Writing GNU Emacs Extensions

中文版

英文作者： Bob Glickstein

英文版本： April 1997: First Edition.

中文翻译： slegetank

中文版本： 二零一八年十二月

中文编译： 张泽鹏

编译时间： 2022 年 11 月 14 日

This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste.

O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-261-1

目录

前言

在你能够扩展 Emacs 之前，它已经是世界上功能最强大的文本编辑器了。它不仅能满足你平常所能想到的一切需求（段落排版，行居中对齐，正则搜索，将一整块文本设置为大写），不仅是它具有很多高级的特性（源代码中的括号匹配跳转，语法高亮，以及每个键位以及其他命令的在线帮助），而且它还具有许许多多你连做梦都想不到的特性。你可以使用 Emacs 来阅读和编写 email 以及浏览网页；你可以在里面使用 FTP 来自由的编写远程文件；你可以让它提醒你即将到来的会议、约会以及纪念日。假如这还不够的话，Emacs 还可以陪你玩五子棋（大多数情况下它会赢）；它可以告诉你今天是玛雅历的哪一天；它甚至还能分解质因数。

看上去 Emacs 用户花费这么多时间来做出如此多种多样的功能简直是疯了。大多数程序员将文本编辑器看成是编写其他软件的工具；为什么要花费这么多时间来改变编辑器本身呢？木匠不会耗费精力改造自己的锤子；管钳工不会耗费时间改造自己的钳子；他们只是使用他们的工具来完成自己手头的工作而已。Emacs 用户为什么要不同呢？

答案是如果木匠和管钳工知道如何使自己的工具变得更好用的话，他们也会这么做。谁比他们自己更明白自己需要什么样的工具呢？但他们毕竟不是工具制作专家。另一方面，Emacs 是一种特殊的工具：它是软件，它和使用它编写的软件并没有什么本质上的不同。Emacs 的用户通常是程序员，而编写 Emacs 其实也就是编程。Emacs 用户完全可以做自己的工具专家。

这本书将使用一系列的实例由浅入深的教给你如何编写 Emacs Lisp。我们将以向 Emacs 启动配置文件中添加简单的配置作为开始，在最后我们将会着手编写“主模式（major modes）”以及修改 Emacs 自己的“命令循环 (command loop)”。在这个过程中我们将会学到变量，键位表，交互式命令，buffers，窗口，处理 I/O 等。本书中所提到的 Emacs 特指 GNU Emacs。有许多编辑器自称为 Emacs。以下是权威的 On-line Hacker Jargon File(version 4.0.0, 24-Jul-1996) 所叙述的一点历史：

> [Emacs] was originally written by Richard Stallman in TECO under ITS at the MIT Al lab; AI Memo 554 described it as ”an advanced, self-documenting, customizable, extensible real-time display

> editor." It has since been re-implemented any number of times, by various hackers, and versions exist that run under most major operating systems. Perhaps the most widely used version, also written by Stallman and now called "GNU EMACS" or GNUMACS, runs principally under UNIX. It includes facilities to run compilation subprocesses and send and receive mail; many hackers spend up to 80% of their tube time inside it. Other variants include GOSMACS, CCA EMACS, UniPress EMACS, Montgomery EMACS, jove, epsilon, and MicroEMACS.

书中的示例全部在 GNU Emacs 19.34 以及 20.1 版本的一个预发行版本上编写以及测试通过。参看附录E，获取从哪里找到 Emacs 版本的信息。

我以自身使用 Emacs 的开发经历作为书中示例选择的指导。书中的示例实际上讲述了我自己的 Emacs 使用经历的演变。例如，我在最初使用 Emacs 的时候就知道一定要让该死的 *BACKSPACE* 键不要触发在线帮助！也许你也有这个问题。解决这个问题将作为下一章的第一个例子。

在我使用了 Emacs 一小段时间之后，我发现自己需要许多处理光标移动的快捷键。就我所知，使用 Emacs 提供的原生功能就可以轻松对其定制。我们将会在第二章中看到一些这样的例子。这之后我需要一种方法来避免我经常遇到的输入错误：我想要按下 *CONTROL-b*，但实际上常常按下的却是 *CONTROL-v*。我本想让光标向左移动一点，却向下移动了好几屏。在第三章中你将会看到这也很容易进行优化。当我开始处理记录着很多巧妙的谚语的文件时，我需要特定的工具来处理特定格式的文件。我们将会在第九章中看到一些这样的例子。

除了每章开始解决问题时提出的最初的例子之外，我们还会看到一些别的简单例子，每个例子都有自己的章节。每个章节都会提出一些需要使用 Emacs Lisp 解决的问题，然后展示出一个或一些方法来解决这些问题。然后，就像现实中的扩展都要变得更实用、更通用，我们通常会在前往下一个主题前对其进行一到两次改进。

每个例子都基于前面的例子，并且带入一些自己的新东西。在本书的最后，我们将覆盖大部分 Emacs Lisp 编程的概念并且讨论如何使用在线文档和其他信息来帮你更快的使用 Emacs Lisp 来满足你自己的需求。俗话说得好：授人以鱼不如授人以渔。

这本书假设你熟悉编程并且正在使用 Emacs。如果你熟悉 Lisp 编程语言中的一些方言（Emacs Lisp 也是其中一种）将会有帮助，但不是必要。通过我们每个章节中使用的例子，Lisp 编程的要领将会很快很容易的理清。你也可以参考附录 B，它简要的描述了 Lisp 的基本知识。

如果你不熟悉 Emacs 的基本概念，请阅读 Debra Cameron，Bill Rosenblatt，和 Eric Raymond 所写的《Learning GNU Emacs, 2nd edition》。你也可以使用 Emacs 自带的在线帮助，特别是 info 手册，也就是《The GNU Emacs Manual》。

如果你希望更全面的理解 Lisp 编程，我推荐阅读 David Touretzky 的《Common Lisp: A Gentle Introduction to Symbolic Computation》。

本书不是 Emacs Lisp 的参考手册；实际上也并没有特别深入的讲解语言方面的知识。主题的选择更倾向于更有教学意义而不是涵盖所有范畴。在阅读时最好以从前往后的顺序阅读以得到最好的效果。自由软件基金会发布的《The GNU Emacs Lisp Reference Manual》是这方面的权威。它有多种印刷本和电子版；详情参见附录E。

0.1 什么是 Emacs？

认为 Emacs 仅仅是一个可编程的文本编辑器是完全错误的。同样的说它是 C 语言编辑器也是不对的。虽然看起来很较真，但实际上编写 C 语言和编写文本是两种完全不同的工作，Emacs 通过变成两个不同的编辑器来包容这种差异。当编写代码时，你不会关心段落结构。当编写文本时，你不会关心自动缩进。

Emacs 同时也是一个 Lisp 代码编辑器。它也是 16 进制数据文件的编辑器。它也可以作为大纲的编辑器。它也可以作为文件目录的编辑器，压缩文件的编辑器，email 的编辑器等等。每一种编辑器都是一种 Emacs 的模式（mode），即一系列将 Emacs 的原生要素和行为组合起来以实现新特性的 Lisp 代码。因此每个模式也就是 Emacs 的一种扩展，也就是说如果你把这些模式都除掉的话—删掉所有扩展并且只剩下 Emacs 的核心—那么你就根本没有了任何的编辑器；你只剩下制作编辑器的原材料。你只剩下了编辑器生成器（editor-builder）。

你能用编辑器生成器生成什么呢？当然是编辑器了，但是什么是编辑器呢？编辑器就是一个用来展示和修改某种数据，以及用来帮助与这些数据更友好的进行交互的程序。当编辑文本文件时，规则很简单：每个可见字符按照顺序展示出来，换行符执行换行；一个光标用来表示用户的下一个操作将会发生在数据的什么位置。当编辑目录时就不是那么直观了—路径文件中的数据必须先转换成可读的格式—最终的交互流程要看起来比较人性化。

这个关于编辑器的定义几乎涵盖了所有交互程序的范畴，而这绝非偶然。交互程序总是用来处理某种数据的编辑器。因此可以说，Emacs 在广义上，是一种交互程序的生成器。它是一个 UI 工具包！就像很多好的工具包一样，Emacs 提供了一套 UI 组件，一套操作它们的方法，一个事件循环，一套成熟的 I/O 机制，以及一种用来把它们整合起来的程序语言。UI 组件看起来可能不如 X11，Windows 或者 Macintosh 所提供的那样漂亮，但是就像 Emacs 程序员所发现的，一个超级漂亮的图形工具集往往是多余的。99% 的程序都是文本形式的，不管是数字列表，菜单项，或者填字游戏里的单词（参考第十章所展示的例子）。对于这些程序，Emacs 在功能性、精巧性、简单性以及性能上都要优于其他。

“为什么 Emacs 用户不同？”，这个问题的真正答案并不仅仅是他们花费时间在改造他们的工具上。他们在使用 Emacs 来达到自己所期望的目的：创造出

无穷无尽的新工具（a universe of new tools）。

0.2 本书的组织形式

书中的每章都基于前一章。我建议你从前往后顺序读这本书；这样的话里面的所有安排就都变得有意义了。

第一章 介绍一些你可以对 Emacs 做出的基本修改。这也会使你熟悉 Emacs Lisp，如何对 Lisp 表达式求值，以及那会如何改变 Emacs 的行为。

第二章 教给你如何编写和安装 Lisp 函数来使其正确执行。钩子和一种称为修饰的特性将会被引入。

第三章 如何在不同函数调用间保存信息以及如何使多组函数共同工作—这是编写系统而不仅仅是编写命令的第一步。符号属性和标记将会在这中间被介绍。

第四章 展示一些你极有可能会经常用到的技术：用来改变当前 buffer 和其中字符串的方法。正则表达式会被介绍。

第五章 讨论加载、自动加载以及包的概念，这些特性会在你创建大量相关函数时用到。

第六章 加入一些关于 Lisp 重要特性的背景知识。

第七章 展示如何将相关函数和变量组装进称为“子模式”的包里。这个章节中的核心例子是使 Emacs 中的段落格式化功能工作的更像一个正常的文本处理软件。

第八章 展示 Emacs Lisp 解释器的灵活性，如何控制在何时执行什么，以及如何编写不受运行时错误影响的代码。

第九章 解释子、主模式间的差别，并且为后者提供一个简单的例子：一个专门用来处理谚语文件的主模式。

第十章 定义一个完全改变 Emacs 默认行为的主模式—一个填字游戏谜题编辑器，通过它向你展示 Emacs 对于开发文本相关的应用是多么灵活。

附录B 对 Lisp 的语法，数据类型以及控制结构提供了一个实用的介绍。

附录C 描述了可以用来追踪你的 Emacs Lisp 代码中问题的工具。

附录D 解释了把代码分享给别人时需要遵循的步骤。

附录E 概述了如何在你的系统上得到一个可用的 Emacs 版本。

0.3 获取示例程序

通过浏览器，你可以获取到示例：ftp://ftp.oreilly.com/published/oreilly/nutshell/emacs_extensions

0.3.1 FTP

要使用 FTP，你需要一台能直接访问网络的电脑。下面是一个例子，你需要输入的是其中粗体的部分：

```
% *ftp ftp.oreilly.com*
Connected to ftp.oreilly.com.
220 FTP server (Version 6.21 Tue Mar 10 22:09:55 EST 1992) ready.
Name (ftp.oreilly.com:yourname): *anonymous*
331 Guest login ok, send domain style e-mail address as password.
Password: *yournameayourhost.com* (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> *cd /published/oreilly/nutshell/emacsextensions*
250 CWD command successful.
ftp> *binary* (Very important! You must specify binary transfer for
               gzipped files.)
200 Type set to I.
ftp> *get examples.tar.gz*
200 PORT command successful.
150 Opening BINARY mode data connection for *examples.tar.gz.*
226 Transfer complete.

ftp> *quit*
221 Goodbye.
```

文件格式为 gzipped tar 归档；输入下面的指令展开它：

```
% gzip -dc examples.tar.gz | tar -xvf -
```

System V 系统需要下面的 tar 指令：

```
% gzip -dc examples.tar.gz | tar -xvof -
```

如果 gzip 在你的系统上不存在，那么单独使用 uncompress 以及 tar 指令。

```
% uncompress examples.tar.gz
% tar xvf examples.tar
```

0.4 致谢

感谢 Nathaniel Borenstein，他帮助我驱散了对于 C 的执念并且教会了我欣赏这个世界上多姿多彩的编程语言。

感谢 Richard Stallman 编写了 Emacs—两次—他提出的令人惊奇的言论是对的：黑客编写更好的代码是为了满足自己而并非为了钱。

感谢 Mike McInerny，他固执的坚持使我开始使用 GNU Emacs—即使开始的几次我都认为这并不值得我花时间。

感谢 Ben Liblit 提供的想法，代码以及对于我的 Defer 包（本书中的一章，直到 Emacs 有了自己的功能相同的包，timer）的 bug 修正。其他的帮助来自于 Simon Marshal，他在他的 defer-lock 中使用并且改进了很多其中的想法。Hi，Si。

感谢 Linda Branagan 向我展示了即使像我这样一个平凡的人也能写书。(她当然并不平凡；一点也不。)

感谢 Emily Cox 和 Henry Rathvon 提供的对于填字游戏谜题的一些内行知识。

感谢对于本书的早期草稿做出校对和建议的朋友们：Julie Epelboim，Greg Fox，David Hartmann，Bart Schaefer，Ellen Siever，以及 Steve Webster。

感谢 Zanshin Inc. 以及 Internet Movie Database 允许我在这些工程和这本书之间分配我的工作精力。

感谢编辑，Andy Oram，能够灵活地应对我上面提到的这种杂乱无章的工作。

感谢 Alex，我的狗，在我写这本书的大部分过程中都在我的脚边开心地打转。

最重要的是，感谢 Andrea Dougherty，她鼓励着我，支持着我，做出了无数的牺牲，提供了数不清的服务，在我需要的时候给我陪伴并且在我需要独处的时候离开（而不是反过来），并且在其他所有方面都对我和这本书有益：这一定是爱。

第一章　自定义 Emacs

在本章：

- Backspace 和 Delete
- Lisp
- 按键和字符串
- C-h 绑定到什么
- C-h 应该绑定到什么
- 执行 Lisp 表达式
- Apropos

本章将介绍基本的 Emacs 定制化，并且在这一过程中教给你一些 Emacs Lisp。最简单、最常见的定制之一就是将一个按键的命令复制到另一个按键上。可能你不喜欢 Emacs 的两次按键（`C-x C-s`）来保存文件，因为其他的编辑器通常都只是简单的 `C-s` 。或者你可能只是想按 `C-x` ，却不小心按成 `C-x C-c` ，也就是退出 Emacs，而你希望按下 `C-x C-c` 不要造成这么酷炫的效果。或者也可能，就像下面的例子所展示的，你可能希望对 Emacs 提供给你的键位做出一些自己的修改。

1.1　Backspace 和 Delete

想象你想要输入“Lisp”但却输成了“List”。要改正你的拼写，你是按下 BACKSPACE 键呢还是 DELETE 键?

这当然由你的键盘而定，但是我要问的并不仅仅是一个按键上标记了什么的问题。有时按键上标记的是“Backspace”，有时它又被标记为“Delete”，有时又是“Erase”，有时是一个向左的箭头或是别的什么图案。对于 Emacs 来说，按键上标记着什么并不重要，它在乎的是按下这个键时所触发的数字代码。“向左移动并且删掉前面的字符”可能会产生一个 ASCII 的“backspace”编码（十进制 8，通常表示为 BS）或者一个 ASCII 的“delete”编码（十进制 127，通常表示为 DEL）。

在 Emacs 的默认配置中，认为只有 DEL 表示“向左移动并且删掉前面的

字符”。如果你的 BACKSPACE/DELETE/ERASE 键产生了一个 BS，那么它将不会按照你所希望的那样执行。

更糟的是当你按下 BACKSPACE/DELETE/ERASE 键，它却产生了一个 BS 时。Emacs 认为既然 BS 并不用来左移并删除前面的字符，那么它就可以用来触发另一个方法。结果是，BS 现在触发的方法和按下 `C-h` 时触发的一样。如果你们那里并不需要 `C-h` 来执行左移并删除前面的字符，那么 `C-h` 更好的选择是作为一个 Help 键，而这也是 Emacs 的选择。不幸的是，这意味着如果你的 BACKSPACE/DELETE/ERASE 触发 BS 的话，那么按下它将不会触发 backspace、delete 或者 erase；它将触发 Emacs 的在线帮助。

当 Emacs 的初学者想要修正一个拼写错误时，他们不止一次的被 Emacs 做出了热烈的欢迎。突然一个新的 Emacs 窗口—帮助窗口—弹了出来，提示不幸的用户来选择一些帮助命令。帮助的内容如此的冗长使得用户更加的瞠目结舌。用户恐慌的按下一大堆的 `C-g`（终止当前的操作），伴随着一大堆的终端错误铃声提示。怪不得许多聪明善良的用户选择继续使用安全无害的 vi，而不是成为 Emacs 的疯狂传道者。我每当想起这件事就很伤心，特别是这一情形很容易被修复。Emacs 启动时，它将读取并且执行你的根目录下的 `.emacs` 文件。它使用 Emacs Lisp 作为语言，而读完本书你会发现，只要编写一些 Emacs Lisp 放到 `.emacs` 里，Emacs 几乎没有什么是你不能改变的。我们要关注的第一件事就是向 `.emacs` 中添加一些代码来把 BS 和 DEL 都指定为“向前退格并且删掉一个字符”，而将帮助命令移动到其他键上去。首先我们需要看一下 `.emacs` 文件所使用的语言，Lisp。

1.2　Lisp

自从 1950 年以来已经产生了很多种不同形式的 Lisp。最初它应用于人工智能，它很胜任这份工作，因为它允许符号运算，可以将代码作为数据处理，可以简化复杂数据结构的构建。但是 Lisp 可以做的要比一门 AI 语言多得多。它应用于非常广的问题处理上，这经常被计算机科学家忽视，而 Emacs 用户却知道的很清楚。Lisp 不同于其他编程语言的特性有：

1.2.1　括号前缀表示法

Lisp 中的所有表达式和方法都由括号括起来[1]，方法名通常在参数之前。所以在其他语言中你通常会这么写：

```
x + y
```

[1]批评者通常认为 Lisp 的括号是它标志性的缺点。他们认为，Lisp 是 “Lots of Infernal Stupid Parentheses” 的简写（实际上是 “List Processing” 的简写）。以我来看，这个更简单的符号使得 Lisp 比其他语言更易读，而我希望你也这么认为。

而在 Lisp 中，你会这么写：

```
(+ x y)
```

“前缀表示法”表示运算符在运算对象的前面。当运算符在运算对象中间时，这称为“中缀表示法”。

虽然与通常的习惯不同，但前缀表示法相对于中缀有一些好处。在中缀语言中，要将 5 个变量加起来需要 4 个加号：

```
a + b + c + d + e
```

Lisp 更简洁：

```
(+ a b c d e)
```

而且，不会产生运算符的优先级问题。例如，

```
3 + 4 * 5
```

的结果是 35 还是 23？这需要知道 * 是否比 + 具有更高的优先级。而在 Lisp 中，就不会有这种疑惑：

```
(+ 3 (* 4 5)) ; 结果是23
(* (+ 3 4) 5) ; 结果是35
```

（Lisp 中的注释使用分号，作用到行尾。）最后，中缀语言需要在方法中使用逗号分隔参数：

```
foo(3 + 4, 5 + 6)
```

Lisp 不需要额外的语法：

```
(foo (+ 3 4) (+ 5 6))
```

1.2.2　List 数据类型

Lisp 有一种内建的数据类型称为列表（list）。列表是一种用括号括起来的，不包含或者包含着其他 Lisp 对象的 Lisp 对象。下面是一些列表：

```
(hello there) ; 包含着两个“符号”的列表
(1 2 "xyz") ; 两个数字和一个字符串
() ; 空列表
```

列表可以作为值赋给其他变量，作为参数传递给方法以及作为返回值传递，使用 `cons` 和 `append` 这种方法来进行组合，使用 `car` 和 `cdr` 来进行拆分。后面我们将会更详细地叙述这些知识。

1.2.3　垃圾回收

Lisp 是一种垃圾回收的语言，这意味着 Lisp 会自动的回收你的程序里的数据结构所使用的内存。与之相反的，比如 C 语言，程序员必须显式的使用`malloc`来分配内存，然后显式的使用`free`来释放内存。（在非垃圾回收语言里，`malloc`/`free`这种语句非常容易出错。过早的释放内存是全世界程序错误中最大的原因之一，而忘记释放内存则会造成内存的泄露。）

除了所有这些垃圾回收机制所具有的有点，它也有一个缺点：Emacs 会不时的停下正在做的所有事情，向用户显示“Garbage collecting...”。用户要等到垃圾回收结束才能继续使用 Emacs[2]。这通常只会持续不到 1s，但是却可能非常频繁。后面我们将会学到如何减少垃圾回收发生的实用技巧。

表达式（expression）通常表示 Lisp 代码中的任何一部分或者任何 Lisp 数据结构。所有 Lisp 表达式，不管是代码还是数据，都可以被 Emacs 中内建的 Lisp 解释器执行。对一个变量求值的结果就是访问之前储存在变量中的 Lisp 对象。就像我们下面将要看到的，用来执行 Lisp 函数的方式就是对一个列表求值。

自从 Lisp 发明以来已经产生了许多 Lisp 方言，它们之间各有不同。MacLisp, Scheme 和 Common Lisp 是其中较为有名的。Emacs Lisp 和它们都不一样。这本书只关注 Emacs Lisp。

1.3　按键和字符串

本章的目的是使所有触发 BS 的键同触发 DEL 的键能一样的工作。当然这将导致 `C-h` 不再触发帮助命令。你需要选择其他的键来使用帮助；我自己的方式是使用 META-question-mark。

[2] Emacs 使用了一种标记-清扫的垃圾回收设计，是最简单的垃圾回收实现方式之一。有一些其他的实现方式会更少打扰用户；例如，一种称为“incremental”的方式在执行时不会使 Emacs 当机。不幸的是，Emacs 没有使用这些方式。

1.3.1　META 键

META 键的工作方式和 CONTROL 键以及 SHIFT 键一样，都是需要在按下其他键的同时按着它。这种键被称为修饰键（modifiers）。虽然不是所有键盘都有 META 键。有时 ALT 键起着同样的作用，但是也不是所有键盘都有 ALT 键。无论如何，你都不是必须使用 META 或者 ALT。单次按键 `META-x` 总是可以使用两键序列 ESC x 来替代。（注意 ESC 不是修饰键—你需要先按下 ESC，松开手，再按下 x 键。）

1.3.2　将按键绑定到命令上

在 Emacs 里，每个按键都触发一条命令或者是一个触发命令的多键序列的一部分。就像我们将要看到的，命令是一种特殊的 Lisp 函数。使一个按键触发类似帮助这种命令的行为被称为绑定。我们需要执行一些 Lisp 代码来将按键绑定到命令上。`global-set-key` 是一个用于做这件事的函数。

下面介绍如何调用 `global-set-key` 。记住在 Lisp 里函数调用就是简单的一个括起来的列表。第一个元素是函数名称，剩下的元素全是参数。函数 `global-set-key` 使用两个参数：要绑定的按键序列，以及要绑定的命令。

```
(global-set-key keysequence command)
```

需要注意 Emacs Lisp 是区分大小写的。

我们选择的按键序列是 META-question-mark。这在 Emacs Lisp 中如何表示呢?

1.3.3　字符串表示按键

在 Emacs Lisp 中有一些不同的方式来表示一个按键序列。最简单的是直接使用字符串。在 Lisp 中，字符串是一些被引号括起来的字符序列。

```
"xyz" ; 三个字母的字符串
```

要在字符串中使用双引号，使用反斜杠（\ ）：

```
"I said, \"Look out!\""
```

这表示如下字符串：

```
I said, "Look out!"
```

要在字符串中表示反斜杠需要使用另一个反斜杠对其转义。

普通的按键使用它所代表的字符来表示它。例如，按键 q 在 Lisp 中被字符串“q”所表示。而反斜杠 \ 则写作“\\ ”。

像 META-question-mark 这种特殊字符在字符串里使用特殊的标识符:“`\M-?` ”来表示。虽然字符串里有四个字母，但 Emacs 会将此字符串读为 META question-mark[3]。

在 Emacs 的术语中，`M-x` 是 META-x 的简写，“`\M-x` ”是字符串版本。CONTROL-x 在 Emacs 文档中简写为 `C-x` ，在字符串中表示为“`\C-x` ”。你也可以组合 CONTROL 和 META 键。CONTROL-META-x 简写作 `C-M-x` ，字符串表示为“`\C-\M-x` ”。顺便，” `\C-\M-x` ” 和” `\M-\C-x` ” (META-CONTROL-x) 等价。

(CONTROL-x 在文档里有时也表示为 `^x` ，那么字符串就表示为” `\^x` ”。)

现在我们知道了如何填写 `global-set-key` 的第一个参数:

```
(global-set-key " \verb|\M-?| " command)
```

(另一种书写” `\M-?` ” 的方式是” `\e?` ”。字符串“`\e` ”表示 escape，而 `M-x` 和 Esc x 等价。)

下面我们必须找出 command 需要填写什么。这个参数应该是我们希望 `M-?` 触发的帮助函数的名称，也就是当前 `C-h` 所触发的函数。在 Lisp 中，函数使用符号（symbols）来表示。符号就像其他语言中的函数名或者变量名，虽然 Lisp 在命名时比大多数语言都允许更宽泛的字符集。例如，合法的符号名包括 `let*` 以及 `up&down-p` 。

1.4　C-h 绑定到什么

要找到帮助命令的符号，我们可以使用 `C-h b` ，这将会触发另一个名为 `describe-bindings` 的命令。这是帮助系统众多的命令之一。它会弹出一个列出所有有效键绑定的窗口。查找 `C-h` ，我们可以找到这一行:

```
C-h help-command
```

这告诉了我们 `help-command` 是指向帮助命令的符号。我们的 Lisp 示例即将完成了，但是我们不能只是写下

```
(global-set-key "\M-?" help-command) ; 几乎对了!
```

[3]你可以使用 `length` 函数查看字符串的长度来确认这件事。如果你执行 `(length "\M -?")` ，结果为 1。如何“执行”在本章的后面有介绍。

这是错误的，因为符号只要出现在 Lisp 表达式里就会马上被解释执行。如果符号出现在列表的第一个位置时，那么它将作为函数的名称来执行。否则，它作为变量的值就要被展开。但是当我们运行 global-set-key 时，我们不需要 help-command 所包含的值，不管那是什么。我们需要的是 help-command 这个符号的本身。简而言之，我们希望在传递给 global-set-key 之前不要对符号进行求值。毕竟就我们所知，help-command 并没有作为变量的值存在。

阻止符号（以及其他任何 Lisp 表达式）被求值的方法是在它的前面加一个单引号（' ）进行引用（quoted）。就像这样：

```
(global-set-key "\M-?" 'help-command)
```

我们的 Lisp 例子现在完成了。如果你把它放到你的.emacs 文件中，那么以后当你打开 Emacs 时 M-? 将会触发 help-command 。（马上我们将会学到如何立即触发 Lisp 表达式。）M-? b 将会像 C-h b 一样触发 describe-bindings（这时 M-? 和 C-h 都绑定到了 help-command ）。

顺便，为了说明引用和非引用的区别，下面两条表达式可以达成同样的效果：

```
(setq x 'help-command) ;  setq分配一个变量
(global-set-key "\M-?" x) ; 使用 x 的变量值
```

第一行使变量 x 保存符号 help-command 。第二行使用 x 的值—符号 help-command —绑定给 M-? 。这个例子与上一个的唯一区别是你现在多使用了一个变量 x 。

符号并不是唯一可以被单引号前缀的；任何 Lisp 表达式都能被引用，包括列表，数字，字符串，以及其他我们后面将要学到的表达式。'expr 是下面的简写：

```
(quote expr)
```

这在执行的时候会延缓求值(yield)。你可能已经注意到了符号 help-command 需要引用而字符串参数 “\M-? ” 却不需要。这是因为在 Lisp 里，字符串是自解释的，当字符串被执行时，它返回的是它本身。所以对其进行引用是无害而多余的。数字，字符以及向量（vector）是其他自解释的 Lisp 表达式。

1.5　C-h 应该绑定到什么？

既然我们已经将 help-command 绑定到 M-? ，下面我们需要给 C-h 绑定一些什么。使用前面所描述的同样的流程—也就是说，触发命令 describe-bindings

(使用 C-h b 或者 M-? b)—我们发现 DEL 触发的命令是 delete-backward-char 。

所以我们可以这样写:

```
(global-set-key "\C-h" 'delete-backward-char)
```

现在 DEL 和 C-h 一样了。如果你把下面的命令放到.emacs 里:

```
(global-set-key "\M-?" 'help-command)
(global-set-key "\C-h" 'delete-backward-char)
```

那么以后在 Emacs 里，BACKSPACE/DELETE/ERASE 将会执行正确的事情，不管发出的是 BS 还是 DEL。但是我们如何使他们马上产生效果呢？这需要显式执行（explicit evaluation）这两个表达式。

1.6　执行 Lisp 表达式

有几种方式来显式执行 Lisp 表达式。

1. 你可以将 Lisp 表达式放到一个文件里，然后载入这个文件。假设你把表达式放到文件 rebind.el 里。(Emacs Lisp 文件的后缀名是.el)。你可以键入 M-x load-file RET rebind.el RET 以使 Emacs 来执行文件的内容。如果你把这些内容放到了.emacs 里，你可以使用同样的方法来载入它。但是在你使用了 Emacs 一段时间后，你的.emacs 将会变得越来越大，它的载入将会变得很慢。因此，你不会希望为了一点点改动就重新载入整个文件。因此我们可以使用下一种选择。
2. 你可以使用命令 eval-last-sexp ，这绑定到了[4] C-x C-e 上。(sexp[5]是 S 表达式（S-expression）的简写，也就是符号表达式的简写，也就是 Lisp 表达式的另一种说法。）这个命令将执行光标左边的 Lisp 表达式。所以你要做的是将光标放到第一行的末尾:

   ```
   (global-set-key "\M-?" 'help-command) |
   (global-set-key "\C-h" 'delete-backward-char)
   ```

   然后按下 C-x C-e ；然后移动到第二行尾:

[4]技术上说，我们应该说按键被绑定到了命令上，而不是命令绑定到了按键上。(说按键“绑定到”了命令上正确的表示了这个按键序列只能做一件事—触发这个命令。说命令“绑定”到了一个按键上则表示只有这个按键序列能够触发这个命令，而这并不是真的。）但是一般来说这种误用的“绑定到”并不会引起什么误会。

[5]遗憾地读作“sex pee.”。

```
(global-set-key "\M-?" 'help-command)
(global-set-key "\C-h" 'delete-backward-char) |
```

然后再次按下 C-x C-e 。执行 global-set-key 的结果—一个特别的符号 nil （我们后面将会再次看到）—展示在了 Emacs 屏幕下方的消息区里。

3. 你可以使用命令 eval-expression ，这绑定到了 M-: [6]。这个命令在 minibuffer（屏幕的底部）中提示你输入一个 Lisp 表达式，然后执行它并输出结果。Emacs 的制作者认为 eval-expression 是少数一些对于初学者来说尝试使用会造成危险的命令之一。以我来看，这简直是胡说；不论如何，这个命令在初始时是被禁用的，所以当你尝试使用时，Emacs 告诉你 “You have typed M-:, invoking disabled command eval-expression.”。然后它会显示 eval-expression 的描述并且如下提示：

```
You can now type
Space to try the command just this once,
but leave it disabled,
Y to try it and enable it (no questions if you use it again),
N to do nothing (command remains disabled).
```

如果你选择 Y，Emacs 将会把下面的表达式加入你的.emacs。

```
(put 'eval-expression 'disabled nil)
```

(put 函数和属性列表 (property list) 有关，我们将会在第三章的符号属性中看到它)我的建议是你可以在获得这个提示之前就把它手动加入到.emacs 里，这样你就不会被 “disabled command” 警告所困扰了。当然，当你把这条语句放到.emacs 里之后，使用前面提到的 eval-last-sexp 使它马上生效是一个不错的想法。

4. 你可以使用 *scratch* buffer。这个 buffer 在 Emacs 启动的时候就会自动创建。它使用了 Lisp 交互模式。在这个模式里，按下 C-j 来执行 eval-print-last-sexp ，它很像 eval-last-sexp ，除了它会将结果插入到光标所在的位置。Lisp 交互模式的另一个特性是你可以使用 M-TAB 进行自动补全(触发 lisp-complete-symbol)。Lisp 交互模式在用来调试太长的 Lisp 表达式或者数据结构太复杂的时候特别有用。

不管你使用哪一种方法，执行 global-set-key 表达式的结果是产生了新的按键绑定。

[6]这个按键绑定是 19.29 新引入的。在之前的版本，eval-expression 默认绑定到 M-ESC。

1.7　Apropos

在结束第一个例子之前，让我们讨论一下 Emacs 的最重要的在线帮助特性，`apropos` 。假设你同时拥有 BS 和 DEL 键，你希望 BS 删除光标前面的字符而 DEL 删除后面的。你现在知道了 `delete-backward-char` 用来完成前面的目的，但是你不知道什么命令完成后面的。你确信 Emacs 一定有这么一个命令。但是如何找到它呢?

