILE 会确保 *stream-var* 中的流在 WITH-OPEN-FILE 返回之前被关闭。这样你可以像下面这样编写代码从一个文件中读取一行：

```
(with-open-file (stream "/some/file/name.txt")
  (format t "~a~%" (read-line stream)))
```

为了创建一个新文件，你可以写成像下面这样：

```
(with-open-file (stream "/some/file/name.txt" :direction :output)
  (format stream "Some text.))
```

你将可能在你所使用的 90–99% 的文件 I/O 中使用 WITH-OPEN-FILE——你需要使用原始 OPEN 和 CLOSE 调用的唯一情况是当你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，你必须注意最终需要由你自己关闭这个流，否则你将泄漏文件描述符并且可能最终导致无法再打开更多文件。

14.6 文件名

目前为止，你只使用了字符串来表示文件名。尽管如此，使用字符串作为文件名会将你的代码捆绑在特定操作系统和文件系统上。同样地，如果你按照一个特定的文件命名模式的规则（比如说，使用 / 来分隔目录）来编程地构造文件名，那么你也会将你的代码捆绑在特定的文件系统上。

为了避免这种不可移植性，Common Lisp 提供了另一个文件名的表示方式：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。而在以本地语法所写成的字符串——称为名字字符串(namestring)——和路径名之间进行来回转换的责任被放在了 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通常使用的文件系统更加丰富多彩。这样带来的结果是，路径名抽象的某些细微之处在你只关心如何表示 Unix

或 Windows 文件名时就没有什么意义了。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的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，你都可以同时使用一个名字字符串或是路径名。具体使用哪一个在很大程度上取决于该名字的来源。由用户所提供的文件名——例如作为参数或是配置文件中的值——通常将是名字字符串，因为用户只知道它们所运行在的文件系统而不应当被期待关心 Lisp 如何表示文件名的细节。但通过编程方法所产生的文件名将是路径名，因为你可以移植地创建它们。一个由 OPEN 所返回的流也代表一个文件名，也就是当初用来打开该流的那个文件名。这三种类型的文件名在一起被总称为路径名描述符 (pathname designator)。所有内置的期待一个文件名参数的函数都接受所有三种类型的路径名描述符。例如，前面章节里所有你使用一个字符串来表示文件名的位置，你也同样可以传递一个路径名对象或一个流。

我们如何到达这里

存在于上世纪 70 和 80 年代的文件系统的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) 描述了如下的情形：

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。它们中的一些只支持大写字母，一些是大小写混合的，另一些是大小写敏感的但却自动做大小写转换（就像 Common Lisp）。它们中的一些将目录视为文件，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，没有设备的文件系统，以及诸如此类。

如果你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，如果你考察两种像 Windows 和 Unix 这样相似的文件系统，你可能已经开始注意到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。使用这种被涉及用来处理过去存在的广泛的文件系统的路径名抽象所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。

14.7 路径名如何表示文件名

路径名是一个使用 6 个组件来表示一个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的值，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

^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因为其对逻辑路径名 (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需命名特定的文件。逻辑路径名可以随后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需担心它们。

列表，其中带有关键字 :absolute 或 :relative 作为前缀。尽管如此，并非所有路径名组件都在所有平台上被需要——这也是路径名让许多新 Lisp 程序员感到无端复杂的原因之一。另一方面，你并不真的需要担心哪个组件在特定文件系统上是否可能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，通常你将通过让具体实现来解析一个文件系统相关的名字字符串到一个路径名对象，或是通过从一个已有的路径名中取得其多数组件来创建一个新路径名，来得到路径名对象。

例如，为了将一个名字字符串转化成一个路径名，你可以使用 PATHNAME 函数。它接受一个路径名描述符并返回一个等价的路径名对象。当该描述符已经是一个路径名时，它被简单地返回。当它是一个流时，最初的文件名被解出然后返回。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被解析。语言标准，作为一个平台中立的文档，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用路径分隔符——在 Unix 上是一个斜杠而在 Windows 上是一个斜杠或反斜杠——被分拆成基本元素。在 Windows 上驱动器字母将被放置在要么设备要么主机组件中。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，具体取决于是否该名字（忽略驱动器字母，如果有的话）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。最后一个元素将在最右边的点处被分拆开，如果有的话，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字。在 Windows 上 PATHNAME-HOST 和 PATHNAME-DEVICE 两者之一将返回驱动器字母。

和其他许多内置对象一样，路径名有它们自己的读取语法，#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是可移植的。

```
(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，为了表达给用户——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

^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；一些实现将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。

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件成一个本地目录名，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函数来构造任意的路径名。它针对每一个路径名组件都接受一个关键字参数并返回一个路径名，其中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，如果你想要你的程序是可移植的，你可能不会想要完全从手工生成路径名：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 OS X 来说就不是一个好的文件名，因为在那 /home/被称为 /Users/。

另一个不推荐完全从手工生成路径名的原因是，不同的实现以稍有不同的方式来使用路径名组件。例如，前面已经提到某些基于 Windows 的 Lisp 实现将驱动器字母保存在设备组件中而其他一些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。

与其从手工生成路径名，你还可以使用 MAKE-PATHNAME的关键字参数 :defaults基于一个已有的路径名来构造一个新路径名。通过这个参数你可以提供一个路径名描述符，它将提供没有被其他参数所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)
```

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将是健壮的，无论文件名是否带有驱动器字母或是它被保存在一个路径名的哪个组件上。^③

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：

^① 由 FILE-NAMESTRING 所返回的名字在支持版本的文件系统上也含有版本组件。

^② 主机组件可能不会缺省为 NIL，但如果不是的话，它将是一个不透明的由实现定义的值。

^③ 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 文件大的多的版本号。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的话最好加上它。

```
(make-pathname :directory '(:relative "backups") :defaults input-file)
```

不过，这样将创建出一个整个目录组件是相对目录 `backups/` 的路径名，而不管 `input-file` 可能有的任何目录组件。例如：

```
(make-pathname :directory '(:relative "backups")
  :defaults #p"/foo/bar/baz.txt") → #p"backups/baz.txt"
```

尽管如此，有时你想要组合两个路径名，其中的至少一个带有相对的目录组件，将它们的目录组件组合在一起。例如，假设你有一个诸如 `#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 `MERGE-PATHNAMES`。

`MERGE-PATHNAMES` 接受两个路径名并合并它们，将第一个路径名中的任何 `NIL` 组件使用来自第二个路径名的对应值填充，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` 会特别地对待目录组件：如果第一个组件名的目录是相对的，那么产生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，获得一个相对于特定根目录的文件名，你可以使用有用的函数 `ENOUGH-NAMESTRING`。

```
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/") →
"html/foo/bar.html"
```

随后你可以组件 `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` 也被用来实现访问文件系统中的文件的标准函数内部用于填充不完全的路径名。例如，假设你生成了一个只有名称和类型的路径名：

```
(make-pathname :name "foo" :type "txt") → #p"foo.txt"
```

如果你试图使用这个路径名作为 `OPEN` 的一个参数，那么缺失的组件，诸如目录，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 将通过合并给定的路径名与变量 `*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所决定但通常是一个路径名，其目录组件表示 Lisp 启动时所在的目录，主机和设备组件如果需要的话也带有适当的值。如果只有一个参数来调用的话，`MERGE-PATHNAMES` 将把该参数与 `*DEFAULT-PATHNAME-DEFAULTS*` 值进行合并。例如，如果 `*DEFAULT-PATHNAME-DEFAULTS*` 是 `#p"/home/peter/"`，那么你将得到下面的结果：

```
(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，你需要注意一件事。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)，将一个目录像其他任何文件一样对待，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
```

当你用 MAKE-PATHNAME 创建路径名时，你可以控制所得到的形式，但你需要在处理名字字符串时加以小心。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，假设你提示用户输入一个保存文件的目录而它们输入了 "/home/peter"。如果你将该值作为 MAKE-PATHNAME 的 :defaults 参数像下面这样传递：

```
(make-pathname :name "foo" :type "txt" :defaults user-supplied-name)
```

那么最后你将把文件保存成 /home/foo.txt 而不是想要的 /home/peter/foo.txt，因为名字字符串中的 "peter" 当 user-supplied-name 被转化成一个路径名时将被放在名称组件中。在下一章里我将讨论的路径名可移植库中，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。使用该函数你可以可靠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 打开文件用于读写，你偶尔也将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，以及获取关于一个文件的信息，诸如谁拥有它、何时它被最后修改，以及它的长度。这就是路径名抽象所带来的通用性开始带来痛苦的地方：因为语言标准没有指定那些与文件系统交互的函数映射到任何特定的文件系统上，这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那些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平滑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——的一个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所命名的文件存在，

那么 `PROBE-FILE` 将返回该文件的真实名称 (`truename`)，一个将诸如解析符号链接这类文件系统层面转换进行过的路径名。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是否一个目录存在。同样，Common Lisp 也不提供一种可移植的方式来测试是否一个给定文件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，它不但可以测试一个目录是否存在，还可以告诉你一个给定的名字究竟是一个文件名还是目录名。

类似的，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` 函数来平滑这些区别。

`DELETE-FILE` 和 `RENAME-FILE` 做它们的名字所建议的事情。`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所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)
  ...)
```

注意到如果你传给 `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` 返回文件被上次写入的时间，表示形式是自从格林威治标准时间 (GMT) 1900 年 1 月 1 日午夜起的秒数，而 `FILE-AUTHOR` 在 Unix 和 Windows 上返回该文件的拥有者。^②

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下的长度。尽管如此，由于在大多数当今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。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。^③

^① 更多的关于错误处理的内容参见第 19 章。

^② 对于需要访问特定操作系统或文件系统上其它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。位于 <http://common-lisp.net/project/osicat/> 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一个 POSIX 操作系统上的多数 Common Lisp 上。

^③ 一个文件中的字节和字符的数量就算你没有在使用多字节字符编码时也可能是不同的。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）字符的数量通常将小于字节的数量。如果你真想知道文件中的字符数量，你将不得不亲自数一下并使用类似下面这样的代码：

```
(with-open-file (in filename)
  (loop while (read-char in nil) count t))
```

或者可能是下面这样更高效的代码：

```
(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时，该函数返回文件中的当前位置——已经被读取或写入该流的元素的数量。当以两个参数被调用时，流和一个位置描述符，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、关键字 :end，或者一个非负的整数。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，对于字符流来说事情变的有一点复杂，因为字符编码因素的存在。当你需要在一个文本数据的文件中做跳转时，你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，它可以使用函数 MAKE-STRING-INPUT-STREAM 和 MAKE-STRING-OUTPUT-STREAM 来创建。

MAKE-STRING-INPUT-STREAM 接受一个字符串和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，然后返回一个字符流可被传递到任何诸如 READ-CHAR、READ-LINE 或 READ 这些基于字符的输入函数中。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 创建一个流可被用于 FORMAT、PRINT、WRITE-CHAR、WRITE-LINE 等。它不接受参数。无论你写了什么，一个字符串输出流将被累积到一个字符串中，该字符串可以随后通过函数 GET-OUTPUT-STREAM-STRING 来获取。每次你调用 GET-OUTPUT-STREAM-STRING 时，该流的内部字符串会被清空，因此你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——它从一个给定字符串中创建一个字符串输入流并在该流绑定到你所提供的变量的情况下执行它主体中的形式。例如，与其使用 LET 形式并带有显式的 UNWIND-PROTECT，你可以写成下面这样：

```
(with-input-from-string (s "1.23")
  (read s))
```

```
(with-open-file (in filename)
  (let ((scratch (make-string 4096)))
    (loop for read = (read-sequence scratch in)
          while (plusp read) sum read)))
```

`WITH-OUTPUT-TO-STRING`与之相似：它绑定一个新创建的字符串输出流到一个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`所返回的值。

```
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往一组输出流上，这些流是作为参数提供给它的构造函数`MAKE-BROADCAST-STREAM`的。^①与之相反的，`CONCATENATED-STREAM`是一个输入流，它从一组输入流中接受其输入，在遇到每个流的结尾时从一个流移动到另一个。`CONCATENATED-STREAM`使用函数`MAKE-CONCATENATED-STREAM`来构造，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流可被同时用于输入和输出函数。

在一个`TWO-WAY-STREAM`中，你所做的每一次读取将返回从底层输入流中所读取的数据，而每次写入将把数据发往底层的输出流上。一个`ECHO-STREAM`本质上以相同的方式工作，除了所有从底层输入流中读取的数据也被回显到输出流中。这样，一个`ECHO-STREAM`中的输出流将含有会话双方的一个副本。

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 标准并没有提及有关网络 API 的内容，但多数实现都支持 socket 编程并且通过将 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。^②

现在你已准备好开始构造一个库来平滑不同 Common Lisp 实现在基本路径名函数行为上的一些区别了。

^① `MAKE-BROADCAST-STREAM`可以通过不带参数调用它来生成一个数据黑洞。

^② Common Lisp 的标准 I/O 机制中所缺失的最大一部分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。不幸的是，当时没有时间解决由这份草案所产生的所有问题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 都基于 Gray 的草案。另一种更新的 API 称为 Simple Streams，它由 Franz 所开发并包括在 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。

第15章 实践：一个可移植路径名库

如同我在前面章节里所讨论的那样，Common Lisp 提供了一种称为路径名的抽象，它可以将你从不同操作系统和文件系统命名文件的方式中隔离出来。路径名提供了一个有用的 API 来管理作为名字的名字，但是当它涉及到实际与文件系统交互的函数时，事情变得有些复杂了。

如同我曾经提到的，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。

尽管如此，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，因此不能写出一个库来对跨越不同实现的多数常见操作提供一个一致的接口。这就是你在本章的任务。编写这个库不但可以让你获得几个未来章节中将会用到的有用函数，还可以给你一个机会来学习如何编写处理不同实现间区别的代码。

15.1 API

该库将要支持的基本操作将是获取一个目录中的文件列表以及检测一个给定名字的文件或目录是否存在。你也将编写一个函数用于递归地遍历一个目录层次，并在目录树中的每个路径名上调用一个给定的函数。

理论上讲，这些列目录和文件存在性操作已经由标准函数 `DIRECTORY` 和 `PROBE-FILE` 所提供了。不过正如你将看到的那样，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——因此你将期望编写一个新的函数在不同实现间提供一致的行为。

15.2 *FEATURES* 和读取期条件化

在你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，我需要首先向你介绍编写实现相关代码的手法。

尽管你所编写的多数代码，从运行在任何符合标准的 Common Lisp 实现上将产生相同行为这

个意义上都是“可移植的”，但你可能偶尔需要依赖于实现相关的功能或是为不同实现编写稍有不同的代码。为了允许你在不完全破坏代码可移植性情况下做到这点，Common Lisp 提供了一种称为读取期条件化的机制，从而允许你条件地包含基于当前所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 读取器所理解的附加语法所构成。`*FEATURES*` 是一个符号的列表；每一个符号代表存在于当前实现或底层平台的一个“特性”。这些符号随后被用在特性表达式中根据表达式中的符号是否存在于 `*FEATURES*` 求值成真或假。最简单的特性表达式是一个单个符号；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，如果你想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，那么你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，他首先读取特性表达式并按照我刚刚描述的方式求值。当一个跟在 `#+` 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。`#-` 以相同的方式工作除了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所定义的。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，一个不带有包限定符的名字将被读取成一个关键字符号。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 中上述代码将被读取为好像原本就写成下面这样：

```
(defun foo ()
  (do-one-thing))
```

而在 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没有简单方法来编写一个 fall-through case。例如，如果你通过在 `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

为不同实现的不同版本付出任何运行期代价。另外，当读取器跳过条件化的表达式时，它不会 intern 其中的符号，因此被跳过的表达式可以安全地含有在其他实现中可能不存在的包中的符号。

对库打包

从包的角度来说，如果你下载了该库的完整代码，你将看到它被定义在一个新的包中，`com.gigamonkeys.pathnames`。我将在第 21 章里讨论定义使用包的细节。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带有相同的名字，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，将可能会得到关于破坏了已有定义的错误或警告。为了避免这种可能性，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.pathnames
  (: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))
```

并加载它。然后在 REPL 中或者在你输入来自本章的定义的文件顶端，输入下列表达式：

```
(in-package #:com.gigamonkeys.pathnames)
```

将库以这种形式打包，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所使用，你在未来几章中将会看到这一点。

15.3 列目录

你可以将用于列一个单独目录的函数 `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，其带有一个或更多的含有特殊值 `: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——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。

`DIRECTORY` 函数有两个你需要在 `list-directory` 中解决的问题。其中主要的一个是，其行为的特定方面在不同的 Common Lisp 的实现间具有相当大的区别，即便在相同的操作系统上。另一个问题在于，尽管 `DIRECTORY` 提供了一个强大的用于列出文件的接口，但正确的使用它需要对路

[#]-之后的 or 特性表达式中添加同样的特性，否则 `ERROR` 形式将会在你的新代码运行以后被求值。

^① 另一个特殊值 `:wild-inferiors` 可能作为一个通配路径名的目录组件的一部分出现，但你在本章里不需要它们。

径名抽象具有某些相当细致的理解才行。在这些不同细微之处和不同实现的特征所影响下，实现编写可移植代码来使用 `DIRECTORY` 去做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会是令人沮丧的经历。你可以通过编写 `list-directory` 来一次性地将所有这些细节和特征处理掉，并从此忘记它们。

一个我曾在第 14 章里讨论过的细节是将一个目录的名字表示成路径名的两种方式：目录形式和文件形式。

为了让 `DIRECTORY` 返回一个 `/home/peter/` 中文件的列表，你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而其名称和类型组件需要是 `:wild`。这样，为了列出 `/home/peter/` 中的文件，看起来你需要写成下面这样：

```
(directory (make-pathname :name :wild :type :wild :defaults home-dir))
```

其中的 `home-dir` 是一个代表 `/home/peter/` 的路径名。如果 `home-dir` 是以目录形式表示的，那么上述写法将正常工作。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"` 被创建出来——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`。

为了避免对目录表示形式作显式的转换，你可以定义 `list-directory` 来接受任何形式的非通配路径名，它将随后被转化成适当的通配路径名。

为了帮助做到这点，你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 `directory-pathname-p` 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化成一个目录形式的路径名。

```
(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
          :directory (append (or (pathname-directory pathname) (list :relative))
                             (list (file-namestring pathname)))
          :name      nil
          :type      nil
          :defaults  pathname)
        pathname)))
```

现在看起来似乎你可以通过在一个由 `pathname-as-directory` 所返回的目录形式名字上调

^① 具体实现被允许返回 `: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所使用时。

用 `MAKE-PATHNAME` 来生成一个通配路径名并传给 `DIRECTORY`。不幸的是，没有那么简单，这需要感谢 CLISP 的 `DIRECTORY` 实现中的一个怪癖。在 CLISP 中，`DIRECTORY` 将不会返回那些没有扩展名的文件除非通配符中的类型组件是 `NIL` 而非 `:wild`。因此你可以定义一个函数 `directory-wildcard`，其接受一个目录或文件形式的路径名并返回给定实现下的一个适当的通配符，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` 的类型组件的路径名，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)))
```

注意到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` 被要么读取要么跳过；同样，在 `#+clisp` 之后，`NIL` 被要么读取要么跳过。

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，该函数将在 SBCL、CMUCL 和 LispWorks 中正常工作。不幸的是，另外一些实现间的区别仍然需要被平滑处理。其中一点是，并非所有实现都将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你为 `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。CLISP 的 `DIRECTORY` 只有当它被传递一个以 `: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，因此你将需要使用不同的通配符调用 `DIRECTORY` 两次并组合结果。

一旦你让所有实现都返回目录了，你将发现它们在返回的目录名上有些是目录形式，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，这样你就可以只通过名字来区分子目录和正规文件。除了 Allegro 之外，所有实现都支持做到这点。另一方面，Allegro 要求你为 `DIRECTORY` 传递一个实现相关的值为 `NIL` 的关键字参数 `:directories-are-files` 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` 只是简单地将不同版本用读取期条件组合起来的事情了。

```
(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 CLISP 相关的，但由于不被任何其他实现所需要，因此你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，因此整个函数定义将被包含或不包含取决于是否 `:clisp` 存在于 `*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 测试一个文件的存在

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所代表的文件存在时返回一个等价的路径名，否则返回 `NIL`。它应当可以接受以无论目录还是文件形式表示的目录名，但应当总是返回一个目录形式的路径名，如果该文件存在并且是一个目录。这将允许你使用 `file-exists-p` 和 `directory-pathname-p` 来一同测试一个任意名字是否是一个文件或目录名。

从理论上讲，`file-exists-p` 和标准函数 `PROBE-FILE` 非常相似；确实，在一些实现里——SBCL、LispWorks 和 OpenMCL——`PROBE-FILE` 已经给了你 `file-exists-p` 的行为。但并且所有实现的 `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——它们将接受任何形式的目录名但不会返回一个目录形式的路径名，而只是简单地将传给它的参数返回。幸运的是，如果以目录形式传递一个非目录的名字给它，它返回 `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` 将在传递一个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：它将在传递一个文件形式的名字或者一个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` 而不是一个路径名。

但就算在 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 所用到的函数 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。该函数，尽管只被 CLISP 用到，但通常是有用的，因此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)
            :defaults pathname)
          pathname)
        pathname)))
```

15.5 遍历一个目录树

最后，为了完成这个库，你可以实现一个称为 walk-directory 的函数。不像前面定义的那些函数，这个函数不需要做任何事情来平滑实现间的区别；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，你将在后续章节里几次用到它。它将接受一个目录的名字和一个函数并在该目录下的所有文件的路径名上递归地调用该函数。它还将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上和正规文件一样调用该函数。:test 参数如果提供了的话，它指定另一个函数在主函数之前被调用在每一个路径名上；且只有当测试参数返回真时，主函数才会被调用。:test 参数的缺省值是一个总是返回真的函数，通过调用标准函数 CONSTANTLY 而生成。

^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。不幸地是，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。

```
(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。正如我所提到的那样，这些函数将在后面的章节里很有用，尤其是第 23 和 27 章，在那里你将使用 `walk-directory` 在含有垃圾信息和 MP3 文件的目录树中爬行，但在我们到达那里之前，我还需要谈论一下面向对象，接下来两章的主题。

第16章 重新审视面向对象：广义函数

由于 Lisp 的发明相对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 Common Lisp 是一种多么彻底的面向对象语言。Common Lisp 的先驱被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面向对象的思想，尤其是 Smalltalk 中所出现的形式，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 或 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 所贡献的特性包括了那些难以避免的相对难懂的 Lisp “语言作为语言的构造工具”这一哲学的具体表现。所有这些特性的完全覆盖超出了本书的范围，但在本章和下一章里我将描述其中最常用的特性并给出 Common Lisp 对象观点的概述。

你应当从一开始就注意到 Common Lisp 的对象系统提供了一个与许多其他语言中相当不同的面向对象原则的体现。如果你能够对面向对象背后的基本思想有一个深刻的理解，那么你将会感谢 Common Lisp 在实现这些思想时所采用的尤其强大和通用的方式。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，那么你应当不难理解这里的解释，尽管有时它可以帮助你忽略对于其他语言做同样事情方式的偶然比较。

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，定义数据类型然后将操作关联在那些数

^① 现在通常被认为是第一个面向对象语言的语言，Simula，被发明于 1960 年代早期，只比 McCarthy 的第一个 Lisp 晚了几年。尽管如此，面向对象直到 1980 年代 Smalltalk 的第一个广泛使用的版本发布以后才真正起飞，几年以后 C++ 才发布。Smalltalk 从 Lisp 那里获得了许多灵感并将它与来自 Simula 思想组合起来而产生出一种动态的面向对象语言，而 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++ 教派的人们倾向于认为 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 教派的人们则认为 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，Smalltalk 的发明者 Alan Kay 就曾被报道说，“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——不认为 Common Lisp 是真正面向对象的。当然，按照那样的定义，就算是 Smalltalk 这种无可争议的最早和最纯粹的面向对象语言也不再是面向对象的了。另一方面，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。

据类型上。特别的是，你希望可以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。在所有关于面向对象的介绍中所使用的经典事例是一个可被用于代表几种几何图形的对象之上的 `draw` 操作。`draw` 操作的不同实现可被用于绘制圆、三角形和矩形，而一个对 `draw` 的调用将实际绘制出一个圆、三角形或矩形，具体取决于 `draw` 操作被应用到的对象类型。`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无论调用方还是任何其他 `draw` 实现的代码。这一面向对象风格称为“多义性”，其源自希腊语 *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是基于类的；所有的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并且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，如同你在下一章里将要看到的，由称为槽（slot）的命名部分所组成。

类通过层次结构组织在一起，形成了所有对象的分类系统。一个类可以被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，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）——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 所建立的基本模式：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）所定义。在这些语言里，一个方法在一个特定对象上被调用，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被称为——使用来自 Smalltalk 的术语——消息传递（message passing）。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的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，因此相同的消息发往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`，用于向一个特定对象发送一条消息。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被写成像下面这样：

```
(send object 'foo)
```

而不是下面这样：

```
(foo object)
```

更重要的是，由于方法不是函数，他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一个方法，他不得不写成下面这样：

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

^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的成为作为类的 `T` 的一个实例。

而不是像这样：

```
(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做到的许多特性。

广义函数是 Common Lisp 对象系统的心脏，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

一个广义函数定义了一个抽象操作，指定了其名字和一个参数列表但却不提供实现。例如，下面就是你可能定义一个广义函数 `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；目前只需注意到该定义并不含有任何实际代码。

一个广义函数的广义性体现在它可以——至少在理论上——接受任何对象作为参数。^① 不过，一个广义函数本身并不能做任何事；如果你只是定义一个广义函数，那么无论你用什么参数来调用它，它都将会报错。一个广义函数的实际实现是由方法 (method) 所提供的。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，后者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，使其可以用于 `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。去掉实际的绘图绘制代码以后它们如下所示：

```
(defmethod draw ((shape circle))
  ...)
(defmethod draw ((shape triangle))
  ...)
```

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来找出可应用 (applicable) 的方法——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` 的那个方法将是可应用的，而如果你传递了一个 `triangle` 实例，那么在 `triangle` 上特化了 `shape` 的那个方法将被应用。在简单的情况下，只有一个方法将是可应用的，并且它将处理该调用。在更复杂的情况下，可能有多个方法均可应用；它们随后将被组合起来——我将在“方法组合”那节里进行讨论——成为一个有效

^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。

(effective) 方法来处理该调用。

你可以用两种方式来特化一个参数——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一个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一个特定对象。

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，调用一个广义参数的结果跟在一个消息传递系统下调用一个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被组合成有效方法，从而在无需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，我将向你展示一些你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`。类层次关系看起来像下面这样：

(图)

第一个广义函数将是 `withdraw`，它将帐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。

`DEFGENERIC` 的基本形式与 `DEFUN` 相似，除了缺少函数体。`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，一个 `DEFGENERIC` 可能含有不同的选项。一个你应当总是带有的选项是 `:documentation`，它提供了一个字符串用来描述该广义函数的用途。由于一个广义函数是纯抽象的，因此让用户和实现者知道它的用途将是重要的。这样，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

一个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所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帐户都是相同的，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 返回当前帐户的余额并且可被用于 SETF——因此也包括 DECF——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) amount))
```

如同这段代码所显示的 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，其要么是一个类的名字要么是一个 EQL 特化符，其形式我将很快讨论到。形参名可以是任何东西——它不需要匹配广义函数中所使用的名字，尽管经常是使用相同的名字。

该方法将在每当 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。

现在假设所有现金帐户都带有透支保护。这就是说，每一个现金帐户都与另一个银行帐户相关联，该帐户将在现金帐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 接受一个 checking-account 对象并返回一个代表了关联帐户的 bank-account 对象。

这样，从一个 checking-account 对象中提款相比从一个标准 bank-account 提款将需要一些额外的步骤。你必须首先检查是否提款量大于该帐户的当前余额，并在出现这种情况时将差额转给透支帐户。然后你可以像一个标准 bank-account 对象那样进行处理。

^① 从技术上来讲你可以完全跳过 DEFGENERIC——如果你用 DEFMETHOD 定义了一个方法而相关的广义函数却没有定义，那么一个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想要的行为。

^② 一个方法“接受”由其广义函数所定义的 &key 和 &rest 参数的方式可以是使用一个 &rest 形参，使用相同的 &key 形参，或是将 &allow-other-keys 与 &key 一起指定。一个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所指定的 &key 参数或任何可应用的方法将被接受。这种一致性规则所带来的一个后果是，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 不支持像诸如 C++ 和 Java 那样的某些静态类型语言里所支持的方法重载 (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。

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` 上的方法来接手。这样一个方法可能看起来像这样：

```
(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。它指示控制应从该方法被传递到特定于 `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`，它并不实际跟踪它自己的余额而是将提款请求代理到其他帐户，那么你可以写一个像下面这样的方法（假设有一个函数 `proxied-account` 的方法可以返回代理的帐户）：

```
(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，假设该银行应用将被部署在一个特定的腐败银行上。假设变量 `*account-of-bank-president*` 保存了一个特定银行帐户的引用，该帐户——如同其名字所显示的——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` 来特别处理他的帐户。

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
    (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*`——只在 `DEFMETHOD` 被求值时求值一次。该方法将特化在方法定义时 `*account-of-bank-president*` 的值上；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一个有效方法的概念是广义函数概念的心脏，并且是允许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地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

^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。

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。随后，这个可应用方法的列表被按照它们的参数特化符中的特化程度（specificity）排序。最后，这些方法根据排序后的列表中的顺序被取出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。一个方法是可应用的当且仅当所有特化符均和对应的参数兼容。

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，因此它们都会影响一个方法是否是可应用的。显式地特化了超过了一个形参的方法被称为多重方法（multimethod）；我将在“多重方法”那一节中讨论它们。

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正在被排序，你可以看出所有的类特化符所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，那么一个将是另一个的子类。在这种情况下，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。

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，一个 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有多个方法带有 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

现在你理解了可应用的方法被找出和排序的方式，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。缺省情况下，广义函数使用一种称为标准方法组合

^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将可以被直接处理。

^② 事实上，特化符被比较的顺序可以通过 DEFGENERIC 的选项 :argument-precedence-order 来定制，尽管该选项很少被用到。

(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 像你所看到的那样工作——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。目前为止我所讨论过的所有方法都称为主方法 (primary method)。主方法如同其名字所显示的，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` 的方法。一个附加方法定义是用 DEFMETHOD 像一个主方法那样写成的但是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` 方法将以下面的定义开始：

```
(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` 方法可以被用来做任何需要确保主方法可以运行的准备工作。例如，你可以使用一个特化在 `checking-account` 上的 `:before` 方法像下面这样来实现对现金帐户的透支保护：

```
(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——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，从而使得一个 `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，由于一个 `:before` 方法不需要调用 CALL-NEXT-METHOD 来将控制传递给其余的方法，因此就不可能因为忘记这点而引入一个 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被运行，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，`:around` 将以非常类似主方法的方式被组合除了它们被运行在所有其他方法的外围。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地运行。在一个 `:around` 的方法的主体中，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` 方法中指向由 `:before` 方法、主方法和 `:after` 方法所组成的复合体。几乎所有的 `:around` 方法都将会含有一个对 CALL-NEXT-METHOD 的调用，因为一个不这样做的 `:around` 的

方法将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其他方法得以运行的一些动态上下文——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。它们并不能实际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。通常一个库将定义一个广义函数并提供一个默认的主方法，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，语言还指定了九种其他的内置方法组合，它们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。我将简要提及如何使用简单内置组合来给你一个对它们功能的感觉。

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 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：产生刚刚所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 方法工作方式相似。

例如，一个使用“+”方法组合的广义函数将返回其所有主方法所返回的结果之和。注意到 AND 和 OR 方法组合由于这些宏的短路行为不一定会运行所有主方法——一个使用 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所返回的非 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 +))
```

缺省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)
```

这可以使你明显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` 的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，这些方法组合不能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有表达性。

这就是说，很可能在 99% 的时间里标准方法组合将是你所需要的东西。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所处理。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相区别的地方。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时被调用。

多重方法 VS. 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。不过这两种语言特性事实上相当不同，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。为了看到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` 来实例化一个 `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，那么上面将代替打印出 “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，因为一个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所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一起来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。

^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。

16.9 未完待续……

我已经谈及广义函数的基础——并且还有一点超出。它是 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。

第17章 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所提到的，一个 Common Lisp 程序中的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 `T` 为根的单一层次体系。

类层次的体系由两个主要的类家族所构成，内置的和用户定义的类。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到目前为止所讨论过的那些函数所管理。你不能创建这些类的子类，但是正如你在前一章里所看到的，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。由于一个类所关联的行为是通过定义广义函数和特化在该类上的方法所决定的，因此 `DEFCLASS` 的责任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（metaclass）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 `STANDARD-CLASS` 的那些类的方式，因此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`

^① 为一个已有的类定义一个新的方法，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，一个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的程序员们将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）。

这样的内置类也是标准的，因为它们是由语言标准所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讨论那些有 DEFCLASS 所定义的使用默认的 STANDARD-CLASS 作为元类的那些类，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分开的名字空间里，因此你可以让一个类、一个函数，和一个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 随后是 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。

暂时省略槽描述符情况下，前一章里你所用到的某些类的 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“多重继承”那一节里讨论在 direct-superclass-name 中列出多于一个直接基类的含义。

17.2 槽描述符

一个 DEFCLASS 形式的大部分是由槽描述符的列表所组成的。每一个槽描述符都定义了一个会成为该类的每个实例的一部分的槽。一个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 SLOT-VALUE 函数来访问。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 一起使用来设置一个对象中某个槽的值。

一个类也从它的所有基类中继承了槽描述符，因此实际存在于任何对象中的槽的集合是一个类的 DEFCLASS 形式中所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，一个命名了槽的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，像下面这样：

```
(defclass bank-account ()
  (customer-name
   balance))
```

该类的每个实例都将含有两个槽，一个用来保存该帐户所属的客户的名字而另一个用来保存当前的余额。借助该定义，你可以用 MAKE-INSTANCE 来创建新的 bank-account 对象。

^① 跟为函数和变量命名时一样，你可以使用任何符号作为一个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。

```
(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，可应用的方法将是由实现所提供的特化在 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，但将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被保存在槽中。另一个选项 :initform 可以让你指定一个 Lisp 表达式在没有 :initarg 参数被传递给 MAKE-INSTANCE 时用来为该槽计算一个值。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

一个包含诸如 :initarg 或 :initform 等选项的槽描述符被写成一个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 的定义来允许 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缺省值，你可以写成这样：

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name)
   (balance
    :initarg :balance
    :initform 0)))
```

^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 所返回的类对象。

^② 另一种影响槽的初始值的方式是通过 DEFCLASS 的 :default-initargs 选项。该选项用来指定将被求值的形式，其用来提供特定初始化形参的参数，当一个特定的 MAKE-INSTANCE 调用没有给定该值时。目前你不需要担心 :default-initargs。

现在你可以创建一个帐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
```

如果你没有提供一个 :balance 参数给 MAKE-INSTANCE，那么 balance 的 SLOT-VALUE 将通过求值由 :initform 选项所指定的形式来计算得到。但如果你没有指定一个 :customer-name 参数，那么 customer-name 槽将是未绑定的，并且在你设置它之前一个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
```

如果你想要确保当帐户被创建时客户名同时被提供，那么你可以在初始化形式中产生一个错误，因为它只在初始化参数没有提供时被求值一次。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 选项的组合就可以很好地初始化一个对象。不过，尽管一个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。

特化在 STANDARD-OBJECT 上的 INITIALIZE-INSTANCE 主方法负责槽的初始化工作，基于它们的 :initarg 和 :initform 选项。由于你不想打扰这些工作，因此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 槽并需要根据该帐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的样子，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
```

^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。

```
: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。^①

```
(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))))
```

形参列表中的 `&key` 是为了保持该方法的形参列表与广义函数一致而要求的——广义函数 `INITIALIZE-INSTANCE` 所指定的形参列表包含了 `&key` 来允许个别方法可以指定它们自己的关键字参数但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，如果一个特化在某个特定类上的 `INITIALIZE-INSTANCE` 方法指定了一个 `&key` 参数，那么该参数在创建该类的实例时就成为了一个 `MAKE-INSTANCE` 的合法参数。例如，银行有时在开户时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)
          (* (slot-value account 'balance) (/ opening-bonus-percentage 100)))))
```

通过定义这个 `INITIALIZE-INSTANCE` 方法，你使得 `:opening-bonus-percentage` 在创建一个 `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。你可能想要做的其他任何事都可以用这两个函数来实现。不过，正如任何熟悉好的面向对象编程实践原

^① 一个在你习惯了使用附加方法之前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。特定的开发环境可能提供图形用户接口来做到同样的事情。

```
(remove-method #'initialize-instance
  (find-method #'initialize-instance () (list (find-class 'bank-account))))
```

则的人所知道的那样，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将你的代码过于紧密地绑定到你的类的具体结构上了。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。直接访问 `balance` 槽的代码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，那么你可以随后重定义它，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可以修改一个槽的方式。^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，因为它帮助你避免了大量相当繁琐的 `SLOT-VALUE` 的使用。

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 `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方法来扩展其定义。因此你可能代替写出下面的定义：

```
(defgeneric balance (account))
(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

正如我已讨论的那样，你不希望调用者可以直接设置余额，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。定义这样的函数最干净的方式是将其定义为一个 `SETF` 函数。

一个 `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。一个 `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。一个 `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。^②例如，你可以定义一个 `SETF` 函数像下面这样设置 `bank-account` 中的 `customer-name` 槽：

```
(defun (setf customer-name) (name account)
```

^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，如果一个类的作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。也有可能使用包系统——我将在第 21 章讨论它——更清楚的说明特定槽不用于直接访问，方法是不导出这些槽的名字。

^② 定义一个 `SETF` 函数——比如说 `(setf foo)`——所带来的后果是，如果你还定义对应的访问函数，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，诸如 `INCF`、`DECREF`、`PUSH` 和 `POP`。

```
(setf (slot-value account 'customer-name) name))
```

在求值该定义之后，一个类似下面这个的表达式：

```
(setf (customer-name my-account) "Sally Sue")
```

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，你将可能希望你的 `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))
```

并且当然你也想为 `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` 提供了三个槽选项来允许你为一个特定槽自动创建读取和写入函数。

`:read` 选项指定了一个将被用作一个接受对象作为其单一参数的广义函数名的名字。当 `DEFCLASS` 被求值时，如果广义函数不存在的话它将被创建。然后一个将其单一参数特化在该新类并返回该槽的值的方法被添加到广义函数中。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的样子：

```
(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。该函数和方法按照一个 `SETF` 函数的要求所创建，接受新值作为其第一个参数并作为结果返回它，因此你可以通过提供一个诸如 `(setf customer-name)` 这样的名字来定义一个 `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))
```

由于同时想要读取和写入函数是很常见的，因此 `DEFCLASS` 还提供了一个选项 `:accessor` 来

同时创建一个读取函数和对应的 SETF 函数。因此代替刚刚给出的槽描述符，你在典型情况下可以写成这样：

```
(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，使用它你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 形式将看起来像下面这样：

```
(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你的代码更易于维护，但使用它们仍然有些繁琐。而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，这时你可能特定地想要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，该方法对一个 bank-account 帐户课以罚款，如果其余额低于一个特定的最小值：

```
(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，它将变得更加混乱。

```
(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 可以帮助减轻这种混乱情况。两个宏都创建了一个代码块，在其中简单的变量名可被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` 元素可以要么是一个槽的名字，它也被用作一个变量名，或是一个两元素列表，其第一个元素是一个用作变量的名字而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。在代码体之内，这些变量名的每一次出现都被翻译成一个对 `SLOT-VALUE` 在该对象和适当的槽名作为参数上的调用。^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
    (when (< bal *minimum-balance*)
      (decf bal (* bal .01)))))
```

如果你已经用一个 `:accessor` 而不只是 `:reader` 来定义 `balance`，那么你还可以使用 `WITH-ACCESSORS`。`WITH-ACCESSORS` 形式和 `WITH-SLOTS` 相同，除了槽列表的每一项都必须是一个两元素列表，其含有一个变量名和一个访问函数的名字。在 `WITH-ACCESSORS` 的主体中，一个对变量的引用将等价为一个对相当的访问函数的调用。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帐户，其中使用两个 `WITH-ACCESSORS` 调用，每个帐户一个。

```
(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))))
```

对于是否使用 `WITH-SLOTS` 还是 `WITH-ACCESSORS` 的选择与 `SLOT-VALUE` 和一个访问函数之间的选择是一样的：提供一个类基本功能的底层代码可以使用 `SLOT-VALUE` 和 `WITH-SLOTS` 来直接修改那些不被访问函数所支持的槽，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` 除非你有特定的理由不这样做。

^① 由 `WITH-SLOTS` 和 `WITH-ACCESSORS` 所提供的“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，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。符号宏被引入到语言中主要是为了支持 `WITH-SLOTS` 和 `WITH-ACCESSORS`，但你也可以将它们用于你自己的目的。我将在第 20 章里讨论它们的更多细节。

17.6 分配在类上的槽

最后一个你需要的槽选项是 `:allocation`。`:allocation` 的值可以是 `:instance` 或 `:class` 并且如果没有指定的话将缺省为 `:instance`。当一个槽带有 `:class` 分配选项时，该槽只有一个单一的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——它们通过 `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实例被创建时求值。另一方面，传递一个 `:initarg` 给 `MAKE-INSTANCE` 将会设置该值，从而影响该类的所有实例。

由于你不能在没有该类的实例的情况下访问一个类分配的槽，因此类分配的槽事实上并不等价于诸如 Java、C++ 和 Python 这些语言里的静态或类字段。^①而且，类分配的槽主要用来节省空间；如果你打算创建一个类的许多实例而所有实例都打算带有一个对同一个对象的引用——比如说一个共享的资源池——那么你可以通过使该槽成为类分配的而省去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所讨论过的，类通过广义函数机制继承了来自其基类的行为——一个特化在类 `A` 上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，但在手法上稍有不同。

在 Common Lisp 中，一个给定对象只能拥有带有给定名字的一个槽。尽管如此，一个给定类的继承层次关系中可能有超过一个类指定了带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带有相同名字的槽描述符，也可能是一个基类指定了带有相同名字的槽。

Common Lisp 解决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为每一个唯一的槽名创建一个单一的描述符。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缺省值，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。这允许一个子类可以指定一个与其本类继承的不同的缺省值。

另一方面，`:initargs` 不需要是互斥的——一个槽描述符中的每一个 `:initarg` 选项都将创建一个可被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。如果一个调用者传递了多个关键字参数来初始化同一个槽，那么 `MAKE-INSTANCE` 调用中最左边的参数会被

^① 元对象协议 (Meta Object Protocol, MOP)，不是语言标准的一部分但被多数 Common Lisp 实现所支持，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。但这并不是正常的用法。

使用。

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 所创建的这些方法已经可以用在新类上。不过，新类可以通过提供它自己的 :reader、:writer 或 :accessor 选项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所决定。这样，有可能一个类的所有实例共享了一个 :class 槽，而它的每一个基类的每个实例可能带有相同名字的它们自己的 :instance 槽。而一个子类的子类可能随后将其重定义回 :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所共享的槽和最初的基类所共享的槽是不同的。

例如，假设你有下面的类：

```
(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 指定一个值，并且事实上必须这样做才能避免出错，因为由 bar 所提供的 :initform 覆盖了继承自 foo 的那一个。为了初始化 b 槽，你可以要么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 槽带有的 :allocation 选项，指定的值将被保存在由 bar 的所有实例所共享的槽中。同样的槽既可以使用特化在 foo 之上的广义函数 b 的方法来访问，也可以使用直接特化在 bar 上的广义函数 the-b 的新方法来访问。为了访问无论 foo 还是 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一个单一的槽。这样，特化在不同类上的方法当应用在一个扩展了这些类的类上时可能最终在操作同一个槽。这在实践中并不是太大的问题，因为你可以使用包系统——你将在第 21 章学到它——来避免互不相关的代码中的名字冲突。

17.8 多重继承

目前为止你所看到的所有类都只有一个单一的直接基类。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。

多重继承并没有本质上改变任何目前为止我所谈及的继承机制——每一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，因此至少有两个基类。多重继承所带来的问题是一个类可以有超过一个直接 (direct) 基类。这使得类的特化性概念当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变得更加复杂了。

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是极其简单的——一个类和它的所有基类将被排序成从该类开始的一条直线，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。但是当一个类带有多个直接基类时，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，那么你不需要同时成为它们的直接子类。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，注意到并不存在所有类的全序——每一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。

为了看到它是如何工作的，让我们向银行应用中添加一个类 `money-market-account`。一个货币市场帐户组合了来自支票帐户和储蓄帐户的特征：一个客户可以用它写支票，但也可以挣得利息。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)
```

注意到该列表是怎样同时满足两条规则的：每一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。

该类没有定义它自己的槽但是会继承来自它的两个直接基类的槽，包括它们继承自它们基类的槽。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一个 `money-market-account` 对象上。由于同一个槽的所有槽描述符被合并了，因此 `money-market-account` 从 `bank-account` 中继承了相同的槽描述符两次而不会有什问题。^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得到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个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` 函数用来生成月对帐单。假设已经有了特化在 `checking-account` 和 `savings-account` 上的 `print-statement` 方法。这两个方法对于 `money-market-account` 实例来说都将是可应用的，但特化在 `checking-account` 上的方法将被认为比 `savings-account` 上的方法更加相关，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

^① 换句话说，Common Lisp 不会遇到像比如说 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一个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。

在 `money-market-account` 上调用 `print-statement`, 特化在 `checking-account` 上的方法将被使用。不过, 这并不一定可以给你想要的行为, 因为你可能想要一个货币市场帐户的对帐单中同时含有来自支票帐户和储蓄帐户对帐单的元素。

你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特化在 `money-market-account` 上的新的主方法。这可以给你对新行为最大的控制但可能需要比我很快将讨论到的其他选项要求更多的新代码。问题在于当你可以用 `CALL-NEXT-METHOD` 来“向上”调用下一个最相关方法时, 也就是特化在 `checking-account` 上的那个方法, 没有办法来调用一个特定的不太相关的方法, 例如特化在 `savings-account` 上的方法。因此, 如果你想要重用那些打印对帐单中 `savings-account` 部分的代码, 你将需要将那些代码打断成单独的函数, 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`。然后特化在 `money-market-account` 上的方法将使用 `CALL-NEXT-METHOD` 来调用特化在 `checking-account` 上的方法。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。

当然, 如果你打算依赖一种编码约定——所有方法都调用 `CALL-NEXT-METHOD`——来确保所有可应用方法都能在某一点处运行, 那么你应该考虑使用附加方法来代替。在这种情况下, 除了为 `checking-account` 和 `savings-account` 定义 `print-statement` 之上的主方法, 你还可以将这些方法定义成 `:after` 方法, 同时在 `bank-account` 上定义一个单一的主方法。然后, 调用在一个 `money-market-account` 上的 `print-statement` 将首先打印出一个由特化在 `bank-account` 上的主方法所输出的基本帐户对帐单, 然后是由特化在 `savings-account` 和 `checking-account` 上的 `:after` 方法所输出的细节。并且如果你想要添加特定于 `money-market-account` 的细节, 那么你可以定义一个特化在 `money-market-account` 上的 `:after` 方法, 它将最后被运行。

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, 而哪个方法只贡献附加的一点儿功能变得非常清楚了。缺点在于你对于这些附加方法的运行顺序无法得到良好的控制——如果你想要对帐单中的 `checking-account` 部分在 `savings-account` 部分之前打印, 那么你就必须改变 `money-market-account`, 子类化这些类的顺序。但那是一个相当大的改变, 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, 那么你可能需要回过头来重新考虑你的设计。

另一方面, 如果你并不关心具体的顺序而只是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` 之外你还有一個 `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, 这样它们中的正规和细化的对帐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。如果你拥有大量的面向对象编程经验, 那么你可能会看到 Common Lisp 的特性是怎样用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，那么你可能需要花费一些时间来吸收面向对象的思想。不幸的是，这是一个相当大的主题并且超出了本书的范围。或者，正如 Perl 的对象系统手册页中所写，“现在你只需离开这里然后买一本面向对象设计方法学的书并在接下来六个月里埋头苦读就好了”。或者你可以等待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，目前你需要和所有这些面向对象理论告一段落，并转而去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。

第18章 一些 FORMAT 秘决

Common Lisp 的 `FORMAT` 函数——以及扩展的 `LOOP` 宏——是 Common Lisp 的两个在许多用户中引起强烈反响的特性之一。有些人喜欢它；而另一些人讨厌它。^①

`FORMAT` 的喜爱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，但至少一个单一的 `FORMAT` 表达式还不会太坏。例如，假设你想要将一个列表中的值以逗号分隔打印出来，你可以写成下面这样：

```
(loop for cons on list
      do (format t "~a" (car cons))
      when (cdr cons) do (format t ","))
```

这不算太坏，但任何读到这些代码的人将不得不在头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，那么你将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。至少理解 `FORMAT` 的基础也是重要的，因为其他标准函数，诸如下一章里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。

更进一步的说，`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，以及使用插入的值来生成人类可读的消息。将表格中的数据作为文本打印现在已经有些过时了；它是 Lisp 几乎和 FORTRAN 一样老的象征之一。事实上，一些你可以用来将浮点值在定长字段中打印的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，因此我将不会讨论 `FORMAT` 的这些方面。

^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。

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 精美打印器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到。^①

因此，我将聚焦在 `FORMAT` 中你可以使用插入的值来生成人类可读字符串的那部分。即便以这种方式限定范围，仍然有大量内容将被谈及。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` 函数接受两个必要参数：一个用于输出的目的地，另一个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。我将把这些参数称为格式化参数（format argument）。

`FORMAT` 的第一个参数，输出的目的地，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，那么输出将被写到该流中。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为相当调整。除了当目的地是 `NIL` 时 `FORMAT` 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。

多数的 `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消耗任何参数。而其他的指令，如同你将要看到的，可以消耗超过一个参数。有个指令甚至允许你在参数列表中跳动从而处理同一个参数超过一次，或是在特定情况下跳过特定参数。但在我讨论特定指令之前，让我们首先查看指令的一般语法。

18.2 FORMAT 指令

所有的指令都以一个波浪线（~）开始并终止在单个字符上。你可以使用无论大写或小写来书写那个字符。某些指令带有前置参数（prefix parameters），它们紧跟在波浪线后面书写，由逗号分隔，并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，一

^① 对于精美打印器感兴趣的读者可能会想要阅读 Richard Waters 的论文《XP: A Common Lisp Pretty Printing System》。它是一个对后来合并到 Common Lisp 中的精美打印器的描述。你可以从 <ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-1102a.pdf> 下载它。

^② 稍微带来混淆的一点是，许多其他的 I/O 函数也接受 `T` 和 `NIL` 作为流标识符，但却带有不同的语义：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*`。

一个用来打印浮点值的指令，缺省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，通过使用一个前置参数，你可以指定它打印出比如说五个小数，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，后者的书写形式是一个单引号后接想要的字符。一个前置参数的值还可以通过两种方式从格式化参数中获得：一个前置参数 `v` 导致 `FORMAT` 消耗一个格式化参数并将其值用作前置参数。而一个前置参数 “`#`” 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“条件格式化”一节中给出一些关于如何使用“`#`”参数的更现实的例子。

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，另一个用来打印浮点值的指令，它也接受一个参数来控制需要打印的十进制位的数量，但这是它的第二个参数而不是第一个。如果你想要使用 `~F` 来打印一个数字到五个十进制位上，那么你可以写成这样：

```
CL-USER> (format t "~,5f" pi)
3.14159
NIL
```

你还可以使用冒号和“`@`”修改符来调整某些指令的行为，它被放在任何前置参数之后和指令的标识字符之前。这些修改符可以细微地改变指令的行为。例如，使用一个冒号修改符，用来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“`@`”修改符可以使 `~D` 在数字为正时带有一个加号。

```
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，你可以组合使用冒号和“`@`”修改符来同时得到两种调整。

```
CL-USER> (format t "~:@d" 1000000)
+1,000,000
NIL
```

在那些两个修改行为不能有意义的组合在一起的指令中，同时使用两个修改符将要么是未定

义的，要么给出第三个含义。

18.3 基本格式化

现在你开始查看特定指令。我将从几个最常用的指令开始，包括一些你在前面章节里已经看到的。

最通用的指令是 `~A`，其消耗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，字符串被输出成没有引用标记或转义字符的形式，而数字被输出成对于该数字的类型来说自然的方式。如果你只是想要产生一个给人看的值，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`，它同样消耗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，而符号将在必要的时候是包限定的，以及诸如此类。那些不带有可读表示的对象将被打印成不可读对象语法 `#<>`。使用一个冒号修改符，`~A` 和 `~S` 指令都可以将 `NIL` 输出成 `()` 而不是 `NIL`。`~A` 和 `~B` 指令也都接受最多四个前置参数，它们可被用于控制是否在值的后面（或者当使用“`@`”修改符时在值的前面）添加占位符，但这些参数只有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一个换行的 `~%`，以及用来产生一个新行（fresh line）的 `~&`。两者的区别在于 `~%` 总是产生一个换行，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在一起的函数特别有用，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的。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，那么当你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一个单一前置参数来指定想要产生的换行的个数。`~%` 指令将简单地输出那些换行符，而 `~&` 指令将输出 `n-1` 或 `n` 个换行，具体取决于是否它在一行的开始处开始输出。

不太常用的是相关的 `~~` 指令，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

在通用指令 `~A` 和 `~S` 以外，`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一个是 `~C` 指令，它被用来输出字符。它不接受前置参数但可被冒号和“`@`”修改符所修改。在不修改的情况下，它的行为和 `~A` 没有区别除了它只能工作在字符上。修改后的版本更加有用。使用一个冒号修改符，`~:C` 可以将诸如空格、制表符和换行符这些不可打印字符输出成它们的名字。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)
```

可以产生下面这样的信息：

```
Syntax error. Unexpected character: a
```

但也可以像下面这样：

```
Syntax error. Unexpected character: Space
```

使用“@”修改符，~@c将把字符输出成 Lisp 的字面字符语法。

```
CL-USER> (format t "~@c~%" #\a)
#\a
NIL
```

同时使用冒号和“@”修改符，~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被打印出来。例如，在 Macintosh 上，在特定应用中你可以通过按下 Control 键然后输入“@”来键入一个空字符(在 ASCII 或诸如 ISO-8859-1 和 Unicode 等 ASCII 超集中字符编码为 0 的字符)。在 OpenMCL 中，如果你使用 ~:@c指令来打印空字符，那么它将告诉你下面这样的信息：

```
(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。并且就算它们实现了，结果也可能不是精确的——例如，如果你正在 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，但如果你想要更好地控制它们被打印的形式，那么你就需要使用特定于字符的指令了。这些数值指令可以被分成两个子类别：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 10 进制输出整数。

```
(format nil "~d" 1000000) → "1000000"
```

如同我前面提到的，使用一个冒号修改符会在输出中添加逗号。

```
(format nil "~:d" 1000000) → "1,000,000"
```

而使用一个“@”修改符，它总是打印一个正负符号。

```
(format nil "~@d" 1000000) → "+1000000"
```

并且两个修改符可以组合使用。

```
(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。缺省的占位符是空格，而占位符总是被插入在数字本身之前。

```
(format nil "~12d" 1000000) → " 1000000"
(format nil "~12,'0d" 1000000) → "000001000000"
```

^① ~c 指令的这个变种在像 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。

这些参数在格式化诸如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改符配合使用的：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。这些参数缺省为一个逗号和数字 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 `~,~,4:d` 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"
```

注意到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缺省值。

`~x`、`~o`、`~b` 指令与 `~d` 指令的工作方式相同，除了它们将数字分别输出成十六进制、八进制和二进制。

```
(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“@”修改符也以相同的方式修改其行为。`~r` 指令当不使用任何前置参数时还有一些特殊行为。我将在“英语指令”那节里讨论它。

18.5 浮点指令

有四个格式化浮点值的指令：`~F`、`~E`、`~G` 和 `~$`。这些指令中的前三个是基于 FORTRAN 的编辑描述符的指令。我将跳过这些指令的多数细节因为它们多数被用于以表格形式格式化浮点值。尽管如此，你可以使用 `~F`、`~E` 和 `~$` 指令将浮点值插入到文本中。而另一方面，通用浮点指令 `~G` 组合了 `~F` 和 `~E` 指令的特性，从而使得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——并可能控制十进制小数点之后的数位数量。不过，`~F` 指令被允许在数字特别大或特别小时使用科学计数法。另一方面，`~E` 指令在输出数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数位个数。

```
(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"
```

`~$` 指令和 `~F` 相似但更简单一些。如同其名字所显示的，它被用于输出货币单元。不带有参数

^① 技术上来讲，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的话，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。

时，它基本上等价于 `~,2F`。为了修改十进制小数点之后打印的数位数量，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数位数量的最小值。

```
(format nil "~$" pi)      → "3.14"
(format nil "~2,4$" pi) → "0003.14"
```

所有三个指令 `~F`、`~E` 和 `~$` 都可以通过使用 “@” 修改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。这些指令允许你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“字符和整数指令”一节中所讨论过的 `~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改符使用时，它将数字输出成作为基数的词。

```
(format nil "~r" 1234) → "one thousand two hundred thirty-four"
```

使用冒号修改符，它将数字输出成序数。

```
(format nil "~:r" 1234) → "one thousand two hundred thirty-fourth"
```

而当使用 “@” 修改符时，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，其中四和九被写成 `IIII` 和 `VIII` 而不是 `IV` 和 `IX`。

```
(format nil "~@r" 1234) → "MCCXXXIV"
(format nil "~:@r" 1234) → "MCCXXXIVI"
```

对于那些以给定形式无法表示的过大的数字，`~R` 将回退到与 `~D` 相同的行为。

为了帮助你生成带有正确复数化单词的消息，`FORMAT` 提供了 `~P` 指令，它简单地输出一个 `s` 除非对应的参数是 1。

```
(format nil "file~p" 1) → "file"
(format nil "file~p" 10) → "files"
(format nil "file~p" 0) → "files"
```

不过，典型情况下你将使用带有冒号修改符的 `~P`，这导致它重新处理前一个格式化参数。

```
(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 “@” 修改符，它也可以与冒号修改符组合使用，`~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"
```

^① 但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。

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，但它对于那些它可以处理的情况是很有用的。而我将很快讨论到的“`~[`”指令可以给你更灵活的方式来条件化输出 `FORMAT`中的某些部分。

最后一个用来输出英语文本的指令是“`~(`”，它可以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，并且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"
```

你可以使用“`@`”符号来修改“`~(`”的行为使其将一段文本中的第一个词变成首字母大写的形式，使用一个冒号可以让它将所有单词首字母大写，而两个修改符同时存在将使全部文本转化成大写形式。(该指令意义上的一个单词，指的是一个以非字母表字符或文本结尾所分隔的字母表字符的序列。)

```
(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“`~[`”。该指令被一个对应的“`~]`”所闭合，并且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，其随后被 `FORMAT`所处理。在没有修改符或参数的情况下，该子句由数值索引所选择；“`~[`”指令消耗一个格式化参数，其应当是一个数字，并且取出第 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，那么将不打印任何东西。

```
(format nil "~[cero~;uno~;dos~]" 3) → ""
```

不过，如果最后一个子句分隔符是“`~:;`”而不是“`~;`”的话，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"
```

也有可能通过使用一个前置参数来指定被选择的子句。尽管使用一个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."
```

注意到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。第一个指令消耗零到两个参数，而第二个指令在需要的时候会再消耗一个参数。`FORMAT`将默默地忽略任何在处理控制字符串中没有被消耗的参数。

如果使用一个冒号修改符，那么“~[”将只含有两个子句；该指令消耗一个单一参数并在该参数为 `NIL` 时处理第一个子句而在其他情况下处理第二个子句。你曾经在第 9 章里使用该“~[”变种来生成“通过或失败”消息，像下面这样：

```
(format t "~:[FAIL~;pass~]" test-result)
```

注意到任何一个子句都可以是空的，但指令中必须含有一个“~;”作为分隔。

最后，使用一个“@”修改符，“~[”指令可以只带有一个子句。该指令消耗一个参数并且当它是非空时可以使该参数回过头来被再次消耗然后再处理其子句。

```
(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

另一个过去你已经见过的 `FORMAT` 指令是迭代指令“~{”。该指令可以让 `FORMAT` 迭代在一个列表的元素上或是一个隐式的 `FORMAT` 参数列表上。

不带有修改符时，“~{”消耗一个格式化参数，其必须是一个列表。和“~[”指令一样，其必须与一个“~]”指令配对，“~{”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“~{”指令所消耗的列表中取得其参数。`FORMAT` 将重复处理该控制字符串，只要被迭代的列表尚有元素剩余。在下面的示例中，“~{”消耗单一格式化参数，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被消耗。

```
(format nil "~{~a, ~}" (list 1 2 3)) → "1, 2, 3, "
```

尽管如此，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显得很讨厌。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，“~^”将导致迭代立即停止而无须处理其余的控制字符串。这样，为了避免一个列表的最后元素之后打印逗号和空格，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，当“`~^`”被处理时列表中尚有未处理的元素。不过，在第三次的时候，在`~a`指令消耗掉3之后，“`~^`”将导致`FORMAT`跳出迭代而不会打印出逗号和空格。

使用一个“`@`”修改符，“`~{`”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。你可以将它和“`~[`”指令一起使用来打印一个逗号分隔的列表并带有一个“`and`”在最后一个元素之前，就像下面这样：

```
(format nil "~{~a~#[~, and ~:~, ~]~}" (list 1 2 3)) → "1, 2, and 3"
```

不过，当列表是两元素长度时上述表达式并没有真的工作正常，因为它添加了一个额外的逗号。

```
(format nil "~{~a~#[~, and ~:~, ~]~}" (list 1 2)) → "1, and 2"
```

你可以有多种方式来修复这个问题。下面的方法利用了“`~@{`”在嵌入到另一个“`~{`”或“`~@{`”指令之中时所具有的一种行为——它迭代在外层“`~{`”所迭代的那个列表的剩余元素上。你可以将它与一个“`~#[`”指令组合使用使得下面的控制字符串可以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，但如果你花一点时间的话还不难理解它。外层的`~{...~}`将消耗并迭代在一个列表上。整个迭代体随后由一个`~#[...~]`所构成；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，其中最后一个是一个缺省子句因为它前置了一个“`~:;`”而不是一个普通的“`~;`”。第一个子句，用于当还有零个元素需要被处理时，是空的，这是合理的——如果没有元素需要被处理了，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。而缺省子句，其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“`~@{`”来迭代在外层“`~{`”所处理的列表的其余元素上。而该迭代的主体是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，那么你有几种方式来做到这点。可能最简单的一种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“`~}`”上添加一个冒号修改符——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改符组合提供了更多的变种。我不会详细讨论它们，但简单地说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改符，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被用作控制字符串中~:{...~}指令的参数。

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，没有修改符的情况下，它简单地跳过下一个参数，消耗它而不输出任何东西。不过更常见的情况是，它和一个冒号修改符一起使用，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“~:*”将一个数值参数以单词的形式打印一次再以数值的形式打印一次：

```
(format nil "~r ~:*(~d)" 1) → "one (1)"
```

或者你也可以通过组合“~:*”与“~[”来实现一个类似于~:_的非常规复数形式的指令。

```
(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还可以被用作“~[”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，你可以像这样只打印一个plist中的键：

```
(format nil "~{~s~*~^ ~}" '(:a 10 :b 20)) → ":A :B"
```

“~*”指令还可以被给定一个前置参数。当没有修改符或使用冒号修改符时，该参数指定了向前或向后移动的参数个数并且缺省为1。当使用“@”修改符时，该前置参数指定了一个用来跳跃的绝对的以零开始的参数索引，缺省为0。“~*”的“@”变种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，并且如果不同的消息需要使用不同顺序的这些参数时，可能是有用的。^②

^① 如果你觉得“*I saw zero elves*”这种说法有点奇怪，那么你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."
```

^② 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。

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，或者“~/”指令，它允许你调用一个任意函数来处理下一个格式化参数。而且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——其他语言的异常和错误处理系统在 Common Lisp 中的相似物。

第19章 超越异常处理：状况和再启动

Lisp 的状况 (condition) 系统是它最伟大的特性之一。它与 Java、Python 和 C++ 中异常处理系统有着相同的目标但却更加灵活。事实上，它的灵活性扩展到了错误处理之外——“状况”相比“异常”的更一般之处在于一个状况可以代表一个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“状况的其他使用”那一节里你将看到状况可被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警告信息被打印。不过，从一开始我将集中在错误处理上。

状况系统比异常系统更灵活之外在于它没有提供在产生错误¹的代码和处理错误²的代码之间这种两部分的划分，相反状况系统将责任分拆成三个部分——产生 (signaling) 一个状况，处理 (handling) 它，以及再启动 (restarting)。在本章里，我将描述你怎样在一个分析日志文件的假想应用程序的一部分中使用状况系统。你将看到自己如何使用状况系统来允许一个底层函数在解析日志文件时检测一个问题并产生一个错误，并允许中层代码提供几种可能方式来从这样一个错误中恢复，再允许该应用程序的最上层代码定义一种方针来选择所使用的恢复策略。

为了更好地开始，我将介绍一些术语：错误 (error)，在我使用该术语时，将采用墨菲法则的定义。认为会出错的，终将出错：一个你的程序需要读取的文件将可能丢失，一个你需要写入的磁盘将会是满的，你正在连接的服务器将会崩溃，或者网络将会断开。如果任何这些情况发生，它可能使一些正在做你想要的事情的代码停止工作。但这里没有 bug；代码中没有位置可以修复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，否则你将引入一个 bug。因此，错误并不是由 bug 所导致的，但忽视处理一个错误将总是一个 bug。

那么，处理一个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，然后诸如此类。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`，后者调用了 `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。它的直接调用者——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`

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。

因为它需要 `low` 做的工作能够完成；如果该工作不能完成，那么 `medium` 就有问题了。但这意味着 `medium` 的调用者 `high` 也有问题了——诸如此类直到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过问题——所有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中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或提升一个异常来返回控制。使用异常相比使用正常函数返回来说是一个巨大的进步，但两种模式有一个共同缺点：在搜索一个可被用来恢复的函数时，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。

考虑函数 `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东西使对 `medium` 的调用得以正常工作，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——一个完整的对于 `medium` 的工作的额外实现将需要提供。并且当栈进一步展开时，还有更多工作需要被重做。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` 的内部工作原理有不必要的了解，这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 的错误处理系统给了你一种跳出这一难题的方式，它让你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低层次函数中而无需决定实际使用何种特定的恢复策略，将决策留给高层次函数中的代码。

为了了解其工作原理，让我们假设你正在编写一个读取某种形式的文本日志文件的应用程序，例如读取一个 Web 服务器的日志。在你应用的某个位置上你将有一个函数用来解析单独的日志项。让我们假设你将编写一个函数 `parse-log-entry`，它将被传递一个含有单一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。该函数将从一个 `parse-log-file` 中被调用，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` 函数将不需要解析不正确格式化的项。不过，它可以检测到有问题的输入。但当它检测到坏的输入以后应当做什么呢？在 C 中你会返回一个特殊值来指示这里出了问题。在 Java 或 Python 中你会抛出或提升一个异常。在 Common Lisp 中，你产生一个状况。

19.2 状况

一个状况（condition）是一个对象，其所属的类代表了该状况的一般性质，而其实例数据

则带有导致该状况被产生的关于特定情形细节的信息。³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。

状态类使用 `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, 并且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; 你必须为任何其值打算被使用的槽分别指定 `:reader` 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` 的形式更进一步地定制一个状况的初始化过程。⁴

当把状况系统用于错误处理时, 你应当将你的状况定义成 `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` 中当你无法解析日志项时你将产生一个 `malformed-log-entry-error`。你使用函数 `ERROR` 来报错, 后者将调用底层函数 `SIGNAL` 并在状况没有被处理时进入调试器。有两种方式来调用 `ERROR`: 你可以传给它一个已经实例化了的状况对象, 或者你可以传给它该状况类的名字以及需要用来构造一个新状况的任何初始化参数, 然后它为你实例化该状况。前者对于重新产生一个已有的状况对象偶尔是有用的, 但后者更普遍。这样, 你可以像下面这样编写 `parse-log-entry`, 其中隐藏了实际解析一个日志项的细节:

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (error 'malformed-log-entry-error :text text)))
```

当错误被产生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, 你必须在导致对 `parse-log-entry` 的调用的某个函数中建立一个状况处理器 (`condition handler`)。当一个状况被产生时, 信号机制会查看一个活跃状况处理器的列表, 并基于状况的类来寻找可以处理被产生状况的处理器。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一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被建立在调用栈的不同层次上。当一个状况被产生时, 信号机制会查找最近建立的其

³ 在这个意义上, 一个状况和 Java 或 Python 中的异常很像, 除了并非所有状况都代表一个错误或异常的 (exceptional) 情形。

⁴ 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。

类型说明符与当前被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地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，后者随后将继续搜索下一个带有兼容的类型说明符的最近建立的处理器。为了处理该状况，该函数必须通过一个非本地退出（`nonlocal exit`）来将控制传递到 `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。一个 `HANDLER-CASE` 的主体必须是一个单一表达式；你可以使用 `PROGN` 将几个表达式组合成一个单一形式。不过，如果该表达式产生了一个状况并且其实例属于任何 `error-clause` 中所指定的 `condition-type` 之一，那么相应的错误子句中的代码将被执行并且其值将被 `HANDLER-CASE` 返回。形参 `var` 当被包含时，将成为处理器代码被执行时用来保存状况对象的变量名。如果代码不需要访问状况对象，你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` 表达式将要么返回由 `parse-log-entry` 所返回的值，要么在 `malformed-log-entry-error` 被产生时返回 `NIL`。（`LOOP` 子句 `collect it` 中的 `it` 是另一个 `LOOP` 关键字，它指向最近求值的条件测试的值，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，它的值将被赋值到变量 `entry` 中并被 `LOOP` 所收集。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` 是与 Java 或 Python 风格的异常处理最接近的 Common Lisp 相似物。你在 Java 中可能写成这样：

```
try {
  doStuff();
  doMoreStuff();
} catch (SomeException se) {
  recover(se);
}
```

或是 Python 写成这样：

```
try:
    doStuff()
    doMoreStuff()
except SomeException, se:
    recover(se)
```