答案是使用 `apropos` 命令，它允许你使用表达式来搜索所有已知的变量名和函数名。试试这么做[7]:

```
M-x apropos RET delete RET
```

返回值是一个列出了所有符合“delete”的变量和函数的 buffer。如果我们在这个 buffer 里搜索“character”，然后翻到这一部分

```
backward-delete-char
Command: Delete the previous N characters (following if N is negative).
backward-delete-char-untabify
Command: Delete characters backward, changing tabs into spaces.
delete-backward-char
Command: Delete the previous N characters (following if N is negative).
delete-char
Command: Delete the following N characters (previous if N is negative).
```

而函数 `delete-char` 正是我们需要的。

```
(global-set-key "\C-?" 'delete-char)
```

(由于历史原因，DEL 由 CONTROL-question-mark 来触发。)

你可以使用前置参数来执行 `apropos` 。在 Emacs 中，在执行命令前按下 `C-u` 将会向命令传递额外的参数。通常，`C-u` 后面跟着一个数字；例如 `C-u 5 C-b` 表示“将光标向前移动 5 个字符”。有时额外的参数就是你按下的 `C-u` 本身。

当 `apropos` 使用了前置参数时，它不只显示所有符合搜索表达式的函数和变量，它还展示出列表中每个命令绑定的按键 (这不是默认的，因为搜索按键绑定很慢)。使用 C-u M-x apropos `RET` delete `RET` 然后搜索“character”，我们将会得到下面的信息:

[7]所有的 Emacs 命令，不管它们绑定到了哪里 (如果有的话)，都可以通过 M-x `command-name` RET 来执行。自然，M-x 自己也是一个绑定到按键上的命令，`execute-extend-command` ，它会提示输入一个要执行的函数名。

```
backward-delete-char (not bound to any keys)
Command: Delete the previous N characters (following if N is negative).
backward-delete-char-untabify (not bound to any keys)
Command: Delete characters backward, changing tabs into spaces.
delete-backward-char C-h, DEL
Command: Delete the previous N characters (following if N is negative).
delete-char C-d
Command: Delete the following N characters (previous if N is negative).
```

这证实了现在 C-h 和 DEL 都会执行 delete-backward-char ，并且告诉了我们 delete-char 已经有了一个绑定：C-d 。在我们执行

```
(global-set-key "\C-?" 'delete-char)
```

之后，如果我们再次执行 apropos ，我们将会得到

```
backward-delete-char (not bound to any keys)
Command: Delete the previous N characters (following if N is negative).
backward-delete-char-untabify (not bound to any keys)
Command: Delete characters backward, changing tabs into spaces.
delete-backward-char C-h
Command: Delete the previous N characters (following if N is negative).
delete-char C-d, DEL
Command: Delete the following N characters (previous if N is negative).
```

如果我们知道我们要搜索的对象是 Emacs 命令，而不是变量或者函数，我们可以使用 command-apropos （M-? a ）来缩小搜索范围。命令和其他 Lisp 函数的区别是命令特别用于交互式的触发，也就是说可以通过按键或者 M-x 触发。非命令的函数只能被其他 Lisp 代码调用或者被类似 eval-epression 和 eval-last-sexp 这样的命令来执行。我们将会在下一章看到更多的函数和命令的知识。

第二章　简单的新命令

在本章：

- 游历窗口
- 逐行滚动
- 其他光标和文本移动命令
- 处理符号链接
- 修饰 Buffer 切换
- 补充：原始的前置参数

本章中我们将会着手写一些很小的 Lisp 函数和命令，介绍很多的概念来帮助我们面对后面章节中将出现的更大的任务。

2.1　游历窗口

当我最初使用 Emacs 时，我对于`C-x o`很满意，也就是 `other-window` 。它将光标从一个窗口移动到另一个。如果我想把光标移动回前一个，我必须使用-1作为参数执行 `other-window` ，这需要输入`C-u -1 C-x o`，这太繁琐了。而同样繁琐的另一种方案是不停`C-x o`直到我逛遍所有窗口最终回到前一个窗口。

我真正需要的是用一个按键绑定表示“下一个窗口”以及另一个表示“前一个窗口”。我知道我可以编写一些 Emacs Lisp 代码将我需要的方法绑定到新的按键上。首先我必须选择这些按键。“Next”和“Previous”自然可以想到`C-n`和`C-p`，但是这些已经被绑定到了 `next-line` 和 `previous-line` 而我并不想修改它们。另一个选择是使用一些前置按键，后面跟着`C-n`和`C-p`。Emacs 已经使用`C-x`作为很多两键命令的前置键（就像`C-x o`自己），所以我选择`C-x C-n`对应“下一个窗口”而`C-x C-p`对应“前一个窗口”。

我使用帮助命令 `describe-key` [1]来查看`C-x C-n`和`C-x C-p`是否已经绑定到其他按键了。我发现`C-x C-p`已经绑定到了 `set-goal-column` ，而`C-x C-p`绑定到了 `mark-page` 。将他们绑定到“下一个窗口”和“上一个窗口”将会覆盖他

[1]如果你像第一章中描述的那样修改了 `help-command` 的绑定，那么 `describe-key` 的按键绑定是 M-? k；否则是 C-h k。

们默认的绑定。而因为这并不是我经常使用的命令，所以我并不介意覆盖他们。如果我需要的话可以使用M-x来触发他们。

在决定了使用C-x C-n表示“下一个窗口”之后，我需要将它绑定到一些触发“下一个窗口”的命令上。而下一个窗口实际上和C-x o所执行的跳到另一个窗口的行为一样，也就是 other-window 。所以C-x C-n的按键绑定很简单。将下面的命令

```
(global-set-key "\C-x\C-n" 'other-window)
```

写入到.emacs 中就完成了。而定义C-x C-p绑定的命令就要动一点脑子了。Emacs 中并不存在一个命令表示“将光标移动到上一个窗口”。是时候定义一个了！

2.1.1 定义 other-window-backward

既然知道了给 other-window 传递一个参数-1 可以使光标移动到上一个窗口，那么我们可以定义一个新的命令 other-window-backward ，如下所示：

```
(defun other-window-backward ()
  "Select the previous window."
  (interactive)
  (other-window -1))
```

让我们看一下这个函数定义的各个部分。

1. Lisp 函数的定义以 defun 开始。
2. 下面跟着要定义的函数名称；这里使用 other-window-backward 。
3. 下面跟着函数的参数列表[2]。这个函数没有参数，所以我们使用了一个空列表。
4. 字符串” Select the previous window.” 是这个新函数的文档字符串，或者叫做 docstring。任何 Lisp 函数定义都可以有一个文档字符串。Emacs 将会在使用命令 describe-function (M-? f) 或者 apropos 展示在线帮助时显示这个字符串。
5. 下一行（interactive）很特殊。这表示这个函数是一个交互式命令。在 Emacs 里，命令表示一个可以交互执行的 Lisp 函数，这表示它可以通过按键绑定或者通过M-x command-name 来进行触发。并不是所有 Lisp 函数

[2] “parameter” 与 “argument” 有什么不同呢？这两个概念通常可以替换使用，但是技术上来讲，“parameter” 是指函数定义中的形参，而 “argument” 是指函数调用时传入的实参。argument 的值会传递给 parameter。

都是命令，但所有命令都是 Lisp 函数。任何 Lisp 函数，包括交互命令，可以被其他 Lisp 代码使用（`function arg ...`）语法来进行调用。函数通过在函数定义的头部（在可选的 docstring 之后）使用特殊的（`interactive`）表达式来表示自己是交互命令。更多信息在之后的“交互声明”中做更多叙述。

6. 跟在函数名，参数列表，文档字符串，以及 `interactive` 声明之后的是函数体，也就是一个 Lisp 表达式序列。这个函数的函数体是一个单独的表达式（`other-window -1`），也就是使用参数-1 调用函数 `other-window` 。

执行 `defun` 表达式用来定义函数。现在我们可以在 Lisp 程序中通过（`other-window-backward`）来调用它；或者通过输入`M-x other-window-backward RET`来调用它；也可以通过`M-? f other-window-backward RET`[3]来查看帮助。现在我们唯一需要做的就是绑定：

```
(global-set-key "\C-x\C-p" 'other-window-backward)
```

2.1.2　为 other-window-backward 添加参数

这个按键绑定已经能满足我们的需求了，但是我们还需要进行一点点改进。当使用`C-x o`（或者我们现在可以使用`C-x C-n`）来调用 `other-window` 时，你可以使用一个数字 `n` 作为参数来改变它的行为。如果使用了 `n` ，`other-window` 可以跳过很多窗口。例如，`C-u 2 C-x C-n`表示“移动到当前窗口后面的第二个窗口”。就像我们已经看到的，`n` 可以是一个负数来向回跳 `n` 个窗口。因此给 `other-window-backward` 添加一个参数来跳过窗口是很自然的想法。而现在，`other-window-backward` 只能每次向后跳一次。

因此，我们需要给这个函数一个参数：要跳过的窗口数。我们可以这么做：

```
(defun other-window-backward (n)
  "Select Nth previous window."
  (interactive "p")
  (other-window (- n)))
```

我们给自己的函数一个参数 `n` 。我们还把交互声明修改为（`interactive "p"`），还把传递给 `other-window` 的参数从 `-1` 改为（`- n`）。让我们从交互声明开始看一下这些改动。

就像我们所看到的，交互命令是一种 Lisp 函数。这意味着命令也可以有参数。从 Lisp 中向函数传递参数是简单的；只要函数调用时写下来就可以了，就

[3]再一次，如果你已经把 help-command 的绑定到 M-? 那么就是 M-? f。从这开始，我将假设你修改过了，或者你至少应该理解我的做法。

像（other-window -1）。但是如果函数是通过交互命令触发的呢？参数怎么传递？这也就是交互声明的目的。

interactive 的参数描述了这个函数如何获取参数。当命令不需要参数时，那么 interactive 也没有参数，就像我们一开始 other-window-backward 中所示的那样。当命令需要参数时，interactive 也有了一个参数：一个字母构成的字符串，每个字母描述一个参数。例子中的字母 p 表示“如果有前置参数，将它解释为一个数字，如果没有前置参数，就将参数默认设为 1。”[4]在命令触发时参数 n 将接收这个值。所以如果用户输入C-u 7 C-x C-p，n 就是 7 。如果输入C-x C-p，则 n 是 1 。当然你也可以在 Lisp 代码中调用 other-window-backward ，例如（other-window-backward 4）。

新版本的 other-window-backward 使用参数（- n）来调用 other-window 。这里将 n 传递给函数-来得到相反数（注意-和 n 之间的空格）。- 通常表示减—例如（- 5 2）得到 3 —但是当只有一个参数时，他表示取负。

默认情况下，n 是 1，（- n）就是-1，对于 other-window 的调用就变成了（other-window -1）—同函数的第一个版本一样。如果用户指定了一个数字前缀参数—例如C-u 3 C-x C-p—那么我们调用的就是（other-window -3），也就是向前移动 3 个窗口，这正是我们需要的。

理解（- n）和-1 的区别很重要。前者是一个函数调用。函数名和参数之间必须有一个空格。后者是一个整数常量。负号和 1 之间并没有空格。当然你也可以将它写成（- 1）（虽然没有必要在能直接写成-1 的情况下而触发一次函数调用）。不能写成-n，因为 n 不是一个常量。

2.1.3　可选参数

我们还可以对 other-window-backward 做出另一个改进，即当调用函数的时候参数 n 是可选的，也就是当交互触发的时候前置参数也是可选的。它应该能够在不提供参数（other-window-backward）时触发默认行为（即（other-window-backward 1））。就像这样实现：

```
(defun other-window-backward (&optional n)
  "Select Nth previous window."
  (interactive "p")
  (if n
      (other-window (- n))                ; 如果n非空
    (other-window -1)))                   ; 如果n为空
```

参数中的关键词 &optional 表示所有后续的参数都是可选的。可选参数可能会也可能不会传递给函数。如果没给，可选参数的值为 nil 。

[4]要查看 interactive 的 code letter，按下 M-? f interactive RET。

关于符号 nil 有三点需要注意：

- 它表示错误。在 Lisp 的判断结构中— if 、cond 、while 、and 、or 以及 not — nil 表示“false ”，其他值表示“true ”。因此，在表达式

```
(if n
  (other-window (- n))
  (other-window -1))
```

(Lisp 版本的 if-then-else 结构) 中，第一个 n 被求值。如果 n 的值是 true（非空），那么

```
(other-window (- n))
```

被执行，否则

```
(other-window -1)
```

被执行。还有另一个符号，t，代表 truth，但是这没有 nil 重要，就像后面表明的。

- 它和空表很难区分。在 Lisp 解释器中，符号 nil 和空表 () 是相同的对象。如果你调用 listp 来判断符号 nil 是否是一个表，你将会得到结果 t ，也就是 truth。同样的，如果你使用 symbolp 来判断空表是否是一个符号，那么也会得到 t 。但是，如果你传递任何其他列表给 symbolp ，或者传递其他符号给 listp ，那么你会得到 nil —即表示非。
- 它的值就是它自身。当你计算符号 nil 时，结果是 nil 。因此，不像其他的符号，当你需要它的名称而不是它的值得时候，nil 不需要引用，因为它的名称就是它的值。所以你可以这样写：

```
(setq x nil) ; 将nil赋给变量x
```

将 nil 赋给变量 x 而不必这样写：

```
(setq x 'nil)
```

虽然这两种写法都可以。同样的，不要试图将任何新的值赋给 nil，[5]虽然它看起来是一个合法的变量名称。

nil 的另一个功能就是区分列表是否正确。这将在第六章中讨论。

另一个符号 t 用来表示正确。就像 nil ，t 也表示着自身的值，因此不需要引用。与 nil 不同的是，t 并没有跟其他什么对象相同。也与 nil 不同的是，

[5]实际上 Emacs 也不允许你把任何值赋给 nil 。

nil 是唯一表示错误的方式，而其他所有 Lisp 值都和 t 一样表示正确。但是，当你仅仅想表示正确时（就像 symbolp 的返回值）你不需要选择一个类似 17 或者“plugin ”这样的值来表示它。

2.1.4 简化代码

就像前面提到的，表达式

```
(if n                                   ; 如果...
    (other-window (- n))                ; ...那么
  (other-window -1))                    ; ...否则
```

是 Lisp 版本的 if-then-else 结构。if 的第一个参数是一个条件。它将被检测结果是真（除 nil 之外的一切值）还是假（nil ）。如果为真，则第二个参数—“then ”分句将会被执行。如果是假，第三个参数—“else ”分句（可选的）—将会被执行。if 的返回值总是最后执行的表达式的结果。附录B会向你展示 if 和其他像 cond 和 while 这样的 Lisp 流程控制函数。

在本例中,我们可以通过提取公有表达式的方式来进行简化。注意到 other-window 在 if 的两个分支中都被调用了。唯一的区别来自传递给 other-window 的参数 n 。因此我们可以将表达式重写:

```
(other-window (if n (- n) -1))
```

通常，

```
(if test
  (a b)
  (a c))
```

可以简写成 (a (if test b c)) 。

我们还观察到在 if 的两个分支上，我们都在取反—不管是 n 的负数还是 1 的负数。所以

```
(if n (-n) -1)
```

可以变为

```
(- (if n n 1))
```

2.1.5　逻辑表达式

另一个 Lisp 程序员的常用技巧甚至可以使这个表达式更简单：(if n n 1) ≡ (or n 1)。

函数 or 跟大多数语言中的逻辑或都一样：如果所有条件为否，则返回否，否则返回是。但是 Lisp 的 or 还有另一个用途：它挨个计算它的参数的值直到找到第一个为真的值并返回。如果没找到，则返回 nil 。所以 or 的返回值并不仅仅是 false 或者 true ，它返回 false 或者表中的第一个为 true 的值。这意味着通常来说，

```
(if a a b)
```

可以替换为

```
(or a b)
```

实际上，通常我们都应该这么写，因为如果 a 是 true ，那么 (if a a b) 会执行两次 a 而 (or a b) 只执行一次。(另一方面，如果你就是想 a 执行两次，那么当然你应该使用 if)。实际上，

```
(if a a ; 如果a为true，返回a
  (if b b ; else if b为true，返回b
    ...
    (if y y z))) ; else if y为true，返回y，否则z
```

（虽然这看上去很夸张但在真正的程序里这是很常见的一种模式）可以转换成下面这种形式。

```
(or a b .. y z)
```

同样的，

```
(if a
  (if b
    ...
    (if y z)))
```

（注意这个例子中没有任何 else）可以被写成

```
(and a b ... y z)
```

因为 and 通过计算每个参数直到遇到一个值为 nil 的参数。如果找到了，就返回 nil，否则它返回最后一个参数的值。

另一个简写需要注意：一些程序员喜欢将

```
(if (and a b ... y) z)
```

转换成

```
(and a b ... y z)
```

我不这么做，因为虽然他们功能上相同，但是前一个有一个细微的暗示—即“如果 a-y 都是 `true` 的话就执行 z”—后一种却不是这样，这可以让人更加容易理解代码。

2.1.6　最好的 other-window-backward

回到 `other-window-backward` 。使用我们自己整理过的 `other-window` 调用，现在函数的定义看起来是这样的：

```
(defun other-window-backward (&optional n)
  "Select Nth previous window."
  (interactive "p")
  (other-window (- (or n 1))))
```

但是最好的定义—最有 Emacs Lisp 风格的—应该是这样：

```
(defun other-window-backward (&optional n)
  "Select Nth previous window."
  (interactive "P")
  (other-window (- (prefix-numeric-value n))))
```

在这个版本中，交互声明中的字母并不是小写的 `p` 了，而是大写的 `P` ；而 `other-window` 的参数变成了 `(- (prefix-numeric-value n))` ，而不是 `(- (or n 1))` 。

大写的 P 表示“当以交互的方式调用时，将前置参数保持为原始形式（raw form）并将其赋值给 `n` ”。前置参数的原始形式是 Emacs 使用的一种内部数据结

构，用于在触发命令之前记录用户提供的前置信息。(查看补充：原始的前置参数得到更多关于原始前置参数数据结构的细节。) 函数 prefix-numeric-value 可以将像 (interactive "p") 那样将数据结构转换为一个数字。而且，如果 other-window-backward 以非交互的方式调用（因此 n 就不再是一个原始形式的前置参数)，prefix-numeric-value 还是会做正确的事情—也就是说，如果 n 是数字则直接返回 n ，如果 n 为 nil 则返回 1 。

可以说，这个定义并不比我们前面定义的 other-window-backward 的功能更强大。但是这个版本更 “Emacs-Lisp-like”，因为它的代码重用性更好。它使用内建的函数 prefix-numeric-value 而不是重复定义函数的行为。

现在，让我们看看另一个例子。

2.2　逐行滚动

在我使用 Emacs 之前，我习惯了一些编辑器上存在而 Emacs 上并没有的特性。自然我很怀念这些功能并且决定找回他们。这其中的一个例子是使用一个键来向上、向下滚屏。

Emacs 有两个滚屏方法，scroll-up 和 scroll-down ，分别绑定到C-v和M-v。每个方法都有一个可选参数来告诉它要滚动多少行。默认的，他们每次翻一屏。(不要把向上、向下滚屏和通过C-n/C-p向上、向下移动光标混淆。)

虽然我可以使用C-u 1 C-v和C-u 1 M-v来每次向上、向下滚动一行，我还是希望只使用一次按键就实现这一功能。使用前面章节所讲述的技术，这很容易实现。

虽然在这之前，我还是要先考虑一件事。我永远也分不清这两个函数实际上分别是干什么的。scroll-up 是不是将文本向上移动，展示出下面的一部分文件? 或者它表示展示上面的一部分文件，而把所有文字下移? 我希望这些方法的名称能够少一些混淆，就像 scroll-ahead 和 scrll-behind 。

我们可以使用 defalias 来指向任意 Lisp 函数。

```
(defalias 'scroll-ahead 'scroll-up)
(defalias 'scroll-behind 'scroll-down)
```

这样就好多了。现在我们就再也不用为这些混淆的名字而头痛了（虽然原来的名字仍然还在)。

现在我们来定义两个函数来使用正确的参数调用 scroll-ahead 和 scroll-behind 。这个过程和之前定义 other-window-backward 一样：

```
(defun scroll-one-line-ahead ()
  "Scroll ahead one line."
```

```
  (interactive)
  (scroll-ahead 1))

(defun scroll-one-line-behind ()
  "Scroll behind one line."
  (interactive)
  (scroll-behind 1))
```

同样，我们可以给他们一个可选参数来使函数更通用：

```
(defun scroll-n-lines-ahead (&optional n)
  "Scroll ahead N lines (1 by default)."
  (interactive "P")
  (scroll-ahead (prefix-numeric-value n)))

(defun scroll-n-lines-behind (&optional n)
  "Scroll behind N lines (1 by default)."
  (interactive "P"))
```

最后，我们需要选择按键来绑定新的命令。我喜欢C-q绑定 scroll-n-lines-behind 而C-z绑定 scroll-n-lines-ahead ：

```
(global-set-key "\C-q" 'scroll-n-lines-behind)
(global-set-key "\C-z" 'scroll-n-lines-ahead)
```

默认的，C-q绑定到了 quoted-insert 。我将这条不常用的函数移动到了C-x C-q：

```
(global-set-key "\C-x\C-q" 'quoted-insert)
```

C-x C-q的默认绑定是 vc-toggle-read-only ，我并不关心它的丢失。

C-z的在 X 系统下默认绑定是 iconify-or-deiconify-frame ，在终端的绑定是 suspend-emacs 。在这两种情况下，函数也绑定到了C-x C-z，所以也没有必要重新绑定他们。

2.3　其他光标和文本移动命令

下面是另外一些绑定到合理键位的简单命令。

```
(defun point-to-top ()
  "Put point on top line of window."
  (interactive)
  (move-to-window-line 0))

(global-set-key "\M-," 'point-to-top)
```

”Point” 指代光标的位置。这个命令将光标移动到窗口的左上角。推荐的按键绑定替换了 tags-loop-continue ，我把它替换到了C-x,:

```
(global-set-key "\C-x," 'tags-loop-continue)
```

下一个函数将光标移动到了窗口的左下角。

```
(defun point-to-bottom ()
  "Put point at beginning of last visible line."
  (interactive)
  (move-to-window-line -1))

(global-set-key "\M-." 'point-to-bottom)
```

这次的按键绑定替换了 find-tag 。我将它放到了C-x.，这回替换了我并不关心的 set-fill-prefix 。

```
(defun line-to-top ()
  "Move current line to top of window."
  (interactive)
  (recenter 0))

(global-set-key "\M-!" 'line-to-top)
```

这条命令将光标所在的行移动到屏幕的最顶端。这条命令替换了 shell-command 。

改变 Emacs 的按键绑定有一个缺点。当你习惯了自己高度定制化的 Emacs 后再在另一个没有这些定制的 Emacs 上工作时（例如在不同的电脑上或者使用了朋友的账号登录），你会很不习惯。这经常困扰着我。我训练着自己在未定制的 Emacs 上工作而不会受太多影响。我很少使用未定制的 Emacs，所以总的来说得大于失。当你疯狂的更改按键绑定之前，你需要权衡一下这些得失。

2.4　处理符号链接

目前为止，我们写的函数都非常简单。本质上，他们都只是重新排列了一下参数来调用其他已经存在的函数。现在让我们看看需要我们更多编程工作的示例。

在 UNIX 里，符号链接（symbol link，或者 symlink）是一个指向另一个文件的文件。当你查看符号链接的内容时，你实际上得到的是它所指向的文件的内容。

假设你在 Emacs 里访问了一个指向其他文件的符号链接。你修改了一下文件内容然后按下C-x C-s来保存 buffer。Emacs 应该做什么呢?

1. 使用编辑的文件替换符号链接，破坏链接，所指向的原始文件保持不变。
2. 覆盖符号链接所指向的文件。
3. 提示用户来选择上面的方案。
4. 其他。

不同的编辑器处理符号链接的方式都不一样，所以习惯一个编辑器的用户可能会对其他编辑器的行为感到不适应。而且，我相信情况不同正确的处理方式也不同，而用户每次遇到这种情况都被迫需要考虑一下。

我的做法是：当我访问一个符号链接文件时，我让 Emacs 自动的将 buffer 变为只读。当我想要修改时会导致一个 “Buffer is read-only” 的错误。这个错误提示我可能正在访问一个符号链接。然后我会选择使用我自己设计的两个特殊命令之一来处理。

2.4.1　钩子

当我希望 Emacs 在我访问某个文件时将其对应的 buffer 变为只读，我必须告诉 Emacs “当我访问这个文件时执行一段特定的 Lisp 代码”。访问文件的动作应该触发一段我写的代码。这时钩子（hooks）就出场了。

钩子是指在特定情况下执行的指向某个函数列表的 Lisp 变量。例如，变量 `write-file-hooks` 是当一个 buffer 保存时 Emacs 执行的函数列表，而 `post-command-hook` 是当执行一个交互命令时执行的函数列表。在本例中我们最感兴趣的钩子是 `find-file-hooks` ，这在当 Emacs 访问一个新文件时会被执行。（有许多钩子，有一些我们将会在后面的内容中看到。要查看所有钩子，可以使用M-x apropos RET hook RET。）

函数 `add-hook` 将一个函数添加到钩子变量上。下面的函数将被添加到 `find-file-hooks` ：

```
(defun read-only-if-symlink ()
  (if (file-symlink-p buffer-file-name)
```

```
    (progn
      (setq buffer-read-only t)
      (message "File is a symlink"))))
```

这个函数用来检测当前 buffer 的文件是否是符号链接。如果是，则 buffer 将变为只读并且显示“File is a symlink”。让我们仔细看一下这个函数。

- 首先，注意参数列表是空的。钩子变量中的函数都没有参数。
- 函数 `file-symlink-p` 用来检测它的参数，也就是 buffer 的文件名称是否是一个符号链接。它是一个断言（predicate），这表示它会返回 `true` 或者 `false` 。在 Lisp 中，断言通常被以 p 或者-p 结尾。
- `file-symlink-p` 的参数是 `buffer-file-name` 。这个预置的变量在每个 buffer 中都有不同的值，因此也称为 buffer 局部变量。它总是保存着当前 buffer 的名字。在这里，当前 buffer 是指 `find-file-hooks` 执行时找到的文件。
- 如果 `buffer-file-name` 指向的是符号链接，我们希望做两件事：将 buffer 变为只读，并且提示一条信息。但是，Lisp 在 if-then-else 中的“then”部分只允许一条表达式。如果我们写成：

  ```
  (if (file-symlink-p buffer-file-name)
    (setq buffer-read-only t)
    (message "File is a symlink"))
  ```

  这表示，“如果 `buffer-file-name` 是符号链接，那么就把 buffer 变成只读的，否则打印信息‘File is a symlink.’”要想两条语句都执行，我们可以把他们放到 `progn` 里，就像下面这样：

  ```
  (progn
    (setq buffer-read-only t)
    (message "File is a symlink"))
  ```

  `progn` 表达式会顺序执行内部的表达式并且返回最后执行的语句的值。
- 变量 `buffer-read-only` 也是 buffer 局部变量，用于控制当前 buffer 是否是只读的。

既然我们已经定义了 `read-only-if-symlink` ，我们就可以调用

```
(add-hook 'find-file-hooks 'read-only-if-symlink)
```

来将其添加到访问新文件就会触发的函数列表中。

2.4.2　匿名函数

当你使用 defun 定义函数的时候，你给了函数一个可以在任何地方调用的名字。但是对于那些并不需要在任何地方都被调用的函数呢？假如它只需要在一个地方生效呢？可以说，read-only-if-symlink 仅需要在 find-file-hooks 的列表里执行；实际上，在 find-file-hooks 之外的地方调用它甚至并不是什么好事。

我们可以在不指定名称的情况下定义函数。这种函数被称为匿名函数。我们使用 Lisp 的关键词 lambda [6]来定义，除了不指定函数名外，它的作用跟 defun 一模一样。

```
(lambda ()
  (if (file-symlink-p buffer-file-name)
    (progn
      (setq buffer-readonly t)
      (message "File is a symlink"))))
```

lambda 后面的空括号是匿名函数的参数列表。这个函数没有参数。匿名函数可以用在任何你使用函数名的地方：

```
(add-hook 'find-file-hooks
          '(lambda ()
             (if (file-symlink-p buffer-file-name)
               (progn
                 (setq buffer-read-only t)
                 (message "File is a symlink")))))
```

这样就只有 add-hook 可以访问它了。[7]

不过也有一个不应该在钩子中使用匿名函数的原因。如果你想要从钩子中移除一个函数的话，你需要使用函数名来调用 remove-hook ，就像这样：

```
(remove-hook 'find-file-hooks 'read-only-if-symlink)
```

而如果使用匿名函数就没法这样做了。

[6] “lambda 演算”是一套用于研究函数及其参数的数学形式。某种意义上来说它是 Lisp（以及其他很多语言）的理论基础。单词“lambda”只是一个希腊语中的单词，并没有什么特殊的含义。

[7]这并不是绝对正确的。其他的代码可以搜索 find-file-hooks 列表的内容并且执行里面的所有函数。这里的意思是这个函数相对于 defun 的显式声明来说隐藏起来了。

2.4.3　处理符号链接

当 Emacs 提醒我在编辑符号链接时，我可能希望打开链接的目标文件来作为当前 buffer 的内容；我也可能希望”clobber” 符号链接（将符号链接文件替换为所指向的真实文件）然后再访问它。下面是这两个的实现方式：

```
(defun visit-target-instead ()
  "Replace this buffer with a buffer visiting the link target."
  (interactive)
  (if buffer-file-name
    (let ((target (file-symlink-p buffer-file-name)))
      (if target
        (find-alternate-file target)
        (error "Not visiting a symlink")))
    (error "Not visiting a file")))

(defun clobber-symlink ()
  "Replace symlink with a copy of the file."
  (interactive)
  (if buffer-file-name
    (let ((target (file-symlink-p buffer-file-name)))
      (if target
        (if (yes-or-no-p (format "Replace %s with %s?"
                                 buffer-file-name
                                 target))
          (progn
            (delete-file buffer-file-name)
            (write-file buffer-file-name)))
        (error "Not visiting a symlink")))
    (error "Not visiting a file")))
```

两个函数都以下面的表达式开始：

```
(if buffer-file-name
  ...
  (error "Not visiting a file"))
```

（我将其他内容省略掉以强调这个 `if` 结构。）因为 `buffer-file-name` 可能为空（当前 buffer 可能没有访问任何文件—例如，*scratch* buffer），所以这是必

要的，而传递 nil 给 file-symlink-p 将会触发错误，“Wrong type argument: stringp,nil”。[8]这个错误表示一个函数的参数应该是字符串——一个符合 stringp 断言的对象—但是却得到了 nil 。visit-target-instead 和 clobber-symlink 都会触发这个错误信息，所以我们自己来检测 buffer-file-name 是不是 nil 。如果是 nil ，那么“else”子句里我们会使用 error 函数生成一个可读性更好的错误信息—“Not visiting a file”。当 error 函数被调用时，当前的命令会被终止，Emacs 将会返回到它的最顶层来等待用户的下一个输入。

为什么 read-only-if-symlink 中不需要检测 buffer-file-name 是否为空呢？因为这个方法只会由 find-file-hooks 调用，而这个钩子只有当访问某个文件时才会触发。

在 buffer-file-name 条件的“then”部分，两个函数都有下面的结构

```
(let ((target (file-symlink-p buffer-file-name))) ...)
```

大多数语言都有方法来创建临时变量（也称为局部变量），它们只存在于某个特定的代码域中，称为变量的作用域。在 Lisp 中，临时变量使用 let 来创建，结构是这样的：

```
(let ((var1 value1)
      (var2 value2)
      ...
      (varn valuen))
  body1 body2 ... bodyn)
```

这会将 value1 赋值给 var1，value2 赋值给 var2，依此类推；var1 和 var2 只能在 bodyi 表达式中使用。此外，使用临时变量能够帮助避免不同域的代码中出现函数名相同的冲突。

所以表达式

```
(let ((target (file-symlink-p buffer-file-name))) ...)
```

创建了一个名为 target 的临时变量，它的值是 (file-symlink-p buffer-file-name) 的返回值。

就像前面提到的，file-symlink-p 是一个断言，也就是说它的返回值是真或者假。但是因为真在 Lisp 中可以被任何除 nil 之外的值表示，如果 file-symlink-p 的参数是一个符号链接时它的返回值并不一定就是 t 。实际上，它会返回符号链

[8]请自己试一下：M-: (file-symlink-p nil) RET。

接所指向的文件名。所以如果 buffer-file-name 是符号链接的名字，target 将会是符号链接的目标的名称。

在临时变量 target 的作用域中，let 的 body 都是这样的：

```
(if target
  ...
  (error "Not visiting a symlink"))
```

在执行完 let 的 body 之后，变量 target 就不存在了。

在 let 中，如果 target 为空（file-symlink-p 可能会返回 nil ，因为 buffer-file-name 可能并不是一个符号链接），那么我们就会在 “else” 里产生一个错误信息，“Not visiting a symlink”。否则每个函数中会执行自己的逻辑。最后我们来看两个函数不一样的地方。

函数 visit-target-instead 中执行

```
(find-alternate-file target)
```

这会访问 target 文件来替换当前的 buffer，并且会提示用户，以免原 buffer 还有未保存的修改。它甚至会触发 find-file-hooks ，因为新文件也可能是一个符号链接！

在 visit-target-instead 调用 find-alternate-file 的地方，clobber-symlink 则如下所示：

```
(if (yes-or-no-p ...) ...)
```

函数 yes-or-no-p 会询问用户一个问题，并会根据用户的选择返回 true 或 false 。本例中，问题是：

```
(format "Replace %s with %s?"
        buffer-file-name
        target)
```

这个字符串的结构和 C 语言的 printf 很相似。第一个参数是一个格式化模式字符串。每个%s 都使用后面的字符串参数来替换。第一个%s 使用 buffer-file-name 的值替换，第二个使用 target 的值替换。所以如果 buffer-file-name 的值是 “foo ”而 target 的值是“bar ”，那么提示就会是“Replace foo with bar?”（format 函数还支持其他的格式化符号。例如，如果参数是 ASCII 值则%c 会打印出一个字母。使用M-? f format RET来查看整个功能列表。）

在检查了 yes-or-no-p 的返回值并且用户选择了“yes”之后，clobber-symlink 将会执行：

```
(progn
  (delete-file buffer-file-name)
  (write-file buffer-file-name))
```

我们已经知道，progn 会把多条 Lisp 表达式组合起来。delete-file 会删除文件（只是个符号链接），write-file 会将当前 buffer 的内容保存到 buffer-file-name 所指向的位置，只是这次保存的是普通文件。

我喜欢将C-x t绑定到 visit-target-instead（默认未被使用）而C-x 1绑定到 clobber-symlink （默认绑定到 count-linespage ）。

2.5　修饰 Buffer 切换

让我们以一个例子总结本章，这个例子将会引入一个称为修饰（advice）的非常有用的 Lisp 工具。

我发现我经常同时编辑许多名称相似的文件；例如，foobar.c 和 foobar.h。当我想从一个 buffer 切换到另一个时，我使用C-x b，也就是 switch-to-buffer ，它会询问我 buffer 的名称。因为我希望尽量少的按键，我使用TAB来补全 buffer 名称。我会输入

```
C-x b fo TAB
```

并且希望 TAB 会将 “fo” 补全为” foobar.c”，然后我只要按下 RET 就可以了。90% 的情况下这工作的很好。另外的情况下，就像这个例子中，按下 fo TAB 将只会补全为 “foobar.”，而让我自己区分是选择” foobar.c” 还是” foobar.h”。出于习惯，我常常按下 RET，结果 buffer 的名称变成了” foobar.”。

这时，Emacs 将会创建一个新的名为 foobar. 的新 buffer，当然这完全不是我想要的。现在我需要杀掉这个新 buffer（使用C-x k，kill-buffer）然后再来一次。虽然我有时也需要新建一个不存在的 buffer，但是这和刚刚这种错误的情况相比很少见。我希望在这种情况中，Emacs 能够在我出错之前提示我。

要达到这点，我们可以使用 advice。advice 是指一段在函数调用之前或之后执行的代码。前置修饰可以在参数传递给函数之前对其进行修改。后置修饰可以修改函数的返回值。修饰跟钩子变量有点像，只是 Emacs 只为一些特定的情况定义了不多的一些钩子，而你却能选择对哪些方法进行修饰。

下面是修饰的第一次尝试：

```
(defadvice switch-to-buffer (before existing-buffer
                                    activate compile)
  "When interactive, switch to existing buffers only."
  (interactive "b"))
```

让我们仔细看看它。函数 defadvice 用于创建一个新的修饰。它的第一个参数是要被修饰的函数名(不必引用,unquoted)—在本例中也就是 switch-to-buffer 。后面跟着的是特定格式的列表。它的第一个元素—在本例中也就是 before —告诉我们这是前置还是后置修饰。(还有一种修饰，称为 “around ”，它能让你在修饰函数的内部调用被修饰的方法。) 后面跟着的是这个修饰的名称；本例中是 existing-buffer 。以后如果你想删除或者修改这个修饰你可以使用这个名称。再后面是一些关键词：activate 表示这个修饰在其定义之后马上生效（可以只是定义修饰而不生效)；compile 表示这个修饰的代码应该被 “byte-compiled ” 提高执行速度（查看第五章)。