而在 Common Lisp 中你需要写成这样：

```
(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：它做的太多了。如同其名字所显示的那样，`parse-log-file` 的任务是分析文件并产生一个 `log-entry` 对象的列表；如果它做不到这一点，决定替代方案就不是它的职责所在。假如你想要在一个应用中使用 `parse-log-file` 而该应用想要告诉用户日志文件被破坏了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？或者可能一个应用对于跳过它们是没问题的但只有当特定数量的有问题日志项被发现以后才需要特别处理。

你可以试图通过将 `HANDLER-CASE` 移动到一个更高层的函数中来修复该问题。不过，这样你将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来提供当前的恢复策略而不要求它总是被用到。

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。你将那些实际从错误中恢复的代码放在再启动（`restart`）中，而状况处理器可以随后通过调用一个适当的再启动来处理一个状况。你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` 使其建立一个再启动而非状况处理器，你可以将 `HANDLER-CASE` 改成 `RESTART-CASE`。`RESTART-CASE` 的形式与 `HANDLER-CASE` 很相似，除了再启动的名字可以只是一些普通的名字而无需是状况类型的名字。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`。该函数的新版本将如下所示：

```
(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)))
```

如果你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; 你最终将进入到调试器中。不过, 在调试器所提供的几个再启动选项中将会有一个称为 `skip-log-entry` 的选项, 如果你选择了它, 将会导致 `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使其可以自动地调用 `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` 可以用在更多的情形里了。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, 如同我将在下一节所展示的那样, 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。

但在我开始谈论这些之前, 你需要看到如何设置一个将会调用 `skip-log-entry` 再启动的状况处理器。你可以在通向 `parse-log-entry` 的函数调用链的任何位置上设置这个处理器。这可能是你应用程序中很上层的某个位置, 不必在 `parse-log-file` 的直接调用者中。例如, 假设你的应用程序的主入口点是一个函数, `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, 它可能看起来像这样:

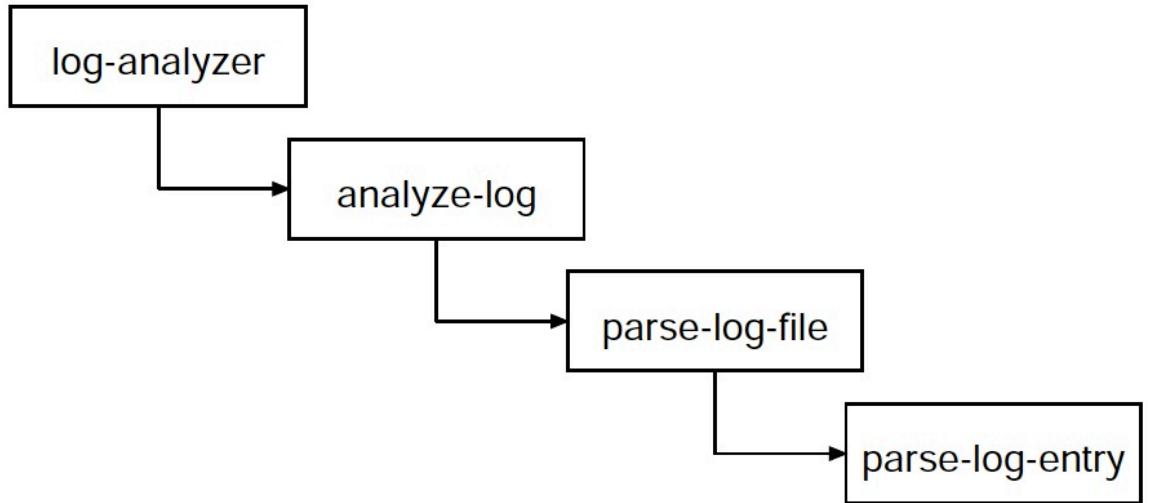
```
(defun log-analyzer ()
  (dolist (log (find-all-logs))
    (analyze-log log)))
```

`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看起来像这样:

```
(defun analyze-log (log)
  (dolist (entry (parse-log-file log))
    (analyze-entry entry)))
```

其中函数 `analyze-entry` 据推测将会实际解出那些你想要从每个日志项中得到的信息并储存在其他地方。

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，那么你可以改变这个函数来建立一个状况处理器去为你调用 `skip-log-entry`。不过，你不能使用 `HANDLER-CASE` 来建立这个状况处理器，因为那样的话栈将会回退到 `HANDLER-CASE` 所在的函数里。代替地，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一个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——当该函数被调用时，控制流将仍然在对 `parse-log-entry` 的调用中。对 `INVOKE-RESTART` 的调用将查找并调用带有给定名字的最后绑定的那个再启动。因此你可以像下面这样添加一个处理器到 `log-analyzer` 中使其可以调用由 `parse-log-file` 所建立的再启动：⁵

```
(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到相同的事并绑定该函数来代替。事实上，在定义再启动时有个常用的实践技巧是同时定义一个函数，它与再启动具有相同的名字并接受一个单一状况参数并调用对应的再启动。这样的函数被称为再启动函数（restart functions）。你可以像下面这样为 `skip-log-entry` 也定义一个再启动函数：

```
(defun skip-log-entry (c)
  (invoke-restart 'skip-log-entry))
```

⁵ 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 形式体中的第一个表达式来消除这类警告。

然后你可以将 `log-analyzer` 的定义修改成下面这样：

```
(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` 再启动函数假设一个 `skip-log-entry` 再启动已经被建立了。如果一个 `malformed-log-entry-error` 被来自 `log-analyzer` 的代码所抛出但却没有建立一个 `skip-log-entry` 再启动，那么对 `INVOKE-RESTART` 的调用将在无法找到 `skip-log-entry` 再启动时产生一个 `CONTROL-ERROR` 报错。如果你想要允许一个 `malformed-log-entry-error` 可以被没有建立 `skip-log-entry` 再启动的代码所报出，那么你可以将 `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` 可以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` 所绑定时，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一个机会来处理该状况。

19.5 提供多个再启动

由于再启动必须被显式地调用才有效果，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。如果我早先提到的，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那些 `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一个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。你可以同时对我刚刚提到的两种恢复策略提供支持，通过为 `parse-log-entry` 增加两个再启动，其中每个都只接受单一参数。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)))))
```

名字 `USE-VALUE` 是这类再启动的一个标准名字。Common Lisp 定义了一个类似于你之前为 `skip-log-entry` 所定义的再启动函数的函数 `USE-VALUE`。因此，如果你想要改变对有问题项的处理策略，在遇到这类问题项时创建一个 `malformed-log-entry` 的实例，那么你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，你通常会想要把再启动放在尽可能低层的代码里。尽管如此，将 `skip-log-entry` 移到 `parse-log-entry` 中并不合适，因为这将导致 `parse-log-entry` 有时会正常返回一个 `NIL`，而这是你需要极力予以避免的。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需要理解 `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 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，但它们还可以被用于其他目的——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` 实现了搜索可应用的状况处理器并调用其处理器函数的机制。一个处理器可以通过正常返回来拒绝处理一个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`，后者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` 在状况被处理之前找不到其他的状况处理器了，那么它也会正常返回。

你曾经使用的 `ERROR` 函数就会调用 `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` 的调用从不正常返回；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` 提供了一个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` 并不会进入调试器——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可以令 `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，并在不存在这样一个再启动时产生一个 `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将一个警告“提升”为一个错误从而像真正的错误一样来处理它。

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可能是稍微有些不正常但仍然是可以解析的，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在做这件事的时候同时用 `WARN` 产生一个状况。然后更大的应用可以选择让这些警告被打印出来、

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` 又提供了另一种协议。和 `ERROR`一样，`CERROR` 将在状况没被处理时把你带进调试器。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，它将确保你始终不会进入调试器。否则，你可以在进入调试器以后使用再启动来立即恢复到 `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` 之上构建你自己的协议——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地被使用。但对于多数目标来说，标准的错误和警告协议应该就足够了。

你将在未来的实践性章节里用到状况系统，无论是用于正常的错误处理，还是像第 25 章的那种帮助处理 ID3 文件解析过程中一些棘手的边界情况。不幸的是，错误处理在编程教材里总是被过于轻视了——正确的错误处理，或者这方面的欠缺，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

对于编写健壮的软件方面的信息，你可以从查找一本 Glenford J. Meyers 编写的叫做《Software Reliability》(John Wiley & Sons, 1976) 的书开始。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。例如，参见他的《Object-Oriented Software Construction》(Prentice Hall, 1997) 一书的第 11 和 12 章。不过要记住，Bertrand Meyer 是 Eiffel 的发明者，后者是一个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的问题的一个很好的综述，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。

在下一章里我将对你尚未有机会使用或者说是至少还没有直接用到的 25 个特殊操作符做一个简短的概述。

第20章 特殊操作符

从某种意义上来说，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。这种可能性是因为 Common Lisp 的特殊操作符——尽管没有哪个是直接用于产生或处理状况的——提供了通向语言底层机制的足够的权限，从而允许做到诸如控制栈的回退这样的事情。

在前面的章节里我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那些。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。有必要熟悉这些特殊操作符，从而当有一天需要时，你至少可以知道它们的存在。其次，因为 25 个特殊操作符——和求值函数调用的基本规则以及内置数据类型一起——提供了语言其余部分的基础，因此对它们有一个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你可以直接用在你自己的代码中的，哪些是用来作为你一直在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

特殊操作符的第一类包括三个提供了对形式求值的基本控制。它们是 `QUOTE`、`IF` 和 `PROGN`，而我们已经全部讨论过它们了。尽管如此，值得注意这些操作符中是如何分别提供了基本的对一个或多个形式求值的控制类型的。`QUOTE` 完全避免求值并且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。¹ 而 `PROGN` 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作用域 (lexical environment) 的操作符所组成。前面已经讨论过的 `LET` 和 `LET*` 就是用于维护词法环境的特殊操作符的例子，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

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。

一个 `LET` 或 `LET*`²。`SETQ` 特殊操作符是用来访问词法环境的特殊操作符之一，因为它可以被用来设置那些由 `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 `LABEL` 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。这些特殊操作符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被使用多次的局部函数时将会非常有用。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被使用，包括 `LABELS` 所定义的函数本身在内。这样，`LABELS` 可以用来定义递归函数，而 `FLET` 就不行。虽然 `FLET` 不能用来定义递归函数看起来是一种限制，但 Common Lisp 同时提供了 `FLET` 和 `LABELS` 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` 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。由于你可以使用 `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的变量，因为它们通常可以被书写成比等价的辅助函数带有更少的参数的形式。这对于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`：

```
(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)))
      versions)))
```

² 从技术上来讲这些构造也可以展开成一个 `LAMBDA` 表达式，因为——正如我在第 6 章里所提到的——`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。

这个函数也可以被写成在 `FLET` 定义的 `count-version` 位置上使用一个匿名函数的形式，但给这个函数一个有意义的名字可以使它易于阅读。

另外，当一个辅助函数需要进行递归时，使用一个匿名函数将不可能做到这点。³当你不想把一个递归的辅助函数定义成全局函数时，你可以使用 `LABELS`。例如，下面的函数 `collect-leaves` 使用递归的辅助函数 `walk` 来遍历一棵树并收集树中所有的原子到一个列表里，该列表（在求逆以后）被 `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)))
```

再次注意到，在 `walk` 函数里你可以引用变量 `leaves`，它是由外围的 `LET` 所引入的。

`FLET` 和 `LABELS` 用于宏展开时也是相当有用的一——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将会调用的函数，也可以只是作为一种组织宏所生成代码的方式。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。

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。当一个 `MACROLET` 宏被求值时，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遮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，但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，或是遮盖一个全局定义的宏。你将在第 31 章里看到后一种用法的一个例子。

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，当你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 `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
```

³ 可能听起来会令人惊讶，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。但是 Y 组合器属于一种有趣的理论结果，并非实用的编程工具，因为完全在本书的讨论范围之外。

```
((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)`。⁴

符号宏通常都是局部的，由 `SYMBOL-MACROLET` 所定义，但是 Common Lisp 也提供了一个宏 `DEFINE-SYMBOL-MACRO` 来定义一个全局的符号宏。一个由 `SYMBOL-MACROLET` 所定义的符号宏将覆盖由 `DEFINE-SYMBOL-MACRO` 或外围 `SYMBOL-MACROLET` 形式所定义的同名的其他符号宏。

20.3 局部控制流

接下来我将讨论的 4 个特殊操作符也会在词法环境中创建并使用名字，但却是为了调整控制流而非定义新的函数和宏。我曾经提到过所有这四个特殊操作符，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 章里讨论了将 `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。另外两个，`TAGBODY` 和 `GO` 提供了一种相当底层的 `goto` 结构作为目前你所见到的所有更上层的循环结构的基础。

一个 `BLOCK` 形式的基本结构是下面这样：

```
(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，而最后那个形式的值作为整个 `BLOCK` 的值返回，除非有一个 `RETURN-FROM` 被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` 形式由打算返回到的块名称，以及一个可选的提供了返回值的形式所组成。当一个 `RETURN-FROM` 被求值时，它会导致该命名的 `BLOCK` 立即返回。如果 `RETURN-FROM` 被调用时带有返回值，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`，都会生成含有名为 `NIL` 的 `BLOCK`。这可以允许你使用 `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)))
```

⁴ `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` 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成，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是否其展开式里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。

```
(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。这就是为什么你可以用 `RETURN-FROM` 来从一个函数中返回。

`TAGBODY` 和 `GO` 之间的关系类似于 `BLOCK` 和 `RETURN-FROM`：一个 `TAGBODY` 形式定义了一个上下文，其中定义的名字可被 `GO` 所使用。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` 特殊操作符来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))
```

注意到尽管标记名必须出现在 `TAGBODY` 的最顶层，而不能被内嵌到其他形式中，但 `GO` 特殊操作符却可以出现在 `TAGBODY` 作用域的任何位置上。这意味着你可以像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做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 并导致控制落到了 `TAGBODY` 的结尾。

`TAGBODY` 很少被直接使用，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 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 程序员得以使用许多 FORTRAN 库。由于许多 FORTRAN 库写于结构化编程革命以前，因为代码里有很多跳转 (`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。⁵

⁵ 某个版本的 `f2cl` 现在是 Common Lisp Open Code Collection (CLOCC) 的一部分，位于 <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

类似地，TAGBODY 和 GO 在转译那些以文字或框图所描述的算法时也很有用。例如，在 Donald Knuth 不朽的《The Art of Computer Programming》经典系列著作中，他使用了一种“菜谱”式的格式来描述算法：第一步，做这个；第二个，做那个；第三步，回到第二步；诸如此类。比如说，在《The Art of Computer Programming, Volume 2: Seminumerical Algorithms》第三版（Addison-Wesley, 1998）的第 142 页里，他以下面的形式描述了算法 S，该算法你将在第 27 章里用到：

算法 S (选择取样技术). 目标是从一个 N 个数的集合里选出随机的 n 个记录，其中 $0 < n \leq N$ 。

S1. [初始化] 设置 $t \leftarrow 0, m \leftarrow 0$. (在本算法中，m 代表目前已选出的记录数，而 t 是我们已经处理过的输入记录的总数量。)

S2. [生成 U] 生成一个随机数 U，其平均分布在 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] 选择用于取样的下一个记录，并将 m 和 t 递增 1。如果 $m < n$, 那么转向步骤 S2；否则取样过程完成并且算法终止。

S5. [跳过] 跳过下一个记录（不将其包含在样本中），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
        (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))))
```

这不算是精美的代码，但很容易验证它是 Knuth 算法的一个公平的转译。但是，这些代码和 Knuth 的文字描述的不同之处在于，它是可以被运行和测试的。然后你可以开始着手重构它，确

的跳转语句，他们首先将 FORTRAN 代码编译成带有那些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。很聪明的做法，但是多么难受。

保在每个改变之后该函数仍然可以工作。⁶

在经过一番优化以后，你可能会最终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)))
```

尽管可能一眼看不出来这些代码是否正确实现了算法 S，但如果你是通过一系列的与最初的 Knuth 算法描述的字面转译具有相同行为的函数到达了这里，那么你有理由相信它是正确的。

20.4 从栈上回退

作为语言的另一方面，特殊操作符还让你可以控制函数调用栈的行为。例如，尽管你可以正常使用 BLOCK 和 TAGBODY 来管理单一函数之内的控制流，但你也可以将它们与闭包一起使用，从栈的更深处函数里产生一个立即的非本地返回。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。但这个 RETURN-FROM 要直到匿名函数被 FUNCALL 或 APPLY 所调用时才会被求值。现在假设 bar 函数看起来像这样：

```
(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，而控制则从 BLOCK 返回。函数 foo、bar 和 baz 中的 FORMAT 表达式显示了这点：

```
CL-USER> (foo)
```

⁶ 由于这个算法取决于 RANDOM 所返回的值，因为你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测试 (sanity test)：

```
(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都求值得到同样的列表。

```

Entering foo
Entering BLOCK
Entering bar
  Entering baz
Leaving foo
NIL

```

注意到唯一打印出的“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，因此一个 `RETURN-FROM` 总是从它所在的词法环境中最小的外围 `BLOCK` 上返回，即便 `RETURN-FROM` 是在一个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，因此这个插进来的块对于 `RETURN-FROM` 的行为没有影响。反过来说，一个 `BLOCK` 的名字只有当 `RETURN-FROM` 出现在该 `BLOCK` 的词法作用域内部时才可以使用；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。当你调用一个含有一个 `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 章里所讨论的，词法绑定具有无限时效 (*indefinite extent*)，意味着绑定即便在绑定形式返回以后也可以保持效果。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，一个捕捉了一个块名或是 `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 `RETURN-FROM` 到这个 `BLOCK` 上，那么你将得到一个错误。同样，试图 `GO` 到一个不存在的 `TAGBODY` 上也将导致一个错误。⁷

你不太可能需要亲自使用 `BLOCK` 和 `TAGBODY` 来做这种栈回退。但你无论何时使用状况系统 (*condition system*) 时恐怕都在使用着它们，因此理解它们的工作方式应该有助于你更好地理解，比如说当调用一个再启动时究竟发生了什么。⁸

`CATCH` 和 `THROW` 是另一对可以强制回退栈的特殊操作符。相比目前为止提到的其他相关操作符来说你将更少使用这些操作符——它们是早期那些没有 Common Lisp 的状况系统的 Lisp 方言所留下的东西。他们绝对不能跟诸如 Java 和 Python 这些语言中的 `try/catch` 和 `try/except` 结构相混淆。

⁷ 这是一个相当合理的限制——从一个已经返回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。

⁸ 如果你是那种对凡事都要刨根问底的人，那么思考一下你怎样才能通过 `BLOCK`、`TAGBODY`、闭包和动态变量实现出状况系统的那些宏来，可能是相当有意义的。

CATCH和 THROW是 BLOCK和 RETURN-FROM的动态版本。这就是说，你用 CATCH包装了一个代码体然后用 THROW来导致 CATCH形式立即从一个特定值上返回。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的标签是个对象，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形式上然后导致它立即返回。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~%"))
```

注意到没有必要向下传递一个闭包了——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
```

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。这个使用了 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 状况系统的机制的古老 Lisp 方言里，CATCH和 THROW被用于错误处理。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是否一个 CATCH和 THROW会在运行期关联在一起。在 Common Lisp 中你将很少有机会使用 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另一个我之前提到过的操作符——UNWIND-PROTECT。UNWIND-PROTECT让你得以控制在栈被回退时所发生的事——为了确保特定代码在无论控制怎样离开 UNWIND-PROTECT作用域的情况下总是可以运行，无论是正常返回，通

过一个被调用的再启动，还是任何在本章中所讨论到的方式。⁹UNWIND-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这些用于清理的形式执行后被 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，因此那些与进入 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你将它作为 WITH-风格宏的基础，类似于 WITH-OPEN-FILE，它在一个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他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：¹⁰

```
(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))
```

而不必担心数据库的关闭，因为 UNWIND-PROTECT 将确保它会被关闭，不论 with-database-connection 形式的主体中发生了什么。

20.5 多值

Common Lisp 的另一个我在过去——第 11 章，当我讨论 GETHASH 时——提到过的特性，就是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，把这些内容放在一个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，而是紧密地集成到了整个语言之中。在处理多值时你最常使用的那些操作符都是宏和函数，而非特殊操作符。但最后获取多重返回值的基本能力是由一个特殊操作符 MULTIPLE-VALUE-CALL 所提供的，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理解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，那么除非你做了一些特别的事情来捕捉多个值，否则除了主值（primary value）以外的其他值都将被悄悄地丢掉。为了看到这一区别，考虑函数 GETHASH，它返回两个

⁹ UNWIND-PROTECT 本质上与 Java 和 Python 中的 try/finally 结构等价。

¹⁰ 事实上 CLSQL，跨 Lisp 平台和数据库的 SQL 接口库，确实提供了一个称为 with-database 的类似的宏。CLSQL 的主页位于 <http://clsql.b9.com>。

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在一个列表中，那么每次你调用 `GETHASH` 时你都必须解析这个列表来取得实际的值，无论你是否关心那第二个返回值。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，使得这个形式可以正常工作。

使用多重返回值有两个方面——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形式，因为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，并返回由其子形式所返回的所有值。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。¹¹

但是当一个形式被放在值的位置上求值时，只有主值将被使用，这也就是为什么之前的加法形式能够以你期待的方式运行的原因。特殊操作符 `MULTIPLE-VALUE-CALL` 提供了让你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` 是一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所返回的函数传递其余子形式所返回的所有值。

```
(funcall #'+ (values 1 2) (values 3 4)) → 4
(multiple-value-call #'+ (values 1 2) (values 3 4)) → 10
```

不过，你很少简单地需要将一个函数所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做处理。你在第 11 章里看到的 `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，然后它所返回的多个值被绑定到那些 `variable` 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
```

¹¹ 有少量有用的宏并不会传递它们所求值形式的额外返回值。特别的是，`PROG1` 宏，其像 `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`，其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` 的一个变种，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。

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所返回的多个值一次性赋值到已有变量上，那么你可以将 `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，你必须理解的一个操作符是 `EVAL-WHEN`。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来说是非常重要的，例如你将在第 24 和 31 章里编写的一些宏。

我在前面的章节里已经简要提过了 `LOAD` 和 `COMPILE-FILE` 之间的关系，但这里有必要再说一次。`LOAD` 的任务是加载一个文件并求值它所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，因为求值文件中靠前面的形式就会影响到后面形式被读取和求值的行为。例如，求值一个 `IN-PACKAGE` 形式将改变 `*PACKAGE*` 的值，这将影响后续形式被读取的方式。¹² 类似地，一个较早出现在文件中的 `DEFMACRO` 可以定义出一个可被文件

¹² 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` 在对做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，而老的值将在 `LOAD` 返回时被恢复。它还以同样方式绑定了变量 `*READTABLE*`，该变量我尚未谈及。

中后续代码所使用的宏。¹³

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`必须求值一些形式，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？在一些 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 通过从 MacLisp 中借鉴了 `EVAL-WHEN` 来避免了对类似例外情况的需要。如同其名字所暗示的，这个操作符允许你控制特定的代码在何时被求值。一个 `EVAL-WHEN` 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`——并且你所制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，同样的代码每情形一个。为了解释三种情形的含义，我将需要解释一下 `COMPILE-FILE`，它也被称为文件编译器，用来编译一个文件。

为了解释 `COMPILE-FILE` 是如何编译 `EVAL-WHEN` 形式的，我需要先介绍编译顶层形式和非顶层形式的区别。一个顶层形式简单地说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的顶层的所有形式都将被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被编译，因为 `PROGN` 本身并不做任何事——它只是把它的子形式组织在一起，然后它们会在 FASL 被加载时运行。¹⁴ 类似地，直接出现在一个 `MACROLET` 或 `SYMBOL-MACROLET` 中的形式将被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——定义了函数并且将其与函数名相关联的那些代码将会在 FASL 被加载时运行——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` 的语义取决于是否它被作为顶层形式来编译，还是作为非顶层形式来编译，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。

情形 `:compile-toplevel` 和 `:load-toplevel` 控制当一个 `EVAL-WHEN` 作为顶层形式来编译时的含义。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一个顶层 `EVAL-WHEN` 的情形列表中，那么编译器将会忽略它。

¹³ 在某些实现中，你可能可以正确求值一个在函数体中使用了未定义宏的 `DEFUN` 定义，只要改宏在函数实际被调用之前定义好即可。但就算可以正常工作，也是仅当从源代码 `LOAD` 这些定义时，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。

¹⁴ 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，但是却在由 `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` 将按照 `PROGN` 来对待，但其实不是——出现在 `LET` 中的子形式不会被作为顶层形式来对待。

当一个 EVAL-WHEN 作为非顶层形式被编译时，它要么在 :execute 情形被指定时像一个 PROGN 那样被编译，要么被忽略。类似地，一个被求值的 EVAL-WHEN——这包括那些被 LOAD 所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 的子形式中的编译期 EVAL-WHEN——要么在 :execute 存在时作为一个 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从源代码加载期都能产生效果：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf *package* (find-package "PACKAGE-NAME")))
```

PACKAGE 在编译期被设置是因为有 :compile-toplevel 情形，当 FASL 被加载时被设置时因为有 :load-toplevel，而在从源代码加载时被设置是因为有 :execute。

你会经常在两种情况下使用 EVAL-WHEN。一种是如果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。

其他你可能需要 EVAL-WHEN 的场合是，如果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，你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，那么这些定义在宏的展开函数运行时就可以使用了。你将很可能想要再加上 :load-toplevel 和 :execute，因为该宏在文件被编译加载或者从源代码不编译而直接加载以后也需要这些函数。

20.7 其他特殊操作符

其余的 4 个特殊操作符 LOCALLY、THE、LOAD-TIME-VALUE 和 PROGV 都可以允许你访问到其他任何方式都无法访问的语言的底层部分。LOCALLY 和 THE 是 Common Lisp 声明系统的一部分，它们被用来与编译器沟通而对你代码的含义没有影响，但可能帮助编译器生成更好的代码——更快、更清晰的错误信息，以及诸如此类。¹⁵我将在第 32 章里简单地谈及有关声明的内容。

另外两个，LOAD-TIME-VALUE 和 PROGV 都很少用到，并且解释你为什么可能会想要使用它们，比解释它们能干什么还费劲。因此我将只是告诉你它们能干什么，从而让你知道它们的存在。日后你将偶尔遇到刚好可以用上它们的场合，那时你就知道该怎么用了。

LOAD-TIME-VALUE 正如其名字所暗示的，被用来创建一个在运行期决定的值。当文件编译器编译到含有一个 LOAD-TIME-VALUE 形式的代码时，它会安排在 FASL 被加载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))
```

¹⁵ 唯一一个对程序语义有影响的声明是第 6 章里提到过的 SPECIAL 声明。

```
(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，这可能是你显式地用 `COMPILE` 编译了一个函数，或是在求值代码的过程中具体实现进行了隐式的编译。在不编译的代码中，`LOAD-TIME-VALUE` 在其每次被求值时都会求值其子形式。

最后，`PROGV` 可以创建名字在运行期才决定的变量绑定。这对于为支持动态作用域变量的语言实现嵌入式解释器将尤其有用。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，然后那些 `body-form` 被求值。`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决定。要我说的话，这并不是你经常需要干的事情。

这就是关于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上并向你展示如何使用 Common Lisp 的包系统来控制你的名字空间，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来。

第21章 编写大型程序：符号和包

在第 4 章里，我讨论了 Lisp 读取器是如何将文本名字转化成用来传递给求值器的对象的，它们是一种称为符号 (symbol) 的对象。实践表明，拥有一种特定用来表示各种名字的内置数据类型对于多种编程任务都是有用的。¹不过，这并不是本章的主题。在本章里，我将讨论处理名字的许多实用方面之一：如何在彼此无关的人开发的代码之间避免名字冲突。

举个例子，假设你正在编写一个程序并决定在其中用到一个第三方库。你不会希望知道那个库中的每一个函数、变量、类和宏的名字，以便避免这些名字与你自己的程序中所使用的名字相冲突。你更希望这个库中的大多数名字和你程序中的名字是彼此无关的，哪怕是它们碰巧使用了相同的文本表示。同时，你希望这个库所定义的特定名字就是用来被访问的——这些名字构成了它的公共 API，你将会在自己的程序中用到它们。

在 Common Lisp 中，这一名字空间问题最后归结到如何控制读取器将文本名称转化成符号这个问题上了：如果你希望相同的名字可以被求值器同等看待，那么你需要确保读取器使用相同的符号来表示每个名字。同样地，如果你希望两个名字被视作不同的，甚至当它们刚好具有相同的文本名字时，那么你就需要让读取器可以分别创建不同的符号来表示它们。

21.1 读取器是如何使用包的

在第 4 章里，我简单讨论了 Lisp 读取器是如何将名字转化成符号的，但我刻意忽略了大部分细节——现在是时候从更近的角度来看看这个过程中究竟发生了什么了。

我将从描述读取器所理解的名字表示语法以及该语法和包之间的关系开始。目前你刻意将包想像成一个字符串和符号的映射表。如同你在下一节里将会看到的，实际的映射过程比一个简单的查找表稍微灵活一些，但这对读者来说通常意义不大。每个包也都有一个名字，该名字可被用来通过函数 `FIND-PACKAGE` 找到对应的包。

读取器用来访问一个包中的名字-符号映射的两个关键函数是 `FIND-SYMBOL` 和 `INTERN`。这两个函数都接受一个字符串和一个可选的包。当未指定后者时，包参数默认为全局变量 `*PACKAGE*` 的值，也称为当前包。

`FIND-SYMBOL` 在包中查找名为给定字符串的符号并将其返回，如果没有符号被找到则返

¹ 依赖于符号数据类型的编程，被恰如其名地称为符号计算。它和基于数值的编程正好相反。一个所有程序员都应当熟悉的主要的符号计算程序的例子是编译器——它将一个程序的文本视为符号数据并将其转化成一种新的形式。

回 NIL。INTERN也会返回一个已有的符号；否则它会创建一个以该字符串为名字的新符号并将其添加到包里。

你所使用的大多数名字都是非限定的 (unqualified)，就是说名字里不带有冒号。当读取器读到这样一个名字时，它先将名字中所有未转义的字母转换成大写形式然后将得到的字符串传给 INTERN，从而将该名字转化成了一个符号。这样，每当读取器读到相同的包里面相同的名字时，它都将得到相同的符号对象。这是很重要的，因为求值器使用符号的对象标识来决定一个给定函数所指向的函数、变量或其他程序元素。因此，诸如 (hello-world) 这样的表达式得以调用一个特定的 hello-world 函数的原因就在于读取器在读取对该函数的调用和定义该函数的 DEFUN 形式时返回了相同的符号。

一个含有单冒号或双冒号的名字是一个包限定了的名字。当读取器读取一个包限定的名字时，它将名字在冒号处拆开，前一部分作为包的名字，后一部分作为符号的名字。读取器查找适当的包并用它来将符号名转化成一个符号对象。

一个只含有单个冒号的名字必须指向一个外部符号 (external symbol) ——一个被包导出 (export) 作为公共接口来使用的符号。如果命名的包不含有一个给定名字的符号，或是含有该符号但并未导出，那么读取器将会产生一个错误。一个双冒号的名字可以指向命名包中的任何符号，尽管这通常不是个好主意——导出符号的集合定义了一个包的公共接口，而如果你不遵守包作者关于哪些名字是公开的而哪些名字是私有的约定，那么你在使用时肯定会遇到麻烦。另一方面，有时一个包的作者可能忽略了导出一个确实应当开放给公众的符号。在这种情况下，一个双冒号的命令可以让你在无需等待该包的下一个版本发布即可完成手头的工作。

读取器所理解的符号语法的另外两点，分别是关键字符和未进入 (uninterned) 的符号。关键字符在书写上以一个冒号开始。这类字符在名为 KEYWORD 的包中创建并自动导出。更进一步，当读取器在 KEYWORD 包中创建一个符号时，它也会定义一个常量，以该符号作为其名字和值。这就是为什么你可以在参数列表中使用关键字而无需引用它们——当它们出现在一个值的位置上时，它们求值到它们自身。这样：

```
(eq1 ':foo :foo) -> T
```

关键字符的名字，和所有符号一样，在她们被创建之前就被读取器全部转换成大写的形式了。名字中不包括前导冒号。

```
(symbol-name :foo) -> "FOO"
```

未进入的字符在写法上以 “#:” 开始。这些名字（去掉 “#:” 后）被正常地转化成大写形式并被转化成符号，但这些符号还没有进入任何包；每当读取器读到一个带有 “#:” 的名字，它都会创建一个新的符号。这样：

```
(eq1 '#:foo '#:foo) -> NIL
```

你很少需要自行书写这种语法，但有时当你打印一个含有由 GENSYM 所返回的符号的 S-表达式时就会看到它们。

```
(gensym) -> #:G3128
```

21.2 包和符号的词汇表

如同我先前提到的，一个包所实现的从名字到符号之间的映射比一个简单的查找表要更加灵活一些。在核心层面上，每一个包都含有一个名字到符号的查找表，但一个符号还有其他几种方式使其可以通过一个给定包中的非全称名字访问到。为了更有意义地谈论其他这些方法，你将需要一点儿词汇表。

首先，所有可在在一个给定包中通过 `FIND-SYMBOL` 找到的符号被称为是在该包中“可访问的”(accessible)。换句话说，一个包中的可访问符号是那些当该包为当前包时非全称名字可指向的那些符号。

一个符号可以通过两种方式被访问到。前一种方式要求包的名字-符号表中含有该符号的项，这时我们称该符号“存在”(present)于该包中。当读取器让一个新符号进入一个包时，该符号会被添加到包的名字-符号表中。这个被该符号首先进入的包被称作该符号的“主包”(home package)。

一个符号可在某个包中访问的另一种方式是当该包继承(inherit)了它时。一个包通过“使用”(use)其他一些包来继承这些包中的符号。只有被使用的包中的“外部”(external)符号才能被继承。可以通过在包中“导出”(export)一个符号来使其成为外部符号。导出操作除了可以让符号被包的其他使用者继承以外，还可以使其能够通过带有单个冒号的限定名称来引用，如同你在上一节里所看到的那样。

为了保证从名字到符号的映射的确定性，包系统只允许每个名字在给定的包中指向单一符号。这就是说，一个包不能同时有一个存在的符号和一个同名的继承符号，或是同时从不同的包中继承了两个具有相同名字的不同的符号。不过，你可以通过使其中一个可访问的符号成为隐蔽(shadow)符号来解决冲突，这可以使其他同名的符号成为不可访问的。每一个包都在它们的名字-符号列表之外维护了一个隐蔽符号的列表。

一个已有的符号可以通过将其添加到另一个包的名字-符号表中来“导入”(import)到这个包。这样，同一个符号可以同时存在于多个包中。有时你导入符号只是因为你希望它们在被导入的包中可以直接访问而无需使用它们的主包。其他时候你导入一个符号则是因为只有存在的符号才可以被导出或是成为隐蔽符号。举个例子，如果一个包需要使用两个带有同名外部符号的包，那么其中一个符号必须被导入进该包，从而可被添加到该包的隐蔽符号列表并使得另一个符号成为不可访问的。

最后，一个存在的符号可以从一个包中退出(unintern)，这导致其从名字-符号表中被清除，并且如果它是一个隐蔽符号也会从隐蔽符号表中被清除。你可能想要让一个符号从一个包中退出，从而消除该符号和一个来自你想要使用的包中的外部符号之间的冲突。一个不存在于任何包中的符号被称为“未进入”(uninterned)的符号，它不能被读取器所读取，并且将采用 `#:foo` 这样的语法进行打印。

21.3 三个标准包

在下一节里我将向你展示如何定义你自己的包，包括如何让一个包使用另一个，以及如何导出、隐蔽以及导入符号。不过首先让我们看一些你已经在使用着的包。当你最初启动 Lisp 时，`*PACKAGE*` 的值通常是 `COMMON-LISP-USER` 包，有时也叫做 `CL-USER`。² `CL-USER` 使用了包 `COMMON-LISP`，后者导出了语言标准所定义的所有名字。因此，当你在 REPL 中键入一个表达式时，所有标准函数、宏、变量之类的名字都将被转化成从 `COMMON-LISP` 包中导出的符号，而所有其他名字将进入到 `COMMON-LISP-USER` 包中。例如，名字 `*PACKAGE*` 是从 `COMMON-LISP` 包中导出的——如果你想要查看 `*PACKAGE*` 的值，你可以像这样输入：

```
CL-USER> *package*
#<The COMMON-LISP-USER package>
```

这是因为 `COMMON-LISP-USER` 使用了 `COMMON-LISP`。或者你也可以使用一个包限定的名字。

```
CL-USER> common-lisp:*package*
#<The COMMON-LISP-USER package>
```

你甚至可以使用 `COMMON-LISP` 的昵称，`CL`。

```
CL-USER> cl:*package*
#<The COMMON-LISP-USER package>
```

但是 `*x*` 不是 `COMMON-LISP` 中的符号，因此如果你输入下面这个：

```
CL-USER> (defvar *x* 10)
*x*
```

那么读取器将把 `DEFVAR` 作为 `COMMON-LISP` 中的符号来读取，而把 `*x*` 作为 `COMMON-LISP-USER` 中的符号来读取。

REPL 不能在 `COMMON-LISP` 包中启动是因为你不被允许在这个包中添加新符号；`COMMON-LISP-USER` 是作为一个“模板”包来提供的，在这里你可以创建你自己的名字，同时还能轻松地访问到 `COMMON-LISP` 的所有符号。³ 通常情况下，你将定义的所有包都将使用 `COMMON-LISP`，因此你不需要写出类似下面这样的代码：

```
(cl:defun (x) (cl:+ x 2))
```

第三个标准包是 `KEYWORD` 包，这个包被 Lisp 读取器用来创建以冒号开始的名字。这样，你

² 每一个包都有一个正式名字和零或多个昵称（nickname），后者可被用在任何需要用到包名的地方，例如包限定的名字，或是在一个 `DEFPACKAGE` 或 `IN-PACKAGE` 形式中指向那个包。

³ `COMMON-LISP-USER` 也被允许提供对其他由语言实现所定义的包导出的符号的访问。尽管这在本意上是为用户提供方便——它使得实现相关的功能易于访问——但它也给 Lisp 新手带来的许多疑惑：Lisp 将会抱怨对于重定义某些语言并未涉及的符号的重定义。要想看到在一个特定的实现中 `COMMON-LISP-USER` 都从哪些包中继承了符号，可以在 REPL 中求值下列表达式：

```
(mapcar #'package-name (package-use-list :cl-user))
```

而要想查出一个符号最初来源于哪个包，可以求值下面这个表达式：

```
(package-name (symbol-package 'some-symbol))
```

同时把 `some-symbol` 替换成你想要的符号。例如：

```
(package-name (symbol-package 'car)) -> "COMMON-LISP"
```

```
(package-name (symbol-package 'foo)) -> "COMMON-LISP-USER"
```

继承自实现定义的包的符号将返回一些其他的值。

也可以像下面这样使用显式的包限定方式来引用任何关键字符号：

```
CL-USER> :a
:A
CL-USER> keyword:a
:A
CL-USER> (eql :a keyword:a)
T
```

21.4 定义你自己的包

工作在 `COMMON-LISP-USER` 包里对于 REPL 中的试验是好的，不过一旦你开始编写实际的程序你将需要定义新的包，这样不同的程序可以加载到同一个 Lisp 环境中而不会破坏彼此的名字。而当你编写可能用于不同环境中的库时，你将希望定义分开的包并且导出那些构成了库的公共 API 的符号。

尽管如此，在你开始定义包之前，理解关于包无法做到的一件事是很重要的。包无法提供对于谁可以调用什么函数或访问什么变量的直接控制。它们只提供你基本的对于名字空间的控制，这通过控制读取器将文本名字转化成符号对象来完成，但在后面求值器中当符号被解释成一个函数或变量或其他任何东西的名字时，包机制就无能为力了。因此，谈论把一个函数或变量从一个包中导出是没有意义的。你可以导出符号以便特定的名字更加易于访问，但包系统并不能允许你限制这些名字是如何被使用的。⁴

记住上述这点以后，下面你可以开始学习如何定义包并将它们捆绑在一起了。你可以通过宏 `DEFPACKAGE` 来定义新的包，它允许你在创建包的同时还能指定它使用哪些包、导出哪些包，以及它从其他包里导入什么符号，还有为了解决冲突需要创建的隐蔽符号。⁵

我将在假设你正在使用包来编写一个将 e-mail 消息组织进一个可搜索数据库的背景下描述所有有关的选项。这个程序是完全假想出来的，包括我将引用到的其他库在内——要点在于查看用于这样一个程序的包可以怎样组织。

你需要的第一个包是那个提供了整个应用程序命名空间的包——你需要命名你的函数、变量和诸如此类的东西，而无需担心和不相关的代码产生名字冲突。因此你最好用 `DEFPACKAGE` 定义一个新的包。

如果应用程序写得足够简单，没有用到超出语言本身所提供的能力之外的库，那么你可以定义一个下面这样的简单包：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp))
```

这定义了一个包，名为 `COM.GIGAMONKEYS.EMAIL-DB`，其继承了由 `COMMON-LISP` 包所导出的

⁴ 这与 Java 的包系统不同，后者在提供了类的名字空间的同时还引入了 Java 的访问控制机制。非 Lisp 语言中包系统最接近 Common Lisp 的语言是 Perl。

⁵ `DEFPACKAGE` 进行的所有处理也都可以通过管理包对象的函数来完成。尽管如此，由于一个包在通常情况下都要在使用前完全定义，所以这些函数很少用到。另外，`DEFPACKAGE` 可以采用正确的顺序来完成所有的包管理操作——例如，`DEFPACKAGE` 可以在使用那些用到的包之前将有关符号添加到隐蔽符号列表中。

所有符号。⁶

事实上对于如何表示包的名字你有几种选择，并且如同你将会看到的，一个 DEFPACKAGE 中符号的名字也有几种表示方法。包和符号都是用字符串来命名的。不过，在一个 DEFPACKAGE 形式中，你可以使用“字符串描述符”(string designator) 来指定包和符号的名字。一个字符串描述符要么是一个字符串，代表其自身；要么是一个符号，代表其名字；要么是一个字符，代表一个含有该字符的单字符的串。像上面的 DEFPACKAGE 那样使用关键字符串，是一种允许你把名字书写成小写字母的常用风格——读取器将为你把名字转换成大写。你也可以用字符串来书写 DEFPACKAGE，但这样的话你需要全部使用大写，因为多数符号和包真正的名字由于读取器的大小写转换约定在事实上都是大写的。⁷

```
(defpackage "COM.GIGAMONKEYS.EMAIL-DB"
  (:use "COMMON-LISP"))
```

你也可以使用非关键字的符号——DEFPACKAGE 中的名字不会被求值——但是随后读取 DEFPACKAGE 形式的操作将导致这些符号进入到当前的包中，这在某种程度上泄露了名字空间并可能也会在以后你试图使用该包时带来问题。⁸

为了读取这个包中的代码，你需要使用 IN-PACKAGE 宏来使其成为当前包：

```
(in-package :com.gigamonkeys.email-db)
```

如果你在 REPL 中输入这个表达式，它将改变 *PACKAGE* 的值，从而影响 REPL 读取后续表达式的方式，直到你通过另一个 IN-PACKAGE 调用来改变它。类似地，如果你在用 LOAD 加载的文件或 COMPILE-FILE 编译的文件中包含了一个 IN-PACKAGE，那么它将改变当时的包，从而影响文件中后续表达式的读取方式。⁹

通过将当前包设置为 COM.GIGAMONKEYS.EMAIL-DB 包，除了那些继承自 COMMON-LISP 的名字以外，你可以使用几乎任何你想要的名字用于你想要的任何目的。这样，你可以定义一个新的 hello-world 函数，与先前定义在 COMMON-LISP-USER 中的函数共存。这是已有函数的行为：

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

现在你可以用 IN-PACKAGE 切换到新的包上。¹⁰ 注意到提示符的改变——具体的形式取决于开

⁶ 在许多 Lisp 实现里，如果你只是使用 COMMON-LISP 包的话，那么 :use 子句是可选的——如果你省略了它，包将自动从一个由具体实现所定义的包列表中继承名字，而这个包列表中通常都包括 COMMON-LISP。尽管如此，如果你总是显式地指定你想要 :use 的包列表，那么你的代码将更加可移植。那些珍惜键盘的人可以使用包的简称而写成 (:use :cl)。

⁷ 使用关键字来代替字符串还有另一个优点——Allegro 支持一种“现代模式”的 Lisp，其中读取器并不做大小写转换，并且它在带有大写名称的 COMMON-LISP 包以外还提供了一个使用小写字母的 common-lisp 包。严格来讲，这种 Lisp 并不符合 Common Lisp 标准，因为所有标准中定义的名字都被定义成大写的。但如果你使用关键字符串来编写你的 DEFPACKAGE 形式，那么它将可以同时工作在 Common Lisp 和与之相近的另一种模式下。

⁸ 一些人使用 “#:” 语法的未进入的符号来代替关键字符串。

⁹ 使用 IN-PACKAGE 而不是仅仅用 SETF 来修改 *PACKAGE* 的原因在于，IN-PACKAGE 可以展开成在文件被 COMPILE-FILE 编译时和文件加载时都会执行的代码，从而在编译期就可以改变读取器读取文件其余部分的方式。

¹⁰ 在 SLIME 的 REPL 缓冲区里，你也可以使用 REPL 快捷键来改变当前包。键入一个逗号，然后在 Command: 提示上输入 change-package。

发环境，但在 SLIME 中默认提示符由包名的简化版本构成。

```
CL-USER> (in-package :com.gigamonkeys.email-db)
#<The COM.GIGAMONKEYS.EMAIL-DB package>
EMAIL-DB>
```

你可以在这个包里定义一个新的 hello-world:

```
EMAIL-DB> (defun hello-world () (format t "hello from EMAIL-DB package~%"))
HELLO-WORLD
```

然后像下面这样测试它:

```
EMAIL-DB> (hello-world)
hello from EMAIL-DB package
NIL
```

现在切换回 CL-USER。

```
EMAIL-DB> (in-package :cl-user)
#<The COMMON-LISP-USER package>
CL-USER>
```

旧的函数行为没有被干扰。

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

21.5 打包可重用的库

尽管工作在 e-mail 数据库上，但你可能需要编写几个与存取文本相关的而与 e-mail 无关的函数。你可能意识到这些函数会对其他程序有用并且决定将它们重新打包成一个库。你应当定义一个新的包，但这次你将导出特定的一些名字使其对于其他包可见。

```
(defpackage :com.gigamonkeys.text-db
  (:use :common-lisp)
  (:export :open-db
           :save
           :store))
```

再一次，你使用了 COMMON-LISP 包，因为你将需要在 COM.GIGAMONKEYS.TEXT-DB 中访问标准函数。`:export` 子句指定了将从 COM.GIGAMONKEYS.TEXT-DB 中的外部名字，这些名字可以被所有 `:use` 它的包直接访问。因此，在你定义了这个包以后，你可以将主应用程序包的定义改变成下面的样子:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db))
```

现在 COM.GIGAMONKEYS.EMAIL-DB 中编写的代码可以同时使用非限定名字来引用 COMMON-LISP 和 COM.GIGAMONKEYS.TEXT-DB 中所导出的符号。所有其他的名字都将继续直接进入到 COM.GIGAMONKEYS.EMAIL-DB 包。

21.6 导入单独的名字

现在假设你找到了一个可以处理 e-mail 消息的第三方函数库。该库的 API 中所使用的名字是从 `COM.ACME.EMAIL` 包中导出的，因此你可以通过 `:use` 这个包来轻松获得对这些名字的访问权限。但假设你只需要用到来自这个库的一个函数，而其他的导出符号跟你已经使用（或计划使用）的符号有冲突。¹¹ 在这种情况下，你可以使用 `DEFPACKAGE` 中的 `:import-from` 子句只导入你需要的那个符号。例如，如果你想要使用的函数名为 `parse-email-address`，那么你可以把 `DEFPACKAGE` 改写成下面这样：

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db)
  (:import-from :com.acme.email :parse-email-address))
```

现在 `COM.GIGAMONKEYS.EMAIL-DB` 包中的任何代码里出现的 `parse-email-address` 都将被读取为 `COM.ACME.EMAIL` 中的同名符号。如果你需要从单个包中导入超过一个符号，那你只需在单个 `:import-from` 子句中包名之后包含多个名字即可。一个 `DEFPACKAGE` 也可以含有多个 `:import-from` 子句以便从不同的包中分别导入符号。

有时你可能会遇到相反的情况——一个包可能导出了一大堆你想要使用的名字，但还有少数是你不需要的。与其将所有你需要的符号都列在一个 `:import-from` 子句中，你还可以直接 `:use` 这个包，同时把你不需要继承的符号放在一个 `:shadow` 子句里。例如，假设 `COM.ACME.TEXT` 包导出了许多用于文本处理的函数和类的名字。更进一步假设这些函数和类的多数都是你想要用在你的代码中的，但其中一个名字 `build-index` 却跟你已经使用的名字冲突了。你可以通过隐蔽这个符号将来自 `COM.ACME.TEXT` 的 `build-index` 符号设置为不可见的。

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow build-index))
```

这个 `:shadow` 子句导致一个新的名为 `BUILD-INDEX` 的符号被创建并直接添加到 `COM.GIGAMONKEYS.EMAIL-DB` 的名字-符号映射表中。现在如果读取器读到了名字 `BUILD-INDEX`，它将把它转化成 `COM.GIGAMONKEYS.EMAIL-DB` 的表中的符号，而不会是继承自 `COM.ACME.TEXT` 中的那个符号了。这个新的符号也会被添加到作为 `COM.GIGAMONKEYS.EMAIL-DB` 包的一部分的一个隐蔽符号列表中，所以如果你以后使用了另一个同样导出了 `BUILD-INDEX` 符号的包，包系统将会知道这里没有冲突——就是说你总是希望使用来自 `COM.GIGAMONKEYS.EMAIL-DB` 的符号而不是从其他包里继承得到的同名符号。

一个类似的情形出现在当你想要使用两个导出了同样名字的包时。在这种情况下读取器在读到文本名字时不可能知道你究竟想要使用哪一个继承的符号。在这种情况下你必须通过隐蔽有冲

¹¹ 在开发过程中，如果你试图 `:use` 一个包，它导出了一些与你当前所在包的已有符号同名的符号，那么 Lisp 将会立即报错并通常会提供给你一个再启动，把所用到的包中的那个符号去掉。关于这个过程的更多细节，请参见“包的各种疑难杂症”一节。

突的名字来解决冲突。如果你根本就不需要使用这个名字，那你可以通过一个`:shadow`子句将改名字隐蔽掉，从而在你的包中创建出一个同名的新符号来。但如果你确实想要使用继承来的符号中的一个，那么你需要通过一个`:shadowing-import-from`子句来消除歧义。和`:import-from`子句一样，一个`:shadowing-import-from`子句由一个包名紧接着需要从那个包中导入的名字所构成。举个例子，如果`COM.ACME.TEXT`导出了一个名字`SAVE`，它与`COM.GIGAMONKEYS.TEXT-DB`中导出的名字相冲突，那么你可以使用下面的`DEFPACKAGE`形式来消除歧义：

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index)
  (:shadowing-import-from :com.gigamonkeys.text-db :save))
```

21.7 打包技巧

前面已经讨论了一些常见情形下用包来管理名字空间的方式。尽管如此，关于如何使用包的另一层面的内容也是值得讨论的——关于如何组织代码来使用不同的包的基本技巧。在本节里，我将讨论关于如何组织代码的一些概括性规则——相对于那些通过`IN-PACKAGE`来使用你的包的代码来说应该在哪里保存你的`DEFPACKAGE`形式。

因为包要被读取器使用，因此一个包必须在你可以`LOAD`或`COMPILE-FILE`一个含有切换到该包的`IN-PACKAGE`表达式的文件之前就被定义。包也必须被定义在可能用到它的其他`DEFPACKAGE`形式之前。举个例子，如果你打算在`COM.GIGAMONKEYS.EMAIL-DB`中使用`COM.GIGAMONKEYS.TEXT-DB`包，那么`COM.GIGAMONKEYS.TEXT-DB`的`DEFPACKAGE`必须在`COM.GIGAMONKEYS.EMAIL-DB`的`DEFPACKAGE`之前被求值。

确保包在他们被用到之前总是存在的最佳方法是把你所有的`DEFPACKAGE`定义放在与需要在这些包里读取的源代码分开的文件里。一些人喜欢针对每个单独的包都创建一个形如`foo-package.lisp`的文件，而另一些人则创建单个`packages.lisp`来包含一组相关的包的所有`DEFPACKAGE`形式。两种思路都是合理的，尽管每个包一个文件的方法也对你组织并根据包之间的依赖关系以正确的顺序加载单独的文件提出了要求。

不论用哪种方式，一旦所有的`DEFPACKAGE`形式都从那些用到它们的代码中分离出来了，你就可以调整加载文件的顺序，让那些含有`DEFPACKAGE`的文件在你编译或加载任何其他文件之前进行加载。对于简单的程序你可以手工完成这件事：简单地`LOAD`那些含有`DEFPACKAGE`形式的文件，其中有可能需要先用`COMPILE-FILE`编译它们。然后加载那些使用这些包的文件，再一次可选预先用`COMPILE-FILE`编译它们。不过，需要注意的是，直到你用`LOAD`加载那些无论是以源代码的形式还是`COMPILE-FILE`输出的文件形式存在的包定义之前，包都是不存在的。这样，如果你正在编译所有的文件，你仍然必须先`LOAD`所有的包定义，然后再编译那些需要从这些包里读取符号的文件。

完全手工来做这些事很快就会令人厌烦。对于简单的程序来说你可以通过编写一个文件`load.lisp`来自动完成这些步骤，该文件里含有正确顺序排列的适当的`LOAD`和`COMPILE-FILE`

调用。然后你只需 LOAD 那个文件就好了。对于更复杂的程序，你将希望使用一种系统定义功能 (system definition facility) 来管理正确的加载和编译文件的顺序。¹²

另一个关键的概括性规则是每个文件里应该只含有一个 IN-PACKAGE 形式，并且它应当是该文件中除注释以外的第一个形式。含有 DEFPACKAGE 形式的文件应当以 (in-package "COMMON-LISP-USER") 开始，而所有其他的文件都应当含有一个属于你的某个包的 IN-PACKAGE 形式。

如果你违反了这个规则，在文件中间切换了当前包，那么你将迷惑那些没有注意到第二个 IN-PACKAGE 的人类读者。另外，许多 Lisp 开发环境，尤其是诸如 SLIME 这种基于 Emacs 的环境，通过查看 IN-PACKAGE 来决定与 Common Lisp 通讯时所使用的包。每个文件里有多个 IN-PACKAGE 的话可能会干扰这些工具。

另一方面，多个文件以相同的包来读取是没有问题的，每个文件都使用相同的 IN-PACKAGE 形式。问题只是你想要怎样组织你的代码。

关于打包技巧的最后一点儿内容是如何给包命名。所有包名都存在于扁平的名字空间里——包名只是字符串，而不同的包必须带有文本上可区分的名字。这样，你就不得不考虑包名字冲突的可能性。如果你只用了你自己开发的包，那么你可以随意地为你的包使用短名称。但如果你正在计划使用第三方库或是发布你的代码给其他程序员使用，那么你需要遵守一个可以最小化不同包之间名字冲突的命名约定。近来的许多程序员都采纳了 Java 风格的命名，如同本章里使用的那些包名一样，它由一个逆向的 Internet 域名后跟一个点和一个描述性的字符串所组成。

21.8 包的各种疑难杂症

一旦你熟悉了包，你就不会再花许多时间来思考它们了。其实本来也没什么可思考的。不过，一些困扰新 Lisp 程序员的疑难杂症使得包系统看起来比它实际的情况更加复杂和不友好了。

排名第一的疑难杂症通常出现在使用 REPL 的时候。你当时正在寻找一些定义了特定的兴趣的函数的库。你将试图像下面这样调用它们中的一个：

```
CL-USER> (foo)
```

然后以下列错误信息掉入了调试器：

```
attempt to call 'FOO' which is an undefined function.
[Condition of type UNDEFINED-FUNCTION]
Restarts:
 0: [TRY-AGAIN] Try calling FOO again.
 1: [RETURN-VALUE] Return a value instead of calling FOO.
 2: [USE-VALUE] Try calling a function other than FOO.
 3: [STORE-VALUE] Setf the symbol-function of FOO and call it again.
 4: [ABORT] Abort handling SLIME request.
 5: [ABORT] Abort entirely from this (lisp) process.
```

啊，当然——你忘了使用那个库的包。于是你退出调试器并试图 USE-PACKAGE 该库的包来得

¹² 所有通过本书的 Web 站点可获得的那些来自“实用”章节的代码，都实用了 ASDF 系统定义库。我将在第 32 章里讨论 ASDF。

到对名字 `foo` 的访问然后再调用该函数。

```
CL-USER> (use-package :foolib)
```

但这次却以下列错误信息再次掉入了调试器：

```
Using package 'FOOLIB' results in name conflicts for these symbols: FOO
[Condition of type PACKAGE-ERROR]
Restarts:
 0: [CONTINUE] Unintern the conflicting symbols from the COMMON-LISP-USER package.
 1: [ABORT] Abort handling SLIME request.
 2: [ABORT] Abort entirely from this (lisp) process.
```

啊？问题在于第一次你调用 `foo` 的时候，读取器读取名字 `foo` 并在求值器接手并发现这个新创建的符号不是一个函数的名字之前就将其进入了 `CL-USER` 包。这个新的符号然后又和 `FOOLIB` 包中导出的同名符号相冲突了。如果你在试图调用 `foo` 之前记得了 `USE-PACKAGE FOOLIB`，那么读取器就可以将 `foo` 读取成一个继承而来的符号而不会在 `CL-USER` 中创建一个新符号。

不过，现在还为时不晚，因为调试器给出的第一个再启动将会以正确的方式把事情修补好：它将把 `foo` 符号从 `COMMON-LISP-USER` 中退出，从而把 `CL-USER` 包恢复到了调用 `foo` 之前的状态，从而允许 `USE-PACKAGE` 得以进行并允许继承得来的 `foo` 在 `CL-USER` 中可用。

这类问题也会发生在加载和编译文件时。举个例子，如果你定义了一个包 `MY-APP`，其中的代码打算使用来自 `FOOLIB` 包的函数名字，但是当你在 `(in-package :my-app)` 下编译文件时却忘记了 `:use FOOLIB`，这样读取器会将这些原本打算从 `FOOLIB` 中读取的符号改为在 `MY-APP` 中创建新符号。当你试图运行编译后的代码时，你将得到函数未定义的错误。如果你随后试图重定义 `MY-APP` 包，加上 `:use FOOLIB`，那么你就会得到符号冲突的错误。解决方案是一样的：选择再启动来从 `MY-APP` 中解除有冲突的符号。然后你将需要重新编译 `MY-APP` 包中的代码，这样它们就可以指向继承的那些名字了。

下一个疑难杂症本质上是最初那个的相反形式。在这种情况下，你已经定义了一个用到了比如说 `FOOLIB` 的包——再次假设它是 `MY-APP`。现在你开始编写 `MY-APP` 中的代码。尽管你使用 `FOOLIB` 是为了能够引用 `foo` 函数，但 `FOOLIB` 同时还导出了其他一些符号。如果你把其中一个导出的符号——比如说 `bar`——用作了你自己代码里的某个函数的名字，那么 Lisp 将不会报错。相反，你的函数名将会是 `FOOLIB` 所导出的那个符号，这将会破坏 `FOOLIB` 中 `bar` 的定义。

这个问题的危害更大，因为它不会明显地报错——从求值器的观点来看这只是要求将一个新函数关联到一个旧的名字上，有时这是完全合法的。唯一的可疑之处只是做这个重定义的代码是在一个与函数名符号所在的包名不同的 `*PACKAGE*` 下被读取的。但求值器不必关心这点。不过，在多数 Lisp 环境下你会得到一个关于“`redefining BAR, originally defined in ...`”的警告。你应当留心那些警告。如果你破坏了一个库中的一个定义，那么你可以通过使用 `LOAD` 重新加载库的代码来恢复它。¹³

最后一个包相关的疑难杂症相对来说是比较简单的，但它给多数 Lisp 程序员至少带来了几次麻烦：你定义了一个使用 `COMMON-LISP`，同时可能还有其他一些库的包。然后，在 REPL 中你

¹³ 某些 Common Lisp 实现，包括 Allegro 和 SBCL，提供了一种“锁定”特定包中的符号的机制，这一机制可以确保只有当诸如 `DEFUN`、`DEFVAR` 和 `DEFCLASS` 这类定义形式所在的主包是当前包时才能顺利通过。

切换到那个包里去做一些事。然后你决定完全退出 Lisp 环境并试图调用 (`quit`)。不过, `quit` 并不是来自 `COMMON-LISP` 包的名字——它被具体实现定义在了某个实现相关的包里了, 后者往往是被 `COMMON-LISP-USER` 包所使用的。解决方案很简单——切换回 `CL-USER` 包然后再退出。或者使用 SLIME 的 REPL 快捷键 `quit`, 它可以使你免去记忆特定 Common Lisp 实现的退出函数究竟是 `exit` 还是 `quit` 的烦恼。

至此你基本完成了对 Common Lisp 的了解。在下一章里我将讨论扩展形式的 `LOOP` 宏的细节。然后, 本书的其余部分将致力于“实践”: 一个垃圾过滤器、一个用来解析二进制文件的库, 以及一个带有 Web 接口的流式 MP3 服务器的各部分。

第22章 高阶 LOOP

在第 7 章里我简要讨论了扩展形式的 LOOP 宏。正如我当时提到的，LOOP 本质上提供了用来编写迭代构造的专用语言。

这可能看起来有些无聊——发明一种全新的语言却只为了书写循环。但如果你深入思考循环结构在程序中的各种用法，事实上这样做是合理的。任何尺寸的任何程序都会包含相当数量的循环。而虽然它们不都是完全一样的，但也不是完全不同的；特定编码模式会暴露出来，尤其当你包含了紧跟循环前后执行的代码时——为循环做准备的模式，确保循环过程所需的模式，以及当循环结束时所需操作的模式。LOOP 语言捕捉了这些模式从而让你可以直接地表达各种循环。

LOOP 宏由许多部分组成——LOOP 反对者们的主要论点就是说它过于复杂了。在本章里，我将从头介绍 LOOP，给你一个关于它的多个组成部分以及彼此间配合使用方式的系统化介绍。

22.1 LOOP 的组成部分

你可以在一个 LOOP 中做到下列事情：

- 以数值或多种数据结构为步长来做循环；
- 在循环的过程中收集、计数、求和、求最大或最小值；
- 执行任意 Lisp 表达式；
- 决定何时终止循环；
- 条件执行上述内容。

另外，LOOP 还提供了用于下列事务的语法：

- 创建用于循环内部的局部变量；
- 指定任意 Lisp 表达式在循环开始前和结束后运行。

一个 LOOP 的基本结构是一个子句 (clause) 集合，其中每个子句都以一个循环关键字 (loop keyword) 开始。¹ 每个子句被 LOOP 宏解析的方式取决于具体的关键字。你在第 7 章里已经看到过一些主要的关键字，包括 for、collecting、summing、counting、do 以及 finally。

¹ 术语“循环关键字”取的并不是很好，因为循环关键字并不是正常意义上的 KEYWORD 包中的关键字。事实上，来自任何包的任何符号，只要有适当的名字就可以了；LOOP 宏只关心它们的名字。不过，通常情况下它们都被写成不带有包限定符的形式并在当前包下被读取（必要时会创建新符号）。

22.2 迭代控制

大多数所谓的迭代控制子句都以循环关键字 `for` 开始，或是它的同义词 `as`²，后接一个变量的名字。变量名后面的内容取决于 `for` 子句的类型。

一个 `for` 子句的下级子句可以在下列内容上进行迭代：

- 数字范围，以指定的间隔向上或向下；
- 由单独的项所组成的列表；
- 构成列表的点对单元；
- 向量的元素，包括诸如字符串和位向量这样的向量子类型在内；
- 哈希表的键值对；
- 一个包中的符号；
- 反复求值给定形式得到的结果。

单一循环可以含有多个 `for` 子句，其中每个子句都可以命名其自己的循环变量。当一个循环含有多个 `for` 子句时，循环将在任何一个子句达到其结束条件时终止。例如，下面的循环：

```
(loop
  for item in list
  for i from 1 to 10
  do (something))
```

将迭代至多 10 次但却可以在 `list` 含有少于 10 项时提前终止。

22.3 计数型循环

算术迭代子句可以控制循环体的执行次数，它通过在一个整数范围上步进来到做到这点，每前进一步就执行一次循环体。这些子句由 1 到 3 个紧跟在 `for`（或 `as`）之后的介词短语构成：起始短语、终止短语，以及步长短语。

起始短语指定了该子句的变量初始值。它由介词 `from`、`downfrom` 或 `upfrom` 之一后接一个提供初值（一个数字）的形式所构成。

终止短语指定了循环的终止点。它由介词 `to`、`upto`、`below`、`downto` 或 `above` 之一后接一个提供终值的形式所构成。当使用 `upto` 和 `downto` 时，循环体将在变量通过终止点时终止（通过以后不会再次求值循环体）；而当使用 `below` 和 `above` 时，它会提前一次迭代将循环终止。

步长短语由一个介词 `by` 和一个形式所构成，后者必须求值到一个正的数值上。变量将按照该数量在每次迭代时步进（向上或向下，取决于其他短语）或是在缺省时每次步进一个单位。

你必须指定至少一个上述这些介词短语。缺省值是从零开始，每次迭代时增加一个单位，然后一直加下去，或者更有可能的是直到其他某个子句终止了循环。你可以通过添加适当的介词短

² 因为当初 LOOP 的目标之一就是允许循环表达式可以被写成类似英语的语法，所以许多关键字都有一些同义词，它们对于 LOOP 来说处理方法相同，但在不同的语境下可以更接近英语的语法习惯。

语来修改这些缺省值中的任何一个或全部。唯一需要注意的是如果你想要逐步递减的话，不存在默认的初始值，因此你必须要用 `from` 要么用 `downfrom` 来指定一个。因此，下面的形式：

```
(loop for i upto 10 collect i)
```

会收集到 11 个整数（从零到十），而下面这个形式的行为则是未定义的：

```
(loop for i downto -10 collect i) ; wrong
```

你需要代替写成这样：

```
(loop for i from 0 downto -10 collect i)
```

另外注意到由于 `LOOP` 是一个宏，它运行在编译期，它需要能够完全基于这些介词而不是一些形式的值来决定变量步进的方向，因为后者要直到运行期才会知道。因此，下面的形式：

```
(loop for i from 10 to 20 ...)
```

可以工作得很好，因为默认就是递增步进。但这个：

```
(loop for i from 20 to 10 ...)
```

将不会知道是从 20 向下数到 10。更糟糕的是，它不会给你报错——由于 `i` 已经大于 10 了，所以循环根本不会执行。代替地，你必须写成下面这样：

```
(loop for i from 20 downto 10 ...)
```

或是这样：

```
(loop for i downfrom 20 to 10 ...)
```

最后，如果你只是想要一个循环重复特定的次数，那么你可以将下列形式中的一个子句：

```
for i from 1 to number-form
```

替换成像下面这样的一个 `repeat` 子句：

```
repeat number-form
```

这两个子句在效果上是等价的，只是 `repeat` 子句没有创建一个显式的循环变量。

22.4 在集合和包上循环

用于在列表上迭代的 `for` 子句比算术子句更简单一些。它们只支持两种介词短语，`in` 和 `on`。

一个下列形式的短语：

```
for var in list-form
```

将在求值 `list-form` 所产生的列表的所有元素上步进变量 `var`。

```
(loop for i in (list 10 20 30 40) collect i) -> (10 20 30 40)
```

有时这个子句会在一个 `by` 短语的辅助下使用，后者指定了一个用来在列表中向下移动的函数。默认值是 `CDR`，但可以是任何接受一个列表并返回其子列表的函数。例如，你可以像下面这样收集一个列表中相隔的元素：

```
(loop for i in (list 10 20 30 40) by #'caddr collect i) -> (10 30)
```

on介词短语被用来在构成列表的点对单元上步进变量 var。

```
(loop for x on (list 10 20 30) collect x) -> ((10 20 30) (20 30) (30))
```

该短语也接受一个 by介词:

```
(loop for x on (list 10 20 30 40) by #'caddr collect x) -> ((10 20 30 40) (30 40))
```

在一个向量（包括字符串和位向量）的元素上循环和列表上的循环类似，除了使用介词 across来代替 in。³例如：

```
(loop for x across "abcd" collect x) -> (#\a #\b #\c #\d)
```

在一个哈希表或包上迭代稍微复杂一些，因为哈希表和包含有你可能想要迭代的不同的值集合——哈希表中的键或值，以及一个包中的不同类型的符号。两种迭代都遵循了相同的模式。基本的模式看起来像下面这样：

```
(loop for var being the things in hash-or-package ...)
```

对于哈希表来说，*things*的可能值是 hash-keys 和 hash-values，其使得 var被绑定在哈希表的键或值的后继值上。hash-or-package形式只被求值一次用来产生一个值，其必须是一个哈希表。

要想迭代在一个包上，*things*可以是 symbols、present-symbols 和 external-symbols，其使得 var被绑定在一个包的每个可访问的符号上、一个包的每个当前符号上（换句话说，在包里创建的或导入进该包的），或是每一个从该包中导出的符号。hash-or-package被求值用来产生一个包的名字（这会导致用 FIND-PACKAGE来查找）或包对象本身。同义词对于 for子句的许多部分也是可用的。在 the的位置上你可以用 each；你可以使用 of来代替 in；另外你也可以将 things写成单数形式（例如，hash-key或 symbol）。

最后，由于你可能经常会在一个哈希表上同时迭代键和值，哈希表子句还在其结尾处支持一个 using子句。

```
(loop for k being the hash-keys in h using (hash-value v) ...)
(loop for v being the hash-values in h using (hash-key k) ...)
```

这两个循环都可以将 k绑定到哈希表的每个键上，再把 v绑定到对应的值上。注意到 using子句的第一个元素必须写成单数形式。⁴

22.5 等价-然后 (equals-then) 迭代

如果其他的 for子句都无法确切支持你所需要的变量步进形式，那么你可以通过一个等价-然后 (equals-then) 子句来完全控制步进的方式。这个子句跟一个 do循环中的绑定语句很相似，

³ 你可能想知道为什么 LOOP 不使用同样的介词然后自己检查当前究竟是循环在列表还是向量上。这是 LOOP 作为一个宏所带来的另一个后果：列表或者向量的值直到运行期才会知道，但 LOOP 作为一个宏必须在编译期生成代码。并且 LOOP 的设计者们想要它生成极其高效的代码。为了能够生成高效的代码用来访问，比如说一个向量，它需要在编译期知道这个值在运行期将是一个向量——因此，需要采用不同的介词。