在特定格式的列表之后，跟着一个可选的文档字符串。

本例中的 body 只有一行交互声明，这会替换 switch-to-buffer 的交互声明。switch-to-buffer 接受任何字符串作为 buffer-name 参数，而交互声明中的字符 b 表示 “只接受已存在的 buffer 的名称”。我们在不影响任何以非交互形式调用 switch-to-buffer 的情况下做出了这个更改。所以这个修饰高效的完成了整件工作：它使 switch-to-buffer 只接受已存在的 buffer 名。

不幸的是，这样约束性太大了。还是应该能够切换到不存在的 buffer，但是只在某些特殊的条件下才移除这个限制—例如，当使用前置参数的时候。这样，C-x b将会拒绝切换到不存在的 buffer，而C-u C-x b将允许。

我们可以这么做：

```
(defadvice switch-to-buffer (before existing-buffer
                                    activate compile)
  "When interactive, switch to existing buffers only,
unless given a prefix argument."
  (interactive
    (list (read-buffer "Switch to buffer:"
                       (other-buffer)
                       (null current-prefix-arg)))))
```

又一次，我们使用了前置修饰修改了 switch-to-buffer 的交互声明。但是这次，我们使用了一种未见过的形式调用 interactive ： 我们传递了一个列表作为参数给它，而不是一个字母组成的字符串。

当 interactive 的参数不是字符串而是一些表达式时，这些表达式会进行运算得到一个参数列表传递给函数。所以在这个例子中我们调用了 list ，它使用下面这段表达式的返回值构建：

```
(read-buffer "Switch to buffer: "
             (other-buffer)
             (null current-prefix-arg))
```

函数 read-buffer 是一个底层的用于向用户询问 buffer 名称的函数。说它底层是因为所有其他询问 buffer 名称的函数最终调用的都是它。它的调用需要一个提示字符串和两个可选参数：一个默认切换到的 buffer，以及一个布尔值用于标识输入是否只能是已存在的 buffer。

默认的 buffer，我们传递了（other-buffer）的返回值给它，它的作用是产生一个可用的默认 buffer。（通常它会选择最近使用的但是当前不可见的 buffer。）对于是否限制输入的布尔状态值，我们使用了

```
(null current-prefix-arg)
```

这会查看 current-prefix-arg 是否为 nil 。如果是，则返回 t ；否则返回 nil 。因此，如果没有前置参数（也就是 current-prefix-arg 为 nil ），那么我们调用的是

```
(read-buffer "Switch to buffer: "
             (other-buffer)
             t)
```

表示“读入 buffer 名称，只接受已存在的 buffer”。如果有前置参数，那么我们调用的是

```
(read-buffer "Switch to buffer: "
             (other-buffer)
             nil)
```

表示“读入 buffer 名称而不做任何限制”（允许不存在的 buffer 作为参数）。然后 read-buffer 的返回值被传给了 list，list（包含着一个元素，也就是 buffer 名称）传递给 switch-to-buffer 作为参数列表。

switch-to-buffer 这样修饰之后，Emacs 将不会回应我切换到不存在的 buffer 的要求了，除非我按下C-u来要求这种能力。

完整起见，你还应该同样修饰函数 switch-to-buffer-other-window 和 switch-to-buffer-other-frame 。

2.6　补充：原始的前置参数

变量 current-prefix-arg 总是保存着最后的“原始”前置参数，跟你从 (interactive "P") 中取到的一样。

函数 prefix-numeric-value 可以应用到一个跟你从 (interactive "P") 中取得的“原始”前置参数一样类型的值来得到数值。

原始的前置参数什么样子呢？表格2.1展示出了原始值以及对应的数值。

如果用户输入	原始值	数值
C-u 后面跟一个（可能是负数）数字	数字本身	数字本身
C-u -（后面什么都没有）	符号-	-1
C-u n 一行中 n 次	一个包含数字 4 的 n 次方的表	4 的 n 次方
没有前置参数	nil	1

表 2.1: 前置参数

第三章　协作命令

在本章：

- 症状
- 解药
- 归纳出更一般的解决方法

本章将讲述不同命令协同工作，以完成在一个命令里保存信息而在另一个命令里获取它。共享信息的最简单的方式是创建一个变量并把值存进去。本章中我们当然也会这么做。例如，我们会把当前 buffer 中的位置信息存储起来然后在其他命令中使用它。但是我们也会学到一些更复杂的保存状态的方式，特别是标记（markers）和符号属性（symbol properties）。我们将会把这些跟 buffer 和窗口的信息组合起来写出一套允许你“回滚”翻页动作的函数。

3.1　症状

你正在编写一些复杂的 Lisp 代码。你非常专注，在头脑中努力的理清数据之间的微妙的联系然后将它们编写到屏幕上。你正在处理一个非常精妙的部分而这时你发现了左面的一个拼写错误。你希望按下 `C-b C-b C-b` 来回到那里对其修正，但是—灾难发生了—你按下了 `C-v C-v C-v` ，向下翻了三屏，最终停在了离你的代码几光年以外的地方。你试图找到在这个失误发生之前光标在哪里，以及为什么在那里，以及你正在那里做什么，这无疑打乱了你头脑中的思考。当你翻页，或者搜索，或者查找撤销列表来回到那里时，你已经忘了最初想要修正的那个拼写错误，最终这变成了一个 bug，可能会花费你几个小时来找到它。

Emacs 在这个例子中没有起到帮助作用。它使你如此容易的在你的文档中迷失而找到回去的路却很困难。虽然 Emacs 有一个可扩展的撤销工具，但这只能撤销修改。你不能撤销如此简单的浏览行为。

3.2　解药

假设我们可以这样修改 C-v（scroll-up 命令[1]），当我们按下它之后，Emacs 认为，“可能用户是误按的 C-v ，所以我将记录一些‘撤销’的信息以备未来需要”。然后我们可以编写另一个函数，unscroll ，它可以回滚最后一次的滚动。这样只要记住 unscroll 的按键绑定就能避免再次发生这种迷失。

实际上这并没完全解决问题。如果你一次按下了多个 C-v ，调用一次 unscroll 应该将它们都撤销，而不是只撤销最后一次。这表示只有序列中的第一个 C-v 才需要记住它的起始位置。我们怎样来检测这种情况的发生呢？在我们的 C-v 代码里记录开始位置之前，我们需要知道（a）下一个命令是否还是 scroll-up ，或者（b）前一个命令是否不是 scroll-up 。显然（a）是不可能的：我们不能预见未来。幸运的是（b）很简单：Emacs 有一个称为 last-command 的变量专门记录这一信息。这个变量是我们将使用的在不同命令间传递信息的第一个方法。

现在唯一剩下的问题是：我们如何把这个额外的代码关联到 scroll-up ？修饰特性能很好的解决这一问题。前面讲过一段修饰代码可以在被修饰方法之前或之后执行。在本例中我们需要前置修饰，因为只有在执行 scroll-up 之前我们才能知道开始的位置在哪里。

3.2.1　声明变量

我们将以建立一个保存“撤销”信息的全局变量 unscorll-to 开始，它保存了 unscroll 应该将光标移动到的位置。我们将使用 defvar 来声明变量。

```
(defvar unscroll-to nil
  "Text position for next call to 'unscroll'.")
```

全局变量不需要声明。但是使用 defvar 声明变量有一些优势：

- 使用 defvar 能够关联一个文档字符串给变量，就像 defun 那样。
- 可以给变量赋一个默认值。本例中，unscroll-to 的默认值是 nil 。通过 defvar 给变量赋默认值与使用 setq 不一样。使用 setq 将会强制给变量赋默认值，而 defvar 只会在变量没有任何值时才会赋给默认值。为什么这很重要呢？假设你的 .emacs 文件有这么一行：

  ```
  (setq mail-signature t)
  ```

[1] 虽然在第二章，我们使用了 defalias 将 scroll-ahead 和 scroll-behind 替代了 scroll-up 和 scroll-down ，本章我们依然使用它们之前的名称。

这表示当你使用 Emacs 发送 email 时，你希望在最后添加你自己的签名文件。当你启动 Emacs 时，mail-signature 被设置为了 t ，但是定义邮件发送代码的 Lisp 文件，sendmail，还没有被载入。它只有当你第一次调用 mail 命令时才会加载。当你调用 mail 时，Emacs 将执行 sendmail Lisp 文件中的代码：

```
(defvar mail-signature nil ...)
```

这表示 nil 是 mail-signature 的默认初始值。但是你已经给 mail-signature 赋过值了，而你不希望载入 sendmail 时把你的设置给覆盖了。另一方面，如果你在 .emacs 中并没有给 mail-signature 赋过任何值，你还是希望这个值能够生效。

- 使用 defvar 声明的变量可以通过多个标签相关（tag-related）的命令找到。在工程中通过标签找到变量和函数定义是一种快捷的方式。Emacs 的标签工具，例如 find-tag 函数，可以找到任何通过 def... 函数（ defun ，defalias ，defmacro ，defvar ，defsubst ，defconst ，defadvice ）定义的东西。
- 当你编译字节码时（见第五章），编译器如果发现变量未使用 defvar 声明将会产生一个警告。如果你的所有变量都进行了声明，那么你可以使用这些警告来找出哪些变量名写错了。

3.2.2　保存和取出 point

让我们定义一个变量来存储 scroll-up 队列的最初位置，即最初光标在文本中的位置。光标在文本中的位置被称为 point，其值就是从 buffer 开始位置计算（从 1 开始）有多少个字符。point 的值可以由函数 point 得到。

```
(defadvice scroll-up (before remember-for-unscroll
                             activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (eq last-command 'scroll-up))
      (setq unscroll-to (point))))
```

这个修饰是这么工作的：

1. 函数 eq 告诉我们它的两个参数是否相等。本例中，参数是 last-command 变量的值，以及符号 scroll-up 。last-command 的值是最后一次用户触发的命令的符号（在本章后面的部分使用 this-command中也能看到）。

2. eq 的返回值被传递给 not ，这会对它的参数的布尔值取反。如果 nil 传给 not ，那么返回 t 。如果其他值传给 not ，则返回 nil 。[2]
3. 如果 not 的返回值是 t ，即 last-command 的值并不是 scroll-up ，那么变量 unscroll-to 的的值将被设置为当前的 point 值。

现在定义 unscroll 就很简单了：

```
(defun unscroll ()
  "Jump to location specified by 'unscroll-to'."
  (interactive)
  (goto-char unscroll-to))
```

函数 goto-char 将光标移动到指定的位置。

3.2.3　窗口内容

对于这个解决方案有一些不完美的地方。在一次 unscroll 之后，光标确实返回到了正确的地方，但这时屏幕却看起来和按下 C-v 之前不一样了。例如，我在按下 C-v C-v C-v 之前可能正在屏幕的底部编辑一行代码。在我调用 unscroll 之后，虽然光标确实回到了之前的位置，但是那一行可能显示到了窗口的中间。

既然我们的目的是最小化意料之外的滚动所造成的破坏，那么我们不只希望仅仅恢复光标的位置，我们还希望之前编辑的行的位置也恢复到原处。

因此只保存 point 的值就不够了。我们还必须保存一个值来表示当前窗口中显示什么。Emacs 提供了几个函数来描述窗口中显示什么，例如 window-edges ，window-height ，current-window-configuration 。目前我们只需要使用 window-start ，它表示对于给定的窗口，显示的第一个（窗口左上角）字符在 buffer 中的位置。这样我们只需要在命令间传递更多一点信息就可以了。

更新我们的例子很简单。首相我们要将变量 unscroll-to 的声明替换为两个新的变量：一个用于保存 point 的值，另一个用于保存窗口中第一个字符的位置。

```
(defvar unscroll-point nil
  "Cursor position for next call to 'unscroll'.")

(defvar unscroll-window-start nil
  "Window start for next call to 'unscroll'.")
```

[2]如果你认为 not 的行为看起来跟 null 很像，你是对的—它们就是同一个函数。它们其中一个就是另一个的别名。你使用哪个只是一个可读性方面的问题。当要检测一个列表是否为空列表时使用 null 。当要对一个值取反的时候使用 not 。

然后我们要修改 scroll-up 的修饰以及 unscroll 来使用这两个值。

```
(defadvice scroll-up (before remember-for-unscroll
                             activate compile)
  "Remember where we started from, for `unscroll'."
  (if (not (eq last-command 'scroll-up))
      (progn
        (setq unscroll-point (point))
        (setq unscroll-window-start (window-start)))))

(defun unscroll ()
  "Revert to 'unscroll-point' and 'unscroll-window-start'."
  (interactive)
  (goto-char unscroll-point)
  (set-window-start nil unscroll-window-start))
```

修饰的名称仍然是 remember-for-unscroll ，这会替换之前同名的修饰。

函数 set-window-start 和 goto-char 移动光标位置的方式类似，它会设置窗口开始的位置。不一样的是，set-window-start 有两个参数。第一个参数表明操作的是哪个窗口。如果为 nil ，则默认使用当前选中的窗口。（传递给 set-window-start 的窗口对象可以通过类似 get-buffer-window 以及 previous-window 的函数得到。）

对于回滚我们可能还希望保持另一个信息，即窗口的 hscroll ，它保存着窗口横向翻滚的列数，默认为 0 。我们可以添加另一个变量来保存：

```
(defvar unscroll-hscroll nil
  "Hscroll for next call to 'unscroll'.")
```

然后我们再次更新 unscroll 和 scroll-up 修饰来调用 window-hscroll（获取窗口的 hscroll 值）以及 set-window-hscroll （设置）：

```
(defadvice scroll-up (before remember-for-unscroll
                             activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (eq last-command 'scroll-up))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))
```

```
(defun unscroll ()
  "Revert to 'unscroll-point' and 'unscroll-window-start'."
  (interactive)
  (goto-char unscroll-point)
  (set-window-start nil unscroll-window-start)
  (set-window-hscroll nil unscroll-hscroll))
```

注意在这个 scroll-up 修饰的版本中，progn 的使用：

```
(progn
  (setq ...)
  (setq ...))
```

被合并成了一个 setq ，里面包含了多个“变量-值”对。这是一种简化写法，setq 可以包含任意数量的变量。

3.2.4　错误检查

假如用户在调用任何 scroll-up 之前调用了 unscroll 会发生什么呢?变量 unscroll-point ,unscroll-window-start ,以及 unscroll-scroll 将会包含他们的默认值,也就是 nil 。这个值在传递给函数 goto-char ,set-window-start 以及 set-window-scroll 时是不合适的。当 goto-char 被调用时，unscroll 的触发将会返回如下错误：“Wrong type argument: integer-or-maker-p, nil”。这表示一个函数需要接收一个数字或者标记（满足断言 integer-or-marker-p ），而收到的却是 nil 。（标记在本章前面的部分介绍过了。）

为避免用户被这些神秘的错误信息所折磨，在调用 goto-char 之前进行一个简单的检查并且生成一个更可读的错误信息是一个好主意：

```
(if (not unscroll-point) ; 如果unscroll-point的值为nil
    (error "Cannot unscroll yet"))
```

当错误发生时，unscroll 将会被终止并且提示“Cannot unscroll yet”。

3.3　归纳出更一般的解决方法

当我们想要按 C-b 时按到 C-v 是很常见的一种情况。这也就是我们设计 unscroll 函数的原因。现在让我们来研究同样容易发生的想按下 M-b (backward-word

）却按下了 M-v （scroll-down ）。这是同样的问题，但也有点不一样。如果 unscroll 能够回撤任何方向的滚动就好了。

最直接的方法是像修饰 scroll-down 那样修饰 scroll-up ：

```
(defadvice scroll-down (before remember-for-unscroll
                               activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (eq last-command 'scroll-down))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))
```

（注意这两个函数，scroll-up 和 scroll-down ，它们的修饰的名称，也就是 remember-for-unscroll ，可以一样，而且不会冲突。）

现在我们必须决定当错误的 C-v 和错误的 M-v 同时发生时 unscroll 如何运作。换句话说，假设你错误的按下了 C-v C-v M-v 。它是应该恢复到 M-v 之前的位置呢，还是应该恢复到最初的 C-v 之前?

我选择后者。但是这意味着对于 scroll-up （以及 scroll-down ）的修饰，我们需要同时检测 scroll-up 和 scroll-down 。

```
(defadvice scroll-up (before remember-for-unscroll
                             activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (or (eq last-command 'scroll-up)
               (eq last-command 'scroll-down)))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))

(defadvice scroll-down (before remember-for-unscroll
                               activate compile)
  "Remember where we started from, for 'unscroll'.'"
       (if (not (or (eq last-command 'scroll-up)
                    (eq last-command 'scroll-down)))
       (setq unscroll-point (point)
                unscroll-window-start (window-start)
                unscroll-hscroll (window-hscroll))))
```

让我们花一点时间来确保你理解表达式

```
(if (not (or (eq last-command 'scroll-up)
             (eq last-command 'scorll-down)))
    (setq ...))
```

阅读这段表达式最好的方法是一级一级的向里阅读。从这里开始

```
(if (not ...)
    (setq ...))
```

“如果为假，则设置一些变量。”下面更进一步：

```
(if (not (or ...))
    (setq ...))
```

“如果所有条件都为假，则设置一些变量。”最后，

```
(if (not (or (eq last-command 'scroll-up)
             (eq last-command 'scorll-down)))
    (setq ...))
```

表示，“如果 last-command 不是 scroll-up 并且 last-command 不是 scroll-down ，那么设置一些变量。”

假设之后你希望更多的命令也按照这种方式来修饰；例如 scroll-left 和 scroll-right ：

```
(defadvice scroll-up (before remember-for-unscroll
                             activate compile)
  "Remember where we started from, for 'unscroll'. "
  (if (not (or (eq last-command 'scroll-up)
               (eq last-command 'scroll-down)
               (eq last-command 'scroll-left) ;new
               (eq last-command 'scroll-right))) ;new
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))

(defadvice scroll-down (before remember-for-unscroll
```

```
                                activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (or (eq last-command 'scroll-up)
               (eq last-command 'scroll-down)
               (eq last-command 'scroll-left) ;new
               (eq last-command 'scroll-right))) ;new
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))

(defadvice scroll-left (before remember-for-unscroll
                                activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (or (eq last-command 'scroll-up)
               (eq last-command 'scroll-down)
               (eq last-command 'scroll-left)
               (eq last-command 'scroll-right)))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))

(defadvice scroll-right (before remember-for-unscroll
                                 activate compile)
  "Remember where we started from, for 'unscroll'."
  (if (not (or (eq last-command 'scroll-up)
               (eq last-command 'scroll-down)
               (eq last-command 'scroll-left)
               (eq last-command 'scroll-right)))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))
```

这样写不仅繁琐且容易出错，而且对于每个我们需要回撤的新命令，每个之前写过的回撤命令都需要加入对于新的 `last-command` 的检测。

3.3.1　使用 this-command

有两种方法可以改善这种情况。第一种，既然每个修饰都差不多，我们可以把它们提取一下：

```
(defun unscroll-maybe-remember ()
  (if (not (or (eq last-command 'scroll-up)
               (eq last-command 'scroll-down)
               (eq last-command 'scroll-left)
               (eq last-command 'scroll-right)))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))

(defadvice scroll-up (before remember-for-unscroll
                             activate compile)
  "Remember where we started from, for 'unscroll'."
  (unscroll-maybe-remember))

(defadvice scroll-down (before remember-for-unscroll
                               activate compile)
  "'Remember where we started from, for 'unscroll'."
  (unscroll-maybe-remember))

(defadvice scroll-left (before remember-for-unscroll
                               activate compile)
  "Remember where we started from, for 'unscroll'."
  (unscroll-maybe-remember))

(defadvice scroll-right (before remember-for-unscroll
                                activate compile)
  "Remember where we started from, for 'unscroll'."
  (unscroll-maybe-remember))
```

第二种，不去检测 n 种可能的 last-command 值，我们可以使用一个单独的变量来保存每种情况的 last-command 值。

当前用户触发的命令的名称会保存在变量 this-command 中。实际上，last-command 的值是这样得到的：当 Emacs 执行一个命令时，this-command 保存着命令的名字；当执行完成时，Emacs 会将 this-command 的值赋给 last-command 。

当命令执行时，它会改变 this-command 的值。当下个命令执行时，这个值会保存在 last-command 中。

让我们选择一个符号来代表所有可回撤的命令：例如，unscrollable 。现在我们可以修改一下 unscroll-maybe-remeber ：

```
(defun unscroll-maybe-remember ()
  (setq this-command 'unscrollable)
  (if (not (eq last-command 'unscrollable))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))
```

调用这个函数的命令会将 this-command 设置为 unscrollable 。现在我们只需要检测一个变量而不必检测四种不同情况的 last-command （也许还会更多）了。

3.3.2　符号属性

我们改进版的 unscroll-maybe-remeber 工作的非常好，但是（你可能已经预料到了）我们还是可以做一些改进。首先就是变量 this-command 和 last-command 并不是只有我们自己使用。它们对于 Emacs Lisp 解释器很重要，而 Emacs 的其他功能也依赖于这两个值。而我们知道，有一些使用了这些滚动函数的 Emacs 特性并没有修改 this-commandh 和 last-command 。而且，我们更想要一个专有的值来标识所有可回滚的命令。

这里我们引入好用的符号属性（symbol properties）。Emacs Lisp 符号不只用来保存函数定义，它还有一组关联的属性列表。属性列表是一组键值映射。每个名字是一个 Lisp 符号，每个值可以是任意 Lisp 表达式。

属性使用 put 函数来保存，使用 get 函数来读取。因此，如果我们将 17 保存在符号 a-symbol 的 some-property 的属性中：

```
(put 'a-symbol 'some-property 17)
```

那么

```
(get 'a-symbol 'some-property)
```

将返回 17 。如果我们从一个符号读取一个并不存在的属性，则返回 nil 。

我们可以将 unscrollable 作为一个属性，而不是作为一个储存 this-command 和 last-command 的值的变量。我们可以将支持返回的命令的 unscrollable 属性设为 t ：

```
(put 'scroll-up 'unscrollable t)
(put 'scroll-down 'unscrollable t)
```

```
(put 'scroll-left 'unscrollable t)
(put 'scroll-right 'unscrollable t)
```

这只需要在调用 unscroll-maybe-remember 之前执行一次就行了。

现在如果 x 是 scroll-up ，scroll-down ，scroll-left ，scroll-right 之中的一个的话则（get x unscrollable）会返回 t 。对于其他的符号，因为 unscrollable 属性默认未定义，所以结果为 nil 。

现在我们可以将 unscroll-maybe-remember 中的

```
(if (not (eq last-command 'unscrollable)) ...)
```

修改为

```
(if (not (get last-command 'unscrollable)) ...)
```

而且我们还可以停止将 unscrollable 赋值给 this-command ：

```
(defun unscroll-maybe-remember ()
  (if (not (get last-command 'unscrollable))
      (setq unscroll-point (point)
            unscroll-window-start (window-start)
            unscroll-hscroll (window-hscroll))))
```

3.3.3　标记

我们能否将这段代码改的更好呢？假设你不小心按下了几次 scroll-down 然后你想 unscroll 。但是在你这么做之前，你发现了一些你想要修改的代码，然后你进行了修改。然后你再 unscroll 。这时屏幕并没有被正确的恢复！

这是因为编辑 buffer 中前面的文字将会改变所有后面的文字的位置。添加或者删除 n 个字符将会对所有后续的字符位置添加或减少 n 。因此 unscroll-point 和 unscroll-window-start 所保存的值都会被影响 n （如果 n 为 0，那么你很幸运）。

使用标记（marker）而不是 unscroll-point 和 unscroll-window-start 的绝对位置将会是一个很好的选择。标记是一种就像数字一样用来保存 buffer 位置的特殊对象。但是如果由于插入或者删除造成了 buffer 位置的更改，那么标记也会跟着更改。

既然我们要将 unscroll-point 和 unscroll-window-start 修改为标记，我们就不需要将他们初始化为 nil 了。我们可以使用 make-marker 来将它们初始化为空的标记对象：

```
(defvar unscrollfpoint (make-marker)
  "Cursor position for next call to 'unscroll'.")

(defvar unscroll-window-start (make-marker)
  "Window start for next call to ''unscroll'.")
```

函数 set-marker 被用来设置标记的位置。

```
(defun unscroll-maybe-remember ()
  (if (not (get last-command 'unscrollable))
      (progn
        (set-marker unscroll-point (point))
        (set-marker unscroll-window-start (window-start))
        (setq unscroll-hscroll (window-hscroll)))))
```

progn 又回来了，因为 setq 被拆分成了几个不同的函数调用。我们不对 unscroll-hscroll 使用标记，因为它的值并不是 buffer 位置。

我们并不需要重写 unscroll ，因为 goto-char 和 set-window-start 的参数不管是标记还是数字都会很好的工作。所以前面的定义（为了方便这里在贴一次）还是能够工作：

```
(defun unscroll ()
  "Revert to 'unscroll-point' and 'unscroll-window-start'."
  (interactive)
  (goto-char unscroll-point)
  (set-window-start nil unscroll-window-start)
  (set-window-hscroll nil unscroll-hscroll))
```

3.3.4　附录：关于效率

当我们定义 unscroll-point 和 unscroll-marker 时，我们创建了空的符号对象并且在每次调用 unscroll-remember 时复用它们，而不是每次都创建新的并且扔掉旧的。这是一种优化。这并不仅仅是说我们应该尽可能的避免这种对象创建的消耗，更多的是因为标记要比其他变量的消耗更大。每次 buffer 做出修改的时候标记都会跟着修改。弃用的标记最终会被垃圾回收器回收掉，但是直到那时它都会降低编辑 buffer 的速度。

通常，如果你想要弃用一个标记对象 m （即你不需要再使用它的值了），这么做将会是很好的选择：

```
(set-marker m nil)
```

第四章　搜索和修改 Buffers

在本章：

- 插入当前时间
- 记录戳
- 修改戳

很多场景中你会想要在 buffer 中搜索一个字符串，可能还希望用另一个字符串替换它。本章中我们将会为此展示很多有效的方法。我们将会讲解那些执行搜索功能的函数，并且展示如何构建正则表达式，这会给你的搜索带来巨大的灵活性。

4.1　插入当前时间

当你编辑一个文件时插入当前的日期或者时间有时是很有用的。例如，在我写这一章的时候，是 1996 年 8 月 18 日星期五的晚上 10 点半。几天前，我在编辑一个 Emacs Lisp 代码文件时，我将注释

```
;; Each element of ENTRIES has the form
;; (NAME (VALUE-HIGH . VALUE-LOW))
```

修改为

```
;;NTRIES has the form
;; (NAME (VALUE-HIGH . VALUE-LOW))
;; [14 Aug 96] I changed this so NAME can now be a symbol,
;; a string, or a list of the form (NAME . PREFIX) [bg]
```

我在注释中插入了一个时间戳，因为这在当我以后要修改这段代码时，我可以知道这段代码是在之前什么时候做出的修改。

如果你知道函数 current-time-string 返回今天的日期和时间字符串，那么插入当前时间的命令是很简单的：[1]

```
(defun insert-current-time ()
  "Insert the current time"
  (interactive "*")
  (insert (current-time-string)))
```

本章后面的章节更多的星号魔法将会解释 (interactive "*") 和 insert 的含义。

上面这个简单的函数非常不灵活，因为它只会插入类似“Sun Aug 19 22:34:53 1996”这种格式的字符串（标准 C 的库函数ctime和asctime的形式）。这在你希望的是日期，或者只是时间，或者 12 小时制而不是 24 小时制，或者日期的形式为”18 Aug 1996”或者“8/18/96”或者“18/8/96”时是非常笨重的。

幸运的是，我们只需要做一点额外的工作就可以获得更好的结果。Emacs 有一些其他的时间相关的函数，特别是 current-time ，它会以一种原始形式返回当前时间，以及 format-time-string ，它以时间值以及格式作为参数（C 函数的strftime的形式）。例如，

```
(format-time-string "%l.%M %p" (current-time))
```

返回“10.38 PM”。(这里使用的格式代码 %l ，即“1-12 小时”，%M ，(0-59 分)，以及 %P ，“AM 或者 PM”。使用 describe-function 来查看 format-time-string 的完整各式列表)。

这样我们就可以很容易地提供两个命令，一个用来插入当前的时间，另一个用来插入当前的日期。我们还可以根据用户提供的格式配置变量来返回特定的各式。这两个函数分别命名为 insert-time 和 insert-date 。而格式配置变量分别为 insert-time-format 和 insert-date-format 。

4.1.1　用户选项和文档字符串

首先我们定义变量。

```
(defvar insert-time-format "%X"
  "*Format for \\[insert-time] (c.f. 'format-time-string').")
```

[1] 如何找到它呢？当然是通过 M-x apropos RET time RET。

```
(defvar insert-date-format "%x"
  "*Format for \\[insert-date] (c.f. 'format-time-string').")
```

关于文档字符串我们能看到两个新特性。

- 首先，每个都以星号（* ）开始。defvar 的文档字符串以星号开头有特殊的含义。它表示这个变量是一个用户选项（user option）。用户选项和其他 Lisp 变量没什么区别，除了下面这两种情况：
  - 用户选项可以使用 set-variable 命令以交互的方式进行设置，Emacs 会问用户要一个变量名（Emacs 会自动补全用户的输入）以及一个值。有时，可以不以 Lisp 语法输入值；也就是说，不必在输入的时候带着外面的括号。当你设置非用户选项的变量值时，你必须这样做：

    ```
    M-: (setq variable value) RET
    ```

    需要使用 Lisp 语法来设置值）。
  - 而用户选项可以通过 M-x edit-options RET [2]激活 option-editing 模式来进行编辑。
- 关于文档字符串的第二个新特性是它们都包含一个特殊的结构 \[command] 。（是的，确实是 \[...] ，但是因为是在 Lisp 字符串里面，反斜杠需要被转义：\\[...] ）。这个语法非常神奇。当文本字符串显示给用户时—例如当用户使用 apropos 或者 describe-variable 时— \[command] 将会被替换为 command 所关联的键绑定。例如，如果 C-x t 会触发 insert-time ，那么文本字符串

  ```
  "*Format for \\[insert-time] (c.f. 'format-time-string')"
  ```

  将显示为

  ```
  *Format for C-x t (c.f. 'format-time-string').
  ```

  如果 insert-time 并没有键绑定，那么将会默认显示 M-x insert-time。如果有多个键绑定到了 insert-time ，Emacs 会自己选择一个。如果你希望字符串 \[insert-time] 不被替换，如何阻止它被替换为对应的键绑定呢？对于这种情况有一个特殊的转义序列：\= 。当 \= 出现在 \[...] 之前时，\[...] 将不会发生替换。当然在 Lisp 字符串里面你需要这样写

[2]在 Emacs 20.1 中，在本身编写时还没发布，将会引入一个全新的用于编辑用户选项的称为“customize” 的系统。将用户选项加入到 “customize” 中需要使用特殊的函数 defgroup 和 defcustom 。

“...\\=\\[...]... ”。如果你并不希望一个变量作为用户选项，而你又希望它的文本字符串以星号开头时，\= 也会起作用。

所有在多个函数之间共享的变量都应该使用 defvar 来声明。如何选择哪些变量作为用户选项存在呢？一个经验之谈是如果某个变量直接控制一个用户可见的并且想要控制的特性，并且设置这个变量很简单时（也就是没有复杂的数据结构和特定的编码值），那么就可以将它设置为用户选项。

4.1.2　更多的星号魔法

前面我们定义了用来控制 insert-time 和 insert-date 的变量，下面就是这两个简单函数的定义。

```
(defun insert-time ()
  "Insert the current time according to insert-time-format."
  (interactive "*")
  (insert (format-time-string insert-time-format
                              (current-time))))

(defun insert-date ()
  "Insert the current date according to insert-date-format."
  (interactive "*")
  (insert (format-time-string insert-date-format
                              (current-time))))
```

这两个函数非常相似，除了一个使用 insert-time-format 而另一个使用 insert-date-format 。insert 函数使用任意数量的参数（类型必须为字符串或者字符），顺序的将它们插入到当前文本位置的后面。

对于这两个函数最需要注意的是它们都以下面的结构开始

```
(interactive "*")
```

之前你已经知道了 interactive 将一个函数转变为一个命令，以及指定用户交互输入时如何获取函数的参数。但是我们在之前并没有看到过 * 作为 interactive 的参数，而且，这两个函数并没有参数，那么这个 * 到底代表什么呢？

当星号作为 interactive 的第一个参数时，这表示 “如果当前 buffer 为只读时终止这个函数”。在函数开始之前就去检测 buffer 是否为只读要比函数执行了一半才提示用户 “Buffer is read-only” 错误信息要更好。在本例中，如果我们

忽略对于 buffer 只读的检测，insert 函数将会触发它自己的 “Buffer is read-only” 错误，这当然也没有什么危害会发生。但是在其他更复杂的函数里，这可能会造成一些不可逆的副作用（例如修改了全局变量）。

4.2　记录戳 (Writestamps)

以一种可配置的格式自动插入当前的时间和日期是非常简洁并且可能超过了大多数编辑器的功能，但是这并不是太有用。很显然更有用的能力是将一个记录戳（文件最后修改的日期、时间）保存在文件里。每次文件保存时记录戳会自动更新。

4.2.1　更新记录戳

首先我们要做的是每次文件保存时自动执行我们的 `writestamp-updating` 代码。就像我们在第二章的章节钩子中看到的，把代码跟某些常用动作（例如保存文件）关联的最好方式就是将函数添加到一个钩子变量里。使用 `M-x apropos RET hook RET` ，我们可以找到四个可能的钩子变量：`after-save-hook` , `local-write-file-hooks` , `write-contents-hooks` 以及 `write-file-hooks` 。

首先我们排除掉 `after-save-hook` 。我们并不希望我们的记录戳在文件保存之后才修改，因为这样我们就永远无法保存文件了（死循环）。

其他候选人的差别比较微妙：

`write-file-hooks` 代码将在 buffer 保存时执行。

`local-write-file-hooks` 一个 `buffer-local` 版本的 `write-file-hooks` 。回忆一下第二章的章节钩子中关于 buffer 局部变量的描述，即每个 buffer 都有自己不同的变量。`write-file-hooks` 作用于每个 buffer，而 `local-write-file-hooks` 做只对单个 buffer 起作用。因此，如果你希望保存 Lisp 文件时执行一个函数，而保存文本文件时执行另一个，那么 `local-write-file-hooks` 就是你的选择。

`write-contents-hooks` `local-write-file-hooks` 是一个 buffer 局部变量，每当 buffer 被保存到文件时它将会执行。但是—就像我提醒过你这很微妙—`write-contents-hooks` 作用于 buffer 的内容，而其他两个钩子作用于编辑的文件。实际上，这意味着如果你改掉了 buffer 的主模式，你也改变了内容的行为方式，因此 `write-contents-hooks` 会被重置为 `nil` 而 `local-write-file-hooks` 却不会。另一方面，如果你更改了 Emacs 关于你正编辑哪个文件的想法，例如通过调用 `set-visited-file-name` ，那么 `local-write-file-hooks` 将会被重置为 `nil` 而 `write-contents-hooks` 却不会。