⁴ 不要问我为什么 LOOP 的作者们没有使用不带括号的风格来表达 using子句。

但更接近 Algol 类语言的语法。完整的形式如下所示：

```
(loop for var = initial-value-form [ then step-form ] ...)
```

和通常一样，*var* 是需要步进的变量名。它的初值在首次迭代之前通过求值 *initial-value-form* 一次而获取到。在每一次后续迭代中，*step-form* 被求值，然后它的值成为了 *var* 的新值。如果子句中没有 *then* 部分，那么 *initial-value-form* 将在每次迭代中重新求值以提供新值。注意到这和一个没有步长形式的 *do* 绑定子句的行为是不同的。

step-form 可以引用其他的循环变量，包括由循环中其他后续 *for* 子句所创建的那些。例如：

```
(loop repeat 5
      for x = 0 then y
      for y = 1 then (+ x y)
      collect y) -> (1 2 4 8 16)
```

不过，注意到每个 *for* 子句都是以它们各自出现的顺序来逐个求值的。因此在前面的这个循环中，在第二次迭代时 *x* 会在 *y* 改变（换句话说，变成 1）之前被设置为 *y* 的值。但是 *y* 随后会被设置为他的旧值（仍然是 1）与 *x* 的新值之和。如果 *for* 子句的顺序反过来，那么结果将会改变。

```
(loop repeat 5
      for y = 1 then (+ x y)
      for x = 0 then y
      collect y) -> (1 1 2 3 5)
```

不过，通常你都想要多个变量的步长形式在任何变量被赋予新值之前计算完毕（类似于 *do* 步进其变量的方式）。在这种情况下，你可以将多个 *for* 子句连在一起，将除第一个以外的 *for* 全部替换成 *and*。你已经在第 7 章的 Fibonacci 计算的 LOOP 版本里见过它的公式了。这里是另一个变种，基于前面两个例子的：

```
(loop repeat 5
      for x = 0 then y
      and y = 1 then (+ x y)
      collect y) -> (1 1 2 3 5)
```

22.6 局部变量

尽管一个循环所需要的主要变量通常都会隐式声明在 *for* 子句中，但有时你也将需要额外的变量，这可以通过使用 *with* 子句来声明。

```
with var [= value-form]
```

名字 *var* 将成为一个局部变量的名字，它会在循环结束时被删除。如果 *with* 子句含有一个 = *value-form* 部分，那么变量将会在循环的首次迭代之前初始化到 *value-form* 的值上。

多个 *with* 子句可以同时出现在一个循环里；每个子句将根据其出现的顺序独立地求值，并且其值将在处理下一个子句之前完成赋值，从而允许后面的变量可以依赖于已经声明过的变量。完全无关的变量可以通过用 *and* 连接每个声明而在一个 *with* 子句中进行声明。

22.7 解构变量

一个尚未谈及的 LOOP宏的非常有用特性是其解构那些赋值给循环变量的列表值的能力。这可以让你取出赋值给循环变量的列表里的值，类似于 DESTRUCTURING-BIND的工作方式但没有那么复杂。基本上，你可以将任何出现在一个 for或 with子句中的循环变量替换成一个符号树，然后原本赋值到简单变量上的列表值将改为解构到以树中的符号所命名的变量上。一个简单的例子看起来像下面这样：

```
CL-USER> (loop for (a b) in '((1 2) (3 4) (5 6))
      do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

这棵树还可以包含带点的列表，这时点之后的名字将像一个 &rest参数那样处理，被绑定到列表的其余元素上。这对于 for/on类循环来说特别有用，因为值总是一个列表。例如，下面这个循环（我曾经在第 8 章里用它来输出一个逗号分隔的列表）：

```
(loop for cons on list
      do (format t "~a" (car cons))
      when (cdr cons) do (format t ", "))
```

也可以写成下面这样：

```
(loop for (item . rest) on list
      do (format t "~a" item)
      when rest do (format t ", "))
```

如果你想要忽略一个解构列表中的值，那么你可以在一个变量的名字上使用 NIL。

```
(loop for (a nil) in '((1 2) (3 4) (5 6)) collect a) →(1 3 5)
```

如果解构列表含有比列表中的值更多的变量，那么多余的变量将被设置为 NIL，这使得所有变量本质上都像是 &optional参数。不过，没有任何跟 &key参数等价的东西。

22.8 值汇聚

值汇聚 (value accumulation) 语句可能是 LOOP中最有用的部分。尽管迭代控制语句提供了一个表达基本循环的基本手法，但它们本质上与 DO、DOLIST和 DOTIMES所提供的手法是等价的。

另一方面，值汇聚子句提供了一套在循环过程中用于值汇聚的常用循环用法的简洁表示法。每个汇聚子句都以一个动词后接下列模式开始：

```
verb form [ into var ]
```

每次通过循环时，一个汇聚子句会求值 form并将其按照 verb所决定的方法保存起来。通过一个 into下级子句，这些值被保存在名为 var的变量里。该变量对于循环来说是局部的，就像它是被一个 with子句所声明的那样。如果没有 into下级子句，那么汇聚子句将汇聚出一个作为

整个循环表达式返回值的缺省值。

可能的动词 (verb) 包括 `collect`、`append`、`nconc`、`count`、`sum`、`maximize` 和 `minimize`。还有它们对应进行时形式的同义词：`collecting`、`appending`、`nconcing`、`counting`、`summing`、`maximizing` 和 `minimizing`。

`collect` 子句可以构造一个列表，其包含以它所看到的顺序排列的所有 *form* 的值。这是一个特别有用的构造，因为如果你想手工编写一个像 LOOP 那样有效率的列表收集代码将会非常困难。⁵ 与 `collect` 相关的动词是 `append` 和 `nconc`。这两个动词都将值汇聚到一个列表上，但它们所汇聚的值本身也必须是列表，然后像函数 `APPEND` 或 `NCONC`⁶ 那样将所有列表汇聚成单个列表。

其余的汇聚子句都用来汇聚数值。动词 `count` 统计 *form* 为真的次数，`sum` 收集所有 *form* 的值之和，`maximize` 收集它所看到的 *form* 的最大值，而 `minimize` 则收集最小值。例如，假设你定义了一个变量 `*random*`，其含有一个随机数的列表。

```
(defparameter *random* (loop repeat 100 collect (random 10000)))
```

那么下面的循环将返回关于这些数的一个含有多种统计信息的列表：

```
(loop for i in *random*
      counting (evenp i) into evens
      counting (oddp i) into odds
      summing i into total
      maximizing i into max
      minimizing i into min
      finally (return (list min max total evens odds)))
```

22.9 无条件执行

虽然值汇聚构造是非常有用的，但如果有机会在循环体中执行任意代码的话 LOOP 就不是一个很好的通用迭代机制了。

在一个循环之内执行任意代码的最简单方式是通过一个 `do` 子句。与目前我所讨论过的那些

⁵ 难点在于必须跟踪列表的尾部并通过 `SETF` 其尾部的 CDR 来向列表添加新的点对。一个由 `(loop for i upto 10 collect i)` 所生成代码的手写等价版本将看起来像这样：

```
(do ((list nil) (tail nil) (i 0 (1+ i)))
    ((> i 10) list)
    (let ((new (cons i nil)))
      (if (null list)
          (setf list new)
          (setf (cdr tail) new))
      (setf tail new)))
```

当然，你很少需要编写像这样的代码。你将要么使用 LOOP，要么（如果出于某种原因你不想使用 LOOP 的话）使用标准的收集值的 `PUSH/NREVERSE` 习惯用法。

⁶ 回顾一下，`NCONC` 是 `APPEND` 的破坏性版本——安全使用 `nconc` 子句的场合仅限于你正在收集的值都是全新的列表而没有跟其他列表共享任何结构。例如，下面这个是安全的：

```
(loop for i upto 3 nconc (list i i)) -> (0 0 1 1 2 2 3 3)
```

而下面这个将给你带来麻烦：

```
(loop for i on (list 1 2 3) nconc i) -> undefined
```

后者将很可能进入一个无限循环，因为由 `(list 1 2 3)` 所产生的列表被破坏性地修改以指向了它自身。但即便这个说法也难以保证——其行为根本没有定义。

带有介词和进一步子句的其他子句相比，`do`具有一种 Yoda 式的简洁性。⁷一个 `do`子句有单词 `do`（或 `doing`）后接一个或多个 Lisp 形式所构成，这些形式将在 `do`子句开始运行时全部被求值。`do`子句结束于一个循环的闭合括号或是下一个循环关键字。

例如，为了打印出从 1 到 10 的数字，你可以写成这样：

```
(loop for i from 1 to 10 do (print i))
```

另一个更有趣的立即执行的形式是一个 `return`子句。这个子句由单词 `return`后接单个 Lisp 形式所组成，当该形式被求值时，其得到的结果将立即作为整个循环的值返回。

你也可以使用任何 Lisp 的常规控制流操作符（例如 `RETURN`和 `RETURN-FROM`）来从一个循环中的 `do`子句里跳出。注意一个 `return`子句总是从临近的 `LOOP`表达式里返回，而 `do`子句中的 `RETURN`或 `RETURN-FROM`可以从任意封闭的表达式中返回。举个例子，比较下列这个表达式：

```
(block outer
  (loop for i from 0 return 100) ; 100 returned from LOOP
  (print "This will print")
  200) → 200
```

和下面这个：

```
(block outer
  (loop for i from 0 do (return-from outer 100)) ; 100 returned from BLOCK
  (print "This won't print")
  200) → 100
```

上述 `do`和 `return`子句被统称为无条件执行子句。

22.10 条件执行

由于一个 `do`子句可以包含任意 Lisp 形式，所以你可以使用任何你想要的 Lisp 表达式，包括诸如 `IF`和 `WHEN`这样的控制构造。这样，下面就是编写一个只打印 1 到 10 之间所有偶数的循环的一种写法：

```
(loop for i from 1 to 10 do (when (evenp i) (print i)))
```

不过，有时你将需要循环子句层面上的条件控制。例如，假设你只想用一个 `summing`子句来求和从 1 到 10 之间的偶数。你不可能用一个 `do`子句来写出这样的循环来，因为没有办法在一个正规的 Lisp 形式里“调用” `sum i`。对于类似这种情况，你就需要用到 `LOOP`自己的条件表达式，如下所示：

```
(loop for i from 1 to 10 when (evenp i) sum i) -> 30
```

`LOOP`提供了 3 种条件构造，它们全部遵循下面的基本模式：

conditional test-form loop-clause

其中的 `conditional`可以是 `if`、`when`或 `unless`。其中的 `test-form`可以是任何正规 Lisp

⁷ “No! Try no. Do ... or do not. There is no try.”（不，不要试。要么做要么不做。没有机会可试）——Yoda, The Empire Strikes Back (星球大战 5)。

形式，而 *loop-clause* 则可以是一个值汇聚子句 (`count`、`collect`，诸如此类)、一个无条件执行子句，或是另一个条件执行子句。多个循环子句可以通过 `and` 连接成单一条件。

还有一点儿额外的语法糖，在第一个循环子句里，测试形式之后，你可以使用变量 `it` 来指代由测试形式所返回的值。例如，下面的循环可以收集那些在列表 `some-list` 中查找键时所找到的在哈希表 `some-hash` 中对应的非空值：

```
(loop for key in some-list when (gethash key some-hash) collect it)
```

条件子句在每次通过循环时都会执行。一个 `if` 或 `when` 子句会在它的 `test-form` 求值为真时执行其 *loop-clause*。一个 `unless` 子句可以把测试反过来，仅当 `test-form` 为 `NIL` 时才执行 *loop-clause*。`LOOP` 的 `if` 和 `when` 关键字与他们对应的 Common Lisp 命名含义有所不同的是，它们是同义词——在行为上没有区别。

下面这个相当傻的循环演示了几种不同形式的 `LOOP` 条件子句。函数 `update-analysis` 将在每次通过循环时被调用，其参数是几个变量由条件子句中的不同变量汇聚子句最后更新的值。

```
(loop for i from 1 to 100
      if (evenp i)
          minimize i into min-even and
          maximize i into max-even and
          unless (zerop (mod i 4))
              sum i into even-not-fours-total
          end
          and sum i into even-total
      else
          minimize i into min-odd and
          maximize i into max-odd and
          when (zerop (mod i 5))
              sum i into fives-total
          end
          and sum i into odd-total
      do (update-analysis min-even
                           max-even
                           min-odd
                           max-odd
                           even-total
                           odd-total
                           fives-total
                           even-not-fours-total))
```

22.11 设置和拆除

`LOOP` 语言设计者们关于实际使用中的循环的一项关键的远见卓识是：它们看到一个循环总是以一些设置初始环境的代码开始，循环本身结束后还要有更多的代码来处理由循环所计算出来的那些值。举一个简单的 Perl⁸ 例子，如下所示：

```
my $evens_sum = 0;
my $odds_sum = 0;
foreach my $i (@list_of_numbers) {
    if ($i % 2) {
        $odds_sum += $i;
```

⁸ 我并不是故意选择 Perl 的——这个例子在任何语法基于 C 语言的语言里看起来都是差不多的。

```

    } else {
        $evens_sum += $i;
    }
}
if ($evens_sum > $odds_sum) {
    print "Sum of evens greater\n";
} else {
    print "Sum of odds greater\n";
}

```

这段代码中的循环是 `foreach`语句。但 `foreach`本身并不能独立工作：循环体中的代码引用了循环开始前的两行代码中声明的变量。⁹而循环所做的所有工作假如没有了后面那个 `if` 语句的话也就毫无意义了，后者在循环结束后汇报结果。当然，在 Common Lisp 中 `LOOP`结构也是一个可以返回值的表达式，因此通常更需要做的一件事是在循环结束之后生成一个有用返回值。

所以，`LOOP`的设计者们说，应该提供一种方式可以将原本属于循环的一部分的那些代码也塞进循环代码本身。这样，`LOOP`就提供了两个关键字，`initially`和`finally`，它们引入了那些原本可以运行在循环主体以外的代码。

在 `initially`或 `finally`之后，这些子句由所有需要在下一个循环子句开始之前或者循环结束之后运行的多个 Lisp 形式所组成。所有的 `initially`形式会被组合成单个的“序言”部分，在所有局部循环变量被初始化以后和循环体开始之前运行一次。所有的 `finally` 形式则被简单地组合成一个“尾声”部分，在最后一次循环体的迭代结束以后运行。

序言部分总是会运行，就算循环迭代了零次。但是循环有可能在下列任何情况发生时没有运行尾声部分：

- 执行了一个 `return`子句。
- `RETURN`、`RETURN-FROM`或其他控制构造的传递操作在循环体中的一个 Lisp 形式中被调用¹⁰。
- 循环被一个 `always`、`never`或 `thereis`子句所终止，我将在下一节里讨论这种情况。

在尾声部分的代码中，`RETURN`或 `RETURN-FROM`可被用来显式提供一个循环的返回值。这样一个显式的返回将比任何其他的那些由汇聚或终止测试子句所提供的值具有更高的优先级。

另外，为了允许 `RETURN-FROM`被用来从一个特定的循环中返回（这在嵌套的 `LOOP`表达式中是有用的），你可以使用循环关键字 `named`为 `LOOP`命名。如果一个 `named`子句出现在一个循环中，那么它必须是第一个子句。举一个简单的例子，假设 `lists`是一个列表的列表，而你想要在这些嵌套的列表中查到匹配某些特征的项。你可以像下面这样使用一对嵌套的循环来找到它：

```
(loop named outer for list in lists do
      (loop for item in list do
            (if (what-i-am-looking-for-p item)
                (return-from outer item))))
```

⁹ 在 Perl 里，如果你没有使用 `use strict` 的话，Perl 将允许你随意使用未经声明的变量。但你应当总是在 Perl 中使用 `use strict`。Python、Java 或 C 中的等价代码将总是会要求声明变量。

¹⁰ 你可以使用局部宏 `LOOP-FINISH` 让整个循环从循环体中的某段 Lisp 代码中直接正常返回，同时有机会执行循环的尾声部分。

22.12 终止测试

尽管 `for` 和 `repeat` 子句提供了控制循环次数的基本手法，但有时你将需要更早的中断循环。你已经知道一个 `do` 子句里的 `return` 子句、`RETURN` 或 `RETURN-FROM` 形式可以立即终止循环；但正如存在一些用来汇聚值的通用模式那样，也存在用来决定何时终止循环的通用模式。这些模式在 `LOOP` 中是由终止子句 `while`、`until`、`always`、`never` 和 `thereis` 来提供的。它们全都遵循相同的模式：

loop-keyword test-form

五种子句都会在每次通过迭代时求值 `test-form`，然后基于所得到的值来决定是否终止循环。它们的区别在于如果终止循环的话需要什么条件，以及如何决定这点。

循环关键字 `while` 和 `until` 代表了“温和”的终止子句。当它们决定终止循环时，控制会传递到尾声部分，并跳过循环体的其余部分。尾声部分随后会返回一个值或是做任何想要的事情来结束循环。`while` 子句在测试形式首次为假时终止循环；相反地，`until` 子句在测试形式首次为真时停止。

另一个温和的终止形式是由 `LOOP-FINISH` 宏所提供的。这是一个正规 Lisp 形式，并非一个循环子句，因此它可以用在一个 `do` 子句的 Lisp 形式中的任何地方。它也会导致立即跳转到循环的尾声部分。这在是否跳出循环的判断难以用一个简单的 `while` 或 `until` 子句来表达时是有用的。

另外三个子句——`always`、`never` 和 `thereis`——采用极端偏执的方式来终止循环；它们立即从循环中返回，不但跳过任何手续的循环子句而且还包括尾声部分。它们还为整个循环提供了缺省的返回值，哪怕是它们没有导致循环终止。尽管如此，如果循环不是因为这些终止测试中的一个而终止的，那么尾声部分还是有机会运行并返回一个值以代替终止子句所提供的缺省值的。

由于这些子句提供了它们自己的返回值，因此它们不能跟汇聚类子句配合使用，除非汇聚子句带有一个 `into` 下级子句。编译器（或解释器）应当在编译期汇报此类错误。`always` 和 `never` 子句仅返回布尔值，因此它们在你需要用一个循环表达式来作为谓词的一部分时将是最有用的。你可以使用 `always` 来确认循环的每次迭代过程中测试形式均为真。相反地，`never` 测试每次迭代中测试行为均为假。如果测试形式失败了（在 `always` 子句中返回 `NIL` 或是在 `never` 子句中返回非 `NIL`），那么循环将被立即终止，并返回 `NIL`。如果循环可以一直运行直到完成，那么缺省值 `T` 被提供。

举个例子，如果你想要测试一个列表 `numbers` 中的所有数都是偶数，那么你可以写成下面这样：

```
(if (loop for n in numbers always (evenp n))
      (print "All numbers even."))
```

下面是等价的另一种写法：

```
(if (loop for n in numbers never (oddp n))
      (print "All numbers even."))
```

`thereis` 子句被用来测试是否测试形式“曾经”为真。一旦测试形式返回了一个非 `NIL` 的值，

那么循环就会终止并返回该值。如果循环得以运行到完成，那么 `thereis` 子句提供了缺省值 `NIL`。

```
(loop for char across "abc123" thereis (digit-char-p char)) → 1  
(loop for char across "abcdef" thereis (digit-char-p char)) → NIL
```

22.13 把所有东西放在一起

现在你已经看到了 `LOOP` 功能的所有主要特性。只要你遵循下列规则就可以将我所讨论过的任何子句组合在一起：

- 如果有 `named` 子句的话，它必须是第一个子句。
- 在 `named` 子句的后面是所有的 `initially`、`with`、`for` 和 `repeat` 子句。
- 然后是主体子句：有条件和无条件的执行、汇聚和终止测试。¹¹
- 以任何 `finally` 子句结束。

`LOOP` 宏将展开成完成下列操作的代码：

- 初始化所有由 `with` 或 `for` 子句所声明的局部变量，以及由汇聚子句所创建的隐含局部变量。提供初始值的形式按照它们在循环中出现的顺序进行求值。
- 执行由任何 `initially` 子句——序言部分——所提供的形式，以它们出现在循环中的顺序来执行。
- 迭代，同时按照下面一段文字所描述的过程来执行循环体的代码。
- 执行由任何 `finally` 子句——尾声部分——所提供的形式，以它们出现在循环中的顺序来执行。

当循环在迭代时，循环体被执行的方式是首先步进那些迭代控制变量，然后以出现在循环中的顺序执行任何有条件或无条件的执行、汇聚或终止测试子句。如果循环中的任何子句终止了循环，那么循环体的其余部分将被跳过，然后整个循环在可能运行了尾声部分以后返回。

这基本上就是所有的内容了。¹² 你将在本书后面的代码中频繁用到 `LOOP`，因为有必要对他做些了解。除此之外，用不用它就完全取决于你了。

有了这些基础，现在你可以开始进入作为本书其余部分的实践性章节了——首先是编写一个垃圾过滤器。

¹¹ 一些 Common Lisp 实现将允许你交替使用主体子句和 `for` 子句，但这在严格来讲是未定义的，并且另一些实现会拒绝这样的循环。

¹² 关于 `LOOP`，我尚未讨论过的一个方面是用来声明循环变量类型的语法。当然，我也还没有讨论过 `LOOP` 之外的类型声明。我将在第 32 章里谈及这个一般主题。对于它们与 `LOOP` 配合使用的细节，请参考你所喜爱的 Common Lisp 手册。

第23章 实践：一个垃圾过滤器

在 2002 年，Paul Graham 在他把 Viaweb 卖给 Yahoo 之后腾出一些时间写了一个关于《一个处理垃圾邮件的计划》的随笔¹，这导致了垃圾过滤技术的一个小的革命。在 Graham 的文章发表之前，多数垃圾过滤器都采用手工编写的规则来编写：如果一个消息在标题中带有 XXX，那么他可能是一个垃圾邮件；如果一个消息在一行中有三个或更多的词是全部大写的，那么它可能是一个垃圾邮件。Graham 花了几个月时间试图编写这样一个基于规则的过滤器并最终意识到这基本上是一个令人痛苦的任务。

为了识别个别的垃圾特性，你必须站在垃圾发送者的角度来思考，而坦白地说我想在这件事上花尽可能少的时间。

为了避免站在垃圾发送者的角度，Graham 决定尝试从非垃圾中区分垃圾，通过统计哪些词出现在哪一类 Email 中来做到这点。这个过滤器将持续跟踪特定单词出现在垃圾和正常信息中的频繁程度，然后在一个新的消息中通过使用与单词相关联的频率数据来计算出该消息是垃圾或正常信息的可能性。他把他的方法称为贝叶斯（Bayesian）过滤，他使用的这种统计技术可以将单独的词汇频率组合成一个整体可能性。²

23.1 垃圾过滤器的心脏

在本章里，你将实现一个垃圾过滤引擎的核心。你将不会编写一个完整的垃圾过滤应用程序；相反，你将把精力集中在对新消息进行分类以及训练过滤器等功能上。

这个应用将会写得足够大，因此有必要定义一个新的包来避免名字冲突。例如，在你从本书的 Web 站点可以下载到的源代码中，我使用了包名 COM.GIGAMONKEYS.SPAM 定义出一个同时用到标准 COMMON-LISP 包和来自第 15 章的 COM.GIGAMONKEYS.PATHNAMES 包的新包，像下面这样：

```
(defpackage :com.gigamonkeys.spam
  (:use :common-lisp :com.gigamonkeys.pathnames))
```

任何含有这个应用程序代码的文件都应当以下面这行开始：

```
(in-package :com.gigamonkeys.spam)
```

¹ 此文可从 <http://www.paulgraham.com/spam.html> 获得，也包含在《Hackers & Painters:Big Ideas from the Computer Age》(O'Reilly, 2004)

² 关于 Graham 所描述的技术是否真的是“贝叶斯”一直以来有些不同的看法。不过这个名字已经广为流传并已经成为谈论垃圾过滤时“统计”的代名词。

你可以使用相同的包名或者将 `com.gigamonkeys` 替换成你所控制的某个域。³

你还可以通过在 REPL 中输入相同的形式来切换到这个包从而测试你所编写的函数。在 SLIME 中这将使提示符从 `CL-USER>` 变成 `SPAM>`，像下面这样：

```
CL-USER> (in-package :com.gigamonkeys.spam)
#<The COM.GIGAMONKEYS.SPAM package>
SPAM>
```

一旦你定义了一个包，那么就可以开始实际的编码工作了。你将需要实现的主函数有一个简单的任务——接受一个消息的文本作为参数并将该消息分类成垃圾、有用信息或不确定。你可以简单地实现这个基本函数，通过使用你将在后面编写的其他函数来定义它。

```
(defun classify (text)
  (classification (score (extract-features text))))
```

从里向外读这个函数，分类一个消息的第一步是从文本中解出传递给 `score` 函数的那些特性。在 `score` 中你将计算出一个值，该值随后通过函数 `classification` 翻译成三个分类——垃圾、有用信息或不确定——之一。在这三个函数中，`classification` 是最简单的。你可以假设 `score` 将在消息为垃圾时返回一个接近 1 的值，当它是正常信息时返回接近 0 的值，而在不确定时返回接近 0.5 的值。

因此你可以像下面这样实现 `classification`：

```
(defparameter *max-ham-score* .4)
(defparameter *min-spam-score* .6)

(defun classification (score)
  (cond
    ((<= score *max-ham-score*) 'ham)
    ((>= score *min-spam-score*) 'spam)
    (t 'unsure)))
```

函数 `extract-features` 也几乎是相当直接的，尽管它需要更多一些的代码。目前你所解出的特性将是出现在文本中的单词。对于每个单词，你需要跟踪它们在一个垃圾中出现的次数以及在正常信息中出现的次数。一种将这些数据与单词本身保存在一起的便利方式是定义一个类 `word-feature`，它带有三个槽。

```
(defclass word-feature ()
  ((word
    :initarg :word
    :accessor word
    :initform (error "Must supply :word")
    :documentation "The word this feature represents.")
   (spam-count
    :initarg :spam-count
    :accessor spam-count
    :initform 0
    :documentation "Number of spams we have seen this feature in.")
   (ham-count
    :initarg :ham-count
    :accessor ham-count
    :initform 0))
```

³ 尽管如此，并不推荐你使用一个以 `com.gigamonkeys` 开头的包来分发该应用的一个版本，因为你并不控制那个域。

```
:documentation "Number of hams we have seen this feature in.")))
```

你将把所有特性的数据库保存在一个哈希表中从而可以方便地查找代表一个给定特性的对象。你可以定义一个特殊变量 `*feature-database*` 来保存对这个哈希表的引用。

```
(defvar *feature-database* (make-hash-table :test #'equal))
```

你应当使用 `DEFVAR` 而不是 `DEFPARAMETER` 来定义它，因为你不希望 `*feature-database*` 在你开发过程中重新加载了含有该定义的文件后被重置——你可能不想丢失那些保存在 `*feature-database*` 中的数据。当然，这意味着如果你确实想要清空这个特性数据库，那么你不能只是重新求值那个 `DEFVAR` 形式。所以你应该定义一个函数 `clear-database`。

```
(defun clear-database ()
  (setf *feature-database* (make-hash-table :test #'equal)))
```

为了查找一个给定消息中的特性，代码将需要解出单独的词然后在 `*feature-database*` 中查找对应的 `word-feature` 对象。如果 `*feature-database*` 不含有这个特性，那么它将创建出一个代表该词的新的 `word-feature`。你可以将这些逻辑封装在一个函数 `intern-feature` 中，它接受一个单词并返回对应的特性，在必要时会创建它。

```
(defun intern-feature (word)
  (or (gethash word *feature-database*)
      (setf (gethash word *feature-database*)
            (make-instance 'word-feature :word word))))
```

你可以使用一个正则表达式从消息文本中解出单个的词。例如，使用 Edi Weitz 编写的 Common Lisp 可移植 Perl 兼容正则表达式 (CL-PPCRE) 库，你可以像这样编写 `extract-words`:⁴

```
(defun extract-words (text)
  (delete-duplicates
   (cl-ppcre:all-matches-as-strings "[a-zA-Z]{3,}" text)
   :test #'string=))
```

现在为了实现 `extract-features` 所剩下的就是将 `extract-words` 和 `intern-feature` 放在一起。由于 `extract-words` 返回了一个字符串的列表而你想要的是一个列表其中每个字符串都被翻译成对应的 `word-feature` 对象，因此正好可以使用 `MAPCAR`。

```
(defun extract-features (text)
  (mapcar #'intern-feature (extract-words text)))
```

你可以像这样在 REPL 中测试这些函数：

```
SPAM> (extract-words "foo bar baz")
("foo" "bar" "baz")
```

同时你可以确保 `DELETE-DUPLICATES` 像下面这样工作：

```
SPAM> (extract-words "foo bar baz foo bar")
("baz" "foo" "bar")
```

你还可以测试 `extract-features`。

```
SPAM> (extract-features "foo bar baz foo bar")
```

⁴ CL-PPCRE 的一个版本被包含在本书的源代码中。或者你可以从 Weitz 位于 <http://www.weitz.de/cl-ppcre/> 的站点上下载。

```
(#<WORD-FEATURE @ #x71ef28da> #<WORD-FEATURE @ #x71e3809a>
 #<WORD-FEATURE @ #x71ef28aa>)
```

不过正如你所看到的，打印任意对象的默认方法说明性不足。对于你所编写的这个程序来说，如果可以更透明地打印出 `word-feature` 对象将是有用的。幸运的是，如同我在第 17 章里提到的，所有对象的打印是由广义函数 `PRINT-OBJECT` 来实现的，因此为了改变 `word-feature` 的打印方式，你只需定义一个特化在 `word-feature` 上的 `PRINT-OBJECT` 方法。为了让实现这样的方法更简单，Common Lisp 提供了一个宏 `PRINT-UNREADABLE-OBJECT`。⁵

`PRINT-UNREADABLE-OBJECT` 基本形式如下所示：

```
(print-unreadable-object (object stream-variable &key type identity)
  body-form*)
```

其中的 `object` 参数是一个求值到被打印对象的表达式。在 `PRINT-UNREADABLE-OBJECT` 主体中，`stream-variable` 被绑定到一个流，你可以向其中打印你想要的任何东西。你打印到该流中的任何东西都将被 `PRINT-UNREADABLE-OBJECT` 输出并封装在不可读对象的标准语法 `#<>` 中。⁶

`PRINT-UNREADABLE-OBJECT` 还可以通过关键字参数 `type` 和 `identity` 让你包含对象的类型和一个对象标识的指示。如果它们是非 `NIL` 的，那么输出将以对象类的名字开始并以对象的标识结束，这与 `STANDARD-OBJECT` 的默认 `PRINT-OBJECT` 方法所打印的形式相似。对于 `word-feature` 来说，你可能想要定义一个 `PRINT-OBJECT` 方法来包含其类型而没有标识并同时带有 `word`、`ham-count` 和 `spam-count` 等槽的值。这个方法将看起来像下面这样：

```
(defmethod print-object ((object word-feature) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (word ham-count spam-count) object
      (format stream "~S :hams ~D :spams ~D" word ham-count spam-count))))
```

现在当你在 REPL 中测试 `extract-features` 时，你可以更清楚地看到那些被解出的特性。

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE "baz" :hams 0 :spams 0>
 #<WORD-FEATURE "foo" :hams 0 :spams 0>
 #<WORD-FEATURE "bar" :hams 0 :spams 0>)
```

23.2 训练过滤器

现在你已经有了跟踪单独特性的方法，你几乎可以开始实现 `score` 了。但首先你需要编写用来训练垃圾过滤器的代码这样 `score` 才会有数据可用。你将定义一个函数 `train` 接受一些文本和一个指示消息类型——`ham` 或 `spam`——的符号，然后递增文本中出现的所有特性的 `ham` 或 `spam` 计数器，它们代表目前所处理过的有用信息或垃圾信息的全局计数。再一次，你可以采用至顶向下的方法用其他尚不存在的函数来实现它。

```
(defun train (text type)
```

⁵ 使用 `PRINT-UNREADABLE-OBJECT` 的主要原因是它会在某人试图可读地打印你的对象时负责报一个适当的错误，例如在使用 `FORMAT` 指令 `~S` 时。

⁶ `PRINT-UNREADABLE-OBJECT` 也会在打印控制变量 *`PRINT-READABLY`* 为真时报错。这样，一个完全由 `PRINT-UNREADABLE-OBJECT` 形式所组成的 `PRINT-OBJECT` 方法将正确实现遵守 *`PRINT-READABLY`* 协议的 `PRINT-OBJECT`。

```
(dolist (feature (extract-features text))
  (increment-count feature type))
(increment-total-count type))
```

你已经编写了 `extract-features`，因此下一个需要编写的是 `increment-count`，它接受一个 `word-feature` 和一个消息类型并递增该特性的相应槽。由于没有理由认为递增这些计数器的逻辑对于不同类型的对象会有所变化，因此你可以将它写成一个正规函数。⁷ 因为你将 `ham-count` 和 `spam-count` 都定义成带有一个 `:accessor` 选项，因此你可以使用 `INCF` 和 `DEFCLASS` 所创建的访问函数来递增相应的槽。

```
(defun increment-count (feature type)
  (ecase type
    (ham (incf (ham-count feature)))
    (spam (incf (spam-count feature)))))
```

其中的 `ECASE` 构造是 `CASE` 的一个变种，两者都类似于源自 Algol 的语言中的 `CASE` 语句（在 C 和它的后裔中重命名成 `switch` 了）。它们都求值其第一个参数——键形式，然后找出第一个元素——键——`EQL` 相等的子句。在本例中，这意味着当变量 `type` 被求值时，得到了作为 `increment-count` 的第二个参数所传递的值。

键不会被求值。换句话说，`type` 的值将与 Lisp 读取器作为 `ECASE` 形式一部分所读取的字面对象进行比较。在这个函数中，这意味着键是符号 `ham` 和 `spam`，而不是任何名为 `ham` 和 `spam` 的变量的值。因此，如果 `increment-count` 像这样被调用：

```
(increment-count some-feature 'ham)
```

那么 `type` 的值将是符号 `ham`，而 `ECASE` 的第一个分支将被求值并且对应特性的 `ham` 计数将会递增。另一方面，如果它像这样被调用：

```
(increment-count some-feature 'spam)
```

那么第二个分支将运行，从而递增了 `spam` 计数。注意符号 `ham` 和 `spam` 在调用 `increment-count` 时被引用了，因为否则它们就会作为变量的名字被求值。但是当它们出现在 `ECASE` 中时却不是引用的，因为 `ECASE` 并不求值它的键。⁸

`ECASE` 中的 `E` 代表“无遗漏的 (exhaustive)” 或“错误 (error)”，意味着当键值是任何列出的键之外的东西时 `ECASE` 应当报错。正常的 `CASE` 相对宽松，当没有匹配的子句被找到时返回 `NIL`。

为了实现 `increment-total-count`，你需要决定将计数保存在哪里；目前使用两个特殊变

⁷ 如果你以后决定需要为不同的类编写不同版本的 `increment-feature`，那么你可以将 `increment-count` 重定义成一个广义函数而将该函数定义成一个特化在 `word-feature` 上的方法。

⁸ 技术上来讲，一个 `CASE` 或 `ECASE` 的每个子句中的键都将被解释成一个列表指示符，一个指定了对象列表的对象。一个单一的非列表对象作为列表指示符对待时代表了一个只含有一个对象的列表，而一个列表将指代它本身。这样，每个子句可以有多个键；`CASE` 和 `ECASE` 将选择键列表中含有键形式的值的子句。例如，如果你想把 `good` 作为 `ham` 的同义词而把 `bad` 作为 `spam` 的同义词，那么你可以像下面这样来编写

```
increment-count:
(defun increment-count (feature type)
  (ecase type
    ((ham good) (incf (ham-count feature)))
    ((spam bad) (incf (spam-count feature)))))
```

量 `*total-spams*` 和 `*total-hams*` 将会做得很好。

```
(defvar *total-spams* 0)
(defvar *total-hams* 0)

(defun increment-total-count (type)
  (ecase type
    (ham (incf *total-hams*))
    (spam (incf *total-spams*)))))
```

你应当使用 `DEFVAR` 来定义这两个变量，所依据的理由与用在 `*feature-database*` 时相同——它们将在你运行程序期间始终保持其中的数据，你不会希望只是因为在开发过程中重新加载了你的代码就扔掉这些数据。但是你将希望在你重置了 `*feature-database*` 之后可以顺便重置这两个变量，因此你应当如下所示在 `clear-database` 添加几行：

```
(defun clear-database ()
  (setf
    *feature-database* (make-hash-table :test #'equal)
    *total-spams* 0
    *total-hams* 0))
```

23.3 每单词的统计

一个统计型的垃圾过滤器的心脏当然是那些计算基于统计的概率的函数。关于这些计算究竟为什么能够工作的数学原理⁹超出了本书的范围——有兴趣的读者可以参考 Gray Robinson 的几篇论文。¹⁰我将集中在它们是怎样被实现的。

统计计算的起始点是测量值的集合——保存在 `*feature-database*`、`*total-spams*` 和 `*total-hams*` 中的频率数据。假设用于训练的消息集合是统计上有代表性的，那么你可以将观察到的频率视为在未来消息中同样特性在有用信息和垃圾信息中出现的概率。

分类一条消息的基本计划是，解出消息中含有的特性，计算消息中含有的垃圾特性的单独概率，然后再将所有这些单独的概率合并成该消息的一个整体评分，带有许多垃圾特性和很少有用特性的消息将收到一个接近于 1 的评分，而带有许多有用特性和很少垃圾特性的消息将会评分接近 0。

你需要的第一个统计函数用来计算一个含有给定特性的消息是垃圾时的基本概率。从某种观点看来，一个含有该特性的给定消息是垃圾的概率就是含有该特性的垃圾消息与含有该特性的所

⁹ 从数学的角度来说，本章中对概率一词有时较宽松的用法可能会冒犯严肃的统计学家。不过，由于即便是该用法的赞成者们，其中还进一步划分成贝叶斯论者和频率论者，也无法对概率究竟是什么达成统一意见，因此我将不会担心这一点。毕竟这是一本关于编程而不是统计的书。

¹⁰ Robinson 的文章中直接引出本章的是“*A Statistical Approach to the Spam Problem*”（发表在 Linux Journal 上并可从 <http://www.linuxjournal.com/article.php?sid=6467> 获得并且一个简化版本发表在 Robinson 位于 <http://radio.weblogs.com/0101454/stories/2002/09/16/spamDetection.html> 的博客上）以及“*Why Chi? Motivations for the Use of Fisher's Inverse Chi-Square Procedure in Spam Classification*”（可从 <http://garyrob.blogs.com/whychi93.pdf> 获得）另一篇可能有用的文章是“*Handling Redundancy in Email Token Probabilities*”（可从 <http://garyrob.blogs.com//handlingtokenredundancy94.pdf> 获得）。SpamBayes 项目（<http://spambayes.sourceforge.net/>）的存档邮件列表里也含有关于测试垃圾过滤器的不同算法和思想的许多有用的信息。

有消息的比值。这样，你可以用下面的方式来计算它：

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (/ spam-count (+ spam-count ham-count))))
```

该函数所计算出的值的问题在于它被任何消息将是一个垃圾或有用信息的总体概率所影响。例如，假设你通常获得的正常信息是垃圾的九倍。那么一个完全正常的特性将在每九个正常信息后出现在一个垃圾信息里，从而根据这个函数计算出 1/10 的垃圾概率。

但是你更感兴趣的是一个给定特性将会出现在一个垃圾消息中的概率，与获得一个垃圾或正常消息的整体概率无关。这样，你需要将垃圾数量除以接受训练的垃圾总数，有用消息数量除以有用消息总数。为了避免除零错误，如果 *total-spams* 或 *total-hams* 两者任何一个为零，那么你应当将相应的频率视为零。（很明显，如果垃圾或有用消息的任一总数为零，那么相应的每特性计数也将是零，因此你可以将结果频率视为零而不会带来不良影响）

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (let ((spam-frequency (/ spam-count (max 1 *total-spams*)))
          (ham-frequency (/ ham-count (max 1 *total-hams*))))
      (/ spam-frequency (+ spam-frequency ham-frequency)))))
```

这个版本还有另一个问题——它没有在每单词概率上计入到达并分析的消息数量。假设你已经训练了 2000 条消息，一半垃圾一半正常。现在考察两个只出现在垃圾中的特性。一个出现在所有 1000 条消息中，而另一个仅出现一次。根据当前的 `spam-probability` 定义，两个特性的出现预测了一个消息是垃圾的概率是相等的，都是 1。

尽管如此，那个仅出现一次的特性很可能实际上是一个中性的特性——它很明显在无论垃圾还是有用信息中都很罕见，在 2000 条消息中仅出现一次。如果你训练了另外 2000 条消息，它很可能又出现了一次，这次出现在一条正常消息中，使得它突然成为了垃圾可能性为 0.5 的一条中性特性。

所以看起来你想要计算一个概率，它以某种方式影响了进入到每个特性中的数据点数。Robinson 在他的论文中推荐了一个基于贝叶斯概念的函数，将观察到的数据与先验知识或假设相合并。基本上，你以一个假设的先验概率开始计算一个新的概率并在添加新的信息之前给假设的概率一个权重。Robinson 的函数是下面这个：

```
(defun bayesian-spam-probability (feature &optional
                                      (assumed-probability 1/2)
                                      (weight 1))
  (let ((basic-probability (spam-probability feature))
        (data-points (+ (spam-count feature) (ham-count feature))))
    (/ (+ (* weight assumed-probability)
          (* data-points basic-probability))
        (+ weight data-points))))
```

Robinson 建议使用 1/2 作为 `assumed-probability` 的值以及 1 作为 `weight` 的值。使用这些值，一个出现在一条垃圾消息中而没有出现在有用消息中特性具有 0.75 的 `bayesian-spam-probability`，一个出现在 10 条垃圾消息而没有出现在有用消息中的特性具有大约 0.55 的 `bayesian-spam-probability`，而一条匹配了 1000 个垃圾消息却没有有用消息的特性将具有大约 0.9995 的垃圾概率。

23.4 合并概率

现在你可以计算在一条消息中所找到的每一个单独特性的 `bayesian-spam-probability`, 实现 `score` 函数的最后一步是找出一种方式将大量的概率个体合并成介于 0 和 1 之间的单个值。

如果单独的特性概率是彼此无关的, 那么从数学上来讲可以将它们相乘在一起得到一个合并的概率。但是它们实际上不可能是彼此无关的——特定的特性很可能会一起出现, 而其他一些却从不这样。¹¹

Robinson 提议使用由统计学家 R. A. Fisher 所发明的概率组合方法。在不讨论为什么它的技术可以工作的具体细节的前提下, 方法是这样的: 首先你通过将所有概率相乘将它们组合在一起。这给了你一个接近于 0 的远低于最初概率集合中概率的值。然后取该值的对数并乘以 -2。Fisher 在 1950 年展示, 如果这些单独的概率是彼此无关的并且来自 0 和 1 之间的统一分布。那么得到的值将满足卡方 (*chi-square*, χ^2) 分布。改值和概率数量的两倍可以输入到一个反向卡方分布函数中, 然后返回一个反映了通过组合同样数量的随机选择概率得到越来越大的值的可能性。当这个反向卡方分布函数返回一个较低的概率时, 这意味着在单独的概率中存在相当多的低概率。(要么是许多相对的低概率, 要么是少量非常低的概率)

为了使用这个概率来检测一个给定的消息是否是垃圾, 你从一个空假设 (*null hypothesis*) 开始, 一个你想要击倒的稻草人。这个空假设是被分类的消息事实上只是一个特性的随机集合。如果它是的话, 那么单独的概率——每个特性出现在一个垃圾消息中的可能性——也将是随机的。这就是说, 一个特性的随机选择将通常含有一些经常出现在垃圾中的特性和另一些很少出现在垃圾中的特性。如果你根据 Fisher 的方法合并这些随机选择的概率那么你将得到一个中间的合并值, 然后反向卡方分布函数将很可能返回一个比较高的值, 事实也是如此。但如果反向卡方分布返回了一个非常低的概率, 这意味着产生该合并值的那些概率不太可能是随机选择的; 可能这里面有太多的低概率值。因此你可以拒绝这个空假设而代替采纳另一个替代假设: 所有引入的特性来自一个有偏的样本——一个带有少量高垃圾概率特性和许多低垃圾特性的样本。换句话说, 它一定是条有用的消息。

尽管如此, Fisher 方法并不是对称的, 因为反向卡方分布函数对于给定数量的随机选择的概率所返回的合并后的概率将比你从合并实际概率中的值大得多。这种非对称性对你的用法有利, 因为当你拒绝空假设时你知道更好的假设是什么。当你通过 Fisher 方法合并单独的垃圾概率时, 而他告诉你有很高的概率表明空假设是错误的——消息并不是一个单词的随机集合——那么这意味着该消息很可能是有用的消息。所返回的数值就算并非该消息是有用消息的字面概率, 至少也是对它有用程度的一个好的衡量。相反的, 对于单独的有用概率的 Fisher 合并可以给你关于该消息垃圾程度的一个衡量。

为了得到一个最终的评分, 你需要将这两个指标合并成一个单一的值来给你一个范围从 0 到 1 的组合有用程度-垃圾程度评分。Rabinson 所推荐的方法是将有用程度和垃圾程度之间差异的一半与 $1/2$ 相加, 换句话说, 就是垃圾程度和 1 减去有用程度的平均值。这在两个评分相反(高

¹¹ 技术上来讲, 对一些事实上无关的概率进行非无关的概率合并, 这称为原生贝叶斯 (Naive Bayesian)。Graham 最初发表的本质上是一个原生贝叶斯分类器, 其中带有一些“经验驱动”的常量因子。

的垃圾程度和低的有用程度，或者反过来）时可以带来很好的效果，这时你将得到一个接近 0 或 1 的强烈的指示值。但是当垃圾程度和有用程度的评分都高或都低时，你将得到一个接近 1/2 的最终值，从而得到一个“不确定”的分类。

实现了这一模型的 `score` 函数看起来像下面这样：

```
(defun score (features)
  (let ((spam-probs ()) (ham-probs ()) (number-of-probs 0))
    (dolist (feature features)
      (unless (untrained-p feature)
        (let ((spam-prob (float (bayesian-spam-probability feature) 0.0d0)))
          (push spam-prob spam-probs)
          (push (- 1.0d0 spam-prob) ham-probs)
          (incf number-of-probs))))
      (let ((h (- 1 (fisher spam-probs number-of-probs)))
            (s (- 1 (fisher ham-probs number-of-probs))))
        (/ (+ (- 1 h) s) 2.0d0))))
```

你接受一个特性的列表并循环在它们之上，构建起两个概率的列表，一个列出含有每个特性的消息是垃圾的概率而另一个列出含有每个特性的消息是有用消息的概率。作为一项优化，你也可以在循环过程中统计概率的数量，然后将这个计数传给 `fisher` 来避免在 `fisher` 本身再次对它们计数。由 `fisher` 所返回的值当单独的概率中含有许多来自随机文本的低概率值时也将非常低。这样，一个低的 `fisher` 垃圾概率评分意味着存在许多有用的特性；将这个评价减去 1 就得到该消息是有用的消息的概率。相反从有用概率中减去 `fisher` 评分将得到该消息是垃圾的概率。将这两个概率组合在一起就可以给你一个介于 0 和 1 之间的整体垃圾程度评分。

在循环内部，你可以使用函数 `untrained-p` 来跳过那些从消息中解出的从未在训练中出现过的特性。这些特性将具有值为 0 的垃圾计数和有用计数。`untrained-p` 函数非常简单。

```
(defun untrained-p (feature)
  (with-slots (spam-count ham-count) feature
    (and (zerop spam-count) (zerop ham-count))))
```

剩下的唯一一个新的函数是 `fisher` 本身。假设你已经有了一个 `inverse-chi-square` 函数，那么 `fisher` 在概念上很简单。

```
(defun fisher (probs number-of-probs)
  "The Fisher computation described by Robinson."
  (inverse-chi-square
   (* -2 (log (reduce #'* probs)))
   (* 2 number-of-probs)))
```

不幸的是，在这个相当直接的实现中有一个小的问题。尽管使用 `REDUCE` 是一个将数字列表相乘的简洁方法，但在这个特定应用中乘积将会过小而无法表示成一个浮点数。在这种情况下，结果将会下溢到 0。并且因为概率的乘积下溢，所有努力都将白费，因为对 0 求 `LOG` 将要么报错，要么在某些实现中得到一个特殊的负无穷大值，这将使得所有后续的计算本质上都变成无意义的。这在本函数中尤其不幸，因为 `fisher` 方法当输入的概率值较低——接近 0——时最为敏感并且因此非常容易导致乘法下溢。

幸运的是，你可以使用一点高中数学来避免这个问题。一个乘法的对数等价于所有因数的对数之和，因此代替将所有概率相乘然后取对数，你可以将每个概率的对数相加。并且由于 `REDUCE` 接受一个 `:key` 关键字参数，你可以用它来完成整个计算。替换下面的写法：

```
(log (reduce #'* probs))
```

而写成下面这样：

```
(reduce #'+ probs :key #'log)
```

23.5 反向卡方分布函数

本节中的 `inverse-chi-square` 实现是 Rebinson 所写的一个 Python 版本的相当直接的转换。该函数的确切数学含义超出了本书的范围，但你可以通过思考你传递给 `fisher` 的值将怎样影响结果来得到一个关于其作用的直观印象：你传给 `fisher` 的低概率值越多，概率的乘积将会越小。一个小的乘积的对数将会是一个绝对值较大的负数。这样传递给 `fisher` 的低概率值越多，它传递给 `inverse-chi-square` 的值就越大。当然，引入的概率数量也会影响传递给 `inverse-chi-square` 的值。由于概率的定义是小于或等于 1 的，一个乘积中的概率越多，它的结果就将会越小并且传给 `inverse-chi-square` 的值越大。这样，`inverse-chi-square` 将在 `fisher` 合并值相比进入它的概率数量异乎寻常地大时返回一个较低的概率。下面的函数精确地做到了这点：

```
(defun inverse-chi-square (value degrees-of-freedom)
  (assert (evenp degrees-of-freedom))
  (min
   (loop with m = (/ value 2)
         for i below (/ degrees-of-freedom 2)
         for prob = (exp (- m)) then (* prob (/ m i))
         summing prob)
   1.0))
```

回忆第 10 章里 `EXP` 计算 e 的给定参数次方。这样，`value` 的值越大，`prob` 的初始值将会越小。但这个初始值随后将不断地被每个自由度微调，只要 `m` 大于自由度的数量。由于 `inverse-chi-square` 所返回的值因当是另一个概率，因此有必要用 `MIN` 来固定返回值，因为乘法和指数计算中的边界错误可能导致 `LOOP` 返回一个稍大于 1 的和。

23.6 训练过滤器

由于你编写 `classify` 和 `train` 来接受一个字符串参数，因此你可以轻松地在 REPL 中测试它们。如果你还没有这样做过，那么你应当通过在 REPL 中求值一个 IN-PACKAGE 形式或是使用 SLIME 的快捷命令 `change-package` 将当前包切换到你编写这些代码所在的包中。在输入包名的时候按 Tab 将会根据你的 Lisp 所知道的包来自动补全。现在你可以调用任何属于垃圾应用一部分的函数了。你应当首先确保数据库为空。

```
SPAM> (clear-database)
```

现在你可以用一些文本来训练过滤器。

```
SPAM> (train "Make money fast" 'spam)
```

然后看分类器是怎样判断的。

```
SPAM> (classify "Make money fast")
```

```
SPAM
SPAM> (classify "Want to go to the movies?")
UNSURE
```

尽管最终你所关心的全部只是那个分类，但可以看到原始的评分也是很有用的。得到两个值而不会干扰任何其他代码的最简单方法是改变 `classification` 来返回多个值。

```
(defun classification (score)
  (values
    (cond
      ((<= score *max-ham-score*) 'ham)
      ((>= score *min-spam-score*) 'spam)
      (t 'unsure))
    score))
```

你可以做出这个改变然后只重新编译这一个函数。因为 `classify` 返回 `classification` 所返回的任何东西，因此它也将返回两个值。但由于主返回值和以前相同，因此这两个函数的那些只需要一个值的调用者们将不会受到影响。现在当你测试 `classify` 时你能够精确地看到进入到分类中的评分。

```
SPAM> (classify "Make money fast")
SPAM
0.863677101854273D0
SPAM> (classify "Want to go to the movies?")
UNSURE
0.5D0
```

现在你可以看到如果你用更多的一些有用文本来训练过滤器的话将发生什么。

```
SPAM> (train "Do you have any money for the movies?" 'ham)
1
SPAM> (classify "Make money fast")
SPAM
0.7685351219857626D0
```

它仍然是垃圾消息但有一点不确定，因为“money”在有用文本中也出现了。

```
SPAM> (classify "Want to go to the movies?")
HAM
0.17482223132078922D0
```

而现在它被清楚地识别成有用的文本，这要感谢单词“movies”的存在，这是一个有用的特点。

不过，你可能并不真的想手工训练这个过滤器，你真正喜欢的是一种简单的方式来指向一堆文件并在其上训练它。并且因为你想要测试该过滤器实际工作的效果，你会希望随后使用它来分类另外一些已知类型的文件并查看分类的效果。因此你在本章中将要编写的最后一点代码将是一套测试系统，它在一个已知类型的消息库上测试该过滤器，使用其中的特定比例用于训练，然后在其余的部分上测量该过滤器在分类时的精度。

23.7 测试过滤器

为了测试该过滤器，你需要一个已知类型的消息库。你可以使用你邮箱中的消息，或者你可

以从 Web 上获得一个可用的消息库。例如，SpamAssassin 消息库¹²上含有手工分类成垃圾、轻度有用和重度有用的消息几千条。为了更容易地使用你所拥有的文件，你可以定义一个驱动在一个文件/类型对的数组上的测试平台。你可以定义一个函数来接受一个文件名和一个类型并像下面这样将其添加到消息库中：

```
(defun add-file-to-corpus (filename type corpus)
  (vector-push-extend (list filename type) corpus))
```

corpus 的值应当是一个带有填充指针的可调整向量。例如，你可以像这样创建一个新的库：

```
(defparameter *corpus* (make-array 1000 :adjustable t :fill-pointer 0))
```

如果你的有用消息和垃圾消息已经分别放在了不同的目录中，那么你可能想要一次性将一个目录中的所有文件作为相同的类型添加到库中。下面这个函数使用了来自第 15 章的 list-directory 函数，它实现了上述想法：

```
(defun add-directory-to-corpus (dir type corpus)
  (dolist (filename (list-directory dir))
    (add-file-to-corpus filename type corpus)))
```

例如，假设你有一个 mail 目录，它含有两个子目录 spam 和 ham，每个都含有指定类型的消息；你可以像下面这样添加这两个目录中的所有文件到 *corpus* 中：

```
SPAM> (add-directory-to-corpus "mail/spam/" 'spam *corpus*)
NIL
SPAM> (add-directory-to-corpus "mail/ham/" 'ham *corpus*)
NIL
```

现在你需要一个函数来测试分类器。基本的策略将是选择库中的一个随机片断来训练然后通过分类库的其余部分来测试这个库，将由 classify 所返回的分类与已知的分类进行比较。你主要想知道的是分类器的精度——多少百分比的消息被正确分类了？但你还可能对错误分类的消息以及错误的方向感兴趣——究竟是假阳性更多还是假阴性更多？为了方便地对分类器的行为进行不停的分析，你应当定义测试函数来构造一个原始结果的列表，你可以随后用任何想要的方法来分析它。

主测试函数可能看起来像这样：

```
(defun test-classifier (corpus testing-fraction)
  (clear-database)
  (let* ((shuffled (shuffle-vector corpus))
         (size (length corpus))
         (train-on (floor (* size (- 1 testing-fraction)))))
    (train-from-corpus shuffled :start 0 :end train-on)
    (test-from-corpus shuffled :start train-on)))
```

这个函数从清空特性数据库开始。¹³然后它对整个库进行洗牌，使用一个你将很快实现的函数，基于其 testing-fraction 参数来找出它将训练在多少消息上以及多少消息将被保留用来测试。两个辅助函数 train-from-corpus 和 test-from-corpus 都将带有 :start 和 :end 关键字参

¹²包括 SpamAssassin 消息库在内的几个垃圾消息库可以从

<http://nexp.cs.pdx.edu/~psam/cgi-bin/view/PSAM/CorpusSets> 下载到。

¹³ 如果你想要进行一个测试而不想干扰已有的数据库那么你可以用一个 LET 绑定 *feature-database*、*total-spams* 和 *total-hams*，但这样你在测试结束之后就没有办法查看数据库了——除非你在函数中返回了你所用到的这些值。

数，从而允许它们操作在给定消息库的一个子序列上。

`train-from-corpus` 函数相当简单——简单地循环在库的适当部分，使用 `DESTRUCTURING-BIND` 从每个元素中解出文件名和类型，然后将命名文件的文本和类型传给 `train`。由于某些邮件消息，尤其是那些带有附件的，尺寸会比较大，因此你应当限制它从消息中所获取的字符数量。它将使用一个你将很快实现的函数 `start-of-file` 来获取文本，该函数接受一个文件名和一个字符的最大个数来返回相应的文本。`train-from-corpus` 看起来像下面这样：

```
(defparameter *max-chars* (* 10 1024))

(defun train-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) do
        (destructuring-bind (file type) (aref corpus idx)
          (train (start-of-file file *max-chars*) type))))
```

`test-from-corpus` 函数和上述函数相似，除了你想要返回一个含有每个分类结果的列表从而可以稍后来分析它们。这样，你应当同时捕捉由 `classify` 所返回的分类和评分数据，然后将文件名、实际类型、由 `classify` 所返回的类型，以及评分收集在一个列表中。为了使结果更加可读，你可以在列表中包含一些关键字来指示每个值的含义。

```
(defun test-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) collect
        (destructuring-bind (file type) (aref corpus idx)
          (multiple-value-bind (classification score)
              (classify (start-of-file file *max-chars*))
            (list
              :file file
              :type type
              :classification classification
              :score score)))))
```

23.8 一组工具函数

为了完成 `test-classifier` 的实现，你还需要编写两个事实上跟垃圾过滤没有特别关系的工具函数，`shuffle-vector` 和 `start-of-file`。

一个实现 `shuffle-vector` 的简单有效方法是使用 Fisher-Yates 算法¹⁴。你可以从实现一个函数 `nshuffle-vector` 开始，它可以就地重排一个向量。这个名字遵循了与诸如 `NCONC` 和 `NREVERSE` 等其他破坏性函数同样的命名规则。它看起来像这样：

```
(defun nshuffle-vector (vector)
  (loop for idx downfrom (1- (length vector)) to 1
        for other = (random (1+ idx))
        do (unless (= idx other)
             (rotatef (aref vector idx) (aref vector other))))
  vector))
```

¹⁴ 这个算法以发明了概率合并方法的同一个 Fisher 和 Frank Yates 来命名，后者是 Fisher《Statistical Tables for Biological, Agricultural and Medical Research》(Oliver & Boyd, 1938) 一书的共同作者。根据 Knuth 的说法，他们提供了该算法的第一个公开发表的描述。

非破坏性的版本简单地对最初的向量做一个拷贝，然后将它传给破坏性的版本。

```
(defun shuffle-vector (vector)
  (nshuffle-vector (copy-seq vector)))
```

另一个工具函数 `start-of-file` 除了一点以外也是非常直接的。读取一个文件的内容到内存中最有效的方式是创建一个适当尺寸的数组并使用 `READ-SEQUENCE` 来填充其内容。因此看起来你应该创建一个字符数组，其长度要么是文件的尺寸要么是你想要读取的字符的最大数量，后者相对小一些。不幸的是，如同我在第 14 章里所提到的，函数 `FILE-LENGTH` 在处理字符流时完全没有很好地定义，因为一个文件中所编码的字符个数可能同时取决于所使用的字符编码和文件中的特定文本。在最坏的情况下，对一个文件中字符数的精确测量的唯一方法是实际读取整个文件。这样，`FILE-LENGTH` 在处理字符流时存在歧义；而多数实现中，`FILE-LENGTH` 总是返回文件的字节数，这可能大于可从文件中读取的字符数。

不过，`READ-SEQUENCE` 可以返回实际读取的字符数。因此，你可以尝试读取由 `FILE-LENGTH` 所报告的字符数并在实际读取的字符数较少时返回一个子串。

```
(defun start-of-file (file max-chars)
  (with-open-file (in file)
    (let* ((length (min (file-length in) max-chars))
           (text (make-string length))
           (read (read-sequence text in)))
      (if (< read length)
          (subseq text 0 read)
          text))))
```

23.9 分析结果

现在你开始编写一些代码来分析由 `test-classifier` 所生成的结果。回顾 `test-classifier` 返回了由 `test-from-corpus` 所返回的列表，其中每个元素是一个代表了分类一个文件的结果的 plist。这个 plist 含有该文件的名字、文件的实际类型、分类以及由 `classify` 所返回的评分。你应当编写的分析性代码的第一步是一个函数，它可以返回一个符号来代表一个给定结果究竟是正确的、假阳性的、假阴性的、错过的有用消息或错过的垃圾消息。你可以使用 `DESTRUCTURING-BIND` 从一个单独的结果列表中取出 `:type` 和 `:classification` 元素（使用 `&allow-other-keys` 来告诉 `DESTRUCTURING-BIND` 忽略任何其他的键值对）然后使用嵌套的 `ECASE` 将不同的配对转换成单一符号。

```
(defun result-type (result)
  (destructuring-bind (&key type classification &allow-other-keys) result
    (ecase type
      (ham
       (ecase classification
         (ham 'correct)
         (spam 'false-positive)
         (unsure 'missed-ham)))
      (spam
       (ecase classification
         (ham 'false-negative)
         (spam 'correct)
         (unsure 'missed-spam))))))
```

你可以在 REPL 中测试这个函数。

```
SPAM> (result-type '(:FILE #p"foo" :type ham :classification ham :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type spam :classification spam :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type ham :classification spam :score 0))
FALSE-POSITIVE
SPAM> (result-type '(:FILE #p"foo" :type spam :classification ham :score 0))
FALSE-NEGATIVE
SPAM> (result-type '(:FILE #p"foo" :type ham :classification unsure :score 0))
MISSED-HAM
SPAM> (result-type '(:FILE #p"foo" :type spam :classification unsure :score 0))
MISSED-SPAM
```

有了这个函数就可以方便地以多种方式切分 `test-classifier` 的结果了。例如，你可以从为每种结果类型定义谓词函数开始。

```
(defun false-positive-p (result)
  (eql (result-type result) 'false-positive))

(defun false-negative-p (result)
  (eql (result-type result) 'false-negative))

(defun missed-ham-p (result)
  (eql (result-type result) 'missed-ham))

(defun missed-spam-p (result)
  (eql (result-type result) 'missed-spam))

(defun correct-p (result)
  (eql (result-type result) 'correct))
```

使用这些函数你可以轻易地使用我在第 11 章里讨论的列表和序列操作函数来解出并统计特定类型的结果。

```
SPAM> (count-if #'false-positive-p *results*)
6
SPAM> (remove-if-not #'false-positive-p *results*)
((:FILE #p"ham/5349" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999983107355541d0)
 (:FILE #p"ham/2746" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.6286468956619795d0)
 (:FILE #p"ham/3427" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9833753501352983d0)
 (:FILE #p"ham/7785" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9542788587998488d0)
 (:FILE #p"ham/1728" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.684339162891261d0)
 (:FILE #p"ham/10581" :TYPE HAM :CLASSIFICATION SPAM :SCORE
0.9999924537959615d0))
```

你还可以使用 `result-type` 所返回的符号作为哈希表或 `alist` 中的键。例如，你可以编写一个函数来打印结果中每种类型的个数和比例，该函数使用一个 `alist` 将每种类型和一个额外的符号 `total` 映射到一个计数上。

```
(defun analyze-results (results)
  (let* ((keys '(total correct false-positive
                        false-negative missed-ham missed-spam))
         (counts (loop for x in keys collect (cons x 0))))
    (dolist (item results)
      (incf (cdr (assoc 'total counts)))
      (incf (cdr (assoc (result-type item) counts)))))
    (loop with total = (cdr (assoc 'total counts))
          for (label . count) in counts
```

```
do (format t "~&~@(~a~):~20t~5d~,5t: ~6,2f%~%"  
         label count (* 100 (/ count total))))))
```

该函数当被传递了一个由 `test-classifier` 所生成的结果列表时将给出下面的输出：

```
SPAM> (analyze-results *results*)  
Total: 3761 : 100.00%  
Correct: 3689 : 98.09%  
False-positive: 4 : 0.11%  
False-negative: 9 : 0.24%  
Missed-ham: 19 : 0.51%  
Missed-spam: 40 : 1.06%  
NIL
```

而在分析的最后你可能想要查看为什么一个单独的消息被分类成某种类型。下面的函数将为你显示这点：

```
(defun explain-classification (file)  
  (let* ((text (start-of-file file *max-chars*))  
         (features (extract-features text))  
         (score (score features))  
         (classification (classification score)))  
    (show-summary file text classification score)  
    (dolist (feature (sorted-interesting features))  
      (show-feature feature))))  
  
(defun show-summary (file text classification score)  
  (format t "~&~a" file)  
  (format t "~2%~a~2%" text)  
  (format t "Classified as ~a with score of ~,5f~%" classification score))  
  
(defun show-feature (feature)  
  (with-slots (word ham-count spam-count) feature  
    (format  
     t "~&~2t~a~30thams: ~5d; spams: ~5d;~,10tprob: ~,f~%"  
     word ham-count spam-count (bayesian-spam-probability feature))))  
  
(defun sorted-interesting (features)  
  (sort (remove-if #'untrained-p features) #'< :key  
 #'bayesian-spam-probability))
```

23.10 接下来的工作

很明显，你可以用这些代码做更多的事。为了将它变成一个真正的垃圾过滤应用程序，你需要找出一种方式来将它集成到你正常的 e-mail 基础设施中。一种使它方便地与几乎任何 e-mail 客户端相集成的思路是编写一点代码来使它成为一个 POP3 代理——这是多数 e-mail 客户端用来从邮件服务器上获取邮件所使用的协议，这样一个代理将从你的实际 POP3 服务器中获取邮件并为你的 e-mail 客户端提供服务，在这个过程中它要么将垃圾邮件标记上一个你的 e-mail 客户端过滤器可以理解的信头，或是直接把它放在一边。当然你可能还需要一种方式来与过滤器沟通有关错误分类的信息——只要你把它设置成一个服务器，你就可以提供一个 Web 接口。我将在第 26 章里谈及如何编写 Web 接口，并且你将在第 29 章里为一个不同的应用构建一个 Web 接口。

或者你可能想要工作在改进基本分类上——一个可能开始的地方是令 `extract-features` 更加专业。特别地，你可以使分词器更聪明地处理 e-mail 的内部结构——你可以为出现在消息体

和消息头中的单词解出不同类型的特性。并且你可以解码诸如 base64 和 quoted printable 在内的多种类型的消息编码，因为垃圾发送者经常使用这些编码来扰乱它们的消息。

但是我将把这些改进留给你。现在你已准备好继续前进来构建一个流式 MP3 服务器，先从编写一个解析二进制文件的通用库开始。

第24章 实践：解析二进制文件

在本章里，我将向你介绍如何构建一个库，用来编写那些读取和写入二进制文件的代码。你将在第 25 章里使用这个库来编写一个 ID3 标记的解析器，后者是用来保存 MP3 文件中诸如艺术家和专辑名这类元数据的机制。这个库同样也是关于如何使用宏来为语言扩展新的控制构造的一次示例，它将通用语言转化成了一种用于处理特定问题的专用语言，在本例中则是读取和写入二进制数据。由于你将循序渐进地开发这个库，包括几个部分可用的版本，因此看起来你将会编写很多代码。但是当一切都已完成时，整个库的规模将会少于 150 行代码，而其中最长的宏也只有 20 行。

24.1 二进制文件

在一个足够低的抽象层面上，所有文件都是“二进制”的，因为看起来它们只是含有一些以二进制形式编码的数字罢了。不过，通常会把所谓“文本”文件和“二进制”文件区别看待，前者的所有数字都可以被解释成表示人类可读文本的字符，而后者所含有的数据如果被解释成字符的话，将会得到不可打印的字符。¹

二进制文件格式通常被设计成可以简洁高效地进行解析——这是它们相对于基于文本的格式的主要优点。为了同时满足这些要求，它们通常采用可以轻易映射到程序内存中数据结构的磁盘结构来保存。²

即将编写的这个库将提供你一种简单的方式来定义那些在二进制文件所定义的磁盘结构和内存中 Lisp 对象之间的映射关系。使用这个库，编写一个可以读取二进制文件的程序将会很容易，先将其转化成可管理的 Lisp 对象，然后再写回到另一个正确格式化的二进制文件中。

¹ 在 ASCII 编码中，前 32 个字符时不可打印的控制字符，最初用来控制一个终端服务器的行为，让其做到诸如通过喇叭发声、回退一个字符、换行以及将光标移到行首之类的操作。在这 32 个控制字符串里，只有三个通常可以在文本文件中看到：换行、回车，以及水平制表 (tab)。

² 某些二进制文件格式确实是内存数据结构——在许多操作系统中有可能将一个文件直接映射进内存，然后诸如 C 语言这样的底层语言就可以将含有文件内容的内存区域当作任何其他的内存一样处理；写入该内存区域的数据在解除映射时会被写回到文件中。不过，这些格式都是平台相关的，因为即便像整数这样的简单数据类型在内存中的表示方式也取决于程序所运行的硬件。这样，任何倾向于可移植的文件格式都必须为所有其数据类型定义规范的数据表示，使其可以映射到特定类型的机器或编程语言的实际内存中数据表示上。

24.2 二进制格式基础

读写二进制文件的起始点是打开一个用于读写单个字节的文件。如同我在第 14 章里所讨论的那样，`OPEN` 和 `WITH-OPEN-FILE` 都接受一个关键字参数 `:element-type`，它可以控制用于流传的基本单元。当你在处理二进制文件时，你将需要把该参数设定为 (`unsigned-byte 8`)。一个通过这样的 `:element-type` 打开的输入流将在每次传给 `READ-BYTE` 时返回一个介于 0 到 255 之间的整数。同样地，你可以通过向 `WRITE-BYTE` 传递介于 0 到 255 之间的数字来向一个 (`unsigned-byte 8`) 输出流写入字节。

在单独字节的层面之上，多数的二进制格式都使用了一小组基本数据类型——以多种方式编码的数字、文本字符串、位字段等——然后再复合成更复杂的结构。所以你的首要任务是定义一个用来编写那些读写给定二进制格式中使用的基本数据类型的框架。