我们排除掉 `write-file-hooks` ，因为我们只想在拥有记录戳的 buffer 保

存时才调用我们的函数，而并不是所有 buffer 都触发。而撇除语法上的吹毛求疵，我们会排除掉 write-contents-hooks，因为我们希望所选择的钩子变量对于 buffer 的主模式的变更不做回应。这样就只剩下了 local-write-file-hooks。

现在，我们要在 local-write-file-hooks 中放置什么样的函数呢？我们必须定位每个记录戳，删除掉它，并且用新的记录戳来替换它。最简单直接的方法是将每个记录戳用特殊的字符串标记括起来。例如我们可以使用“WRITESTAMP((”放在左边而“))”放在右边，这样它在文件里看起来是这样的：

```
went into the castle and lived happily ever after.
The end. WRITESTAMP((12:19pm 7 Jul 97))
```

假设 WRITESTATMP((...)) 当中的东西是由 insert-date 放入的（我们之前已经定义了），那么它的格式可以通过 insert-date-format 进行控制。

现在，假设文件里已经有了一些记录戳，[3]我们可以在保存文件时这么更新它们：

```
(add-hook 'local-write-file-hooks 'update-writestamps)

(defun update-writestamps ()
  "Find writestamps and replace them with the current time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (while (search-forward "WRITESTAMP((" nil t)
          (let ((start (point)))
            (search-forward "))")
            (delete-region start (- (point) 2))
            (goto-char start)
            (insert-date))))))
  nil)
```

这里有很多的新知识。让我们一行一行的来阅读这个函数。

首先我们看到函数体被包在了一个函数 save-excursion 中。save-excursion 的作用是记录光标的位置，执行参数中的子表达式，然后将光标移动回原处。

[3]插入记录戳与插入日期或者时间很类似。编写这么一个函数就留给读者作为练习了。

在这里它很有用，因为我们的函数体会将光标在 buffer 中到处移动，而在函数结束时我们希望函数的调用者感觉不到这些。在第八章中将会有更多关于 save-excursion 的信息。

下一步调用了 save-restriction 。它的作用方式跟 save-excursion 相似，也是记录了某些信息，然后执行它的参数，然后将信息恢复。这里它记录的是 buffer 的 restriction，它是 narrowing 的结果。narrowing 在第九章中将会做具体描述。现在我们只要知道 narrowing 是 Emacs 的一种只展示 buffer 的一部分的能力。因为 update-writestamps 将会调用 widen ，这会移除掉所有 narrowing 的效果，我们需要 save-restriction 来在我们做完之后恢复现场。

下一步我们要调用 save-match-data ,就像 save-excursion 和 save-restriction ，它保存了一些信息，执行它的参数，然后恢复信息。这一次保存的信息是最后一次搜索的结果。每次查找动作执行时，查找的结果将会被保存到一些全局变量里（我们马上会看到）。每次搜索都会替换掉前面的结果。我们的函数将会执行一次搜索，但是如果出现了其他函数调用我们的函数的情况，我们不希望破坏全局的数据。

下面调用 widen 。就像前面提到的，这会移除所有 narrowing 的效果。它使得整个 buffer 都可以被访问，因为我们需要找到整个 buffer 的记录戳，所以这是必须的。

下面我们使用 (goto-char (point-min)) 将光标移动到了 buffer 的开头，然后开始函数的主循环，也就是搜索整个 buffer 的记录戳并将其更新。函数 point-min 返回 point 的最小值，通常为 1。唯一 (point-min) 不为 1 的情况就是使用了 narrowing。因为我们调用了 widen ，所以 narrowing 不会生效，因此代码也可以写成 (goto-char 1) 。但是使用 point-min 是一种很好的实践)。

主循环看起来是这样的：

```
(while (search-forward "WRITESTAMP((" nil t)
  ...)
```

这是一个 while 循环，它跟其他语言中的 while 循环功能相似。第一个参数是每次循环时的判断表达式。如果表达式返回真，则其他参数被执行，循环继续。

表达式 (search-forward "WRITESTAMP((" nil t) 将会从当前位置开始，搜索第一个匹配的字符串。nil 表示将会一直搜索到 buffer 的结尾。稍后将介绍更多细节。t 表示如果没发现匹配项，search-foward 将会简单的返回 nil 。(如果不设 t , search-forward 在未找到匹配项时将会触发一个错误，终止当前的命令。) 如果搜索成功了，point 将会移动到匹配的字符串之后的第一个字符，search-forward 将会返回这个位置（可以通过使用 match-beginning 来找到搜索开始的位置，如图4.1所示)。

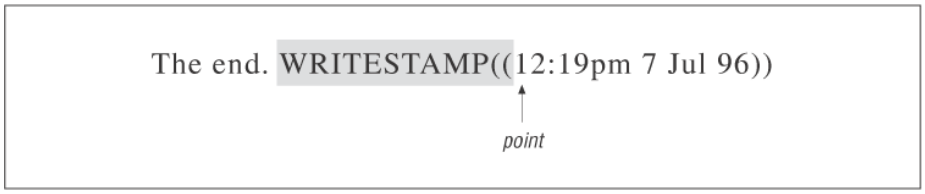

图 4.1: 在搜索了 WRITESTAMP((之后

while 的循环体是

```
(let ((start (point)))
```

这会创建一个临时变量 `start`，用于保存 point 的位置，也就是 WRITESTARMP((...) 分隔符中日期字符串的开始位置。

`start` 定义了之后，`let` 的 `body` 包含如下：

```
(search-forward "))")
(delete-region start (- (point) 2))
(goto-char start)
(insert-date)
```

这里 `search-forward` 会把 point 放置到两个反括号的后面。我们仍然知道时间戳的开头位置，因为它已经保存到了 `start` 中，如图4.2所示。

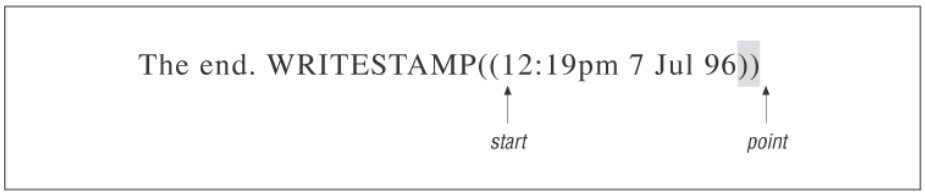

图 4.2: 在搜索了”))”之后

这一次，我们只提供了第一个参数作为搜索字符串。前面我们还看到了两个额外参数：搜索的范围，以及是否触发错误。当省略的时候，他们默认为 `nil`（不限制搜索范围）以及 `nil` （如果搜索失败触发错误）。

在 `search-forward` 成功之后—如果失败了，则函数产生错误并且终止—`delete-region` 将会删除记录戳中的日期，从 `start` 的位置开始到 `(- (point) 2)` 的位置（point 左边两个字符）结束，结果如图4.3所示。

图 4.3: 在删除了start和(- (point) 2)之间的区域之后。

下一步，(goto-char start) 将会把光标移动到记录戳分隔符里里面，最后，(insert-date) 插入当前的日期。

while 循环会在找到匹配项时一直循环下去。每次找到匹配项，光标都必须在匹配项的右面。否则，循环将只能一直搜索到第一项而不会进行下去。

当 while 循环结束后，save-match-data 返回，恢复搜索的全局数据；然后 save-restriction 返回，恢复所有生效的 narrowing；然后 save-excursion 返回，将 point 恢复到原始位置。

update-writestamps 在 save-excursion 调用之后的最后一个表达式，是一个

---

nil

这是函数的返回值。Lisp 函数的返回值就是函数体的最后一个表达式的值。(所有的 Lisp 函数都有一个返回值，但是至今为止我们所写的每个函数都没有返回有意义的返回值，而只是作为一种 “副作用” 存在。) 本例中我们强制它返回 nil 。原因是 local-write-file-hooks 中的函数需要特殊处理。通常，钩子变量中的函数的返回值并不重要。但是对于 local-write-file-hooks （以及 write-file-hooks 和 write-contents-hooks ）中的函数来说，非空的返回值表示，“这个钩子函数接管了将 buffer 写入文件的工作”。如果返回非空值，则钩子变量中的其他函数将不会被执行，而 Emacs 自己的写文件的函数将不会被执行。既然 update-writestamps 没有接替将 buffer 写入文件的工作，我们需要它的返回值为 nil 。

4.2.2　归纳更一般的记录戳

我们实现的记录戳工作了，但是仍然有一些问题。首先，我们的记录戳字符串 “WRITESTAMP((” 和 “)) ” 对于用户来说非常的缺乏美感并且不灵活。第二，用户可能并不希望使用 insert-date 来插入记录戳。

这些问题的修正很简单。我们可以引入三个新的变量：一个就像 insert-date-format 和 insert-time-format 那样描述要使用的时间格式；另外两个用来描述将记录戳括起来的分隔符。

```
(defvar writestamp-format "%C"
  "*Format for writestamps (c.f. 'format-time-string').")

(defvar writestamp-prefix "WRITESTAMP(("
  "*Unique string identifying start of writestamp.")

(defvar writestamp-suffix "))"
  "*String that terminates a writestamp.")
```

现在我们可以修改 update-writestamps 来使它更加灵活。

```
(defun update-writestamps ()
  "Find writestamps and replace them with the current time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (while (search-forward writestamp-prefix nil t)
          (let ((start (point)))
            (search-forward writestamp-suffix)
            (delete-region start (match-beginning 0))
            (goto-char start)
            (insert (format-time-string writestamp-format
                                        (current-time))))))))
  nil)
```

在这个版本的 update-writestamps 里，我们将" WRITESTAMP((" 和 ")) " 替换成了 writestamp-prefix 和 writestamp-suffix ，并且将 insert-date 替换为了

```
(insert (format-time-string writestamp-format
                            (current-time)))
```

我们还改变了 delete-region 的调用。前面它看起来是这样的：

```
(delete-region start (- (point) 2))
```

之前我们的记录戳的后缀被写死为 “)) ”，而它的长度为 2。但是现在我们的后缀被储存在一个变量中，我们并不能提前知道它的长度。我们当然可以通过调用 length 来获得它：

```
(delete-region start (- (point)
                        (length writestamp-suffix)))
```

但是一个更好的解决方案是使用 match-beginning 。记得我们在调用 delete-region 之前是

```
(search-forward writestamp-suffix)
```

不管 writestamp-suffix 是什么，search-forward 都会找到第一个匹配项，并且返回它之后的第一个位置。而关于匹配的其他额外信息，特别是匹配项的开始的位置，被存储在了 Emacs 的一个全局的匹配项变量里。访问这个数据需要通过函数 match-beginning 以及 match-end 。由于稍后可见的原因，match-beginning 需要一个参数 0 来告诉你最后一次搜索的匹配项的开始的位置。本例中，这也就是记录戳后缀的开始的位置，也就是记录戳里面日期的末尾，也就是要删除的范围的结尾：

```
(delete-region start (match-beginning 0))
```

4.2.3　正则表达式

假设用户选择“Written:”和“.”作为 writestamp-prefix 和 writestamp-suffix 的值，那么记录戳看起来将会是这样的：“Written: 19 Aug 1996.”。这是一个很有可能的用户选择，但是字符串 “Written:” 并不像 “WRITESTAMP((” 这么特殊。换句话说，文件中很有可能包含其他 “Written:” 字符串而它并不是一个记录戳。当 updatewritestamps 搜索 writestamp-prefix 时，它将会找到其中一个，然后它会搜索后缀，删掉它们之间所有的东西。更糟糕的是，这种异常删除的发生几乎是不可察的，因为当文件保存之后它就可能发生。

解决这个问题的一种方式是加强记录戳格式的限制，使错误的匹配更难发生。一种自然的可以做出的限制是将记录戳单独存于一行：换句话说，只有当 writestamp-prefix 作为一行的开始而 writestamp-suffix 作为一行的结束时，字符串才有可能是记录戳。

这样

```
(search-forward writestamp-prefix ...)
```

就并不满足用来搜索记录戳了，因为这并不会只在行的开始搜索匹配项。

这就是正则表达式出场的好时机了。正则表达式（regular expression）—简写为 regexp 或者 regex—是一种类似 `search-forward` 的第一个参数那样的搜索模式。并不像通常的搜索模式，正则表达式有一些语法规则提供给我们更强大的搜索功能。例如，在正则表达式‘`^Written:`’中，符号（`^`）是一个特殊符号，表示“这个模式必须匹配行的开始”。表达式‘`^Written:`’剩下的字符在正则中并没有什么特殊的含义，所以他们和普通的搜索模式所表达的意思一样。特殊的字符有时被称为元字符（metacharacters）或者（更有诗意的）魔法字符。

许多 UNIX 程序使用了正则，这包括 sed，grep，awk 以及 pert。不幸的是每个程序的正则都或多或少的不一样；但是在所有情况下，大多数字符是非“魔法”的（特别是字母和数字）并且可以被用来搜索他们自己；更长的正则可以由短一些的正则拼接而成。下面是 Emacs 中使用的正则表达式的语法。

1. 点号（`.`）匹配除换行符外的所有单个字符。
2. 反斜杠后面跟任何元字符则匹配该字符本身。例如，`\.` 将匹配点号。而且反斜杠本身是一个元字符，`\\` 将会匹配 `\` 本身。
3. 中括号里的字符匹配任何括号里的字符。所以 `[aeiou]` 匹配任何 a 或者 e 或者 i 或者 o 或者 u。这个规则有一些例外—正则表达式的方括号语法有自己的“子语法”，如下：
   (a) 连续的字符，例如 abcd，可以简写为 `a-d` 。这个范围可以为任意长度，并且可以和其他单个字符混合。所以 `[a-dmx-z]` 可以匹配任何 a, b, c, d, m, x, y, 或者 z。
   (b) 如果第一个字符是（`^`），那么表达式匹配任何不在方括号内的字符。所以 `[^a-d]` 匹配除了 a, b, c, 或者 d 之外的字符。
   (c) 要包括一个右中括号，它必须是第一个字符。所以 `[]a]` 匹配 `]` 或者 a。同样的，`[^]a]` 匹配任何除 `]` 和 a 之外的字符。
   (d) 要包括一个中横线，它必须出现在一个不能被表意为范围的地方；例如，它可以是第一个或者最后一个字符，或者跟在某个范围的后面。所以 `[a-e-z]` 匹配 a, b, c, d, e，`-`，或者 z。
   (e) 要包括一个（`^`），它必须出现在除第一个字符之外的地方。
   (f) 其他正则中的元字符，例如 `*` 和 `.` 在方括号中作为普通字符存在。
4. 正则表达式 x 可能有以下后缀：
   (a) 星号 `*` ，匹配 0 或多个 x
   (b) 加号 `+` ，匹配 1 或多个 x
   (c) 问号 `?` ，匹配 0 或 1 个 x 所以 `a*` 表示 a, aa, aaa 甚至空字符串（0 个 a）；[4]a+ 匹配 a, aa, aaa, 但是不能为空；`a?`匹配空字符串和 a。可

[4] * 在正则表达式中被计算机科学家称为“Kleene Closure”。

以注意到x+等同于 xx* 。

5. 正则表达式 ^x 匹配任何行首 x 所匹配的值。x$ 匹配任何行尾 x 匹配的值。这表示 ^x$ 匹配一行只包含 x 的值。而你也可以把 x 去掉；^$ 匹配不包含任何字符的行。
6. 两个正则表达式 x 和 y 被 \| 分割表示匹配任何 x 匹配的或者 y 匹配的值。所以 hello\|goodbye 匹配 hello 或者 goodbye。
7. 正则表达式 x 被转义的括号所包裹— \(和 \) —匹配任何 x 匹配的东西。这可以被用于分组复杂的表达式。所以 \(ab\)+ 匹配 ab, abab, ababab, 等等。同样，\(ab\)\|\(cd\)ef 匹配 abef 或者 cdef。作为副作用，任何被括起来的子表达式匹配的文本被称为子匹配项（submatch）并且被储存在一个编号的记录器内。子匹配项根据 \(从左到右出现的位置而编号为 1 到 9。所以如果用正则表达式 ab\(cd*e\) 匹配文字 abcddde，那么只会匹配到子匹配项 cddde。如果使用 ab\(cd\|ef\(g+h\)\)j\(k*\) 匹配文字 abefgghjkk，那么第一个子匹配项是 efggh，第二个是 ggh，第三个是 kk。
8. 反斜杠后面跟一个数字 n 表示匹配和第 n 个括起来的子表达式相同的文本。所以表达式 \(a+b\)\1 匹配 abab，aabaab，和 aaabaaab，但是不匹配 abaab（因为 ab 和 aab 不同）。
9. 有很多种方法可以匹配空字符串。
   (a) \` 匹配在 buffer 开始处的空字符串。所以 \`hello 匹配 buffer 开头处的 hello，而不匹配任何其他的 hello。
   (b) \' 匹配 buffer 末尾处的空字符串。
   (c) \= 匹配当前 point 位置处的空字符串。
   (d) \b 匹配单词开始或结尾处的空字符串。所以 \bgnu\b \verb 匹配词“gnu”但是不能匹配单词“interegnum”中的“gnu”。
   (e) \B匹配任何除单词开始和结尾处的空字符串。所以 \Bword 匹配“sword”中的“word”而不匹配“words”中的“word”。
   (f) \< 只匹配单词开始处的空字符串。
   (g) \> 值匹配单词结束处的空字符串。

如你所见，正则表达式在很多情况下使用反斜杠。Emacs Lisp 字符串语法也如此。而由于在编写 Emacs 时正则表达式是使用 Lisp 字符串写的，这两种使用反斜杠的规则将会引起一些令人烦恼的结果。例如，正则表达式 ab\ cd|，当以 Lisp 字符串写出时，需要写成“ab\\ cd|”。更奇怪的是当你想要使用正则表达式 \\ 匹配反斜杠 \ 时：你必须写成“\\\\ ”。提示你输入正则表达式的 Emacs 命令（例如 apropos 和 keeplines ）允许你在输入时只写正则而不用写成 Lisp 字符串的形式。

4.2.4　正则引用

现在我们知道了如何使用正则表达式，看起来搜索行首的 writestamp-prefix 只需要在它前面加一个（^）而行尾的 writestamp-suffix 只需要在后面加一个 $，就像这样：

```
(re-search-forward (concat "^"
                           writestamp-prefix) ...) ; 错啦!

(re-search-forward (concat writestamp-suffix
                           "$") ...) ; 错啦!
```

函数 concat 将它的参数合成一个单独字符串。函数 re-search-forward 是 search-forward 的正则表达式版本。

这几乎是正确的。但是，它有一个常见的错误：writestamp-prefix 或者 writestamp-suffix 都可能包含元字符。实际上，writestamp-suffix 确实有，在我们的例子里即“.”。因为点号匹配任何字符（除了换行符），这个表达式：

```
(re-search-forward (concat writestamp-suffix
                           "$") ...)
```

等同于表达式：

```
(re-search-forward ".$" ...)
```

这会匹配任何行尾的字符，而我们只想要匹配点号（.）。

当像本例中那样构建一个正则表达式，而 writestamp-prefix 的内容却超出了程序员的控制时，移除字符串中包含的元字符的“魔力”而让他们只表达字面意思是必须的。Emacs 为此提供了一个函数 regexp-quote，它理解正则的语法然后将一个正则表达式字符串转换为对应的“非魔法”的字符串。例如 (regexp-quote ".") 会产生“\\.”。你应该总是使用 regexp-quote 来移除作为变量提供的字符串中的魔力。

我们现在知道了如何开始编写新版本的 update-writestamps：

```
(defun update-writestamps ()
  "Find writestamps and replace them with the current time."
  (save-excursion
    (save-restriction
```

```
      (save-match-data
        (widen)
        (goto-char (point-min))
        (while (re-search-forward
                (concat "^"
                        (regexp-quote writestamp-prefix))
                nil t)
          ...))))
  nil)
```

4.2.5　有限搜索

让我们编写 while 循环的 body 来完成新版本的 update-writestamp 。在 re-search-forward 完成后，我们需要知道当前行是否以 writestamp-suffix 结束。但是我们不能简单的这么写

```
(re-search-forward (concat (regexp-quote writestamp-suffix)
                           "$"))
```

因为这可能会匹配到非本行的匹配项。我们只对本行是否匹配感兴趣。

我们的解决方式是只把搜索限制在本行。search-forward 和 re-search-forward 的第二个可选参数，如果不是 nil 的话，是指搜索时不超过的位置。如果我们将当前行的末尾位置作为参数传入：

```
(re-search-forward (concat (regexp-quote writestamp-suffix)
                           "$")
                   end-of-line-position)
```

那么搜索就会限制到本行之内，这正是我们需要的。那么问题是我们如何得到 end-of-line-position 的值呢？我们可以简单的使用 endf-of-line 将光标移动到行尾，然后得到 point 的值。但是要记住在这样做之后我们需要把光标移动到它原来的地方。移动光标然后恢复场景的工作正是 save-excursion 所做的。所以我么可以这么写：

```
(let ((end-of-line-position (save-excursion
                             (end-of-line)
                             (point))))
  (re-search-forward (concat (regexp-quote writestamp-suffix)
```

```
                                "$")
                        end-of-line-position))
```

这会创建一个临时变量 end-of-line-position 来限制 re-search-forward 的搜索范围；但是不使用这个变量更简单：

```
(re-search-forward (concat (regexp-quote writestamp-suffix)
                           (save-excursion
                             (end-of-line)
                             (point))))
```

注意 save-excursion 表达式的返回值是它的最后一条语句 (point) 的值。所以 update-writestamps 可以被写成：

```
(defun update-writestamps ()
  "Find writestamps and replace them with the current time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (while (re-search-forward
                (concat "^"
                        (regexp-quote writestamp-prefix))
                nil t)
          (let ((start (point)))
            (if (re-search-forward (concat (regexp-quote
                                            writestamp-suffix)
                                           "$")
                                   (save-excursion
                                     (end-of-line)
                                     (point))
                                   t)
                (progn
                  (delete-region start (match-beginning 0))
                  (goto-char start)
                  (insert (format-time-string writestamp-format
                                              (current-time))))))))))
  nil)
```

4.2.6　更强大的正则能力

我们已经把我们最初的 update-writestamps 转换成了正则的形式，但是却并没有真正的展现出正则强大的能力。实际上，上面那长长的用于找到记录戳，检测同一行内的记录戳后缀，然后将其替换的代码可以被简化为下面的两个表达式：

```
(re-search-forward (concat "^"
                           (regexp-quote writestamp-prefix)
                           "\\(.**\\)"
                           (regexp-quote writestamp-suffix)
                           "$"))
(replace-match (format-time-string writestamp-format
                                   (current-time))
               t t nil 1)
```

第一个表达式，使用下面的正则调用了 re-search-forward ：

```
^prefix\(.*\)suffix$
```

这里的 prefix 和 suffix 是 regexp-quote 版本的 writestamp-prefix 和 writestamp-suffix 。这个正则表达式匹配以记录戳前缀开始，跟着任何字符串（使用 \(...\) 构建的子匹配项），以记录戳后缀结束的一行。

第二个表达式调用了 replace-match ，它将会替换部分或者所有前一次搜索的匹配项。它的用法如下：

```
(replace-match new-string
               preserve-case
               literal
               base-string
               subexpression)
```

第一个参数是要插入的新字符串，本例中也就是 format-time-string 的返回值。剩下的参数都是可选参数，解释如下：

preserve-case　我们将它设为 t ,告诉 replace-match 从前往后匹配 new-string 。如果设为 nil ， replace-match 将会尝试进行智能匹配。

literal　我们使用 t 来表示“按照字面理解 new-string ”。如果使用 nil ，那么 replace-match 将会使用一些特殊的语法规则理解 new-string （可以使用 describe-function replace-match 来具体查看）。

base-string　我们使用 nil 来表示“更改当前 buffer”。如果使用一个字符串，那么 replace-match 将会在那个字符串里执行替换。

subexpression　我们使用 1 表示“替换子匹配项 1，而不是整个匹配的字符串”（这将包括前缀和后缀）。

所以在使用 re-search-forward 查找记录戳然后找到分隔符之间的“子匹配项”之后，我们调用 replace-match 将分隔符之间的文本删掉并且插入了一个根据 writestamp-format 生成的新字符串。

作为对于 update-writestamps 的最终改进，我们可以看到如果我们这样写

```
(while (re-search-forward (concat ...) ...)
  (replace-match ...))
```

那么 concat 函数在每次循环里都会调用，即使参数没有改变，每次都会生成一个新的字符串。这样效率很低。更好的方式是在循环之前计算出我们需要的字符串，然后存储在一个临时变量里。因此，最好的 update-writestamps 是这样的：

```
(defun update-writestamps ()
  "Find writestamps and replace them with the current time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (let ((regexp (concat "^"
                              (regexp-quote writestamp-prefix)
                              "\\(.*\\) "
                              (regexp-quote writestamp-suffix)
                              "$")))
          (while (re-search-forward regexp nil t)
            (replace-match (format-time-string writestamp-format
                                               (current-time))
                           t t nil 1))))))
  nil)
```

4.3　修改戳

好的，时间戳（timestamps）挺有用，而记录戳（writestamps）也不错，但是修改戳（modifystamps）可能更有用。一个修改戳是一个记录着文件最后修改时间的记录戳，这可能和文件最后存储到磁盘上的时间不一样。例如，如果你访问了一个文件并且在没做任何修改的情况下将其保存在磁盘上，你就不应该更新修改戳。

在本节，我们将大略的探索两种非常简单的方式来实现修改戳。

4.3.1　简单的方式 #1

Emacs 有一个称为 `first-change-hook` 的钩子变量。每当 buffer 自保存之后第一次被修改，变量中的函数将会被调用。使用这个钩子来实现修改戳只是把我们之前的 `update-writestamps` 函数从 `local-write-file-hooks` 变为 `first-change-hook` 。当然，我们还要把它的名字改为 `update-modifystamps` , 并且引入一些新的变量— `modifystamp-format` , `modifystamp-prefix` , 以及 `modifystamp-suffix` —而不影响原来记录戳的那些变量。`update-modifystamps` 需要使用这些新的变量。

在此之前，`first-change-hook` 是一个全局变量，而我们需要一个 buffer 局部的。如果我们将 `update-modifystamps` 添加到 `first-change-hook` 而 `first-change-hook` 是全局的，那么任何 buffer 保存的时候都会触发这个方法。我们需要将它变为 buffer 局部的，而其他 buffer 则继续使用默认的全局变量。

```
(make-local-hook 'first-change-hook)
```

虽然可以使用 `make-localvariable` 或者 `make-variable-buffer-local` 来使普通变量变为 buffer 局部的(下面会看到),但是钩子变量必须使用 `make-local-hook` 。

```
(defvar modifystamp-format "%C"
  "*Format for modifystamps (c.f. 'format-time-string').")

(defvar modifystamp-prefix "MODIFYSTAMP (("
  "*String identifying start of modifystamp.")

(defvar modifystamp-suffix "))"
  "*String that terminates a modifystamp.")
```

```
(defun update-modifystamps ()
  "Find modifystamps and replace them with the current time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (let ((regexp (concat "^"
                              (regexp-quote modifystamp-prefix)
                              " \\(.*\\) "
                              (regexp-quote modifystamp-suffix)
                              "$")))
          (while (re-search-forward regexp nil t)
            (replace-match (format-time-string modifystamp-format
                                               (current-time))
                           t t nil 1))))))
  nil)
(add-hook 'first-change-hook 'update-modifystamps nil t)
```

add-hook 中的 nil 参数只是一个占位符。我们只关注最后一个参数 t，它表示“只更改 first-changehook 的 buffer 局部备份”。

这种方式的问题是如果你在保存文件前对其进行了十处修改，那么修改戳会记录第一次的时间，而不是最后一次的时间。某些情况下这已经足够用了，但是我们还可以做得更好。

4.3.2　简单的方式 #2

这一次我们再次使用 local-write-file-hooks ,但是我们只在 buffer-modified-p 返回 true 的时候才调用 update-modifystamps，也就是说只在当前 buffer 被改动的情况下才调用它：

```
(defun maybe-update-modifystamps ()
  "Call 'update-modifystamps' if the buffer has been modified."
  (if (buffer-modified-p)
      (update-modifystamps)))
(add-hook 'local-write-file-hooks 'maybe-update-modifystamps)
```

现在我们有了跟方式 #1 相反的问题：最后修改的时间一直到文件保存的时候才会计算，这可能比最后一次修改的时间晚很久。如果你在 2:00 的时候修

改了文件，而在 3:00 的时候做了保存，那么修改戳将会把 3:00 作为最后保存的时间。这更接近了，但是并不完美。

4.3.3　聪明的方式

理论上，我们可以在每次更改 buffer 之后调用 update-modifystamps，但是实际中在每次按键之后都搜索整个文件并且对其进行修改是代价昂贵的一件事。但是每次 buffer 更改之后将时间记录下来就不那么难以接受。然后，当 buffer 保存到文件时，记录的时间就可以用来计算修改戳中的时间了。

钩子变量 after-change-functions 包含着每次 buffer 更改时要调用的函数。首先我们让它变为 buffer-local 的：

```
(make-local-hook 'after-change-functions)
```

然后我们定义一个 buffer-local 的变量来保存这个 buffer 最后一次修改的时间：

```
(defvar last-change-time nil
  "Time of last buffer modification.")
(make-variable-buffer-local 'last-change-time)
```

函数 make-variable-buffer-local 使得它后面的变量在每个 buffer 都具有独立的、buffer-local 的值。这根 make-local-variable 有些不同，其作用是使变量在当前 buffer 获得一个 buffer-local 的值，而让其他 buffer 共享一个相同的全局值。在这里，我们使用 make-variable-buffer-local 是因为所有 buffer 共享一个全局的 last-change-time 是没有意义的。

现在我们需要一个函数来在每次 buffer 改变的时候修改 last-change-time 的值。让我们将其命名为 remember-change-time 并且将它添加到 after-change-functions 里：

```
(add-hook 'after-change-functions 'remember-change-time nil t)
```

after-change-functions 中的函数有三个参数来描述刚刚发生的改变（参照第七章中的Mode Meat）。但是 remember-change-time 并不关心刚才发生了什么更改；它只关心发生了更改这件事本身。所以我们可以选择忽略这些参数。

```
(defun remember-change-time (&rest unused)
  "Store the current time in 'last-change-time'."
  (setq last-change-time (current-time)))
```

关键字 &rest ，后面跟着一个参数名称，只能出现在函数的参数列表最后。它表示“将剩下的参数收集到一个列表里并且赋给最后的参数”（本例中为 unused）。函数可能还有其他的参数，包括 &optional 的可选参数，但是这些都要出现在 &rest 的前面。在所有其他参数按照正常格式分配完成后，&rest 将其他剩下的参数放到一个列表里。所以如果一个函数这么定义

```
(defun foo (a b &rest c)
  ...)
```

那么当 (foo 1 2 3 4) 调用时，a 将为 1 ，b 为 2 ，c 将会是列表 (3 4) 。

在有些情况下，&rest 非常有用，甚至是必须的；但在这里我们只是出于懒惰（或者节约，如果你希望这么称呼的话），来规避给三个我们并不希望使用的参数命名。

现在我们来修改 update-modifystamps :它必须使用储存在 last-change-time 中的时间而不是使用 (current-time)。从效率考虑，它还需要在执行完成后将 last-change-time 置为 nil ，这样可以避免以后当文件在未修改的情况下进行保存时对 update-modifystamps 的额外调用。

```
(defun update-modifystamps ()
  "Find modifystamps and replace them with the saved time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (let ((regexp (concat "^"
                              (regexp-quote modifystamp-prefix)
                              "\\(.*\\)"
                              (regexp-quote modifystamp-suffix)
                              "$")))
          (while (re-search-forward regexp nil t)
            (replace-match (format-time-string modifystamp-format
                                               last-change-time)
                           t t nil 1))))))
  (setq last-change-time nil)
  nil)
```

最后,我们不希望在 last-change-time 为 nil 时调用 update-modifystamps :

```
(defun maybe-update-modifystamps ()
  "Call 'update-modifystamps' if the buffer has been modified."
  (if last-change-time ; 替换对于(buffer-modified-p)的检测
      (update-modifystamps)))
```

maybe-update-modifystamps 中仍然存在很大的问题。在阅读下一部分前，你能找出那是什么吗?