先取一个简单的例子，假设你正在处理一个将无符号 16 位整数作为基本数据类型的二进制格式。为了读取这样一个整数，你需要读取两个字节，将一个字节乘以 256，也就是 2^8 ，再跟另一个字节相加，从而将它们组合成单个整数。举个例子，假设指定了这个 16 位量的二进制格式是以 big-endian³ 形式保存的，那么以最重要字节优先的顺序，你可以用下面的函数来读取这样一个数：

```
(defun read-u2 (in)
  (+ (* (read-byte in) 256) (read-byte in)))
```

不过，Common Lisp 提供了一种更便利的方式进行这些按位处理。函数 `LDB`，就是加载字节 (`load byte`) 的意思，可被用来从一个整数中解出和设置（通过 `SETF`）任意数量的连续位。⁴ 整数中的位数量和它们的位置由 `BYTE` 函数所创建的一个位描述符所指定。`BYTE` 接受两个参数，需要解出（或设置）的位数量以及最右边那一位相对整数中最不重要位来说以零为开始的位置。`LDB` 接受一个字节描述符和需要解出位数据的那个整数，然后返回由解出的位所代表的整数。这样，你可以像下面这样解出一个整数的最不重要的八位元：

```
(ldb (byte 8 0) #xabcd) -> 205 ; 205 is #xcd
```

为了得到下一个八位元，你可以使用字节描述符 (`byte 8 8`)，如下所示：

```
(ldb (byte 8 8) #xabcd) -> 171 ; 171 is #xab
```

你可以将 `LDB` 与 `SETF` 配合使用来设置一个保存在可 `SETF` 的位置上的整数的指定位。

```
CL-USER> (defvar *num* 0)
*NUM*
```

³ 术语 big-endian 和它的反义词 little-endian 来自 Jonathan Swift 的《Gulliver's Travels》(格列佛游记)，用来表达一个多字节的数字在诸如内存和文件中保存时所采用的字节顺序。例如，数字 43981，十六进制为 abcd，当表示成 16 位量时由两个字节所组成，ab 和 cd。只要各方意见一致，对于一台电脑来说以何种顺序来保存这两个字节都是无关紧要的。当然，尽管你可以在同样好的两种方式中任意选择，但可以保证的是并非人人都同意。要想了解关于此事的更多隐情，以及术语 big-endian 和 little-endian 是最早以这种含义应用在哪里的，可以阅读 Danny Cohen 的“On Holy Wars and a Plea for Peace”，地址是 <http://khavrinen.lcs.mit.edu/wollman/ien-137.txt>

⁴ `LDB` 跟与之相关的函数 `DPB` 均来自 DEC PDP-10 计算机的汇编函数，它们本质上做的事情相同。两个函数都操作在以二进制补码表示的整数上，无论特定 Common Lisp 实现使用的是何种内部表示法。

```
CL-USER> (setf (ldb (byte 8 0) *num*) 128)
128
CL-USER> *num*
128
CL-USER> (setf (ldb (byte 8 8) *num*) 255)
255
CL-USER> *num*
65408
```

因此，你也可以像下面这样来编写 `read-u2`:⁵

```
(defun read-u2 (in)
  (let ((u2 0))
    (setf (ldb (byte 8 8) u2) (read-byte in))
    (setf (ldb (byte 8 0) u2) (read-byte in)))
  u2))
```

为了把一个数字写成 16 位整数，你需要解出单独的 8 位字节并逐个地写它们。为了解出单独的字节，你只需要以同样的字节描述符来使用 LDB 就可以了。

```
(defun write-u2 (out value)
  (write-byte (ldb (byte 8 8) value) out)
  (write-byte (ldb (byte 8 0) value) out))
```

当然，你也可以用许多其他的方式来编码整数——使用不同的字节数，不同的尾部处理 (endianness)，以及有符号或无符号的格式。

24.3 二进制文件中的字符串

文本字符串是另一种你将在许多二进制格式中遇到的基本数据类型。当你逐个字节地读文件时，你不能直接地读写字符串——你需要每次一个字节地对它们进行解码或者编码，就像你对二进制编码的数字所做的那样。并且正如你可以用多种方式来编码一个整数，你也可以用多种方式来编码一个字符串。不过最起码来讲，二进制格式必须指定究竟需要编码多少个单独的字符。

为了将字节转化成字符，你既需要知道字符的编码 (code) 也需要知道你正在使用的编码方式 (encoding)。一个字符编码定义了从正整数到字符之间的映射。映射表中的每个数字被成为一个代码点 (code point)。例如，ASCII 就是一种字符编码，它将 0-127 之间的数字映射到了用于拉丁字母表的一些特定字符上。另一方面，字符编码定义了代码点在诸如文件这种基于字节的媒体中是如何被表示成一个字节序列的。对于那些使用八位或者更少位的编码，例如 ASCII

⁵ Common Lisp 也提供了用来对整数进行移位和掩码处理的函数，这种方式可能对 C 和 Java 程序员来说更熟悉些。例如，你还可以用第三种方式编写 `read-u2`，像下面这样使用那些函数：

```
(defun read-u2 (in)
  (logior (ash (read-byte in) 8) (read-byte in)))
该函数几乎跟下面的 Java 方法完全等价：
public int readU2 (InputStream in) throws IOException {
  return (in.read() << 8) | (in.read());
}
```

名字 LOGIOR 和 ASH 是 LOGical Inclusive OR (逻辑同或) 和 Arithmetic SHift (算术移位) 的简称。ASH 以给定的位数对一个整数进行移位，当其第二个参数为正时左移，为负时右移。LOGIOR 通过对整数的每个位做逻辑或来将它们合并在一起。另一个函数 LOGAND 可以做按位与，这可用来掩盖特定的位。尽管如此，对于本章和接下来的章节里你将用到的各种按位操作，使用 LDB 和 BYTE 将是既便利又符合习惯的 Common Lisp 风格。

和 ISO-8859-1，编码方式是相当直接的——每一个数值刚好编码成单个字节。

相对直接的是纯粹的双字节编码方式，例如 UCS-2，它在 16 位值和字符之间做映射。双字节编码方式比单字节编码方式更加复杂的唯一理由就是你可能还需要知道那些 16 位的值究竟打算编码成 big-endian 还是 little-endian 格式的。

变长的编码方式对于不同的数值使用不同数量的八位元，这使其更加复杂但却令它们在许多时候更加紧凑。例如，UTF-8，一种设计用于 Unicode 字符代码的编码方式，使用单个八位元来编码 0–127 之间的值，同时使用至多四个八位元来编码最多 1,114,111 个不同的值。⁶

由于在 Unicode 字符集中 0–127 的代码点映射到与 ASCII 相同的字符上，因此一段 UTF-8 编码的只由 ASCII 字符构成的文本与 ASCII 编码的结果是相同的。另一方面，几乎完全由 UTF-8 中需要四字节来表达的字符所构成的文本如果用直接的双字节编码方式来编码的话反而会更紧凑。

Common Lisp 提供了两个函数用来在数值的字符代码和字符对象之间进行转化：`CODE-CHAR` 接受一个数值代码并返回一个字符，而 `CHAR-CODE` 则接受一个字符并返回其数值的代码。语言标准并未指定一个实现必须使用的字符编码方式，因此并不保证你可以表示有可能需要作为一个 Lisp 字符被编码进给定二进制文件格式中的每一个字符。不过，几乎所有的现代 Common Lisp 实现都使用 ASCII、ISO-8859-1 或 Unicode 作为其原生的字符编码。由于 Unicode 是 ISO-8859-1 的超集，后者则是 ASCII 的超集，所以如果你正在使用一个支持 Unicode 的 Lisp 平台，那么 `CODE-CHAR` 和 `CHAR-CODE` 将可以直接被用来转化这三种编码方式的任何一种。⁷

除了指定字符编码方式以外，一个字符串的编码工作还必须指定如何编码字符串的长度。有三种技术通常用在二进制文件格式中。

最简单的方式是不编码，而是让它成为字符串在更大的结构中某个位置上的隐含值：一个文件中的特定元素可能总是一个特定长度的字符串，或者一个字符串可能是一个变长数据类型中的最后一个元素，结构的总长度决定了有多少剩余字节可被用来作为字符串数据读取。这两种方式都被用在了 ID3 标签中，正如你在下一章里将会看到的那样。

另外两种技术可被用来在无需依赖上下文的情况下编码变长的字符串。一种方式是先编码字符串的长度然后再跟上字符数据——解析器首先读到一个整数值（以某种特定的整数格式）然后再读取相应数量的字符。另一种方式是先写入字符数据然后跟上一个不可能出现在字符串中的定界符，例如空字符。

不同的表示法各自具有不同的优点和缺点，但当你已经在处理指定的二进制格式时，你就无法控制究竟哪种编码方式被使用了。不过，没有哪种编码方式比其他方式时特别难以读写的。作为一个例子，下面是一个用来读取空字符结尾的 ASCII 字符串的函数，假设你的 Lisp 实现使用了 ASCII 或是诸如 ISO-8859-1 或完全的 Unicode 这两个它的超集作为原生字符编码：

⁶ 最初 UTF-8 被设计用来表示 31 位的字符代码，并在每个代码点上使用至多六个字节。不过，Unicode 代码点的最大值是 #x10ffff，因此一个 UTF-8 编码的 Unicode 字符在每个代码点上只需要至多四个字节就够了。

⁷ 如果你需要解析一个用到了其他字符编码的文件格式，或者如果你需要使用一个非 Unicode 的 Common Lisp 实现来解析一个含有任意 Unicode 字符串的文件，那么你总是可以将这些字符串在内存中表示成整数代码点的向量。它们将不会成为 Lisp 字符串，因为你将无法使用字符串函数来管理或比较它们，但你将有可能像对任意向量那样对它们做任何事情。

```
(defconstant +null+ (code-char 0))

(defun read-null-terminated-ascii (in)
  (with-output-to-string (s)
    (loop for char = (code-char (read-byte in))
          until (char= char +null+) do (write-char char s))))
```

其中的 `WITH-OUTPUT-TO-STRING` 宏，我在第 14 章里提到过，这是一种在你不知道长度的情况下构造字符串的简单方式。它创建了一个 `STRING-STREAM` 并将其绑定到特定的变量名上，这里是 `s`。所有写入流的字符都被收集到一个字符串中，并随后作为 `WITH-OUTPUT-TO-STRING` 形式的结果返回。

为了写回一个字符串，你只需要将字符转换回可以用 `WRITE-BYTE` 来写的数值形式，然后在字符串内容后面写入一个空终止符即可。

```
(defun write-null-terminated-ascii (string out)
  (loop for char across string
        do (write-byte (char-code char) out))
  (write-byte (char-code +null+) out))
```

如同这些示例所显示的，读写二进制文件中基本元素的主要智力挑战是要理解究竟该如何解释出现在一个文件中的字节并将其映射到 Lisp 数据类型。如果一个二进制格式是良好定义的，那么这将是一个相当直接的命题。事实上编写函数来读写一个特定的编码，正如它们所说的，根本是小事一桩。

现在你可以转而考虑读写更复杂的磁盘结构和如果将它们映射到 Lisp 对象上的问题了。

24.4 复合结构

由于二进制格式通常用来表达那些可以轻易映射到内存数据结构上的数据，因此不难理解那些复合的磁盘结构通常是以一种接近于编程语言定义内存中数据结构的方式来定义的。通常一个复合磁盘结构将有一些命名的部分所组成，每个部分其本身要么是诸如数字或字符串这样的基本类型，要么是另一个复合结构，或者可能是这些值的一个集合。

例如，一个定义在版本 2.2 规范中的 ID3 标签的组成部分包括：一个三字符的 ISO-8859-1 字符串（总是“ID3”）的头部；两个单字节无符号整数用来指定规范的主版本和修订号；八位的布尔旗标；以及四个字节以特定于 ID3 规范的编码方式编码的整个标签的长度。紧接着头部的是一个帧的列表，每个都有其自己的内部结构。在帧之后是用来填满头部所指定的标签长度所需数量的空字节。

如果你以面向对象的眼光来看待世界的话，复合结构看起来会和类很像。例如，你可以编写一个类来表示一个 ID3 标签。

```
(defclass id3-tag ()
  ((identifier :initarg :identifier :accessor identifier)
   (major-version :initarg :major-version :accessor major-version)
   (revision :initarg :revision :accessor revision)
   (flags :initarg :flags :accessor flags)
   (size :initarg :size :accessor size)
   (frames :initarg :frames :accessor frames)))
```

这个类的实例将成为用来保存 ID3 标签的完美仓库。你可以随后编写函数来读写该类的实例。例如，假设已有了特定的其他函数用来读取适当的基本数据类型，那么一个 `read-id3-tag` 函数可能看起来像下面这样：

```
(defun read-id3-tag (in)
  (let ((tag (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) tag
      (setf identifier      (read-iso-8859-1-string in :length 3))
      (setf major-version   (read-ul in))
      (setf revision        (read-ul in))
      (setf flags           (read-ul in))
      (setf size            (read-id3-encoded-size in))
      (setf frame           (read-id3-frames in :tag-size size)))
    tag))
```

函数 `write-id3-tag` 将会具有类似的结构——你需要使用适当的 `write-*` 函数来输出那些保存在 `id3-tag` 对象中的值。

不难看出你可以怎样来编写一个适当的类来表示一个规范中的所有复合数据类型，以及用于每个类和必要的基本类型的 `read-foo` 和 `write-foo` 函数。但是很容易也可以看出所有用来读和写的函数都将会非常相似，区别仅在于指定它们要读取的类型和它们所保存在的槽的名字。这实在是太浪费笔墨了，尤其是当你发现在 ID3 规范中它只花了四行文本来描述一个 ID3 标签的结构，而你已经写了八行代码却还没写到 `write-id3-tag`。

你真正想要的是一种以类似规范中伪代码的形式来描述像 ID3 标签这样的结构的方式，随后这些描述可以被展开成定义了 `id3-tag` 类和用来在磁盘上的字节和类实例之间相互转化函数的代码。听起来这正是宏的任务。

24.5 设计宏

由于你已经对你得宏需要生成怎样的代码有了大致的想法，下一步，根据我在第 8 章里归纳的宏编写过程，就是要切换视角，转而思考这样一个宏的具体调用将看起来是怎样的。因为目标是可以书写像 ID3 规范中的伪代码一样紧凑的东西，所以你可以从那里开始。一个 ID3 标签的头是像下面这样指定的：

```
ID3/file identifier      "ID3"
ID3 version              $02 00
ID3 flags                %xx000000
ID3 size                 4 * %0xxxxxxxx
```

在规范的写法里，这意味着一个 ID3 标签的“文件标识符”是 ISO-8859-1 编码的字符串“ID3”。版本部分由两个字节构成，其中的第一个字节——对于当前版本的规范来说——具有值 2，第二个字节——再一次，对于当前版本来说——是 0。用于保存旗标的槽有 8 个位，其中除了前两个以外都是 0，然后长度由 4 个字节构成，每个字节的最重要的位上都是 0。

还有一些信息没有被上面的伪代码所覆盖到。例如，究竟编码了长度的 4 个字节应当如何解释是用几行文字来描述的。同样地，规范用文字描述了怎样才能编写代码来读和写由这个伪代码所指定的一个 ID3 标签。这样，你应该可以写出该伪代码的一个 S-表达式版本并将其展开成原本需要手写的类和函数的定义——比如说可能是类似下面这样的东西：

```
(define-binary-class id3-tag
  ((file-identifier (iso-8859-1-string :length 3))
   (major-version u1)
   (revision u1)
   (flags u1)
   (size id3-tag-size)
   (frames (id3-frames :tag-size size))))
```

这个形式的基本思想是定义一个类似于 `DEFCLASS` 所定义的一个 `id3-tag` 类，但和指定诸如 `:initarg` 和 `:accessor` 之类的东西所不同的是，每个槽描述符由槽的名字——`file-identifier`、`major-version` 等——和关于该槽在磁盘中如何表示的信息所构成。由于目前这些都还只是一点儿随想，所以你不必担心宏 `define-binary-class` 究竟是如何对诸如 `(iso-8859-1-string :length 3)`、`u1`、`id3-tag-size` 和 `(id3-frames :tag-size size)` 这些表达式进行处理的；对你来说，只要每个表达式都含有对于如何读写一个特定数据编码的必要信息就可以了。

24.6 把梦想变成现实

OK，对于优美代码的幻想就到此为止吧；现在你需要开始编写 `define-binary-class` 了——编写代码将那个关于 ID3 标签的样子的简洁表达方式转化成实际可用的代码：在内存中表示它、从磁盘中读取，以及将其写入磁盘。

作为开始，你应该为这个库定义一个包。下面是来自你可以从本书 Web 站点上下载到的版本中的包定义文件：

```
(in-package :cl-user)

(defpackage :com.gigamonkeys.binary-data
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :define-binary-class
           :define-tagged-binary-class
           :define-binary-type
           :read-value
           :write-value
           :*in-progress-objects*
           :parent-of-type
           :current-binary-object
           :+null+))
```

其中的 `COM.GIGAMONKEYS.MACRO-UTILITIES` 包里含有来自第 8 章的 `with-gensyms` 和 `once-only` 宏。

由于你已经有了想要展开成的代码的手写版本，因为编写这样一个宏应该不会太难。可以分而治之，先写一个只生成 `DEFCLASS` 形式的 `define-binary-class` 版本。

如果你回过头来观察那个 `define-binary-class` 形式，你将看到它接受两个参数：名字 `id3-tag` 以及一个槽描述符的列表，后者的每一个本身都是两元素列表。你需要从这些材料中构造出适当的 `DEFCLASS` 形式来。很明显第，在 `define-binary-class` 形式与一个正确的 `DEFCLASS` 形式之间最大的区别就在槽描述符中。来自 `define-binary-class` 的单个槽描述符看起来类似下面这样：

```
(major-version u1)
```

但这并不是一个合法的 DEFCLASS 槽描述符。相反，你需要类似下面这样的东西：

```
(major-version :initarg :major-version :accessor major-version)
```

其实很简单。首先定义一个简单的函数将一个符号转化成对应的关键字符。

```
(defun as-keyword (sym) (intern (string sym) :keyword))
```

现在定义一个函数，其接受一个 define-binary-class 槽描述符并返回一个 DEFCLASS 槽描述符。

```
(defun slot->defclass-slot (spec)
  (let ((name (first spec)))
    `',(name :initarg ,(as-keyword name) :accessor ,name)))
```

在你使用一个 IN-PACKAGE 调用切换到新包以后，你可以在 REPL 中测试这个函数。

```
BINARY-DATA> (slot->defclass-slot '(major-version u1))
(MAJOR-VERSION :INITARG :MAJOR-VERSION :ACCESSOR MAJOR-VERSION)
```

看起来不错。现在 define-binary-class 的第一个版本可以轻松搞定了。

```
(defmacro define-binary-class (name slots)
  `',(defclass ,name ()
    ,(mapcar #'slot->defclass-slot slots)))
```

这是一个简单的模板风格的宏——define-binary-class 通过插入类的名字和槽描述符列表来生成一个 DEFCLASS 形式，其中槽描述符列表的构造方法是将函数 slot->defclass-slot 应用到来自 define-binary-class 形式的槽描述符列表的每个元素上。

为了查看这个宏究竟生成了什么代码，你可以在 REPL 中求值下面的表达式。

```
(macroexpand-1 '(define-binary-class id3-tag
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)
   (frames         (id3-frames :tag-size size)))))
```

得到的结果，这里为了更好的可读性稍微重新格式化了一下，应当看起来眼熟，因为正是你早些时候手工编写的那个类定义：

```
(defclass id3-tag ()
  ((identifier      :initarg :identifier      :accessor identifier)
   (major-version  :initarg :major-version  :accessor major-version)
   (revision       :initarg :revision       :accessor revision)
   (flags          :initarg :flags          :accessor flags)
   (size           :initarg :size           :accessor size)
   (frames         :initarg :frames         :accessor frames)))
```

24.7 读取二进制对象

下一步你需要让 define-binary-class 也能生成一个函数用来读取这个新类的一个实例。回

顾你之前写的 `read-id3-tag` 函数，看起来有些滑稽，因为 `read-id3-tag` 的存在并不是很正常——为了读取每一个槽的值，你都要调用一个不同的函数。更不用说函数 `read-id3-tag` 的名字，尽管来自你所定义的类的名字，但其本身却并不是 `define-binary-class` 的参数，因此没有办法像类名那样直接插入到模板中。

你可以通过设计并遵循一个命名约定来处理这两个问题，让宏可以基于槽描述符中的类型名来找出需要调用的函数名。不过，这将需要 `define-binary-class` 来生成名字 `read-id3-tag`，这是有可能的但不是个好主意。创建全局定义的宏通常应当仅使用那些由调用者传递给它们的名字；背后生成名字的宏可能会在生成的名字和其他地方使用的名字刚好同名时导致难以预测——并且难以调试——的名字冲突。⁸

你可以避免这些不便，只要你注意到所有这些读取一个特定类型值的函数都有相同的基本目的：从一个流中读取指定类型的一个值。说白了它们都是单个通用操作的实例。并且对于“通用”（generic）的使用应当让你直接想到问题的解决方案：与其定义一堆互不相关的函数，各有各的名字，还不如定义一个广义函数，`read-value`，以及特定用来读取不同类型值的方法。

这就是说，代替了定义函数 `read-iso-8859-1-string` 和 `read-u1`，现在你可以将 `read-value` 定义成一个接受两个必要参数的广义函数，一个类型和一个流，以及可能的一些关键字参数。

```
(defgeneric read-value (type stream &key)
  (:documentation "Read a value of the given type from the stream."))
```

通过不带任何实际关键字参数地指定 `&key`，你可以允许不同的方法定义它们自己的 `&key` 参数而不做具体要求。不过这意味着每个特化在 `read-value` 上的方法都将在它们的形参列表中至少包括 `&key` 或 `&rest` 参数中的一个，这样才能与广义函数兼容。

然后你定义使用 `EQL` 特化符将类型参数特化在你想要读取的类型名上的方法。

```
(defmethod read-value ((type (eql 'iso-8859-1-string)) in &key length) ...)
(defmethod read-value ((type (eql 'u1)) in &key) ...)
```

然后你就可以让 `define-binary-class` 生成一个特化在类型名 `id3-tag` 上的 `read-value` 方法了，而该方法可以通过调用带有适当的槽类型作为第一个参数的 `read-value` 来实现。你想要生成的代码将看起来像下面这样：

```
(defmethod read-value ((type (eql 'id3-tag)) in &key)
  (let ((object (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) object
      (setf identifier (read-value 'iso-8859-1-string in :length 3))
      (setf major-version (read-value 'u1 in))
      (setf revision (read-value 'u1 in))
      (setf flags (read-value 'u1 in))
      (setf size (read-value 'id3-encoded-size in))
      (setf frames (read-value 'id3-frames in :tag-size size)))
    object))
```

因此，就像你需要一个函数来将一个 `define-binary-class` 槽描述符转化成一个 `DEFCLASS`

⁸ 不幸的是，语言本身在这个观点上并没有提供一个好的榜样：宏 `DEFSTRUCT`——由于被 `DEFCLASS` 所取代因此我不打算讨论它——可以基于给定结构的名字来生成新的函数名。`DEFSTRUCT` 的不良示例导致了许多新的宏编写者效仿。

槽描述符那样，现在你也需要一个函数接受 `define-binary-class` 槽描述符作为参数然后生成适当的 `SETF` 形式，也就是说，接受下面这个形式：

```
(identifier (iso-8859-1-string :length 3))
```

并返回下面结果的函数：

```
(setf identifier (read-value 'iso-8859-1-string in :length 3))
```

不过，上面的代码和 `DEFCLASS` 的槽描述符有一点儿区别：它包含了对一个变量 `in`——来自 `read-value` 方法的方法参数——的引用，该变量并非来源于槽描述符。它不一定非叫做 `in`，但无论你使用什么名字，它都必须你用在方法参数列表和其他 `read-value` 调用中的名字相同。眼下你可以避开关于这个名字的来源的问题，定义 `slot->read-value` 来接受一个流变量作为第二个参数。

```
(defun slot->read-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(setf ,name (read-value ',type ,stream ,@args))))
```

函数 `normalize-slot-spec` 用来正则化槽描述符的第二个元素，将类似 `u1` 这样的一个符号转化成列表 (`u1`) 从而让 `DESTRUCTURING-BIND` 可以解析它。它看起来像这样：

```
(defun normalize-slot-spec (spec)
  (list (first spec) (mklist (second spec))))
```



```
(defun mklist (x) (if (listp x) x (list x)))
```

你可以使用各种类型的槽描述符来测试 `slot->read-value`。

```
BINARY-DATA> (slot->read-value '(major-version u1) 'stream)
(SETF MAJOR-VERSION (READ-VALUE 'U1 STREAM))
BINARY-DATA> (slot->read-value '(identifier (iso-8859-1-string :length 3))
'stream)
(SETF IDENTIFIER (READ-VALUE 'ISO-8859-1-STRING STREAM :LENGTH 3))
```

有了这些函数你就可以将 `read-value` 添加到 `define-binary-class` 中了。如果你取一个手写的 `read-value` 方法并去掉任何特定类相关的内容，那么你将剩下这样的一个骨架：

```
(defmethod read-value ((type (eql ...)) stream &key)
  (let ((object (make-instance ...)))
    (with-slots (...) object
      ...
      object)))
```

所有你需要做的就是将这个骨架添加到 `define-binary-class` 模板中，其中的省略号部分替换成适当的名字和代码。你也可能会想要把变量 `type`、`stream` 和 `object` 替换成符号生成的名字以避免潜在的槽名字冲突，⁹ 这可以通过使用来自第 8 章的 `with-gensyms` 宏做到。

另外由于一个宏必须展开成单一形式，所以你需要在 `DEFCLASS` 和 `DEFMETHOD` 的外面包装一些形式。`PROGN` 习惯上用来让宏可以展开成多个定义，因为当它出现在一个文件的顶层时可以得到文件编译器的特殊处理，我曾在第 20 章里讨论过这点。

⁹ 技术上讲 `type` 或 `object` 不可能与槽名字冲突——最坏情况下它们会在 `WITH-SLOTS` 形式中被掩盖掉。不过简单地把一个宏模版中用到的所有局部变量都用 `GENSYM` 来生成肯定是无害的。

所以，你可以将 `define-binary-class` 改成下面这样：

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
      (defclass ,name ()
        ,(mapcar #'slot->defclass-slot slots))

      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let ((,objectvar (make-instance ',name)))
          (with-slots ,(mapcar #'first slots) ,objectvar
            ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
          ,objectvar))))
```

24.8 写二进制对象

生成用来写一个二进制类实例的代码将会做类似的处理。首先你可以定义一个 `write-value` 广义函数。

```
(defgeneric write-value (type stream value &key)
  (:documentation "Write a value as the given type to the stream."))
```

然后你定义一个助手函数将一个 `define-binary-class` 槽描述符转化成使用 `write-value` 来输出槽数据的代码。和 `slot->read-value` 函数一样，这个助手函数需要接受流变量的名字作为一个参数。

```
(defun slot->write-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    ` (write-value ',type ,stream ,name ,@args)))
```

现在你可以添加一个 `write-value` 模板到 `define-binary-class` 宏。

```
defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
      (defclass ,name ()
        ,(mapcar #'slot->defclass-slot slots))

      (defmethod read-value ((,typevar (eql ',name)) ,streamvar)
        (let ((,objectvar (make-instance ',name)))
          (with-slots ,(mapcar #'first slots) ,objectvar
            ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
          ,objectvar))

      (defmethod write-value ((,typevar (eql ',name)) ,streamvar ,objectvar)
        (with-slots ,(mapcar #'first slots) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

24.9 添加继承和标记的结构

尽管这个版本的 `define-binary-class` 能够处理独立的结构，但二进制文件格式通常定义了一些可以自然地采用子类和基类来建模的磁盘结构。因此你可能想要扩展 `define-binary-class` 来支持继承。

一个相关的用在许多二进制格式中的技术是存在于一些磁盘上的结构，其确切类型只有在读取了一些用来指示如何解析后续字节的数据以后才能决定。例如，ID3 标签中的大量帧全都共享了一个由字符串标识和长度所构成的统一的头结构。为了读取一个帧，你需要先读取标识符再用它的值来检测你正在查看的是哪一种帧类型以及如何解析该帧的主体。

当前的 `define-binary-class` 宏没有办法处理这种类型的读取操作——你可以使用 `define-binary-class` 来定义一个代表每种帧类型的类，但如果你没有至少读取标识符部分的话就无法知道这是哪个类型的帧。而如果其他代码读取了标识符以检测用来传给 `read-value` 的类型，那么这会打断 `read-value` 的运行，因为它期待读取构成它所实例化的类实例的全部数据。

你可以为 `define-binary-class` 添加继承来解决这个问题，并编写另一个宏 `define-tagged-binary-class`，用来定义那些“抽象”类，后者并不直接被实例化而是可以被那些知道如何读取足够数据来决定创建何种类型的类的 `read-value` 方法们所特化的。

为 `define-binary-class` 添加继承的第一步是为该宏添加一个参数来接受一个基类的列表。

```
(defmacro define-binary-class (name (&rest superclasses) slots) ...)
```

然后，在 `DEFCLASS` 模板中，插入该值以取代原先的空列表。

```
(defclass ,name ,superclasses
  ...)
```

不过，这里面还有些事要做。你还需要改变 `read-value` 和 `write-value` 方法，这样在定义一个基类时所生成的方法才可以被那些由子类所生成的方法用来读写继承的槽。

当前的 `read-value` 工作方式尤其有问题，因为它在填入内容之前就要实例化对象——很明显，你可能让方法根据读取基类的字段来实例化一个对象，同时让子类的方法去实例化并填充另一个不同的对象。

你可以通过将 `read-value` 划分成两部分来解决这个问题——一部分用来实例化正确类型的对象，而另一部分则用来填充一个已存在对象的槽。在写的方面其实会更简单，但你可以使用同样的技术。

因此你将定义两个新的广义函数，`read-object` 和 `write-object`，它们都接受一个已有的对象和一个流。定义在这些广义函数上的方法将用来读或写特定于它们所特化在的对象所属的类的槽。

```
(defgeneric read-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Fill in the slots of object from stream."))

(defgeneric write-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Write out the slots of object to the stream."))
```

把这些广义函数定义成使用带有 `:most-specific-last` 选项的 `PROGN` 方法组合的形式允许你定义特化在 `object` 的每个二进制类上的方法并让它们只处理实际定义在该类中的槽；`PROGN` 方法组合将组合所有可应用的方法并让继承体系中最不相关类首先运行，读写定义在该类中的槽，然后特化在下一个最不相关子类上的方法再运行，诸如此类。而由于所有对于特定类的重量

级操作现在都由 `read-object` 和 `write-object` 来完成了，所以你甚至不需要再定义特化了的 `read-value` 和 `write-value` 方法了；你可以定义缺省方法，其中假设类型参数就是一个二进制类的名字。

```
(defmethod read-value ((type symbol) stream &key)
  (let ((object (make-instance type)))
    (read-object object stream)
    object))

(defmethod write-value ((type symbol) stream value &key)
  (assert (typep value type))
  (write-object value stream))
```

注意到你是怎样将 `MAKE-INSTANCE` 用作一个通用的对象工厂的——尽管通常情况下由于确切知道想要实例化的类所以你使用一个引用了的符号作为第一个参数来调用 `MAKE-INSTANCE`，但你也可以使用任何求值成一个类名的表达式来调用这个函数，在本例中则使用了 `read-value` 方法中的 `type` 参数。

`define-binary-class` 中的实际改变是相对较少的，现在是定义 `read-object` 和 `write-object` 而不是 `read-value` 和 `write-value` 上的方法了。

```
(defmacro define-binary-class (name superclasses slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn (,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn (,objectvar ,name) ,streamvar)
         (with-slots ,(mapcar #'first slots) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))
```

24.10 跟踪继承的槽

目前的定义将会工作在很多情形下。不过，它无法处理一种相当普遍的情形，换句话说，当你的子类需要引用其自己的槽规范中所继承的槽时。例如，在当前的 `define-binary-class` 定义下，你可以像下面这样定义单个类：

```
(define-binary-class generic-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)
   (data (raw-bytes :bytes size))))
```

`data` 规范中对 `size` 的引用可以按照你预想的方式工作，因为这些表达式是在该对象的全部槽都列出的一个 `WITH-SLOTS` 的封装下读写 `data` 槽的。不过，如果你试图将上面的类像下面这样分开定义在两个槽里：

```
(define-binary-class frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)))
```

```
(define-binary-class generic-frame (frame)
  ((data (raw-bytes :bytes size))))
```

你将在编译 `generic-frame` 定义时得到一个编译期警告，然后在你试图使用它时得到一个运行期错误，因为在特化在 `generic-frame` 的 `read-object` 和 `write-object` 方法中没有以词法形态出现的变量 `size`。

你需要做的是跟踪由每个二进制类所定义的槽并将继承得到的槽包含在 `read-object` 和 `write-object` 方法的 `WITH-SLOTS` 形式中。

跟踪这类信息最简单的方法是从命名类的符号下手。如同我在第 13 章里讨论过的，每一个符号对象都有一个与之关联的属性列表，后者可通过函数 `SYMBOL-PLIST` 和 `GET` 来访问。你可以通过将任意的键值对用 `GET` 的 `SETF` 添加到一个符号的属性表中从而将这些信息与该符号相关联。举个例子，如果二进制类 `foo` 定义了三个槽——`x`、`y` 和 `z`——那么你在跟踪这一事实时可以采用下面的表达式将一个 `slots` 键添加到符号 `foo` 的属性表中，值为 `(x y z)`：

```
(setf (get 'foo 'slots) '(x y z))
```

你希望这份备忘能够作为求值 `foo` 的 `define-binary-class` 的一部分。不过，对于在何处放置这个表达式仍然不甚明了。如果你在计算宏的展开式时求值它，那么它将在你编译 `define-binary-class` 形式的时候求值，但当你以后加载了含有编译后代码的文件时就不会再求值了。另一方面，如果你将该表达式包含到展开式中，那么它将不会在编译期被求值，这意味着如果你编译了一个带有几个 `define-binary-class` 形式的文件，在编译过程中关于这些类都定义了哪些槽的信息将是不可见的，直到整个文件被加载以后才有，而这已经太晚了。

这就是我在第 20 章里讨论的特殊操作符 `EVAL-WHEN` 用来处理的问题。通过将一个形式封装在 `EVAL-WHEN` 中，你可以控制它是在编译期还是在编译后代码的加载期运行，或是在两个时期都运行。你希望在编译一个宏形式的过程中窃取一些信息并且希望在编译后的形式被加载时仍然有效，对于这样的需求，你应当把它包装在一个类似下面这样的 `EVAL-WHEN` 中：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get 'foo 'slots) '(x y z)))
```

然后把这个 `EVAL-WHEN` 包括在宏所生成的展开式中。这样，你可以通过将下列形式添加到由 `define-binary-class` 所生成的展开式中，从而保住一个二进制类和它的直接基类的槽信息：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ',name 'slots) ',(mapcar #'first slots))
  (setf (get ',name 'superclasses) ',superclasses))
```

现在你可以定义 3 个助手函数来访问这些信息。第一个函数简单地访问一个二进制类直接定义的槽。让该函数返回列表的拷贝将是个好主意，因为你不希望其他代码在二进制类已经被定义之后去修改其槽列表。

```
(defun direct-slots (name)
  (copy-list (get name 'slots)))
```

下一个函数返回从其他二进制类中继承的槽。

```
(defun inherited-slots (name)
  (loop for super in (get name 'superclasses)
        nconc (direct-slots super)))
```

```
nconc (inherited-slots super)))
```

最后，你可以定义一个函数，其返回含有所有直接定义和继承得到的槽名称的列表。

```
(defun all-slots (name)
  (nconc (direct-slots name) (inherited-slots name)))
```

当你在计算一个 `define-binary-class` 形式的展开式时，你想要生成一个含有由新类及其全部基类所定义的所有槽的名字的 `WITH-SLOTS` 形式。不过，你不能在生成展开式的时候使用 `all-slots`，因为所需的信息只有在展开式被编译以后才可用。代替地，你应当使用下面的函数，它接受传递给 `define-binary-class` 的类描述符和基类列表并用它们来计算所有新类的槽列表：

```
(defun new-class-all-slots (slots superclasses)
  (nconc (mapcan #'all-slots superclasses) (mapcar #'first slots)))
```

一旦定义了这些函数，你就可以改变 `define-binary-class` 来保存当前被定义类的信息并用已保存的基类的槽信息来生成你想要的 `WITH-SLOTS` 形式，就像下面这样：

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

24.11 带有标记的结构

有了定义那些扩展其他二进制类的二进制类的能力，你就可以定义一个新的宏用来定义那些表示“带有标记”的结构的类了。读取带有标记的结构的策略是定义一个特化的 `read-value` 方法，它知道如何读取结构开始部分的值然后使用这些值来决定实例化的哪个子类。然后它用 `MAKE-INSTANCE` 生成该类的一个实例，同时将已经读取的值作为起始参数来传递，然后再将该对象传给 `read-object`，从而允许该对象实际所属的类来决定如何读取结构的其余部分。

这个新的宏 `define-tagged-binary-class` 将看起来像是带有附加的一个 `:dispatch` 选项的 `define-binary-class`，该选项用来指定一个求值到某二进制类名的形式。这个 `:dispatch` 形式将在带有标记的类所定义的槽名称被绑定到从文件中所读取到的值的上下文中被求值。它所返回的类必须接受对应于带有标记的类所定义的槽名称的起始参数。如果 `:dispatch` 形式总是求值到该标记类的子类上，那么这个要求直接可以满足。

举个例子，假设你有一个函数 `find-frame-class`，它将一个字符串标识符映射到代表特定

类型的 ID3 帧的二进制类上，那么你可以定义一个带有标记的二进制类 `id3-frame`，像下面这样：

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

一个 `define-tagged-binary-class` 的展开式将含有一个 `DEFCLASS` 和一个就像 `define-binary-class` 的展开式那样的 `write-object` 方法，但代替了 `read-object` 方法，它将含有一个看起来像下面这样的 `read-value` 方法：

```
(defmethod read-value ((type (eql 'id3-frame)) stream &key)
  (let ((id (read-value 'iso-8859-1-string stream :length 3))
        (size (read-value 'u3 stream)))
    (let ((object (make-instance (find-frame-class id) :id id :size size)))
      (read-object object stream)
      object)))
```

由于 `define-tagged-binary-class` 和 `define-binary-class` 的展开式除了读方法以外将是相同的，所以你可以将它们的共同点分离出来放在一个助手宏 `define-generic-binary-class` 里，它接受读方法作为一个参数并将其插入到自己的展开式里。

```
(defmacro define-generic-binary-class (name (&rest superclasses) slots
                                         read-method)
  (with-gensyms (objectvar streamvar)
    `(progn
       (eval-when (:compile-toplevel :load-toplevel :execute)
         (setf (get ',name 'slots) ',(mapcar #'first slots))
         (setf (get ',name 'superclasses) ',superclasses))

       (defclass ,name ,superclasses
         ,(mapcar #'slot->defclass-slot slots))

       ,read-method

       (defmethod write-object progn ((,objectvar ,name) ,streamvar)
         (declare (ignorable ,streamvar))
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

现在你可以同时定义 `define-binary-class` 和 `define-tagged-binary-class` 来展开成一个对 `define-generic-binary-class` 的调用了。下面是一个新版本的 `define-binary-class`，当其完全展开时可以生成和之前的版本相同的代码：

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
       (defmethod read-object progn ((,objectvar ,name) ,streamvar)
         (declare (ignorable ,streamvar))
         (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
           ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))))))
```

而下面是 `define-tagged-binary-class` 的定义以及它所用到的两个新的助手函数：

```
(defun slot->binding (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(&,name (read-value ',type ,stream ,@args)))))

(defun slot->keyword-arg (spec)
```

```
(let ((name (first spec)))
  `((,as-keyword name) ,name)))

(defmacro define-tagged-binary-class (name (&rest superclasses) slots &rest
options)
  (with-gensyms (typevar objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let* ,(mapcar #'(lambda (x) (slot->binding x streamvar)) slots)
          (let ((,objectvar
            (make-instance
              ,(or (assoc :dispatch options)
                (error "Must supply :dispatch form."))
              ,(mapcan #'slot->keyword-arg slots))))
            (read-object ,objectvar ,streamvar)
            ,objectvar)))))))
```

24.12 基本二进制类型

尽管 `define-binary-class` 和 `define-tagged-binary-class` 令复合结构的定义变得简单了，但你仍然不得不手工编写用于基本数据类型的 `read-value` 和 `write-value` 方法。你可以决定保持现状，指定该库的用户必须编写适当的 `read-value` 和 `write-value` 方法来支持他们的二进制类所使用的基本类型。

不过，除了文档化如何编写合适的 `read-value/write-value` 对以外，你还可以提供一个宏来自动地做到这点。这样做的另一个优点是让 `define-binary-class` 所创建的抽象更加圆满。目前，`define-binary-class` 依赖于以特殊方式定义的 `read-value` 和 `write-value` 方法，但这只是一种实现细节。通过定义一个对基本类型生成 `read-value` 和 `write-value` 方法的宏，你可以将那些细节隐藏在你所控制的抽象层面上。如果你以后决定改变 `define-binary-class` 的实现，那么你可以改变你的基本类型定义宏来满足新的需求而无需对使用二进制格式库的代码做任何改变。

所以你应当定义最后一个宏 `define-binary-type`，它将生成用来读写代表已有类的实例的值，而不是由 `define-binary-class` 所定义的类的实例的值。

举一个简单的例子，考虑一个用在 `id3-tag` 类中的类型，一个以 ISO-8859-1 编码的定长字符串。我将如以往一样假设你的 Lisp 所使用的原生字符集是 ISO-8859-1 或它的一个超集，这样你就可以使用 `CODE-CHAR` 和 `CHAR-CODE` 来将字节和字符做相互转化了。

和以往一样，你的目标是编写一个宏，它允许你仅表达必要的用来生成所需代码的信息。在本例中，共有 4 个部分的本质信息：类型名 `iso-8859-1`；应当被 `read-value` 和 `write-value` 方法所接受的 `&key` 参数，在这里是 `length`；用来从流中做读操作的代码；用来向一个流中做写操作的代码。下面是一个含有这四部分信息的表达式：

```
(define-binary-type iso-8859-1-string (length)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in)))))
      string))
  (:writer (out string)
```

```
(dotimes (i length)
  (write-byte (char-code (char string i)) out))))
```

现在你只需要一个宏来接受上面的形式再将两个 DEFMETHOD 的形式放在一起封装到一个 PROGN 中就可以了。如果你像下面这样定义了 define-binary-type 的参数列表：

```
(defmacro define-binary-type (name (&rest args) &body spec) ...)
```

那么在宏里，参数 spec 将是一个含有读写器定义的列表。你可以随后用 ASSOC 以及标签 :reader 和 :writer 来解出 spec 中的元素，然后再用 DESTRUCTURING-BIND 来取出每个元素的 REST 部分。¹⁰

从这里开始剩下的问题只是将解出来的值插入到 read-value 和 write-value 方法的反引用模板了。

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (with-gensyms (type)
    `(progn
      ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
        `',(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
            ,@body))
      ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
        `',(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
            ,@body))))
```

注意到反引用模板是如何嵌套的：最外层的模板以反引用的 PROGN 形式开始。这个模板由符号 PROGN 和双逗号解除反引用的 DESTRUCTURING-BIND 表达式组成。这样，外层模板是通过求值 DESTRUCTURING-BIND 表达式并插入得到的值来实现的。每一个 DESTRUCTURING-BIND 表达式又含有另外的反引用模板，其用来生成被插入到外层模板的方法定义。

有了这个宏，之前给出的 define-binary-type 形式将展开成下面的代码：

```
(progn
  (defmethod read-value ((#:g1618 (eql 'iso-8859-1-string)) in &key length)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (defmethod write-value ((#:g1618 (eql 'iso-8859-1-string)) out string &key
    length)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out))))
```

当然，现在你已经让这个漂亮的宏可用来定义二进制类型的，不过它似乎还是多做了一些事。目前你应该只需要一个小的改进就可以让它在你开始使用这个库来处理诸如 ID3 标签这样的实际格式时成为相当有用的工具了。

和其他二进制格式一样，ID3 标签使用的许多基本类型都是同一个主题下的变种，例如一个、两个、三个和四个字节的无符号整数。你当然可以逐个地用 define-binary-type 来定义每个类型。或者你也可以将用来读写 n 字节无符号整数的通用算法分离成助手函数。

¹⁰ 使用 ASSOC 来解出 spec 的 :reader 和 :writer 元素可以允许 define-binary-type 的用户以任何顺序包含这些元素；如果你要求 :reader 元素必须总是第一个，那么你可以使用 (rest (first spec)) 来解出读取器，再用 (rest (second spec)) 来解出写入器。不过，只要你要求使用 :reader 和 :writer 关键字来改进 define-binary-type 形式的可读性，那么你就总是可以使用它们来解出正确的数据来。

但是假设你已经定义了一个二进制类型 `unsigned-integer`, 其接受一个 `:bytes` 参数来指定一次读写多少个字节。使用这个类型, 你可以用 `(unsigned-integer :bytes 1)` 这个类型说明符来指定一个表示单字节无符号整数的槽。但假如一个特定的二进制格式指定了许多这样类型的槽, 那么如果可以将其定义成一个代表同样类型的新类型, 比如说 `u1`, 就会很方便了。看起来容易改变 `define-binary-type` 来支持两个形式, 一个由 `:reader` 和 `:writer` 对构成的长形式, 以及一个用已有类型来定义新二进制类型的短形式。使用一个短形式的 `define-binary-type`, 你可以像下面这样定义 `u1`:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
```

它将展开成下面的代码:

```
(progn
  (defmethod read-value ((#:g161887 (eql 'u1)) #:g161888 &key)
    (read-value 'unsigned-integer #:g161888 :bytes 1))
  (defmethod write-value ((#:g161887 (eql 'u1)) #:g161888 #:g161889 &key)
    (write-value 'unsigned-integer #:g161888 #:g161889 :bytes 1)))
```

为了同时支持长短两种形式的 `define-binary-type` 调用, 你需要基于 `spec` 参数的值来做区分。如果 `spec` 是两项的, 那么它将代表一个长形式的调用, 其中的两项应当分别是 `:reader` 和 `:writer` 规范, 你可以像之前那样处理。另一方面, 如果 `spec` 只有一项, 那么这个唯一的项应当是一个类型说明符, 需要区别地进行处理。你可以使用 `ECASE` 在 `spec` 的 `LENGTH` 上做切换, 并随后解析 `spec` 来生成可分别用于长短两种形式的适当展开式。

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (ecase (length spec)
    (1
      (with-gensyms (type stream value)
        (destructuring-bind (derived-from &rest derived-args) (mklist (first
spec)))
          `(progn
            (defmethod read-value ((,type (eql ',name)) ,stream &key ,@args)
              (read-value ',derived-from ,stream ,@derived-args))
            (defmethod write-value ((,type (eql ',name)) ,stream ,value
&key ,@args)
              (write-value ',derived-from ,stream ,value ,@derived-args)))))))
    (2
      (with-gensyms (type)
        `(progn
          ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
            `'(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
                ,@body))
          ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
            `'(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
                ,@body)))))))
```

24.13 当前对象栈

你在下一章里将会用到的最后一点儿功能是在读取和写入过程中获得当前二进制对象的方式。在更一般的情况下, 当读取或写入嵌套的复合对象时, 能够获得当前正在读写的任何层面的对象将是非常有用的。感谢动态变量和 `:around` 方法, 你可以仅用几行代码来添加这一增强特性。一开始, 你应当定义一个用来保存当前正在读取或写入的对象栈的动态变量。

```
(defvar *in-progress-objects* nil)
```

然后你可以在 `read-object` 和 `write-object` 上定义 `:around` 方法来将正在被读写的对象在调用 `CALL-NEXT-METHOD` 之前推送到该变量里。

```
(defmethod read-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))

(defmethod write-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))
```

注意到你是如何重绑定 `*in-progress-objects*` 到一个头部带有新项的列表上而不是将其赋予新值的。以这种方式的话，在 `LET` 形式结束的时候，`CALL-NEXT-METHOD` 返回以后，`*in-progress-objects*` 的旧值将被恢复，从而相当于把对象从栈上弹出了。

一旦定义了这两个方法，你还可以提供两个用来获取当前进度栈中特定对象的便利的函数。函数 `current-binary-object` 将返回栈的头部，也就是 `read-object` 和 `write-object` 最近被调用的那个对象。另一个函数 `parent-of-type` 接受一个应当是某个二进制类的名字的参数并返回最近推入栈中的该类型的对象，它使用 `TYPEP` 函数来测试一个给定的对象是否为一个特定类型的实例。

```
(defun current-binary-object () (first *in-progress-objects*))

(defun parent-of-type (type)
  (find-if #'(lambda (x) (typep x type)) *in-progress-objects*))
```

这两个函数可以用在一个 `read-object` 和 `write-object` 调用的所在的动态上下文中的任何代码之内。你将在下一章里看到关于 `current-binary-object` 用法的一个例子。

现在你终于有了用来处理一个 ID3 解析库的所有工具，现在你可以进入下一章来做这件事了。

第25章 实践：一个 ID3 解析器

有了一个解析二进制数据的库以后，你就可以开始编写一些代码用来读写实际的二进制格式了，首先是 ID3 标签。ID3 标签用来在 MP3 音频文件中嵌入元数据。处理 ID3 标签将是对二进制数据处理库的一个好的测试，因为 ID3 格式是一个真正的现实世界的文件格式——工程权衡和特定设计选择的混合体，不管怎么说确实可以满足需要。由于你错过了文件共享领域的革命，下面是关于 ID3 标签是什么以及它们与 MP3 文件之间关系的简要介绍。

MP3，也称为 MPEG Audio Layer 3，是一种用来保存压缩的音频数据的格式，由 Fraunhofer IIS 的研究者们所设计并由 Moving Picture Experts Group 标准化，后者是国际标准化组织(ISO)和国际电工技术委员会(IEC)所组成的联合委员会。不过，MP3 格式本身只定义了如何保存音频数据。只要你所有的 MP3 文件都被单一的应用程序所管理，将元数据外部保存并跟踪元数据所关联的文件，那么就不会有太大的问题。不过，当人们开始在 Internet 上通过诸如 Napster 这样的文件共享平台相互传递单独的 MP3 文件时，他们很快发现需要一种方式将元数据嵌入进 MP3 文件本身。

由于 MP3 标准已经定案并且相当数量的软件和硬件已经知道如何解析已有的 MP3 格式了，所以任何在一个 MP3 文件中嵌入信息的方法都必须对于 MP3 解码器来说是不可见的。下面进入到 ID3 环节。

最初的 ID3 格式由程序员 Eric Kemp 所发明，它由连接到一个 MP3 文件结尾处的 128 个字节所构成，大多数 MP3 软件都会忽略它。它进一步包括四个 30 字符的字段，分别用于歌曲标题、专辑标题、艺术家名和一个评论；一个四字节的年份字段；以及一个单字节的风格代码。Kemp 提供了对于前 80 个风格代码的标准含义。Nullsoft 公司，也就是一个流行的 MP3 播放器 Winamp 的作者，后来又向这个列表中提供了另外 60 种风格。

这个格式易于解析但明显地带有很多局限性。它没有办法编码长度超过 30 字符的名字；它受限于 256 种风格，并且风格代码必须被所有 ID3 敏感的软件的用户所同意才行。起初甚至没有办法编码一个特定 MP3 文件的 CD 音轨号，知道另一个程序员 Michael Mutschler 提议将音轨号嵌入到评论字段中，用一个空字节与评论的其余部分隔开，这样一个已有的倾向于读取每个文本字段第一个空字符之前内容的 ID3 软件将会忽略它。Kemp 的版本现在被称为 ID3v1，而 Mutschler 的版本称为 ID3v1.1。

尽管有上述局限性，但版本 1 确实提供了对于元数据问题的一个部分解决方案，因此它们被许多 MP3 烧录程序（它们必须将 ID3 标签放进 MP3 文件里）和 MP3 播放器（它们将解出 ID3 标签

中的信息并显示给用户) 所采纳了。¹

尽管如此, 到了 1998 年, 这些限制变得越发令人难以忍受, 于是一个由 Martin Nilsson 所领导的新的小组开始了设计全新的标签模式的工作, 其成果后来被称为 ID3v2。ID3v2 格式极其灵活, 允许包含多种多样的信息, 同时几乎没有长度限制。它还利用了 MP3 格式的特定细节, 从而允许 ID3v2 标签被放置在一个 MP3 文件的开始处。

不过, ID3v2 标签在解析方面相比版本 1 标签来说是一项挑战。在本章里, 你将使用来自前一章的二进制数据解析库来开发可以读写 ID3v2 标签的代码。或者至少你将有一个合理的开始——ID3v1 太简单了, 而 ID3v2 则从完全过分工程化的角度来看又太复杂了。实现其规范中的每一处细枝末节, 尤其是当你想要支持已规范化的所有三个版本时, 将需要相当多的工作。不过, 你可以忽略掉规范中的许多很少被实际使用的特性。对于新人来说, 目前你可以忽略掉整个版本 2.4, 因为它尚未被广泛采纳并且相比版本 2.3 来说基本上只是增加了更多的不需要的灵活性。我将把注意力集中在版本 2.2 和 2.3, 因为它们都已被广泛使用并且互相之间的区别大到了足够让事情保持有趣的程度。

25.1 一个 ID3v2 标签的结构

在你开始写代码之前, 你将需要熟悉一个 ID3v2 标签的整体结构。标签以一个含有关于整个标签的信息的头部开始。这个头部的最初三个字节以 ISO-8859-1 字符集编码了字符串 “ID3”。换句话说, 它们是字节 73、68 和 51。接下来的两个字节编码了代表当前标签所符合的 ID3 规范的主版本和修订号。它们的后面又跟了一个字节, 其单独的位被视为标志位。这意味着这些单独标志的含义依赖于规范的版本。一些标志可以影响到整个标签其余部分的解析方式。所谓“主版本”实际上用来记录规范的副版本, 而“修订号”则是规范的子副版本。这样, “主版本”字段对于一个遵守 2.3.0 规范的标签来说就是 3。修订号字段总是零, 因为每一个新的 ID3v2 规范都在副版本号上跳跃, 总是将子副版本保持在零上。正如你将会看到的, 这个保存在标签的主版本字段上的值将对你解析标签其余部分的方式产生深远的影响。

标签头部的最后一个字段是一个整数, 它以四个字节进行编码但只用到每个字节的前七位, 给出了整个标签不包括头部在内的长度。在版本 2.3 标签里, 头部可能还跟有几个扩展头部字段; 否则, 标签数据的其余部分将被划分成多个帧。不同类型的帧保存不同类型的信息, 从诸如歌曲名这类简单的文本信息到嵌入的图像。每个帧以一个含有字符标识符和尺寸的头开始。在版本 2.3 中, 帧头还含有总长两字节的标志位, 以及取决于某个标志位的一个可选的单字节代码以指示帧的其余部分是如何加密的。

帧是带有标记的数据结构的一个完美的例子——为了知道如何解析一个帧的主体, 你需要读取它的头部并使用标识符来检测你正在读取的帧的类型。

ID3 标签头中没有包含关于一个标签中究竟有多少个帧的直接指示——标签头只告诉你标签有多大, 但由于许多标签都是变长的, 因此找出标签中含有的帧的数量的唯一方法就是实际去

¹ 所谓烧录 (ripping) 是将一张音乐 CD 中的某支歌曲转化成你硬盘中的一个 MP3 文件的过程。近年来大多数的烧录软件也都可以自动地从诸如 Gracenote (也就是 Compat Disc Database [CDDB]) 或 FreeDB 这些在线数据库中获取关于歌曲的信息, 然后再以 ID3 标签的形式嵌入到 MP3 文件中。

读取这些帧数据。另外，由标签头所给出的大小可能会超过帧数据的实际字节数；帧后面可能跟有足够的数量的空字节用以将标签补齐到指定的大小。这个设计使得标签编辑器可以在修改标签时无需重写整个 MP3 文件。²

因此，你将要面对的主要问题是在读取 ID3 头部时：检测你正在读取版本 2.2 还是 2.3 的标签；然后读取帧数据，并在你已经读取了标签长度范围内的所有标签或是遇到补白字节的时候停下来。

25.2 定义一个包

和目前你开发的其他库一样，你将在本章里编写的代码值得放进它自己的包里。你将需要引用到来自第 24 和 15 章的二进制数据和路径名的库，并且你也会希望导出那些构成该包公共 API 的那些函数名。下面的包定义做到了所有这些事：

```
(defpackage :com.gigamonkeys.id3v2
  (:use :common-lisp
        :com.gigamonkeys.binary-data
        :com.gigamonkeys.pathnames)
  (:export
   :read-id3
   :mp3-p
   :id3-p
   :album
   :composer
   :genre
   :encoding-program
   :artist
   :part-of-set
   :track
   :song
   :year
   :size
   :translated-genre))
```

和往常一样，你可以，并且也应该将包名中的 `com.gigamonkeys` 部分改成你自己的域。

25.3 整数类型

你可以从定义用来读写几种 ID3 格式用到的基本类型的二进制类型开始整个工作，这包括不同长度的无符号整数，以及四种字符串。

ID3 用到了编码在一到四个字节中的无符号整数。如果你第一次编写一个通用的 `unsigned-integer` 二进制类型，其中接受读取的字节数作为一个参数，那么你可以随后再用短形式的 `define-binary-type` 来定义特定的类型。通用的 `unsigned-integer` 类型看起来像下面

² 几乎所有的文件系统都提供了覆盖一个文件中已有字节的能力，但也有少数文件系统允许在一个文件的开始或中间位置添加或删除数据而无需重写文件其余部分。由于 ID3 标签通常存放在一个文件的开始处，因此为了重写一个 ID3 标签而不干扰文件的其余部分，你必须将旧标签替换成一个完全相同长度的新标签。通过在写入 ID3 标签时带有特定数量的补白，你就有机会更好地做到这点——如果新标签带有比最初标签更多的数据，你可以使用较少的补白，而如果变得更短了就使用更多的补白。

这样：

```
(define-binary-type unsigned-integer (bytes)
  (:reader (in)
    (loop with value = 0
      for low-bit downfrom (* 8 (1- bytes)) to 0 by 8 do
        (setf (ldb (byte 8 low-bit) value) (read-byte in))
      finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* 8 (1- bytes)) to 0 by 8
      do (write-byte (ldb (byte 8 low-bit) value) out))))
```

现在你可以使用短形式的 `define-binary-type` 像下面这样为 ID3 格式里用到的每种尺寸的整数分别定义一个类型：

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
(define-binary-type u2 () (unsigned-integer :bytes 2))
(define-binary-type u3 () (unsigned-integer :bytes 3))
(define-binary-type u4 () (unsigned-integer :bytes 4))
```

另一个你将需要用来读写的类型是用在头部中的 28 位值。这个值使用 28 位而非诸如 32 位这样的 8 的倍数来编码，因为一个 ID3 标签中不能含有字节 `#xff` 后跟一个前三位为 1 的字节的模式，因为这对于 MP3 解码器来说有另外的特殊含义。ID3 头部的其他字段也都不允许含有这样的字节序列，但如果你将标签尺寸编码成一个正规的 `unsigned-integer` 的话，就有可能出现问题了。为了避免这种可能性，这个尺寸被编码成只使用每个字节的底下 7 位，并让最上面一个位总是零。³

这样，它可以像一个 `unsigned-integer` 那样进行读写，除了你传给 `LDB` 的字节说明符的尺寸应当是 7 而不是 8。这种相似性表明，假如你为已有的 `unsigned-integer` 二进制类型添加一个参数 `bits-per-byte`，那么你就可以用短形式的 `define-binary-type` 直接定义出一个新类型 `id3-tag-size` 来。这个新版本的 `unsigned-integer` 和旧版本非常像，除了 `bits-per-byte` 被用在旧版本的所有硬编码了数字 8 的位置上。它看起来像下面这样：

```
(define-binary-type unsigned-integer (bytes bits-per-byte)
  (:reader (in)
    (loop with value = 0
      for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte do
        (setf (ldb (byte bits-per-byte low-bit) value) (read-byte in))
      finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte
      do (write-byte (ldb (byte bits-per-byte low-bit) value) out))))
```

那么 `id3-tag-size` 的定义就很简单了。

```
(define-binary-type id3-tag-size () (unsigned-integer :bytes 4 :bits-per-byte 7))
```

你还需要改变 `u1` 到 `u4` 的定义，像下面这样明确指定每个字节里读取 8 位：

```
(define-binary-type u1 () (unsigned-integer :bytes 1 :bits-per-byte 8))
(define-binary-type u2 () (unsigned-integer :bytes 2 :bits-per-byte 8))
(define-binary-type u3 () (unsigned-integer :bytes 3 :bits-per-byte 8))
```

³ ID3 头部后跟的帧数据也可能潜在地含有这一不合法的序列。这可以使用一种不同的模式来避免出现，通过打开标签头上的某个标记位来控制。本章中的代码并不考虑该标记位被设定的可能性；它在实际上也很少被用到。

```
(define-binary-type u4 () (unsigned-integer :bytes 4 :bits-per-byte 8))
```

25.4 字符串类型

ID3 格式中其余的无处不在的基本类型是字符串。在前一章里我讨论了在处理字符串二进制文件中的字符串时你必须考虑的一些问题，例如字符编码和字符编码方式之间的区别。

ID3 使用两种不同的字符编码，ISO 8859-1 和 Unicode。ISO 8859-1，也称为 Latin-1，是一种八位字符编码，它将 ASCII 用西欧语言中用到的字符扩展而成。换句话说，从 0-127 之间的代码点在 ASCII 和 ISO 8859-1 中映射到相同的字符上，但 ISO 8859-1 还提供了最大到 255 的其余代码点的映射。Unicode 是设计用于为世界上所有语言的几乎每一个字符提供代码点的字符编码。Unicode 是 ISO 8859-1 的超集，正如 ISO 8859-1 是 ASCII 的超集那样——从 0-255 的代码点在 ISO 8859-1 和 Unicode 中都映射到相同的字符上。(因此，Unicode 也是 ASCII 的一个超集。)

由于 ISO 8859-1 是一个 8 位字符编码，因此它使用每字符一个字节的方式进行编码。对于 Unicode 字符串来说，ID3 使用带有前导字符序标记的 UCS-2 编码方式。⁴我将很快讨论什么是一个字节序标记。

读写这两种编码方式不是问题——这不过是以不同的格式读写无符号整数罢了，而你已经写好了做这件事的代码。难点在于如何将这些数值转化成 Lisp 字符对象。

你所使用的 Lisp 实现很可能使用了 Unicode 或 ISO 8859-1 作为其内部字符编码。而由于从 0-255 之间的所有值在 ISO 8859-1 和 Unicode 中都映射到相同的字符上，所以你可以使用 Lisp 的 `CODE-CHAR` 和 `CHAR-CODE` 函数来转化两个编码中的这些值。不过，如果你的 Lisp 仅支持 ISO 8859-1，那么你将只能把前 255 个 Unicode 字符表示成 Lisp 字符。换句话说，在这样的 Lisp 实现里，如果你试图处理一个用到 Unicode 字符串并且其中含有代码点超出 255 的字符的 ID3 标签，那么你将在试图把代码点转化成一个 Lisp 字符时遇到错误。目前我将假设你正在使用一个基于 Unicode 的 Lisp 或者你不会处理任何含有超出 ISO 8859-1 范围的字符的文件。

字符串编码时所带来的另一个问题是得知需要将多少个字节解释成字符数据。ID3 使用了我在前一章里提到的两种策略——一些字符串是采用空字符结尾的，而另一些字符串出现在你可以决定读取多少个字节的位置上，要么是因为那个位置上的字符串总是具有相同的长度，要么是因为字符串处在一个总长度已知的符合结构的结尾处。不过，需要注意的是，字节的数量不一定与字符串中字符的数量相同。

考虑了所有这些特征，ID3 格式使用四种方式来读写字符串——由两种字符编码方式和两种字符串数据分界方式交叉而成。

很明显，读写字符串的很多业务逻辑将会非常相似。因此，你可以从定义两种二进制类型开始，一种用于读取指定（字符）长度的字符串，而另一种用来读取带有终止符的字符串。这两种类型利用 `read-value` 和 `write-value` 的类型参数的方法是由另外的代码提供的；你可以让字符类型来读取一个关于其类型的参数。这种技术你在本章里还会多次看到。

⁴ 在 ID3v2.4 中，UCS-2 被替换成几乎等价的 UTF-16，并且 UTF-16BE 和 UTF-8 被增加为附加的编码方式。

```
(define-binary-type generic-string (length character-type)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (read-value character-type in)))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-value character-type out (char string i)))))

(define-binary-type generic-terminated-string (terminator character-type)
  (:reader (in)
    (with-output-to-string (s)
      (loop for char = (read-value character-type in)
            until (char= char terminator) do (write-char char s))))
  (:writer (out string)
    (loop for char across string
          do (write-value character-type out char)
          finally (write-value character-type out terminator))))
```

有了这些类型，读取 ISO 8859-1 字符串就很容易了。由于你传递给一个 `generic-string` 的 `read-value` 和 `write-value` 方法的 `character-type` 参数必须是一个二进制类型的名字，因此你需要定义一个 `iso-8859-1-char` 二进制类型。这也给了你一个很好的位置用来放置一些一致性检查的代码，检查你所读写字符的代码点。

```
(define-binary-type iso-8859-1-char ()
  (:reader (in)
    (let ((code (read-byte in)))
      (or (code-char code)
          (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (if (<= 0 code #xff)
          (write-byte code out)
          (error
            "Illegal character for iso-8859-1 encoding: character: ~c with code:
~d"
            char code)))))
```

现在使用 `define-binary-type` 的短形式来定义 ISO 8859-1 字符串类型就很简单了，如下所示：

```
(define-binary-type iso-8859-1-string (length)
  (generic-string :length length :character-type 'iso-8859-1-char))

(define-binary-type iso-8859-1-terminated-string (terminator)
  (generic-terminated-string :terminator terminator
                            :character-type 'iso-8859-1-char))
```

读取 UCS-2 字符串只是稍微复杂一些。其复杂性源自你可以用两种方式来编码一个 UCS-2 代码点：最重要的字节优先 (big-endian) 或最不重要的字节优先 (little-endian)。因此 UCS-2 字符串以两个附加的字节开始，称为字节序标记 (byte order mark)，它由以 big-endian 或 little-endian 形式编码的数值 `#xfeff` 所构成。当读取一个 UCS-2 字符串时，你需要读取这个字节序标记，然后根据其值来读取 big-endian 或 little-endian 的字符。这样，你将需要两个不同的 UCS-2 字符类型。但是你只需要一个版本的一致性检查代码，因此你可以像下面这样来定义一个参数化的二进制类型：

```
(define-binary-type ucs-2-char (swap)
  (:reader (in)
    (let ((code (read-value 'u2 in)))
      (when swap (setf code (swap-bytes code)))
      (or (code-char code) (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (unless (<= 0 code #xffff)
        (error "Illegal character for ucs-2 encoding: ~c with char-code: ~d"
               char code))
      (when swap (setf code (swap-bytes code)))
      (write-value 'u2 out code))))
```

其中的 `swap-bytes` 函数可以像下面这样来定义，它利用了 `LDB` 函数可被 `SETF` 和 `ROTATEF` 的特点：

```
(defun swap-bytes (code)
  (assert (<= code #xffff))
  (rotatef (ldb (byte 8 0) code) (ldb (byte 8 8) code))
  code)
```

使用 `ucs-2-char`，你可以定义两个用作通用字符串函数的 `character-type` 参数的字符类型。

```
(define-binary-type ucs-2-char-big-endian () (ucs-2-char :swap nil))

(define-binary-type ucs-2-char-little-endian () (ucs-2-char :swap t))
```

然后你需要一个函数，它基于字节序标记的值来返回具体所使用的字符类型。

```
(defun ucs-2-char-type (byte-order-mark)
  (ecase byte-order-mark
    (#xefff 'ucs-2-char-big-endian)
    (#xffffe 'ucs-2-char-little-endian)))
```

现在你可以用于 UCS-2 编码字符串的长度和终止符定界的字符串类型了，它们将读取字节序标记并用这个标记来决定究竟向 `read-value` 和 `write-value` 的 `character-type` 参数传递哪个 UCS-2 字符变种。其余唯一的亮点是你需要将代表字节个数的 `length` 参数根据字节序标记转化成需要读取的字符数。

```
(define-binary-type ucs-2-string (length)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in))
          (characters (1- (/ length 2))))
      (read-value
       'generic-string in
       :length characters
       :character-type (ucs-2-char-type byte-order-mark)))
  (:writer (out string)
    (write-value 'u2 out #xefff)
    (write-value
     'generic-string out string
     :length (length string)
     :character-type (ucs-2-char-type #xefff)))))

(define-binary-type ucs-2-terminated-string (terminator)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in)))
      (read-value
       'generic-terminated-string in
```

```

:terminator terminator
:character-type (ucs-2-char-type byte-order-mark)))
(:writer (out string)
  (write-value 'u2 out #xefff)
  (write-value
   'generic-terminated-string out string
   :terminator terminator
   :character-type (ucs-2-char-type #xefff)))

```

25.5 ID3 标签头

基本类型的工作完成以后，现在你可以切换到更高层次的视角并开始定义二进制类来表示 ID3 标签整体和单独的帧了。

如果你是首次接触 ID3v2.2 规范，那么你将看到标签的基本结构是下面的这个头：

```

ID3/file identifier      "ID3"
ID3 version             $02 00
ID3 flags               %xx000000
ID3 size                4 * %xxxxxxxx

```

后跟帧数据和补白。由于你已经定义了读写头部所有字段的二进制类型，因此定义一个类来读取一个 ID3 标签的整个头部只是将已有的成果合并在一起罢了。

```

(define-binary-class id3-tag ()
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)))

```

如果你手头有一些 MP3 文件的话，那么你可以测试目前的这些代码，同时也看看你的 MP3 都含有哪些版本的 ID3 标签。首先你编写一个函数从一个文件的开始处读取刚刚定义的这个 id3-tag。不过，请注意 ID3 标签不一定出现在一个文件的开始处，尽管目前它们总是这样的。为了在一个文件的其他位置上找到 ID3 标签，你可以在文件中搜索字节序列 73、68、51（换句话说，字符串“ID3”）。⁵ 目前你可以简单地假设这些标签总是出现在文件的开始处。

```

(defun read-id3 (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (read-value 'id3-tag in)))

```

在这个函数的基础上你可以构造一个函数，它接受一个文件名并打印出连同文件名在内的标签中的信息。

```

(defun show-tag-header (file)
  (with-slots (identifier major-version revision flags size) (read-id3 file)
    (format t "~a ~d.~d ~8,'0b ~d bytes -- ~a~%" 
            identifier major-version revision flags size (enough-namestring
            file))))

```

它可以打印出类似下面这样的输出：

⁵ ID3 格式的 2.4 版也支持在一个标签的结尾处放置一个脚标，这使得一个附加在文件结尾处的标签可以更容易地被找到。

```
ID3V2> (show-tag-header "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
ID3 2.0 00000000 2165 bytes -- Kitka/Wintersongs/02 Byla Cesta.mp3
NIL
```

当然，为了检测你的 MP3 库里哪个版本的 ID3 是最普遍的，如果可以有一个函数能返回一个给定目录下所有 MP3 文件的汇总将是更有用的。你可以用第 15 章里定义的 `walk-directory` 函数轻松地写出这个函数。首先定义一个助手函数来测试一个给定的文件名是否带有 `.mp3` 扩展名。

```
(defun mp3-p (file)
  (and
    (not (directory-pathname-p file))
    (string-equal "mp3" (pathname-type file))))
```

然后你可以将 `show-tag-header`、`mp3-p` 和 `walk-directory` 组合起来，打印出给定目录下每个 MP3 文件的 ID3 头的汇总。

```
(defun show-tag-headers (dir)
  (walk-directory dir #'show-tag-header :test #'mp3-p))
```

不过，如果你有许多 MP3 文件，你可能只想知道你的 MP3 收藏中每个版本的 ID3 标签分别有多少个。为了得到这个信息，你可以写一个像下面这样的函数：

```
(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)))
      versions)))
```

另一个你将在第 29 章里用到的函数是一个用来测试给定文件是否以一个 ID3 标签开始的函数，你可以像下面这样来定义它：

```
(defun id3-p (file)
  (with-open-file (in file)
    (string= "ID3" (read-value 'iso-8859-1-string in :length 3))))
```

25.6 ID3 帧

如同我之前所讨论的，一个 ID3 标签从整体上被划分成了多个帧。每个帧都具有类似于整个标签的内部结构。每个帧都以一个指示了该帧类型和字节长度的头开始。帧头的结构在 ID3 格式的版本 2.2 和版本 2.3 之间稍微有些变化，而最终你将不得不同时处理两种形式。刚开始，你可以集中在解析版本 2.2 的帧上。

一个 2.2 帧头由编码了一个三字符 ISO 8859-1 字符串的 3 个字节和一个三字节的无符号整数所构成，后者指定了该帧的字节长度，其中不包括 6 字节的头。字符串表明该帧的类型，从而决定了你解析帧长度后面其他数据的方式。这正好是你定义的 `define-tagged-binary-class` 宏所适用的场合。你可以定义一个带有标签的类来读取帧头并随后使用一个从 ID 映射到类名的函数派发到适当的具体类上。

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

现在你可以开始实现具体的帧类了。不过，规范里定义了许多的帧类——版本 2.2 中共有 63 个，后续版本里还有更多。即便将那些共享了同样的基本结构的帧类型视为等价的，最后你仍然可以在版本 2.2 中得到 24 种不同的帧类型。但是它们中只有很少的一些是被“广泛使用的”。因此，与其立即开始为每个帧类型定义类，你还不如从编写一个通用帧类开始，这个类可以让你读取一个标签中的帧而无需解析帧里面的数据。这将给你一种方式来找出你想要处理的 MP3 中实际上都有哪些帧。反正你最终也需要这样一个类，因为规范中允许实验性帧的存在，对于这些帧你可以无需解析地读取它们。

由于帧头中的尺寸字段可以告诉你一个帧究竟有多少个字节，因此你可以定义一个 generic-frame 类，它扩展了 id3-frame 并增加了一个字段 data 用来保存一个字节数组。

```
(define-binary-class generic-frame (id3-frame)
  ((data (raw-bytes :size size))))
```

其中数据字段的类型 raw-bytes 只需要用来保存一个字节数组。你可以像下面这样来定义它：

```
(define-binary-type raw-bytes (size)
  (:reader (in)
    (let ((buf (make-array size :element-type '(unsigned-byte 8))))
      (read-sequence buf in)
      buf))
  (:writer (out buf)
    (write-sequence buf out)))
```

现阶段你将希望所有的帧都被读取为 generic-frames，所以你可以定义用在 id3-frame 的 :dispatch 表达式中的 find-frame-class 函数，让它总是返回 generic-frame，无论帧的 id 是什么。

```
(defun find-frame-class (id)
  (declare (ignore id))
  'generic-frame)
```

现在你需要修改 id3-tag，让其可以读取头部字段后面的那些帧。读取帧数据的唯一难点是：尽管标签头告诉了你该标签有多少字节，但这个数值包括了跟在帧数据之后的补白。由于标签头无法告诉你该标签含有多少个帧，因此知道你遇到补白的唯一办法就是在你期待一个帧标识符的时候却找到了一个空字节。

为了处理这点，你可以定义一个二进制类型 id3-frames，它负责读取一个标签的其余部分，创建代表它所发现的所有帧的对象，并且跳过任何补白。这个类型接受标签尺寸作为一个参数，该参数可用来避免读取到超过标签结尾的位置上。但是读取代码也将需要检测跟在帧数据之后的补白的开始处。因此不能直接在 id3-frames 的 :reader 部分直接调用 read-value，你应当使用一个函数 read-frame，你将定义它在检测到补白时返回 NIL，而在其他时候返回一个使用 read-value 来读取到的 id3-frame 对象。假设你已定义了 read-frame 并让它在前一个帧的结尾处读取额外的一个字节来检测补白的开始，那么你可以像下面这样来定义 id3-frame 二进制类型：

```
(define-binary-type id3-frames (tag-size)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
```

```

for frame = (read-frame in)
while frame
do (decf to-read (+ 6 (size frame)))
collect frame
finally (loop repeat (1- to-read) do (read-byte in))))
(:writer (out frames)
(loop with to-write = tag-size
for frame in frames
do (write-value 'id3-frame out frame)
(decf to-write (+ 6 (size frame)))
finally (loop repeat to-write do (write-byte 0 out))))))

```

你可以使用这个类型来为 id3-tag 增加一个帧槽。

```

(define-binary-class id3-tag ()
  ((identifier      (iso-8859-1-string :length 3))
   (major-version  u1)
   (revision       u1)
   (flags          u1)
   (size           id3-tag-size)
   (frames         (id3-frames :tag-size size))))))

```

25.7 检测标签补白

现在剩下的就只是实现 `read-frame` 了。这有一点儿麻烦，因为实际从流中读取字节的代码位于 `read-frame` 的数层以下。

你真正想要在 `read-frame` 中做的是读取一个字节并在它为空时返回 `NIL`，否则使用 `read-value` 来读取一个帧。不幸的是，如果你在 `read-frame` 中读取了这个字节，那么它在被 `read-value` 读取时就不再可用了。⁶

看起来这是一个使用状况系统的好机会——你可以在从流中进行读取的底层代码中检查空字节并在你读到一个空字节时抛出一个状况；`read-frame` 随后可以处理该状况并在读取更多字节以前将栈回退。这个方法不但是对于检测标签的补白开始处的一个优美的解决方案，还是一个将状况系统用于错误处理之外目的的例子。

你可以从定义一个状况类型开始，它将被底层代码所汇报并被上层代码所处理。这个状况不需要任何槽——你只需要一个可区分的状况类来确保没有其他的代码可能抛出或处理它即可。

```
(define-condition in-padding () ())
```

接下来你需要定义一个二进制类型，其 `:reader` 部分读取指定数量的字节，它先读一个字节并在该字节为空时抛出一个 `in-padding` 状况，否则继续按照 `iso-8859-1-string` 来读取其余的字节并将得到的结果与前面读取的第一个字节组合起来。

```

(define-binary-type frame-id (length)
  (:reader (in)
    (let ((first-byte (read-byte in)))
      (when (= first-byte 0) (signal 'in-padding))
      (let ((rest (read-value 'iso-8859-1-string in :length (1- length))))
        (concatenate

```

⁶ 字符流支持两个函数，`PEEK-CHAR` 和 `UNREAD-CHAR`，这两个函数中的任何一个都将是对于该问题的完美解决方案，但是二进制流不支持任何等价的函数。

```
'string (string (code-char first-byte)) rest))))  
(:writer (out id)  
  (write-value 'iso-8859-1-string out id :length length)))
```

如果你重定义了 `id3-frame`, 使其 `id`槽的类型从 `iso-8859-1-string` 变成 `frame-id`, 那么状况将在每次 `id3-frame` 的 `read-value` 方法读到一个空字节而非每个帧开始读取时抛出。

```
(define-tagged-binary-class id3-frame ()  
  ((id (frame-id :length 3))  
   (size u3))  
  (:dispatch (find-frame-class id)))
```

现在 `read-frame` 需要做的只是将一个对 `read-value` 的调用包装在一个 `HANDLER-CASE` 中, 后者处理 `in-padding` 状况并返回 `nil`。

```
(defun read-frame (in)  
  (handler-case (read-value 'id3-frame in)  
    (in-padding () nil)))
```

一旦定义了 `read-frame`, 现在你就可以读取一个完整的版本 2.2 的 ID3 标签了, 其中的帧将用 `generic-frame` 的实例来表示。在“你实际需要哪些帧?”那一节里, 你将在 REPL 中做一些实验来检测你需要实现的帧类。但首先让我们添加对版本 2.3 的 ID3 标签的支持。

25.8 支持 ID3 的多个版本

目前, `id3-tag` 是用 `define-binary-class` 定义的, 但如果你想要支持多个版本的 ID3, 那么使用一个 `define-tagged-binary-class` 将更加合理, 因为它可以派发在 `major-version` 的值上。看起来所有版本的 ID3v2 直到 `size` 字段都具有相同的结构, 因此你可以像下面这样来定义一个带有标签的二进制类, 其定义了基本的结构并派发到适当的版本相关子类上。

```
(define-tagged-binary-class id3-tag ()  
  ((identifier (iso-8859-1-string :length 3))  
   (major-version u1)  
   (revision u1)  
   (flags u1)  
   (size id3-tag-size))  
  (:dispatch  
   (ecase major-version  
     (2 'id3v2.2-tag)  
     (3 'id3v2.3-tag))))
```

版本 2.2 和版本 2.3 的标签在两方面上有所区别。首先, 一个版本 2.3 标签的头部可能被至多四个可选的扩展头部字段所扩展, 这可以通过 `flags` 字段的值来检测到。其次, 帧格式在版本 2.2 和版本 2.3 之间发生了变化, 这意味着你将使用不同的类来表示版本 2.2 的帧和对应的版本 2.3 的帧。

由于新的 `id3-tag` 类基于你最初为了表示版本 2.2 标签所写的那个, 所以新的 `id3v2.2-tag` 类的定义比较简单也就不奇怪了, 它继承了来自新的 `id3-tag` 类的大部分槽并添加了一个缺失的槽 `frames`。由于版本 2.2 和版本 2.3 使用了不同的帧格式, 所以你必须将 `id3-frames` 类型改成根据所读取的帧类型进行参数化选择的形式。目前, 假设你将通过为 `id3-frames` 类型描述符添加一个 `:frame-type` 参数来做到这点, 如下所示:

```
(define-binary-class id3v2.2-tag (id3-tag)
  ((frames (id3-frames :tag-size size :frame-type 'id3v2.2-frame))))
```

id3v2.3 类由于带有可选的字段因此会稍微更加复杂一些。4 个可选字段中的前 3 个将在 `flag` 的第 6 位被设置时包含在标签中。它们包括一个四字节的整数用来指定扩展头的大小，两个字节的标志位，以及另一个四字节的整数用来指定标签中含有多少个字节的补白。⁷第 4 个可选字段，当第 15 个扩展的头标志位被设置时会被包含进来，它是标签其余部分的一个四字节的循环冗余校验（CRC）。

二进制数据处理库没有提供对于二进制类中的可选字段的任何特别的支持，但是看来正规的参数化二进制类型就足够好了。你可以使用一个类型的名字和一个代表是否实际读写该类型的变量来参数化地定义这个新类型。

```
(define-binary-type optional (type if)
  (:reader (in)
    (when if (read-value type in)))
  (:writer (out value)
    (when if (write-value type out value))))
```

使用 `if` 作为参数的名字可能看起来有些奇怪，但它使得这个 `optional` 类型描述符变得更加可读了。举个例子，下面是使用了 `optional` 槽的 `id3v2.3-tag` 的定义：

```
(define-binary-class id3v2.3-tag (id3-tag)
  ((extended-header-size (optional :type 'u4 :if (extended-p flags)))
   (extra-flags          (optional :type 'u2 :if (extended-p flags)))
   (padding-size         (optional :type 'u4 :if (extended-p flags)))
   (crc                 (optional :type 'u4 :if (crc-p flags extra-flags)))
   (frames              (id3-frames :tag-size size :frame-type 'id3v2.3-frame))))
```

其中 `extended-p` 和 `crc-p` 是用来测试特定标志位是否被传递的助手函数。为了测试一个整数中的个别位是否被设置了，你可以使用 `LOGBITP`，另一个位操作函数。它接受一个索引和一个整数并在该整数中的指定位被设置时返回真。

```
(defun extended-p (flags) (logbitp 6 flags))

(defun crc-p (flags extra-flags)
  (and (extended-p flags) (logbitp 15 extra-flags)))
```

和版本 2.2 的标签类一样，帧槽被定义为类型 `id3-frames`，其中帧类型的名字作为参数传递。尽管如此，你需要对 `id3-frames` 和 `read-frame` 做一些小的改动以使其支持额外的 `frame-type` 参数。

```
(define-binary-type id3-frames (tag-size frame-type)
  (:reader (in)
    (loop with to-read = tag-size
          while (plusp to-read)
          for frame = (read-frame frame-type in)
          while frame
          do (decf to-read (+ (frame-header-size frame) (size frame)))
          collect frame
          finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
```

⁷ 如果一个标签带有扩展的头部，那么你可以用这个值来检测帧数据的结束位置。不过，如果这个扩展的头部没有使用，那么你将不得不继续使用老方法，因此不值得添加新代码来以不同的方式来做这件事。

```

for frame in frames
do (write-value frame-type out frame)
(decf to-write (+ (frame-header-size frame) (size frame)))
finally (loop repeat to-write do (write-byte 0 out)))))

(defun read-frame (frame-type in)
(handler-case (read-value frame-type in)
(in-padding () nil)))

```

改动发生在对 `read-frame` 和 `write-value` 的调用中，这里你需要传递 `frame-type` 参数，并且在计算帧大小的时候，你需要使用一个函数 `frame-header-size` 来代替字面数值 6，因为帧头的大小在版本 2.2 和版本 2.3 之间发生了改变。由于该函数在结果上的区别取决于帧的类，所以像下面这样将其定义成一个广义函数是合理的：

```
(defgeneric frame-header-size (frame))
```

下一节在你定义了新的帧类以后，你将在该广义函数上定义必要的方法。

25.9 版本化的帧基础类

之前你定义了单一的基础类用于所有的帧类型，现在你将需要两个类，`id3v2.2-frame` 和 `id3v2.3-frame`。其中 `id3v2.2-frame` 类将和最初的 `id3-frame` 类完全相同。

```

(define-tagged-binary-class id3v2.2-frame ()
  ((id (frame-id :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))

```

另一方面，`id3v2.3-frame` 需要更多的修改。帧标识符和尺寸字段在版本 2.3 里各自从 3 个字节增加到 4 个字节，而另有两个字节的标志位被添加进来。另外，和版本 2.3 的标签一样，帧可以含有可选字段，具体由 3 个帧标志位的值来控制。⁸考虑到这些变化，你可以像下面这样定义版本 2.3 的帧基础类以及相关的助手函数：

```

(define-tagged-binary-class id3v2.3-frame ()
  ((id              (frame-id :length 4))
   (size            u4)
   (flags           u2)
   (decompressed-size (optional :type 'u4 :if (frame-compressed-p flags)))
   (encryption-scheme (optional :type 'u1 :if (frame-encrypted-p flags)))
   (grouping-identity (optional :type 'u1 :if (frame-grouped-p flags))))
  (:dispatch (find-frame-class id)))

(defun frame-compressed-p (flags) (logbitp 7 flags))

(defun frame-encrypted-p (flags) (logbitp 6 flags))

(defun frame-grouped-p (flags) (logbitp 5 flags))

```

有了这两个函数，现在你可以实现广义函数 `frame-header-size` 上的方法了。

⁸ 这些标志位，除了控制是否包含可选字段以外，还可以影响标签中其余部分的解析方式。特别地，如果第七个标志位被设定，那么实际的帧数据将使用 zlib 压缩算法进行压缩，而如果第六个标志位被设定，那么数据将被加密。实践中这些选项很少出现，但如果真的出现的话，目前你只能忽略它们。不过如果你打算实现一个产品级的 ID3 库，那么就不得不涉及到这些领域了。一个简单的不完整解决方案是改变 `find-frame-class` 来接受第二个参数并向其传递所有标志位；如果帧被压缩或加密了，那么你应当实例化一个通用帧来保存数据。

```
(defmethod frame-header-size ((frame id3v2.2-frame)) 6)
(defmethod frame-header-size ((frame id3v2.3-frame)) 10)
```

版本 2.3 帧中的可选字段并不在这些计算中作为帧头的一部分而计入，因为它们已经被包括在帧的 `size` 值中了。

25.10 版本化的具体帧类

在最初的定义中，`id3-frame` 是 `generic-frame` 的子类。但现在 `id3-frame` 已经被替换成了两个版本相关的基础类，`id3v2.2-frame` 和 `id3v2.3-frame`。所以，你需要定义两个新版本的 `generic-frame`，为每个基础类定义一个。定义这些类的一种方法将如下所示：

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame)
  ((data (raw-bytes :size size))))
(define-binary-class generic-frame-v2.3 (id3v2.3-frame)
  ((data (raw-bytes :size size))))
```

不过，这里面不太好的一点是这两个类除了基类以外其余部分都相同。在本例中由于它们只有唯一的附加字段，所以看起来还不算太坏。但如果你将这种思路用在其他具体的帧类上，尤其是那些带有更复杂的内部结构但在两个 ID3 版本上却又完全相同的帧类，那么这些重复定义将浪费很多笔墨。

另一种你实际应当采用的思路是，将 `generic-frame` 类定义为一个合成类 (`mixin`)：一个用来作为基类的类，它和一个版本相关的基类可以共同使用来产生一个具体的版本相关的帧类。这种思路唯一的难点是，如果 `generic-frame` 没有扩展任何一个帧基础类的化，那么你就无法在其定义中访问 `size` 槽。所以你必须使用我在前一章结尾处讨论的 `current-binary-object` 函数来访问你正在读写的对象并将其传递给 `size`。并且你需要考虑到整个帧的尺寸在字节数上的区别，尤其当版本 2.3 的帧里含有任何可选字段时。因此，你需要定义一个广义函数 `data-bytes` 以及相应的在版本 2.2 和版本 2.3 的帧下都可以正确工作的方法。

```
(define-binary-class generic-frame ()
  ((data (raw-bytes :size (data-bytes (current-binary-object))))))

(defgeneric data-bytes (frame))

(defmethod data-bytes ((frame id3v2.2-frame))
  (size frame))

(defmethod data-bytes ((frame id3v2.3-frame))
  (let ((flags (frame)))
    (- (size frame)
       (if (frame-compressed-p flags) 4 0)
       (if (frame-encrypted-p flags) 1 0)
       (if (frame-grouped-p flags) 1 0))))
```

然后你可以扩展版本相关的基础类和 `generic-frame` 类，来定义出版本相关的通用帧类。

```
(define-binary-class generic-frame-v2.2 (id3v2.2-frame generic-frame) ())
(define-binary-class generic-frame-v2.3 (id3v2.3-frame generic-frame) ())
```

有了这些类的定义，现在你可以重定义 `find-frame-class` 函数，根据标识符的长度来返回正确的版本化的类了。

```
(defun find-frame-class (id)
  (ecase (length id)
    (3 'generic-frame-v2.2)
    (4 'generic-frame-v2.3)))
```

25.11 你实际需要哪些帧？

一旦有了使用通用帧来同时读取版本 2.2 和版本 2.3 标签的能力，那么你就可以开始实现那些代表你所关心的特定帧的类了。不过，在你就此深入下去之前，你应该停下来先思考一下究竟哪些帧是你所关心的，因为正如我之前所提到的，ID3 标签规范中指定了许多几乎从不使用的帧。当然，你所关心的帧取决于你正在编写哪种类型的应用。如果你最关心的是从已有的 ID3 标签中解出信息，那么你只需实现那些含有你所关心的信息的帧所对应的类即可。另一方面，如果你打算编写一个 ID3 标签编辑器，那么你可能需要实现所有的帧。

与其猜测哪些帧将会是最有用的，还不如使用已有的代码在 REPL 中实际测试一下，找出在你所拥有的 MP3 中哪些帧被实际用到了。你需要从一个 `id3-tag` 的实例开始，这可以通过 `read-id3` 函数得到。

```
ID3V2> (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
#<ID3V2.2-TAG @ #x727b2912>
```

由于你可能会多次用到这个对象，所以最好把它保存在一个变量里。

```
ID3V2> (defparameter *id3*
           (read-id3 "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
```

现在你可以看到，比如说它有多少个帧。

```
ID3V2> (length (frames *id3*))
11
```

看来并不是很多——让我们具体看看它们是什么。

```
ID3V2> (frames *id3*)
(#<GENERIC-FRAME-V2.2 @ #x72dabdda> #<GENERIC-FRAME-V2.2 @ #x72dabec2>
 #<GENERIC-FRAME-V2.2 @ #x72dabfa2> #<GENERIC-FRAME-V2.2 @ #x72dac08a>
 #<GENERIC-FRAME-V2.2 @ #x72dac16a> #<GENERIC-FRAME-V2.2 @ #x72dac24a>
 #<GENERIC-FRAME-V2.2 @ #x72dac32a> #<GENERIC-FRAME-V2.2 @ #x72dac40a>
 #<GENERIC-FRAME-V2.2 @ #x72dac4f2> #<GENERIC-FRAME-V2.2 @ #x72dac632>
 #<GENERIC-FRAME-V2.2 @ #x72dac7b2>)
```

好吧，几乎看不到什么有用的信息。你其实更想知道这些帧都是什么类型的。换句话说，你想知道这些帧的 ID，这可以通过下面这样一个简单的 `MAPCAR` 来实现：

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

如果你在 ID3v2.2 规范中查找这些标识符，那么你将发现所有那些带有字母 T 开头标识符的帧都是文本信息帧并且都具有相似的结构。而 COM 是评论帧的标识符，其结构也跟文本信息帧相

似。这里所辨认出的一些特定的文本信息帧其实是用来表示歌曲标题、艺术家、专辑、音轨、歌曲集的部分，年份，风格，以及编码程序的帧。

当然，这还只是一个 MP3 文件。其他文件里也许还用到了其他的帧。要想全部找出来并不难。首先定义一个函数将前面的 MAPCAR 表达式和一个对 `read-id3` 的调用组合起来，再将所有这些封装在一个 `DELETE-DUPLICATES` 中以保证结果的简洁性。你应当在 `DELETE-DUPLICATES` 中使用一个值为 `#'string=` 的 `:test` 参数以指定当两个元素是相同的字符串时被视为等价。

```
(defun frame-types (file)
  (delete-duplicates (mapcar #'id (frames (read-id3 file))) :test #'string=))
```

这应该得到和之前一样的结果，只不过当该函数用在相同的文件名时每个标识符只有一个。

```
ID3V2> (frame-types "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

然后你可以使用第 15 章里的 `walk-directory` 函数与 `mp3-p` 一起来找出一个目录下的每个 MP3 文件并将在这些文件上调用 `frame-types` 得到的结果组合在一起。回顾 `NUNION` 是 `UNION` 函数的回收性版本；由于 `frame-types` 对于每个文件都会建立新的列表，所以使用它是安全的。

```
(defun frame-types-in-dir (dir)
  (let ((ids ()))
    (flet ((collect (file)
              (setf ids (nunion ids (frame-types file) :test #'string=))))
      (walk-directory dir #'collect :test #'mp3-p)
      ids)))
```

现在传给它一个目录的名字，然后它将告诉你该目录下的所有 MP3 文件总计用到的标识符的集合。根据你的 MP3 文件规模，该函数可能用掉几秒的时间，但最后你将很可能得到类似下面这样的东西：

```
ID3V2> (frame-types-in-dir "/usr2/mp3/")
("TCOM" "COMM" "TRCK" "TIT2" "TPE1" "TALB" "TCP" "TT2" "TP1" "TCM"
 "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM")
```

其中的四字母标识符是版本 2.2 标识符在版本 2.3 中的等价物。由于保存在这些帧中的信息正是你将在第 27 章里所需要的，因此只为实际用到的帧实现具体的类是比较合理的，换句话说，就是下面两节里将要讨论的文本信息帧和评论帧。如果你以后决定还要支持其他的帧类型，那么无非就是将其 ID3 规范转化成适当的二进制类定义了。

25.12 文本信息帧

所有的文本信息帧都有两个字段所组成：一个单字节用来指示该帧所采用的字符串编码方式，以及一个编码在帧的其余字节中的字符串。如果编码方式字节为 0，那么字符串将用 ISO 8859-1 来编码；如果该字节为 1，那么字符串将是一个 UCS-2 字符串。

你已经定义了代表四种不同类型字符串的二进制类型——两种不同的编码方式，其中每个分别使用不同的字符串定界方法。尽管如此，`define-binary-class` 并没有提供基于对象中的其他值来检测所要读取的值类型的直接支持。相反，你可以定义一个二进制类型，它接受你所传递的编码方式字节的值然后读写对应类型的字符串。

一旦你定义出了这样一个类型，你也可以定义它来接受两个参数 :length 和 :terminator，并通过具体指定了那个参数来选择正确的字符串类型。为了实现这个新类型，你必须首先定义一些助手函数。前两个函数可以根据编码方式字节来返回对应的字符串类型的名字。

```
(defun non-terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-string)
    (1 'ucs-2-string)))

(defun terminated-type (encoding)
  (ecase encoding
    (0 'iso-8859-1-terminated-string)
    (1 'ucs-2-terminated-string)))
```

然后 string-args 函数使用编码方式字节、长度和终止符来决定在 id3-encoded-string 的 :reader 和 :writer 中传递给 read-value 和 write-value 的参数。string-args 的 length 和 terminator 参数两者之一应当总是 NIL。

```
(defun string-args (encoding length terminator)
  (cond
    (length
      (values (non-terminated-type encoding) :length length))
    (terminator
      (values (terminated-type encoding) :terminator terminator))))
```

有了这些助手函数，id3-encoded-string 的定义就很简单了。一个需要注意的细节是用在 read-value 和 write-value 调用中的关键字——无论 :length 还是 :terminator 都只是由 string-args 所返回的数据。尽管参数列表中的关键字几乎总是字面关键字，但不必总是如此。

```
(define-binary-type id3-encoded-string (encoding length terminator)
  (:reader (in)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (read-value type in keyword arg)))
  (:writer (out string)
    (multiple-value-bind (type keyword arg)
        (string-args encoding length terminator)
      (write-value type out string keyword arg))))
```

现在你可以定义一个名为 text-info 的合成类了，就像你之前定义的 generic-frame 那样。

```
(define-binary-class text-info-frame ()
  ((encoding ul)
   (information (id3-encoded-string :encoding encoding :length (bytes-left
     1)))))
```

正如当你定义 generic-frame 时需要访问帧的大小，在本例中为了计算传递给 id3-encoded-string 的 :length 参数也需要同样的信息。由于你在接下来定义的其他类里也需要做类似的计算，所以最好定义另一个助手函数 bytes-left，它使用 current-binary-object 来得到该帧的大小。

```
(defun bytes-left (bytes-read)
  (- (size (current-binary-object)) bytes-read))
```

现在，就像你定义 generic-frame 合成类一样，你可以使用最少的重复代码来定义两个版本相关的具体类。

```
(define-binary-class text-info-frame-v2.2 (id3v2.2-frame text-info-frame) ())
(define-binary-class text-info-frame-v2.3 (id3v2.3-frame text-info-frame) ())
```

为了启用这些类，你需要修改 `find-frame-class`，当 ID 表明该帧是一个文本信息帧时返回适当的类名，换句话说，当 ID 以 T 开头并且不是 TXX 或 XXXX 时。

```
(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "XXXX") :test #'string=)))
     (ecase (length name)
       (3 'text-info-frame-v2.2)
       (4 'text-info-frame-v2.3)))
    (t
     (ecase (length name)
       (3 'generic-frame-v2.2)
       (4 'generic-frame-v2.3))))
```

25.13 评论帧

另一个常用的帧类型是评论帧，它就像一个带有额外字段的文本信息帧。和文本信息帧一样，它以代表帧中所采用的字符串编码方式的单个字节开始。该字节后跟一个三字符的 ISO 8859-1 字符串（无论字符串编码方式字节的值是什么），其代表了评论所使用的语言。它以 ISO 639-2 格式的代码来表示，例如“eng”代表英语，而“jpn”代表日语。这个字段之后是两个根据第一个字节的值所编码的字符串。前一个字符串是空字节结尾的，含有评论的简要描述。后一个字符串是整个帧的最后部分，保存评论文本。

```
(define-binary-class comment-frame ()
  ((encoding u1)
   (language (iso-8859-1-string :length 3))
   (description (id3-encoded-string :encoding encoding :terminator +null+))
   (text (id3-encoded-string
           :encoding encoding
           :length (bytes-left
                     (+ 1 ; encoding
                        3 ; language
                        (encoded-string-length description encoding t)))))))
```

和 `text-info` 混合类的定义一样，你可以使用 `bytes-left` 来计算最后一个字符串的大小。不过，由于 `description` 字段是变长的字符串，因此在 `text` 开始前所需读取的字节数并非常量。更糟糕的是，用来编码 `text` 的字节数取决于编码方式。因此，你应当定义一个助手函数，在给定字符串、编码方式和一个指明字符串是否以某个额外字符结尾等参数的情况下，返回用来编码这个字符串所需的字节数。

```
(defun encoded-string-length (string encoding terminated)
  (let ((characters (+ (length string) (if terminated 1 0))))
    (* characters (ecase encoding (0 1) (1 2)))))
```

然后，和前面一样，你可以定义具体的版本相关的评论帧类并将其嵌入到 `find-frame-class` 中。

```
(define-binary-class comment-frame-v2.2 (id3v2.2-frame comment-frame) ())
```

```
(define-binary-class comment-frame-v2.3 (id3v2.3-frame comment-frame) ())

(defun find-frame-class (name)
  (cond
    ((and (char= (char name 0) #\T)
          (not (member name '("TXX" "XXX") :test #'string=)))
     (ecase (length name)
       (3 'text-info-frame-v2.2)
       (4 'text-info-frame-v2.3)))
    ((string= name "COM") 'comment-frame-v2.2)
    ((string= name "COMM") 'comment-frame-v2.3)
    (t
     (ecase (length name)
       (3 'generic-frame-v2.2)
       (4 'generic-frame-v2.3)))))
```

25.14 从 ID3 标签中解出信息

现在你有了读写 ID3 标签的基本能力，你有许多方向来发展这些代码。如果你想要开发一个完整的 ID3 标签编辑器，那么你将需要实现用于所有帧类型的具体类。你还需要定义以一致的方式来管理标签和帧对象的方法（比如说，如果你改变了一个 `text-info-frame` 中的字符串的值，那么就可能需要调整其大小）；目前的代码无法保证这点。⁹

或者，如果你只需要从 MP3 文件的 ID3 标签里解出关于它的特定信息——如同你即将在第 27、28 和 29 章里开发一个流式 MP3 服务器时所做的那样——那么你将需要编写函数来查找适当的帧并解出你想要的信息。

最后，为了使其成为产品级的代码，你需要仔细确认 ID3 规范并处理所有那些被我跳过的感兴趣部分之外的细节。特别地，某些标签和帧中的标志位可能影响标签或帧的读取方式；除非你编写了代码在这些标志位被设定时做正确的处理，否则就可能会有一些 ID3 标签无法被正确解析。但是来自本章的代码应当可以解析你实际遇到的几乎所有 MP3 文件。

目前你可以再编写几个用来从一个 `id3-tag` 中解出个别信息的函数来结束这项工作。你将在第 27 章或者有可能在用到该库的其他代码里需要这些函数。它们属于该库是因为它们依赖于 ID3 格式的细节，而这是库的用户不应当关心的。

比如说，为了从一个被解出的 `id3-tag` 中获得 MP3 的歌曲名，你需要查找带有特定标识符的 ID3 帧并解出其 `information` 字段。而另外一些信息，例如歌曲的风格，可能还需要进行后续的解码。幸运的是，所有包含你所关心信息的帧都是文本信息帧，因此解出一段特定信息的操作基本上可以细化成使用正确的标识符来查找对应的帧。当然，ID3 的作者们可能决定将所有标识符从 ID3v2.2 改为 ID3v2.3，所以你必须考虑到这点。

没有什么太复杂的东西——你只需找出正确的路径来得到不同的信息就好了。这正是采用交

⁹ 确保这类跨字段的一致性将是访问型广义函数的 `:after` 方法的良好应用场合。例如，你可以定义下面的 `:after` 方法来确保 `size` 与 `information` 字符串同步：

```
(defmethod (setf information) :after (value (frame text-info-frame))
  (declare (ignore value))
  (with-slots (encoding size information) frame
    (setf size (encoded-string-length information encoding nil))))
```

互式开发的大好机会，跟你之前用来找出需要实现哪些帧的方法大致相同。一开始，你需要得到一个 `id3-tag` 对象来进行后续的操作。假设你手头刚好有一个 MP3 文件，你可以像下面这样来使用 `read-id3`:

```
ID3V2> (defparameter *id3* (read-id3 "Kitka/Wintersongs/02 Byla Cesta.mp3"))
*ID3*
ID3V2> *id3*
#<ID3V2.2-TAG @ #x73d04c1a>
```

其中的 `Kitka/Wintersongs/02 Byla Cesta.mp3` 需要替换成你自己的 MP3 文件名。一旦得到了 `id3-tag` 对象，你就可以拿它做实验了。例如，你可以使用 `frames` 函数检出所有帧对象的列表。

```
ID3V2> (frames *id3*)
(#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04dba>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04ea2>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d04f9a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05082>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0516a>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d05252>
 #<TEXT-INFO-FRAME-V2.2 @ #x73d0533a>
 #<COMMENT-FRAME-V2.2 @ #x73d0543a>
 #<COMMENT-FRAME-V2.2 @ #x73d05612>
 #<COMMENT-FRAME-V2.2 @ #x73d0586a>)
```

现在假设你想要解出歌曲标题。它很可能就藏在上面的那些帧里，但为了找到它，你需要查找带有 “TT2” 标识符的帧。好吧，一个足够简单的方法是像下面这样解出所有的标识符来查看标签中是否含有这样一个帧。

```
ID3V2> (mapcar #'id (frames *id3*))
("TT2" "TP1" "TAL" "TRK" "TPA" "TYE" "TCO" "TEN" "COM" "COM" "COM")
```

第一个帧就是。不过，无法保证它总是第一个帧，因此你应当总是通过标识符而不是确定的位置来寻找它。另外也可以直接使用 `FIND` 函数。

```
ID3V2> (find "TT2" (frames *id3*) :test #'string= :key #'id)
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

现在，为了得到帧中的实际信息，可以这样做:

```
ID3V2> (information (find "TT2" (frames *id3*) :test #'string= :key #'id))
"Byla Cesta^@"
```

晕倒。那个 `^@` 是 Emacs 打印一个空字符的方式。在一次从 ID3v1 升级到 ID3v1.1 的行动中，一个文本信息帧的 `information` 槽，尽管没有正式地被定义为空终止的字符串，却可以含有一个空字符，并且 ID3 读取器本该忽略掉空字符以后的任何字符。因此，你需要一个函数来接受一个字符串并返回该字符串直到第一个空字符之前的内容。通过使用来自二进制数据处理库的 `+null+` 常量可以轻易做到这点。

```
(defun upto-null (string)
  (subseq string 0 (position +null+ string)))
```

现在你可以得到正确的标题了。

```
ID3V2> (upto-null
```

```
(information (find "TT2" (frames *id3*) :test #'string= :key #'id))
"Byla Cesta"
```

你可以将这些代码直接封装到一个接受 `id3-tag` 作为参数的名为 `song` 的函数里，然后工作就算是完成了。不过，这些代码与你用来解出其他你需要的信息（例如专辑名、艺术家和风格）的代码的唯一区别仅在于标识符。因此，最好可以将代码拆分一下。对于新手来说，你可以编写一个函数，像下面这样通过给定一个 `id3-tag` 和一个标识符来查找一个帧：

```
(defun find-frame (id3 id)
  (find id (frames id3) :test #'string= :key #'id))

ID3V2> (find-frame *id3* "TT2")
#<TEXT-INFO-FRAME-V2.2 @ #x73d04cca>
```

然后另外一些代码，也就是从 `text-info-frame` 中解出具体信息的那部分，可以写在另一个函数里。

```
(defun get-text-info (id3 id)
  (let ((frame (find-frame id3 id)))
    (when frame (upto-null (information frame)))))

ID3V2> (get-text-info *id3* "TT2")
"Byla Cesta"
```

现在 `song` 函数的定义就只剩下传递正确的标识符了。

```
(defun song (id3) (get-text-info id3 "TT2"))

ID3V2> (song *id3*)
"Byla Cesta"
```

不过，这个 `song` 的定义只能工作在版本 2.2 的标签上，因为标识符在版本 2.2 和版本 2.3 之间从 “TT2” 变成了 “TIT2”。所有其他的标签也改变了。考虑到该库的用户在获取歌曲标题这么简单的事情上不应该被迫去关注 ID3 格式的不同版本，因此你应该帮用户处理好这些细节。一个简单的方法是修改 `find-frame`，让其不只是接受单个标识符，而是像下面这样接受一个标识符的列表：

```
(defun find-frame (id3 ids)
  (find-if #'(lambda (x) (find (id x) ids :test #'string=)) (frames id3)))
```

然后稍微改变 `get-text-info` 使其可以通过 `&rest` 参数接受更多的标识符。

```
(defun get-text-info (id3 &rest ids)
  (let ((frame (find-frame id3 ids)))
    (when frame (upto-null (information frame)))))
```

为了允许 `song` 同时支持版本 2.2 和版本 2.3 的标签，随后需要改动的只是将版本 2.3 的标识符添加进来。

```
(defun song (id3) (get-text-info id3 "TT2" "TIT2"))
```

接下来你只需为那些你想要提供访问函数的字段查找适当的版本 2.2 和版本 2.3 标识符。下面是一些你将在第 27 章里用到的函数：

```
(defun album (id3) (get-text-info id3 "TAL" "TALB"))
(defun artist (id3) (get-text-info id3 "TP1" "TPE1"))
```

```
(defun track (id3) (get-text-info id3 "TRK" "TRCK"))

(defun year (id3) (get-text-info id3 "TYE" "TYER" "TDRC"))

(defun genre (id3) (get-text-info id3 "TCO" "TCON"))
```

最后的难点是保存在 TCO 或 TCON 帧中的 genre 并不总是人类可读的。回顾在 ID3v1 中，风格被保存在单个字节中，使用来自一个固定列表的特别风格进行编码。不幸的是，这些代码继续存在于 ID3v2 中——如果风格字段的文本是一个位于括号中的数字，那么这个数字将被解释成一个 ID3v1 风格代码。但话又说回来，这个库的用户可能并不关心这些年代久远的历史。所以，你应该提供一个函数用来自动地转化这些风格。下面的函数使用刚刚定义的 genre 函数来解出实际的风格文本并检查其是否以一个左括号开始，然后在检测通过时使用一个即将定义的函数来解码版本 1 的风格代码：

```
(defun translated-genre (id3)
  (let ((genre (genre id3)))
    (if (and genre (char= #\\( (char genre 0)))
             (translate-v1-genre genre))
        genre)))
```

由于一个版本 1 风格代码本质上只是到一个标准名称数组的索引，因此实现 translate-v1-genre 的最简单方法就是从风格字符串中解出那个数字并将其作为访问一个实际数组的索引。

```
(defun translate-v1-genre (genre)
  (aref *id3-v1-genres* (parse-integer genre :start 1 :junk-allowed t)))
```

然后你需要做的就是定义这些名字数组了。下面的名字数组里包含了 80 种官方的版本 1 风格，外加由 Winamp 作者所创建的附加风格。

```
(defparameter *id3-v1-genres*
  #(
    ; These are the official ID3v1 genres.
    "Blues" "Classic Rock" "Country" "Dance" "Disco" "Funk" "Grunge"
    "Hip-Hop" "Jazz" "Metal" "New Age" "Oldies" "Other" "Pop" "R&B" "Rap"
    "Reggae" "Rock" "Techno" "Industrial" "Alternative" "Ska"
    "Death Metal" "Pranks" "Soundtrack" "Euro-Techno" "Ambient"
    "Trip-Hop" "Vocal" "Jazz+Funk" "Fusion" "Trance" "Classical"
    "Instrumental" "Acid" "House" "Game" "Sound Clip" "Gospel" "Noise"
    "AlternRock" "Bass" "Soul" "Punk" "Space" "Meditative"
    "Instrumental Pop" "Instrumental Rock" "Ethnic" "Gothic" "Darkwave"
    "Techno-Industrial" "Electronic" "Pop-Folk" "Eurodance" "Dream"
    "Southern Rock" "Comedy" "Cult" "Gangsta" "Top 40" "Christian Rap"
    "Pop/Funk" "Jungle" "Native American" "Cabaret" "New Wave"
    "Psychedelic" "Rave" "Showtunes" "Trailer" "Lo-Fi" "Tribal"
    "Acid Punk" "Acid Jazz" "Polka" "Retro" "Musical" "Rock & Roll"
    "Hard Rock"

    ; These were made up by the authors of Winamp but backported into
    ; the ID3 spec.
    "Folk" "Folk-Rock" "National Folk" "Swing" "Fast Fusion"
    "Bebob" "Latin" "Revival" "Celtic" "Bluegrass" "Avantgarde"
    "Gothic Rock" "Progressive Rock" "Psychedelic Rock" "Symphonic Rock"
    "Slow Rock" "Big Band" "Chorus" "Easy Listening" "Acoustic" "Humor"
    "Speech" "Chanson" "Opera" "Chamber Music" "Sonata" "Symphony"
    "Booty Bass" "Primus" "Porn Groove" "Satire" "Slow Jam" "Club"
    "Tango" "Samba" "Folklore" "Ballad" "Power Ballad" "Rhythmic Soul")
```

```
"Freestyle" "Duet" "Punk Rock" "Drum Solo" "A capella" "Euro-House"
"Dance Hall"

;; These were also invented by the Winamp folks but ignored by the
;; ID3 authors.
"Goa" "Drum & Bass" "Club-House" "Hardcore" "Terror" "Indie"
"BritPop" "Negerpunk" "Polsk Punk" "Beat" "Christian Gangsta Rap"
"Heavy Metal" "Black Metal" "Crossover" "Contemporary Christian"
"Christian Rock" "Merengue" "Salsa" "Thrash Metal" "Anime" "Jpop"
"Synthpop"))
```

再一次，你可能感觉自己在本章里写了大量代码。但如果你将它们全部放在一个文件里，或是你下载了来自本书 Web 站点上的版本。你会发现其实并没有多少行——编写这个库的主要难点全部来自对 ID3 格式本身的复杂性的理解。不管怎么说，现在你得到了将在第 27、28 和 29 章里编写的流式 MP3 服务器的一个主要部分。其他你所需要的主要基础设施是一种编写服务器端 Web 软件的方式，这就是下一章的主题。

第26章 实践：使用 AllegroServe 进行 Web 编程

在本章里，你将学习在 Common Lisp 中开发基于 Web 的程序的一种方法，使用开源的 AllegroServe Web 服务器。这并不意味着一次对 AllegroServe 的完整介绍。并且我确定只打算谈及关于 Web 编程这个大型话题的冰山一角。我的目标是覆盖到足够多的 AllegroServe 基本用法以确保你可以在第 29 章里用它来开发一个可以浏览 MP3 文件库并将它们以流的方式发送到 MP3 客户端的应用程序。类似地，本章也为 Web 编程的初学者们提供了一个简单的介绍。

26.1 30 秒介绍服务端 Web 编程

尽管当今的 Web 程序开发通常都会用到相当数量的软件框架和不同的协议，但 Web 编程的核心部分自从它们在 1990 年代早期被发明出来以后几乎没有什么变化。对于诸如你将在第 29 章里编写的这类简单应用，你只需理解几个关键的概念就可以了，因此我将在这里快速地回顾一下。有经验的 Web 程序员可以粗略阅读或是干脆跳过本节的其余部分。¹

首先，你需要理解 Web 浏览器和 Web 服务器在 Web 编程中所处的角色。尽管一个现代浏览器通常带有大量花哨的功能，但一个 Web 浏览器的核心功能只是从一个 Web 服务器上请求 Web 页并将它们渲染出来。通常这些页面是使用超文本标记语言（HTML）来编写的，它可以告诉浏览器如何渲染页面，包括在哪里插入内嵌的图像和指向其他 Web 页的链接。HTML 由带有标签的文本所组成，这些标签为文本添加了结构，使浏览器得以渲染页面。例如，一个简单的 HTML 文档可能看起来像下面这样：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
    <p>This is a picture: </p>
    <p>This is a <a href="another-page.html">link</a> to another page.</p>
  </body>
</html>
```

¹ 初学 Web 编程的读者们可能需要在这篇介绍的基础上补充阅读一两篇更加深入的介绍。你可以在 <http://www.jmarshall.com/easy/> 找到一些很好的在线指导。

图 26-1 显示了浏览器是如何渲染这个页面的。

(图 26-1)

浏览器和服务器之间使用一种称为超文本传输协议 (HTTP) 的协议进行通信。尽管你不需要关心该协议的细节，但有必要知道它是完全由一系列由浏览器所发起的请求和服务器所生成的回执所构成的。这就是说，浏览器连接到 Web 服务器上并发送了一个至少包括目标 URL 和浏览器所支持的 HTTP 版本的请求。浏览器也可以在它的请求中包含数据；这就是浏览器向服务器提交 HTML 表单的方式。

为了回应一个请求，服务器发送了一个包括一系列的头和一个主体在内的回执。头中含有关于主体的信息，诸如数据的类型是什么（比如说 HTML、纯文本，或一个图片），而主体就是数据本身，随后将被浏览器所渲染。服务器有时也会发送一个错误回执来告诉浏览器其请求因为某种原因无法正确回应。

情况基本上就是这样。一旦浏览器从服务器那里收到了完整的回执，那么直到下一次浏览器决定从服务器请求一个页面之前，在浏览器和服务器之间将不再有任何通信。²这就是 Web 编程的主要约束所在——没有办法让运行在服务器上的代码影响到用户在浏览器中所看到的内容，除非浏览器向服务器发起了一个新请求。³

有些称为静态页面的 Web 页只是保存在 Web 服务器上的 HTML 文件，在被浏览器请求时直接被发送出去。另一方面，动态页面是由每次页面被浏览器所请求时所生成的 HTML 构成的。举个例子，一个动态页面可能会在查询一个数据库时被生成并构造出 HTML 来表示查询的结果。⁴

当针对一个请求生成回执时，服务器端的代码需要处理四种主要信息。第一种信息时被请求的 URL。不过，URL 通常被 Web 服务器本身用来决定使用哪些代码来生成回执。接下来，如果 URL 中含有一个问号，那么问号之后的所有内容将被视为一个查询字符串，后者通常会被 Web 服务器所忽略，除非将它传给用来生成回执的代码。多数时候查询字符串由一组键/值对所组成。来自浏览器的请求也可以含有发送数据，这些数据通常也由键值对所构成。发送数据一般用来提交 HTML 表单。无论查询字符串还是发送数据中的键值对都被统称为查询参数 (query parameter)。

最后，为了将来自同一个浏览器的一系列请求串接在一起，服务器中运行的代码可以设置一个 cookie，并在浏览器的回执中发送一个特殊的头，里面含有一些称为 cookie 的不透明数据。一旦 cookie 被一个特定的服务器所设置，那么浏览器将在每次向该服务器发送请求时都带上这个 cookie。浏览器并不关心 cookie 中的数据——它只是将其回显给服务器，让服务器端的代码按照它们想要的方式来解释。

² 加载单个 Web 页面可能实际上会产生多个请求——为了渲染一个含有内嵌图片的页面的 HTML，浏览器必须单独地请求每个图片，再将它们分别插入到渲染后的 HTML 中的适当位置。

³ Web 编程的许多复杂性都是试图解决这个基本限制的结果，目标是提供类似桌面应用那样的用户体验。

⁴ 不幸的是，“动态”一词在 Web 世界中被重载了。术语“动态 HTML”指的是含有嵌入式代码的 HTML，其代码通常采用 JavaScript 语言编写，后者可在不跟 Web 服务器进行通信的情况下在浏览器中执行。如果谨慎使用，动态 HTML 可以改进一个基于 Web 的应用程序的可用性，因为即便在高速的 Internet 连接下，向一个 Web 服务器发出请求、接受回执并渲染新页面也需要使用数量可观的时间。更加令人困惑的是，动态生成的页面（换句话说，是在服务器上生成的）也可以含有动态 HTML（运行在客户端的代码）。对于本书中的应用，你将只是动态地生成简单的非动态 HTML。

以上就是 99% 的服务器端 Web 编程所依赖的基础元素。浏览器发起一个请求，服务器查找用来处理该请求的代码并运行它，然后代码使用查询参数和 cookie 来决定所做的事。

26.2 AllegroServe

你有很多种方式可以用 Common Lisp 来提供 Web 内容；至少有三种用 Common Lisp 写的开源 Web 服务器，还有诸如 mod_lisp⁵ 和 Lisplets⁶ 这类系统可以允许 Apache Web 服务器或任何 Java Servlet 容器将请求代理到运行在分离进程中的 Lisp 服务器上。

对于本章来说，你将使用开源 Web 服务器 AllegroServe 的某个版本，它最初由 Franz Inc. 的 John Foderaro 开发。AllegroServe 被包含在来自 Franz 的用于本书的 Allegro 版本里。如果你没在使用 Allegro，那么你可以使用 PortableAllegroServe，一个 AllegroServe 代码树的友好分支，后者还包括一个让 PortableAllegroServe 得以运行在多数 Common Lisp 平台上的兼容层。你将在本章和第 29 章里编写的代码应该可以同时运行在原版 AllegroServe 和 PortableAllegroServe 上。

AllegroServe 提供了一个与 Java Servlet 类似的编程模型——每当浏览器请求一个页面时，AllegroServe 会解析请求并查找一个称为项 (entity) 的对象来处理该请求。一些作为 AllegroServe 一部分提供的项类知道如何处理静态内容——无论是单独的文件还是一个目录树的内容。而另一些则是我将用本章的多数篇幅进行讨论的，它们运行任意 Lisp 代码来生成回执。⁷

但在开始之前，你需要知道如何启动 AllegroServe 并让它服务一些文件。第一步是将 AllegroServe 加载到你的 Lisp 映像中。在 Allegro 中，你可以简单地键入 (`(require :aserve)`)。在其他 Lisp 环境（也包括 Allegro 在内）下，你可以通过加载 portableaserve 目录树顶层的文件 `INSTALL.lisp` 来加载 PortableAllegroServe。加载 AllegroServe 将会创建三个新包，`NET.ASERVE`、`NET.HTML.GENERATOR` 和 `NET.ASERVE.CLIENT`。⁸

加载了服务器以后，你可以通过 `NET.ASERVE` 包中的函数 `start` 来启动它。为了可以方便地访问来自 `NET.ASERVE`、来自 `COM.GIGAMONKEYS.HTML`（一个即将讨论到的新包），以及来自 Common Lisp 其余部分的导出符号，你应该像下面这样创建一个新包：

```
CL-USER> (defpackage :com.gigamonkeys.web
  (:use :cl :net.aserve :com.gigamonkeys.html))
#<The COM.GIGAMONKEYS.WEB package>
```

现在使用下面的 `IN-PACKAGE` 表达式切换到该包上：

```
CL-USER> (in-package :com.gigamonkeys.web)
#<The COM.GIGAMONKEYS.WEB package>
WEB>
```

⁵ http://www.fractalconcept.com/asp/html/mod_lisp.html

⁶ <http://lisplets.sourceforge.net/>

⁷ AllegroServe 也提供了一个称为 Webactions 的框架，后者类似于 Java 世界中的 JSP——与其编写代码来生成 HTML，通过 Webactions 你可以直接编写本质上是 HTML 的页面，但其中的某些内容将在页面提供服务时作为代码来运行。我在本书里将不会谈及 Webactions。

⁸ 加载 PortableAllegroServe 将为相关的兼容库创建出其他一些包，但你需要关心的只是那三个包。

现在你可以无需限定符来使用来自 `NET.ASERVE` 的导出符号了。函数 `start` 用来启动服务器。它接受相当数量的关键字参数，但你现在唯一需要传递的是 `:port`，它指定了监听的端口。你可能需要使用诸如 2001 这种较大的端口而不是 HTTP 服务器的标准端口 80，因为在类 Unix 的操作系统里只有 `root` 用户才能监听在 1024 以下的端口上。为了在 Unix 上运行监听在 80 端口上的 `AllegroServe`，你将需要以 `root` 用户启动 Lisp 然后使用 `:setuid` 和 `:setgid` 参数来告诉 `start` 在打开端口以后切换到指定的身份。你可以像下面这样启动一个监听在端口 2001 的服务器：

```
WEB> (start :port 2001)
#<WSERVER port 2001 @ #x72511c72>
```

服务器现在在你的 Lisp 环境中运行了。你有可能会在试图启动服务器时得到类似“`port already in use`”这样的错误提示。这表明端口 2001 已经被你系统里的其他服务器占用了。在这种情况下，最简单的修复方法是使用一个不同的端口，为 `start` 提供一个不同的参数，然后在本章的其余部分的 URL 里始终用该值来代替 2001。

你可以继续通过 REPL 与 Lisp 环境进行交互，因为 `AllegroServe` 启动了它自己的线程来处理来自浏览器的请求。这意味着，别的不说，你可以通过 REPL 来观察你当前运行中的服务器，这使得调试和测试工作比面对一个完全黑箱的服务器要容易得多。

假设你正在运行的 Lisp 环境与你的浏览器是在同一台机器上，那么你可以通过将浏览器指向 `http://localhost:2001/` 来检查服务器是否已经启动并运行了。此刻你应该会在浏览器中得到一个页面未找到的错误信息，因为你还没有发布任何内容。但这个错误信息将会来自 `AllegroServe`；可以从页面的底部看到这点。另一方面，如果浏览器显示了一个错误对话框并提示说“The connection was refused when attempting to contact localhost:2001”，那么这意味着要么服务器没在运行，要么是从不同于 2001 的端口启动的。

现在你可以发布一些文件了。假设你在 `/tmp/html` 目录下有一个文件 `hello.html`，其内容如下：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

你可以使用 `publish-file` 函数单独地发布它。

```
WEB> (publish-file :path "/hello.html" :file "/tmp/html/hello.html")
#<NET.ASERVE::FILE-ENTITY @ #x725eddea>
```

其中的 `:path` 参数将出现在浏览器所请求的 URL 中，而 `:file` 参数则是文件系统中的文件名。在求值 `publish-file` 表达式之后，你可以将浏览器指向 `http://localhost:2001/hello.html`，然后它将显示出类似图 26-2 这样的一个页面。

(图 26-2)

你也可以使用 `publish-directory` 函数来发布整个目录树中的文件。首先让我们使用下面的 `publish-file` 调用将已发布的项清除：

```
WEB> (publish-file :path "/hello.html" :remove t)
NIL
```

现在你可以使用 `publish-directory` 函数将整个 `/tmp/html/` 目录（包括它的所有子目录）发布了。

```
WEB> (publish-directory :prefix "/" :destination "/tmp/html/")
#<NET.ASERVE::DIRECTORY-ENTITY @ #x72625aa2>
```

在本例中，`:prefix` 参数指定了应由该项接手的 URL 路径部分的开始。这样，如果服务器收到了一个来自 `http://localhost:2001/foo/bar.html` 的请求，那么其路径部分是 `/foo/bar.html`，以“`/`”开始。这个路径随后通过将其前缀“`/`”替换成目标“`/tmp/html/`”从而变成了一个文件名。同样道理，`http://localhost:2001/hello.html` 也将被转化成一个对文件 `/tmp/html/hello.html` 的请求。

26.3 用 AllegroServe 生成动态内容

发布生成动态内容的项几乎和发布静态内容一样简单。函数 `publish` 和 `publish-prefix` 是 `publish-file` 和 `publish-directory` 对应的动态版本。这两个函数的基本思想是，你可以发布一个函数，它将被调用来生成一个指定 URL 或带有给定前缀的任何 URL 的回执。这个函数将用两个参数来调用：一个代表请求的对象以及一个被发布的项。多数时候你不需要对那个项对象做任何操作，除非将它和一些后面即将讨论到的宏一起传递。另一方面，你将使用请求对象来获取由浏览器所提交的信息——包含在 URL 或使用 HTML 表单发送的数据中的查询参数。

对于使用一个函数来生成动态内容的简单示例，让我们编写一个函数在每次请求时生成一个带有不同随机数的页面。

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (format
        (request-reply-stream request)
        "<html>~@
<head><title>Random</title></head>~@
<body>~@
<p>Random number: ~d</p>~@
</body>~@
</html>~@
"
        (random 1000)))))
```

宏 `with-http-response` 和 `with-http-body` 是 AllegroServe 的一部分。前者开始了生成一个 HTTP 回执的过程并且可以像这里这样指定诸如返回内容的类型之类的东西。它还可以处理 HTTP 的其他部分，例如处理 `If-Modified-Since` 请求。`with-http-body` 实际发送 HTTP 回执头并执行其主体，后者应当含有用来生成回执内容的代码。在 `with-http-response` 中 `with-http-body` 之前的地方，你可以添加或修改在回执中发送的 HTTP 头。函数 `request-reply-stream` 也是 AllegroServe 的一部分，它返回一个流用来向浏览器中写入想要的输出。

正如该函数所显示的，你可以只用 `FORMAT` 将 HTML 打印到由 `request-reply-stream` 所返回

的流上。在下一节里，我将向你展示更方便的方法来程序化生成 HTML。⁹

现在你可以发布这个函数了。

```
WEB> (publish :path "/random-number" :function 'random-number)
#<COMPUTED-ENTITY @ #x7262bab2>
```

参数 :path 与它在 publish-file 函数中的用法相同，它指定导致该函数被调用的 URL 的路径部分。:function 参数用来指定函数的名字或实际的函数对象。像这里这样使用一个函数的名字可以允许你以后重定义该函数而无需重新发布即可令 AllegroServe 使用新的函数定义。在求值了 publish 调用以后，你可以让浏览器指向 <http://localhost:2001/random-number> 来得到一个带有一个随机数的页面，如图 26-3 所示。

(图 26-3)

26.4 生成 HTML

尽管使用 FORMAT 来生成 HTML 对于目前为止我所讨论的简单页面工作得很好，但如果你开始构建更复杂的页面，那么如果可以有一种更简洁的 HTML 生成方式就再好不过了。有几个库可以用来从 S-表达式形式的表示生成 HTML，其中之一的 htmlgen 就包含在 AllegroServe 中。在本章里你将使用一个称为 FOO¹⁰ 的库，它在很大程度上来自 Franz 的 htmlgen，并且对于它的具体实现你将在第 30 和 31 章里看到更多的细节。不过目前你只需要知道如何使用 FOO。

从 Lisp 里生成 HTML 是件相当自然的事情，因为本质上 S-表达式跟 HTML 是同构的。你可以用 S-表达式来表示 HTML 元素，方法是将 HTML 中的每个元素视为一个以适当头元素“标记”的列表，例如一个与 HTML 标签同名的关键字符串。这样，HTML <p>foo</p> 就可以用 S-表达式 (:p "foo") 来表示了。由于 HTML 元素嵌套的方式与 S-表达式中的列表嵌套方式相同，因此上述表示法可以扩展到更复杂的 HTML 上。例如，下面的 HTML：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

可以用下列 S-表达式来表示：

```
(:html
  (:head (:title "Hello"))
  (:body (:p "Hello, world!")))
```

带有属性的 HTML 元素令事情稍微变得复杂一些，但也不是无法克服的。FOO 支持两种方式

⁹ ~@后接一个新行可以告诉 FORMAT 忽略换行之后的所有空白，这允许你可以精美地缩进你的代码而不会在 HTML 中增加大量的空白。由于 HTML 中的空白通常会被忽略，因此这不会影响到浏览器，但它可以让产生的 HTML 看起来更美观。

¹⁰ FOO 是来源于 FOO Outputs Output 的递归伪技术缩略语。

在标签中加入属性。一种方式是简单地在列表的第一个元素之后跟上一个键值对。跟在键值对后面的第一个不是关键字符的元素代表该 HTML 元素的内容的开始。这样，你可以将下面的 HTML：

```
<a href="foo.html">This is a link.</a>
```

用下列 S-表达式来表示：

```
(:a :href "foo.html" "This is a link.")
```

FOO 所支持的另一种语法是将标签名和属性组织在它们自己的列表中，如下所示：

```
((:a :href "foo.html") "This is a link.")
```

FOO 可以通过这两种方式使用 S-表达式来表示 HTML。函数 `emit-html` 接受一个 HTML 的 S-表达式并输出相应的 HTML。

```
WEB> (emit-html '(:html (:head (:title "Hello")) (:body (:p "Hello, world!"))))  
<html>  
  <head>  
    <title>Hello</title>  
  </head>  
  <body>  
    <p>Hello, world!</p>  
  </body>  
</html>  
T
```

不过，`emit-html` 并非总是最有效的 HTML 生成方式，因为其参数必须是想要生成的 HTML 的完整 S-表达式表示。尽管构造这样一个表示很容易，但它却并不总是高效的。例如，假设你要生成一个含有 10,000 个随机数的列表的 HTML 页面。你可以像下面这样使用一个反引用模板来构造 S-表达式并将其传给 `emit-html`：

```
(emit-html  
  `(:html  
    (:head  
      (:title "Random numbers"))  
    (:body  
      (:h1 "Random numbers")  
      (:p ,(loop repeat 10000 collect (random 1000) collect " "))))
```

不过，这会导致再实际开始生成 HTML 之前就先要构造出一个含有 10,000 个元素的列表的树来，而整个 S-表达式一旦 HTML 生成出来以后就没有任何用处了。为了避免这种低效，FOO 还支持一个宏 `html`，它允许你在一个 HTML 的 S-表达式中嵌入一点儿 Lisp 代码。

位于 `html` 宏的输入中的诸如字符串和数字这样的字面值将被插入到输出的 HTML 中。同样，符号将被视为对变量的引用，宏所生成的代码会在运行期输出它们的值。这样，下面两个形式：

```
(html (:p "foo"))  
(let ((x "foo")) (html (:p x)))
```

都将生成下面的代码：

```
<p>foo</p>
```

不以一个关键字符开始的列表形式会被视为代码，并被嵌入到生成的代码中。被嵌入的代码所返回的任何值都将被忽略，但是代码可以通过调用 `html` 宏本身来产生更多的 HTML。例如，

为了在 HTML 中输出一个列表的内容，你可以写成下面这样：

```
(html (:ul (dolist (item (list 1 2 3)) (html (:li item)))))
```

它将产生下面的 HTML：

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

如果你想输出一个列表形式的值，你必须将其包装在伪标签 `:print` 中。这样，下面的表达式：

```
(html (:p (+ 1 2)))
```

在计算并丢弃值 3 以后会生成下面的 HTML：

```
<p></p>
```

为了输出那个 3，你必须写成下面这样：

```
(html (:p (:print (+ 1 2))))
```

或者你也可以先计算出该值并将其保存在一个 `html` 调用之外的变量里，像下面这样：

```
(let ((x (+ 1 2))) (html (:p x)))
```

这样，你可以使用 `html` 宏来生成随机数的列表，像下面这样：

```
(html
  (:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p (loop repeat 10 do (html (:print (random 1000)) "))))))
```

宏版本将比 `emit-html` 版本更加高效。不仅你不再需要生成一个代表整个页面的 S- 表达式了，而且 `emit-html` 的很多在运行期解释 S- 表达式的工作现在可以在宏展开时一次性完成，而不必在每次代码运行时来做了。

你可以通过宏 `with-html-output` 来控制由 `html` 和 `emit-html` 所生成的输出被发送到哪里，该宏是 FOO 库的一部分。这样，你可以使用来自 FOO 的 `with-html-output` 和 `html` 宏来重写 `random-number`，就像下面这样：

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random 1000))))))))))
```

26.5 HTML 宏

FOO 的另一个特性是它允许你定义一种可将任意形式转化成 html 宏可理解的 HTML S-表达式的 HTML “宏”。例如，假设你经常发现自己会编写下列形式的页面：

```
(:html
  (:head (:title "Some title"))
  (:body
    (:h1 "Some title")
    ... stuff ...))
```

那么你应该定义一个 HTML 宏来捕捉下面这个模式：

```
(define-html-macro :standard-page ((&key title) &body body)
  `(:html
    (:head (:title ,title))
    (:body
      (:h1 ,title)
      ,@body)))
```

现在你可以在你的 S-表达式 HTML 中使用“标签”`:standard-page`了，它将在被解释或编译之前展开。例如，下面的形式：

```
(html (:standard-page (:title "Hello") (:p "Hello, world.")))
```

可以生成这样的 HTML：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Hello, world.</p>
  </body>
</html>
```

26.6 查询参数

当然，生成 HTML 输出还只是 Web 编程的一半。其他你需要做的是得到来自用户的输入。正如我在“30 秒介绍服务端 Web 编程”一节中所讨论的，当浏览器从 Web 服务器上请求一个页面时，它可以在 URL 和投递数据中发送查询参数，两者都是向服务器端代码提供输入的途径。

和多数 Web 编程框架一样，AllegroServe 可以帮你解析这两种输入来源。等到你发布的函数被调用时，所有来自查询字符串和/或投递数据的键/值对都已被解码并放置在一个 alist 中，后者可以使用函数 `request-query` 从请求对象中获取到。下面的函数可以返回一个页面，其中显示了所有它收到的查询参数：

```
(defun show-query-params (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
```

```
(:standard-page
  (:title "Query Parameters")
  (if (request-query request)
      (html
        (:table :border 1
          (loop for (k . v) in (request-query request)
            do (html (:tr (:td k) (:td v))))))
      (html (:p "No query parameters."))))))