4.3.4 一个小 Bug

缺陷是每次 update-modifystamps 重写修改戳时，会引起 buffer 的改变，这又会造成 last-change-time 的改变！这样只有第一次修改戳会被正确的修改。后续的修改戳会是一个与文件储存的时间相近的时间而不是最后一次修改的时间。

一个绕过这个问题的方法是当执行 update-modifystamps 时暂时将 after-change-functions 置为 nil ：

```
(add-hook 'local-write-file-hooks
          '(lambda ()
            (if last-change-time
                (let ((after-change-functions nil))
                  (update-modifystamps)))))
```

let 创建了一个临时变量 after-change-functions ，用来在调用 let 体中的 update-modifystamps 时替代全局变量 after-change-functions 。当 let 退出后，临时变量 after-change-functions 就销毁了，而全局变量又再次发生作用。

这个方法有一个缺点：如果 after-change-functions 中有其他的函数，那么在你调用 update-modifystamps 时它们也会暂时失效，而这并不是你希望看到的。

一个更好的方法是在每次更新修改戳之前“截取” last-change-time 的值。这样，当更新修改戳造成 last-change-time 改变时，新的 last-change-time 的值将不会影响其他的修改戳，因为 update-modifystamps 并不会使用当前储存在 last-change-time 中的值。

“截取”last-change-time 最简单的方式是将其作为参数传递给 update-modifystamps :

```
(add-hook 'local-write-file-hooks
          '(lambda ()
```

```
            (if last-change-time
                (update-modifystamps last-change-time))))
```

这需要修改 update-modifystamps ,使其具有一个参数,并且在调用 format-time-string 时使用它:

```
(defun update-modifystamps (time)
  "Find modifystamps and replace them with the given time."
  (save-excursion
    (save-restriction
      (save-match-data
        (widen)
        (goto-char (point-min))
        (let ((regexp (concat "^"
                              (regexp-quote modifystamp-prefix)
                              "\\(.*\\)"
                              (regexp-quote modifystamp-suffix)
                              "$")))
          (while (re-search-forward regexp nil t)
            (replace-match (format-time-string modifystamp-format
                                               time)
                           t t nil 1))))))
  (setq last-change-time nil)
  nil)
```

你可能会觉得为了使修改戳工作，你写出了许多表达式，建立了很多变量，而这看起来很难维护。你是对的。所以在下一章，我们来看看如何在 Lisp 文件中封装相关的方法和变量。

第五章　Lisp 文件

在本章：

- 创建 Lisp 文件
- 加载文件
- 编译文件
- eval-after-load
- 局部变量列表
- 补充：安全考虑

到目前为止，我们编写的大多数 Emacs Lisp 都可以放进你的.emacs 文件里。一种替代的方案是将 Emacs Lisp 的代码根据功能放进不同的文件里。这需要更多的努力，但是这同样也比把代码直接放到.emacs 里多了一些好处：

- .emacs 中的代码在 Emacs 启动时总是会执行，即使我们在当前的工作中可能根本不需要它。这会使启动时间延长并且会消耗内存。相反的，分离的 Lisp 文件只有在我们需要的时候才去载入它。
- .emacs 中的代码通常并不会进行字节编译（byte-compiled）。字节编译会将 Emacs Lisp 转换成载入更高效、执行更快并且使用更少内存的格式（就像其他语言的程序一样，这会将代码变成不适于程序员阅读的格式）。字节编译过的 Lisp 文件通常以.elc（“Emacs Lisp, compiled”）作为后缀名，而未进行编译的文件通常以.el（“Emacs Lisp”）后缀。
- 将所有代码放到.emacs 里将会使文件膨胀成难以管理的乱麻。

前面章节所编写的代码就是这种可以根据功能划分到不同 Lisp 文件中的很好的例子，它们只应该在需要的时候载入，并且应该进行字节编译以提高执行效率。

5.1　创建 Lisp 文件

Emacs Lisp 文件名通常以.el 为后缀，所以作为开始，让我们创建 timestamp.el 并且将上一章中最后完成的代码放进去，如下所示。

```
(defvar insert-time-format ...)
(defvar insert-date-format ...)
(defun insert-time () ...)
(defun insert-date () ...)
(defvar writestamp-format ...)
(defvar writestamp-prefix ...)
(defvar writestamp-suffix ...)
(defun update-writestamps () ...)
(defvar last-change-time ...)
(make-variable-buffer-local 'last-change-time)
(defun remember-change-time ...)
(defvar modifystamp-format ...)
(defvar modifystamp-prefix ...)
(defvar modifystamp-suffix ...)
(defun maybe-update-modifystamps () ...)
(defun update-modifystamps (time) ...)
```

先不要加入 `add-hook` 和 `make-local-hook` 。我们后面再来关注他们。现在，你需要注意的是当编写 Lisp 文件的时候，它应该能在任意时机被加载，甚至加载多次，而并不会带来你不希望的副作用。一个这种副作用的例子是，假如你不希望当前 buffer 的 `after-change-functions` 变为局部变量，而又把 `(make-local-hook 'after-change-functions)` 加入了到 timestamp.el 中。

5.2 加载文件

当把代码写入到 timestamp.el 中后，我们必须进行配置，以使我们在需要的时候能够正确的访问到它们。这是通过加载（loading）完成的，也就是使 Emacs 读取和执行它的内容。在 Emacs 中有许多方式来加载 Lisp 文件：交互式的（interactively）、非交互式的（non-interactively）、明确的（explicitly）、模糊的（implicitly）、以及使用或者不使用路径搜索（pathsearching）的。

5.2.1 找到 Lisp 文件

Emacs 能够根据像 `/usr/local/share/emacs/site-lisp/foo.el` 这样的绝对路径来加载文件，但是通常更为方便的方式是直接使用如 bo.el 这样的文件名，而让 Emacs 在加载路径（load path）中去找到它。加载路径简单来说就是 Emacs 用来搜索要加载的文件的目录列表，跟 UNIX shell 中使用环境变量 PATH 来找到要执行的程序相似。Emacs 的加载路径储存在一个字符串列表变

量 load-path 里。

当 Emacs 启动的时候，load-path 的初始设置大概看起来跟这个差不多：

```
("/usr/local/share/emacs/19.34/site-lisp"
 "/usr/local/share/emacs/site-lisp"
 "/usr/local/share/emacs/19.34/lisp")
```

搜索路径时的顺序与其在列表中出现的顺序相同。要在 load-path 的头部加入一个路径，在你的.emacs 文件中加入下面的代码：

```
(setq load-path
      (cons "/your/directory/here"
            load-path))
```

要在其末尾加入，则使用：

```
(setq load-path
      (append load-path
              '("/your/directory/here")))
```

注意到在第一个例子中，“/your/directory/here ”只作为一个普通字符串，而在第二个例子中，它却出现在一个括起来的列表中。第六章将会解释 Lisp 中各种处理列表的方式。

如果你要求 Emacs 在加载路径中查找一个 Lisp 文件并且忽略掉后缀名的话—例如你使用 foo 而不是 foo.el—Emacs 会首先查找 fool.elc，也就是 foo.el 的字节编译格式。如果找不到，那么它将会尝试 foo.el，最后是 foo。通常加载文件时最好忽略掉后缀名。因为这样不仅会使你得到更有效的查找行为，而且这还会让 eval-after-load 工作的更好（阅读本章后面的eval-after-load章节获得更多信息）。

5.2.2 交互式加载

有两个 Emacs 命令用来交互式的加载 Lisp 文件：load-file 和 load-library 。当你输入 M-x load-file RET 时，Emacs 会提示你输入一个 Lisp 文件的绝对路径（例如 /home/bobg/emacs/foo.el ）而不去搜索 load-path 。这会使用通常的文件名提示方法，所以文件名会进行自动补全。而另一种方式，当你输入 M-x load-library RET 时，Emacs 将会提示你输入库的基本名称（例如 foo）并且尝试在 load-path 中找到它。这不会使用文件名提示方法，因此也就不会进行自动补全。

5.2.3 以代码加载

当在 Lisp 代码中加载文件时，你可以选择显式加载、条件加载或者自动加载。

显式加载

可以显式的调用 load （和交互式加载中的 load-library 行为类似）或者 load-file 加载文件。

```
(load "lazy-lock")
```

会搜索 load-path 来查找 lazy-lock.elc、lazy-lock.el 或者 lazy-lock。

```
(load-file "/home/bobg/emacs/lazy-lock.elc")
```

将不会使用 load-path 。

显式加载应该在你确定需要马上加载某个文件时才使用，并且你应该确信文件并没有加载过或者你并不关心。事实上，如果使用下面的替代方式，你基本上不会用到显式加载这种方式。

条件加载

当 n 处不同的 Lisp 代码想要载入同一个文件时，有两个 Emacs Lisp 函数，即 require 和 provide ，给出了一种方法来确保文件只会被加载一次而不是 n 次。

一个 Lisp 文件通常包含着一系列相关的函数。我们可以认为这些函数是一个单独的特性（feature）。加载这个文件会使得其包含的特性可用。

Emacs 明确了特性这个概念。特性通过 Lisp 符号进行命名，使用 provide 进行声明，使用 require 进行请求。

它是这么工作的。首先，我们为 timestamp.el 提供的特性来选择一个代表的符号。让我们使用一个明显的，timestamp。我们通过在 timestamp.el 中写入

```
(provide 'timestamp)
```

来表明 timestamp.el 提供了特性 timestamp。通常它出现在文件的末尾，这样只有当前面所有语句都正确执行后，这个特性才会被标识为 “provided”。(如果发生了什么异常，那么文件的加载将会在调用 provide 之前终止)。

现在假设在某处的代码需要使用时间戳功能。使用 require ：

```
(require 'timestamp "timestamp")
```

这表示，“如果 timestamp 特性还不可用，那么加载 timestamp”（这会使用 `load`，并且会搜索 `load-path`）。如果 timestamp 特性已经提供了（也就是 timestamp 之前已经加载了），那么什么都不做。

通常，对于 `require` 的调用通常都出现在 Lisp 文件的头部—就像 C 程序中通常以一大堆`#includes`开头一样。但是有些程序员喜欢在需要某个特性时才在那里调用 `require`。这可能有很多个地方，而如果每次都会真的去加载文件的话，程序将会慢的像爬，每载入一个文件都会耗费大量的时间。使用“特性”将会使得文件只载入一次，大量的节省时间！

调用 `require` 时，如果文件名是特性名的“字符串等价式”，那么文件名可以被省略而根据特性名推断出来。符号的“字符串等价式”就是简单的符号名称的字符串格式。特性符号 timestamp 的字符串等价式为“timestamp”，所以我们可以这么写：

```
(require 'timestamp)
```

来替换（`require 'timestamp "timestamp"`）。（函数 `symbol-name` 可以得到符号的字符串等价式）。

如果 `require` 使得相关的文件被加载（这时其特性还未被提供），那么那个文件应该 `provide` 请求的特性。否则，`require` 将会报告加载的文件并没有提供需要的特性。

自动加载

当使用自动加载（autoloading）时，你可以推迟一个文件的加载直到它必须加载的时候—也就是直到你调用了它里面的方法。设置自动加载代价很小，因此通常都是在.emacs 文件里做。

函数 `autoload` 将一个函数名称与一个定义它的文件联系在一起。当 Emacs 尝试调用一个还未被定义的函数时，它将根据 `autoload` 来载入指定的文件，并且假定它做出了定义。而不使用 `autoload`，尝试执行一个未定义的函数将会报错。

下面是如何使用：

```
(autoload 'insert-time "timestamp")
(autoload 'insert-date "timestamp")
(autoload 'update-writestamps "timestamp")
(autoload 'update-modifystamps "timestamp")
```

当第一次调用这四个函数 insert-time ,insert-date ,updatewritestamps ，或者 update-modifystamps 中的任意一个时，Emacs 都将加载 timestamp。这不仅会载入被调用的函数定义，并且会使另外的三个也被载入，所以后续的对于这几个函数的调用不会重新载入 timestamp。

autoload 函数有多个可选参数。第一个参数是文档字符串。文档字符串允许用户甚至在还未从文件中加载函数的定义之前就得到帮助(通过 describe-function 和 apropos)。

```
(autoload 'insert-time "timestamp"
  "Insert the current time according to insert-time-format.")
(autoload 'insert-date "timestamp"
  "Insert the current date according to insert-date-format.")
(autoload 'update-writestamps "timestamp"
  "Find writestamps and replace them with the current time.")
(autoload 'update-modifystamps "timestamp"
  "Find modifystamps and replace them with the given time.")
```

下一个可选参数定义了当函数被加载后是一个交互式命令还是一个普通函数。如果忽略或者为 nil ，函数将被认为是非交互的；否则它就是一个命令。函数被加载之前，像 command-apropos 这样的函数也可以使用这个信息来区分函数是交互式还是非交互式的。

```
(autoload 'insert-time "timestamp"
  "Insert the current time according to insert-time-format."
  t)
(autoload 'insert-date "timestamp"
  "Insert the current date according to insert-date-format."
  t)
(autoload 'update-writestamps "timestamp"
  "Find writestamps and replace them with the current time."
  nil)
(autoload 'update-modifystamps "timestamp"
  "Find modifystamps and replace them with the given time."
  nil)
```

如果你在 autoload 中错误的将交互式函数标记为非交互的，或者相反，当函数被载入之后就不要紧了。真正的函数定义会将所有 autoload 所给出的信息替换掉。

最后一个可选参数我们这里不会讲。如果自动加载对象的类型不是函数的话，它会指定类型。就像它所指出的，键映射表和宏（我们后面的章节将会讲到）也可以被自动加载。

5.3 编译文件

就像在本章开始所提到的，当我们将 Lisp 代码储存在各自的文件里之后，我们可以对它进行字节编译（byte-compile）。字节编译将 Emacs Lisp 转换成更紧凑、执行更快的格式。就像在其他编程语言中的编译一样，程序员很难阅读字节编译的结果。但不像其他的编译，字节编译的结果在不同硬件平台和操作系统上是可移植的（但可能不兼容老版本的 Emacs）。

字节编译过的 Lisp 代码比未编译的代码执行速度快很多。

字节编译的 Lisp 文件后缀名为.elc 文件。就像前面提到的，不使用后缀名调用 `load` 和 `load-library` 将会优先加载.elc 文件而不是.el 文件。

有几种字节编译文件的方式。最直接的方式是

- 在 Emacs 里：执行 `M-x byte-compile-file RET` file.el `RET` 。
- 在 UNIX shell 里：执行`emacs -batch -f batch-byte-compile file.el`。

你可以对 Lisp 文件的路径执行 `byte-recompile-directory` 来进行字节编译。

当 Emacs 要载入的.elc 文件的日期比对应的.el 文件的日期旧时，Emacs 将仍然会载入它，但是会提示一个警告。

5.4 eval-after-load

如果你希望直到某个特定文件加载之后才执行某些代码，`eval-after-load` 就是你需要的。例如，假如你搞出了一个比 dired（Emacs 的目录编辑模块）自身的 `dired-sort-toggle` 更好的函数定义。你不能简单的把它放入.emacs 中，因为一旦你编辑一个目录，dired 将会被自动加载，而这将会把你自己的定义替换掉。

你应该做的是：

```
(eval-after-load
    "dired"
  '(defun dired-sort-toggle ()
     ...))
```

这将会在 dired 加载之后马上执行 `defun` ，将 dired 自己的 `dired-sort-toggle` 替换为其他的版本。但是要注意，这只有当使用 dired 这个名称加载时才会工

作。如果使用名称 dired.elc 或者 /usr/local/share/emacs/19.34/lisp/dired 加载 dired 的话就不会执行。load 或者 autoload 或者 require 必须使用同 eval-after-load 中一模一样的名称才可以。这也就是前面提到的为什么最好只用文件的基本名称加载文件的原因。

eval-after-load 的另一个作用是当你希望在一个包中使用某个变量、函数或者按键映射，而你又不想强制这个包加载的时候：

```
(eval-after-load
    "font-lock"
  '(setq lisp-font-lock-keywords lisp-font-lock-keywords-2))
```

这里使用了 font-lock 定义的变量 lisp-font-lock-keywords-2 。如果你在 fontlock 加载之前使用 lisp-font-lock-keywords-2 ，你将会得到一个“Symbol's value as variable is void” 错误。但是不要急着加载 font-lock，因为这个 setq 只是为了将 lisp-font-lock-keywords-2 设置给 font-lock 的另一个变量 lisp-font-lock-keywords ，而这只有当 font-lock 由于什么原因加载的时候才会用到。所以我们使用 eval-after-load 来保证 setq 不会发生的太早而引起错误。

如果你调用 eval-after-load 而文件已经被加载会发生什么呢？那么后面的 Lisp 表达式会马上执行。如果同一个文件有多个 eval-after-load 会发生什么呢？它们会在文件加载时一个接一个的全部执行。

你可能发现 eval-after-load 的工作方式和钩子变量很相似。这是对的，但是一个重要的区别是钩子只执行 Lisp 函数（通常为 lambda 表达式的形式），而 eval-after-load 可以执行任何 Lisp 表达式。

5.5 局部变量列表

本章前面的内容已经足够创建并且根据需要加载 Lisp 代码文件了。但是对于 timestamp 的例子，事情还是有些不同。当调用 update-writestamps 时会自动载入 timestamp，但是谁来调用 update-writestamps 并且加载 timestamp 呢？回想一下前一章中 update-writestamps 是由 local-write-file-hooks 调用的。那么如何将 update-writestamps 放进 local-write-file-hooks 里呢？而因为前面的章节创建 Lisp 文件中所提到的副作用，我们一定不能在加载文件时这么做。

我们需要一种方法将 update-writestmpas 加入到需要它的 buffer 的局部变量 local-write-file-hooks 里，这样当 local-write-file-hooks 第一次触发时就会自动加载 timestamp。

一种很好的完成这个需求的手段是使用文件尾部的局部变量列表（local variables list）。当 Emacs 访问一个新文件的时候，它会扫描文件的尾部是否有这样的文本块[1]：

```
Local variables:
var1: value1
var2: value2
...
End:
```

当 Emacs 找到这样一个块的时候，它会将每个 `value` 赋给对应的 `var`，并且使其变为 buffer 的局部变量。Emacs 甚至能解析出以某个前缀开头的这种块，只要它们的前缀相同。而在 Lisp 代码文件中必须将其作为注释，这样它们就不会被当做是 Lisp 代码执行：

```
; Local variables:
; var1: value1
; var2: value2
; ...
; End:
```

`values` 被当做引用来对待；它们在赋给对应的 `vars` 之前不会被计算。所以在包含下面这个块

```
; Local variables:
; foo: (+ 3 5)
; End:
```

的文件中 buffer 局部变量 `foo` 的值为 `(+ 3 5)`，而不是 `8`。

因此任何需要将 `update-writestamps` 加入到 `local-write-file-hooks` 中的文件都可以这样指定：

```
Local variables:
local-write-file-hooks: (update-writestamps)
End:
```

[1] “文件的尾部”的意思是：文件的最后 3000 个字节—是的，这很随意—直到最后一行以 CONTROL-L 开头的行，如果存在的话。

实际上，文件可以根据需要建立起自己所有的变量：

```
Local variables:
local-write-file-hooks: (update-writestamps)
writestamp-prefix: "Written:"
writestamp-suffix: "."
writestamp-format: "%D"
End:
```

使用这种方式设置 local-write-file-hooks 的一个问题是它会将 local-write-file-hooks 替换为上面例子中所示的一个新的列表，而不是一种更好的方式—保留原来 local-write-file-hooks 里的其他值并向其中添加 update-writestamps 。虽然这样做需要执行 Lisp 代码。即，你需要执行下面的表达式：

```
(add-hook 'local-write-file-hooks 'update-writestamps)
```

Emacs 在局部变量列表中引入一个“伪变量（pseudovariable）” eval 来完成这个功能。当

```
eval: value
```

出现在局部变量列表中时，value 将被计算。计算的结果将被忽略；它将不会存储到局部变量 eval 中。因此完整的解决方案是将

```
eval: (add-hook 'local-write-file-hooks 'update-writestamps)
```

添加到局部变量里。

实际上，设置 local-write-file-hooks 的正确做法应该是编写一个子模式（minor mode），这将是第七章的主题。

5.6　补充：安全考虑

局部变量列表是一个潜在的安全漏洞，会导致用户受到“特洛伊木马”类型的攻击。例如一个变量的设置使得 Emacs 的工作不正常；或者一个 eval 有一些不可预知的副作用，例如删除文件或者使用你的名字伪造邮件。而攻击者只要引诱你访问一个在局部变量列表中包含这些设置的文件就可以了。只要你访问了这个文件，这些代码就会被执行。

保护你自己的方式是将

```
(setq enable-local-variables 'query)
```

加入到你的.emacs 文件里。这将会使 Emacs 在执行任何局部变量列表时都会询问你。也可以使用 `enable-local-eval` 来控制伪变量 `eval` 的执行。

第六章　列表

在本章：

- 列表初探
- 列表细节
- 递归列表函数
- 迭代列表函数
- 其他有用的列表函数
- 破坏性列表操作
- 循环列表?!

目前为止，我们已经看到了一些列表（list）的使用，但是我们并没有真的去探索它们如何工作以及为什么它们如此有用。既然列表作为 Lisp 的核心内容，本章我们就来全面的观察一下这个数据结构。

6.1　列表初探

就像我们已经看到的，Lisp 中的列表就是一个括号包裹起来的 0 个或多个 Lisp 表达式的序列。列表可以嵌套；也就是说括号里的子表达式还可以包含一个或多个列表。下面是一些例子：

```
(a b c) ; 三个符号组成的列表
(7 "foo") ; 一个数字和字符串组成的列表
((4.12 31178)) ; 列表只有一个元素：一个两个数字组成的自列表
```

空列表 () 等价于符号 `nil` 。

函数 `car` 和 `cdr` [1]用来访问列表的一部分：`car` 得到列表的第一个元素，`cdr` 得到列表剩下的元素（除第一个元素之外）。

[1]读作 “could-er”。这些名称是最初 Lisp 设计时电脑架构的历史遗留。

```
(car'(abc)) -> a
(cdr'(abc)) -> (b c)
(car (cdr '(a b c))) -> b
```

(回忆一下引用（quote）表达式—可能是一个完整的列表—表示按照字面来解释表达式。所以 '(a b c) 表示列表包含 a 、b 、和 c ，而不是用 b 和 c 作为参数调用 a)

只包含一个元素的列表的 cdr 为 nil ：

```
(cdr '(x)) -> nil
```

空列表的 car 和 cdr 都为 nil ：

```
(car '()) -> nil
(cdr '()) -> nil
```

注意对于只包含 nil 的列表也是如此：

```
(car '(nil)) -> nil
(cdr '(nil)) -> nil
```

但这并不表明 () 等价于 (nil) 。

对于这些你并不需要完全只听信我的言语。你只需要像第一章中执行 Lisp 表达式章节中所描述的那样，到 *scratch* buffer 里自己尝试执行这些语句。

列表通过函数 list ， cons 以及 append 创建。函数 list 可以使用任意数量的参数构建列表：

```
(list 'a "b" 7) -> (a "b" 7)
(list '(x y z) 3) -> ((x y z) 3)
```

函数 cons 使用一个 Lisp 表达式和一个列表作为参数。它通过将表达式添加到列表的前面构成新列表：

```
(cons 'a '(3 4 5)) -> (a 3 4 5)
(cons "hello" '()) -> ("hello")
(cons '(a b) '(c d)) -> ((a b) c d)
```

注意对列表使用 cons 构建新列表并不会影响之前的列表：

```
(setq x '(a b c)) ;将(a b c)赋值给变量x
(setq y (cons 17 x)) ;cons 17给它并且赋值给y
y -> (17 a b c) ;正常工作
x -> (a b c) ;并不会改变x
```

函数 append 使用任意数量的列表作为参数，并将其顶层元素连接成一个新列表。这会高效的去掉每个列表的外层括号，把剩下的元素放到一起，然后使用一对新的括号把它们括起来：

```
(append '(a b) '(c d)) -> (a b c d)
(append '(a (b c) d) '(e (f))) -> (a (b c) d e (f))
```

函数 reverse 使用一个列表作为参数，将其顶层元素反转成为一个新的列表。

```
(reverse '(a b c)) -> (c b a)
(reverse '(1 2 (3 4) 5 6)) -> (6 5 (3 4) 2 1)
```

注意 reverse 并不会反转子列表中的元素。

6.2　列表细节

这一章节将解释 Lisp 中列表的内部工作机制。既然大部分 Lisp 程序都会不同程度的使用列表，理解它们的工作机制是很有益处的。这会让你理解列表擅长做什么、不擅长什么。

列表由更小的称为 cons cells 的数据结构构成。cons cell 由两个 Lisp 表达式构成，你可能并不惊讶如何访问它们—使用 car 和 cdr。

函数 cons 使用它的两个参数创建一个新的 cons cell。不像前一小节中所讲的，cons 的两个参数可以是任意 Lisp 表达式。第二个参数不能是一个已存在的列表。

```
(cons 'a 'b) -> 一个由car a和cdr b组成的cons cell
(car (cons 'a 'b)) -> a
(cdr (cons 'a 'b)) -> b
```

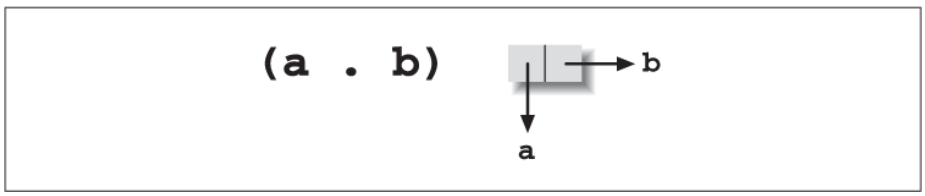

图 6.1: (cons 'a 'b) 的结果

生成的 cons cell 如图6.1所示。

当你将一些其他元素与一个列表执行 cons 时，例如

```
(cons 'a '(b c))
```

结果是 (a b c)，也就是一个 car 为 a，cdr 为 (b c) 的 cons cell。后面会更详细的讲述它。

对于 cdr 不是列表的 cons cell 有一种特殊的语法。它被称为 dotted pair notation，而 cons cells 有时也被称为 dotted pairs:

```
(cons a b) -> (a . b)
(cons '(1 2) 3) -> ((1 2) . 3)
```

当如图6.2所示的那样，一个 cons cell 的 cdr 为 nil 时，可以省略掉点号和 nil 。

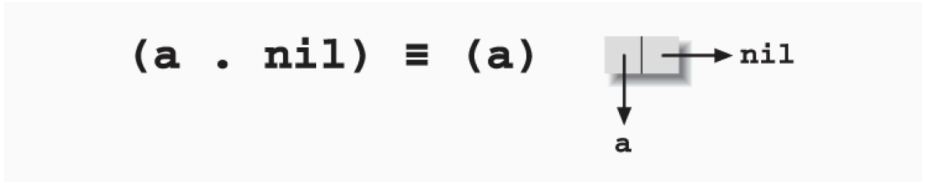

图 6.2: 一个单元素 list: (a)

另一个省略的规则是当 cons cell 的 cdr 是另一个 cons cell 时，那么点号以及包裹 cdr 的括号都可以省略。见图6.3。

当把这条规则和前一条忽略 cdr 为 nil 的规则组合起来的时候，我们就会发现下面的列表我们已经很熟悉了: (a . (b . nil)) ≡ (a b . nil) ≡ (a b)。

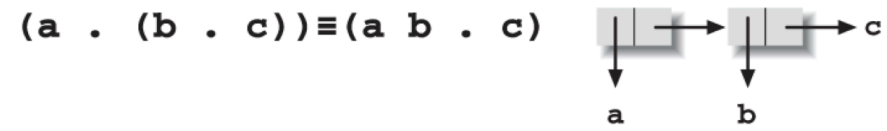

图 6.3: 一个 cons cell 指向另一个

通常来说，Lisp 的列表是一个由 cons cells 组成的链表，每个 cell 的 `cdr` 是另一个 cell，最后一个 cell 的 `cdr` 为空。cons cells 的 `car` 是什么并不重要。图6.4展示了一个列表作为另一个列表的一部分存在。

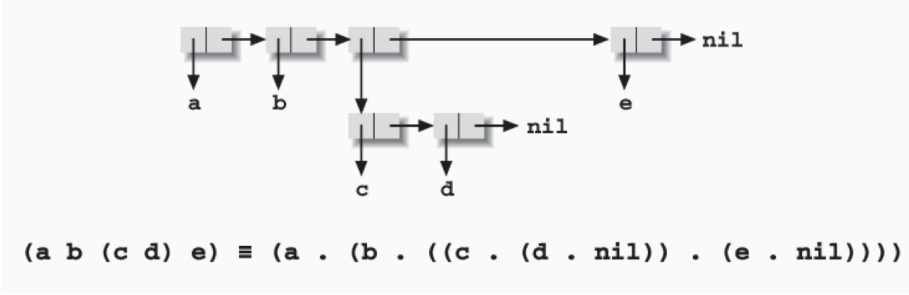

图 6.4: 一个列表包含一个子列表

当你编写

```
(setq x '(a b c))
```

这会将 `x` 指向这个由三个 cons cell 组成的链表的第一个 cell。如果你编写

```
(setq y (cdr x)) ; 现在y是(b c)
```

这会将 `y` 指向上面列表中的第二个 cons cell。一个列表事实上只是一个指向 cons cell 的指针。

最后一个 `cdr` 不为 `nil` 的列表有时被称作 improper list。通常 association list 总是 improper lists。

有多个函数用来检测一个 Lisp 对象是列表还是组成列表的一部分。

`consp` 检测它的参数是不是一个 cons cell。对于 `(consp x)`，当 `x` 为除空表之外的任何列表时返回 `true`，其他返回 `false`。

`atom`　检测它的参数是否为原子。(`atom x`) 功能与 (`consp x`) 相反—任何不是 cons cell 的元素，包括 `nil` 、数字、字符串以及符号都是原子。

`listp`　检测它的参数是否为列表。对于 (`listp x`) ，如果 `x` 为 cons cells 或者 `nil` 则返回 `true` ，其他返回 `false` 。

`null`　检测它的参数是否为 `nil` 。

现在你已经知道了 cons cells，你可能会觉得 (`car nil`) 和 (`cdr nil`) 都定义为 `nil` 很奇怪，因为 `nil` 甚至不是一个 cons cell，因此它也没有 `car` 和 `cdr` 。实际上，有一些 Lisp 方言在你对 `nil` 调用 `car` 和 `cdr` 时会报错。大多数 Lisps 的行为跟 Emacs Lisp 一样，主要是为了方便—但是这个特例会造成一些奇怪的副作用，就像上面提到的，() 和 (`nil`) 在 `car` 和 `cdr` 的时候的结果是一样的。

6.3　递归列表函数

传统的 Lisp 教材使用一系列简短的编程练习来阐明列表和 cons cells 的行为。让我们花一点时间看一下两个广为人知的例子，然后再往下进行。

在这个练习中我们的目标是定义一个名为 `flatten` 的函数，将指定的列表的所有内部的子列表都释放出来平铺到一层上。例如：

```
(flatten '(a ((b) c) d)) -> (a b c d)
```

解决方案是使用递归，将 `car` 和 `cdr` 分别平铺，然后将他们合并到一层上来。假如输入的列表为

```
((a (b)) (c d))
```

它的 `car` 是 (`a (b)`) ，平铺之后是 (`a b`) 。`cdr` 是 (`(c d)`) ，平铺之后的结果是 (`c d`) 。函数 `append` 可以将 (`a b`) 和 (`c d`) 组合起来并且保持平铺，结果是 (`a b c d`) 。所以 `flatten` 的核心代码是：

```
(append (flatten (car lst))
        (flatten (cdr lst)))
```

(我将 `lst` 作为 `flatten` 的参数名称。不能使用 `list` ，因为它是一个 Lisp 函数的名称) 现在，`flatten` 只能对列表工作，所以对于 (`flatten (car lst)`) ，如果 (`car lst`) 不是一个列表的话将会报错。我们因此需要这么改进：

```
(if (listp (car lst))
    (append (flatten (car lst))
            (flatten (cdr lst))))
```

这个 `if` 没有“else”分支。如果 `(car lst)` 不是列表怎么办？例如，假设 `lst` 为

```
(a ((b) c))
```

`car` 不是一个列表。这时，我们只要简单的平铺 `cdr` ，`(((b) c))` ，得到 `(b c)` ；然后用 `cons` 将 `car` 组装上去。

```
(if (listp (car lst))
    (append (flatten (car lst))
            (flatten (cdr lst)))
  (cons (car lst)
        (flatten (cdr lst))))
```

最后，我们需要一个方法来终止这个递归。在处理列表越来越小的分片的递归函数里，你能用来作为结束分片的最小分片是 `nil` ，而 `nil` 几乎总是作为这种函数的“默认选择”。在本例中，平铺 `nil` 的结果就是 `nil` ，所以完整的函数定义为

```
(defun flatten (lst)
  (if (null lst) ; lst是nil吗？
      nil ; 是的话，返回nil
    (if (listp (car lst))
        (append (flatten (car lst))
                (flatten (cdr lst)))
      (cons (car lst)
            (flatten (cdr lst))))))
```

试着在 *scratch* buffer 里用这个函数处理一些列表，并且试着通过一些例子来理清函数逻辑。记住 Lisp 函数的返回值是其最后执行的表达式的值。

6.4　迭代列表函数

递归并不总是列表相关编程问题的正确解决方案。有时朴实直接的迭代也是需要的。在本例中，我们将会展示 Emacs Lisp 每次处理列表中一个元素的语

法风格，有时这也被称为列表的“cdr-ing down”（因为每次迭代，列表都会因取其 cdr 而缩短）。

假设我们需要一个用来计数列表中符号个数，并且跳过像数字、字符串和子列表等其他元素的函数。这个递归函数是错误的：

```
(defun count-syms (lst)
  (if (null lst)
      0
    (if (symbolp (car lst))
        (+ 1 (count-syms (cdr lst)))
      (count-syms (cdr lst)))))
```

递归—特别是深度递归—引入了非常多的额外资源来记录嵌套函数的调用和返回值，而这些应该尽量避免。而且，这个问题用迭代的方式解决显然更合理，而代码通常应该反映出解决问题的合理方式，而不是自作聪明地将解决问题地方式复杂化。

```
(defun count-syms (lst)
  (let ((result 0))
    (while lst
      (if (symbolp (car lst))
          (setq result (+ 1 result)))
      (setq lst (cdr lst)))
    result))
```

6.5　其他有用的列表函数

下面是其他一些 Emacs 定义的列表相关函数。

length　返回列表的长度。对于 improper list 它不会工作。

```
(length nil) -> 0
(length '(x y z)) -> 3
(length '((x y z))) -> 1
(length '(a b . c)) -> error
```

nthcdr　对列表调用 n 次 cdr 。

```
(nthcdr 2 '(a b c)) -> (c)
```

nth　返回列表的第 n 个元素（第一个元素序号为 0）。这与 nthcdr 的 car 等价。

```
(nth 2 '(a b c))-> c
(nth 1 '((a b) (c d) (e f))) -> (c d)
```

mapcar　使用一个函数和一个列表作为参数。它对列表包含的每个元素都调用一次函数，即将列表里的元素作为参数传给那个函数。mapcar 的返回值是一个包含对每个元素调用函数之后的列表。所以如果你有一个字符串列表而你想要让其中的字符串首字母大写的话，可以这么写：

```
(mapcar '(lambda (x)
           (capitalize x))
        '("lisp" "is" "cool")) -> ("Lisp" "Is" "Cool")
```

equal　检测它的两个参数是否相等。它与第三章中的章节保存和取出 point中介绍的 eq 并不相同。不像 eq 判断它的参数是否为同一个对象，equal 判断的是两个对象是否具有相同的结构和内容。这个区别很重要。例如：

```
(setq x (list 1 2 3))
(setq y (list 1 2 3))
```

x 和 y 是两个不同的对象。也就是说，第一次调用 list 创建了一个包含三个 cons cells 的链表，而第二次创建了另外一个包含三个 cons cells 的链表。所以 (eq x y) 值为 nil ，即使两个列表实际上包含着相同的结构和内容。也因此，(equal x y) 为 true 。在 Lisp 编程中，每当你希望判断两个对象是否相等时，你都需要注意调用 eq 还是 equal 更合适。另一点需要注意的是 eq 是一个瞬发操作，而 equal 可能需要递归比较两个参数的内部结构。注意下面的 eq 值为 true 。

```
(setq x (list 1 2 3))
(setq y x)
(eq x y)
```

assoc　会帮助你以键值的方式使用列表。当列表的形式为

```
((key1 . value1)
 (key2 . value2)
 ...
 (keyn . valuen))
```

被称为 association list，或者简写为 assoc list[2]。函数 `assoc` 会找到列表中第一个的 `car` 为指定参数的子列表。所以：

```
(assoc 'green
       '((red . "ff0000")
         (green . "00ff00")
         (blue . "0000ff"))) -> (green . "00ff00")
```

如果没有匹配的子列表，`assoc` 返回 `nil` 。这个函数使用 `equal` 来检测每个键 `keyn` 是否匹配输入参数。另一个函数，`assq` ，功能与 `assoc` 相同但是使用 `eq` 来做匹配。有些程序员不喜欢使用 dotted pairs，所以他们建立的字典看起来可能不是这样的：

```
((red . "ff000")
 (green . "00ff00")
 (blue . "00ff"))
```

而是这样的：

```
((red "ff0000")
 (green "00ff00")
 (blue "000ff"))
```

这没问题，因为对于 `assoc` 来说，列表中的每个元素仍然为 dotted pair：

```
((red . ("ff0000"))
 (green . ("00ff00"))
 (blue . ("0000ff")))
```

唯一的区别是在前面的例子里，assoc list 中的每一项都只需要储存在一个单独的 cons cell 里，而现在需要两个。而在前面的列表中获取与 key 匹配的值时只需要这么做：

```
(cdr (assoc 'green ...)) -> "00ff00"
```