(publish :path "/show-query-params" :function 'show-query-params)
```

如果你给浏览器一个类似下面这样的带有查询字符串的 URL:

```
http://localhost:2001/show-query-params?foo=bar&baz=10
```

那么你应该可以得到一个类似图 26-4 中所示的页面。

(图 26-4)

为了生成一些投递数据，你需要一个 HTML 表单。下面的函数可以生成一个简单的表单，其数据将发送到 show-query-params:

```
(defun simple-form (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let ((*html-output* (request-reply-stream request)))
        (html
          (:html
            (:head (:title "Simple Form"))
            (:body
              (:form :method "POST" :action "/show-query-params"
                (:table
                  (:tr (:td "Foo")
                    (:td (:input :name "foo" :size 20)))
                  (:tr (:td "Password")
                    (:td (:input :name "password" :type "password" :size 20))))
                (:p (:input :name "submit" :type "submit" :value "Okay")
                  (:input :type "reset" :value "Reset")))))))))

(publish :path "/simple-form" :function 'simple-form)
```

将你的浏览器指向 `http://localhost:2001/simple-form`，然后你应该可以看到一个类似图 26-5 所示的页面。

(图 26-5)

如果你在表单中填入“abc”和“def”两个值，那么点击 Okay 按钮应该会把你带到一个类似图 26-6 所示的页面里。

(图 26-6)

尽管如此，多数时候你不需要在所有查询参数上迭代；你只需要提取单独的参数。例如，你可能想要修改 `random-number`，令你传给 `RANDOM` 的限制值可以通过一个查询参数来提供。在这种情况下，你可以使用函数 `request-query-value`，它接受一个请求对象和你想要查询的参数名并将其值以字符串的形式返回，或者当参数没有提供时返回 `NIL`。一个参数化的 `random-number` 版本可能看起来如下所示：

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let* ((*html-output* (request-reply-stream request))
             (limit-string (or (request-query-value "limit" request) ""))
             (limit (or (parse-integer limit-string :junk-allowed t) 1000)))
        (html
         (:html
          (:head (:title "Random"))
          (:body
           (:p "Random number: " (:print (random limit))))))))))
```

由于 `request-query-value` 可能返回 `NIL` 或一个空字符串，因此你在把参数解析成一个用来传给 `RANDOM` 的数字时需要同时考虑这两种情况。你可以在绑定 `limit-string` 时当没有 `"limit"` 查询参数的情况下将它绑定到空字符串 `" "`，从而处理 `NIL` 的情形。然后你可以使用带有 `:junk-allowed` 参数的 `PARSE-INTEGER` 来确保它要么返回 `NIL`（如果不能从给定字符串中解析出整数的话）要么返回一个整数。在“一个小型应用框架”一节中，你将开发一些宏来使查询参数的提取和到多种类型的转换工作变得更加容易。

26.7 Cookie

在 AllegroServe 中你可以发送一个 `Set-Cookie` 头来告诉浏览器保存一个 cookie 并将其随着后续请求一起发送，方法是在 `with-http-response` 的主体中调用 `with-http-body` 以前，调用函数 `set-cookie-header`。该函数的第一个参数是请求对象，其余参数都是用来设定 cookie 中不同属性的关键字参数。其中唯一的两个你必须传递的是 `:name` 和 `:value` 参数，两者都应该是字符串。其他可能影响发送到浏览器的 cookie 的参数包括 `:expires`、`:path`、`:domain` 和 `:secure`。

当然，你只需要担心 `:expires`。它控制浏览器应该保存 cookie 多久。如果 `:expires` 是 `NIL`（缺省值），那么浏览器只把 cookie 保存到退出时。其他可能的值是 `:never`，这意味着 cookie 应当永远被保持下去，或者一个由 `GET-UNIVERSAL-TIME` 或 `ENCODE-UNIVERSAL-TIME` 所返回的全局时间。一个值为零的 `:expire` 参数告诉客户立即丢弃一个已有的 cookie。¹¹

在你设置了一个 cookie 以后，你可以使用函数 `get-cookie-values` 得到一个 `alist`，其中含有浏览器所发送的每个 cookie 所对应的一个键值对。从这个 `alist` 中，你可以使用 `ASSOC` 和 `CDR` 来提取单独的 cookie 值。

下面的函数可以显示出浏览器所发送的所有 cookie 的名字和值：

```
(defun show-cookies (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
         (:standard-page
          (:title "Cookies")
          (if (null (get-cookie-values request))
              (html (:p "No cookies."))
              (html
               (:table
```

¹¹ 关于其他参数的含义的有关信息，可参见 AllegroServe 文档和 RFC 2109，这些文档里描述了 cookie 机制。

```
(loop for (key . value) in (get-cookie-values request)
      do (html (:tr (:td key) (:td value)))))))))))
```

当你第一次加载页面 `http://localhost:2001/show-cookies` 时它应该会说 “No cookies”，如图 26-7 所示，因为你还没有设置任何 cookie。

(图 26-7)

为了设置一个 cookie，你需要另外一个函数，例如下面这个：

```
(defun set-cookie (request entity)
  (with-http-response (request entity :content-type "text/html")
    (set-cookie-header request :name "MyCookie" :value "A cookie value")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request)))
        (html
          (:standard-page
            (:title "Set Cookie")
            (:p "Cookie set.")
            (:p (:a :href "/show-cookies" "Look at cookie jar.")))))))
  (publish :path "/set-cookie" :function 'set-cookie))
```

如果你输入 URL `http://localhost:2001/set-cookie`，那么你的浏览器应该会显示一个如图 26-8 所示的页面。同时，服务器将发送一个 Set-Cookie 头，其中带有一个名为 “MyCookie” 值为 “A cookie value”的 cookie。如果你点击链接 Look at cookie jar，那么你将被带到 `/show-cookies` 页面，在那里你将看到新的 cookie，如图 26-9 所示。由于你并非指定一个 :expires 参数，因此浏览器将继续在每个请求中发送该 cookie，直到你退出了浏览器。

(图 26-8)

(图 26-9)

26.8 一个小型应用框架

尽管 AllegroServe 为你提供了用来编写服务端 Web 代码（访问 URL 和提交数据中的查询字符串；设置和获取 cookie 值的能力；以及，当然了，生成发还给浏览器的回执的能力）几乎所有基本功能的相当直接的访问，但这样需要写很多令人讨厌的重复性代码。

举个例子，你所编写的每一个 HTML 生成函数都需要带有参数 `request` 和 `entity`，并且它们都将会包含对 `with-http-response`、`with-http-body` 以及——如果你打算使用 `FOO` 来生成 HTML 的话——`with-html-output` 的调用。然后，在需要获取查询参数的函数里，还会有大量的 `request-query-value` 调用以及随后更多的代码来将这些字符串转化成你实际需要的任何类型。最后，你还需要记得 `publish` 这些函数。

为了减少你不得不书写的样板代码的数量，你可以在 AllegroServer 之上编写一个小型的框架，使其可以更容易地用来定义那些处理特定 URL 请求的函数。

基本的思路将是定义一个宏，`define-url-function`，你可以用它来定义可以自动通过 `publish` 发布的函数。这个宏将展开成一个含有适当样板代码的 `DEFUN`，以及在同名的 URL 下

发布该函数的代码。它也负责生成代码来从查询参数和 cookie 中解出值并将它们绑定到声明在函数参数列表中的变量上。这样，一个 `define-url-function` 定义的基本形式将是下面这样：

```
(define-url-function name (request query-parameter*)
  body)
```

其中 `body` 是产生页面 HTML 的代码。它将被包装在一个对 FOO 的 `html` 宏的调用中，因此对于简单的页面来说它除了含有 S-表达式形式的 HTML 以外就别无他物了。

在宏的主体中，查询参数变量将被绑定到同名查询参数或来自一个 cookie 的值上。在最简单的情形下，一个查询参数的值将是从同名的查询参数或投递数据字段中的字符串。如果查询参数使用列表来指定，你还可以指定一个自动的类型转换、一个缺省值，以及是否在一个 cookie 中查找并保存该值。`query-parameter` 的完整语法如下所示：

```
name | (name type [default-value] [stickiness])
```

其中的 `type` 必须是一个 `define-url-function` 可以识别的名字。我将很快讨论如何定义新的类型。`default-value` 必须是该给定类型的一个值。最后，如果有 `stickiness`，它表示参数的值应当在没有查询参数被提供的情况下从一个适当命名的 cookie 中获取，并且一个 Set-Cookie 头应当在回执中发送，其中保存了同名 cookie 的值。这样，一个粘滞参数，在显式地通过一个查询参数的值来指定以后，将在该页面的后续请求中保持该值，即便没有查询参数被提供。

所使用的 cookie 名取决于 `stickiness` 的值：使用一个值 `:global`，cookie 将采用与参数相同的命名。这样，使用同名的全局粘滞参数的不同函数将共享其值。如果 `stickiness` 是 `:package`，那么 cookie 的名字将根据参数的名字和函数名所在的包构造出来；这允许来自同一个包的函数可以共享一些值而不必担心被其他包里的函数参数所破坏。最后，一个带有 `stickiness` 值为 `:local` 的参数将根据参数名、函数名所在的包，以及函数名来生成的 cookie 的名字，这使其成为该函数所独有的。

举个例子，你可以使用 `define-url-function` 来将之前 `random-page` 的 17 行定义替换成下面的 5 行版本：

```
(define-url-function random-number (request (limit integer 1000))
  (:html
    (:head (:title "Random"))
    (:body
      (:p "Random number: " (:print (random limit))))))
```

如果你想要限制参数为粘滞的，你可以将 `limit` 的声明改成 `(limit integer 1000 :local)`。

26.9 实现

我将会自顶向下地解释 `define-url-function` 的实现。该宏本身如下所示：

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
      `(progn
        (defun ,name (,request ,entity)
          (with-http-response (,request ,entity :content-type "text/html")
```

```
(let* (,@(param-bindings name request params))
  ,@(set-cookies-code name request params)
  (with-http-body (,request ,entity)
    (with-html-output ((request-reply-stream ,request))
      (html ,@body))))))
(publish :path ,(format nil "/(~a~)" name) :function ',name))))
```

让我们一点一点地分析它，首先看最初的几行。

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params))))
```

直到这里你才开始生成代码。你用 `GENSYM` 生成一个符号以便在后面的 `DEFUN` 里作为项参数的名字来使用。然后你正则化所有参数，使用下列函数将简单的符号转化成列表形式：

```
(defun normalize-param (param)
  (etypecase param
    (list param)
    (symbol `(,param string nil nil))))
```

换句话说，只用一个符号来声明一个参数等价于声明一个不带有缺省值的非粘滞字符串参数。

接下来是 `PROGN`。你必须展开成一个 `PROGN`，因为你需要生成代码来做两件事：用 `DEFUN` 定义一个函数，以及调用 `publish`。你应当首先定义该函数，这样如果定义中出现错误，那么该函数将不会被发布。`DEFUN` 的前两行只是一些样板代码。

```
(defun ,name (,request ,entity)
  (with-http-response (,request ,entity :content-type "text/html")
  ...)
```

现在你开始做实际的工作。接下来两行为 `define-url-function` 中指定的除 `request` 以外的参数生成绑定，以及为粘滞性参数调用 `set-cookie-header` 的代码。当然，实际的工作是由你即将看到的助手函数来完成的。¹²

```
(let* (,@(param-bindings name request params))
  ,@(set-cookies-code name request params))
```

其余的就只是些样板代码了，将来自 `define-url-function` 定义的主体放在适当的 `with-http-body`、`with-html-output` 和 `html` 宏的上下文中。然后是对 `publish` 的调用。

```
(publish :path ,(format nil "/(~a~)" name) :function ',name))
```

表达式 `(format nil "/(~a~)" name)` 在宏展开阶段求值，生成一个由 “/” 后跟你定义的这个函数名的全小写版本。该字符串随后成为 `publish` 的 `:path` 参数，而函数名则作为 `:function` 参数被插入。

现在让我们查看用来生成 `DEFUN` 形式的助手函数。为了生成参数绑定，你需要在 `params` 上循环并收集来自每个参数的 `param-binding` 所生成的代码片段。该片段将是一个含有需要绑定

¹² 你需要使用 `LET*` 而非 `LET` 来允许参数的缺省值形式可以引用更早出现在参数列表中的参数。例如，你可以写成下面这样：

```
(define-url-function (request (x integer 10) (y integer (* 2 x))) ...)
从而允许当 y 的值没有显式提供时，将使用 x 值的两倍。
```

的变量名和用来计算该变量值的代码的列表。用来计算值的代码的确切形式取决于参数的类型、是否为粘滞的，以及其缺省值，如果有的话。因为你已经正则化了所有的参数，所以你可以在 `param-bindings` 中使用 `DESTRUCTURING-BIND` 来将各部分取出。

```
(defun param-bindings (function-name request params)
  (loop for param in params
    collect (param-binding function-name request param)))

(defun param-binding (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (let ((query-name (symbol->query-name name))
          (cookie-name (symbol->cookie-name function-name name sticky)))
      `(~(,name (or
                  (string->type ',type (request-query-value ,query-name ,request))
                  ,(if cookie-name
                      (list `(string->type ',type
                                         (get-cookie-value ,request ,cookie-name)))
                      ,default)))))))
```

函数 `string->type` 用来将那些从查询参数和 cookie 中获取的字符串转化成想要的类型，它是一个下列形式的广义函数：

```
(defgeneric string->type (type value))
```

为了让一个特定的名字可被用作某个查询参数的类型名，你只需在 `string->type` 上定义一个方法。你将需要至少定义一个特化在符号 `string` 上的方法，因为这是缺省类型。当然，这很容易做到。由于浏览器有时会提交带有空字符串的表单，以表明某个特定的变量没有值被提供，因此你将需要像下列方法这样把空字符串转化成 `NIL`：

```
(defmethod string->type ((type (eql 'string)) value)
  (and (plusp (length value)) value))
```

你可以为你应用程序所需要的其他类型添加转换方法。例如，为了使 `integer` 成为一个可用的查询参数类型，从而可以处理 `random-page` 中的 `limit` 参数，你可以定义下列方法：

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))
```

另一个在 `param-binding` 所生成的代码中用到的助手函数是 `get-cookie-value`，它只是由 AllegroServe 所提供的 `get-cookie-values` 函数外围的一点儿语法糖。它看起来像下面这样：

```
(defun get-cookie-value (request name)
  (cdr (assoc name (get-cookie-values request) :test #'string=)))
```

类似地，用来计算查询参数和 cookie 名的函数也相当直接。

```
(defun symbol->query-name (sym)
  (string-downcase sym))

(defun symbol->cookie-name (function-name sym sticky)
  (let ((package-name (package-name (symbol-package function-name))))
    (when sticky
      (ecase sticky
        (:global
         (string-downcase sym))
        (:package
         (format nil "~(~a:~a~)" package-name sym))
        (:local
```

```
(format nil "~(~a:~a:~a~)" package-name function-name sym))))))
```

为了生成为粘滞性参数设置 cookie 的代码，你需要再次循环在参数列表上，但这一次只收集来自每个粘滞性参数的代码片段。你可以使用 `when` 和 `collect it` 这两个 LOOP 形式来只收集那些由 `set-cookie-code` 所返回的非空值。

```
(defun set-cookies-code (function-name request params)
  (loop for param in params
        when (set-cookie-code function-name request param) collect it))

(defun set-cookie-code (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (declare (ignore type default))
    (if sticky
        `(~(when ,name
              (set-cookie-header
                ,request
                :name ,(symbol->cookie-name function-name name sticky)
                :value (princ-to-string ,name)))))))
```

像这样用助手函数来定义宏的一大优点是很容易确保你所生成的单独代码看起来是正确的。例如，你可以检查下面的 `set-cookie-code`:

```
(set-cookie-code 'foo 'request '(x integer 20 :local))
```

是否生成了下面的代码:

```
(WHEN X
  (SET-COOKIE-HEADER REQUEST
    :NAME "com.gigamonkeys.web:foo:x"
    :VALUE (PRINC-TO-STRING X)))
```

假设这些代码将会出现在 `x` 为某个变量名的上下文中，那么它看起来是正确的。

再一次，宏允许你将你的代码蒸馏到它的精髓——在本例中，就是你想要从请求中解出的数据和你想要生成的 HTML。这就是说，该框架并不意味着一个大而全的 Web 应用框架——它只是一些可以让类似你将在第 29 章里所编写的那种简单应用更容易编写的语法糖罢了。

但在你到达那里之前，你还需要编写应用程序的功能性部分，然后让第 29 章的应用作为它的用户接口。在下一章里，你将开始编写一个之前在第 3 章里编写的那个数据库的增强版，这一次用来跟踪从 MP3 文件中解出的 ID3 数据。

第27章 实践：一个 MP3 数据库

在本章里，你将回顾在第 3 章里首次遇到的思想——从基本 Lisp 数据结构中构建出一个内存数据库。这次你的目标是保存那些你将使用来自第 25 章的 ID3v2 库从一组 MP3 文件中解出的信息。然后你将在第 28 和 29 章里使用这个数据库作为一个基于 Web 的流式 MP3 服务器的一部分。当然，这一次你可以使用那些自从第 3 章依赖你所学到的语言特性来构建一个更专业的版本了。

27.1 数据库

第 3 章的那个数据库的主要问题是它只有一个表，也就是保存在变量 `*db*` 中的列表。另一个问题是代码并不清楚关于保存在不同字段中值的类型的任何东西。在第 3 章里你避开了这个问题，通过使用相当通用的 `EQUAL` 方法来比较从数据库中选出的行的不同列的值，但如果你想要保存一些无法用 `EQUAL` 来比较的值或者想要对数据库的行排序的话就会遇到麻烦了，因为不存在像 `EQUAL` 那样通用的比较函数。

这次你将通过定义一个类 `table` 来表达单独的数据库表，从而同时解决上述两个问题。每一个 `table` 实例都由两个槽构成——一个用来保存表的数据，而另一个保存关于表中各列的信息以供各种数据库操作使用。这个类如下所示：

```
(defclass table ()  
  ((rows :accessor rows :initarg :rows :initform (make-rows))  
   (schema :accessor schema :initarg :schema)))
```

和第 3 章里一样，你可以使用 `plist` 来表示单独的行，但这一次你将创建一层抽象使得以后可以轻松调整实现细节而不会带来太多麻烦。并且这一次你将把行保存在一个向量而非列表中，因为你将要支持的特定操作，比如通过数值索引对行的随机访问以及排序一个表的能力，使用向量可以更为高效的实现。

用来初始化 `rows` 槽的函数 `make-rows` 可以简单地封装在 `MAKE-ARRAY` 之外，后者构建一个空的、可调整的、带有填充指针的向量。

相关的包

用于你将在本章中开发的代码的包如下所示：

```
(defpackage :com.gigamonkeys.mp3-database  
  (:use :common-lisp  
        :com.gigamonkeys.pathnames  
        :com.gigamonkeys.macro-utilities)
```

```
:com.gigamonkeys.id3v2)
(:export :*default-table-size*
         :*mp3-schema*
         :*mp3s*
         :column
         :column-value
         :delete-all-rows
         :delete-rows
         :do-rows
         :extract-schema
         :in
         :insert-row
         :load-database
         :make-column
         :make-schema
         :map-rows
         :matching
         :not-nullable
         :nth-row
         :random-selection
         :schema
         :select
         :shuffle-table
         :sort-rows
         :table
         :table-size
         :with-column-values))
```

其中的 :use 部分可以让你访问那些从第 14、8 和 25 章所定义的包中导出名字所对应的函数和宏，而 :export 部分导出了该库将要提供的 API，你将在第 29 章里用到它们。

```
(defparameter *default-table-size* 100)

(defun make-rows (&optional (size *default-table-size*))
  (make-array size :adjustable t :fill-pointer 0))
```

为了表示一个表的模式 (schema)，你需要定义另一个类 column，其每个实例都将含有关于表中一个列的信息：它的名字、如果比较列中值的等价性和顺序、缺省值，以及一个将在向表中插入数据或查询表时被用来正则化列中值的函数。schema 槽将保存一个 column 对象的列表。该类的定义如下所示：

```
(defclass column ()
  ((name
    :reader name
    :initarg :name)

  (equality-predicate
    :reader equality-predicate
    :initarg :equality-predicate)

  (comparator
    :reader comparator
    :initarg :comparator)

  (default-value
    :reader default-value
    :initarg :default-value
    :initform nil))
```

```
(value-normalizer
 :reader value-normalizer
 :initarg :value-normalizer
 :initform #'(lambda (v column) (declare (ignore column)) v))))
```

一个 `column` 对象的 `equality-predicate` 和 `comparator` 槽用来保存用作比较给定列中值的等价性和顺序的函数。这样，一个含有字符串值的列可以使用 `STRING=` 作为其 `equality-predicate` 的值，而用 `STRING<` 作为其 `comparator`，而当一个列含有数字时是可以使用 “=” 和 “<”。

`default-value` 和 `value-normalizer` 槽用在向数据库中插入行时，其中 `value-normalizer` 也用在查询数据库时。当你向数据库中插入新行时，如果特定的列没有提供值，那么你可以使用保存在 `column` 的 `default-value` 槽中的值。然后该值——无论缺省的还是其他——将通过连同列对象一起传递给保存在 `value-normalizer` 槽中的函数从而被正则化。你传递整个列以便 `value-normalizer` 函数可能需要使用与该列对象所关联的一些数据。（你将在下一节里看到一个相关的例子）你也应该正则化传递给查询的值，在将它们与数据库中的值进行比较之前。

这样，`value-normalizer` 的责任主要是返回一个可被安全和正确地传递给 `equality-predicate` 和 `comparator` 函数的值。如果 `value-normalizer` 不能找出一个适当的返回值，那么它将报错。

在你向数据库中保存值之前将其正则化的另一个理由是为了节省内存和 CPU 时钟周期。举个例子，如果你有一个打算包含字符串值的列，但将会保存在该列中的不同字符串的数量较少——例如 MP3 数据库中的风格列——那么你可以通过使用 `value-normalizer` 来 `intern` 这些字符串（将所有 `STRING=` 的值都转化成单个字符串对象）。这样，你只需要保存相当于所有不同的值那么多的字符串，无论表中会有多少行，并且你将可以使用 `EQL` 而非相对缓慢的 `STRING=` 来比较列中的这些值。¹

27.2 定义一个模式

这样，为了生成一个 `table` 实例，你需要构建一个 `column` 对象的列表。你可以手工构建这个列表，通过 `LIST` 和 `MAKE-INSTANCE`。但你将很快注意到你经常在生成许多带有同样比较器和等价谓词组合的列对象。这是因为一个比较器和等价谓词的组合本质上定义了一个列类型。如果可以有一种方式允许你只需给出这些类型的名字就可以让你表达出一个给定列是一个字符串列，而无需每次指定 `STRING<` 作为其比较符和 `STRING=` 作为其等价谓词就好了。一种方式是定义一个广义函数 `make-column`，如下所示：

```
(defgeneric make-column (name type &optional default-value))
```

现在你可以在这个广义函数上实现通过 `EQL` 特化符特化在 `type` 上的方法，并返回一个填充

¹ `Intern` 对象的一般理论是，如果你打算多次比较一个特定的值，那么值得花时间先 `intern` 它。

`value-normalizer` 在你将一个值插入到表中时运行一次，然后，如同你将要看到的，会在每个查询的开始运行一次。由于一个查询可能涉及到对表中的每一行都调用一次 `equality-predicate`，因此 `intern` 这些值的摊余成本将快速地收敛到零。

了适当值的 `column` 对象。下面是为类型名 `string` 和 `number` 定义列类型的广义函数和方法：

```
(defmethod make-column (name (type (eql 'string)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'string<
   :equality-predicate #'string=
   :default-value default-value
   :value-normalizer #'not-nulliable))

(defmethod make-column (name (type (eql 'number)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'<
   :equality-predicate #'=
   :default-value default-value))
```

下面的函数 `not-nulliable` 用作 `string` 列的 `value-normalizer`，它简单地返回给定值，除非它为 `NIL`，在后一种情况下它直接报错：

```
(defun not-nulliable (value column)
  (or value (error "Column ~a can't be null" (name column))))
```

这很重要，因为 `STRING<` 和 `STRING=` 如果在 `NIL` 上被调用的话将会报错；在有问题的值进入表之前将其捕捉到，比等到你试图使用它们时再报错要好很多。²

另一个你将在 MP3 数据库中用到的列类型是 `interned-string`，如同之前所讨论的那样，它的值是被 `intern` 过的。由于你需要一个哈希表来 `intern` 这些值，因此你应当定义一个 `column` 的子类 `interned-values-column`，它增加了一个槽以保存那个用来做 `intern` 操作的哈希表。

为了实现实际的 `intern` 过程，你还将需要为 `value-normalizer` 提供一个 `:initform` 以传递一个函数用来 `intern` 该列的 `interned-values` 哈希表中的值。并且由于 `intern` 这些值的一个主要原因是允许你使用 `EQL` 作为等价谓词，因此你还应该为 `equality-predicate` 添加一个值为 `#'eql` 的 `:initform`。

```
(defclass interned-values-column (column)
  ((interned-values
    :reader interned-values
    :initform (make-hash-table :test #'equal))
   (equality-predicate :initform #'eql)
   (value-normalizer :initform #'intern-for-column)))

(defun intern-for-column (value column)
  (let ((hash (interned-values column)))
    (or (gethash (not-nulliable value column) hash)
        (setf (gethash value hash) value))))
```

然后你可以定义一个特化在名字 `interned-string` 上的 `make-column` 方法来返回一

² 和通常一样，编程书籍中简要解说的最大牺牲品是正确的错误处理；在产品代码中你可能想要定义你自己的错误类型，例如下面这样的，然后报错时使用它：

```
(error 'illegal-column-value :value value :column column)
```

接下来你可能会考虑在哪里放置再启动以便可以从这个状况中恢复。并且，最终在给定的应用中你可以建立一些状况处理器在这些再启动中进行选择。

个 interned-values-column 的实例。

```
(defmethod make-column (name (type (eql 'interned-string)) &optional
default-value)
  (make-instance
   'interned-values-column
   :name name
   :comparator #'string<
   :default-value default-value))
```

有了定义在 make-column 上的这些方法，你现在可以定义一个函数 make-schema，它从包括列名、列类型名以及可选缺省值的列规范的列表中构建出一个 column 对象的列表。

```
(defun make-schema (spec)
  (mapcar #'(lambda (column-spec) (apply #'make-column column-spec)) spec))
```

例如，你可以为那个你将要用来保存从 MP3 中解出的数据的表定义类似下面这样的模式：

```
(defparameter *mp3-schema*
  (make-schema
   '(:file      string)
   (:genre     interned-string "Unknown")
   (:artist    interned-string "Unknown")
   (:album     interned-string "Unknown")
   (:song      string)
   (:track     number 0)
   (:year      number 0)
   (:id3-size  number))))
```

为了生成一个实际的表用来保存关于 MP3 的信息，你将 *mp3-schema* 作为 :schema 初始化参数传给 MAKE-INSTANCE。

```
(defparameter *mp3s* (make-instance 'table :schema *mp3-schema*))
```

27.3 插入值

现在你可以开始定义你的第一个表操作 insert-row 了，它接受一个由名字和值组成的 plist 以及一个表，然后在表中添加含有给定值的一行。大量的工作是在一个助手函数 normalize-row 中完成的，它为每个列构建了一个带有缺省值并正则化了的 plist，尽可能使用来自 names-and-values 的值，否则使用每个列的 default-value。

```
(defun insert-row (names-and-values table)
  (vector-push-extend (normalize-row names-and-values (schema table)) (rows
table)))

(defun normalize-row (names-and-values schema)
  (loop
    for column in schema
    for name = (name column)
    for value = (or (getf names-and-values name) (default-value column))
    collect name
    collect (normalize-for-column value column)))
```

值得定义一个单独的助手函数 normalize-for-column，它接受一个值和一个 column 对象并返回正则化的值，因为你将需要在查询参数上做同样的正则化处理。

```
(defun normalize-for-column (value column)
  (funcall (value-normalizer column) value column))
```

现在你可以将这些数据库代码与来自前面章节的代码组合起来构建一个从 MP3 文件中解出的数据的数据库了。你可以定义一个函数 `file->row`, 使用来自 ID3v2 库的 `read-id3` 从一个文件中解出一个 ID3 标签并将其转化成一个可以传给 `insert-row` 的 plist。

```
(defun file->row (file)
  (let ((id3 (read-id3 file)))
    (list
      :file   (namestring (truename file))
      :genre  (translated-genre id3)
      :artist (artist id3)
      :album   (album id3)
      :song    (song id3)
      :track   (parse-track (track id3))
      :year    (parse-year (year id3))
      :id3-size (size id3))))
```

你不必担心值的正则化问题, 因为 `insert-row` 可以替你处理这些事情。不过, 你确实需要将 `track` 和 `year` 返回的字符串值转化成数字。ID3 标签中的音轨号有时被保存成音轨号的 ASCII 表示, 有时则保存成一个数字后跟左斜杠, 然后是该专辑的音轨总数。由于你只关心实际的音轨号, 因此你应当使用 `PARSE-NUMBER` 的 `:end` 参数来指定它只解析到左斜杠之前的位置, 如果有的话。³

```
(defun parse-track (track)
  (when track (parse-integer track :end (position #\/ track))))
```



```
(defun parse-year (year)
  (when year (parse-integer year)))
```

最后, 你可以将所有这些函数放在一起, 再加上来自可移植路径名库的 `walk-directory` 和来自 ID3v2 库的 `mp3-p` 函数, 定义出一个函数: 它从给定目录里找出所有 MP3 文件并解出其中的数据, 然后再把这些数据加载到一个数据库中。

```
(defun load-database (dir db)
  (let ((count 0))
    (walk-directory
      dir
      #'(lambda (file)
          (princ #\.)
          (incf count)
          (insert-row (file->row file) db))
      :test #'mp3-p)
    (format t "~&Loaded ~d files into database." count)))
```

³ 如果任何 MP3 文件里在音轨和年代帧里带有格式错误的数据, 那么 `PARSE-INTEGER` 可能会报错。一种处理该问题的方式是传给 `PARSE-INTEGER` 一个值为 `T` 的 `:junk-allowed` 参数, 这将导致它忽略掉跟在数字后面的任何非数字的垃圾, 或是在字符串中没有数字时返回 `NIL`。或者, 如果你想实践一下对状况系统的使用, 你可以定义一个错误类型并在这些函数中当数据的格式错误时报错, 同时也建立一些再启动以允许这些函数得以恢复。

27.4 查询数据库

一旦你加载了带有数据的数据库，那么你将需要一种方式来查询它。对于 MP3 应用来说你将需要一个比你在第 3 章里所写的更加专业一些的查询函数。这一次你不仅要找出那些匹配特定条件的行，还要将结果限制在特定的一些列上，或是将结果限制在那些唯一的行上，同时还可能会在特定的列上排序这些行。为了保持关系型数据库的精髓，一个查询的结果将是一个含有想要的行和列的新 `table` 对象。

你即将编写的查询函数 `select` 很大程度上出自结构化查询语言（SQL）的 `SELECT` 语句。它将接受五个关键字参数：`:from`、`:columns`、`:where`、`:distinct` 和 `:order-by`。其中 `:from` 参数是你想要查询的 `table` 对象。`:column` 参数指定了哪些列应当包含在结果中。其值应当是一个列名字的列表、一个单独的列名，或者缺省值 `t` 表示返回所有的列。`:where` 参数如果被提供的话应当是一个函数，其接受一行并在该行应当被包含在结果中时返回真。你将很快编写两个函数 `matching` 和 `in`，它们可以返回适用于 `:where` 参数的函数。`:order-by` 参数如果被提供的话应当是一个列名的列表；结果将按照命名的列被排序。和 `:columns` 参数的情况一样，你可以只用一个名字来指定单一的列，这等价于一个含有同样名字的单元素列表。最后，`:distinct` 参数是一个布尔值，它表明是否需要从结果中清除重复的行。`:distinct` 的缺省值为 `NIL`。

下面是一些使用 `select` 的例子：

```
; Select all rows where the :artist column is "Green Day"
(select :from *mp3s* :where (matching *mp3s* :artist "Green Day"))

; Select a sorted list of artists with songs in the genre "Rock"
(select
  :columns :artist
  :from *mp3s*
  :where (matching *mp3s* :genre "Rock")
  :distinct t
  :order-by :artist)
```

`select` 和它直接用到的助手函数的实现如下所示：

```
(defun select (&key (columns t) from where distinct order-by)
  (let ((rows (rows from))
        (schema (schema from)))

    (when where
      (setf rows (restrict-rows rows where)))

    (unless (eql columns 't)
      (setf schema (extract-schema (mklist columns) schema)))
      (setf rows (project-columns rows schema)))

    (when distinct
      (setf rows (distinct-rows rows schema)))

    (when order-by
      (setf rows (sorted-rows rows schema (mklist order-by)))))

    (make-instance 'table :rows rows :schema schema)))

(defun mklist (thing)
```

```
(if (listp thing) thing (list thing))

(defun extract-schema (column-names schema)
  (loop for c in column-names collect (find-column c schema)))

(defun find-column (column-name schema)
  (or (find column-name schema :key #'name)
      (error "No column: ~a in schema: ~a" column-name schema)))

(defun restrict-rows (rows where)
  (remove-if-not where rows))

(defun project-columns (rows schema)
  (map 'vector (extractor schema) rows))

(defun distinct-rows (rows schema)
  (remove-duplicates rows :test (row-equality-tester schema)))

(defun sorted-rows (rows schema order-by)
  (sort (copy-seq rows) (row-comparator order-by schema)))
```

当然，`select` 中真正有趣的部分在于你如何实现函数 `extractor`、`row-equality-tester` 和 `row-comparator`。

通过它们的用法你可以看出，这些函数中的每一个都必须返回一个函数。例如，`project-columns` 使用由 `extractor` 所返回的值作为提供给 `MAP` 的函数参数。由于 `project-columns` 的目标是返回仅含有特定列值的一些行，因此你可以推断出 `extractor` 将返回一个接受一行作为参数的函数，并返回仅含有传递的模式中指定的那些列的一个新行。下面是可能实现它的方式：

```
(defun extractor (schema)
  (let ((names (mapcar #'name schema)))
    #'(lambda (row)
        (loop for c in names collect c collect (getf row c)))))
```

注意到你完成这项工作的方式——在闭包主体之外从模式中解出了所有的名字：由于闭包将被多次调用，因此你希望它在每次调用时可以尽可能少地做事。

函数 `row-equality-tester` 和 `row-comparator` 的实现方式很相似。为了决定两行是否等价，你需要将用于每一列的相应等价谓词应用在适当的列值上。回顾在第 22 章里 `LOOP` 子句 `always` 将在一对值测试失败以后立即返回 `NIL`，否则将导致整个 `LOOP` 返回 `T`。

```
(defun row-equality-tester (schema)
  (let ((names (mapcar #'name schema))
        (tests (mapcar #'equality-predicate schema)))
    #'(lambda (a b)
        (loop for name in names and test in tests
              always (funcall test (getf a name) (getf b name))))))
```

排序两行稍微更复杂一些。在 Lisp 中，比较操作符当它们的第一个参数应该排在第二个参数的前面时返回真，否则返回假。这样，一个 `NIL` 可能意味着第二个参数应该被排在第一个参数的前面，或者说明它们其实相等。你希望你的行比较器具有相同的行为：当第一个行应当排在第二个的前面时返回真，否则返回假。

这样，为了比较两个行，你应当比较来自用于排序的列中的值，其中采用每个列的对应比较

器。首先以来自第一个行的值作为第一个参数来调用比较器。如果比较器返回真，那就意味着第一行绝对应该排在第二个行的前面，所以你可以立即返回 T。

但如果列比较器返回了 NIL，那么你需要检测这是否是因为第二个值应当排在第一个值的前面，或是因为它们相等。因此你应当以相反的参数再次调用比较器。如果比较器这是返回了真，那么就意味着第二个列值排在了第一个列值的前面并且因此第二个行也应该排在第一个行的前面，所以你可以立即返回 NIL。否则，两个列值就是等价的，那么你应当继续比较下一个列。如果你通过了所有的列而始终没有遇到来自一个行的值赢得了比较，那么这两行就是等价，于是你返回 NIL。一个实现了该算法的函数如下所示：

```
(defun row-comparator (column-names schema)
  (let ((comparators (mapcar #'comparator (extract-schema column-names
schema))))
    #'(lambda (a b)
        (loop
          for name in column-names
          for comparator in comparators
          for a-value = (getf a name)
          for b-value = (getf b name)
          when (funcall comparator a-value b-value) return t
          when (funcall comparator b-value a-value) return nil
          finally (return nil))))
```

27.5 匹配函数

`select` 的 `:where` 参数可以是任何接受一个行对象并在该行应当被包括在结果中时返回真的函数。不过在实践中，你将很少需要用任意代码来表达查询条件的完全能力。因此你应当提供两个函数 `matching` 和 `in`，它们将用来构造查询函数从而允许你表达常用类型的查询并帮你处理每个列的正确等价性谓词和值正则化器的使用。

主力的查询函数构造器将是 `matching`，其返回一个将匹配带有给定列值的行的函数。你在早先的 `select` 例子里看到过它的用法。例如，下面这个对 `matching` 的调用：

```
(matching *mp3s* :artist "Green Day")
```

返回可以匹配`:artist` 值为“Green Day”的行的函数。你也可以传递多个名字和值；返回的函数当所有列都匹配时才算是匹配。例如，下面的例子返回了一个匹配艺术家为“Green Day”和专辑名为“American Idiot”的行的闭包：

```
(matching *mp3s* :artist "Green Day" :album "American Idiot")
```

你必须将整个表对象传给 `matching`，因为它需要访问表的模式以获得它所要匹配的那些列的等价谓词和值正则化器。

你可以从较小的函数中逐步构造出 `matching` 所返回的函数，其中每个底层函数负责匹配一个列的值。为了构造出这些函数，你需要定义一个函数 `column-matcher`，它接受一个 `column` 对象和一个你想要匹配的未经正则化的值，并返回一个接受单一行并在该行的给定列的值匹配给定值的正则化版本时返回真的函数。

```
(defun column-matcher (column value)
```

```
(let ((name (name column))
      (predicate (equality-predicate column))
      (normalized (normalize-for-column value column)))
  #'(lambda (row) (funcall predicate (getf row name) normalized))))
```

然后你可以为那些你所关心的名字和值构造出一个列匹配函数的列表，通过使用下列函数 `column-matchers`:

```
(defun column-matchers (schema names-and-values)
  (loop for (name value) on names-and-values by #'cddr
        when value collect
        (column-matcher (find-column name schema) value)))
```

现在你可以实现 `matching` 了。再次注意到你尽可能多地在闭包之外做事以确保有些事情只做一次而不是在表的每一行上都要做。

```
(defun matching (table &rest names-and-values)
  "Build a where function that matches rows with the given column values."
  (let ((matchers (column-matchers (schema table) names-and-values)))
    #'(lambda (row)
        (every #'(lambda (matcher) (funcall matcher row)) matchers))))
```

这个函数就像一个闭包的迷宫，但值得花点儿时间来思考一下作为第一类对象的函数给编程带来的可能性。

`matching` 的工作是返回一个函数，其将在一个表的每个行上被调用来检测其是否应被包含在新表中。因此，`matching` 返回一个带有单个参数 `row` 的闭包。

现在回顾函数 `EVERY` 可以接受一个谓词函数作为其第一个参数并当且仅当该函数应用在作为 `EVERY` 第二个参数传递进来的列表的每一个元素上均为真时才返回真。不过在本例中，你传递给 `EVERY` 的列表本身是一个由函数所组成的列表，那些列匹配器。你想要知道的是，当每个列匹配器在你当前测试的行上被调用时，是否每一个均返回真。因此，作为 `EVERY` 的谓词参数，你传递了另一个闭包给它，该闭包是向列匹配器传递当前行的 `FUNCALL` 调用。

另一个你偶尔会觉得有用的匹配函数是 `in`，其返回一个匹配那些特定列在给定的值集合中的行的函数。你将定义 `in` 来接受两个参数：一个列名和一个含有你想要匹配的那些值的列表。例如，假设你想要在 MP3 数据库中找出所有与 Dixie Chicks 的歌曲同名的歌曲。你可以像下面这样通过 `in` 和一个子 `select` 写出这个 `where` 字句:⁴

```
(select
  :columns '(:artist :song)
  :from *mp3s*
```

⁴ 这个查询也会返回所有 Dixie Chicks 的歌曲。如果你想把查询限制在 Dixie Chicks 之外的艺术家的歌曲上，那么你需要一个更复杂的 `:where` 函数。由于 `:where` 参数可以是任何函数，所以这确实是可能的；你可以通过下列查询移除 Dixie Chicks 自己的歌曲：

```
(let* ((dixie-chicks (matching *mp3s* :artist "Dixie Chicks"))
      (same-song
        (in :song (select :columns :song :from *mp3s* :where dixie-chicks)))
      (query
        #'(lambda (row)
            (and (not (funcall dixie-chicks row)) (funcall same-song row)))))
  (select :columns '(:artist :song) :from *mp3s* :where query))
```

这样显然不是很方便。如果你打算编写一个需要做很多复杂查询的应用程序，那么你可能会考虑设计更加复杂的查询语言。

```
:where (in :song
  (select
    :columns :song
    :from *mp3s*
    :where (matching *mp3s* :artist "Dixie Chicks"))))
```

尽管查询更复杂了，但 `in` 本身的定义却比 `matching` 要简单的多。

```
(defun in (column-name table)
  (let ((test (equality-predicate (find-column column-name (schema table)))))
    (values (map 'list #'(lambda (r) (getf r column-name)) (rows table)))
      #'(lambda (row)
        (member (getf row column-name) values :test test))))
```

27.6 获得结果

由于 `select` 返回了另一个 `table`，因此你思考一下如何才能得到一个表中单独的行和列值。如果你确定你将永不改变你在一张表中表达数据的方式，那么你可以直接把表结构作为 API 的一部分——就是说该 `table` 中带有一个 `rows` 槽，类型为 `plist` 构成的向量——然后使用所有正常的 Common Lisp 函数操作向量和 `plist` 来得到表中的值。但这些表示可能确实是以后会改变的内部细节。另外，你也不希望其他代码可以直接操作这些数据结构——例如，你不希望任何人使用 `SETF` 在一个行中放置一个未经正则化的列值。因此定义一些抽象来提供你想要支持的操作就可能是个好主意了。然后如果你决定以后改变表的内部表示，你将只需要改变这些函数和宏的实现。并且，尽管 Common Lisp 并不能允许你绝对避免人们获得“内部”数据，但通过提供一个官方 API 你至少可以清楚地表明边界在哪里。

也许你将需要对一个查询结果做的最常见的事情就是迭代在单独的行上并解出特定列的值。因此你需要提供一种方式来同时做到这两件事而无需直接碰 `rows` 向量或是用 `GETF` 来获取一个行中的列值。

目前这些操作都很容易实现；它们几乎就是包装在没有这些抽象时你所编写的代码之上的。你可以提供两种在一个表的行上迭代的方式：一个宏 `do-rows` 用来提供基本的循环构造，以及一个函数 `map-rows` 可以构造出含有将一个函数应用在表的每一行时所得到的结果的列表。⁵

```
(defmacro do-rows ((row table) &body body)
  `(loop for ,row across (rows ,table) do ,@body))

(defun map-rows (fn table)
  (loop for row across (rows table) collect (funcall fn row)))
```

为了得到一行中个别列的值，你应该提供一个函数 `column-value`，它接受一个行和一个列名并返回对应的值。再一次，这只是对你本该自行编写的代码的简单封装。但如果你以后改变了这个表的内部表示，那么 `column-value` 的用户可以不必受到影响。

```
(defun column-value (row column-name)
  (getf row column-name))
```

⁵ 在 Common Lisp 被标准化以前，M. I. T. 所实现的 LOOP 版本含有一种用来扩展 LOOP 语法以支持迭代在新数据结构上的机制。一些从该代码树上继承了 LOOP 实现的 Common Lisp 实现可能仍然支持这一功能，从而使 `do-rows` 和 `map-rows` 不再有必要了。

尽管 `column-value` 对于获取列的值来说是个高效的实现，但你将经常想要一次性得到多个列的值。因此你可以提供一点儿语法糖，一个宏 `with-column-values`，它将一组变量绑定到通过适当的关键字名从一个行中解出的值上。这样，代替下面的写法：

```
(do-rows (row table)
  (let ((song (column-value row :song))
        (artist (column-value row :artist))
        (album (column-value row :album)))
    (format t "~a by ~a from ~a~%" song artist album)))
```

你可以简单地写成下面这样：

```
(do-rows (row table)
  (with-column-values (song artist album) row
    (format t "~a by ~a from ~a~%" song artist album)))
```

再一次，实际的实现并不复杂，如果你使用来自第 8 章 `once-only` 宏。

```
(defmacro with-column-values ((&rest vars) row &body body)
  (once-only (row)
    `(let ,(column-bindings vars row) ,@body)))

(defun column-bindings (vars row)
  (loop for v in vars collect `',(v (column-value ,row ,(as-keyword v)))))

(defun as-keyword (symbol)
  (intern (symbol-name symbol) :keyword))
```

最后，你应当提供抽象来获取一个表中所有行的个数以及通过数值索引来访问指定行的能力。

```
(defun table-size (table)
  (length (rows table)))

(defun nth-row (n table)
  (aref (rows table) n))
```

27.7 其他数据库操作

最后，你将实现其他一些将在第 29 章里用到的数据库操作。前两个是 SQL `DELETE` 语句的相似物。函数 `delete-rows` 被用来从一个表中删除匹配特定条件的行。和 `select` 一样，它接受 `:from` 和 `:where` 关键字参数。和 `select` 不同的是，它并不返回一个新表——它实际修改了作为 `:from` 参数传递的表。

```
(defun delete-rows (&key from where)
  (loop
    with rows = (rows from)
    with store-idx = 0
    for read-idx from 0
    for row across rows
    do (setf (aref rows read-idx) nil)
    unless (funcall where row) do
      (setf (aref rows store-idx) row)
      (incf store-idx)
    finally (setf (fill-pointer rows) store-idx)))
```

出于对效率的兴趣，你可能想要提供一个单独的函数用来从一个表中删除所有的行。

```
(defun delete-all-rows (table)
  (setf (rows table) (make-rows *default-table-size*)))
```

其余的表操作并没有映射到正常的关系型数据库操作中，但将在 MP3 浏览器应用中非常有用。首先是一个在表中直接排序所有行的函数。

```
(defun sort-rows (table &rest column-names)
  (setf (rows table)
        (sort (rows table) (row-comparator column-names (schema table))))
  table)
```

另一方面，在 MP3 浏览器应用中，你将需要一个函数直接在表中打乱所有的行，它用到了来自第 23 章的 nshuffle-vector。

```
(defun shuffle-table (table)
  (nshuffle-vector (rows table))
  table)
```

而最后，再次处于 MP3 浏览器的目的，你应当提供一个函数来选择 n 个随机行，然后作为新表返回。它也用到了 nshuffle-vector，另外还有一个版本的 random-sample，后者基于我在第 20 章里讨论过的 Donald Knuth 的《The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 第 3 版》(Addison-Wesley, 1998)。

```
(defun random-selection (table n)
  (make-instance
   'table
   :schema (schema table)
   :rows (nshuffle-vector (random-sample (rows table) n)))

(defun random-sample (vector n)
  "Based on Algorithm S from Knuth. TAOCP, vol. 2. p. 142"
  (loop with selected = (make-array n :fill-pointer 0)
        for idx from 0
        do
        (loop
         with to-select = (- n (length selected))
         for remaining = (- (length vector) idx)
         while (>= (* remaining (random 1.0)) to-select)
         do (incf idx))
        (vector-push (aref vector idx) selected)
        when (= (length selected) n) return selected))
```

有了这些代码你就可以在第 29 章里构建一个用于浏览 MP3 文件集合的一个 Web 接口。但在你到达那里之前，你还需要实现服务器中使用 Shoutcast 协议流式播放 MP3 的部分，这是下一章的主题。

第28章 Shoutcast 服务器

在本章里，你将开发基于 Web 的流式 MP3 应用的另一个重要部分，也就是实际向诸如 iTunes、XMMS¹ 和 Winamp 等客户端发送 MP3 流的 Shoutcast 协议。

28.1 Shoutcast 协议

Shoutcast 协议是由 Nullsoft 的人发明的，他们也是 Winamp MP3 软件的作者。它被设计用来支持 Internet 音频广播——Shoutcast DJ 从他们的个人电脑将音频数据发送到一个中央的 Shoutcast 服务器上，后者转身再把它以流的形式发送到任何已连接的听众那里。

你即将构建的服务器实际上只是半个真实的 Shoutcast 服务器——你将使用 Shoutcast 服务器所使用的协议来流式传送 MP3 到听众，但你的服务器将只能提供那写已经保存在服务器所运行在的计算机的文件系统中的歌曲。

你只需要关心 Shoutcast 协议的两部分：一个客户端开始接收一个流时所产生的请求，以及回执的格式，其中包括将当前正在播/放歌曲的元数据嵌入到流中的机制。

从 MP3 客户端到 Shoutcast 服务器的初始请求被格式化成了一个正常的 HTTP 请求。在回应部分，Shoutcast 服务器发送了 ICY 回执，后这看起来就像是一个 HTTP 回执，除了字符串“ICY”² 出现在正常 HTTP 版本字符串的位置上并带有不同的头。在发送了头和一个空行之后，服务器开始流式发送数量潜在无止境的 MP3 数据。

关于 Shoutcast 协议的唯一难点是那些正在流式发送的歌曲的元数据被嵌入在发给客户端的数据中的方式。Shoutcast 设计者所面临的问题是提供一种方式让 Shoutcast 服务器可以在每次开始播放一首新歌时与客户端沟通新的标题信息，这样客户端可以将其显示在 UI 里。（回顾第 25 章里说 MP3 格式本身并不提供对编码元数据的支持。）尽管 ID3v2 的一个设计目标是使其更适合用在流式 MP3 中，但 Nullsoft 的人们还是决定走他们自己的路线并从头发明了一种在服务器和客户端都相当容易实现的新格式。这对他们来说也是理想的，因为他们也是其自己的 MP3 客户端的作者。

¹ RedHat 8.0 和 9.0 以及 Fedora 中附带的 XMMS 版本不再知道如何播放 MP3 了，这是因为 Red Hat 的人对 MP3 相关的解码器存在担忧。为了在这些版本的 Linux 上得到带有 MP3 支持的 XMMS，你需要从 <http://www.xmms.org> 获取源代码并自行编译它。或者，对于其他可能性可以参见 <http://www.fedorafaq.org/#xmms-mp3> 上的信息。

² 为了进一步产生混淆，还存在一个不同的称为 Icecast 的流协议。看起来在 Shoutcast 和 Icecast 协议所使用的 ICY 头之间不存在明显的关联。

他们的方法简单地忽略了 MP3 数据的结构并在每 n 个字节里嵌入一个分界的元数据片段。客户端有义务分离出这些元数据使其不被视为 MP3 数据。由于发送到不支持该格式的客户端的元数据将导致杂音的出现，因此服务器仅在客户端的原始请求中含有一个特殊的 Icy-Metadata 头时才发送元数据。并且为了让客户端知道元数据的发送频率，服务器必须发回一个 Icy-Metainit 头，其值为每个相邻的元数据片段之间被发送的 MP3 数据的字节数。

元数据的基本内容是一个形如 "StreamTitle='title';" 的字符串，其中的 title 是当前歌曲的标题并且不能带有单引号。这一载荷采用定长的字节数组来编码：一个单字节被发送以指示接下来有多少个 16 字节的块，然后这些块再被发送。它们中含有作为 ASCII 字符串的字符串载荷，其中最后一个块使用必要的空字节作为补白。

这样，最小的合法元数据片段是单个字节零，其代表零个后续块。如果服务器不需要更新元数据，那么它可以发送这样的一个空片段，但它至少发送一个字节才能让客户端不会丢掉实际的 MP3 数据。

28.2 歌曲源

由于一个 Shoutcast 服务器必须在客户端连接的情况下保持流式发送歌曲，因此你需要为你的服务器提供一个进行操作的歌曲来源。在基于 Web 的应用中，每个连接的客户端都将拥有一个可以通过 Web 接口管理的播放列表。但考虑到为了避免过度耦合，你应当定义一个接口让 Shoutcast 服务器用来获得播放的歌曲。你可以现在编写一个该接口的简单实现，然后在第 29 章里构建一个更复杂的作为 Web 应用的一部分。

包定义

用于你将在本章里开发的代码的包如下所示：

```
(defpackage :com.gigamonkeys.shoutcast
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.id3v2)
  (:export :song
           :file
           :title
           :id3-size
           :find-song-source
           :current-song
           :still-current-p
           :maybe-move-to-next-song
           :*song-source-type*))
```

该接口背后的思想是，Shoutcast 服务器将根据从 AllegroServe 请求对象中解出的 ID 来查找歌曲源。然后它可以对给定的歌曲源做三件事：

- 获得歌曲源中的当前歌曲
- 告诉歌曲源当前歌曲结束

□ 询问歌曲源是否之前给出的某个歌曲仍是当前歌曲

最后一个操作是必要的，因为可能存在某种方式——第 29 章里就会这样做——在 Shoutcast 服务器之外管理歌曲源。你可以用下列广义函数来表达 Shoutcast 服务器所需的操作：

```
(defgeneric current-song (source)
  (:documentation "Return the currently playing song or NIL."))

(defgeneric still-current-p (song source)
  (:documentation
   "Return true if the song given is the same as the current-song."))

(defgeneric maybe-move-to-next-song (song source)
  (:documentation
   "If the given song is still the current one update the value
  returned by current-song."))

```

函数 `maybe-move-to-next-song` 如此定义可以允许用单一操作来检查一首歌曲是否为当前歌曲，并且如果是的话就将歌曲源移向下一首歌曲。这种设计对于下一章里当你需要实现一个可以安全地从两个不同线程管理的歌曲源时将会很重要。³

为了表示 Shoutcast 服务器所需要的关于一首歌曲的信息，你可以定义一个类 `song`，其槽用来保存 MP3 文件的名字、在 Shoutcast 元数据中发送的标题，以及 ID3 标签的大小用来在发送文件时调过标签部分。

```
(defclass song ()
  ((file :reader file :initarg :file)
   (title :reader title :initarg :title)
   (id3-size :reader id3-size :initarg :id3-size)))
```

由 `current-song`（从而也就是 `still-current-p` 和 `maybe-move-to-next-song` 的第一个参数）所返回的值将是一个 `song` 的实例。

此外，你需要定义一个广义函数，让服务器可以基于想要的歌曲源类型和请求对象来查找一个歌曲源。其方法将特化在 `type` 参数上以便返回不同类型的歌曲源，并且从请求对象中将所需的信息取出用来检测应返回哪个源。

```
(defgeneric find-song-source (type request)
  (:documentation "Find the song-source of the given type for the given request."))

```

不过，对于本章的目标，你可以使用一个该接口的简单实现，让其总是使用相同的对象，一个可从 REPL 管理的歌曲对象的简单队列。你可以从定义一个类 `simple-song-queue` 和一个保存该类的一个实例的全局变量 `*songs*` 开始。

```
(defclass simple-song-queue ()
  ((songs :accessor songs :initform (make-array 10 :adjustable t :fill-pointer
  0))
   (index :accessor index :initform 0)))

(defparameter *songs* (make-instance 'simple-song-queue))
```

然后你可以定义一个 `find-song-source` 之上的通过符号 `singleton` 上的 `EQL` 特化符特化

³ 技术上讲，本章的实现也可以从两个线程管理——运行着 Shoutcast 服务器的 AllegroServe 线程和 REPL 线程。但目前你可以接受竞争状况。我将在下一章里讨论如何用锁来确保代码是线程安全的。

在 type 上的方法，其返回保存在 *songs* 中的那个实例。

```
(defmethod find-song-source ((type (eql 'singleton)) request)
  (declare (ignore request))
  *songs*)
```

现在你只需实现 Shoutcast 服务器将会用到的三个广义函数上的方法就可以了。

```
(defmethod current-song ((source simple-song-queue))
  (when (array-in-bounds-p (songs source) (index source))
    (aref (songs source) (index source)))

(defmethod still-current-p (song (source simple-song-queue))
  (eql song (current-song source)))

(defmethod maybe-move-to-next-song (song (source simple-song-queue))
  (when (still-current-p song source)
    (incf (index source))))
```

另外出于测试的目的你可以提供一种方式向队列中添加歌曲。

```
(defun add-file-to-songs (file)
  (vector-push-extend (file->song file) (songs *songs*)))

(defun file->song (file)
  (let ((id3 (read-id3 file)))
    (make-instance
     'song
     :file (namestring (truename file))
     :title (format nil "~a by ~a from ~a" (song id3) (artist id3) (album id3))
     :id3-size (size id3)))
```

28.3 实现 Shoutcast

现在你可以开始实现 Shoutcast 服务器了。由于 Shoutcast 协议很大程度上基于 HTTP，因此你可以将该服务器实现成 AllegroServe 中的一个函数。不过，由于你需要与 AllegroServe 的一些底层特性交互，因此你不能使用来自第 26 章的 define-url-function 宏。代替地，你需要编写一个像下面这样的正规函数：

```
(defun shoutcast (request entity)
  (with-http-response
    (request entity :content-type "audio/MP3" :timeout *timeout-seconds*)
    (prepare-icy-response request *metadata-interval*)
    (let ((wants-metadata-p (header-slot-value request :icy-metadata)))
      (with-http-body (request entity)
        (play-songs
         (request-socket request)
         (find-song-source *song-source-type* request)
         (if wants-metadata-p *metadata-interval*)))))
```

然后像下面这样将该函数发布在路径 /stream.mp3 下：⁴

```
(publish :path "/stream.mp3" :function 'shoutcast)
```

⁴ 当编写这些代码时，另一件你可能想做的事情将是求值形式 (net.aserve::debug-on :notrap)。这告诉 AllegroServe 不要捕捉由你代码所抛出的异常，这可以允许你在一个正常的 Lisp 调试器中调试它们。在 SLIME 中这将会弹出一个 SLIME 调试器，和其他任何错误的情况一样。

在 `with-http-response` 调用中，除了通常的 `request` 和 `entity` 参数以外，你还需要传递 `:content-type` 和 `:timeout` 参数。其中 `:content-type` 参数告诉 AllegroServe 如何设置它所发送的 Content-Type 头。而 `:timeout` 参数指定了 AllegroServe 给该函数用来生成回执的秒数。缺省情况下 AllegroServe 判定每个请求在五分钟后超时。由于你打算通过流发送一个本质上无止境的 MP3 序列，因此你需要更长得多的时间。没有办法告诉 AllegroServer “永不” 超时一个请求，所以你应当将这个时间设置成 `*timeout-seconds*` 的值，其中你可以定义一些诸如 10 年的秒数这类相当大的值。

```
(defparameter *timeout-seconds* (* 60 60 24 7 52 10))
```

然后，在 `with-http-response` 的主体中导致回执头被发送的 `with-http-body` 调用之前的地方，你需要处理 AllegroServe 将要发送的回执。函数 `prepare-icy-response` 封装了必要的处理：将协议字符串从缺省的“HTTP”改为“ICY”并添加了 Shoutcast 特定的头。⁵为了处理 iTunes 中的一个 bug，你还需要告诉 AllegroServe 不要使用分段传输编码（`chunked transfer-encoding`）。⁶ 函数 `request-reply-protocol-string`、`request-uri` 和 `reply-header-slot-value` 都是 AllegroServe 的一部分。

```
(defun prepare-icy-response (request metadata-interval)
  (setf (request-reply-protocol-string request) "ICY")
  (loop for (k v) in (reverse
    `(((:|icy-metaint| ,(princ-to-string metadata-interval))
      (:|icy-notice1| "<BR>This stream blah blah blah<BR>")
      (:|icy-notice2| "More blah")
      (:|icy-name| "MyLispShoutcastServer")
      (:|icy-genre| "Unknown")
      (:|icy-url| ,(request-uri request))
      (:|icy-pub| "1")))
    do (setf (reply-header-slot-value request k) v)))
  ;; iTunes, despite claiming to speak HTTP/1.1, doesn't understand
  ;; chunked Transfer-encoding. Grrr. So we just turn it off.
  (turn-off-chunked-transfer-encoding request))

(defun turn-off-chunked-transfer-encoding (request)
  (setf (request-reply-strategy request)
    (remove :chunked (request-reply-strategy request))))
```

在函数 `shoutcast` 的 `with-http-body` 中，你实际流出的是 MP3 数据。函数 `play-songs` 接受用来发送数据的流、歌曲源，以及应当使用的元数据间隔或者在客户端不想要元数据时为 `NIL`。该流是从请求对象中获取的 `socket`，歌曲源通过调用 `find-song-source` 获取到，而元数据间隔则来之全局变量 `*metadata-interval*`。歌曲源的类型由变量 `*song-source-type*` 来控制，目前你可以将其设置成 `singleton` 以使用你之前实现的 `simple-song-queue`。

```
(defparameter *metadata-interval* (expt 2 12))
(defparameter *song-source-type* 'singleton)
```

⁵ Shoutcast 头通常以小写字母发送，因此你需要转义那些用来在 AllegroServe 中标识它们的关键字符的名字，从而避免让 Lisp 读取器将它们全部转换成大写。这样，你应当写成：`:|icy-metaint|` 而不是 `:icy-metaint`。你还可以写成：`\i\c\y-\m\e\t\a\i\n\t`，但这样非常难看。

⁶ 函数 `turn-off-chunked-transfer-encoding` 使了一点儿诡计。没有办法在不指定内容长度的情况下通过 AllegroServe 的官方 API 来关闭分段传输编码，因为任何声称支持 HTTP/1.1 的客户端都应当理解它，包括 iTunes 在内。但这段代码做到了。

函数 `play-songs` 本身并不做太多事——它循环调用函数 `play-current`, 后者干所有的粗活, 包括发送单个 MP3 文件的内容、跳过 ID3 标签, 以及嵌入 ICY 元数据。唯一的亮点是你需要跟踪何时发送元数据。

由于你必须以特定的间隔来发送元数据片段, 而与你何时碰巧从一个 MP3 文件切换到下一个文件无关, 因此每次调用 `play-current` 时你需要告诉它下一个元数据何时到期, 而当它返回时, 它必须告诉你相同的事情, 这样才能将该信息传递到下一个 `play-current` 调用里。如果 `play-current` 从歌曲源里得到了 `NIL`, 那么它也返回 `NIL`, 这允许 `play-songs` 中的循环得以停下来。

除了处理循环以外, `play-songs` 还提供了一个 `HANDLE-CASE` 来捕捉当 MP3 客户端从服务器上断开并导致对 `socket` 的写入失败时在 `play-current` 中抛出的错误。由于 `HANDLER-CASE` 在 `LOOP` 之外, 对错误的处理将中断循环, 从而允许 `play-songs` 返回。

```
(defun play-songs (stream song-source metadata-interval)
  (handler-case
    (loop
      (for next-metadata = metadata-interval
           then (play-current
                  stream
                  song-source
                  next-metadata
                  metadata-interval)
           while next-metadata)
      (error (e) (format *trace-output* "Caught error in play-songs: ~a" e))))
```

最终, 你可以实现 `play-current` 了, 它用来实际发送 Shoutcast 数据。基本思想是, 你从歌曲源里得到当前的歌曲, 打开该歌曲的文件, 然后循环地从文件中读取数据并写入到 `socket` 中, 直到你要么遇到文件结尾, 要么当前歌曲不再是当前歌曲了。

这里只有两个复杂之处: 一个是你需要确保在正确的间隔上发送元数据。另一个是如果文件以一个 ID3 标签开始, 那么你需要跳过它。如果你不过多地考虑 I/O 性能, 那么你可以像下面这样来实现 `play-current`:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song))))
        (with-open-file (mp3 (file song))
          (unless (file-position mp3 (id3-size song))
            (error "Can't skip to position ~d in ~a" (id3-size song) (file song)))
          (loop for byte = (read-byte mp3 nil nil)
                while (and byte (still-current-p song song-source)) do
                  (write-byte byte out)
                  (decf next-metadata)
                when (and (zerop next-metadata) metadata-interval) do
                  (write-sequence metadata out)
                  (setf next-metadata metadata-interval))

                (maybe-move-to-next-song song song-source)))
        next-metadata)))
```

该函数从歌曲源中得到当前歌曲, 并得到一个含有将要通过传递标题给 `make-icy-metadata` 来发送的元数据的缓冲区。然后它打开文件并使用两参数形式的 `FILE-POSITION` 跳过 ID3 标签。

然后它开始从文件中读取字节并将它们写到请求的流中。⁷

当到达文件的结尾或是当歌曲源的当前歌曲发生改变时，循环将被中断。同时，无论何时 `next-metadata` 得到了零（如果你被允许发送元数据），那么它就将 `metadata` 写入到流中并重置 `next-metadata`。一旦它完成了循环，它会检查歌曲是否仍是歌曲源的当前歌曲；如果是的话，那么这意味着它是因为读取了整个文件才跳出的循环，在这种情况下它告诉歌曲源移动到下一首歌上。否则，它跳出循环是因为某个改变了当前正在播放的歌曲，那么函数就只是返回。无论哪种情况，它都返回在下一个元数据到期前剩余的字节数以便传给 `play-current` 的下一次调用。⁸

函数 `make-icy-metadata` 接受当前歌曲的标题并生成一个含有正确格式化的 ICY 元数据片段的字节数组，它的实现也是相当直接的。⁹

```
(defun make-icy-metadata (title)
  (let* ((text (format nil "StreamTitle='~a';" (substitute #\Space #\' title)))
         (blocks (ceiling (length text) 16))
         (buffer (make-array (1+ (* blocks 16))
                           :element-type '(unsigned-byte 8)
                           :initial-element 0)))
    (setf (aref buffer 0) blocks)
    (loop
      for char across text
      for i from 1
      do (setf (aref buffer i) (char-code char)))
    buffer))
```

取决于你的具体 Lisp 实现是如何处理它的流的，以及你需要一次服务多少个 MP3 客户端，这个简单版本的 `play-current` 可能足够高效，也可能不是。

这个简单实现的潜在问题是不得不为你所传输的每个字节调用 `READ-BYTE` 和 `WRITE-BYTE`。有可能每个调用都产生相对成本高昂的系统调用来读写一个字节。并且即便 Lisp 实现了自己的带有内部缓冲区的流，从而并非每个 `READ-BYTE` 和 `WRITE-BYTE` 都产生系统调用，函数调用本身的成本仍然存在。特别的是，在使用所谓的 Gray Streams 提供用户可扩展流的实现里，`READ-BYTE` 和 `WRITE-BYTE` 可能导致对广义函数的调用，后者在底层派发到流参数的类上。因为广义函数派发通常足够高效从而你不必担心它们，但它还是比一个非广义的函数调用成本更高一些，因此如果能够避免的话，你绝不想在几分钟里对其调用数百万次。

实现 `play-current` 的一种更高效但也更复杂的方式是使用 `READ-SEQUENCE` 和 `WRITE-SEQUENCE` 来一次性读写多个字节。你也给你一个机会将你的文件读取操作与文件系统的自然块大小相匹配，从而为你带来最佳的磁盘吞吐量。当然，无论你使用多大的缓冲区，跟踪

⁷ 多数 MP3 播放软件都会在用户接口的某个地方显示元数据。不过，Linux 上的 XMMS 程序缺省不这样做，为了让 XMMS 显示 Shoutcast 元数据，需要按 Ctrl+P 打开 Preferences 面板。然后在 Audio I/O Plugins 标签栏（在版本 1.2.10 下是最左边的标签），选择 MPEG Layer 1/2/3 Player (`libmpg123.so`) 并按下 Configure 按钮。然后选择配置窗口中的“流”标签，并在标签底部的 SHOUTCAST/Icecast 部分里，选中“Enable SHOUTCAST/Icecast title streaming”复选框。

⁸ 那些从 Common Lisp 迁移到 Scheme 的人们可能想知道为什么 `play-current` 不是递归地调用其自身。在 Scheme 中这确实工作得很好，因为 Scheme 实现被 Scheme 规范要求必须支持“无限次活跃尾递归”（unbounded number of active tail calls）。Common Lisp 实现也被允许带有这一属性，但这不是语言标准所要求的。因此，在 Common Lisp 中习惯上使用一个循环构造来编写循环，而不是使用递归。

⁹ 和你所编写的其他代码一样，这个函数假设你的 Lisp 实现的内部字符编码方式是 ASCII 或 ASCII 的一个超集，因此你可以使用 `CHAR-CODE` 将 Lisp 的 `CHARACTER` 对象转化成 ASCII 数据的字节。

何时发送元数据都将变得更复杂一些。一个 使用了 READ-SEQUENCE 和 WRITE-SEQUENCE 的更高效的 play-current 版本可能如下所示：

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song)))
            (buffer (make-array size :element-type '(unsigned-byte 8))))
        (with-open-file (mp3 (file song))
          (labels ((write-buffer (start end)
                    (if metadata-interval
                        (write-buffer-with-metadata start end)
                        (write-sequence buffer out :start start :end end)))

                  (write-buffer-with-metadata (start end)
                    (cond
                      ((> next-metadata (- end start))
                       (write-sequence buffer out :start start :end end)
                       (decf next-metadata (- end start)))
                      (t
                        (let ((middle (+ start next-metadata)))
                          (write-sequence buffer out :start start :end middle)
                          (write-sequence metadata out)
                          (setf next-metadata metadata-interval)
                          (write-buffer-with-metadata middle end)))))

                  (multiple-value-bind (skip-blocks skip-bytes)
                      (floor (id3-size song) (length buffer))

                    (unless (file-position mp3 (* skip-blocks (length buffer)))
                      (error "Couldn't skip over ~d ~d byte blocks."
                            skip-blocks (length buffer)))

                    (loop for end = (read-sequence buffer mp3)
                          for start = skip-bytes then 0
                          do (write-buffer start end)
                          while (and (= end (length buffer))
                                     (still-current-p song song-source)))

                    (maybe-move-to-next-song song song-source))))))
      next-metadata)))
```

现在你可以将所有东西放在一起了。在下一章里你将编写一个本章所开发的 Shoutcast 服务器的 Web 接口，它使用来自第 27 章的 MP3 数据库作为歌曲源。

第29章 一个 MP3 浏览器

构建 MP3 流应用的最后一步是提供一个 Web 接口从而允许用户查找他们想听的歌曲并且将其添加到一个播放列表中，这样当用户的 MP3 客户端请求该流的 URL 时 Shoutcast 服务器将会播放指定的歌曲。为了开发应用的这个组件，你需要把来自前面几章的一些代码整合起来：MP3 数据库、来自第 26 章的 `define-url-function`，当然还有 Shoutcast 服务器本身。

29.1 播放列表

接口背后的基本思想是每个 MP3 客户端连接到 Shoutcast 服务器上来获取它们自己的播放列表（playlist），其作为 Shoutcast 服务器所需的歌曲源。一个播放列表还将提供超出 Shoutcast 服务器需要之外的功能：用户将通过 Web 接口来向播放列表中添加歌曲，删除已在播放列表中的歌曲，以及通过排序和乱序来重新调整播放列表。

你可以像下面这样来定义一个表示播放列表的类：

```
(defclass playlist ()
  ((id           :accessor id           :initarg :id)
   (songs-table  :accessor songs-table  :initform (make-playlist-table))
   (current-song :accessor current-song :initform *empty-playlist-song*)
   (current-idx  :accessor current-idx  :initform 0)
   (ordering     :accessor ordering    :initform :album)
   (shuffle      :accessor shuffle    :initform :none)
   (repeat       :accessor repeat    :initform :none)
   (user-agent   :accessor user-agent :initform "Unknown")
   (lock         :reader lock      :initform (make-process-lock))))
```

一个播放列表的 `id` 是其关键字，你从请求对象中解出它并传递给 `find-song-source` 来查询一个播放列表。你实际上并不需要将其保存在 `playlist` 对象中，但这可以使调试更加方便，如果你可以从一个任意的播放列表对象中找出它的 `id` 是什么的话。

播放列表的心脏是 `songs-table` 槽，它用来保存一个 `table` 对象。用于这个表的模式将和用于主 MP3 数据库的模式相同。用来初始化 `songs-table` 的函数 `make-playlist-table` 简单地如下所示：

```
(defun make-playlist-table ()
  (make-instance 'table :schema *mp3-schema*))
```

包定义

你可以使用下列 DEFPACKAGE 来定义用于本章中代码的包：

```
(defpackage :com.gigamonkeys.mp3-browser
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.html
        :com.gigamonkeys.shoutcast
        :com.gigamonkeys.url-function
        :com.gigamonkeys.mp3-database
        :com.gigamonkeys.id3v2)
  (:import-from :acl-socket
                :ipaddr-to-dotted
                :remote-host)
  (:import-from :multiprocessing
                :make-process-lock
                :with-process-lock)
  (:export :start-mp3-browser))
```

由于这是一个高阶应用，因此它用到了许多底层包。它还从 ACL-SOCKET 包中导入了三个符号，并从 MULTIPROCESSING 包中导入了其余两个，这是因为它只需要这两个包中导出的 5 个符号而不需要其他的 139 个符号。

通过将歌曲的列表保存在一个表中，你可以使用来自第 27 章的数据库函数来操作播放列表：你可以通过 `insert-row` 向播放列表中添加，用 `delete-rows` 删除歌曲，以及用 `sort-rows` 和 `shuffle-table` 重排播放列表。

`current-song` 和 `current-idx` 槽用来跟踪当前正在播放哪首歌曲：`current-song` 是实际的 `song` 对象，而 `current-idx` 是 `song-table` 中代表当前歌曲的行的索引。你将在“操作播放列表”一节中看到如何在每当 `current-idx` 改变时确保更新 `current-song`。

`ordering` 和 `shuffle` 槽保存关于 `songs-table` 中的歌曲顺序的信息。其中 `ordering` 槽保存一个关键字用来告诉 `songs-table` 在其不是乱序时应当怎样排序。合法的值包括 `:genre`、`:artist`、`:album` 和 `:song`。`shuffle` 槽保存下列关键字之一：`:none`、`:song` 或 `:album`，其指定了 `songs-table` 应当如何被乱序。

`repeat` 槽也保存一个关键字，`:none`、`:song` 或 `:all` 之一，其指定了播放列表的重复模式。如果 `:repeat` 是 `:none`，那么在 `songs-table` 的最后一首歌播放完以后，`current-song` 回滚到一个缺省的 MP3 上。当 `:repeat` 为 `:song` 时，播放列表将不断地返回到同样的 `current-song` 上。而如果是 `:all` 的话，在最后一首歌结束以后 `current-song` 将回到第一首歌上。

`user-agent` 槽保存 MP3 客户端在其对流的请求中发送的 User-Agent 头。你纯粹是为了 Web 接口才保存这个值的——User-Agent 头标识了产生请求的程序，因此你可以将该值显示在列出所有播放列表的页面上从而容易看出当有多个用户连接时播放列表与连接的对应关系。

最后，`lock` 槽中保存了一个由函数 `make-process-lock` 创建的“进程锁”，该函数是 Allegro 的 MULTIPROCESSING 包的一部分。你将需要在操作 `playlist` 对象的特定函数中用到这个锁，以确保每次只有一个线程在操作给定的播放列表对象。你可以定义下面的宏来容易地包装一组需要在保持一个播放列表锁的情况下进行处理的代码，该宏是从来自 MULTIPROCESSING 的 `with-process-lock` 宏构建出来的：

```
(defmacro with-playlist-locked ((playlist) &body body)
```

```
`(with-process-lock ((lock ,playlist))
  ,@body))
```

其中的 `with-process-lock` 宏要求获得对给定进程锁的排他访问，然后再执行其主体形式，最后再释放锁。缺省情况下 `with-process-lock` 允许递归加锁，这意味着同一个线程可以安全地对同一个锁对象加锁多次。

29.2 作为歌曲源的播放列表

为了将 `playlist` 用作 Shoutcast 服务器的歌曲源，你将需要在来自第 28 章的广义函数 `find-song-source` 上实现一个方法。由于你打算拥有多个播放列表，因此你需要一种方式来为连接到服务器的每个客户端找出正确的那个播放列表来。具体的映射部分很简单——你可以定义一个变量来保存一个 `EQUAL` 哈希表并用它来将一些标识符映射到 `playlist` 对象上。

```
(defvar *playlists* (make-hash-table :test #'equal))
```

你还将需要定义一个进程锁来保护对这个哈希表的访问，如下所示：

```
(defparameter *playlists-lock* (make-process-lock :name "playlists-lock"))
```

然后定义一个函数根据给定 ID 来查询一个播放列表，如果必要的话就创建一个新的 `playlist` 对象并用 `with-process-lock` 来确保每次只有一个线程在操作哈希表。¹

```
(defun lookup-playlist (id)
  (with-process-lock (*playlists-lock*)
    (or (gethash id *playlists*)
        (setf (gethash id *playlists*) (make-instance 'playlist :id id)))))
```

然后你就可以在该函数和另一个函数 `playlist-id` 的基础上实现 `find-song-source` 了，其接受一个 AllegroServe 请求对象并返回适当的播放列表标识符。`find-song-source` 函数也负责从请求对象中抓取 User-Agent 字符串并保存在播放列表对象中。

```
(defmethod find-song-source ((type (eql 'playlist)) request)
  (let ((playlist (lookup-playlist (playlist-id request))))
    (with-playlist-locked (playlist)
      (let ((user-agent (header-slot-value request :user-agent)))
        (when user-agent (setf (user-agent playlist) user-agent))))
```

接下来的难点是如何实现 `playlist-id`，一个从请求对象中解出标识符的函数。你有很多选项，每个分别对应于不同的用户接口实现。你可以从请求对象中取得任何你想要的信息，但无论你决定怎样标识一个客户端，你都需要一些方式让 Web 接口的用户与正确的播放列表关联在一起。

目前你可以采用一个“勉强可用”的方法，只要每台连接到服务器的机器上只有一个 MP3

¹ 并发编程的复杂度超出了本书的范围。基本的思想是，如果你有多个控制线程——就像在当前的应用里这样，一些线程运行 `shoutcast` 函数而另一些线程回应来自浏览器的请求——那么你需要确保每次只有一个线程在操作给定的某个对象以避免一个线程工作在该对象时另一个线程看到了不一致的状态。例如，在当前这个函数中，如果两个新的 MP3 客户端正在同时连接，它们都试图添加一项到 `*playlists*` 中，那么这有可能互相影响。`with-process-lock` 确保了每个线程都可以获得对哈希表的排他访问以便有足够长的时间来完成它们想做的事。

客户端并且浏览 Web 接口的用户就来自运行着 MP3 客户端的那台机器：你将使用客户机的 IP 地址作为标识符。通过这种方式你可以为一个请求找出正确的播放列表而无论该请求是来自 MP3 客户端还是一个 Web 浏览器。尽管如此，你将在 Web 接口中提供一种方式来从浏览器中选择一个不同的播放列表，因此这一选择在应用上施加的实际约束是每个客户端 IP 地址上只能有一个连接的 MP3 客户端。² `playlist-id` 的实现如下所示：

```
(defun playlist-id (request)
  (ipaddr-to-dotted (remote-host (request-socket request))))
```

函数 `request-socket` 是 AllegroServe 的一部分，而 `remote-host` 和 `ipaddr-to-dotted` 都是 Allegro 的 `socket` 库的一部分。

为了创建一个可被 Shoutcast 服务器用作歌曲源的播放列表，你需要在 `current-song`、`still-current-p` 和 `maybe-move-to-next-song` 上定义将 `source` 参数特化在 `playlist` 上的方法。`current-song` 方法已经准备好了：通过在 `current-song` 槽上定义同名的访问函数，你自动地获得了特化在 `playlist` 上的可返回该槽的值的 `current-song` 方法。不过，为了使对 `playlist` 的访问是线程安全的，你需要在访问 `current-song` 槽之前锁定该 `playlist`。在本例中，最简单的方法是像下面这样定义一个 `:around` 方法：

```
(defmethod current-song :around ((playlist playlist))
  (with-playlist-locked (playlist) (call-next-method)))
```

实现 `still-current-p` 也很简单，假设你确保 `current-song` 只有在当前的歌曲实际发生了改变时才被更新到新的 `song` 对象上。再一次，你需要获取一个进程锁以确保你可以对 `playlist` 的状态得到一致的视图。

```
(defmethod still-current-p (song (playlist playlist))
  (with-playlist-locked (playlist)
    (eql song (current-song playlist))))
```

剩下的难点是确保 `current-song` 槽在正确的时间得到更新。当前的歌曲有几种改变的方式。最明显的一种是当 Shoutcast 服务器调用 `maybe-move-to-next-song` 时。但它还可以在歌曲被添加到播放列表时更新，比如说当 Shoutcast 服务器播完了所有歌曲，或者甚至是在播放列表的重复模式被改变时。

与其试图编写代码特定于上述每种情形来检测是否更新 `current-song`，你还可以定义一个函数 `update-current-if-necessary`，它会在 `current-song` 中的 `song` 对象不再匹配 `current-idx` 槽对应的当前应播放文件时更新 `current-song`。然后，如果你在任何对可能导致这两个槽不同步的播放列表操作之后调用该函数，你就可以确保 `current-song` 总是被正确设置了。下面是 `update-current-if-necessary` 和它的助手函数：

```
(defun update-current-if-necessary (playlist)
  (unless (equal (file (current-song playlist))
                 (file-for-current-idx playlist))
    (reset-current-song playlist)))

(defun file-for-current-idx (playlist)
```

² 这种方法也假设了每个客户机都有独立的 IP 地址。这个假设在所有用户都在同一个 LAN 下是成立的，但如果用户是来自一个做网络地址转换的防火墙之后的话就不成立了，但如果你想要将这个应用部署在更广的 Internet 上的话，你最好对网络有足够的理解从而找出最适合你自己的方法。

```
(if (at-end-p playlist)
    nil
    (column-value (nth-row (current-idx playlist) (songs-table
playlist)) :file))

(defun at-end-p (playlist)
  (>= (current-idx playlist) (table-size (songs-table playlist))))
```

你不需要为这些函数加锁，因为它们将只在那些预先加锁过播放列表的函数中被调用。

函数 `reset-current-song` 引入了又一个亮点：由于你想要播放列表对客户端提供无休止的 MP3 流，因此你不希望 `current-song` 被设置成 `NIL`。代替地，当一个播放列表没有歌曲可播时——当 `songs-table` 为空或是当 `repeat` 设置成 `:none` 时最后一首歌已经播完了——你需要将 `current-song` 设定在一个其文件为 MP3 静音³ 的特殊歌曲上，并且其标题要能够解释为何没有歌曲在播放。下面的一些代码定义了两个参数，`*empty-playlist-song*` 和 `*end-of-playlist-song*`，每个都被设置成一个以 `*silence-mp3*` 所命名的文件作为文件并带有适当标题的歌曲：

```
(defparameter *silence-mp3* ...)

(defun make-silent-song (title &optional (file *silence-mp3*))
  (make-instance
   'song
   :file file
   :title title
   :id3-size (if (id3-p file) (size (read-id3 file)) 0)))

(defparameter *empty-playlist-song* (make-silent-song "Playlist empty."))
(defparameter *end-of-playlist-song* (make-silent-song "At end of playlist.))
```

`reset-current-song` 会在 `current-idx` 没有指向 `songs-table` 的任何一行时使用这些参数。否则，它将 `current-song` 设置成代表当前行的一个 `song` 对象。

```
(defun reset-current-song (playlist)
  (setf
   (current-song playlist)
   (cond
     ((empty-p playlist) *empty-playlist-song*)
     ((at-end-p playlist) *end-of-playlist-song*)
     (t (row->song (nth-row (current-idx playlist) (songs-table playlist)))))))

(defun row->song (song-db-entry)
  (with-column-values (file song artist album id3-size) song-db-entry
    (make-instance
     'song
     :file file
     :title (format nil "~a by ~a from ~a" song artist album)
     :id3-size id3-size)))

(defun empty-p (playlist)
  (zerop (table-size (songs-table playlist))))
```

³ 不幸的是，由于 MP3 格式的授权问题，我不太确定在没有向 Fraunhofer IIS 付费的情况下提供给你一个这样的 MP3 是否合法。我的这个 MP3 来自 Slim Devices 的 Slimp3 的配套软件的一部分。你可以通过 Web 从他们的 Subversion 库中获得它，地址是 http://svn.slimdevices.com/*checkout*/trunk/server/HTML/EN/html/silentpacket.mp3?rev=2。或者可以买一个 Squeezebox，Slimp3 的新的无线版本，然后你将作为随机软件的一部分得到 `silentpacket.mp3`。或者还可以查找 John Cage 的一个 4' 33'' 的 MP3。

现在，最后你可以实现 `maybe-move-next-song` 上的方法，基于播放列表的重复模式将 `current-idx` 移到其下一个值上，然后调用 `update-current-if-necessary`。当 `current-idx` 已在播放列表的结尾时你不需要改变 `current-idx`，因为你希望它保持在当前值上，这样它就可以指向你添加到播放列表的下一首歌。这个函数必须在操作播放列表前锁定它，因为它是被 Shoutcast 服务器代码所调用的，而后者并没有做任何锁定。

```
(defmethod maybe-move-to-next-song (song (playlist playlist))
  (with-playlist-locked (playlist)
    (when (still-current-p song playlist)
      (unless (at-end-p playlist)
        (ecase (repeat playlist)
          (:song) ; nothing changes
          (:none (incf (current-idx playlist)))
          (:all (setf (current-idx playlist)
                      (mod (1+ (current-idx playlist))
                            (table-size (songs-table playlist)))))))
      (update-current-if-necessary playlist))))
```

29.3 操作播放列表

播放列表代码的其余部分被 Web 接口用来操作 `playlist` 对象，包括添加和删除歌曲，排序和乱序，以及设置重复模式。和上一节的那些助手函数一样，你不需要在这些函数中担心锁定问题，因为，如同你将要看到的，锁将被调用它们的 Web 接口函数所获取。

添加和删除基本上是一个 `songs-table` 的管理问题。你唯一需要做的额外工作是保持 `current-song` 和 `current-idx` 同步。例如，无论何时播放列表为空，它的 `current-idx` 都将是零，而 `current-song` 将是 `*empty-playlist-song*`。如果你向空的播放列表里添加一首歌曲，那么这个零索引将在范围内，因此你应该将 `current-song` 改变成新添加的歌曲。同样的情况，当你已经播放完一个播放列表中的所有歌曲且 `current-song` 为 `*end-of-playlist-song*`，这时添加一首歌应导致 `current-song` 被重置。所有这些实际意味着，你需要在适当的点上调用 `update-current-if-necessary`。

由于 Web 接口沟通所需添加歌曲的方式，向播放列表添加歌曲的过程有点儿复杂。出于我将在下一节里讨论的一些原因，Web 接口代码无法只是给你一个简单的用来从数据库中选择歌曲的条件集合。代替地，它给你一个列的名字和一个值的列表，然后你被期望从主数据库中添加给定列具有值列表中某个值的所有歌曲。这样，为了添加正确的歌曲，你需要首选构造一个含有想要值的表对象，然后你可以将它和一个歌曲数据库上的 `in` 查询一起使用。因此，`add-songs` 看起来像下面这样：

```
(defun add-songs (playlist column-name values)
  (let ((table (make-instance
                'table
                :schema (extract-schema (list column-name) (schema *mp3s*)))))
    (dolist (v values) (insert-row (list column-name v) table))
    (do-rows (row (select :from *mp3s* :where (in column-name table)))
      (insert-row row (songs-table playlist))))
    (update-current-if-necessary playlist)))
```

删除歌曲会简单一些；你只需从 `songs-table` 中删除匹配特定条件的歌曲——无论是一个特

定歌曲还是一个特定风格、特定艺术家或来自特定专辑的所有歌曲。因此，你可以提供一个 `delete-song` 函数，其接受一些键值对，后者用来构造你可以传给 `delete-rows` 数据库函数的一个基于 `matching` 的 `:where` 子句。

当删除歌曲时出现的另一点复杂之处是 `current-idx` 可能需要改变。假设当前歌曲并非刚刚删除的歌曲之一，那么你希望它仍旧是当前歌曲。但如果 `songs-table` 中在它之前歌曲被删除，在删除以后它将处在表中的一个不同的位置上。因此在一个 `delete-rows` 调用之后，你需要查看含有当前歌曲的行并重设 `current-idx`。如果当前歌曲本身被删除了，那么如果没有其他法子，你可以将 `current-idx` 重设到零。在更新了 `current-idx` 之后，调用 `update-current-if-necessary` 将会处理 `current-song` 的更新。而如果 `current-idx` 改变了却仍然指向了同一首歌，那么 `current-song` 将保持不变。

```
(defun delete-songs (playlist &rest names-and-values)
  (delete-rows
   :from (songs-table playlist)
   :where (apply #'matching (songs-table playlist) names-and-values))
  (setf (current-idx playlist) (or (position-of-current playlist) 0))
  (update-current-if-necessary playlist))

(defun position-of-current (playlist)
  (let* ((table (songs-table playlist))
         (matcher (matching table :file (file (current-song playlist)))))
    (pos 0))
  (do-rows (row table)
    (when (funcall matcher row)
      (return-from position-of-current pos)))
  (incf pos))))
```

你还可以提供一个函数来完全清空播放列表，它使用 `delete-all-rows` 并且不再需要查找当前歌曲，因为当前歌曲明显也要被删除。对 `update-current-if-necessary` 的调用将使得 `current-song` 设置到 `empty-playlist-song` 上。

```
(defun clear-playlist (playlist)
  (delete-all-rows (songs-table playlist))
  (setf (current-idx playlist) 0)
  (update-current-if-necessary playlist))
```

排序和乱序播放列表是彼此相关的操作，因为播放列表总是要么排序的要么乱序的。`shuffle` 槽表明是否播放列表应当被乱序，以及如果是的话该怎样做。如果它被设置为 `:none`，那么播放列表将按照 `ordering` 槽的值来排序。当 `shuffle` 是 `:song` 时，播放列表将被随机的调整顺序。而当它被设置成 `:album` 时，专辑的列表将被随机调整顺序，但每个专辑中的歌曲仍然以音轨的顺序列出。这样，当用户选择一个新的顺序时 Web 接口代码所调用的 `sort-playlist` 函数，其需要在调用实际完成排序工作的 `order-playlist` 之前将 `ordering` 设置成想要的顺序而将 `shuffle` 设置成 `:none`。和 `delete-songs` 里的情况一样，你需要使用 `position-of-current` 来重设 `current-idx` 到当前歌曲的新位置。不过，这时你不需要调用 `update-current-if-necessary`，因为你知道当前歌曲仍然在表中。

```
(defun sort-playlist (playlist ordering)
  (setf (ordering playlist) ordering)
  (setf (shuffle playlist) :none)
  (order-playlist playlist)
  (setf (current-idx playlist) (position-of-current playlist)))
```

在 `order-playlist` 中，你可以使用数据库函数 `sort-rows` 来实际进行排序，基于 `ordering` 的值传递一个列的列表来进行排序。

```
(defun order-playlist (playlist)
  (apply #'sort-rows (songs-table playlist)
    (case ordering
      (:genre '(:genre :album :track))
      (:artist '(:artist :album :track))
      (:album '(:album :track))
      (:song '(:song))))
```

当用户选择一个新的乱序模式时 Web 接口代码所调用的函数 `shuffle-playlist` 以类似的方式工作，只是它不需要改变 `ordering` 的值。这样，当使用一个 `:none` 的 `shuffle` 来调用 `shuffle-playlist` 时，播放列表将根据最近一次的排序状态进行排序。按歌曲乱序比较简单——只需在 `songs-table` 上调用 `shuffle-table`。按专辑乱序稍微复杂一些但也没什么大不了的。

```
(defun shuffle-playlist (playlist shuffle)
  (setf (shuffle playlist) shuffle)
  (case shuffle
    (:none (order-playlist playlist))
    (:song (shuffle-by-song playlist))
    (:album (shuffle-by-album playlist)))
  (setf (current-idx playlist) (position-of-current playlist)))

(defun shuffle-by-song (playlist)
  (shuffle-table (songs-table playlist)))

(defun shuffle-by-album (playlist)
  (let ((new-table (make-playlist-table)))
    (do-rows (album-row (shuffled-album-names playlist))
      (do-rows (song (songs-for-album playlist (column-value album-row :album)))
        (insert-row song new-table)))
    (setf (songs-table playlist) new-table)))

(defun shuffled-album-names (playlist)
  (shuffle-table
    (select
      :columns :album
      :from (songs-table playlist)
      :distinct t)))

(defun songs-for-album (playlist album)
  (select
    :from (songs-table playlist)
    :where (matching (songs-table playlist) :album album)
    :order-by :track))
```

你需要支持的最后一个操作是设置播放列表的重复模式。多数时候你在设置 `repeat` 时不需要做任何额外的操作——它的值只在 `maybe-move-to-next-song` 中用到。不过，你需要在一种情况下作为改变 `repeat` 的结果来更新 `current-song`，换句话说，如果 `current-idx` 位于一个非空播放列表的结尾，同时 `repeat` 从 `:song` 改变成 `:all`。在这种情况下，你希望可以继续播放，要么重复最后一首歌曲，要么从播放列表的起始处开始。因此，你应该在广义函数 (`setf repeat`) 上定义一个 `:after` 方法。

```
(defmethod (setf repeat) :after (value (playlist playlist))
  (if (and (at-end-p playlist) (not (empty-p playlist)))
```

```
(ecase value
  (:song (setf (current-idx playlist) (1- (table-size (songs-table
playlist))))))
  (:none)
  (:all (setf (current-idx playlist) 0)))
  (update-current-if-necessary playlist)))
```

现在你有了你需要的所有底层支持。其余的代码将只是提供一个基于 Web 的用户接口来浏览 MP3 数据库和操作播放列表了。这个接口将由 3 个通过 `define-url-function` 定义的主函数构成：一个用于浏览歌曲数据库，一个用于查看和管理单个播放列表，最后一个用来列出所有可用的播放列表。

但在你开始编写这三个函数之前，你还需要从它们将用到的一些助手函数和 HTML 宏开始。

29.4 查询参数类型

由于你将使用 `define-url-function`，因此你需要在来自第 28 章的 `string->type` 广义函数上定义一些方法，使得 `define-url-function` 可以用来将字符串查询参数转化成 Lisp 对象。在当前的应用下，你将需要一些方法来将字符串分别转化成整数、关键字符以及一个值的列表。

前两个方法很简单。

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))

(defmethod string->type ((type (eql 'keyword)) value)
  (and (plusp (length value)) (intern (string-upcase value) :keyword)))
```

最后一个 `string->type` 方法稍微更复杂一些。出于我即将谈到的一些原因，你将需要生成页面来显示一个表单，其中含有一个隐含字段，其值是一个字符串的列表。由于你要负责生成这个隐含字段中的值并且当它提交回来以后还要解析它，因此你可以使用任何觉得方便的编码方式。你可以使用函数 `WRITE-TO-STRING` 和 `READ-FROM-STRING`，它们使用 Lisp 的打印机和读取器向字符串中写入数据，以及从字符串中读取数据，除非字符串的打印表示中可能含有引号和其他在嵌入到一个 `INPUT` 元素的值属性时可能带来问题的字符。因此，你需要以某种方式转义这些字符。与其试图引入你自己的转义策略，你还可以直接使用 base 64，一种通常用来保护通过 e-mail 发送的二进制数据的编码方式。AllegroServe 带有两个函数 `base64-encode` 和 `base64-decode`，它们可以为你做 base 64 的编码和解码，因此你所要做的就只是编写一对函数：一个用来编码 Lisp 对象，先用 `WRITE-TO-STRING` 将其转化成可读的字符串然后在对其进行 base 64 编码，另一个用来从上述编码的字符串中进行 base 64 解码然后再把结果传给 `READ-FROM-STRING`。你将需要把对 `WRITE-TO-STRING` 和 `READ-FROM-STRING` 的调用包装在 `WITH-STANDARD-IO-SYNTAX` 中以确保所有可能影响打印机和读取器的变量都被设置在它们的标准值上。不过，由于你打算读取来自网络的数据，你必然希望关掉读取器的一个特性——在读取过程中求值任意 Lisp 代码的能力！⁴ 你可以定义你自己的宏 `with-safe-io-syntax`，它将其主体形式包装在一个将 `*READ-EVAL*` 绑

⁴ 读取器支持一种语法 “#.”，它导致接下来的 S-表达式在读取期被求值。这在源代码中偶尔会有用，但显然会在你读取不可信任的数据时打开了一个巨大的安全漏洞。不过，你可以通过将 `*READ-EVAL*` 设置为 `NIL` 来关闭该语法，这将导致读取器在它遇到 “#.” 时报错。

定到 NIL 的 LET 外围的 WITH-STANDARD-IO-SYNTAX 里。

```
(defmacro with-safe-io-syntax (&body body)
  `(^ (with-standard-io-syntax
        (let ((*read-eval* nil))
          ,@body)))
```

然后编码和解码函数就很容易写了。

```
(defun obj->base64 (obj)
  (base64-encode (with-safe-io-syntax (write-to-string obj)))))

(defun base64->obj (string)
  (ignore-errors
    (with-safe-io-syntax (read-from-string (base64-decode string)))))
```

最终，你可以使用这些函数来定义一个 `string->type` 上的方法，为查询参数类型 `base64-list` 定义转换方法。

```
(defmethod string->type ((type (eql 'base-64-list)) value)
  (let ((obj (base64->obj value)))
    (if (listp obj) obj nil)))
```

29.5 样板 HTML

接下来你需要定义一个 HTML 宏和助手函数，以便让应用中的不同页面获得一致的外观。你可以从一个定义了应用中页面基本结构的 HTML 宏开始。

```
(define-html-macro :mp3-browser-page ((&key title (header title)) &body body)
  `(:html
    (:head
      (:title ,title)
      (:link :rel "stylesheet" :type "text/css" :href "mp3-browser.css"))
    (:body
      (standard-header)
      (when ,header (html (:h1 :class "title" ,header)))
      ,@body
      (standard-footer))))
```

出于两个理由，你应该将 `standard-header` 和 `standard-footer` 定义成单独的函数。首先，在开发过程中你可能会不断重定义这些函数并立即观察其效果而不许重新编译那些使用了 `:mp3-browser-page` 宏的函数。其次，可以看出你以后编写的某些页面将不会使用 `:mp3-browser-page` 但却可能仍然需要标准的页头和页脚。该宏如下所示：

```
(defparameter *r* 25)

(defun standard-header ()
  (html
    ((:p :class "toolbar")
     "[" (:a :href (link "/browse" :what "genre") "All genres") "] "
     "[" (:a :href (link "/browse" :what "genre" :random *r*) "Random genres") "]"
     "
     "[" (:a :href (link "/browse" :what "artist") "All artists") "] "
     "[" (:a :href (link "/browse" :what "artist" :random *r*) "Random artists") "]"
     "
     "[" (:a :href (link "/browse" :what "album") "All albums") "] "
     "[" (:a :href (link "/browse" :what "album" :random *r*) "Random albums") "]")
```

```

"
  "[" (:a :href (link "/browse" :what "song" :random *r*)) "Random songs") "
"
  "[" (:a :href (link "/playlist") "Playlist") "] "
  "[" (:a :href (link "/all-playlists") "All playlists") "]"))
)

(defun standard-footer ()
  (html
    (:hr)
    ((:p :class "footer") "MP3 Browser v" *major-version* "." *minor-version*)))

```

一些较小的 HTML 宏和助手函数自动化了其他一些常用的模式。HTML 宏 `:table-row` 可以让生成 HTML 中表的单行更加容易。它使用了 `FOO` 的一个我将在第 31 章里讨论的特性，一个 `&attributes` 参数，它导致任何收集到一个列表中并绑定到 `&attributes` 参数上的属性可被宏作为正常 S-表达式 HTML 形式来解析。它看起来像下面这样：

```
(define-html-macro :table-row (&attributes attrs &rest values)
  `(:tr ,@attrs ,@(loop for v in values collect `(:td ,v))))
```

另一个 `link` 函数用来生成可用作 A 元素的 HREF 属性的应用内部 URL，它可从一组键值对中构造出一个查询字符串并确保所有的特殊字符都被正确的转义。例如，代替下面的写法：

```
(:a :href "browse?what=artist&genre=Rhythm+Blues" "Artists")
```

你可以写成下面这样：

```
(:a :href (link "browse" :what "artist" :genre "Rhythm & Blues") "Artists")
```

该函数如下所示：

```
(defun link (target &rest attributes)
  (html
    (:attribute
      (:format "~a~@[?~{~(~a~)=-a~^&~}~]" target (mapcar #'urlencode
        attributes)))))
```

为了编码用于 URL 的键和值，你用到了助手函数 `urlencode`，这是一个包装在函数 `encode-form-urlencoded` 上的函数，后者是来自 AllegroServe 的一个非公开的函数。这种做法一方面并不是很好；由于名字 `encode-form-urlencoded` 并非 `NET.ASERVE` 所导出的名字，因此 `encode-form-urlencoded` 将来有可能消失或在你不知道的情况下被重命名。在另一方面，使用这个没有导出的函数可以让你立刻完成手头的工作；通过将 `encode-form-urlencoded` 封装在你自己的函数中，你将有风险的代码隔离在了一个函数里，将来如果需要的话你可以重写它。

```
(defun urlencode (string)
  (net.aserve::encode-form-urlencoded string))
```

最后，你需要 `:mp3-browser-page` 用到的 CSS 样式表 `mp3-browser.css`。由于它并非动态的，因此最简单的方法就是用 `publish-file` 发布一个静态文件。

```
(publish-file :path "/mp3-browser.css" :file filename :content-type "text/css")
```

一个示例的样式表被包含在了本书 Web 站点上可下载到的本章配套源代码中。你将在本章结尾处定义一个函数来启动这个 MP3 浏览器应用。它将在完成其他工作的同时顺便发布这个文件。

29.6 浏览页

第一个 URL 函数将生成一个用来浏览 MP3 数据库的页面。它的查询参数将告诉它用户正在浏览什么类型的东西并提供他们所感兴趣的数据库元素的查询条件。它将给你一种方式来查询匹配一个特定风格、艺术家或专辑的数据库项。为了增加奇遇的可能性，你还可以提供一种方式来选择匹配项的一个随机子集。当用户在单个歌曲的层面上浏览时，歌曲的标题将是一个导致该歌曲被添加到播放列表的链接。否则，每个项都将被表示成可以让用户浏览通过其他分类所列出的项的链接。例如，如果用户正在浏览风格，其中的项“Blues”将含有浏览所有带有 Blues 风格的专辑、艺术家和歌曲。更进一步，浏览页面里还带有一个“Add all”按钮可将匹配页面中所给条件的每一首歌曲都添加到该用户的播放列表中。该函数如下所示：

```
(define-url-function browse
  (request (what keyword :genre) genre artist album (random integer))

  (let* ((values (values-for-page what genre artist album random))
         (title (browse-page-title what random genre artist album))
         (single-column (if (eql what :song) :file what))
         (values-string (values->base-64 single-column values)))
    (html
     (:mp3-browser-page
      (:title title)
      (:form :method "POST" :action "playlist")
      (:input :name "values" :type "hidden" :value values-string)
      (:input :name "what" :type "hidden" :value single-column)
      (:input :name "action" :type "hidden" :value :add-songs)
      (:input :name "submit" :type "submit" :value "Add all"))
     (:ul (do-rows (row values) (list-item-for-page what row)))))))
```

这个函数首先使用函数 `values-for-page` 来获得一个含有它需要表示的值的列表。当用户按歌曲来浏览时——这时 `what` 参数为 `:song`——你需要从数据库中选择完成的行。但是当用户按风格、艺术家或专辑名来浏览时，你将只想选择给定分类中不同的值。数据库函数 `select` 完成了几乎所有的重活儿，其中 `values-for-page` 多数时候负责根据 `what` 的值来向 `select` 传递正确的参数。后者也是你用来在必要时选择一个匹配行的随机子集的地方。

```
(defun values-for-page (what genre artist album random)
  (let ((values
         (select
          :from *mp3s*
          :columns (if (eql what :song) t what)
          :where (matching *mp3s* :genre genre :artist artist :album album)
          :distinct (not (eql what :song))
          :order-by (if (eql what :song) '(:album :track) what))))
        (if random (random-selection values random) values)))
```

为了生成浏览页的标题，你可以将浏览条件传递给下列函数 `browse-page-title`：

```
(defun browse-page-title (what random genre artist album)
  (with-output-to-string (s)
    (when random (format s "~:(~r~) Random " random))
    (format s "~:(~a~p~)" what random)
    (when (or genre artist album)
      (when (not (eql what :song)) (princ " with songs" s))
      (when genre (format s " in genre ~a" genre))
      (when artist (format s " by artist ~a" artist))))
```

```
(when album (format s " on album ~a" album))))
```

一旦你有了想要表示的那些值，你需要对它们做两件事。当然，主要的任务是将它们表示出来，这主要是由 `do-rows` 循环来做的，然后把每行的渲染工作交给 `list-item-for-page`。该函数以一种方式来渲染用于 `:song` 的行，而用另一种方式来渲染其他类型的行。

```
(defun list-item-for-page (what row)
  (if (eql what :song)
      (with-column-values (song file album artist genre) row
        (html
         (:li
          (:a :href (link "playlist" :file file :action "add-songs") (:b song))
          " from "
          (:a :href (link "browse" :what :song :album album) album)
          " by "
          (:a :href (link "browse" :what :song :artist artist) artist)
          " in genre "
          (:a :href (link "browse" :what :song :genre genre) genre))))
      (let ((value (column-value row what)))
        (html
         (:li value " - "
              (browse-link :genre what value)
              (browse-link :artist what value)
              (browse-link :album what value)
              (browse-link :song what value))))))

(defun browse-link (new-what what value)
  (unless (eql new-what what)
    (html
     "["
     (:a :href (link "browse" :what new-what what value)
         (:format "~(~as~)" new-what))
     "]")))
```

`browse` 页上要做的另一件事是提供一个带有几个隐含 `INPUT` 字段的表单和一个“Add all”提交按钮。你需要使用一个 HTML 表单而不是一个正常的链接来确保应用的无状态性——确保拥有所需的信息来回应一个来自该请求本身的请求。由于浏览页中的结果可能是部分随机的，因此你需要向服务器提交相当多的数据才能重构添加到播放列表的歌曲列表。如果你没有允许浏览页返回随机结果的话，你将不必需要太多的数据——你只需提交一个带有浏览页所使用的条件的请求来添加歌曲。但如果你以这种方式提交一个含有 `random` 参数的条件来添加歌曲的话，那么最终你所添加的歌曲将与用户点击“Add all”按钮时在页面中看到的歌曲属于不同的随机集合。

你将使用的解决方案是发回一个含有足够多信息的表单，其中带有一个隐含的 `INPUT` 元素以允许服务器可以重构匹配浏览页条件的歌曲列表。该信息就是由 `values-for-page` 所返回的值列表以及 `what` 参数的值。这就是你用到 `base64-list` 参数类型的地方；函数 `values->base64` 从 `values-for-page` 所返回的表中将一个指定列的值解出并放在一个列表中，然后再从该列表中生成一个 base 64 编码的字符串嵌入到表单里。

```
(defun values->base-64 (column values-table)
  (flet ((value (r) (column-value r column)))
    (obj->base64 (map-rows #'value values-table))))
```

当该参数以 `values` 查询参数的形式回到一个将 `values` 声明为类型 `base-64-list` 的 URL 函数中时，它将被自动转换回一个列表。正如你将很快看到的，该列表随后可被用来构造出一个

返回正确的歌曲列表的查询。⁵当你正在按 :song 浏览时，你使用来自 :file 列的值，因为它们可以唯一地标识实际的歌曲，而通过歌曲名可能不行。

29.7 播放列表

本节将我们带到了下一个 URL 函数 `playlist`。这是三个页面中最复杂的一个——它负责显示用户播放列表的当前内容，同时提供了操作播放列表的接口。但在大部分繁文缛节都由 `define-url-function` 所处理的背景下，不难看出函数 `playlist` 是如何工作的。下面是其定义的开始部分，只给出了参数列表：

```
(define-url-function playlist
  (request
    (playlist-id string (playlist-id request) :package)
    (action keyword) ; Playlist manipulation action
    (what keyword :file) ; for :add-songs action
    (values base-64-list) ;
    file ; for :add-songs and :delete-songs actions
    genre ; for :delete-songs action
    artist ; "
    album ; "
    (order-by keyword) ; for :sort action
    (shuffle keyword) ; for :shuffle action
    (repeat keyword)) ; for :set-repeat action
```

除了强制出现的 `request` 参数以外，`playlist` 还接受大量的查询参数。某种程度来讲最重要的是 `playlist-id`，它标识了页面应显示和管理的那个 `playlist` 对象。对于这个参数，你可以利用 `define-url-function` 的“粘滞参数”特性。正常情况下 `playlist-id` 无需显式提供，缺省为 `playlist-id` 函数所返回的值，也就是浏览器所在客户机的 IP 地址。不过，通过允许显式地指定该值，用户也可以从不同于运行他们的 MP3 客户端的机器上管理其播放列表。并且如果该参数被指定过一次，那么 `define-url-function` 将通过在浏览器中设置一个 cookie 来使其成为“粘滞的”。后面你将定义一个 URL 函数来生成一个全部已有播放列表的列表，其中用户可以选取一个与他们正在机器上浏览的不同的播放列表。

参数 `action` 指定了一些在用户的播放列表对象上所做的操作。该参数的值——将会自动地转化成一个关键字符——包括：`:add-songs`、`:delete-songs`、`:clear`、`:sort`、`:shuffle` 或 `:set-repeat`。其中 `:add-songs` 操作用于浏览页中的“Add all”按钮，也用于那些用来添加单独歌曲的链接。其他的操作都用于播放列表页面本身的链接。

参数 `file`、`what` 和 `values` 与 `:add-songs` 操作配合使用。通过将 `values` 声明为类型 `base-64-list`，`define-url-function` 底层将负责解码由“Add all”形式所提交的值。其余的参数按照注释里所描述的方式分别用于其他操作。

⁵ 这个解决方案也有其负面效果——如果一个浏览页返回了许多结果，那么大量的数据将在底层来回发送。另外，数据库查询也未必是最有效的。但它确实可以保证应用的无状态性。一个替代的方法是反过来在服务器端保存由 `browse` 所返回的结果，然后当一个添加歌曲的请求进来时，查找适当的一点儿信息以重建正确的歌曲集。例如，你可以只是将这个值列表保存下来，而不必将它放在表单里发回服务器。或者你可以在生成浏览结果之前将 `RANDOM-STATE` 复制下来以便后面可以重建出同样的“随机”结果。但这个思路也有它自己的问题：你永远不知道用户何时可能点击了它们浏览器的回退按钮返回到一个旧的浏览页然后再点击那个“Add all”按钮。总之，欢迎来到丰富多彩的 Web 编程世界。

现在让我们查看 `playlist` 的函数体。你需要做的第一件事是使用 `playlist-id` 来查找一个队列对象并使用下面的两行来获取该播放列表的锁：

```
(let ((playlist (lookup-playlist playlist-id)))
  (with-playlist-locked (playlist))
```

由于 `lookup-playlist` 将在必要时创建一个新的播放列表，因此它将总是返回一个 `playlist` 对象。然后你进行必要的队列处理，派发 `action` 参数的值以便调用一个 `playlist` 系列的函数。

```
(case action
  (:add-songs      (add-songs playlist what (or values (list file))))
  (:delete-songs   (delete-songs
    playlist
    :file file :genre genre
    :artist artist :album album))
  (:clear          (clear-playlist playlist))
  (:sort           (sort-playlist playlist order-by))
  (:shuffle        (shuffle-playlist playlist shuffle))
  (:set-repeat     (setf (repeat playlist) repeat)))
```

函数 `playlist` 中其余的部分就是实际的 HTML 生成了。再一次，你可以使用 `:mp3-browser-page` 这个 HTML 宏来确保页面的基本样式匹配应用程序中的其他页面，尽管这一次你要向 `:header` 参数传递 `NIL` 以避免生成那个 H1 头。下面是该函数的其余部分：

```
(html
  (:mp3-browser-page
    (:title (:format "Playlist - ~a" (id playlist)) :header nil)
    (playlist-toolbar playlist)
    (if (empty-p playlist)
      (html (:p (:i "Empty.")))
      (html
        ((:table :class "playlist")
         (:table-row "#" "Song" "Album" "Artist" "Genre")
         (let ((idx 0)
               (current-idx (current-idx playlist)))
           (do-rows (row (songs-table playlist))
             (with-column-values (track file song album artist genre) row
               (let ((row-style (if (= idx current-idx) "now-playing" "normal")))
                 (html
                   ((:table-row :class row-style)
                    track
                    (:progn song (delete-songs-link :file file))
                    (:progn album (delete-songs-link :album album))
                    (:progn artist (delete-songs-link :artist artist))
                    (:progn genre (delete-songs-link :genre genre)))))))
               (incf idx))))))))))
```

其中的函数 `playlist-toolbar` 生成一个含有到 `playlist` 页面的链接的工具栏以进行多种 `:action` 操作。而 `delete-songs-link` 可以生成一个带有设置为 `:delete-songs` 的 `:action` 参数和其他适当参数的 `playlist` 链接以删除一个单独文件、一个专辑里的所有文件、特定艺术家的文件，或是指定风格的文件。

```
(defun playlist-toolbar (playlist)
  (let ((current-repeat (repeat playlist))
        (current-sort (ordering playlist))
        (current-shuffle (shuffle playlist)))
    (html
```

```

(:p :class "playlist-toolbar"
  (:i "Sort by:")
  " [ "
  (sort-playlist-button "genre" current-sort) " | "
  (sort-playlist-button "artist" current-sort) " | "
  (sort-playlist-button "album" current-sort) " | "
  (sort-playlist-button "song" current-sort) " ] "
  (:i "Shuffle by:")
  " [ "
  (playlist-shuffle-button "none" current-shuffle) " | "
  (playlist-shuffle-button "song" current-shuffle) " | "
  (playlist-shuffle-button "album" current-shuffle) " ] "
  (:i "Repeat:")
  " [ "
  (playlist-repeat-button "none" current-repeat) " | "
  (playlist-repeat-button "song" current-repeat) " | "
  (playlist-repeat-button "all" current-repeat) " ] "
  "[ " (:a :href (link "playlist" :action "clear") "Clear") " ] ")))

(defun playlist-button (action argument new-value current-value)
  (let ((label (string-capitalize new-value)))
    (if (string-equal new-value current-value)
        (html (:b label))
        (html (:a :href (link "playlist" :action action argument new-value)
          label)))))

(defun sort-playlist-button (order-by current-sort)
  (playlist-button :sort :order-by order-by current-sort))

(defun playlist-shuffle-button (shuffle current-shuffle)
  (playlist-button :shuffle :shuffle shuffle current-shuffle))

(defun playlist-repeat-button (repeat current-repeat)
  (playlist-button :set-repeat :repeat repeat current-repeat))

(defun delete-songs-link (what value)
  (html " [ " (:a :href (link "playlist" :action :delete-songs what value) "x")
  " ] "))

```

29.8 查找播放列表

三个 URL 函数中的最后一个是最简单的。它可以表示一个列出了当前已创建的所有播放列表的表。通常用户不需要用到这个页面，但在开发过程中它可以给你一个有用的系统状态视图。它还提供了用来选择一个不同的播放列表的机制——每个播放列表 ID 都是一个带有显式 `playlist-id` 查询参数的 `playlist` 页面的链接，该查询参数随后会被 `playlist` URL 函数设置成粘滞的。注意到你需要获取 `*playlists-lock*` 以确保 `*playlists*` 哈希表在你对其迭代时不发生改变。

```

(define-url-function all-playlists (request)
  (:mp3-browser-page
   (:title "All Playlists")
   ((:table :class "all-playlists")
    (:table-row "Playlist" "# Songs" "Most recent user agent")
    (with-process-lock (*playlists-lock*)
      (loop for playlist being the hash-values of *playlists* do
            (html
             (:table-row

```

```
(:a :href (link "playlist" :playlist-id (id playlist))
  (:print (id playlist)))
(:print (table-size (songs-table playlist)))
(:print (user-agent playlist))))))
```

29.9 运行应用程序

这样就写完了。为了使用这个应用程序，你只需使用来自第 27 章的 `load-database` 函数加载 MP3 数据库，发布那个 CSS 样式表，将 `*song-source-type*` 设置成 `playlist` 以便 `find-song-source` 可以使用播放列表来代替前面章节里定义的单一歌曲源，最后启动 AllegroServe。下面的函数为你完成了所有这些步骤，你只需在两个参数中填入适当的值就可以了：`*mp3-dir*` 指定了你的 MP3 集所在的根目录，而 `*mp3-css*` 则是 CSS 样式表的文件名：

```
(defparameter *mp3-dir* ...)
(defparameter *mp3-css* ...)

(defun start-mp3-browser ()
  (load-database *mp3-dir* *mp3s*)
  (publish-file :path "/mp3-browser.css" :file *mp3-css* :content-type
"text/css")
  (setf *song-source-type* 'playlist)
  (net.aserve::debug-on :notrap)
  (net.aserve:start :port 2001)))
```

当你调用这个函数时，它将在从你的 ID3 文件中加载 ID3 信息时打印出一些点。然后你可以将你的 MP3 客户端指向下面的 URL：

`http://localhost:2001/stream.mp3`

然后再将你的浏览器指向一个好的起始点，比如说：

`http://localhost:2001/browse`

这可以让你从缺省的风格分类开始浏览。在你为播放列表添加了一些歌曲以后，你可以点击 MP3 客户端的播放按钮，然后它就应该开始播放第一首歌了。

很明显，你可以从几方面改进用户接口——例如，如果你的库里有许多 MP3，那么可以通过艺术家或专辑名的第一个字母来浏览就会很有用。或者也许你可以为播放列表页面添加一个“Play whole album”按钮从而立刻把来自同一张专辑的所有歌曲全部放入播放列表的最顶端并作为当前播放的歌曲。或者你还可以改变 `playlist` 类，当没有歌曲在队列中时不再播放静音，而是从数据库中随机选择一首歌曲来播放。但所有这些思路都属于应用设计的范畴，实际上并不是本书的主题。相反，接下来两章将回到软件基础设施的层面上，探索 HTML 生成库 FOO 是如何工作的。

第30章 实践：一个 HTML 生成库，解释器部分

在本章和下一章里，你将审视过去几章里用到的 HTML 生成器 FOO 的底层实现。FOO 是 Common Lisp 中相当普遍但在非 Lisp 语言——换句话说，“面向语言”的编程——里相对罕见的编程类型的一个示例。相比在函数、类和宏的基础上构建出来的 API，FOO 提供了一个可以嵌入到你的 Common Lisp 程序中的领域相关语言。

FOO 提供了用于同样 S-表达式语言的两个语言处理器。一个是解释器，其接受作为数据的一个 FOO “程序”然后解释它来生成 HTML。另一个是编译器，其编译可能嵌入在 Common Lisp 代码中的 FOO 表达式，将它编译成生成 HTML 的 Common Lisp 代码并执行这些嵌入的代码。解释器以函数 `emit-html` 暴露出来，而编译器是作为宏 `html` 提供的，你在前面的章节里用到过它们。

在本章里，你将首先查看一些在解释器和编译器之间共享的基础设施，然后查看解释器的实现。在下一章里，我将向你展示编译器是如何工作的。

30.1 设计一个领域相关语言

设计一个嵌入式语言需要两个步骤：首先，设计那个可以让你表达你想要表达的事物的语言，其次，实现一个或几个处理器，其接受一个以该语言表达的“程序”并要么进行该语言所表达的操作，要么将该程序转译成进行等价行为的 Common Lisp 代码。

因此，第一步是设计 HTML 生成语言。设计一个好的领域相关语言的关键是寻求表达能力和简洁性之间的正确平衡。举个例子，一个用来生成 HTML 的高度可表达性但并不太简洁的“语言”是字面 HTML 字符串的语言。该语言的合法“形式”是那些含有字面 HTML 的字符串。该“语言”的语言处理器可以通过简单地原样输出它们来处理这些形式。

```
(defvar *html-output* *standard-output*)

(defun emit-html (html)
  "An interpreter for the literal HTML language."
  (write-sequence html *html-output*))

(defmacro html (html)
  "A compiler for the literal HTML language."
  `(write-sequence ,html *html-output*))
```

这个“语言”是高度可表达的，因为它可以表达“任何”你可能生成的 HTML。¹另一方面，该语言没有什么简洁性可言，因为它带给你的是零压缩率——它的输入就是输出。

为了设计一个可以给你一些有用的压缩但不会牺牲太多可表达性的语言，你需要识别出输出的细节中那些重复和无关的部分。然后你可以让输出的这些方面隐含在该语言的语义中。

例如，由于 HTML 的结构，每一个开放的标签都有一个配对的闭合标签。²当你手工编写 HTML 时，你不得不书写这些闭合标签，但可以通过隐式生成这些闭合标签从而改进你的 HTML 生成语言的简洁性。

另一种以牺牲少许可表达性为代价提高简洁性的方式是让语言处理器负责在元素之间添加适当的空白——包括空行和缩进。当你程序化地生成 HTML 时，你通常不太关心元素前后的换行，以及是否不同的元素相对于它们的父元素正确缩进了。让语言处理器根据一些规则来插入空白就意味着你不需要担心它们了。看起来 FOO 事实上支持两种模式——一种使用最少量的空白，这允许它生成极其高效和紧凑的 HTML，另一种可以生成良好格式化的 HTML，其中不同的元素根据它们的角色相对其他元素缩进并分离开。

另一个最好可以交给语言处理器来做的细节是对特定字符的转义，诸如“<”、“>”和“&”这些字符在 HTML 中有特殊的含义。显然，如果你只是通过将字符串打印到一个流中来生成 HTML 的话，那么将字符串中出现的任何这类特殊字符替换成对应的转义序列“<”、“>”和“&”的工作将由你来完成。但如果语言处理器可以知道那些字符串将被输出成元素数据的话，那么它就可以帮你自动地转义这些字符。

30.2 FOO 语言

理论已经说得足够多了。下面我将给你一个关于 FOO 所实现的语言的快速概述，然后你将查看两个 FOO 语言处理器的实现——本章是解释器，然后下一章是编译器。

和 Lisp 本身一样，FOO 语言的基本语法由构成 Lisp 对象的形式定义而成。该语言定义了合法的 FOO 形式是如何被转化成 HTML 的。

最简单的 FOO 形式是诸如字符串、数字和关键字符³这样的自求值 Lisp 对象。你将需要一个函数 `self-evaluating-p` 来测试一个给定对象在 FOO 的意义下是否是自求值的。

```
(defun self-evaluating-p (form)
  (and (atom form) (if (symbolp form) (keywordp form) t)))
```

满足该谓词的对象将通过使用 `PRINC-TO-STRING` 把它们转换成字符串然后转义任何诸如“<”、“>”或“&”这类保留字符后输出。当值被作为属性输出时，字符“.”和“:”也需要转

¹ 事实上，它可能是太过于可表达了，因为它还可以生成所有那些甚至不是合法 HTML 的输出。当然，如果你需要生成用来补偿有 bug 的 Web 浏览器的并非严格正确的 HTML 的话，这也可以成为一个特性。另外，语言处理器通常也会接受那些词法严格正确并且形态良好但运行起来以后却不可避免地产生未定义行为的程序。

² 好吧，其实是几乎所有标签。诸如 IMG 和 BR 等特定标签不是这样的。你将在“基本求值规则”一节里处理它们。

³ 在 Common Lisp 标准所描述的严格语言里，关键字符不是自求值的，尽管它们在事实上确实求值到它们自身。参见语言标准的第 3.1.2.1.3 节或 HyperSpec 里的简要讨论。

义。这样，你可以在一个自求值对象上调用 `html` 宏从而将其输出到 `*HTML-OUTPUT*`（初始绑定到 `*STANDARD-OUTPUT*`）。表 30-1 给出了一些不同的自求值对象是如何被输出的。

Table 30-1. 自求值对象的 FOO 输出

FOO 形式	生成的 HTML
<code>(html "foo")</code>	<code>foo</code>
<code>(html 10)</code>	<code>10</code>
<code>(html :foo)</code>	<code>FOO</code>
<code>(html "foo & bar")</code>	<code>foo & bar</code>

当然，多数 HTML 都由带有标签的元素所构成。用来描述每个元素的三部分信息分别是标签、一个属性集合，以及一个含有文本和/或更多 HTML 元素的主体。这样，你需要一种方式将这三部分信息表示成 Lisp 对象，最好是 Lisp 读取器已经知道如何读取的对象。⁴如果你暂时不考虑属性的话，那么在 Lisp 列表和 HTML 元素之间存在一个明显的映射：任何 HTML 元素可被表示成一个列表，其 `FIRST` 部分是一个名字与该元素的标签名相同的符号，而 `REST` 部分是一个代表其他 HTML 元素的由自求值对象或列表所组成的列表。这样：

```
<p>Foo</p> ~DBLARR (:p "Foo")

<p><i>Now</i> is the time</p> ~DBLARR (:p (:i "Now") " is the time")
```

现在唯一的问题是在哪里插入属性。由于多数元素都没有属性，因此如果可以继续使用前面用于无属性元素的语法就最好了。FOO 提供了两种方式来表示带有属性的元素。第一种是简单地将属性包含在列表中紧随符号之后的位置上，其中命名了属性的关键字符串和代表属性值形式的对象交替出现。元素的主体开始于列表中第一个在属性名的位置上却并非关键字符串的那一项。因此：

```
HTML> (html (:p "foo"))
<p>foo</p>
NIL
HTML> (html (:p "foo" (:i "bar") " baz"))
<p>foo <i>bar</i> baz</p>
NIL
HTML> (html (:p :style "foo" "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html (:p :id "x" :style "foo" "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

对于那些喜欢在元素的属性和主体间有更明显界限的人们，FOO 还支持另一种语法：如果一个列表的第一个元素本身是一个以关键字符串为其首元素的列表，那么外层的列表就代表一个以该关键字为标签的 HTML 元素，嵌套列表的 `REST` 部分作为其属性，外层列表的 `REST` 部分作为其

⁴ 使用 Lisp 读取器知道如何读取的对象的要求并不是一个困难而高效的要求。由于 Lisp 读取器本身是可定制的，因此你还可以定义一个新的读取器层面语法来处理新的对象类型。但这样做不值得并且会带来更多麻烦。

主体。这样，你可以像下面这样书写前面两个表达式：

```
HTML> (html ((:p :style "foo") "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html ((:p :id "x" :style "foo") "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

下面的函数测试是否一个给定对象匹配这两种语法：

```
(defun cons-form-p (form &optional (test #'keywordp))
  (and (consp form)
        (or (funcall test (car form))
            (and (consp (car form)) (funcall test (caar form))))))
```

你应当将 `test` 函数参数化，因为以后你将需要在该名字上使用稍微不同的谓词来测试相同的两种语法。

为了完全地抽象掉两种语法变种之间的差异，你可以定义一个函数 `parse-cons-form`，它接受一个形式并将其解析成 3 个元素：标签、属性列表和主体列表，然后以多值的形式返回它们。实际求值点对形式的代码将使用该函数而不必担心它所采用的语法。

```
(defun parse-cons-form (sexp)
  (if (consp (first sexp))
      (parse-explicit-attributes-sexp sexp)
      (parse-implicit-attributes-sexp sexp)))

(defun parse-explicit-attributes-sexp (sexp)
  (destructuring-bind ((tag &rest attributes) &body body) sexp
    (values tag attributes body)))

(defun parse-implicit-attributes-sexp (sexp)
  (loop with tag = (first sexp)
        for rest on (rest sexp) by #'cddr
        while (and (keywordp (first rest)) (second rest))
        when (second rest)
        collect (first rest) into attributes
        collect (second rest) into attributes
        end
        finally (return (values tag attributes rest))))
```

现在你已经基本规范了语言，然后你可以考虑如何实际来实现语言的处理器了。怎样才能将一系列的 FOO 形式转化成想要的 HTML 呢？如同我前面提到的，你将要实现 FOO 的两个语言处理器：一个解释器负责遍历一棵 FOO 形式树并直接输出对应的 HTML，以及一个编译器遍历一棵树并将其转化成可以输出同样 HTML 的 Common Lisp 代码。无论解释器还是编译器都将构建一个共同的代码基础之上，后者提供对诸如转义保留字符和生成美观的缩进输出之类特性的支持，因此我们有理由从这里开始。

30.3 字符转义

你将需要依赖的首要基础设施是知道如何转义那些 HTML 中带有特殊含义字符的代码。有三个这样的字符一定不能出现在一个元素或属性值的文本中；它们是“<”、“>”和“&”。在元素文

本或属性值中，这些字符必须被替换成字符引用项“<”、“>”和“&”。类似地，在属性值中，用来给值定界的引号必须被转义，“‘”变成“'”，以及“”变成“"”。此外，任何字符都可被表示成一个数值的字符引用项，后者依次由“&”、“#”、一个以 10 进制数表示的数值代码，以及一个分号所组成。这些数值转义项有时用来在 HTML 中嵌入非 ASCII 的字符。

包定义

由于 FOO 是一个底层库，因此你所开发的这个包并不依赖很多外部代码——只有通常依赖的来自 COMMON-LISP 包的名字以及几乎同样经常依赖的来自 COM.GIGAMONKEYS.MACRO-UTILITIES 的宏生成宏的名字。另一方面，该包需要导出所有被使用 FOO 的代码所需要的名字。下面是来自可从本书 Web 站点上下载的源代码中的 DEFPACKAGE 定义：

```
(defpackage :com.gigamonkeys.html
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :with-html-output
           :in-html-style
           :define-html-macro
           :html
           :emit-html
           :&attributes))
```

下面的函数接受单个字符并返回一个含有该字符的字符引用项的字符串：

```
(defun escape-char (char)
  (case char
    (#\& "&amp;")
    (#\< "&lt;")
    (#\> "&gt;")
    (#\` "&apos;")
    (#\\" "&quot;")
    (t (format nil "&#~d;" (char-code char)))))
```

你可以使用该函数来作为一个函数 escape 的基础，后者接受一个字符串和一个字符序列，然后返回一个对第一个参数的拷贝，其中所有在第二个参数中出现过的字符都被替换成由 escape-char 所返回的对应的字符项。

```
(defun escape (in to-escape)
  (flet ((needs-escape-p (char) (find char to-escape)))
    (with-output-to-string (out)
      (loop for start = 0 then (1+ pos)
            for pos = (position-if #'needs-escape-p in :start start)
            do (write-sequence in out :start start :end pos)
            when pos do (write-sequence (escape-char (char in pos)) out)
            while pos))))
```

你还可以定义两个参数：*element-escapes*，其含有需要在正常元素数据中转义的所有字符，以及 *attribute-escapes*，其含有在属性值中需要转义的字符集。

```
(defparameter *element-escapes* "<>&")
(defparameter *attribute-escapes* "<>&\\"")
```

下面是一些例子：

```
HTML> (escape "foo & bar" *element-escapes*)
"foo &amp; bar"
HTML> (escape "foo & 'bar'" *element-escapes*)
"foo &amp; 'bar'"
HTML> (escape "foo & 'bar'" *attribute-escapes*)
"foo &amp; &apos;bar&apos;"
```

最后，你将需要一个变量 `*escapes*`，它将被绑定到需要进行转义的字符集上。它被初值设置成 `*element-escapes*` 的值，但是当生成属性时，你将看到它会被重新绑定在 `*attribute-escapes*` 的值上。

```
(defvar *escapes* *element-escapes*)
```

30.4 缩进打印机

为了处理生成精美缩进输出的问题，你可以定义一个类 `indenting-printer`，其包装在一个输出流的外围，以及一些函数在使用该类的一个实例来将字符串输出到该流的同时可以跟踪何时开始新的一行。该类如下所示：

```
(defclass indenting-printer ()
  ((out           :accessor out           :initarg :out)
   (beginning-of-line-p :accessor beginning-of-line-p :initform t)
   (indentation      :accessor indentation      :initform 0)
   (indenting-p       :accessor indenting-p       :initform t)))
```

操作在 `indenting-printers` 上的主函数是 `emit`，其接受该打印机和一个字符串并将该字符串输出到打印机的输出流上，同时跟踪它何时输出一个换行以重置 `beginning-of-line-p` 槽。

```
(defun emit (ip string)
  (loop for start = 0 then (1+ pos)
        for pos = (position #\Newline string :start start)
        do (emit/no-newlines ip string :start start :end pos)
        when pos do (emit-newline ip)
        while pos))
```

为了实际输出该字符串，它用到了函数 `emit/no-newlines`，后者通过助手函数 `indent-if-necessary` 来输出任何需要的缩进，然后再将字符串写入该流。该函数也会被其他用来输出一个确定不需要换行的字符串的代码所直接调用。

```
(defun emit/no-newlines (ip string &key (start 0) end)
  (indent-if-necessary ip)
  (write-sequence string (out ip) :start start :end end)
  (unless (zerop (- (or end (length string)) start))
    (setf (beginning-of-line-p ip) nil)))
```

助手函数 `indent-if-necessary` 通过检测 `beginning-of-line-p` 和 `indenting-p` 来决定是否需要输出缩进，以及当两者均为真时输出由 `indentation` 的值所指定数量的空格。使用 `indenting-printer` 的代码可以通过操作 `indentation` 和 `indenting-p` 槽来控制缩进。递增和递减 `indentation` 可以改变前导空格的数量，而将 `indenting-p` 设置为 `NIL` 可以临时关闭缩进。

```
(defun indent-if-necessary (ip)
  (when (and (beginning-of-line-p ip) (indenting-p ip)))
```

```
(loop repeat (indentation ip) do (write-char #\Space (out ip)))
  (setf (beginning-of-line-p ip) nil)))
```

indenting-printer API 中的最后两个函数是 `emit-newline` 和 `emit-freshline`，两者都用来输出一个换行符，类似于 FORMAT 指令 `~%` 和 `~&`。这就是说，唯一的区别在于 `emit-newline` 总是输出一个换行，而 `emit-freshline` 则只有在 `beginning-of-line-p` 为假时才输出。这样，中间没有任何 `emit` 的多个 `emit-freshline` 将不会导致一个空行。这在一些代码希望生成一些以换行结束的输出，而另一些代码希望生成一些以换行开始的输出，但你不希望两者之间产生空行时非常有用。

```
(defun emit-newline (ip)
  (write-char #\Newline (out ip))
  (setf (beginning-of-line-p ip) t))

(defun emit-freshline (ip)
  (unless (beginning-of-line-p ip) (emit-newline ip)))
```

有了这些先决条件，现在你可以开始进入 FOO 处理器的核心地带了。

30.5 HTML 处理器接口

现在你可以定义将被 FOO 语言处理器用来输出 HTML 的接口了。你可以将该接口定义成一组广义函数，因为你将需要两个实现——一个实际输出 HTML，而另一个可被 `html` 宏用来收集一个需要进行的操作的列表，然后可以优化并编译成以更高效方式生成同样输出的代码。我将把这些广义函数称为后台接口。它由下面 8 个广义函数构成：

```
(defgeneric raw-string (processor string &optional check-for-newlines))

(defgeneric newline (processor))

(defgeneric freshline (processor))

(defgeneric indent (processor))

(defgeneric unindent (processor))

(defgeneric toggle-indenting (processor))

(defgeneric embed-value (processor value))

(defgeneric embed-code (processor code))
```

尽管这些函数中的一些明显地有其对应的 indenting-printer 系列函数，但重要的是理解这些广义函数定义了 FOO 语言处理器所使用的抽象操作，并且它们不总是通过对 indenting-printer 系列函数的调用来实现的。

这就是说，也许理解这些抽象操作语义的最简单方式是去查看特化在 `html-pretty-printer` 类上的方法的具体实现，该类被用来生成人类可读的 HTML。

30.6 美化打印机后台

你可以从定义一个带有两个槽的类开始——一个槽用来保存一个 `indenting-printer` 的实例，另一个槽用来保存制表符宽度——对于 HTML 元素每一层嵌套缩进你想要增加的空格数。

```
(defclass html-pretty-printer ()
  ((printer :accessor printer :initarg :printer)
   (tab-width :accessor tab-width :initarg :tab-width :initform 2)))
```

现在你可以实现特化在 `html-pretty-printer` 上的构成后台接口的 8 个广义函数上的方法了。

F00 处理器使用 `raw-string` 函数来输出不需要字符转义的字符串，这要么是因为你实际想要输出一个正常保留的字符，要么是所有的保留字已经被转义了。通常 `raw-string` 以不含有换行的字符串来调用，因此缺省的行为是使用 `emit/no-newlines`，除非调用者指定了一个非 NIL 的 `newlines-p` 参数。

```
(defmethod raw-string ((pp html-pretty-printer) string &optional newlines-p)
  (if newlines-p
      (emit (printer pp) string)
      (emit/no-newlines (printer pp) string)))
```

函数 `newline`、`freshline`、`indent`、`unindent` 和 `toggle-indenting` 实现了对于底层 `indenting-printer` 的相当直接的管理。唯一的亮点是 HTML 美化打印机只有当动态变量 `*pretty*` 为真时才会生成美化的输出。当它是 NIL 时，你应当生成没有不必要的紧凑 HTML。因此下面的方法，除了 `newline` 之外，全部会在做任何事之前检查 `*pretty*`:⁵

```
(defmethod newline ((pp html-pretty-printer))
  (emit-newline (printer pp)))

(defmethod freshline ((pp html-pretty-printer))
  (when *pretty* (emit-freshline (printer pp))))

(defmethod indent ((pp html-pretty-printer))
  (when *pretty*
    (incf (indentation (printer pp)) (tab-width pp)))))

(defmethod unindent ((pp html-pretty-printer))
  (when *pretty*
    (decf (indentation (printer pp)) (tab-width pp)))))

(defmethod toggle-indenting ((pp html-pretty-printer))
  (when *pretty*
    (with-slots (indenting-p) (printer pp)
      (setf indenting-p (not indenting-p)))))
```

最后，函数 `embed-value` 和 `embed-code` 只被 F00 编译器使用——`embed-value` 用来生成将会输出一个 Common Lisp 表达式值的代码，而 `embed-code` 用来嵌入一点代码来运行并丢弃其结果。

⁵ 另一个更纯粹面向对象的方法将是定义两个类，也许是 `html-pretty-printer` 和 `html-raw-printer`，然后对于那些只有在 `*pretty*` 为真时做事的方法上定义特化在 `html-raw-printer` 上的空操作 (`no-op`) 方法。不过，在本例中，在定义了所有空操作方法以后，你将得到更多的代码，并且随后你还需要确保你在正确的时候创建了正确类的实例。但从一般上讲，使用多态来替换条件语句是一个好的策略。

在解释器中，你无法有意义地求值嵌入的 Lisp 代码，因此这些函数上的相应方法将总是报错。

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (error "Can't embed values when interpreting. Value: ~s" value))

(defmethod embed-code ((pp html-pretty-printer) code)
  (error "Can't embed code when interpreting. Code: ~s" code))
```

使用状况系统来解决你的问题

一个替代的方法将是使用 EVAL 来求值解释器中的 Lisp 表达式。这种方法的问题在于 EVAL 无法访问词法环境。因此，没有办法让类似下面的代码正常工作：

```
(let ((x 10)) (emit-html '(:p x)))
```

其中 x 是一个词法变量。在运行期传递给 emit-html 的符号 x 与同名的词法变量没有特别的关联。Lisp 编译器将代码中对 x 的引用指向该变量，但在代码被编译以后，名字 x 和该变量之间就不再有必要的关联的。这就是当你认为 EVAL 是一个对你问题的解决方案时，你可能犯错误的主要原因。

不过，如果 x 是一个以 DEFVAR 或 DEFPARAMETER 声明的动态变量（从而可能会被命名为 *x* 以代替 x），那么 EVAL 就可以得到它的值。这样，允许 FOO 解释器在某些情况下使用 EVAL 将是有用的。但总是使用 EVAL 显然不是个好主意。所以你可以将 EVAL 跟状况系统一起使用，将两种思想组合在一起从而同时得到两者的优点。

首先定义当 embed-value 和 embed-code 在解释器中调用时你将抛出的一些错误类。

```
(define-condition embedded-lisp-in-interpreter (error)
  ((form :initarg :form :reader form)))

(define-condition value-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed values when interpreting. Value: ~s" (form c)))))

(define-condition code-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed code when interpreting. Code: ~s" (form c)))))
```

现在你可以实现 embed-value 和 embed-code 来抛出这些错误并提供一个将使用 EVAL 来求值形式的再启动。

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (restart-case (error 'value-in-interpreter :form value)
    (evaluate ()
      :report (lambda (s)
        (format s "EVAL ~s in null lexical environment." value))
      (raw-string pp (escape (princ-to-string (eval value)) *escapes* t)))))

(defmethod embed-code ((pp html-pretty-printer) code)
  (restart-case (error 'code-in-interpreter :form code)
    (evaluate ()
      :report (lambda (s)
        (format s "EVAL ~s in null lexical environment." code)))
    (eval code))))
```

现在你可以做类似下面的事情：

```
HTML> (defvar *x* 10)
*x*
HTML> (emit-html '(:p *x*))
```

然后你将以下列信息进入调试器：

```
Can't embed values when interpreting. Value: *x*
[Condition of type VALUE-IN-INTERPRETER]
Restarts:
 0: [EVALUATE] EVAL *x* in null lexical environment.
 1: [ABORT] Abort handling SLIME request.
 2: [ABORT] Abort entirely from this process.
```

如果你调用 evaluate 再启动，那么 embed-value 将 EVAL *x*，得到值 10，然后生成下面的 HTML：

```
<p>10</p>
```

然后，出于方便的考虑你可以在特定的情形下提供再启动函数——可以调用 evaluate 再启动的函数。evaluate 再启动函数无条件地调用同名的再启动，而 eval-dynamic-variables 和 eval-code 仅当其状况中的形式是一个动态变量或潜在的代码时才调用再启动。

```
(defun evaluate (&optional condition)
  (declare (ignore condition))
  (invoke-restart 'evaluate))

(defun eval-dynamic-variables (&optional condition)
  (when (and (symbolp (form condition)) (boundp (form condition)))
    (evaluate)))

(defun eval-code (&optional condition)
  (when (consp (form condition))
    (evaluate)))
```

现在你使用 HANDLER-BIND 来设置一个处理器，自动地为你调用 evaluate 再启动。

```
HTML> (handler-bind ((value-in-interpreter #'evaluate)) (emit-html '(:p *x*)))
<p>10</p>
T
```

最后，你可以定义一个宏来提供绑定两种类型错误的处理器的一个漂亮的语法。

```
(defmacro with-dynamic-evaluation ((&key values code) &body body)
  `(handler-bind (
    ,(if values `((value-in-interpreter #'evaluate)))
    ,(if code `((code-in-interpreter #'evaluate))))
    ,@body))
```

一旦定义了这个宏，你就可以写出下面的代码：

```
HTML> (with-dynamic-evaluation (:values t) (emit-html '(:p *x*)))
<p>10</p>
T
```

30.7 基本求值规则

现在将 FOO 语言与它的处理器接口连接，你所需要的全部就是一个函数，它接受一个对象并处理它，其中调用适当的处理器函数来生成 HTML。例如，当给定类似下面这样的简单形式时：

```
(:p "Foo")
```

该函数可以在处理器上执行下面的调用序列：

```
(freshline processor)
(raw-string processor "<p" nil)
(raw-string processor ">" nil)
(raw-string processor "Foo" nil)
(raw-string processor "</p>" nil)
(freshline processor)
```

目前你可以定义一个简单的函数，它只是检查一个形式是否是合法的 FOO 形式并在是的情况下将其交给 process-sexp-html 来处理。在下一章里，你将为该函数添加一些额外的代码来允许其处理宏和特殊操作符。但目前它看起来像下面这样：

```
(defun process (processor form)
  (if (sexp-html-p form)
      (process-sexp-html processor form)
      (error "Malformed FOO form: ~s" form)))
```

函数 sexp-html-p 检查一个给定对象是否是合法的 FOO 表达式，后者要么是一个自求值形式，要么是一个正确格式的点对。

```
(defun sexp-html-p (form)
  (or (self-evaluating-p form) (cons-form-p form)))
```

自求值形式很容易处理：只需用 PRINC-TO-STRING 将其转化成一个字符串并转义其中出现在变量 *escapes* 中的字符，该变量如同你将回顾的那样被初始绑定到 *element-escapes* 的值上。点对形式则传递给 process-cons-sexp-html 来处理。

```
(defun process-sexp-html (processor form)
  (if (self-evaluating-p form)
      (raw-string processor (escape (princ-to-string form) *escapes*) t)
      (process-cons-sexp-html processor form)))
```