而现在必须这么做：

```
(car (cdr (assoc 'green ...))) -> "00ff00"
```

[2]我一直找不到统一的读法到底应该是 a-SOAK，a-SOASH 或者 a-SOCK list。这三种我都听到过。有些人会将其读作“a-list”来避免这个问题。

6.6　破坏性列表操作

目前为止，我们所看到的所有列表操作都是非破坏性的。例如，当你把一个对象 cons 到一个已存在的列表上时，结果是产生了一个全新的 cons cell，它的 cdr 指向了原来未做改动的列表。任何其他引用之前列表的对象或变量都未受影响。同样的，append 会创建一个新列表以及新 cons cells 来保存参数中列表的元素。它不会将 x 最后的 cdr 指向 y，或者将 y 最后的 cdr 指向 z，因为这样的话最后的 nil 指针就改变了。而这样的话就影响了 x 和 y 原来的使用。实际上 append 对这些列表分别创建了一个未命名的拷贝，如图6.5所示。注意 z 不需要拷贝；append 总是直接使用最后一个参数[3]。

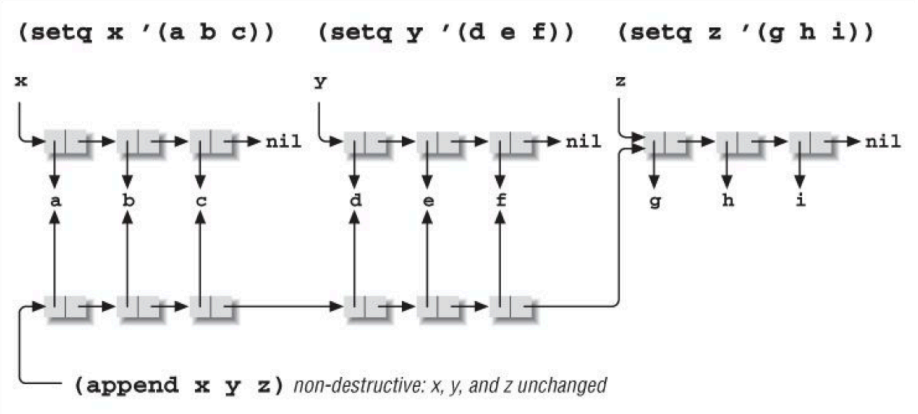

图 6.5: append 函数不会影响它的参数

下面是非破坏性的 append 在 Lisp 代码中的表示：

```
(setq x '(a b c))
(setq y '(d e f))
(setq z '(g h i))
(append x y z) -> (a b c d e f g h i)
```

因为 append 并不会修改它的参数，所以这些变量储存的仍然是之前的值：

```
x -> (a b c)
y -> (d e f)
z -> (g h i)
```

[3]因为是直接指向的，所以 append 的最后一个参数甚至不用是一个列表！自己试试看。

但是如果做出了破坏性的修改，那么每个变量都会指向 append 时制作出的长链表的一部分，如图6.6所示。执行破坏性 append 的函数称为 nconc 。

```
(nconc x y z) -> (a b c d e f g h i)
x -> (a b c d e f g h i)
y -> (d e f g h i)
z -> (g h i)
```

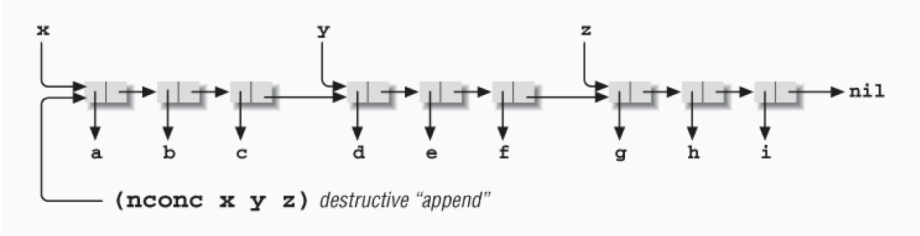

图 6.6: 不像 append，nconc 会影响它的参数

通常破坏性的修改列表并不明智。很多其他的变量和数据结构可能正在使用你修改的列表，所以最好不要修改它以致造成不可预知的影响。

另一方面，有时你确实希望破坏性的修改一个列表。可能你希望利用 nconc 的高效并且你确实地知道没有其他代码会因为列表的改变而受到影响。

使用破坏性操作的最常见的一个场景是改变 assoc list 中的值。例如，假如你有一个对应保存着人员名称和它们 email 的 assoc list：

```
(setq e-addrs
      '(("robin" . "rl@sherwood.uk")
        ("marian" . "mf@sherwood.uk")
        ...))
```

现在假设有人的 email 地址改变了。你需要这样来更新它：

```
(setq e-addrs (alist-replace e-addrs "john" "johnl@exile.fr"))
```

而 alist-replace 实际上是一个非常低效地递归操作，它的机制是重新拷贝整个列表：

```
(defun alist-replace (alist key new-value)
  (if (null alist)
```

```
    nil
  (if (and (listp (car alist))
           (equal (car (car alist))
                  key))
      (cons (cons key new-value)
            (cdr alist))
    (cons (car alist)
          (alist-replce (cdr alist) key new-value)))))
```

不仅仅是低效（特别是当输入很大时），而且有可能你确实希望改变任何引用这个数据结构的对象和变量。显然，alist-replace 并没有改变原数据结构。它创建了一个全新的拷贝，而任何引用老数据的对象都没有得到更新。以代码来表示这种情况就是：

```
(setq alist '((a . b) (c . d))) ; alist 是一个 assoc list.
(setq alist-2 alist) ; alist-2 指向了同一个列表
(setq alist (alist-replace alist 'c 'q)) ; alist 是一个新列表
alist -> ((a . b) (c . q)) ;alist 响应了改动
alist-2 -> ((a . b) (c . d)) ;alist-2 仍然指向之前的列表
```

这里引入 setcar 和 setcdr [4]。给出一个 cons cell 和一个新值，这两个函数会将 cell 的 car 或者 cdr 替换为新值。例如：

```
(setq x (cons 'a 'b)) -> (a . b)
(setcar x 'c)
x -> (c . b)
(setcdr x 'd)
x -> (c . d)
```

我们现在可以轻松的编写 alist-replace 的破坏性版本了：

```
(defun alist-replace (alist key new-value)
  (let ((sublist (assoc key alist)))
    (if sublist
        (setcdr sublist new-value))))
```

[4]也称为 rplaca 和 rplacd，跟 car 和 cdr 的历史原因相同。

这会查找 alist 的子列表中谁的 car 与 key 匹配—例如，("john" . "jl@nottingham.co.uk")—并且将 cdr 替换为 new-value 。而由于这会改变原数据结构—也就是说，这并没有创建任何新的拷贝—所有引用这个 cons cell 的的变量和其他对象，特别是包含它的 assoc list，都会反映出这个改变。

还有另一个重要的破坏性列表操作：nreverse ，reverse 的非拷贝版本。

```
(setq x '(a b c))
(nreverse x) -> (c b a)
x -> (a)
```

为什么上面的例子中最后 x 等于 (a) 呢？这是因为 x 仍然指向同一个 cons cell，在前面的操作中已经倒转过来了。(a b c) 由三个 cons cells 组成，car 分别为 a 、b 、c 。一开始，x 是通过指向链表的第一个 cons cell 引用列表的—它的 car 为 a 而 cdr 指向下一个 cons cell（也就是包含 b 的那个 cell）。但是在 nreverse 之后，所有 cons cells 的 cdrs 都变了。现在 car 为 c 的 cons cell 变为了链表的第一个元素，而它的 cdr 变成了包含 b 的 cons cell。同时，x 的值却没变：它仍然指向之前的 cons cell，也就是 car 为 a 的 cell。但是现在这个 cell 由于变成了链表的末尾，所以 cdr 却变成了 nil 。因此，x 等价于 (a) 。

如果你需要 x 也适应列表的改变，那么你必须这么写

```
(setq x (nreverse x)) -> (c b a)
```

6.7　循环列表 ?!

由于我们可以破坏性地修改我们创建的列表，我们就可以不受只用预定义元素构建列表的限制。列表可以引用自己的一部分！例如：

```
(setq x '(a b c))
(nthcdr 2 x) -> (c)
(setcdr (nthcdr 2 x) x) ;先不要这么做!
```

这个例子会发生什么呢？开始我们创建了一个包含三个元素的列表并且将其赋给 x 。然后我们通过 nthcdr 找到最后一个 cons cell。最后，我们将这个 cell 的 cdr 替换为 x —也就是这个列表中的第一个 cell。现在这个 list 变成环了：之前的列表的尾巴指回了头部。

这个列表长什么样呢？好吧，它的开头看起来是这样的：

```
(a b c ab c ab c a b c a b c a b c a b c a b c . . .
```

而这永远不会停止。我在上面写“先不要这么做！”的原因是如果你在 *scratch* buffer 里执行这段代码的话，Emacs 将会试着去显示结果—而这永远不可能完成。这将会进入一个死循环，虽然你可以用 C-g 终止这个过程。现在你可以去试试了，当然在 Emacs 卡死之后尽快按下 C-g 。你等的时间越久，*scratch* buffer 中填充的 a b c 就越多。

显然，打印并不是环状结构能把 Emacs 搞得无限循环的唯一一件事。任何迭代执行这个列表里所有元素的动作都不会终止。下面是一个很好的例子：

```
(eq x (nthcdr 3 x)) -> t ; 第三个cdr与x指向同一个对象
(equal x (nthcdr 3 x)) -> ; 永不停止!
```

既然循环列表会导致 Emacs 进入无限循环，那它有什么用呢？通常我们都不会想让列表变为环状，但是如果你不将其认为是列表，而是相互连接在一起的 cons cells 的话，你就可以构建任何种类的链表结构了，比如树和 lattices 。有些数据结构是自引用的，例如环。如果你曾经需要构建这类数据结构的话，你就不会被 Emacs 可能会为了显示它而造成无限循环这件事吓倒了。不要在需要展示结果的情况下使用它就可以了。例如，如果你将上面的 setcdr 改为下面这样

```
(setqx '(a b c))
(progn
  (setcdr (nthcdr 2 x) x)
  nil)
```

那么 Emacs 将不会尝试展示 setcdr 的结果，而现在 x 就是一个我们可以操作的但是却不用全部展示的环状数据结构了。

```
(nth 0 x) -> a
(nth 1 x) -> b
(nth 412 x) -> b
```

!

第七章　子模式

在本章：

- 段落填充
- 模式
- 定义子模式
- Mode Meat

有时我们希望扩展（extension）只影响某些特定类型的 buffer 而不是所有的 buffer。本章我们将通过思考这个问题来提高我们在 Emacs 编程中的灵活性。例如，你在 Lisp 模式下按下 `C-M-a` 会跳转到最近的函数定义，但是你不希望在编辑文档的时候也这样。Emacs 的“模式（mode）”机制使得 `C-M-a` 只会在 Lisp 模式下才产生效果。

关于 Emacs 中模式的相关主题是很复杂的。我们将以学习“子模式（minor modes）”来作为一个轻松良好的开始。在 buffer 中，子模式是可以与主模式共同存在的，它的作用是添加较少的一些特定功能的新行为。每个 Emacs 用户都对像 Lisp、C 以及 Text 这种主模式很熟悉，但是他们可能对于出现在模式栏（mode line）中的例如“Fill”这种表示自动填充的标识并不熟悉。

我们将基于 Emacs 自身的段落填充功能创建一个子模式。我们的子模式，Refill，将会在你编辑段落的时候进行动态填充。

7.1　段落填充

填充一个段落是将段落中所有行的长度变得适当的过程。每行的长度都应该大概相等并且不会越过右边距（right margin）。过长的行应该在词之间的空格处进行拆分。短行应该用后面行的文字进行填充。填充有时还包括左右对齐（justification），也就是通过在每一行添加空格来使得左右边距相等。

大多数现代文字处理软件都会保证段落的填充。每次文字修改时，段落中的文字会“流动”以完成正确的布局。一些 Emacs 的批评者指出 Emacs 在填充段落时的表现不如其他软件。Emacs 虽然提供了 `auto-fill-mode` ，但是这只在当前行生效，而且只有当超过“右边距”并且插入空格时才会触发。在删除字

符时并不会触发；除了当前行之外的行都不会被填充；并且在行的左边插入文字而使右边的文字超过右边距时也不会触发。

作为 Emacs 狂热者，对于像 neplus ultra 这样的编辑器的支持者们你可以给出下面的三个答复之一：

1. 像动态段落填充这种华丽的特性只能被用来掩饰这个软件其他不如 Emacs 的地方（你可以根据所需列出来）。
2. 你认为内容要比格式更重要，所以不需要自动的段落填充，当你觉得自己需要时，只需要简单的按下 `M-q` 来触发 `fill-paragraph` 就好了。
3. 做一点简单的 Lisp hacking，Emacs 就可以像别的程序那样完成动态段落填充了（然后你也可以问一下他们的编辑器是否也能模仿 Emacs 的这种行为）。

本章是关于选择 3 的。

为了确保当前段落一直正确地填充，我们需要在每次执行插入和删除后进行一次检查。这在计算上可能开销很大，所以我们希望能够对其进行开关；由于并不是所有 buffer 都需要这个功能，因此当它打开时，我们只希望它在当前 buffer 生效。

7.2　模式

Emacs 使用模式这个概念来封装一系列编辑行为。换句话说，使用不同的模式，Emacs 在 buffers 里的表现是不同的。举一个小例子，在 Text 模式中 TAB 键插入一个 ASCII 的制表符，而在 Emacs Lisp 模式中这将会通过插入或者删除空格来将代码缩进到正确的列上。再举一个例子，当你在 Emacs Lisp 模式的 buffer 里执行命令 `indent-for-comment` 时，你将会得到一个以 Lisp 注释符 ";" 开头的空注释。而当你在 C 模式的 buffer 里，你得到的是 C 语言的注释`/* */`。

每个 buffer 总是属于一个主模式（major mode）。主模式指定了 buffer 用于某种特定类型的编辑行为，例如 Text、Lisp 或者 C。名为 Fundamental 的主模式并不为特定类型的编辑存在，你可以认为它是一种空模式。通常 buffer 的主模式的选择是根据你访问的文件的名称，或者 buffer 中的一些设置进行的。你可以通过执行模式的命令来改变主模式，例如 `text-mode` 、`emacs-lisp-mode` 、或者 `c-mode` 。[1]当你这么做之后，buffer 就使用新的主模式替换之前的模式了。

与此不同的是，子模式向 buffer 里添加一系列功能而并不完全改变 buffer 原本的编辑方式。子模式可以与主模式以及其他子模式单独打开关闭。buffer 除了主模式之外还可能在 0、1、2、3 或者多个子模式之下。举几个子模式的例子：

[1]除了我列出的这几个外还有很多其他的主模式。他们能用来编辑 HTML 文件、LATEX 文件、ASCII 文件、troff 文件、二进制文件、目录等等等等。而且，主模式也用来实现许多例如新闻阅读或者网页浏览这种非编辑特性。试着输入`M-x finder-by-keyword RET`来浏览 Emacs 具有的模式和其他插件。

auto-save-mode，使 buffer 在编辑的时候每隔一段时间就存储到特定名称的文件里(当系统崩溃时这些缓存文件就可以避免编辑的丢失);font-lock-mode，根据当前 buffer 的语法以不同颜色显示文本(如果显示器支持);line-number-mode，在 buffer 的模式栏里显示当前编辑的行号（在底部)。

通常，如果一个包需要在不同的 buffer 中分别打开与关闭，那么它就应该被实现为子模式而不是主模式。这与我们上一部分中所描述的段落填充的需求是一致的，因此可以知道我们的段落填充功能应该是一个子模式。我们将在第九章中再关注主模式的实现。

7.3　定义子模式

在定义子模式时需要有下面这些步骤。

1. 选择一个名字。我们的模式名称为 refill。
2. 定义一个名为 name-mode 的变量。使其成为 buffer 局部的。buffer 的子模式在这个变量的值为非空的情况下表示打开，否则关闭。

```
(defvar refill-mode nil
  "Mode variable for refill minor mode.")
(make-variable-buffer-local 'refill-mode)
```

3. 定义一个名为 name-mode 的命令。[2]这个命令应该具有一个可选参数。如果不提供参数，它将打开或关闭模式。如果提供参数，且参数的 prefix-numeric-value 大于 0 则打开模式，否则关闭模式。也就是说，C-u M-x name-mode RET 总是执行打开，而 C-u - M-x name-mode RET 总是关闭模式（查看第二章中的补充：原始的前置参数获得更多信息)。下面是开关 Refill 模式的命令定义：

```
(defun refill-mode (&optional arg)
  "Refill minor mode."
  (interactive "P")
  (setq refill-mode
  (if (null arg)
      (not refill-mode)
          (> (prefix-numeric-value arg) 0)))
  (if refill-mode
      code for turning on refill-mode
      code for turning off refill-mode))
```

[2]函数和变量的名称可以相同；它们不会冲突。

`setq` 语句看起来有些奇怪，但这在子模式定义中是一种常见的格式。如果 `arg` 为 `nil`（没有前置参数），它会将 `refill-mode` 设置为 `(not refill-mode)` —也就是 `refill-mode` 之前值的相反值，`t` 或者 `nil` 。否则，它将 `refill-mode` 设置为

```
(> (prefix-numeric-value arg) 0)
```

的值，当 `arg` 的值为大于 0 的数字时为 `t` ，否则为 `nil` 。

4. 向 `minor-mode-alist` 中添加一项，它是一个这种形式的 assoc list（查看第六章中其他有用的列表函数章节）：

```
((model string1)
 (mode2 string2)
 ...)
```

新的项会将 `name-mode` 关联到一个将会在 buffer 的模式栏中使用的短字符串。模式栏（mode line）是每个 Emacs 窗口底部的信息栏；它会显示每个 buffer 的主模式名称以及其他处于激活状态的子模式名称，以及其他的一些信息。描述子模式的短字符串应该以空格开头，因为它会追加到信息栏的模式相关部分。下面的例子展示了对于 Refill 模式该如何做：

```
(if (not (assq 'refill-mode minor-mode-alist))
    (setq minor-mode-alist
          (cons '(refill-mode " Refill")
                minor-mode-alist)))
```

（最外层的 if 保证了 `(refill-mode " Refill")` 不会二次添加到 `minor-mode-alist` 里，例如当两次加载 refill.el。）这让使用了 refill-mode 的 buffer 的模式栏看起来是这样的：

```
--**-Emacs: foo.txt (Text Refill) --L1--Top---
```

在定义子模式时还有一些其他步骤在这个例子中没涉及。例如，子模式可能有一个 `keymap` ，一个与之关联的语法表 (syntax table)，或者一个 `abbrev` 表，但是因为 `refill-mode` 用不到，我们这里暂且忽略。

7.4　Mode Meat

现在我们有了基本结构，让我们开始定义 Refill mode 的内容。

我们已经弄清了 refill-mode 的基本特性：每次插入和删除都必须保证当前段落的正确缩进。当 buffer 改变时触发代码执行的正确做法，你可以回想一下第四章，就是当 refill-mode 激活时向钩子变量 after-change-functions 添加一个函数 (关闭时移除)。我们将会添加一个 refill 函数 (还未定义) 来确保当前段落会被正确缩进。

```
(defun refill-mode (&optional arg)
  "Refill minor mode."
  (interactive "P")
  (setq refill-mode
        (if (null arg)
            (not refill-mode)
          (> (prefix-numeric-value arg) 0)))
  (make-local-hook 'after-change-functions)
  (if refill-mode
      (add-hook 'after-change-functions 'refill nil t)
    (remove-hook 'after-change-functions 'refill t)))
```

add-hook 和 remove-hook 后面的参数保证了我们修改的只是 buffer 局部的 after-change-functions 。不管在调用这个函数时 refill-mode 有没有打开，我们都调用 (make-local-hook 'after-change-functions) 来使其变为 buffer 局部的。这是因为在这两种情况—打开 refill-mode 或关闭—我们都需要对每个 buffer 单独操作 after-change-functions 。总是先调用 make-local-hook 是最简单的方式，而且如果一个钩子变量已经是 buffer 局部的，再次调用也没有什么副用。

现在剩下的事情就是定义 refill 函数了。

7.4.1 Naïve 的首次尝试

就像第四章中提到的，钩子变量 after-change-functions 有些特殊，因为其中的函数需要三个参数 (普通的钩子函数通常不需要参数)。三个参数指明了在 after-change-functions 执行之前，buffer 的改变发生的地方。

- 改变开始处，称为 start
- 改变结束处，称为 end
- 影响的文本长度，称为 len

start 和 end 指向 buffer 改变发生之后的位置。len 指向与改变发生之前相比影响的文本长度。在插入发生之后，len 为 0(因为并不影响之前 buffer 中的文本)，而新插入的文本在 start 和 end 之间。在删除发生之后，len 为删除

的文本的长度，文本已经被丢掉，而 start 和 end 为同一个数字，因为删除文本使它们指向了同一处，也就是删除内容的两端合二为一了。

现在我们知道了 refill 的参数应该是什么，我们可以做出一个朴素的尝试来对其进行定义：

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (fill-paragraph nil))
```

这种实现是非常不严谨的，因为每次按键都调用 fill-paragraph 代价太大了！它还有一个问题，就是每次向行尾添加一个空格时，fill-paragraph 都会把它立即删除—它会在缩进的时候自动把尾部空格删除掉—因此，当你打字的时候，你将会花费最多的时间在行尾，唯一向两个单词间插入空格的方式就是先把两个单词打出来，likethis，然后向其中插入一个空格。但是这个尝试证明了我们的理论，并且给了我们一个可以对其进行改进的起点。[3]

7.4.2　限制 refill

要优化 refill ，让我们先对问题进行一下分析。首先，是否每次整个段落都需要重排?

不。当插入和删除文本时，只有被影响的行和下面的行需要重排。前面的行并不需要。如果插入文本，行可能会变得太长，有些文本会挤入下一行 (这可能会导致下一行也变得太长，因此这个过程是会重复的)。如果文本被删除，文本可能会变得太短，后续的行需要拿出一些文本来填补 (这可能会导致下一行变得太短，因此这个过程也是会重复的)。所以变化并不会影响前面的行。

实际上，有一种情况的变化会影响前面一行。考虑下面这一段：

```
Glitzy features like on-the-fly filling of paragraphs are
needed only to hide the programs's many inadequacies
compared to Emacs
```

假设我们删除第三行开头处的”compared”：

```
Glitzy features like on-the-fly filling of paragraphs are
needed only to hide the programs's many inadequacies
to Emacs
```

[3]有的读者可能已经会指出当调用 fill-paragraph 的时候会改变 buffer，而这会导致 after-change-functions 再次执行，再次递归的触发 refill 并且可能会导致无限循环，或者说是无限递归。说的不错，但是 Emacs 会在 after-change-functions 中的函数执行时重置它来避免这个问题。

单词”to” 现在可以移动到上一行的末尾处，就像这样：

```
Glitzy features like on-the-fly filling of paragraphs are
needed only to hide the programs's many inadequacies to
Emacs
```

前面的例子应该可以告诉你前面的一行也需要重排—并且只有当前的行的第一个词被缩短或者删除的时候才会出现。

所以我们可以将段落重排操作限制为当前行，可能会影响前一行，以及后续的行。我们不使用 `fill-paragraph` ，因为它会自己判断段落边界，相反的我们自己选择” 段落边界”，然后使用 `fill-region` 。

我们为 `fill-region` 选择的边界应该包含段落中整个受影响的部分。对于插入，左边界就是简单的 `start` ，也就是插入的点，右边界是当前段落的结尾。对于删除，左边界是前一行的开始 (也就是，包含 `start` 的前一行)，右边界是行末尾。所以下面就是我们新的 `refill` 函数的概要：

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if this is an insertion
                  start
                  beginning of previous line))
        (right end ofparagraph))
    (fill-region left right ...)))
```

对于插入，完善这个函数是很简单的。之前说过，调用 `refill` 时，`len` 为 0 则表示插入，非 0 则表示删除。

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (zerop len) ; len是否为0?
                  start
                beginning of previous line))
        (right end ofparagraph))
    (fill-region left right ...)))
```

要计算前一行的开始，我们先要把光标移动到 `start` ，然后将光标移动到前一行的末尾 (很奇怪，这可以通过 (`beginning-of-line 0`) 来得到) ，然后使用 (`point`) 来得到这个值，所有这些都放在 `save-excursion` 里：

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (zerop len)
                  start
                (save-excursion
                  (goto-char start)
                  (beginning-of-line 0)
                  (point))))
              (right end ofparagraph))
        (fill-region left right ...)))
```

我们可以对段落的结束采用类似的计算方式，但是我们可以更方便的利用 fill-region 的特性：它将为我们找到段落结尾。fill-region 的第五个参数(有两个必要参数和三个可选参数)，如果非空，将会告诉 fill-region 一直重排到下一段之前。所以实际上我们并不需要计算 right 。

我们新版本的 refill 还没完成。我们必须首先解决 fill-region 会将光标放置到影响区域的末尾的问题。显然每次输入都把光标移动到段落末尾是不可接受的！将 fill-region 的调用包装在 save-excursion 里会解决这个问题。

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (zerop len)
                  start
                (save-excursion
                  (goto-char start)
                  (beginning-of-line 0)
                  (point))))
        (save-excursion
          (fill-region left end nil nil t)))))
```

(fill-region 的第二个参数被忽略了，因为我们使用了它找寻段落结尾的特性。我们传递 end 只是因为这很方便而且对于读者来说并不是完全无意义的。)

7.4.3　小调整

好的，上面的只是基本的想法，还剩下许多事情要做。例如，当计算 left 时，如果前一行已经不在这个段落那么就没有必要再计算前一行了。所以我们应该得到行的开始以及前一行的开始，然后使用更大的那个值。

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (zerop len)
                  start
                (max (save-excursion
                       (goto-char start)
                       (beginning-of-line 0)
                       (point))
                     (save-excursion
                       (goto-char start)
                       (backward-paragraph 1)
                       (point))))))
    (save-excursion
      (fill-region left end nil nil t))))
```

(函数 max 会返回参数里更大的那个。)

现在我们有三个地方调用了 save-excursion，而这是个代价有点大的函数。更好的做法是将其中两个合并在一起然后计算两个需要的值。

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (zerop len)
                  start
                (save-excursion
                  (max (progn
                         (goto-char start)
                         (beginning-of-line 0)
                         (point))
                       (progn
                         (goto-char start)
                         (backward-paragraph 1)
                         (point)))))))
    (save-excursion
      (fill-region left end nil nil t))))
```

下一步，回想我们关于重排前一行的观察:" 前面的一行也需要重排—并且只有当前行的第一个词被缩短或者删除的时候才会出现。" 但是在我们的代码

里，我们在删除的时候每次都会计算前一行。让我们看看在删除发生在非第一个词之外的地方时能否避免这个计算。

我们可以通过将下面的代码

```
(if (zerop len)
    start
  find previous line)
```

修改为

```
(if (or (zerop len)
        (not (before-2nd-word-p start)))
    start
  find previous line)
```

来实现。before-2nd-word-p 是一个用来告诉它的参数，一个 buffer 位置，是否出现在第二个单词之前的函数。

现在我们必须写出 before-2nd-word-p 。它应该找出当前行的第二个单词的位置，并且跟它的参数进行比较。

如何才能找到行中的第二个单词呢?

我们可以到行的开始，然后调用 forward-word 来跳过第一个单词。这个方法的问题是我们得到的是第一个单词的末尾，而非第二个单词的开头，它们之间可能有很多空格。

我们可以到行的开始，然后调用 forward-word 两次 (实际上，我们可以调用 forward-word 一次，传入参数 2)，然后调用 backward-word ，这就会把我们置于第二个单词的开头。这不错，但是我们认识到 forward-word 和 backward-word 定义的” word ” 跟我们需要的定义并不相同。根据这些函数，标点符号 (例如破折号) 会分开单词，所以 (例如)” forward-word ” 是两个单词。这对我们来说并不好，因为我们的函数只认为被空格分割才算两个单词。

我们可以到行的开始，然后跳过所有非空格的字符 (第一个单词)，然后跳过所有空格字符 (第一个单词之后的空格)，然后我们就在第二个单词的开头了。这听起来好一些；让我们试一下。

```
(defun before-2nd-word-p (pos)
  "Does POS lie before the second word on the line?"
  (save-excursion
    (goto-char pos)
    (beginning-of-line)
```

```
    (skip-chars-forward "^ ")
    (skip-chars-forward " ")
    (< pos (point))))
```

函数 skip-chars-forward 非常实用。它会向前移动光标，直到遇到一个你所指定的字符集里包含或者不包含的字符。字符集的工作方式跟正则表达式中的方括号语法一样 (参考第四章中的正则表达式中的规则 3). 所以

```
(skip-chars-forward "^ ")
```

表示” 跳过不是空格的所有字符”，而

```
(skip-chars-forward " ")
```

表示” 跳过所有空格”。
这个方式的一个问题就是当一行里没有空格时，

```
(skip-chars-forward "^ ")
```

会直接跳到下一行!我们不希望这样。所以我们通过向第一个 skip-chars-forward 添加一个换行符来确保我们不会略过太多：

```
(skip-chars-forward "^ \n") ; 跳到第一个空格或者换行符
```

另一个问题是有时 tab(” \t ”) 制表符也有可能用来像空格一样分割单词。所以我们必须这样来修改我们的两个 skip-chars-forward 调用：

```
(skip-chars-forward "^ \t\n")
(skip-chars-forward " \t")
```

还有其他的类似空格和制表符一样的被认为是空格的字符吗？也许有。换页符 (ASCII 12) 通常被认为是空格。而如果 buffer 使用了非 ASCII 的编码，有可能还有一些其他的字符会被认为是分隔单词的空格。例如，对于 Latin-1 这样 8 位字符编码，字符数字 32 和 160 都是空格—虽然 160 表示” 非折断空格”，即行不应该在此处折断。

与其我们关心这些细节，为什么不让 Emacs 自己判断呢？这就是语法表 (syntax tables) 发挥作用的时候了。语法表是一个与模式关联的将字符对应到”

语法类别 (syntax classes)" 的映射表。类别包括"word constituent"(通常包括单词、省略号，有时包括数字)，"balanced brackets"(通常为 () , [] , {} , 有时包括 <>)，"comment delimiters"(对于 Lisp mode 来说就是 "; "，对于 C mode 则为 /* 和 */)，"punctuation"，以及当然的，"whitespace"。

语法表被一些像 forward-word 和 backward-word 这样的函数用来找出一个词的类别是什么。因为不同的 buffer 有不同的语法表，同一个词的的定义可能会各有不同。我们将会使用语法表来找出在当前 buffer 中哪些字符被认为是空格。

我们所需要做的就是将我们两次的 skip-chars-forward 调用替换为 skip-syntax-forward ，就像这样：

```
(skip-syntax-forward "^ ")
(skip-syntax-forward " ")
```

对于每个语法类别，都有一个对应的 code letter。[4]空格是"whitespace" 的 code letter，所以上面的两行表示" 跳过所有非空格" 和" 跳过所有空格"。

不幸的是，前面的 skip-syntax-forward 调用也有跳到下一行的问题。更坏的是，这次我们不能简单的将 \n 添加到 skip-syntax-forward 的参数里，因为 \n 并不是换行符语法类别的 code letter。实际上，在不同 buffer 里的换行字符的 code letter 是不同的。

我们能做的是请求 Emacs 告诉我们换行字符的 code letter 是什么，然后使用这个结果来构建 skip-syntax-forward 的参数：

```
(skip-syntax-forward (concat "^ "
                             (char-to-string
                              (char-syntax ?\n))))
```

函数 char-syntax 会返回字符的 code letter。然后我们使用 char-to-string 将其转换为一个字符串并且添加到" ^ "。

这是 before-2nd-word-p 的最终形态：

```
(defun before-2nd-word-p (pos)
  "Does POS lie before the second word on the line?"
  (save-excursion
    (goto-char pos)
    (beginning-of-line)
```

[4]要了解更多关于语法表的细节，执行 describe-functions 查看 modify-syntax-entry 。

```
    (skip-syntax-forward (concat "^ "
                                 (char-to-string
                                  (char-syntax ?\n))))
    (skip-syntax-forward " ")
    (< pos (point))))
```

记住计算 before-2nd-word-p 的代价可能会超过它本身想提供的好处 (即，在 refill 中避免调用 end-of-line 和 backward-paragraph)。如果你感兴趣的话，你可以试着使用性能分析器 (参见附录C) 来查看哪个版本的 refill 更快，是使用 before-2nd-word-p 的还是不使用的。

7.4.4　排除不希望的重排

在每次插入发生的时候我们并不需要重排段落。一个微小的并不会将任何文本推到右边界的插入并不会影响除它之外的任何其他行，所以如果当前改变是一次插入，并且 start 和 end 在同一行，并且行的末尾并没有超过右边界，那么我们没有必要调用 fill-region 。

这意味着我们需要把 fill-region 的调用包裹在一个 if 里，如下所示：

```
(if (and (zerop len) ; 如果是插入...
         (same-line-p start end) ; ...并且没有跨行
         (short-line-p end)) ; ...并且行仍然够短
    nil ; 那么什么都不做
  (save-excursion
    (fill-region ...))) ; 否则，refill
```

我们现在必须定义 same-line-p 和 short-line-p 。

现在看来编写 same-line-p 应该很简单。我们只需要简单的检测 end 是否在 start 和行尾之间就可以了。

```
(defun same-line-p (start end)
  "Are START and END on the same line?"
  (save-excursion
    (goto-char start)
    (end-of-line)
    (<= end (point))))
```

编写 short-line-p 也差不多同样明了。用于控制右边界的变量称为 fill-column ，而 current-column 返回一个点的横座标。

```
(defun short-line-p (pos)
  "Does line containing POS stay within 'fill-column'?"
  (save-excursion
    (goto-char pos)
    (end-of-line)
    (<= (current-column) fill-column)))
```

下面是 refill 的新的定义：

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (or (zerop len)
                      (not (before-2nd-word-p start)))
                  start
                (save-excursion
                  (max (progn
                         (goto-char start)
                         (beginning-of-line 0)
                         (point))
                       (progn
                         (goto-char start)
                         (backward-paragraph 1)
                         (point)))))))
    (if (and (zerop len)
             (same-line-p start end)
             (short-line-p end))
        nil
      (save-excursion
        (fill-region left end nil nil t)))))
```

7.4.5　尾空格

我们还是没有解决 fill-region 会删除每行最后尾部空格的问题，也就是当你进行编辑的时候，你需要输入 likethis，然后将光标移动到中间再插入一个空格！

我们的策略是当光标在行末空格的后面，或者光标在行末空格之间的时候不进行 refill 。这个条件可以被实现为

```
(and (eq (char-syntax (preceding-char))
         ?\ )
     (looking-at "\\s *$"))
```

当光标前的字符为空格而光标后面只有空格的时候为真。让我们仔细看一下它。

首先我们计算 `(char-syntax (preceding-char))`，这将会得到光标前面的字符的语法类别，然后跟 `'?\'` 进行比较。这个奇怪的结构—问号，斜杠，空格—是 Emacs Lisp 中书写空格字符的方式。回想空格字符是”whitespace” 语法类别的 code letter，所以这个是用来检测前面的字符是否为空格的。