函数 process-cons-sexp-html 随后负责输出开放的标签、任何属性、主体，以及闭合的标签。这里主要的复杂之处在于为了生成美化的 HTML，你需要根据被输出的元素类型来输出空行以及调整缩进。你可以将 HTML 中定义的所有元素分为三类：块、段落和内联元素。块元素——例如 body 和 ul——输出时在它们的开放标签之前和闭合标签之后都要换行，并且它们的内容需要缩进一层。段落元素——例如 p、li 和 blockquote——输出时在开放标签之前和闭合标签之后都要换行。内联元素只是简单地在行内输出。下面 3 个参数列出了每种类型的元素：

```
(defparameter *block-elements*
  '(:body :colgroup :dl :fieldset :form :head :html :map :noscript :object
    :ol :optgroup :pre :script :select :style :table :tbody :tfoot :thead
    :tr :ul))

(defparameter *paragraph-elements*
  '(:area :base :blockquote :br :button :caption :col :dd :div :dt :h1
    :h2 :h3 :h4 :h5 :h6 :hr :input :li :link :meta :option :p :param
    :td :textarea :th :title))

(defparameter *inline-elements*
  '(:a :abbr :acronym :address :b :bdo :big :cite :code :del :dfn :em
    :i :img :ins :kbd :label :legend :q :samp :small :span :strong :sub
    :sup :tt :var))
```

函数 `block-element-p` 和 `paragraph-element-p` 测试一个给定标签是否是对应列表的一个成员。⁶

```
(defun block-element-p (tag) (find tag *block-elements*))  
(defun paragraph-element-p (tag) (find tag *paragraph-elements*))
```

其他两个带有它们自己的谓词的分类是那些总是空的元素，例如 `br` 和 `hr`，以及空白需要保留的三个元素 `pre`、`style` 和 `script`。前者在生成正规 HTML（换句话说，不是 XHTML）时需要特别处理，因为它们没有对应的闭合标签。而在输出那三个内部空白需要保留的标签时，你可以临时关闭缩进，这样美化打印器就不会添加任何不属于元素实际内容的空白。

```
(defparameter *empty-elements*  
  '(:area :base :br :col :hr :img :input :link :meta :param))  
  
(defparameter *preserve-whitespace-elements* '(:pre :script :style))  
  
(defun empty-element-p (tag) (find tag *empty-elements*))  
  
(defun preserve-whitespace-p (tag) (find tag *preserve-whitespace-elements*))
```

当生成 HTML 时你所需要的最后一点儿信息是，你是否在生成 XHTML，因为这将影响到你输出空元素的方式。

```
(defparameter *xhtml* nil)
```

有了全部这些信息，现在你可以开始处理一个点对 FOO 形式了。你使用 `parse-cons-form` 来将列表解析成 3 个部分：标签符号、一个可能为空的属性键值对，以及一个可能为空的主体形式。然后你用助手函数 `emit-open-tag`、`emit-element-body` 和 `emit-close-tag` 来分别输出开放的标签、主体，以及闭合的标签。

```
(defun process-cons-sexp-html (processor form)  
  (when (string= *escapes* *attribute-escapes*)  
    (error "Can't use cons forms in attributes: ~a" form))  
  (multiple-value-bind (tag attributes body) (parse-cons-form form)  
    (emit-open-tag processor tag body attributes)  
    (emit-element-body processor tag body)  
    (emit-close-tag processor tag body)))
```

在 `emit-open-tag` 中你需要在适当的时候调用 `freshline` 然后用 `emit-attributes` 来输出属性。你需要将元素的主体传递给 `emit-open-tag`，这样当它在输出 XHTML 时，它就知道究竟是用 “`/>`” 还是 “`>`” 来结束标签。

```
(defun emit-open-tag (processor tag body-p attributes)  
  (when (or (paragraph-element-p tag) (block-element-p tag))  
    (freshline processor))  
  (raw-string processor (format nil "<(~a~)" tag))  
  (emit-attributes processor attributes)  
  (raw-string processor (if (and *xhtml* (not body-p)) "/>" ">"))))
```

在 `emit-attributes` 中，属性名由于必须是关键字符号因此不会被求值，但你应当调用顶层 `process` 函数来求值那些属性值，同时将 `*escapes*` 绑定到 `*attribute-escapes*`。为了给指

⁶ 你不需要一个用于 `*inline-elements*` 的谓词，因为你只可能测试块和段落元素。我在这里只是出于完备性才包括了该参数。

定布尔属性带来方便，这时属性的值就是属性名本身，如果一个属性的值为 `t`——只是 `t`而非任何其他的真值——那么你将其值替换成该属性的名字。⁷

```
(defun emit-attributes (processor attributes)
  (loop for (k v) on attributes by #'cddr do
    (raw-string processor (format nil " ~(~a~)='" k))
    (let ((*escapes* *attribute-escapes*))
      (process processor (if (eql v t) (string-downcase k) v)))
    (raw-string processor "'")))
```

元素主体的输出过程与属性值的输出类似：你可以在主体上循环调用 `process` 来求值其中的每一个形式。其余的代码用来输出换行并根据元素的类型来调整缩进。

```
(defun emit-element-body (processor tag body)
  (when (block-element-p tag)
    (freshline processor)
    (indent processor))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (dolist (item body) (process processor item))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (when (block-element-p tag)
    (unindent processor)
    (freshline processor)))
```

最后，`emit-close-tag` 如你可能想到的那样，输出闭合的标签（除非不需要闭合标签，例如当主体为空并且你要么在输出 XHTML，要么该元素是特殊的空元素之一）。无论你是否实际输出一个闭合标签，你都需要为块和段落元素输出一个结束的换行。

```
(defun emit-close-tag (processor tag body-p)
  (unless (and (or *xhtml* (empty-element-p tag)) (not body-p))
    (raw-string processor (format nil "</~(~a~)>" tag)))
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor)))
```

函数 `process` 是基本的 FOO 解释器。为了让其更易于使用，你可以定义一个函数 `emit-html`，它调用 `process` 并向其传递一个 `html-pretty-printer` 和一个需要求值的形式。你可以定义并使用一个助手函数 `get-pretty-printer` 来得到美化打印器对象，该函数在 `*html-pretty-printer*` 被绑定的情况下返回该变量的值；否则，它生成 `html-pretty-printer` 的一个新实例，其中使用 `*html-output*` 作为其输出流。

```
(defun emit-html (sexp) (process (get-pretty-printer) sexp))

(defun get-pretty-printer ()
  (or *html-pretty-printer*
      (make-instance
        'html-pretty-printer
        :printer (make-instance 'indenting-printer :out *html-output*))))
```

有了这个函数，你就可以将 HTML 输出到 `*html-output*` 了。与其将变量 `*html-output*` 作为 FOO 公共 API 的一部分暴露出来，你应该定义一个宏 `with-html-output`，让它来为你处理流的绑定。它还可以让你指定是否想要美化的 HTML 输出，默认值是变量 `*pretty*` 的值。

⁷ 尽管 XHTML 要求布尔属性必须被用其名字作为值以表示一个真值，但在 HTML 中简单地包含一个没有值的属性名也是合法的，例如使用 `<option selected>` 而非 `<option selected='selected'>`。所有的 HTML 4.0 兼容浏览器都应当同时理解两种形式，但一些有 bug 的浏览器对于特定属性可能只理解没有值的那种形式。如果你需要为这样的浏览器生成 HTML，那么你将需要修改 `emit-attributes` 来以不同的方式输出那些属性。

```
(defmacro with-html-output ((stream &key (pretty *pretty*)) &body body)
  `(~(let* ((*html-output* ,stream)
            (*pretty* ,pretty))
       ,@body))
```

这样，如果你想要使用 `emit-html` 来生成 HTML 到一个文件里，那么你可以写成下面这样：

```
(with-open-file (out "foo.html" :direction output)
  (with-html-output (out :pretty t)
    (emit-html *some-foo-expression*)))
```

30.8 下一步是什么？

在下一章里，你将看到如何实现一个宏来将 FOO 表达式编译成 Common Lisp，这样你就可以将 HTML 生成代码直接嵌入到你的 Lisp 程序中了。你还会扩展 FOO 语言，通过添加它自己的特殊操作符和宏来使其具有更多的表达能力。

第31章 实践：一个 HTML 生成库，编译器部分

现在你可以查看 FOO 编译器是如何工作的了。一个编译器和一个解释器之间的主要区别在于，一个解释器处理一个程序并直接产生某些行为——对于 FOO 解释器来说就是生成 HTML——但一个编译器处理同样的程序却可以产生以其他语言表示的完成同样行为的代码。在 FOO 中，编译器是一个将 FOO 转译成 Common Lisp 代码的 Common Lisp 宏，因此可以直接嵌入到 Common Lisp 程序中。编译器通常比解释器更有优势，这是因为编译过程是预先完成的，因此可以多花一点儿时间来优化它们生成的代码从而使其更加高效。FOO 编译器能够做到这点是因为它通过将字面文本尽可能的合并在一起从而使用比解释器更少量的写入操作来输出同样的 HTML。当编译器是一个 Common Lisp 宏时，你还可以获得额外的优势：让含有嵌入式 Common Lisp 代码的语言更容易被编译器所理解——编译器只需将这些代码识别出来并直接嵌入到其所生成代码的正确位置上就可以了。FOO 编译器将利用这种能力。

31.1 编译器

编译器的基本架构包括 3 层。首先你将实现一个类 `html-compiler`，其带有一个槽用来保存一个可调整的向量，后者用来集聚那些代表执行 `process` 函数的过程中对后台接口广义函数的调用的那些 `op`。

随后你将实现后台接口中广义函数上的方法，在向量中保存操作序列。每一个 `op` 被表示成一个列表，它由一个命名了该操作的关键字和传递给产生该 `op` 的函数的参数所构成。函数 `sexp->ops` 实现了编译器的第一阶段，通过在一个 FOO 形式列表的每个形式上调用带有 `html-compiler` 实例的 `process` 函数来编译该列表。

这个编译器所保存的 `op` 向量随后被传给一个用来做优化的函数，将相邻的 `raw-string op` 合并成一次性输出组合后字符串的单个 `op`。该优化函数也可以可选地将那些只用于美化打印的 `op` 提取出来，这在多数时候会很重要，因为它允许你得以合并更多的 `raw-string op`。

最后，优化后的 `op` 向量被传递给第 3 个函数 `generate-code`，它返回一个实际输出 HTML 的 Common Lisp 表达式的列表。当 `*pretty*` 为真时，`generate-code` 生成使用特化在 `html-pretty-printer` 上的方法的代码来输出美化的 HTML。而当 `*pretty*` 为 `NIL` 时，它生成直接向流 `*html-output*` 写入的代码。

宏 `html` 实际上会生成一个含有两个展开式的主体，一个是在 `*pretty*` 绑定到 `T` 时生成的，而另一个是在 `*pretty*` 绑定到 `NIL` 时生成的。究竟使用哪个展开式取决于 `*pretty*` 在运行期的值。这样，每个含有对 `html` 宏调用的函数都将同时含有生成美化和紧凑输出的代码。

编译器和解释器之间的另一个明显区别是，编译器可以在它所生成的代码中嵌入 Lisp 形式。为了利用这点，你需要修改 `process` 函数，使其在处理一个并非 FOO 形式的表达式时可以调用 `embed-code` 和 `embed-value` 函数。由于所有的自求值对象都是合法的 FOO 形式，因此不需要传递给 `process-sexp-html` 的形式只有那些不匹配 FOO 点对形式语法的列表和非关键字的符号——唯一的非自求值原子。你可以假设任何非 FOO 的点对都是用来内联运行的代码，而所有的符号都是其值打算嵌入的变量。

```
(defun process (processor form)
  (cond
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form)      (embed-code processor form))
    (t                  (embed-value processor form))))
```

现在让我们查看编译器的代码。首先你应当创建两个函数，它们略微地抽象了将在编译的前两阶段用来保存 `op` 的向量。

```
(defun make-op-buffer () (make-array 10 :adjustable t :fill-pointer 0))

(defun push-op (op ops-buffer) (vector-push-extend op ops-buffer))
```

接下来你可以定义 `html-compiler` 类和特化在其上的方法，以实现后台接口。

```
(defclass html-compiler ()
  ((ops :accessor ops :initform (make-op-buffer)))

(defmethod raw-string ((compiler html-compiler) string &optional newlines-p)
  (push-op `(:raw-string ,string ,newlines-p) (ops compiler)))

(defmethod newline ((compiler html-compiler))
  (push-op `(:newline) (ops compiler)))

(defmethod freshline ((compiler html-compiler))
  (push-op `(:freshline) (ops compiler)))

(defmethod indent ((compiler html-compiler))
  (push-op `(:indent) (ops compiler)))

(defmethod unindent ((compiler html-compiler))
  (push-op `(:unindent) (ops compiler)))

(defmethod toggle-indenting ((compiler html-compiler))
  (push-op `(:toggle-indenting) (ops compiler)))

(defmethod embed-value ((compiler html-compiler) value)
  (push-op `(:embed-value ,value ,*escapes*) (ops compiler)))

(defmethod embed-code ((compiler html-compiler) code)
  (push-op `(:embed-code ,code) (ops compiler)))
```

有了这些方法，你就可以实现编译器的第一阶段 `sexp->ops` 了。

```
(defun sexp->ops (body)
  (loop with compiler = (make-instance 'html-compiler)
        for form in body do (process compiler form))
```

```
finally (return (ops compiler))))
```

在目前的阶段里你不需要担心 *pretty* 的值：只是记录下被 process 调用的所有函数即可。下面是 `sexp->ops` 从一个简单的 FOO 形式中产生的结果：

```
HTML> (sexp->ops '(:p "Foo"))
#((:FRESHLINE) (:RAW-STRING "<p" NIL) (:RAW-STRING ">" NIL)
  (:RAW-STRING "Foo" T) (:RAW-STRING "</p>" NIL) (:FRESHLINE))
```

下一个阶段 `optimize-static-output` 接受一个 op 的向量并返回一个含有优化后版本的新向量。算法是简单的——对于每个 :raw-string op，它将字符串写入到一个临时字符串缓冲区里。这样，相邻的 :raw-string op 将构造出含有所需输出字符串的拼接后的单个字符串。无论何时你遇到了一个不是 :raw-string 的 op，你都将当前位置构造的字符串通过助手函数 `compile-buffer` 转化成一个交替出现 :raw-string 和 :newline 两个 op 的序列，然后再添加下一个 op。这个函数也负责在 *pretty* 为 NIL 时清除美化打印的 op。

```
(defun optimize-static-output (ops)
  (let ((new-ops (make-op-buffer)))
    (with-output-to-string (buf)
      (flet ((add-op (op)
               (compile-buffer buf new-ops)
               (push-op op new-ops)))
        (loop for op across ops do
              (ecase (first op)
                (:raw-string (write-sequence (second op) buf))
                (:newline :embed-value :embed-code) (add-op op))
                (:indent :unindent :freshline :toggle-indenting)
                (when *pretty* (add-op op))))
          (compile-buffer buf new-ops)))
      new-ops)))
      
(defun compile-buffer (buf ops)
  (loop with str = (get-output-stream-string buf)
        for start = 0 then (1+ pos)
        for pos = (position #\Newline str :start start)
        when (< start (length str))
        do (push-op `(:raw-string ,(subseq str start pos) nil) ops)
        when pos do (push-op '(:newline) ops)
        while pos))
```

最后一步时将这些 op 转化成相应的 Common Lisp 代码。这一阶段也会关注 *pretty* 的值。当 *pretty* 为真时，它生成在 *html-pretty-printer* 上调用后台广义函数的代码，该变量绑定在 `html-pretty-printer` 上。当 *pretty* 为 NIL 时，它生成的代码将直接向 *html-output* 写入，后者就是美化打印器用来发送输出的那个流。

实际的函数 `generate-code` 很简单。

```
(defun generate-code (ops)
  (loop for op across ops collect (apply #'op->code op)))
```

所有的工作都是由 `op->code` 广义函数上的方法来完成的，它们全部使用 op 名的 EQL 特化符特化在了参数 op 上。

```
(defgeneric op->code (op &rest operands))
(defmethod op->code ((op (eql :raw-string)) &rest operands)
  (destructuring-bind (string check-for-newlines) operands
```

```
(if *pretty*
  `(~(raw-string *html-pretty-printer* ,string ,check-for-newlines)
    `(~(write-sequence ,string *html-output*))))

(defmethod op->code ((op (eql :newline)) &rest operands)
  (if *pretty*
    `(~(newline *html-pretty-printer*)
      `(~(write-char #\Newline *html-output*))))

(defmethod op->code ((op (eql :freshline)) &rest operands)
  (if *pretty*
    `(~(freshline *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :indent)) &rest operands)
  (if *pretty*
    `(~(indent *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :unindent)) &rest operands)
  (if *pretty*
    `(~(unindent *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))

(defmethod op->code ((op (eql :toggle-indenting)) &rest operands)
  (if *pretty*
    `(~(toggle-indenting *html-pretty-printer*)
      (error "Bad op when not pretty-printing: ~a" op)))
```

其中的两个最有趣的 `op->code` 方法是用来为 `:embed-value` 和 `:embed-code` 这两个 `op` 生成代码的方法。在 `:embed-value` 方法中，你可以根据 `escapes` 操作数的值来生成稍有不同的代码，因为当 `escapes` 为 `NIL` 时你不需要生成一个对 `escape` 的调用。而当 `*pretty*` 和 `escapes` 均为 `NIL` 时，你可以生成使用 `PRINC` 来直接向流中输出值的代码。

```
(defmethod op->code ((op (eql :embed-value)) &rest operands)
  (destructuring-bind (value escapes) operands
    (if *pretty*
        (if escapes
            `(~(raw-string
                  *html-pretty-printer* (escape (princ-to-string ,value) ,escapes) t)
              `(~(raw-string *html-pretty-printer* (princ-to-string ,value) t))
            (if escapes
                `(~(write-sequence (escape (princ-to-string ,value) ,escapes)
                  *html-output*)
                    `(~(princ ,value *html-output*)))))
        `(~(princ ,value *html-output*))))
```

这样，类似下面的代码将可以正常工作：

```
HTML> (let ((x 10)) (html (:p x)))
<p>10</p>
NIL
```

因为 `html` 将 `(:p x)` 转译成了类似下面的形式：

```
(progn
  (~(write-sequence "<p>" *html-output*)
   (~(write-sequence (escape (princ-to-string x) "<>&") *html-output*)
    (~(write-sequence "</p>" *html-output*))))
```

当上述代码在 `LET` 上下文中代替了对 `html` 的调用时，你可以得到下面的形式：

```
(let ((x 10))
  (progn
    (write-sequence "<p>" *html-output*)
    (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
    (write-sequence "</p>" *html-output*)))
```

并且在生成的代码中对 `x` 的引用将变成一个对来自 `html` 形式外围 `LET` 的词法变量的引用。

另一方面，`:embed-code` 方法有趣是因为它是如此的简单。因为 `process` 传递形式到 `embed-code`，后者又将其插入到 `:embed-code op` 中，因此你所要做的全部只是再把它拉出来并返回。

```
(defmethod op->code ((op (eql :embed-code)) &rest operands)
  (first operands))
```

这允许类似下面的代码得以工作：

```
HTML> (html (:ul (dolist (x '(foo bar baz)) (html (:li x)))))  
<ul>  
  <li>FOO</li>  
  <li>BAR</li>  
  <li>BAZ</li>  
</ul>  
NIL
```

其中外层的 `html` 调用可以展开成类似下面的形式：

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz)) (html (:li x)))
  (write-sequence "</ul>" *html-output*)))
```

然后如果你在 `DOLIST` 的主体中展开对 `html` 的调用，那么你将得到类似下面的东西：

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz))
    (progn
      (write-sequence "<li>" *html-output*)
      (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
      (write-sequence "</li>" *html-output*)))
    (write-sequence "</ul>" *html-output*)))
```

事实上这些代码将会生成你所见过的输出。

31.2 FOO 特殊操作符

你可以就此打住；FOO 语言的表达性已经足够用来生成你所关心的几乎任何 HTML 了。尽管如此，你还可以为该语言添加两个特性，只需要再写一点儿代码就可以让其更加强大：特殊操作符和宏。

FOO 中的特殊操作符类似于 Common Lisp 中的特殊操作符。特殊操作符提供了在语言中表达那些无法通过语言的基本求值规则来表达的事物的方式。或者，从另一个角度来看，特殊操作符

提供了访问语言求值器所使用的底层机制的能力。¹

举一个简单的例子，在 FOO 编译器中，语言求值器使用 `embed-code` 函数来生成那些将把变量的值嵌入到输出的 HTML 中的代码。不过，由于传递给 `embed-value` 的只是符号，因此在目前所描述的语言里没有办法嵌入一个任意 Common Lisp 表达式的值；`process` 函数将点对单元传给了 `embed-code` 而非 `embed-value`，因此返回的值被忽略了。通常这就是你所需要的东西，因为在 FOO 程序中嵌入 Lisp 代码的主要原因是使用 Lisp 的控制构造。不过，有时你也希望在生成的 HTML 中嵌入计算后的值。例如，你可能希望下面的 FOO 程序可以生成一个含有随机数的段落标签：

```
(:p (random 10))
```

但这不会正常工作，因为代码被运行，然后值被丢掉了。

```
HTML> (html (:p (random 10)))
<p></p>
NIL
```

在目前你所实现的语言中，你可以通过在 `html` 的外面计算该值然后再通过一个变量嵌入它，从而绕过这个限制。

```
HTML> (let ((x (random 10))) (html (:p x)))
<p>1</p>
NIL
```

但这样做很麻烦，尤其是当你考虑把 `(random 10)` 传递给 `embed-value` 而非 `embed-code` 时，最初的形式刚好可以完成你想要做的事。因此，你可以定义一个特殊操作符 `:print`，它被 FOO 语言处理器按照一个和正常 FOO 表达式不同的规则来处理。确切地说，与其输出一个 `<print>` 元素，它实际上会将其主体中的形式传给 `embed-value`。这样，你就可以像下面这样生成一个含有一个随机数的段落：

```
HTML> (html (:p (:print (random 10))))
<p>9</p>
NIL
```

很明显，这个特殊操作符只在编译的 FOO 代码中才有用，因为 `embed-value` 不能工作在解释器中。另一个可以同时用在解释和编译的 FOO 代码中的特殊操作符是 `:format`，它让你使用 `FORMAT` 函数来产生输出。特殊操作符 `:format` 的参数是一个用作格式控制字符串的字符串和任何需要插入的参数。当所有 `:format` 的参数都是自求值对象时，一个字符串通过将它们传递给 `FORMAT` 而被生成出来，并且随后该字符串像其他任何字符串那样被输出。这允许 `:format` 形式被用在传递给 `emit-html` 的 FOO 中。在编译的 FOO 中，`:format` 的参数可以是任意 Lisp 表达式。

其他特殊操作符提供了对哪些字符被自动转义以及显式输出换行的控制：`:noescape` 特殊操作符导致其主体中的所有形式作为正常 FOO 形式来求值，除了 `*escapes*` 绑定到 `NIL`，而 `:attribute` 可以在 `*escapes*` 绑定为 `*attribute-escapes*` 的情况下求值其主体中的形式。而 `:newline` 则被转译成输出一个显式换行的代码。

那么，你怎样才能定义特殊操作符呢？对特殊操作符的处理有两个方面：语言处理器如何识

¹ FOO 特殊操作符和宏之间的相似之处，我将在下一节里讨论到，与 Lisp 本身的情况是一样的。事实上，对 FOO 特殊操作符和宏的工作方式的理解也有助于让你理解 Common Lisp 的有关设计理念。

别那些使用了特殊操作符的形式，以及在处理每个特殊操作符的时候如何得知需要运行的代码？

你可以修改 `process-sexp-html` 来识别每一个特殊操作符并用适当的方法来处理它们——特殊操作符在逻辑上是语言实现的一部分，并且它们不会有太多。不过，如果有更加模块化的方法来添加新的特殊操作符就更好了——这并不是为了方便 FOO 的用户而只是为了方便你自己。

我们将“特殊形式”定义为任何其 `CAR` 是一个特殊操作符名字的符号的列表。你可以通过将一个非 `NIL` 值添加到该符号属性表中关键字 `html-special-operator` 下以标记特殊操作符的名字。因此，你可以像下面这样定义一个函数来测试一个给定形式是否为特殊形式：

```
(defun special-form-p (form)
  (and (consp form) (symbolp (car form)) (get (car form) 'html-special-operator)))
```

实现每个特殊操作符的代码负责按照它们需要的方式提取该列表的其余部分并按照特殊操作符所要求的语义来做事。假设你将定义一个函数 `process-special-form`，它将接受语言处理器和一个特殊形式并运行适当的代码来生成处理器对象上的一个调用序列，那么你可以修订顶层的 `process` 函数像下面这样来处理特殊形式：

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((sexp-html-p form)   (process-sexp-html processor form))
    ((consp form)         (embed-code processor form))
    (t                   (embed-value processor form))))
```

你必须将 `special-form-p` 子句放在最前面，因为特殊形式可能看起来在词法上跟正常的 FOO 表达式一样，就好像 Common Lisp 的特殊形式也可能看起来像正常函数调用一样。

现在你只需实现 `process-special-form` 就好了。与其定义实现了所有特殊操作符的单个函数，你应当转而定义一个宏，它允许你像正规函数一样地定义特殊操作符并负责在特殊操作符的名字的属性表中添加 `html-special-operator` 项。事实上，你保存在属性表中的值可以是一个实现该特殊操作符的函数。下面就是这个宏：

```
(defmacro define-html-special-operator (name (processor &rest other-parameters)
                                         &body body)
  `(eval-when (:compile-toplevel :load-toplevel :execute)
    (setf (get ',name 'html-special-operator)
          (lambda (,processor ,@other-parameters) ,@body))))
```

这是一个相当高级的宏类型，但如果你逐行地观察它就会发现其实也没有特别的。为了看到它的工作方式，我们取该宏的一个简单用例，特殊操作符 `:noescape` 的定义，然后查看其宏展开式。如果你写出了下面的定义：

```
(define-html-special-operator :noescape (processor &rest body)
  (let ((*escapes* nil))
    (loop for exp in body do (process processor exp))))
```

那么它将相当于下面的写法：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ':noescape 'html-special-operator)
        (lambda (processor &rest body)
          (let ((*escapes* nil))
            (loop for exp in body do (process processor exp))))))
```

如同我在第 20 章里所讨论的，`EVAL-WHEN`特殊操作符确保了当你用 `COMPILE-FILE` 编译时，其主体中的代码产生的效果在编译期可见。如果你想要在一个文件中使用 `define-html-special-operator` 然后在同样的文件中使用刚刚定义的特殊操作符的话，那么这个 `EVAL-WHEN` 将是必要的。

随后的 `SETF` 表达式将符号 `:noescape` 的属性 `html-special-operator` 设置成一个匿名函数，其参数列表与 `define-html-special-operator` 中指定的参数列表相同。通过将 `define-html-special-operator` 定义成参数列表分成两部分的形式，`processor` 和其余的部分，你可以确保所有的特殊操作符都至少接受一个参数。

该匿名函数的主体就是提供给 `define-html-special-operator` 的主体。该匿名函数的任务是在通过在后台接口上做适当的调用生成正确的 HTML 或可以生成它们的代码，从而实现该特殊操作符。它还可以使用 `process` 来求值一个作为 FOO 形式的表达式。

特殊操作符 `:noescape` 尤其简单——它所做的全部就是在 `*escapes*` 绑定到 `NIL` 的情况下将其实体中的形式传递给 `process`。换句话说，这个特殊操作符禁止了由 `process-sexp-html` 所进行的正常字符转义。

有了以这种方式定义的特殊操作符，`process-special-form` 所要做的就是查询特殊操作符名字的属性表中的匿名函数并将其应用在处理器和形式的其余部分上。

```
(defun process-special-form (processor form)
  (apply (get (car form) 'html-special-operator) processor (rest form)))
```

现在你可以开始定义其余的 5 个 FOO 特殊操作符了。与 `:noescape` 类似的是 `:attribute`，它在 `*escapes*` 绑定到 `*attribute-escapes*` 的情况下求值其实体中的形式。这个特殊操作符在你想要编写用来输出属性值的助手函数时将会很有用。如果你编写一个类似下面的函数：

```
(defun foo-value (something)
  (html (:print (frob something))))
```

其中的 `html` 宏将用来生成在 `*element-escapes*` 中转义字符的代码。但如果你正在计划像下面这样来使用 `foo-value`：

```
(html (:p :style (foo-value 42) "Foo"))
```

那么你希望它可以生成使用 `*attribute-escapes*` 的代码。因此，你可以代替地像下面这样来编写该函数：²

```
(defun foo-value (something)
  (html (:attribute (:print (frob something)))))
```

`:attribute` 的定义如下所示：

```
(define-html-special-operator :attribute (processor &rest body)
  (let ((*escapes* *attribute-escapes*))
    (loop for exp in body do (process processor exp))))
```

下面两个特殊操作符 `:print` 和 `:format` 被用来输出值。如同早先所讨论的，特殊操作

² `:noescape` 和 `:attribute` 特殊操作符必须被定义成特殊操作符，这是因为 FOO 是在编译期决定如何转义的，而不是运行期。这允许 FOO 在编译期转义字面值，这比在运行期扫描所有输出更加高效得多。

符 :print 差不多等价于先用 (format nil ...) 生成一个字符串然后再嵌入它。将 :format 定义为特殊操作符的主要原因是出于方便考虑。下面这个：

```
(:format "Foo: ~d" x)
```

比下面这个更好看一些：

```
(:print (format nil "Foo: ~d" x))
```

它还有另外的一点优势，如果你将 :format 与全部是自求值的参数一起使用，那么 FOO 可以在编译期求值 :format 而无需等到运行期再做。:print 和 :format 的定义如下所示：

```
(define-html-special-operator :print (processor form)
  (cond
    ((self-evaluating-p form)
     (warn "Redundant :print of self-evaluating form ~s" form)
     (process-sexp-html processor form))
    (t
     (embed-value processor form)))

(define-html-special-operator :format (processor &rest args)
  (if (every #'self-evaluating-p args)
      (process-sexp-html processor (apply #'format nil args))
      (embed-value processor `(format nil ,@args))))
```

特殊操作符 :newline 强制输出一个字面换行，这有时是有用的。

```
(define-html-special-operator :newline (processor)
  (newline processor))
```

最后，特殊操作符 :progn 和 Common Lisp 中的 PROGN 特殊操作符相似。它简单地按顺序处理其主体中的形式。

```
(define-html-special-operator :progn (processor &rest body)
  (loop for exp in body do (process processor exp)))
```

换句话说，下面的形式：

```
(html (:p (:progn "Foo" (:i "bar") "baz")))
```

将生成与下面形式相同的代码：

```
(html (:p "Foo" (:i "bar") "baz"))
```

这可能看起来是个奇怪的需要，因为正常的 FOO 表达式可以在其主体中有任意多个形式。不过，这个特殊操作符将在一种情形里变得非常有用——当编写 FOO 宏的时候，这将把你带到你需要实现的最后一个语言特性上。

31.3 FOO 宏

FOO 宏类似于 Common Lisp 宏。一个 FOO 宏是一点儿代码，它接受一个 FOO 表达式作为参数并返回一个新的 FOO 表达式作为结果，后者随后按照正常 FOO 求值规则来求值。实际的实现与特殊操作符的实现非常相似。

和特殊操作符一样，你可以定义一个谓词来测试是否一个给定形式是一个宏形式。

```
(defun macro-form-p (form)
  (cons-form-p form #'(lambda (x) (and (symbolp x) (get x 'html-macro)))))
```

你使用前面定义的函数 `cons-form-p`, 因为你想要允许宏被用在所有非宏的 FOO 点对形式语法中。不过, 你需要传递一个不同的谓词函数, 它用来测试形式名是否是一个带有非 `NIL` 的 `html-macro` 属性的符号。另外, 和特殊操作符的实现一样, 你将定义一个宏用来定义 FOO 宏, 它负责将一个函数保存在宏名的属性表中, 位于关键字 `html-macro` 之下。不过, 定义一个宏的过程更加复杂一些, 因为 FOO 支持两种类型的宏。一些你将定义的宏将类似于正常的 HTML 元素并且可能想要容易地访问一个属性列表。其他的宏将简单地想要对它们主体元素的直接访问。

你可以将这两种类型的宏的区别隐式地决定: 当你定义一个 FOO 宏时, 参数列表可以包含一个 `&attributes` 参数。如果有这个参数的话, 那么宏形式将按照正常点对形式来解析, 并且宏函数将被传递两个值, 一个属性的 plist 和一个构成形式体的表达式列表。一个没有 `&attributes` 参数的宏形式将不会解析属性, 并且宏函数将被应用在单一参数上: 一个含有主体表达式的列表。前者对于本质上的 HTML 模板很有用。例如:

```
(define-html-macro :mytag (&attributes attrs &body body)
  `((:div :class "mytag" ,@attrs) ,@body))

HTML> (html (:mytag "Foo"))
<div class='mytag'>Foo</div>
NIL
HTML> (html (:mytag :id "bar" "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
HTML> (html ((:mytag :id "bar") "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
```

后一种类型的宏对于编写管理其主体中形式的宏更加有用。这个类型的宏可以作为一种 HTML 控制构造类型来使用。作为一个简单的例子, 考虑下面这个实现了 `:if` 构造的宏:

```
(define-html-macro :if (test then else)
  `(if ,test (html ,then) (html ,else)))
```

该宏允许你写出下面的代码:

```
(:p (:if (zerop (random 2)) "Heads" "Tails"))
```

而不必写成下面这个更加冗长的版本:

```
(:p (if (zerop (random 2)) (html "Heads") (html "Tails"))))
```

为了决定你需要生成哪种类型的宏, 你需要一个函数来解析 `define-html-macro` 的参数列表。该函数返回两个值, `&attributes` 参数的名字或者不存在时为 `NIL`, 以及一个含有 `args` 中去掉 `&attributes` 标记及其后续列表元素后剩下的所有元素的列表。³

```
(defun parse-html-macro-lambda-list (args)
  (let ((attr-cons (member '&attributes args)))
    (values
      (cadr attr-cons)
      (nconc (ldiff args attr-cons) (cddr attr-cons)))))
```

³ 注意到 `&attributes` 只不过是另一个符号罢了; 以“`&`”开头的名字本质上没有什么特别之处。

```
HTML> (parse-html-macro-lambda-list '(a b c))
NIL
(A B C)
HTML> (parse-html-macro-lambda-list '(&attributes attrs a b c))
ATTRS
(A B C)
HTML> (parse-html-macro-lambda-list '(a b c &attributes attrs))
ATTRS
(A B C)
```

参数列表中 `&attributes` 后面跟着的元素也可以是一个解构参数列表。

```
HTML> (parse-html-macro-lambda-list '(&attributes (&key x y) a b c))
(&KEY X Y)
(A B C)
```

现在你可以开始定义 `define-html-macro` 了。取决于是否指定了 `&attributes` 参数，你需要 HTML 宏的一种或另一种形式，因此主宏简单地检测其正在定义哪种类型的 HTML 宏并随后调用一个助手函数来生成正确类型的代码。

```
(defmacro define-html-macro (name (&rest args) &body body)
  (multiple-value-bind (attribute-var args)
    (parse-html-macro-lambda-list args)
    (if attribute-var
        (generate-macro-with-attributes name attribute-var args body)
        (generate-macro-no-attributes name args body))))
```

实际生成展开式的函数如下所示：

```
(defun generate-macro-with-attributes (name attribute-args args body)
  (with-gensyms (attributes form-body)
    (if (symbolp attribute-args) (setf attribute-args `(&rest ,attribute-args)))
      `(eval-when (:compile-toplevel :load-toplevel :execute)
        (setf (get ',name 'html-macro-wants-attributes) t)
        (setf (get ',name 'html-macro)
              (lambda (,attributes ,form-body)
                (destructuring-bind (,@attribute-args) ,attributes
                  (destructuring-bind (,@args) ,form-body
                    ,@body)))))))

(defun generate-macro-no-attributes (name args body)
  (with-gensyms (form-body)
    `(eval-when (:compile-toplevel :load-toplevel :execute)
      (setf (get ',name 'html-macro-wants-attributes) nil)
      (setf (get ',name 'html-macro)
            (lambda (,form-body)
              (destructuring-bind (,@args) ,form-body ,@body))))))
```

你将定义的宏函数接受一个或两个参数，然后使用 `DESTRUCTURING-BIND` 来将参数提取出来并绑定到在对 `define-html-macro` 的调用中所定义的参数上。在两个展开式中你都需要将宏函数保存在其名字的属性表中 `html-macro` 之下，并且在属性 `html-macro-wants-attributes` 下保存一个布尔值以指示是否该宏接受一个 `&attributes` 参数。你使用在下面的函数 `expand-macro-form` 中使用该属性来决定宏函数被调用的方式：

```
(defun expand-macro-form (form)
  (if (or (consp (first form))
          (get (first form) 'html-macro-wants-attributes))
      (multiple-value-bind (tag attributes body) (parse-cons-form form)
        (funcall (get tag 'html-macro) attributes body)))
```

```
(destructuring-bind (tag &body body) form
  (funcall (get tag 'html-macro) body))))
```

最后一步是通过在顶层 process 函数的派发 COND 语句里添加一个子句来集成对宏的支持。

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((macro-form-p form)   (process processor (expand-macro-form form)))
    ((sexp-html-p form)    (process-sexp-html processor form))
    ((consp form)          (embed-code processor form))
    (t                     (embed-value processor form))))
```

这就是 process 的最终版本。

31.4 公共 API

现在，你最终完成了对 html 宏的实现，它就是 FOO 编译器的主入口点。FOO 公共 API 的其余部分还包括我在上一章里讨论过的 emit-html 和 with-html-output，以及上一节里讨论过的 define-html-macro。define-html-maco 之所以需要成为公共 API 的一部分是因为 FOO 的用户也将希望编写它们自己的 HTML 宏。另一方面，define-html-special-operator 不是公共 API 一部分的理由是它需要太多的关于 FOO 内部的知识来定义一个新的特殊操作符。并且应该几乎没有功能是无法使用已有的语言和特殊操作符来完成的。⁴

公共 API 的最后一个元素——在我到达 html 之前——是另一个宏 in-html-style。该宏通过设置 *xhtml* 变量来控制 FOO 生成 XHTML 还是正常的 HTML。将其定义为宏的原因是你将希望把设置 *xhtml* 的代码包装在一个 EVAL-WHEN 中，这样你就可以在一个文件里设置它并让其影响同一个文件中的所有后续对 html 宏的使用。

```
(defmacro in-html-style (syntax)
  (eval-when (:compile-toplevel :load-toplevel :execute)
    (case syntax
      (:html (setf *xhtml* nil))
      (:xhtml (setf *xhtml* t)))))
```

最后让我们来查看 html 宏本身。实现 html 的唯一难点是必须生成那种可同时生成美观和紧凑输出的代码，具体取决于变量 *pretty* 在运行期的值。这样，html 需要生成一个含有 IF 表达式和两个版本代码的展开式，一个在 *pretty* 绑定为真时编译，另一个在它绑定为 NIL 时编译。更复杂的是，一个 html 调用经常会含有嵌入的 html 调用，就像这样：

```
(html (:ul (dolist (item stuff)) (html (:li item))))
```

如果外层的 html 展开成一个带有两个版本代码的 IF 表达式，一个用在 *pretty* 为真时，另一个用在其为假时，那么如果内嵌的 html 形式也展开成两个版本的话就太傻了。事实上，这将导致指数爆炸，因为内嵌的 html 也打算展开两次——一次是在 *pretty* 为真的分支里，另一次在 *pretty* 为假的分支里。如果每个展开式都生成两个版本，那么你将总共有 4 个版本。而

⁴ 在底层的语言处理基础设施里，目前为止还没有充分地通过特殊操作符暴露出来的一个元素，是对缩进的处理。如果你想要令 FOO 更加灵活——尽管这要以增加其 API 的复杂度为代价——那么你可以添加用于管理底层缩进打印器的特殊操作符。不过看起来解释额外的特殊操作符的代价将远远超出在语言表达性上获得的微小提升。

如果这个内嵌的 `html` 形式还含有另一个内嵌的 `html` 形式，那么你最后将得到该代码的 8 个版本。如果编译器足够聪明，它将最终意识到其生成的多数代码都是没用的并将清除它们，但即便找出这点也需要相当多的时间，从而减慢了任何使用嵌套 `html` 调用的函数的编译时间。

幸运的是，你可以通过生成一个用 `MACROLET` 局部重定义 `html` 宏的展开式，让其只生成正确类型的代码，从而避免无用代码的爆炸。首先你定义一个助手函数，其接受由 `sexp->ops` 返回的 `op` 向量并在 `*pretty*` 绑定为指定值的情况下对其应用 `optimize-static-output` 和 `generate-code`——受 `*pretty*` 影响的两个阶段，然后再将得到的结果插入到一个 `PROGN` 中。`(PROGN` 返回 `NIL` 是为了让输出更简洁。)

```
(defun codegen-html (ops pretty)
  (let ((*pretty* pretty))
    `(progn ,@(generate-code (optimize-static-output ops) nil))))
```

有了这个函数，你随后就可以像下面这样来定义 `html`:

```
(defmacro html (&whole whole &body body)
  (declare (ignore body))
  `(*pretty*
    (macrolet ((html (&body body) (codegen-html (sexp->ops body) t)))
      (let ((*html-pretty-printer* (get-pretty-printer))) ,whole))
    (macrolet ((html (&body body) (codegen-html (sexp->ops body) nil)))
      ,whole)))
```

其中的 `&whole` 参数代表最初的 `html` 形式，而由于它被插入到两个 `MACROLET` 主体的展开式中，因此它将使用每个 `html` 的新定义重新处理，一个生成美化打印的代码，而另一个生成非美化打印的代码。注意到变量 `*pretty*` 被同时用于宏展开期和结果代码运行期。在宏展开期，它被 `codegen-html` 用来使 `generate-code` 生成一种或另一种类型的代码。而在运行期，它被顶层 `html` 宏所生成的 `IF` 表达式用来决定实际上应该运行美化打印还是非美化打印的代码。

31.5 结束语

和往常一样，你可以继续工作在这些代码上，以不同的方式来增强它。一个有趣的方向是使用底层的输出框架来产生其他类型的输出。在你可从本书 Web 站点上下载的 FOO 版本中，你将找到一些实现了 CSS 输出的代码，它们可在解释器和编译器中与 HTML 输出集成在一起。这是一个有趣的案例，因为 CSS 的语法不能像 HTML 那样简单地映射到 S-表达式上。不过，如果你实际查看那些代码，你将看到定义一种 S-表达式语法来表示 CSS 中的多种结构仍然是可能的。

一项更有歧义的底层处理将是添加对生成嵌入式 JavaScript 的支持。如果做法得当，那么为 FOO 添加 JavaScript 支持可以得到两大好处。一个好处是，在你定义出一种可映射到 JavaScript 语法的 S-表达式语法以后，你就可以开始编写 Common Lisp 的宏为你用来编写客户端代码的语言添加新的构造，然后再将它们编译成 JavaScript。另一个好处是，作为从 FOO 的 S-表达式 JavaScript 到正常 JavaScript 转换的一部分，你可以轻松处理不同浏览器的 JavaScript 实现中的那些细微但令人讨厌的区别。这就是说，FOO 所生成的 JavaScript 代码可以要么含有适当的条件代码在不同的浏览器里做不同的事，要么直接根据你想要支持的浏览器来生成不同的代码。然后如果你把 FOO 用在动态生成的页面中，它可以使用来自 User-Agent 中的信息来让请求可以生成针对该浏览器的 JavaScript 代码。

如果这些事情激起了你的兴趣，那么你应当自行去实现它们，因为这已经是本书实践性内容的最后一章了。在下一章里我将做个总结，同时简要讨论一些我尚未在本书其他部分里涉及到的主题，包括如何查找第三方库，如何优化 Common Lisp 代码，以及如何分发 Lisp 应用程序。

第32章 结论：下一步是什么？

我希望到目前为止你终于明白本书的标题并非反语了。不过，确实还有一些领域对你的编程实践非常重要但我却尚未谈及。例如，我还没有提到任何有关开发图形用户接口（GUI）的内容，也没有提过如何连接到关系型数据库、如何解析 XML，或是如何编写模拟多种网络协议客户端的程序。类似地，还有两个当你用 Common Lisp 来编写实际的应用程序时将变得非常重要的主题我尚未讨论：优化你的 Lisp 代码，以及为你用于分发的应用程序打包。

我明显不打算在这最后一章里深入讨论所有这些主题。相反，我将给你一些指点以便你去追求 Lisp 编程中你最关心的任何方面。

32.1 查找 Lisp 库

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

获取一个用来做你需要做的事情的库的最简单方式可能就是简单地查看你的 Lisp 实现。多数实现都提供了至少一些语言标准里没有指定的功能。商业 Common Lisp 厂商尤其倾向于通过提供用于它们的实现的附加库来证明它们的价值。例如，Franz 的 Allegro Common Lisp 企业版就自带了用于解析 XML，进行 SOAP 通信、生成 HTML、连接到关系型数据库，以及多种构建图形接口的方式，诸如此类。另一个卓越的商业 Lisp 平台 LispWorks 提供了几个类似的库，其中包括广泛使用的可移植 GUI 工具箱 CAPI，它可被用来开发能够运行在任何 LispWorks 支持的操作系统之上的 GUI 应用程序。

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

使用具体实现相关库的唯一缺点是它们将你捆绑在了提供这些库的实现上。如果你正在分发面向最终用户的应用程序或者正在一台你所控制的服务器上部署一个基于服务器的应用程序，那么这可能不是什么大问题。但如果你想要编写可以共享给其他 Lisp 程序员的代码或者你只是简单地不想把自己捆绑在特定实现上，那么这就有些讨厌了。

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

为它们包含适当的读取期条件化以工作在多个实现上¹——你获得它们的最佳途径就是通过 Web。虽然考虑到很多 URL 通常会在被打印到纸上以后立即失效，但下面是三个目前最佳的开始点：

- ❑ **Common-Lisp.net** (<http://www.common-lisp.net>) 是一个承载免费和开源 Common Lisp 项目的站点，它提供了版本控制、邮件列表以及项目页面的 Web 服务。在该站点启动的前一年半里，有将近 100 个项目被注册。
- ❑ **Common Lisp Open Code Collection (CLOCC)** (<http://clocc.sourceforge.net/>) 是一个免费软件库的相对古老的集合，它倾向于在 Common Lisp 实现之间可移植和自包含，不依赖于任何没有包含在 CLOCC 本身的库。
- ❑ **Cliki** (<http://www.cliki.net/>) 是一个提供给 Common Lisp 中各种免费软件的 Wiki 站点。尽管和其他任何 Wiki 一样，它可能随时发生变化，但通常它含有相当多的到各种库和多种开源 Common Lisp 实现的链接。该站点的运行所依赖的软件也是用 Common Lisp 写成的。

运行 Debian 或 Gentoo 发行版的 Linux 用户也可以轻松地安装数量不断增长中的 Lisp 库，它们通过这些发行版的打包工具——Debian 的 `apt-get` 和 Gentoo 的 `emerge`——被打包在了这些发行版之中。

我将不会在这里推荐任何特定的库，因为这些库的情况每天都在发生变化，无论是开源的还是商业的。在对 Perl、Python 和 Java 的库集合觊觎了多年以后，Common Lisp 程序员在过去的几年里也终于开始接受为 Common Lisp 提供类似库集合的挑战了。

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

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

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

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

¹ Common Lisp 的读取期条件化和宏使得开发可移植库成为可能，这些库本身不做任何事，但却在不同实现为语言标准中没有指定的那些功能所提供的 API 之上提供了一个通用的 API 层。来自第 15 章的可移植路径名库就是这类库的一个例子，它在具体实现相关的 API 上尽可能地平滑了对于语言标准的不同解释。

32.2 与其他语言接口

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

语言标准并未指定一种让 Lisp 代码得以调用其他语言所写成的代码的机制，甚至也没有要求具体实现提供这样一种机制。不过近年来，几乎所有的 Common Lisp 发行版都支持所谓的“外部函数接口”(Foreign Function Interface 或简称 FFI)。²FFI 的基本工作是允许你给 Lisp 足够的信息以便其可以链接外部代码。这样，如果你打算调用一个来自某 C 库的函数，你需要告诉 Lisp 如何将传递给该函数的 Lisp 对象转化成 C 类型，然后再将该函数的返回值转化回 Lisp 对象。不过，每个实现都提供了它们自己的 FFI，每种 FFI 都有稍微不同的功能和语法。某些 FFI 允许从 C 向 Lisp 回调，而另一些则不允许。Universal Foreign Function Interface (UFFI) 项目提供了超过半打的不同 Common Lisp 实现上的可移植 FFI 兼容层。它的工作方式是定义其自己的宏，然后展开成其所运行在的实现上的适当 FFI 代码。UFFI 的思路是选取最底层的共同特征，这意味着它无法充分利用不同实现的 FFI 的所有特性，但它确实提供了一种构建基本 C API 外围 Lisp 封装的好方法。³

32.3 让它工作，让它正确，让它更快

正如以前多次被提到，并分别由 Donald Knuth、C. A. R. Hoare 和 Edsger Dijkstra 所强调过，过早地进行优化是万恶之源。⁴Common Lisp 是一门杰出的语言，使用它你在拥有充分表达能力的同时还可以得到很高的性能。如果你曾经听到过有关 Lisp 很慢的传言的话可能会感到惊奇。在 Lisp 的早期岁月里，当计算机还在使用穿孔卡片来编程时，Lisp 的高水平特性可能确实让其慢于竞争对手，换句话说，汇编和 FORTRAN。但那已经是很久以前的事情了。在同一时期，Lisp 曾经被用于从创建复杂 AI 系统到编写操作系统等一系列任务中，并且大量的工作被用来找出如何将 Lisp 编译成高效的代码。在本节里我将讨论关于为何 Common Lisp 是用来编写高性能代码的杰出语言的一些理由，以及相关的一些技术。

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

Common Lisp 的动态特性确保了代码的灵活性，这使其易于尝试不同的方法。给定有限的时间来编写一个程序的话，如果你不必花很多时间从死胡同里出来的话，多半会写出一个高性能的

² 外部函数接口基本上等价于 Java 中的 JNI、Perl 中的 XS，或者 Python 中的扩展模块 API。

³ 在写这本书时，UFFI 的两大缺点是缺少对从 C 到 Lisp 回调的支持——很多但并非全部实现的 FFI 都支持，以及缺少对 CLISP 的支持，后者的 FFI 很好但与其他实现的区别很大以至于无法轻易集成到 UFFI 模型之中。

⁴ Kunth 过去曾在其出版物中说过许多次，包括在他的 1974 年 ACM 图灵奖论文《作为艺术的计算机编程》和他的论文《带有 goto 语句的结构化程序》中。在他的论文《TeX 的错误》中，他将该说法归功于 C. A. R. Hoare。而 Hoare 在一封 2004 年发给 phobia.com 的 Hans Genwitz 的 e-mail 中说他不记得这一说法的起源了，但他可以将其归功于 Dijkstra。

版本来。在 Common Lisp 中，你可以尝试一种思路，如果发现其不可行就立即切换到下一个，而不必花费大量时间来使编译器得以通过你的代码并等待编译完成。你可以先写出一个函数的某个直接而低效的版本——一份代码草图——来检查你的基本思路是否可行，如果是的话再将该函数替换成一个更复杂但却更高效的实现。并且就算整个思路被发现存在问题，你也不必将时间浪费再调整一个不再需要的函数上，这意味着你有更多的时间用来找出一个更好的方法。

Common Lisp 是用于开发高性能软件的下一个原因在于，大多数 Common Lisp 实现都带有成熟的编译器，可产生相当高效的机器码。我将很快谈及如何帮助这些编译器来生成可与 C 编译器生成的代码相媲美的代码，但这些实现已经比那些实现不成熟或使用更简单的编译器或解释器的语言快得多了。另外，由于 Lisp 编译器在运行期可用，因此 Lisp 程序员拥有一些其他语言难以模拟的可能性——你的程序可以在运行期生成 Lisp 代码，然后再被编译成机器码来运行。如果生成的代码打算运行足够多次的话，那么由此带来的好处将是巨大的。或者，即便不使用运行期的编译器，闭包也给了你另一种将机器码与运行期数据混合使用的方式。例如，CL-PPCRE 正则表达式库运行在 CMUCL 上的时候，在某些基准测试上比 Perl 的正则表达式引擎还要快，即便 Perl 的引擎是用高度优化的 C 写成的。这在很大程度上是因为在 Perl 中一个正则表达式被转化成了本质上是字节码的东西，然后被插入到了正则引擎中，而 CL-PPCRE 直接将一个正则表达式转译成了一个编译了的闭包树，这些闭包通过正常的函数调用机制来调用彼此。⁵

不过，即便使用了正确的算法和高质量的编译器，你可能仍然无法达到你所需要的原始速度。那么接下来就要思考性能分析和调优了。在 Lisp 中，和在任何语言中一样，关键在于首先要进行分析以找到你程序中最花时间的性能瓶颈，然后再考虑如何让这些部分提速。⁶

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

你可以使用 TIME 来完成一些简单粗暴的分析以缩小你对瓶颈的搜索范围。例如，假设你由一个花费很长时间来运行的函数，并且它还调用了其他两个函数——类似下面这样：

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

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

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

⁵ CL-PPCRE 还利用了另一个我没有讨论过的 Common Lisp 特性，编译器宏 (compiler macro)。一个编译器宏是一个特殊类型的宏，它提供了优化一个特定的函数调用的机会，方法是将对该函数的调用转化成更高效的代码。CL-PPCRE 为其接受正则表达式参数的函数定义了编译器宏。这个编译器宏通过在编译期解析正则表达式来优化那些带有常量正则表达式的函数调用，而不是将它们留到运行期再处理。关于编译器宏的更多信息，参见你所喜爱的 Common Lisp 参考中的 DEFINE-COMPILER-MACRO 部分。

⁶ “过早地优化”中的词汇“过早”完全可以被定义成“在分析之前”。请记住就算你可以将一些代码的速度提高到几乎不需要花时间的程度，你的整个程序的性能提升也仅限于那段代码在程序运行时间中所占的比率。

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

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

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

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

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

```
CL-USER> (foo)

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

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

从这些输出中，很明显可以看出 `foo` 的大多数时间都花在了 `baz` 上。

当然，如果你想要分析的形式被重复调用的话，那么来自 `TIME` 的输出就显得笨拙了。你可以使用函数 `GET-INTERNAL-REAL-TIME` 和 `GET-INTERNAL-RUN-TIME` 来构造你自己的测量工具，它们返回一个按照常量 `INTERNAL-TIME-UNITS-PER-SECOND` 的值逐秒递增的数值。`GET-INTERNAL-REAL-TIME` 测量的是“挂钟时间”，也就是实际流逝的时间量，而 `GET-INTERNAL-RUN-TIME` 则测量某种实现定义的值，例如 Lisp 实际执行的时间量或是 Lisp 执行用户代码而不包括垃圾收集等内部例行公事在内的时间量。下面是一个简单而有用的分析工具，其使用了一些宏和 `GET-INTERNAL-RUN-TIME`：

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

(defmacro with-timing (label &body body)
```

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

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

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

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

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

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

你可以明显地从多个角度让这段分析代码变得更专业。此外，你的 Lisp 实现也很有可能提供了它自己的分析工具，并且由于它们具有对实现内部的访问权限，因此能够得到对用户层代码未必可见的一些信息。

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

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

取一个简单的例子，考虑下面这个 Common Lisp 函数：

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

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

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

那么你很可能你会发现 Common Lisp 版本会慢很多，即便当你的 Common Lisp 实现号称带有高质量的原生编译器时。

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

另一方面，在 C 语言中，由于所有变量的类型都是声明了的，因此编译器精确地知道 `a` 和 `b` 将保存何种类型的值。并且由于 C 语言中的算术在加法的结果过大而无法用返回值的类型表示时只是简单地溢出，不做任何溢出检测，也不会在数学意义上的和过大而无法填入单个机器字时分配一个 `bignum` 对象。

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

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

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

其中的 `DECLARE` 表达式并非 Lisp 形式；相反，它是 `DEFUN` 语法的一部分，并且必须出现在函数体中其他任何代码之前。⁷ 该声明声称形参 `x` 和 `y` 传递的参数将总是 `fixnum` 的。换句话说，这是一个对编译器的承诺，并且编译器被允许生成假设你所言为真的代码。

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

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

不过，即便是这个版本也需要更多的一个声明，从而让 Common Lisp 编译器像 C 编译器那样生成快速但危险的代码。`OPTIMIZE` 声明被用来告诉编译器如何平衡 5 个量：被生成的代码的速

⁷ 声明可以出现在引入新变量的多数形式中，例如 `LET`、`LET*` 和 `DO` 家族的循环宏。`LOOP` 有其自己的用于声明循环变量类型的语法。第 20 章里提过的特殊操作符 `LOCALLY` 除了创建一个可用来书写声明的作用域以外不做任何其他事。

度；运行期错误检查的程度；代码的内存使用，包括代码本身的大小和运行期的内存占用；随代码提供的调试信息的数量；以及编译过程本身的速度。一个 `OPTIMIZE` 声明由一个或多个列表构成，其中每个列表都含有符号 `SPEED`、`SAFETY`、`SPACE`、`DEBUG` 和 `COMPILE-SPEED` 中的一个，以及一个从 0 到 3 的数值，包括 0 和 3 在内。该数值指定了编译器应当给予对应量的相对权重，其中 3 代表最重要，而 0 意味着完全不重要。这样，为了让 Common Lisp 将 `add` 编译到跟 C 编译器差不多的程度，你可以写成下面这样：

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

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

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

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

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

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

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

;; code start: #x737496f4:
0: 55          pushl  ebp
1: 8b ec        movl   ebp,esp
3: 56          pushl  esi
4: 83 ec 24    subl   esp,$36
7: 83 f9 02    cmpl   ecx,$2
10: 74 02      jz    14
12: cd 61      int   $97 ; SYS::TRAP-ARGERR
14: 80 7f cb 00  cmpb   [edi-53],$0 ; SYS::C_INTERRUPT-PENDING
18: 74 02      jz    22
20: cd 64      int   $100 ; SYS::TRAP-SIGNAL-HIT
22: 8b d8      movl   ebx,eax
```

```

24: 0b da    orl    ebx,edx
26: f6 c3 03 testb  bl,$3
29: 75 0e    jnz    45
31: 8b d8    movl   ebx,eax
33: 03 da    addl   ebx,edx
35: 70 08    jo     45
37: 8b c3    movl   eax,ebx
39: f8        clc
40: c9        leave
41: 8b 75 fc movl   esi,[ebp-4]
44: c3        ret
45: 8b 5f 8f movl   ebx,[edi-113] ; EXCL:::_ZOP
48: ff 57 27 call   *[edi+39] ; SYS::TRAMP-TWO
51: eb f3    jmp    40
53: 90        nop
; No value

```

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

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

现在再次反汇编 add 并查看是否这些声明产生了任何效果。

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

;; code start: #x7374dc34:
0: 03 c2    addl eax,edx
2: f8        clc
3: 8b 75 fc movl esi,[ebp-4]
6: c3        ret
7: 90        nop
; No value
```

看来确实有效果。

32.4 分发应用程序

另一个对于实践有重要意义但我却没有在本书其余部分谈论过的主题，是如何分发用 Lisp 编写的软件。我忽略该主题的主要原因是有许多种不同的方式可以做到这点，并且具体哪一种是最好的取决于你需要分发的软件类型、目标用户，以及所使用的 Common Lisp 实现。在本节里我将对其中的一些不同的选择做一个概括。

如果你在编写打算共享给其他 Lisp 程序员的代码，那么发行它们的最直接方式就是提供源代码。⁸ 你可以将一个简单的库以单个源代码文件的形式来发布，然后程序员们可以用 LOAD 将其

⁸ COMPILE-FILE 所产生的 FASL 文件是实现相关的，并且不一定能在同一个 Common Lisp 实现的不同版本间兼容。这样，使用它们就不是分发 Lisp 代码的一种良好方式。它们可能有用的一种情况是作为应用在特定实现的已知版本里运行的应用程序的补丁来提供。追加一个补丁的方法就是 LOAD 这个 FASL 文件，而由于 FASL 可以包含任意代码，它可被用来通过提供新的代码定义来升级已有的数据。

加载到他们的 Lisp 映像里，或者有可能先用 `COMPILE-FILE` 将其编译然后再加载。

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

近年来最为广泛使用的系统定义工具是 ASDF，它的全称是 Another System Definition Facility。⁹ ASDF 背后的基本思想是，你在 ASD 文件中定义系统，然后 ASDF 提供了一些系统上的操作，包括加载或编译它们等。一个系统也可以被定义成依赖于其他的系统，后者将在必要时被加载。例如，下面给出了 `html.asd` 的内容，它是来自第 31 和 32 章的 FOO 库的 ASD 文件：

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

(defsystem html
  :name "html"
  :author "Peter Seibel <peter@gigamonkeys.com>"
  :version "0.1"
  :maintainer "Peter Seibel <peter@gigamonkeys.com>"
  :license "BSD"
  :description "HTML and CSS generation from sexps."
  :long-description ""
  :components
  ((:file "packages")
   (:file "html" :depends-on ("packages"))
   (:file "css" :depends-on ("packages" "html")))
  :depends-on (:macro-utilities))
```

如果你在变量 `asdf:*central-registry*`¹⁰ 中所列出的一个目录里添加了一个到该文件的符号链接，那么你就可以通过键入下面的内容：

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

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

你将发现大多数免费和开源的 Common Lisp 库都带有一个 ASD 文件。它们中的一些还有可能实用其他系统定义工具，例如相对比较古老的 `MK:DEFSYSTEM` 或者甚至是库作者自己设计的工具，但整个流行趋势是向 ASDF 发展。¹¹

⁹ ASDF 最初是由 SBCL 的开发者之一 Daniel Barlow 所编写的，并且长久以来就是 SBCL 的一部分，也以独立库的形式来分发。它最近已经被采纳并包含在诸如 OpenMCL 和 Allegro 等其他实现中。

¹⁰ 在不支持符号链接的 Windows 上，其工作方式略有不同但几乎也是差不多的。

¹¹ 另一个工具 ASDF-INSTALL，其构建在 ASDF 和 MK:DEFSYSTEM 之上，提供了一种从网络上自动下载和安装库的

当然，尽管 ASDF 让 Lisp 程序员可以容易地安装 Lisp 库了，但它对于你想要给一个不了解或不关心 Lisp 的最终用户打包一个应用程序时不能带来多大的帮助。如果你正在分发一个纯面向最终用户的应用程序，那么可以假定你想要提供一些东西，让用户可以下载、安装和运行而无需知道任何有关 Lisp 的知识。你不能期待它们单独地下载并安装一个 Lisp 实现。并且你希望他们能否像运行其他应用程序一样地运行你的程序——通过双击 Windows 或 OS X 上的一个图标，或者在 Unix 命令行下输入该程序的名字。

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

更复杂的问题在于，“程序”这个概念在 Lisp 中并没有很好的定义。正如你在本书中所看到的，在 Lisp 中开发软件的过程是一个不断修改你的 Lisp 映像中的定义和数据集的增量过程。“程序”只是映像在通过加载含有创建适当定义和数据的代码的.lisp 或.fasl 文件所达到的一个特定状态。随后你可以将一个 Lisp 应用程序分发成一个 Lisp 运行时环境外加一组 FASL 文件，以及一个负责启动运行时环境、加载 FASL 文件并以某种方式调用适当的启动函数的可执行程序。不过，由于事实上加载 FASL 可能需要花很多时间，有其是当你需要做一些计算来设置环境的状态时，因此多数 Common Lisp 实现都提供了一种导出映像文件的方式——将一个运行中的 Lisp 环境的状态保存在一个称为“映像文件 (image file)”或“核心 (core)”的文件里。当一个 Lisp 运行时环境启动时，它做的第一件事就是加载一个映像文件，这可以比通过加载 FASL 文件来重建所有状态花费的时间少得多。

正常情况下这个映像文件时一个只含有语言所定义的标准包和具体实现所提供的附加包的缺省映像。但在多数实现里，你都有某种方法可以指定一个不同的映像文件。这些，与其将一个应用程序打包成一个 Lisp 运行时环境外加一堆 FASL，你还可以将其打包成一个 Lisp 运行时环境外加含有构成你应用程序的所有定义和数据的单个映像文件的形式。然后你所需要的就是一个程序，它可以用适当的映像文件来启动 Lisp 运行时环境并调用作为该应用程序入口点的那个函数。

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

在 Mac OS X 上事情甚至变得更奇妙了——由于 Mac OS X 上的所有应用程序都被打包成了.app 应用程序捆绑 (bundle)，其本质上就是带有特定结构的一个目录，因此将一个 Lisp 应用程序的所有部分打包成一个可以双击的.app 应用程序捆绑完全没有任何困难。Mikel Evins 的 Bosco 工具可以让创建用于运行在 OpenMCL 上的应用程序的.app 捆绑变得更容易。

简单方式。学习 ASDF-INSTALL 的最佳途径是阅读 Edi Weitz 的“*A tutorial for ASDF-INSTALL*” (<http://www.weitz.de/asdf-install/>)。

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

32.5 何去何从

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

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

另外，除了我在“查找 Lisp 库”那一节所提到的站点，你可能还需要浏览 Common Lisp HyperSpec（也称为 HyperSpec 或 CLHS），一个 ANSI 语言标准的 HTML 版本，它由 Kent Pitman 制作并通过 LispWorks 发布在 <http://www.lispworks.com/documentation/HyperSpec/index.html>。HyperSpec 并不是一个学习向导，但它是你在无需从 ANSI 购买语言标准的打印拷贝的情况下可以获得的关于这门语言的权威指导，并且它更适合日常使用。¹²

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

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

如果你想要继续啃书本，那么这里有一些推荐书目。如果想要一本与你桌面厚度差不多的精美的参考书，可以选择 David Margolies 的《ANSI Common Lisp Reference Book》(Apress, 2005 年)。¹³

对于 Common Lisp 对象系统的更多内容，你可以从 Sonya E. Keena 的《Object-Oriented

¹² SLIME 带有一个 Elisp 库，它允许你自动跳转到任何由标准所定义的名字在 HyperSpec 中的对应项上。你还可以下载一份 HyperSpec 的完整拷贝以实现本地的离线浏览。

¹³ 另一本经典的参考书是 Guy Steele 的《Common Lisp: The Language》(Digitool Press, 1984 和 1990 年)。该书的第一版，也称为 CLtL1，在很多年里都是该语言的事实标准。在等待官方的 ANSI 标准完成时，Guy Steele——标准化委员会的成员之一——决定发布第二版以填补 CLtL1 和未来标准之间的鸿沟。该书的第二版，现在称为 CLtL2，本质上标准化委员会在接近完成的一个特定时间里的工作的快照，接近但并不等于标准化进程的结束。相应地，CLtL2 与最终的标准在一些方面有所区别，这使得它不是一个很好的日常参考。不过，它是一份极具用的历史文献，尤其是它包含了关于某些特性在完成之前被标准所丢弃从而没能成为标准的一部分的相关说明，以及为何特定的一些特性采用了标准中所定义的方式。

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

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

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

对于那些更喜欢用理论的眼光来看事物的读者——或者只是想知道作为 MIT 的计算机学院新生是什么感觉——Harold Abelson、Gerald Jay Sussman 和 Julie Sussman 的《Structure and Interpretation of Computer Programs》第二版(M. I. T. Press, 1996 年) 是已经使用 Scheme 来教授重要编程概念的经典计算机科学文献。每个程序员都可以从该书中受益良多——不过要注意的是 Scheme 和 Common Lisp 之间有许多重要的区别。

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

另一方面, Bertrand Meyer 的《Object-Oriented Software Construction》(Prentice Hall, 1997 年) 给出了一个关于来自 Eiffel 发明人的静态语言思想的杰出解释, Eiffel 是 Simula 和 Algol 的一个经常受到重视的后裔。它含有很多值得思考的东西, 即便是对于那些工作在诸如 Common Lisp 这类动态语言的程序员来说也是这样。特别的是, Meyer 关于契约式设计(Design By Contract)的思想对于应该如何使用 Common Lisp 的状况系统也有很多参考价值。

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

最后, 出于乐趣的考虑, 并且也是为了了解 Lisp 和 Lisp 程序员对黑客文化的影响, 可以看看 Eric S. Raymond 所编译的《The New Hacker's Dictionary》第三版(MIT Press, 1996 年), 该书基于 Guy Steele 所编辑的最初的《The Hacker's Dictionary》(Harper & Row, 1983 年)。

但是不要让所有这些建议影响你的编程——真正想学好一门语言的唯一方法是去实际使用

它。如果你已经学到这里了，那么你肯定已经准备好这样做了。那么祝你玩得开心！



田春

网名“冰河”，Glority Software资深软件工程师，前网易杭州研究院高级开发工程师和系统管理员，资深Common Lisp程序员。2001~2005年就读于浙江大学。2003年起开始学习Common Lisp，精通Lisp史和各种实现，2007年起成为 LispWorks 付费用户，Common Lisp社区的网络专家，开源项目cl-net-snmp（SNMP协议库）的作者，usocket跨平台网络库的主要维护者，common-lisp.net站点管理员，水木社区（newsmth.net）函数型编程语言（FuncProgram）版主，美国Versata/Gensym公司技术顾问。曾在2008年翻译了Paul Graham的*On Lisp*一书，在ILC 2009（国际Lisp会议）上发表学术论文，在《程序员》杂志上发表Common Lisp专题文章，并在网上撰写过大量相关的技术文章。

Practical Common Lisp

实用Common Lisp编程

Lisp是计算机科学领域的经典语言之一，它被许多资深程序员视为编程语言中的圣杯。Lisp语言由约翰·麦卡锡提出，它构建在久经考验的各种编程思想之上，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。

《Unix编程艺术》的作者Eric Raymond认为Lisp语言对黑客特别重要，他在“如何成为黑客”一文中说过：“Lisp很值得学习。掌握它以后，你会感受到它带来的极大启发。这会大大提高你的编程水平，使你成为一名更好的程序员，即使你在实际工作中很少用到Lisp。”

《黑客与画家》的作者Paul Graham更是Lisp的倡导者，他在宣传Lisp的文章“拒绝平庸”中说到：“Lisp语言的好处不在于它有一些狂热爱好者才明白的优点，而在于它是目前最强大的编程语言。它没有得到广泛使用的原因就是因为编程语言不仅仅是技术，也是一种习惯性思维，而习惯非常难以改变。”

书中展示了如何用Common Lisp编写一些真实可用的软件，远远超越了笨拙的学院派训练或简单的编辑器定制，读者可以发现，Lisp即便在其许多特性已被其他语言采纳以后仍然有其独到之处。书中用超过三分之一的篇幅讲述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、通过网络流传输MP3的服务器。

Apress®

图灵社区：www.ituring.com.cn

反馈/投稿/推荐信箱：contact@turingbook.com

热线：(010)51095186转604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-26374-2



ISBN 978-7-115-26374-2

定价：89.00元