下一步我们调用 `looking-at`，一个用来检测光标后的文本是否符合一个给定的正则表达式的函数。这个例子里的正则表达式是 `\s *$` (之前说过，在 Lisp 字符串里斜杠需要加倍)。在 Emacs Lisp 正则表达式里，`\s` 表示引入基于当前 buffer 语法表的一个语法类别。`\s` 之后的字符表示使用哪个语法类别。在这个例子里，也就是空格，表示”whitespace”。所以 `'\s '` 表示” 匹配一个 whitespace 字符”，而 `\s *$` 表示” 匹配 0 个或多个 whitespace 字符，直到行末尾”。

我们为 `refill` 的最后版本添加上这个检测。

```
(defun refill (start end len)
  "After a text change, refill the current paragraph."
  (let ((left (if (or (zerop len)
                      (not (before-2nd-word-p start)))
                  start
                (save-excursion
                  (max (progn
                         (goto-char start)
                         (beginning-of-line 0)
                         (point))
                       (progn
                         (goto-char start)
                         (backward-paragraph 1)
                         (point)))))))
    (if (or (and (zerop len)
                 (same-line-p start end)
                 (short-line-p end))
            (and (eq (char-syntax (preceding-char))
                     ?\ )
```

```
                (looking-at "\\s *$")))
        nil
      (save-excursion
        (fill-region left end nil nil t)))))
```

考虑到效率因素，通常应该避免将函数放到 `after-change-hooks` 里，特别是像 `refill` 这种复杂的函数。如果你的电脑够快，你可能注意不到每次按键执行这个函数的时间消耗；否则，你可能会发现你的 Emacs 变得反应缓慢。在下一章，我们将会找到一种方式来加速它。

第八章　求值和纠错

在本章：

- 有限版本的 save-excursion
- eval
- 宏函数
- 反引用和去引用
- 返回值
- 优雅的失败
- 点标记

在前面的章节里，我们注意到 `save-excursion` 是一个消耗比较大的函数，而我们尝试在 `refill` 函数中尽量少的调用它 (因为每次 buffer 改变时都会执行，所以我们需要它尽可能的快)。然而，在 `refill` 中仍然包含了五次对于 `save-excursion` 的调用。

我们可以尝试将 `save-excursion` 的使用进行合并—例如，可以将 `refill` 的整个函数体用 `save-excursion` 包起来，将里面其他的 `save-excursion` 都删掉，重新编写里面的代码来确保光标能够正确的放置。但是这会影响到我们代码布局的清晰性。当然，在某些情况下为了优化可以牺牲掉代码的清晰性，但是在我们考虑合并 `save-excursion` 之前，让我们看看我们是否还有别的方法。我们可以寻找另一个更适合的函数来替代它。

在本章中我们将会尝试编写一个更快的，有限版本的 `save-excursion` 。我们将会遇到很多有趣的有着一个共同目的的特性：控制表达式何时执行以及它们对于周围代码的影响。我们将会关注这些返回值相关的问题以及在发生错误时如何清理。我们将会看到如何使 Lisp 解释器延后对于表达式的求值，直到你希望它做为止。我们甚至能找到更改函数执行的顺序的方法。

8.1　有限版本的 save-excursion

`save-excursion` 的目的是在执行完一些 Lisp 表达式之后恢复”point” 的值；但这并不是它所有的能力。它也会恢复”mark” 的值，恢复 Emacs 对于哪

个 buffer 是当前 buffer 的认知。对于 `refill` 来说这过于强大了；毕竟，我们只改变了 point 的值。我们并没有切换 buffer 或者移动 mark。

我们可以编写一个仅仅满足我们需求的缩减版本的 `save-excursion` 。也就是说，我们需要编写一个接收任何数量 Lisp 表达式作为参数的函数，做如下事情：

1. 记录 point 的位置
2. 按顺序对每个子表达式求值
3. 将 point 恢复到初始值

我们遇到的第一个问题是在执行 Lisp 函数时，它的参数会在这个函数获得控制权之前执行。换句话说，如果我们编写一个名为 `limited-save-excursion` 的函数并且这么调用它：

```
(limited-save-excursion
 (beginning-of-line)
 (point))
```

那么调用顺序如下：

1. (`beginning-of-line`) 执行，point 移动至当前行的开始并且返回 nil。
2. (`point`) 执行，返回光标移动到的位置。
3. `limited-save-excursion` 执行，参数为刚才函数返回的值—也就是 nil 和一个数字。

在这个例子里，`limited-save-excursion` 无法记录子表达式执行之前的 point 位置；而计算出来的 nil 和光标的位置很显然也并没有什么用。

8.2　eval

我们可以通过引用来绕过这个问题：

```
(limited-save-excursion
 '(beginning-of-line)
 '(point))
```

这一次 `limit-save-excursion` 的参数是 (`beginning-of-line`) 和 (`point`) 。它能够记录 point 的值，正确的按顺序执行子表达式，然后恢复 point 并且返回。它大概可以实现为下面的样子。

```
(defun limited-save-excursion (&rest exprs)
  "Like save-excursion, but only restores point."
  (let ((saved-point (point))) ; 记录point
    (while exprs
      (eval (car exprs)) ; 执行下一个参数
      (setq exprs (cdr exprs)))
    (goto-char saved-point))) ; 恢复point
```

这个函数有一些新的东西：即 `eval` 的调用，它使用一个 Lisp 表达式作为参数然后执行它。乍看起来这可能并没有什么新鲜的，因为毕竟 Lisp 解释器能够自动执行 Lisp 表达式，而不必额外的调用 `eval` 。但是有时表达式执行的返回值是另一个你想要执行的 Lisp 表达式，而 Lisp 本身并不会自动执行这个表达式。如果我们只执行（`car exprs`），我们将会提取出列表中的第一个子表达式，然后就把它丢掉了！我们需要显式调用 `eval` 使那个表达式能为我们所用。

下面是一个用来展示 Emacs 通常的执行行为以及 `eval` 的必要性的简单例子：

```
(setq x '(+ 3 5))
x -> (+ 3 5) ; 对x求值
(eval x) -> 8 ; 对x的值求值
```

8.3　宏函数

虽然 `limited-save-excursion` 在给参数加上引用符之后就正常工作了，但是这对于调用者来说很繁琐，而且这会导致它不能成为 `save-excursion` 的合格替代者 (毕竟 `save-excursion` 并没有这个限制)。

我们可以实现一种称为宏函数 (macro function)[1]的特殊函数，它的参数都是默认被引用的。也就是说，当宏函数执行时，它的参数在它获得控制权之前都是不会被执行的。作为替代的，宏函数会产生一些值，通常是其参数的重新排列，然后这些值会被执行。

这里有一个简单的例子。假设我们想要一个称为 `incr` 的函数，它的作用是将一个数值变量增加 1。我们希望它有这种行为：

```
(setq x 17)
(incr x)
x -> 18
```

[1]不要把宏函数与键盘宏混淆，也就是 Emacs 的名字（“editor macros”）的来源。

如果 incr 是一个普通函数，那么它的参数将会是 17，而不是 x ，因此也不会影响 x 的值。所以 incr 必须是一个宏函数。它的输出必须是一个表达式，当这个表达式执行的时候就会对其参数里引用的变量的值加 1。

宏函数使用 defmacro 定义 (语法跟 defun 类似)。incr 的写法如下：

```
(defmacro incr (var)
  "Add one to the named variable."
  (list 'setq var (list '+ var 1)))
```

宏函数的函数体是对于入参的一个展开式。然后这个展开式会被求值。(incr x) 的展开式是：

```
(setq x (+ x 1))
```

当对这个表达式求值的时候，x 将会被加一。

你可以使用函数 macroexpand 函数来调试宏函数。这是一个把 Lisp 表达式作为输入，返回它的宏展开结果的函数。如果输入的表达式不是一个宏，那么将会返回原表达式。所以：

```
(macroexpand '(incr x)) -> (setq x (+ x 1))
```

8.4　反引用和去引用 (Backquote and Unquote)

既然 limited-save-excursion 必须是一个宏函数，我们所要做的就是想象出 limited-save-excursion 如何展开。让我们开始：

```
(limited-save-excursion
 subexpr1
 subexpr2
 ...)
```

它需要被展开成

```
(let ((orig-point (point)))
  subexpr1
  subexpr2
  ...
  (goto-char orig-point))
```

然后我们要将其编写成宏函数：

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  (append '(let ((orig-point (point))))
          subexprs
          '((goto-char orig-point))))
```

回忆之前讲过的 append 是将每个列表的括号剥掉，然后将他们组合在一起，最后将一个新的括号包在结果的外面。所以这个 append 的参数为三个列表：

```
(let ((orig-point (point))))
(subexpr1 subexpr2 ...)
((goto-char orig-point))
```

剥掉他们最外面的括号：

```
let ((orig-point (point)))
subexpr1 subexpr2 ...
(goto-char orig-point)
```

然后将结果包在新的括号里：

```
(let ((orig-point (point)))
  subexpr1
  subexpr2
  ...
  (goto-char orig-point))
```

这就是宏的展开式，然后再对其求值。

这就能完成我们的需求了，但是阅读理解宏定义是很困难的一件事。幸运的是，还有更好的办法。看起来几乎所有的宏都会调用 list 和 append 这种函数来重新组合他们的参数，一些表达式会被括起来而另一些不会。实际上，这是如此常见，以至于 Emacs Lisp 提供了一个特殊表达式来模板化地编写宏扩展。

记得 'expr 吗，它会展开成 (quote expr)？好吧，还有一个 `expr，它将会展开成 (backquote expr)。[2]反引用 (backquote) 跟引用 (quote) 很像，即对反引用表达式求值的结果仍然是表达式本身：

```
`(a b c) -> (a b c)
```

但是有一个重要的区别。一个反引用的列表的子表达式可以各自独立的使用去引用符 (unquoted) 进行修饰。即当反引用表达式求值时，其中的去引用子表达式也会被求值—而列表中其他的子表达式仍然保持引用状态！

```
`(a ,b c) -> (a value-of-b c)
```

要理解这为什么有用，让我们回到 incr 的例子。我们可以这么重写 incr：

```
(defmacro incr (var)
  "Add one to the named variable."
  `(setq ,var (+ ,var 1)))
```

每个逗号表示子表达式被去引用，所以在这个例子里，一个这种列表：

```
(setq ... (+ ... 1))
```

其中 var 的值（某个变量名）被插入了两次。结果跟我们第一个版本的 incr 相同，但是这一次表达的如此清晰。

将反引用和去引用应用到 limited-save-excursion 上并不能马上变得正确：

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  `(let ((orig-point (point)))
     ,subexprs ; 错啦!
     (goto-char orig-point)))
```

对于反引用还有一个细节需要学习。subexprs 是一个 &rest 的参数，他是一个包含着所有传递给 limited-save-excursion 的参数的列表。因此当它替换到上面的模板里面的时候，它也会是一个列表。换句话说，

[2]这个表达式是 Emacs 19.29 新引入的。在之前的版本里，它们必须像函数调用一样触发，例如：(` expr)。

```
(limited-save-excursion
 (beginning-of-line)
 (point))
```

将会展开为：

```
(let ((orig-point (point)))
  ((beginning-of-line)
   (point))
  (goto-char orig-point))
```

而这会造成语法错误，因为有括号多余了。我们需要的是一种将 `subexprs` 中的值提取到一个列表中，并且移除外面括号的方法。为此，Emacs Lisp 提供了另一个特殊语法（最后一个，我保证）：拼接去引用操作符 (splicing unquote operator)，`,@` 。这个版本：

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  `(let ((orig-point (point)))
     ,@subexprs
     (goto-char orig-point)))
```

将会获取到正确的结果：

```
(let ((orig-point (point)))
  (beginning-of-line)
  (point)
  (goto-char orig-point))
```

8.5　返回值

要完成 `limited-save-excursion` 我们还有很多事情要做。比如，它并没有返回 `subexprs` 的最后一个表达式，而 `save-excursion` 会。`limited-save-excursion` 返回了并没有什么帮助的（`goto-char orig-point`）的值，也就是 `orig-point` 的值，因为 `goto-char` 会返回它的参数。而当你希望使用这个值的时候，这显然是不正确的：

```
(setq line-start (limited-save-excursion
                   (beginning-of-line)
                   (point))
```

为了修复这个问题，我们必须记录最后一个表达式的值，然后恢复 point，然后返回之前储存起来的值。我们可能会这么做：

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  `(let ((orig-point (point))
         (result (progn ,@subexprs)))
     (goto-char orig-point)
     result))
```

注意到 `progn` 的使用，它的作用是执行每个传递给它的参数然后返回最后一个参数的值—这正是我们的宏所希望的。但是，这个版本因为两个原因是错误的。第一个原因跟 `let` 的工作机制有关。当下面这个表达式执行时：

```
(let ((var1 val1)
      (var2 val2)
      ...
      (varn valn))
  body ...)
```

所有 `vals` 会在任何 `vars` 赋值之前执行，所以没有 `val` 能引用到 `var` 。而且，它们执行的顺序也是随机的。所以，如果我们使用上面版本的 `limited-save-excursion` 来将

```
(limited-save-excursion
 (beginning-of-line)
 (point))
```

扩展为

```
(let ((orig-point (point))
      (result (progn (beginning-of-line)
                     (point))))
```

```
    (goto-char orig-point)
    result)
```

那么很有可能，当对这个表达式求值时，beggining-of-line 会先于写在前面的 point 执行，而这会导致 orig-point 的值的错误。

对于这个问题的解决方法是使用 let* 代替 let 。当使用 let* 时，就没有了这种不确定性：vals 的执行顺序就是它们在代码中所写的顺序。[3]而且，每个 var 都会在对应的 val 求值之后马上赋值，所以 vali 可以引用从 var1 到 vari-1 之间的值。

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  `(let* ((orig-point (point))
          (result (progn ,@subexprs)))
     (goto-char orig-point)
     result))
```

下一个问题的修复就没这么简单了。假设子表达式中使用了全局变量 orig-point 。就像我们刚刚提到的，每个 val 都可以访问到前面的 vars ，所以如果子表达式中引用了 orig-point ，它将会因此引用到 limited-save-excursion 中定义的那个 orig-point 局部变量—这几乎可以肯定不是子表达式的作者所希望使用的。宏展开的子表达式会使用这个变量。这会对子表达式的编写造成很大的困扰，因为它所希望操作的完全是另一个变量。而假如这些子表达式又对 orig-point 的值进行了修改，这反过来又会影响到 limited-save-excursion 自身。

我们将子表达式的执行放入定义了 orig-point 局部变量的 let* ，却因此将子表达式“真正”希望使用的 orig-point 给隐藏起来了。

你可能会想到规避这个问题的一个好方法是为 orig-point 挑选一个不大可能出现在子表达式中的其他名称。这并不是一个令人满意的解决方案，因为 (a) 不管你定义的变量名如何特殊，总是有可能会发生重复，(b) 况且这件事有正确的解决方法。正确的方法是产生一个肯定不会与其他在使用的变量产生冲突的新变量。那么如何做呢?

要回答这个问题，我们首先需要理解两个符号发生冲突表示什么。两个符号只有在表示同一个对象时才会冲突，而不仅仅是名字相同。当你向 Lisp 程序中输入一个符号名时，Lisp 解释器在内部会将其转化为一个符号对象。符号对

[3]如果 let 如此模糊而 let* 这么清晰，那么为什么不只用 let* 呢? 答案是：let 在某些情况下跟高效。而且，有时你就是会需要在任何 vars 存在之前就计算出所有的 vals 。通常，你应该使用 let 除非你确实需要 let* —当然你应该能够想到，不恰当地选择使用它们是常见的程序异常来源之一。

象包含着比它的名字更多的信息。它包含着这个符号的局部和全局的变量绑定关系；它包含着任何与这个符号绑定的函数定义 (使用 defun)；以及包含着符号的属性列表 (参照第三章的符号属性部分)。

将编写的 Lisp 代码转换成像符号对象 (或者 cons cell 等) 这种内部数据结构的过程称为 reading。当 Lisp" 解释器" 两次看到同一个符号名时，它并不会创建两个内部符号对象—它会重用同一个。

可能你看出来我们需要怎么做了：如果我们能够获取到一个其他的符号对象，而不是通过 Lisp 自己的内部符号和重用机制，那么 Lisp 就不会认为它跟其他符号对象是同一个，即使它们有着同样的名字。创建这种符号的方法是通过函数 make-symbol ，它使用符号的名称 (一个字符串) 来创建一个新的，非内部的，保证与其他对象都不同的对象。

换句话说，

```
(make-symbol "orig-point")
```

将不会与任何其他地方出现过的 orig-point 冲突。新创建的 orig-point 与其他之前创建的对象都不同。

在你想避免与其他变量引用冲突的时候，使用新的、非内部的符号是一种很安全的做法。下面是我们函数的改进版本：

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  (let ((orig-point-symbol (make-symbol "orig-point")))
    `(let* ((,orig-point-symbol (point))
            (result (progn ,@subexprs)))
       (goto-char ,orig-point-symbol)
       result)))
```

第一个 let 创建了一个名为 orig-point 的新符号对象，并且不与任何其他符号相同，包括同样名为 orig-point 的对象。这个新的对象被赋值给 orig-point-symbol ，然后在后面的反引用模板里 (通过去引用) 使用了两次。

乍看起来,我们只是将 orig-point 冲突的危险转换为了 orig-point-symbol 的危险。但是 orig-point-symbol 实际上并不会出现在宏的展开式里，展开式看起来大概是这样的 (orig-point ' 代表了使用 make-symbol 创建的非内部的符号):

```
(let* ((orig-point' (point))
       (result (progn subexprs)))
```

```
  (goto-char orig-point')
  result)
```

所以在 subexprs 执行的时候—在宏展开之后—唯一的临时变量是 orig-point ’，而这是唯一的。临时变量 result 这时还不存在。所以变量冲突的问题彻底解决了。

8.6　优雅的失败

当 Emacs 中发生错误时，当前的计算将会终止而 Emacs 会返回到上层的主循环，在那里它会等待按键或者其他输入。当执行 limited-save-excursion 子表达式发生错误时，整个 limited-save-excursion 将会在调用 goto-char 之前终止，而 point 的值将会变为未知的一个值而不会恢复。但是真正的 save-excursion 即使在错误发生时也可以正确的恢复 point(以及 mark 和当前 buffer)。这是怎么做到的?

调用函数的信息被存在一个称为栈 (stack) 的内部数据结构里。错误发生之后，回到顶层主循环将会弹出这个栈，以相反的顺序每次弹出一个函数调用—所以如果 a 调用了 b ，b 调用了 c ，然后错误发生了，c 将会弹出，然后是 b ，然后是 a ，直到 Emacs 回到” 顶层”。

在栈弹出时执行编写的 Lisp 代码是可能的！这是编写” 优雅的” 失败处理的关键，使得我们可以在函数自己由于一些错误 (或者用户自己触发 C-g) 而无法完成时对其进行清理。我们要使用的函数称为 unwind-protect ，它会正常执行输入的第一个表达式，后面跟着任意数量需要后续执行的表达式—即使异常打断了第一个表达式的执行。它看起来是这样的:

```
(unwind-protect
    normal
  cleanup1
  cleanup2
  ...)
```

显然，我们需要将对于 point 值的恢复行为放到 unwind-protect 的” 清理” 部分:

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  (let ((orig-point-symbol (make-symbol "orig-point")))
    `(let ((,orig-point-symbol (point)))
```

```
      (unwind-protect
          (progn ,@subexprs)
        (goto-char ,orig-point-symbol)))))
```

`unwind-protect` 的一个好的特性是在非错误的情况下，它的返回值是” 正常” 表达式的值 (如果错误发生了，返回值并没有意义)。在这个例子里，也就是 `(progn ,@subexprs)` ，正是我们希望的 `limited-save-excursion` 的返回值，所以我们可以移除掉之前的 `result` 变量，并且将 `let*` 改回 `let` 。

8.7　点标记

在最后对于 `limited-save-excursion` 的改进里，我们将会把 point 记录为一个标记，而非一个数字，就像我们在 `unscroll` 中所定义的那样 (参照第三章标记部分)：也就是说，对于子表达式的执行可能会使保存的 buffer 位置变得不准确，因为文本可能已经被插入或删除掉了。

要做的修改非常简单。将返回数值的 point 调用，替换为将当前位置表达为一个标记的 `point-marker` 就可以了。

```
(defmacro limited-save-excursion (&rest subexprs)
  "Like save-excursion, but only restores point."
  (let ((orig-point-symbol (make-symbol "orig-point")))
    `(let ((,orig-point-symbol (point-marker)))
       (unwind-protect
           (progn ,@subexprs)
         (goto-char ,orig-point-symbol)))))
```

现在剩下的就是将这个定义存入一个名为 limited.el 的文件，后面加上

```
(provide 'limited)
```

然后放入一个 `load-path` 中存在的路径并且对其进行字节编译 (参考第五章)。然后在 refill.el 中我们可以把 `save-excursion` 的调用替换为 `limited-save-excursion` ；在 refill.el 的开头添加

```
(require 'limited)
```

然后对其字节编译。这样当 `refill` 载入的时候才会载入 `limited` ，并且如果你将

```
(autoload 'refill-mode "refill" "Refill minor mode." t)
```

添加到你的.emacs 中，那么直到你触发 `refill-mode` 时才会载入 `refill`。

第九章　主模式

在本章：

- 我的 Quips 文件
- 主模式框架
- 改变段落的定义
- Quip 命令
- 键位表
- Narrowing
- 继承模式

编写一个简单的主模式跟子模式非常像，就像我们在第七章中所看到的那样。在本章中我们将只接触到主模式的基础知识，为创建更复杂的主模式做好准备—实际上，一个全新的应用—在下一章。

9.1　我的 Quips 文件

多年来我一直从互联网上收集一些风趣的谚语，并且使用老牌 UNIX 程序 fortune 所使用的格式将它们存储在一个称为 Quips 的文件里。每个谚语都会使用一行包含 `%%` 的行开始。下面是一个例子：

```
%%
I like a man who grins when he fights.
- Winston Churchill %%
%%
The human race has one really effective weapon, and that is laughter.
- Mark Twain
```

除了 `%%` 的行，其他内容的格式都是没有限制的。

在编辑了我的 Quips 文件一段时间后，我发现编辑它跟编辑普通文本文件有一些区别。例如，我经常需要将我的编辑限制到一个单独的谚语里以防止意

外影响到邻近的谚语。另一点，每当我需要在一个谚语的开头插入一个段落的时候，我首先需要将它跟开头的 %% 之间插入一个空行。否则，%% 就变成了段落的一部分：

```
%%
I like a man who grins when he fights.
- Winston Churchill
%%The human race has one really effective weapon, and that is laughter.
- Mark Twain
```

插入空行告诉 Emacs" %% " 并不是段落的一部分。在填充完段落之后，我会将文本追加到 %% 的后面并且删除空行。

很显然我需要一个新的编辑模式，一个能够避免这么多不方便的模式。问题是，这应该是一个主模式还是子模式呢？前面说过主模式不能跟其他主模式共存，而子模式可以与主模式以及其他激活的子模式独立的激活和关闭。在这个例子中，对于编辑模式的需求来源于数据格式本身，因此我们需要一个主模式而不是子模式。这种数据格式的文件总是需要这个而不是别的主模式。例如，你不会希望使用一个编辑 Lisp 的主模式再额外使用一个编辑谚语的子模式。[1]

9.2　主模式框架

定义主模式有如下步骤。

1. 选择一个名称（name）。我们模式的名称为 quip。
2. 创建一个与名称同名的 name.el 文件来保存模式相关的代码。
3. 创建一个称为 name-mode-hook 的钩子变量。它用来管理进入这个模式的时候用户要执行的钩子函数。

```
(defvar quip-mode-hook nil
  "*List of functions to call when entering Quip mode.*")
```

4. 如果需要的话，定义一个模式关联的键位表（keymap）（参照本章后面的键位表章节）。将它赋值给一个名为 name-mode-map 的变量。像下面这样定义一个模式的键位表：

```
(defvar name-mode-map nil
  "Keymap for name major mode.")
(if name-mode-map
```

[1] 在其他的一些情况下选择主模式还是子模式就没这么清晰了。

```
  nil
(setq name-mode-map (make-keymap))
(define-key name-mode-map keysequence command)
...)
```

如果不使用 make-keymap ，你还可以使用 make-sparse-keymap ，它更适合只包含较少键绑定的键位表。

5. 如果需要的话，定义一个模式关联的语法表（参照第七章的小调整章节）。将它赋值给一个名为 name-mode-syntax-table 的变量。
6. 如果需要的话，定义一个模式关联的缩写表（abbrev table）。将它赋值给一个名为 name-mode-abbrev-table 的变量。
7. 定义一个名为 name-mode 的命令。这是主模式命令，并且没有参数（跟子模式不同，那可以有一个可选参数）。当执行时，它会通过下面的步骤使当前 buffer 进入 name-mode ：
   (a) 它必须调用 kill-all-lcoal-variables ,这会移除掉所有 buffer-local 的变量。这会高效地关闭所有处于激活状态的主模式和子模式。

   ```
   (kill-all-local-variables)
   ```

   (b) 它必须将变量 major-mode 设置为 name-mode 。

   ```
   (seqt major-mode 'quip-mode)
   ```

   (c) 它必须将变量 mode-name 设置为一个用来描述这个模式的缩写字符串，这将会出现在 buffer 的模式栏里。

   ```
   (setq mode-name "Quip")
   ```

   (d) 如果键位表存在的话，它必须对其安装，这是通过将 name-mode-map 传递给 use-local-map 完成的。

   (e) 它必须通过向 run-hooks 传递 name-mode-hook 执行用户的钩子函数。

   ```
   (run-hooks 'quip-mode-hook)
   ```

8. 它必须通过 provide name” 提供” 这个代码所实现的特性（参照第五章中的章节以代码加载）。

```
(provide 'quip)
```

我们第一个版本的 Quip 模式将不会加入键位表，语法表以及缩写表，所以最初 quip.el 看起来是这样的：

```
(defvar quip-mode-hook nil
  "*List of functions to call when entering Quip mode.")
(defun quip-mode ()
  "Major mode for editing Quip files."
  (interactive)
  (kill-all-local-variables)
  (setq major-mode 'quip-mode)
  (setq mode-name "Quip")
  (run-hooks 'quip-mode-hook))
(provide 'quip)
```

这些只是基础，所有主模式都会实现这些。现在让我们开始向其中添加 Quip 模式的内容。

9.3　改变段落的定义

首先，我们必须使得包含 %% 的行不能被认为是段落的一部分。这意味着我们必须更改变量 paragraph-separate ，它的值是一个用来描述用于分隔段落的行的正则表达式。我们还要修改 paragraph-start ，一个用来描述段落开头或者分隔段落的行的正则表达式。[2]

Emacs 使用 paragraph-start 和 paragraph-separate 中的正则表达式来匹配行的开头，即使正则表达式中并不包含元字符 ^ (用来匹配行的开头)。

在 Text 模式中 paragraph-start 的值是"[\t\n\^L]"，这表示如果一行以空格、tab、新行[3]或者 Control-L(ASCII 中的"formfeed" 符) 开头，那么这行要么是段落的第一行，要么是分隔段落的行。

Text 模式中 paragraph-separate 的值是"[\t^L]*$"，这表示如果一行中包含 0 个或多个空格、tabs、formfeeds，或者一些他们的组合，并且没有其他字符，那么它不属于任何段落。

我们要做的是修改这些正则，使其认为" 一个包含着 %% 的行也是一个段落分隔符"。

第一步是使这些变量在 Quip 模式中的时候拥有不同的值。(也就是说，修改这些本来是全局变量的值的时候不会影响其他不在 Quip 模式中的值) 因此，除了在上一部分我们描述的基本框架之外，方法 quip-mode 还需要这么做：

[2]并没有专门用来匹配段落开始的正则变量。作为替代的，段落的开始就是符合 paragraph-start 但是不符合 paragraph-separate 的行。

[3]以一个新行 “开始” 的行当然是一个空行。

```
(make-local-variable 'paragraph-start)
(make-local-variable 'paragraph-separate)
```

下一步，quip-mode 需要设置 paragraph-start 和 paragraph-separate 的 buffer 局部的值。

```
(setq paragraph-start "%%\\|[ \t\n\^L]")
(setq paragraph-separate "%%$\\|[ \t\^L]*$")
```

paragraph-start 表示“%% 或者空格、tab、换行或者 control-L”。paragraph-separate 的值表示“只有 %% 或者只有 0 个或多个空格、tab 或者分页符”。具体查看第四章中的正则表达式章节。

9.4　Quip 命令

Quip 模式还需要做什么呢?

- 它应该允许用户每次向前或者向后移动一条谚语。
- 它应该允许用户只操作一条谚语。
- 它应该展示出文件中谚语的条目数，并且告诉用户当前光标下的谚语是第几条。
- 除此之外，它应该与 Text 模式大概相似。毕竟，操作的内容基本上就是纯文本。

让我们暂停一下,重新观察一下 Emacs 中不同的光标移动命令。forward-char 和 backward-char 每次移动一个字母。还有 forward-word 和 backward-word 。还有 forward-line 和 previous-line 。还有一些命令用来每次移动一句、一个段落、一页。

什么是一页呢? 通常，一页以分页符开始，这是因为传统打字机在新的一页开始的时候，打字员需要发送一个 contrl-L 给设备。但是在 Emacs 里工作的时候，我们可以通过改变 page-delimiter 来重新定义构成“页”的元素。

```
(make-local-variable 'page-delimiter)
(setq page-delimiter "^%%$")
```

这一转换—将“页”转换为“谚语”—解决了大部分我们对于 Quip 模式的需求！现在 Emacs 的很多内置的页相关的命令都能应用于谚语了:

- backward-page 和 forward-page ，通常绑定到 C-x [和 C-x] 上，允许每次移动一条谚语

- narrow-to-page，绑定在 C-x n p 上，通过 narrow 操作来只编辑一条谚语（参照本章中的Narrowing章节）
- what-page 查看当前的谚语是第几条

我们基本上完全借用了 Emacs 操作页面的命令，反正无所谓：在 Quip 模式里，这些命令反正用不到，因为 Quip 文件本身就不会分页。

9.5　键位表

不幸的是，这些命令的名字— backward-page 和 forward-page 以及其他的命令—使得这些方法在 Quip 模式中变得易于混淆，因为我们操作的是谚语而不是页面。因此这么做是明智的：

```
(defalias 'backward-quip 'backward-page)
(defalias 'forward-quip forward-page)
(defalias 'narrow-to-quip 'narrow-to-page)
(defalias 'what-quip 'what-page)
```

但是这并不够。即使定义了这些别名，已经存在的绑定— C-x [，C-x] ，以及 C-x n p —仍然绑定到了“页”的命令上，所以当用户在 Quip 模式中使用 describe-bindings 列出键绑定的时候，他们将会看到：

```
C-x [       backward-page
C-x ]         forward-page
C-x n p    narrow-to-page
```

（以及其他的页相关函数）但是这些与谚语都没关系。如果这些函数的名字与谚语相关就好了—当然只是在 Quip 模式里。而且，我们还可以把 C-x n p（这么定义是因为这表示 narrow to page）改为 C-x n q （narrow to quip）。我们还可以给 what-quip 绑定一个快捷键，默认是没有的。也就是说 Quip 模式需要一个关联的键位表。

键位表（keymap）是一个用来记录函数及触发它的按键的 Lisp 数据结构。例如，当你按下 C-f 时，Emacs 将会查询“global”键位表并且找到 C-f 对应的函数，也就是 forward-char。键位表中的每一条记录都代表着一条按键序列。

像是 C-x C-w（write-file）这种按键需要使用嵌套键位表（nestling keymaps）来实现。在全局表里，C-x 的记录包含着一个嵌套表而不是一条命令。嵌套表里包含着一条 C-w 的记录，它指向 write-file 。C-x 的嵌套表还包含着一条 n 的记录，它指向另一个嵌套表。这个二级嵌套表包含着一条 p 的记录，它指向命令 narrow-to-page 。

指向嵌套表的按键被称为前置键（prefix key）；C-x 是很多其他命令的前置键，而 C-x n 是更多按键的前置键（在 19.16 版本，你可以按下前置键后面跟

着 C-h 来看哪些键绑定以此前置）。

任何时候，都可能有多个键位表同时处于激活状态。前面提到的全局键位表（global keymap）总是处于激活状态。它会被包含着当前 buffer 主模式的特定按键的局部键位表所覆盖。而局部键位表又会被处于激活状态的子模式的键位表里的记录所覆盖。[4]

让我们为本章前面所提到的 Quip 模式创建一个局部表。首先我们创建一个用于包含键位表的变量。它的初始值为 nil。

```
(defvar quip-mode-map nil
  "Keymap for quip major mode.")
```

下一步我们在 quip.el 的最顶层编写一个代码块，用于当文件加载的时候加载键位表。如果 quip-mode-map 已经存在了—例如 quip.el 之前已经加载过了—那么就什么都不做。否则，就创建并且向其中添加键绑定。

```
(if quip-mode-map
    nil ; do nothing if quip-mode-map exists
  (setq quip-mode-map (make-sparse-keymap))
  (define-key quip-mode-map "\C-x[" 'backward-quip)
  (define-key quip-mode-map "\C-x]" 'forward-quip)
  (define-key quip-mode-map "\C-xnq" 'narrow-to-quip)
  (define-key quip-mode-map "\C-cw" 'what-quip))
```

我们使用 make-sparse-keymap 是因为 Quip 模式只比全局表多几个特定的绑定。只有当表里包含很多绑定的时候才需要使用 make-keymap 来创建一个完整的键位表。

每次调用 define-key 都会向 quip-mode-map 添加一条新的记录。当按键定义包含多于一个按键（就像本章中所有例子所展示的那样），define-key 将会自动根据需要创建嵌套表。[5]

我们将 what-quip 绑定到了 C-c w 。根据习惯，模式相关的命令通常绑定到以 C-c 开头的按键序列。其他的命令都来自于已经存在的绑定，所以没有必要再为他们指定新的前缀。

最后，我们需要确保在 Quip 模式进入的时候安装新的键位表。

[4]可以使用一个称为 overriding-local-map 的变量稍微更改一下这个优先级，但是这只在非常少数的情况下才有用。

[5]函数 current-global-map 会返回当前的全局键位表。（有可能通过 use-global-keymap 改变了全局表，虽然这很少出现。）因此，(global-set-key ...) 等价于 (define-key (current-global-map) ...)。

```
(defun quip-mode ()
  "Major mode for editing Quip files."
  (interactive)
  (kill-all-local-variables)
  (setq major-mode 'quip-mode)
  (setq mode-name "Quip")
  (make-local-variable 'paragraph-separate)
  (make-local-variable 'paragraph-start)
  (make-local-variable 'page-delimiter)
  (setq paragraph-start "%%\\I[ \t\n\^L]")
  (setq paragraph-separate "%%$\\ [ \t\^L]*$")
  (setq page-delimiter "^%%$")
  (use-local-map quip-mode-map) ; 这里安装键位表
  (run-hooks quip-mode-hook))
```

如果用户希望更改 Quip 模式的键绑定，他们可以通过使用模式的钩子以及 `local-set-key`（对于 Quip 模式来说就是修改 `quip-mode-map`）来达到目的：

```
(add-hook 'quip-mode-hook
          '(lambda ()
             (local-set-key "\M-p" backward-quip)
             (local-set-key "\M-n" 'forward-quip)
             (local-unset-key "\C-x[") ; 移除了一个绑定
             (local-unset-key "\C-x]")))
```

通常应该将模式的局部键位绑定放到用于描述模式的文档字符串里。但是，不应该像下面这样将默认的绑定“硬编码”到文档字符串里：

```
(defun quip-mode ()
  "Major mode for editing Quip files.
Keybindings include 'C-x [' and 'C-x ]' for backward-quip
and forward-quip, 'C-x n p' for narrow-to-quip, and 'C-c w'
for what-quip."
  ...)
```

因为我们前面已经说过，用户可能会重定义按键，这样的话文档字符串就不准确了。相反的，我们可以这么写：

```
(defun quip-mode ()
  "Major mode for editing Quip files.
Special commands:
\\{quip-mode-map}"
  ...)
```

当用户通过 describe-function 或者通过 describe-mode （使用所有相关模式的文档字符串来描述当前的主模式和子模式）来请求文档字符串的时候，Emacs 会根据这个特殊的语法使用当前 quip-mode-map 中的键位绑定进行替换。

9.6 Narrowing

你可能已经很熟悉 Emacs 的 narrowing 概念了。我们可以定义一个 buffer 的区域并且将 buffer narrow 到这个区域。Emacs 通过隐藏前面和后面的文字而使得整个 buffer 好像只有这个区域。所有的编辑操作，以及大部分的 Lisp 函数，都会被限制到这个区域内（虽然当文件保存的时候，所有的改动都会被保存而不管是否存在 narrowing），直到用户使用 widen 来取消 narrowing，widen 通常被绑定到 C-x n w [6]。所以 narrow-to-quip 满足“将用户的编辑操作限制到一条谚语”这个需求。

Emacs Lisp 代码必须要根据 buffer 是否处于 narrow 状态做处理。大多数情况，Lisp 函数都不必关心这件事。它们可以表现的就像 narrowed 部分就是整个 buffer。当处于 narrowing 状态时用于处理 buffer 边界的函数通常会处理 narrowed 区域的边界。例如，用来检测光标是否在 buffer 末尾的函数 eobp （end-of-buffer-p ）在光标位于 narrowed 区域末尾的时候会返回真。类似的，point-min 和 point-max 在 narrowed 区域存在的时候会返回它的边界而不是整个 buffer 的。可以说，这些函数为 Lisp 程序员编织了一个善意的谎言，否则他们编写代码的时候就需要付出很多努力来处理 narrowing 相关的情况。

但是，这也需要付出代价。在某些情况下，函数需要处理 narrowed 区域之外的 buffer。在这些情况下，需要先调用 widen 来使函数能够访问整个 buffer。如果这个调用被放到 save-restriction 里，那么在代码执行后 narrowing 状态将会被复原。（在第四章里我们用过这个手段。）

让我们定义 count-quips 作为例子，我们必须自己进行实现，因为 Emacs 并没有提供任何可以让我们利用的计算页数的命令。显然不管是否有 narrowing ，count-quips 都需要访问整个 buffer。因此，我们可以这样定义它：

[6] Narrowing 不会嵌套。如果将 buffer narrow 到了一个区域，然后再把这个区域 narrow 到更小的区域，C-x n w 仍然会恢复到整个 buffer（也就是说，它不会恢复到前一次 narrowing）。

```
(defun count-quips ()
  "Count the quips in the buffer."
  (interactive)
  (save-excursion
    (save-restriction
      (widen)
      (goto-char (point-min))
      (count-matches '^%%$'))))
```

函数 count-matches 会返回一个类似于“374 matches”的字符串告诉你从当前位置往后有多少处匹配该正则的地方。

9.7　继承模式

我们现在已经满足了除“它应该与 Text 模式大概相似”之外的所有要求了。实现的方法之一就是在初始化 Quip 模式的时候调用 text-mode ；然后再执行 Quip 模式自己的特定配置。我们可以使用 copy-keymap 而不是使用 make-sparse-map 来创建 quip-mode-map 以利用 text-mode 的特性。

```
(defvar quip-mode-map nil
  "Keymap for Quip major mode.")
(if quip-mode-map
    nil
  (setq quip-mode-map (copy-keymap text-mode-map))
  (define-key quip-mode-map "\C-x[" 'backward-quip)
  (define-key quip-mode-map "\C-x]" 'forward-quip)
  (define-key quip-mode-map "\C-xnq" 'narrow-to-quip)
  (define-key quip-mode-map "\C-cw" 'what-quip))

(defun quip-mode ()
  "Major mode for editing Quip files.
Special commands:
\\{quip-mode-map}"
  (interactive)
  (kill-all-local-variables)
  (text-mode) ; 首先设置为Text模式
  (setq major-mode 'quip-mode) ; 现在，定义Quip模式
  (setq mode-name "Quip")
```

```
  (use-local-map quip-mode-map)
  (make-local-variable 'paragraph-separate)
  (make-local-variable 'paragraph-start)
  (make-local-variable 'page-delimiter)
  (setq paragraph-start "%%\\|[ \t\n\^L]")
  (setq paragraph-separate "%%$\\|[ \t\^L]*\$")
  (setq page-delimiter "^%%$")
  (run-hooks quip-mode-hook))

(provide 'quip)
```

为了更好的与 text-mode 协作，我们还应该拷贝 text-mode-syntax-table（使用 copy-syntax-table），而不只是 text-mode-map 。当然需要处理的还有 text-mode-abbrev-table （但是没有对应的 copy-abbrev-table 函数，也许是因为缩写表太不常用了，以至于没有人在意是不是有这个方法）。

实际上，在你克隆一个模式并且将其定制成一个新的模式的时候有许多需要做的工作。你很容易就会遗漏掉什么。幸运的是，由于继承并且修改一个模式的行为太频繁了—就像我们将文本模式修改为 Quip 模式—所以已经有了一个 Emacs Lisp 包来帮助简化这一任务。这个包被称为 derived，它提供的核心函数被称为 define-derived-mode 。（实际上，define-derived-mode 是一个宏。）下面就是用它来继承 Text 模式生成 Quip 模式的实现：

```
(require 'derived)

(define-derived-mode quip-mode text-mode "Quip"
  "Major mode for editing Quip files.
Special commands:
\\ quip-mode-map}"
  (make-local-variable 'paragraph-separate)
  (make-local-variable 'paragraph-start)
  (make-local-variable 'page-delimiter)
  (setq paragraph-start "%%\\|[ \t\n\^L]")
  (setq paragraph-separate "%%$\\|[ \t\^L]*$")
  (setq page-delimiter "^%%$"))
(define-key quip-mode-map "\C-x[" 'backward-quip)
(define-key quip-mode-map "\C-x]" 'forward-quip)
(define-key quip-mode-map "\C-xnq" 'narrow-to-quip)
(define-key quip-mode-map "\C-cw" 'what-quip)
```

```
(provide 'quip)
```

`define-derived-mode` 的语法是

```
(define-derived-mode new-mode old-mode mode-line-string
  docstring
  body1
  body2
  ...)
```

这会创建 `new-mode` 以及所有相关的数据结构。在 `body` 语句执行的时候，`new-mode-map` ，`new-mode-syntax-table` ，以及 `new-mode-abbrev-table` 就已经存在了。创建 `new-mode` 指令的最后一件事是执行 `new-mode-hook` 。

本章向我们展示了如何较小的修改 Emacs 的行为来编辑特定类型的数据。Quip 模式与 Text 模式并没有多少区别，因为谚语本身与文本也没多大区别。但是在下一章，我们将会创建一个用来编辑与文本差别很大的数据的主模式，它与任何其他的主模式都很不相同。

第十章　一个综合示例

在本章：

- 纽约时报规则
- 数据表示
- 用户界面
- 建立模式
- 追踪未授权的修改
- 解析 Buffer
- 词语查找器

本章是我们编程示例的终点。实现一个填字游戏编辑器（crossword puzzle editor）是个好主意—显然 Emacs 的设计者并没有预先设计这一功能，但这可以被实现。设计和实现 Crossword 模式将会展示出 Emacs 作为应用构建工具的真正潜力。

在为填字游戏编辑器设计完数据模型之后，我们将会为它创建一个用户界面，创建用来展示我们数据模型的函数并且将输入限制在我们允许的范围内。我们将会编写插入到 Emacs 菜单中的命令以及用于跟其他进程通讯的命令。通过这些，我们将会利用我们前面学到的 Lisp 技术来执行复杂的逻辑以及字符串处理。

10.1　纽约时报规则

我是填字游戏的狂热粉丝。我曾经每天都做纽约时报的填字游戏。我经常会思考构建一个填字游戏需要哪些技术，并且希望我自己动手完成它。最初我试图用方格纸来做，但是我很快发现填字游戏的创建包含如此多的尝试和错误（至少对我如此）以至于我才做到中间，我的橡皮已经把纸擦透了！最终我选择编写一个计算机程序来帮助我创建填字游戏。

填字游戏表格（diagram），或者说网格（grid），包含着“空格（blank）”以及“障碍（block）”。空格是一个可能填充了字母的空方格。障碍则是一个用来

分隔单词的没有填写字母的黑色方块。优秀的填字游戏设计者将会尽可能少的使用障碍。

Crossword 模式将会延用被我称作“纽约时报规则”的填字游戏规则（当然这与其他的填字游戏应该是相似或者相同的）：

1. 网格是一个 $n \times n$ 的方块，n 为奇数。纽约时报的每日游戏为 15×15，而周末则为 21×21。
2. 网格符合“180 度对称”，即如果你把网格旋转 180 度，那么空格和障碍仍然保持相同的位置。[1]以数学语言来说，这表示如果网格 `square(x, y)`是一个空白的话，那么 `square(n-x+1, n-y+1)`也一定是（n 代表网格的宽度，x 和 y 从 0 开始）；如果 `(x, y)`是一个障碍，那么 `(n-x+1, n-y+1)`也一定是。如图10.1展示的 180 度对称的例子。

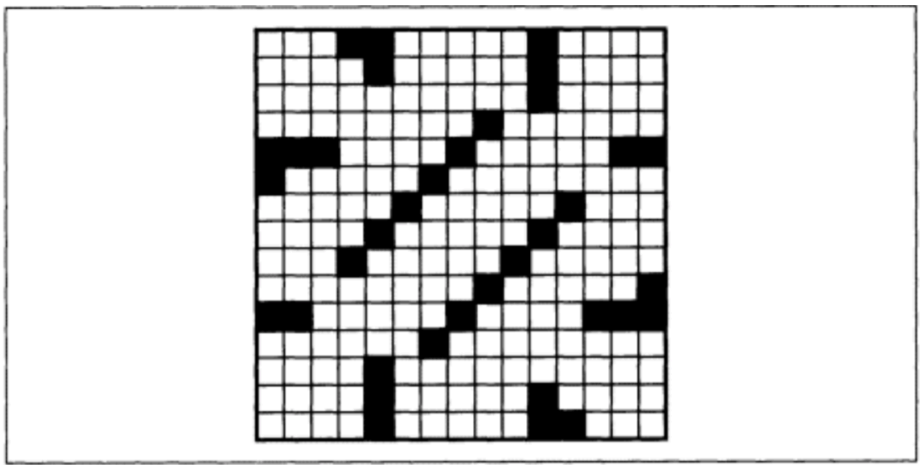

图 10.1: 一个 180 度对称的例子

3. 游戏中的所有单词必须至少包含 2 个字母。（实际上，我被告知纽约时报从未使用过少于三个字母的单词，但是为了简化，我们使用两个。）

10.2　数据表示

让我们以构建数据表示作为开始。一个明显的方法是使用一个二元数组来存储填字游戏的数据，或者说矩阵。Emacs Lisp 并没有这种数据类型，但是我们可以使用向量（vector）来创建一个。

[1] 180 度对称也被称为“双向对称（two-way symmetry）”。还有“四向对称（four-way symmetry）”，即每次旋转 90 度网格仍然相同。

10.2.1　向量（Vectors）

Lisp 的向量跟列表类似，即它是 0 个或多个子表达式（包括嵌套向量或者列表）的序列。但是，向量允许对其元素的随机访问，而列表却必须从头到尾遍历来找到一个元素。（但并不是说向量就比列表更好。向量不像列表一样可以加长或者缩短。所以就像常说的那样，要因地制宜。）

向量使用中括号来代替小括号：

```
[a b c ...]
```

向量是自运算的（self-evaluating）；也就是说，向量的运算结果是向量自身。它的子表达式并不会被求值。所以如果你写

```
[a b c]
```

你将会得到一个包含着三个符号的向量：a、b、和 c。如果你希望向量包含变量 a、b 和 c 的值，那么你必须使用 vector 函数来构建向量：

```
(vector a b c) -> [17 37 42] ; 或者其他什么值
```

10.2.2　矩阵包

使用向量来设计矩阵包是清晰直观的。我们指定一个向量来表示矩阵的行，而每一行都是一个代表列的向量。我们会这样创建：

```
(defun make-matrix (rows columns &optional initial)
  "Create a ROWS by COLUMNS matrix."
  (let ((result (make-vector rows nil))
        (y 0))
    (while (< y rows)
      (aset result y (make-vector columns initial))
      (setq y (+ y 1)))
    result))
```

参数 `initial` 指定了一个 Lisp 表达式作为矩阵内每个元素的初始值。第一次调用 `make-vector` 创建了一个填充了 `nil` 的向量，长度为 `rows` 。然后我们将每个 `nil` 替换为一个新的长度为 `columns` 的向量。函数 `aset` 用来设置向量的元

素而 aref 用来获取。[2]向量的索引从 0 开始。调用 (asset vector index value) 会将 vector 中的第 index 个元素设置为 value。调用 (aref vector index) 会返回位置为 index 的值。

在 while 循环里调用的 make-vector 会将每个嵌套向量的元素设置为 initial，所以在 make-matrix 的最后，result 是一个包含着 rows 个嵌套向量的向量，而每个嵌套向量都是由 columns 个 initial 组成的。

为什么不像下面这样更简单的编写这个函数呢：

```
(defun make-matrix (rows columns &optional initial)
  "Create a ROWS by COLUMNS matrix."
  (make-vector rows (make-vector columns initial))) ; 错啦!
```

原因是里面的那个 make-vector 只会产生一个新的向量。外面的调用会使用这个向量作为外面向量的每个元素的初始值。换句话说，外面向量的每个元素将会共享同一个内部的向量，而我们希望的是每个元素的值是各不相同的嵌套向量。

现在我们定义好了矩阵的结构，那么定义它的基本操作就很简单了：

```
(defun matrix-set (matrix row column elt)
  "Given a MATRIX, ROW, and COLUMN, put element ELT there."
  (let ((nested-vector (aref matrix row)))
    (aset nested-vector column elt)))

(defun matrix-ref (matrix row column)
  "Get the element of MATRIX at ROW and COLUMN."
  (let ((nested-vector (aref matrix row)))
    (aref nested-vector column)))
```

得到矩阵的宽和高的函数也许有用：

```
(defun matrix-columns (matrix)
  "Number of columns in MATRIX."
  (length (aref matrix 0))) ; 子向量的长度

(defun matrix-rows (matrix)
```

[2]这些函数名称中的 “a ” 表示 “array”。为什么不用表示 “vector” 的 vset 和 vref 呢？答案是在 Emacs Lisp 中，向量（vector）是数组（array）的一种。字符串是两一种数组。所以 aset 和 aref 可以像处理向量这样处理字符串—当然这并不表示字符串是向量。

```
  "Number of rows in MATRIX."
  (length matrix)) ; 外部向量的长度
```

当函数的定义非常短，就像上面这四个，通常使用 defsubst 而不是 defun 把它们转换为内联函数（inline functions）是个好主意。使用 defsubst 定义的内联函数与 defun 定义的函数的功能一样，除了在编译时对于内联函数的调用会被替换为函数本身。这有一个主要的好处：在运行时，当前函数并不需要建立一个对其他函数的调用。这会稍微快一点，但是如果是在成千上万次的循环中的话，叠加起来还是很可观的。不幸的是，这么做也有代价。首先每次调用都会拷贝一份，因此这会增加内存的占用。另一个是如果你修改了内联函数的定义，其他的定义仍然会保持编译文件里的那一份。（因此可以说，defsubst 与 C++ 的内联函数相同，或者与 C 语言中的宏函数相同。）

我们可以将上面的代码放到 matrix.el 中，在文件的最后添加一行 (provide 'matrix)，然后在之后的程序中通过 (require 'matrix) 使用它。

10.2.3　矩阵在填字游戏中的变化

现在让我们考虑一个填字游戏网格，也就是一种特殊的矩阵。其中的每个格子只有如下四种状态：

1. 空的，表示我们会向其中填写一个字母或者一个障碍。
2. 半空，表示我们会向其中填写一个字母而不是一个障碍（因为 180 度对称的原因）。
3. 填充了一个障碍。
4. 填充了一个字母。

让我们使用 nil 表示一个格子是空的，符号 letter 表示必须填写字母的半空格子，符号 block 表示一个包含着障碍的格子，以及字母本身（在 Emacs 中以数字表示，即它的 ASCII 码）表示一个包含着字母的格子。

根据上面的定义，让我们使用矩阵来为填字游戏网格定义一种新的数据类型。

```
(require 'matrix)
(defun make-crossword (size)
  "Make a crossword grid with SIZE rows and columns."
  (if (zerop (% size 2)) ; 是偶数吗？（%是取余函数）
      (error "make-crossword: size must be odd"))
  (if (< size 3) ; size是不是太小了？
      (error "make-crossword: size must be 3 or greater"))
  (make-matrix size size nil))
```

```
(defsubst crossword-size (crossword)
  "Number of rows and columns in CROSSWORD."
  (matrix-rows crossword)) ; 或者用matrix-columns, 都一样

(defsubst crossword-ref (crossword row column)
  "Get the element of CROSSWORD at ROW and COLUMN."
  (matrix-ref crossword row column))

(defsubst crossword--set (crossword row column elt)
  "Internal function for setting a crossword grid square."
  (matrix-set crossword row column elt))
```

函数 crossword--set 名字中间使用了双横线。这是一种习惯上定义“私有”函数的方式，它表示这个函数并不是包声明的编程接口。在这个例子里，crossword--set 是私有的，因为它并没有实现我们希望填字游戏网格所具有的纽约时报规则。Crossword 包的用户将不会直接使用 crossword--set ；它们会直接使用下面定义的 crossword-store-letter , crossword-store-block , 以及 crossword-clear-cell 。只有 Crossword 包自身以及一些用于判断 180 度对称和单词长度大于 2 的规则才会使用 crossword--set 。

10.2.4　使用 Cons Cells

让我们创建一个概念“兄弟（cousin）”来表示一个给定格子的对称格子。

```
(defun crossword-cousin-position (crossword row column)
  "Give the cousin position for CROSSWORD ROW and COLUMN."
  (let ((size (crossword-size crossword)))
    (cons (- size row 1) (- size column 1))))
```

这个函数会以 dotted pair 的方式返回兄弟格子的行、列值（参照第六章中列表细节的章节）：(cousin-row . cousin-column) 。下面是两个直接获取和设置兄弟格子的函数：

```
(defun crossword-cousin-ref (crossword row column)
  "Get the cousin of CROSSWORD's ROW,COLUMN position."
  (let ((cousin-position (crossword-cousin-position crossword
                                                    row
                                                    column)))
```

```
    (crossword-ref crossword
                   (car cousin-position)
                   (cdr cousin-position))))

(defun crossword--cousin-set (crossword row column elt)
  "Internal function for setting the cousin of a cell."
  (let ((cousin-position (crossword-cousin-position crossword
                                                    row
                                                    column)))
    (crossword--set crossword
                    (car cousin-position)
                    (cdr cousin-position)
                    elt)))
```

注意到 crossword--cousin-set 是另一个名字中间有双横线的“私有”函数。

现在让我们为纽约时报规则创建用来储存障碍和字母的函数。首先，字母。当向一个格子中填写一个字母的时候，我们必须保证格子的兄弟已经包含了一个字母（我们可以使用 numberp 来检测）。如果它没有，我们必须在那里存储一个符号 letter：

```
(defun crossword-store-letter (crossword row column letter)
  "Given CROSSWORD, ROW, and COLUMN, put LETTER there."
  (crossword--set crossword row column letter)
  (if (numberp (crossword-cousin-ref crossword row column))
      nil
    (crossword--cousin-set crossword row column 'letter)))
```

插入障碍稍微简单一点：

```
(defun crossword-store-block (crossword row column)
  "Given CROSSWORD, ROW, and COLUMN, put a block there."
  (crossword--set crossword row column 'block)
  (crossword--cousin-set crossword row column 'block))
```

现在让我们编写一个用来清空格子的函数。当清空一个格子时，有下面几种可能的情况：

- 格子和它的兄弟都包含着字母。如果是的话，格子变成“半空”状态而兄弟不受影响。

- 格子和它的兄弟都包含着障碍。如果是的话，格子和兄弟都清空。
- 格子已经是半空状态（因为它的兄弟包含着一个字母。）如果这样的话，什么都不发生。
- 格子包含着一个字母但它的兄弟是半空的。如果这样的话，两个格子都清空。
- 格子和兄弟都是空的。如果这样，什么都不发生。

我们可以使用一个简单的规则来处理这情况：如果格子的兄弟包含一个字母，那么格子变为半空并且兄弟不受影响；否则格子自身和它的兄弟都清空。下面就是实现的代码。

```
(defun crossword-clear-cell (crossword row column)
  "Erase the CROSSWORD cell at ROW,COLUMN."
  (if (numberp (crossword-cousin-ref crossword row column))
      (crossword--set crossword row column 'letter)
    (crossword--set crossword row column nil)
    (crossword--cousin-set crossword row column nil)))
```

现在看一下 $n \times n$（n 是奇数）网格中间的方格，它的兄弟是它自己。这表示我们需要对 `crossword-clear-cell` 做一点小的修正。中间的方格一定不能设置成符号 `letter` 。（幸运的是，`crossword-store-block` 和 `crossword-store-letter` 仍然能够正确地工作。）

```
(defun crossword-clear-cell (crossword row column)
  "Erase the CROSSWORD cell at ROW,COLUMN."
  (let ((cousin-position (crossword-cousin-position crossword
                                                    row
                                                    column)))
    (if (and (not (equal cousin-position
                         (cons row column)))
             (numberp (crossword-ref crossword
                                     (car cousin-position)
                                     (cdr cousin-position))))
        (crossword--set crossword row column 'letter)
      (crossword--set crossword row column nil)
      (crossword--set crossword
                      (car cousin-position)
                      (cdr cousin-position)
                      nil))))
```

在这个版本里，只有当 cousin-position 不等于 (row . column) 的时候格子才会被设置为 letter —也就是说，格子本身并不是自己的兄弟。如果格子是自己的兄弟，或者它的兄弟不包含字母，那么（与前一个版本一样）将它与它的兄弟都设置为 nil 。最后一个 crossword--set 对于中间的格子来说是多余的，但也并没有什么副作用。注意我们在函数的开头计算了兄弟的位置，这样我们使用 crossword-ref 替代了 crossword-cousin-ref ，使用 crossword--set 替代了 crossword--cousin-set ，这样就避免了对于兄弟位置的多次求值。

10.2.5　单字母单词

单字母单词会在一个符合“障碍，非障碍，障碍”的行中出现；或者当一个非障碍的格子出现在障碍与边界之间时出现。下面这个函数用来检测一个给定的方格是不是一个单字母单词。

```
(defun crossword-one-letter-p (crossword row column)
  "Is CROSSWORD cell at ROW,COLUMN a one-letter word?"
  (and (not (eq (crossword-ref crossword row column)
                'block))
       (or (and (crossword-block-p crossword (- row 1) column)
                (crossword-block-p crossword (+ row 1) column))
           (and (crossword-block-p crossword row (- column 1))
                (crossword-block-p crossword row (+ column 1))))))
```

这个逻辑有一点复杂，但是我们可以用第三章中学到的技术来理解这种表达式：即每次深入一层子表达式：

```
(and ...)
```

当 crossword-one-letter-p 的子表达式的结果都为 true 时函数将返回 true ，否则返回 false 。

```
(and (not ...)
     (or ...))
```

“如果某些事情不为真并且其他的一些事情为真则返回真。”

```
(and (not (eq ...))
     (or (and ...)
         (and ...)))
```

“如果一些事情不等于另一些并且一些事情都为真或者另一些事情都为真则返回真。”

```
(and (not (eq (crossword-ref crossword row column)
              'block))
     (or (and (crossword-block-p crossword (- row 1) column)
              (crossword-block-p crossword (+ row 1) column))
         (and (crossword-block-p crossword row (- column 1))
              (crossword-block-p crossword row (+ column 1)))))
```

“如果当前的格子不是障碍并且其上面的格子和下面的格子是障碍或者左面的格子和右面的格子是障碍则返回真。”这里包含着一个小技巧：`crossword-block-p` 在访问边界外的方格时则认为它们包含障碍。

下面是 `crossword-block-p` 的定义：

```
(defun crossword-block-p (crossword row column)
  "Does CROSSWORD's ROW,COLUMN cell contain a block?"
  (or (< row 0)
      (>= row (crossword-size crossword))
      (< column 0)
      (>= column (crossword-size crossword))
      (eq (crossword-ref crossword row column) 'block)))
```

10.3　用户界面

现在我们有了一整套用来根据我们制定的规则来处理填字游戏数据结构的函数；但是还没有提供方法让用户与填字游戏网格进行交互。我们必须编写用户界面，这包括多个用来操作填字游戏的命令以及用来实时显示填字游戏网格的方法。

10.3.1　显示

让我们选择一种用来在 Emacs buffer 中显式填字游戏网格的方法。

网格的每一行在 buffer 中都被表示为一行。但是对于网格中的列，应该占用屏幕上的两列—这会使网格在大多数显示器上显得更方正。(在大多数字体里，一个字母的高度会比宽度高很多，这会使得一个 $n \times n$ 的字符块看起来非常窄。)

空格子会用点号（`.`）表示。障碍用井号（`#`）表示。半空格子用问号（`?`）表示。当然，包含字母的格子用字母自身表示。

下面这个方法会把一个网格插入到当前 buffer 里。它不会清空 buffer，也不会放置光标；这应该留给这个函数的调用者来做，我们稍后会定义他们。

```
(defun crossword-insert-grid (crossword)
  "Insert CROSSWORD into the current buffer."
  (mapcar 'crossword-insert-row crossword))
```

回忆我们在第六章中其他有用的列表函数一节中讲到的 mapcar 会把一个方法应用到列表中的每个元素上。而这对向量也有效；因为 crossword 是一个包含着行的向量，所以 crossword-insert-grid 对网格的每一行调用了 cross-insert-row 。

下面是上面用到的 crossword-insert-row 的定义：

```
(defun crossword-insert-row (row)
  "Insert ROW into the current buffer."
  (mapcar 'crossword-insert-cell row)
  (insert "\n"))
```

这与上面的工作方式一样，即对每行中的每个格子调用了 crossword-insert-cell 。在行的最后，我们另起一行。

最后，下面是 crossword-insert-row 所需要的 crossword-insert-cell ：

```
(defun crossword-insert-cell (cell)
  "Insert CELL into the current buffer."
  (insert (cond ((null cell) ".")
                ((eq cell 'letter) "?")
                ((eq cell 'block) "#")
                ((numberp cell) cell))
          " "))
```

这会插入两个字母；一个点号，问号，井号或者一个字母；后面跟着一个空格（这会使一个格子在屏幕上占有两列）。第一个字符插入什么用一个 cond 结构来确定，也就是 if 的一个变体。cond 的每一个参数被称作子句（clause），每个子句是一个列表。每个子句的第一个元素被称为条件（condition），会按次序执行。当一个子句的条件运算结果为真时，那么子句剩下的元素（如果存在的话）就会被执行，而最后一个表达式的值会作为 cond 的返回值。成功执行的子句后面的子句会被忽略。

```
(cond ((condition1 body ...)
       (condition2 body ...)
       ...))
```

如果你希望 cond 中包含一个“else”子句—当其他子句都不为真的时候执行—你可以在最后添加一个条件为真的子句：

```
(cond ((condition1 body ...)
       (condition2 body ...)
       ...
       (t body ...)))
```

函数 insert 会将任意数量的字符串或者字母插入到当前的 buffer 里；这也就是为什么我们可以向它传递格子的值，不管是数字、“”、或者字符串来插入。

10.3.2　放置光标

让我们继续构建我们的模式一定会用到的组件。

现在我们可以展示填字游戏网格了，我们还应该提供一种方法来把光标放到任意一个网格里。光标的位置向用户展示了下一次操作将会影响哪个格子。下面这个函数假定网格已经画到了当前的 buffer 里，并且它起始于（ point-min ）。[3]

```
(defun crossword-place-cursor (row column)
  "Move point to ROW,COLUMN."
  (goto-char (point-min))
  (forward-line row)
  (forward-char (* column 2)))
```

下一步，当用户触发一些操作时，我们需要根据光标的位置得到它在当前网格中对应的座标。

```
(defun crossword-cursor-coords ()
  "Compute (ROW . COLUMN) from cursor position."
```

[3]虽然我们在本章中并没有提到，但是你应该记得（ point-min ）并不一定返回 buffer 中的开始位置；如果 narrowing 生效的话它返回的可能是 buffer 中间的某个位置。

```
  (cons (- (current-line) 1)
        (/ (current-column) 2)))
```

函数 / 是 Emacs Lisp 中的除法函数，在两个参数都是整数的时候执行整除。结果会向 0 取整。感谢这个方法，

```
(/ (current-column) 2)
```

不管光标在前面的列还是后面的空格都会返回正确的列数。

不幸的是，虽然 Emacs 内置了 current-column ，但是却没有一个 current-line 。[4]下面是其中的一种写法：

```
(defun current-line ()
  "Return line number containing point."
  (let ((result 1)) ; Emacs从1开始计算行数
    (save-excursion
      (beginning-of-line) ; 这样bobp将会正常工作
      (while (not (bobp))
        (forward-line -1)
        (setq result (+ result 1))))
    result))
```

函数 bobp 用来检测光标是否在 buffer 的开头。

10.3.3　更新显示

当用户编辑网格时，下层的数据结构的改变需要反映到 buffer 里。如果每次都擦除整个 buffer 然后调用 crossword-insert-grid 插入整个网格是很没有效率的。相反的，我们将只绘制发生改变的格子。

我们已经有了相关的工具：crossword-place-cursor 和 crossword-insert-cell 。下面是一个使用了这些组件的函数。它假定光标在被影响的格子上，然后重画这个格子和它的兄弟。

```
(defun crossword-update-display (crossword)
  "Called after a change, keeps the display up to date."
  (let* ((coords (crossword-cursor-coords))
```

[4]有一个 what-line ，但是这个函数用于交互式命令，而非在程序中使用。它会显示关于当前行号的信息，并且不会返回一个有用的返回值。我们需要一个行为相反的函数：不显示信息，并且返回当前的行号。

```
        (cousin-coords (crossword-cousin-position crossword
                                                  (car coords)
                                                  (cdr coords))))
    (save-excursion
      (crossword-place-cursor (car coords)
                              (cdr coords))
      (delete-char 2)
      (crossword-insert-cell (crossword-ref crossword
                                            (car coords)
                                            (cdr coords)))
      (crossword-place-cursor (car cousin-coords)
                              (cdr cousin-coords))
      (delete-char 2)
      (crossword-insert-cell (crossword-ref crossword
                                            (car cousin-coords)
                                            (cdr cousin-coords))))))
```

你可能会认为这个函数里对于 `crossword-place-cursor` 的第一次调用是多余的，因为它把光标放到了刚刚通过 `crossword-cursor-coords` 取到的同一个位置。但是要知道在网格中一个格子有两列宽，而光标很有可能在第二列里。为了使 `crossword-insert-cell` 正确工作，光标必须在第一列里。`crossword-place-cursor` 保证了这一点。外面包裹的 `save-excursion` 保证了在更新完成后光标返回到它原来的地方。

10.3.4　用户命令

现在我们需要定义用户与 Crossword 模式交互的命令。

网格变更指令

让我们假设使用 Crossword 模式的 buffer 中有一个名为 `crossword-grid` 的 buffer 局部的变量保存着填字游戏的网格。（在下一节我们将会设计在创建 `crossword-mode` 命令时创建 `crossword-grid` 的具体细节。）因此用来清空一个格子的用户命令可以如下编写。

```
(defun crossword-erase-command ()
  "Erase current crossword cell."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-clear-cell crossword-grid
```

```
                          (car coords)
                          (cdr coords)))
  (crossword-update-display crossword-grid))
```

类似的，下面的命令用来插入一个障碍：

```
(defun crossword-block-command ()
  "Insert a block in current cell and cousin."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-store-block crossword-grid
                           (car coords)
                           (cdr coords)))
  (crossword-update-display crossword-grid))
```

用来插入字母的命令会更棘手一点。一共有 26 个字母，而我们并不想编写类似 `crossword-insert-a` 和 `crossword-insert-b` 这样的 26 个不同的命令。我们希望用一个函数绑定到 26 个字母按键，当触发的时候插入对应的字母。一个通用的函数是 `self-insert-command` 。我们将定义一个插入用户按下的字母的函数 `crossword-self-insert` 。

```
(defun crossword-self-insert ()
  "Self-insert letter in current cell."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-store-letter crossword-grid
                            (car coords)
                            (cdr coords)
                            (aref (this-command-keys) 0)))
  (crossword-update-display crossword-grid))
```

这个函数用了 `this-command-keys` 来检测用户按下了哪个按键。`this-command-keys` 将会返回一个字符串或者一个包含着符号事件（symbolic events）的向量（在本章后面的鼠标命令的部分将会描述更多细节）；但是 `crossword-store-letter` 需要的是一个字符，而不是字符串，或者符号，或者向量。我们使用 `aref` 来得到第一个元素并且传递给 `crossword-store-letter` ，是基于我们确信它是一个字符串，并且我们并不关心除了第一个元素之外的东西。这应该没问题，因为当我们在后面的章节按键绑定中做设置的时候，我们只会将 `crossword-self-insert`

绑定到单个按键上（也就是说字母按键），并且我们会让用户不可能，或者至少更困难地输入不合法的字符。

导航

用户需要一些除了 Emacs 原本的光标移动命令之外的方式来在格子之间移动，因为它们并不能很好的适配到网格的导航里。例如，每个网格都是两列宽，所以我们需要按两次 `C-f` 来移动到右面的格子。再比如，移动到网格的最左边不应该自动拐回到下一行的开始处。它应该直接停止。

导航命令的定义是很直观的。只需要知道用户希望移动的方向，以及移动多远。我们需要定义向左、右、上、下移动一个格子的命令；用来移动到行首、尾的命令；移动到列首、尾的命令；以及移动到网格的开始（左上角）、结束（右下角）的命令。

首先，横向移动的命令：

```
(defun crossword-cursor-right (arg)
  "Move ARG cells to the right."
  (interactive "p") ; 前置参数为数字
  (let* ((coords (crossword-cursor-coords))
         (new-column (+ arg (cdr coords))))
    (if (or (< new-column 0)
            (>= new-column (crossword-size crossword-grid)))
        (error "Out of bounds"))
    (crossword-place-cursor (car coords)
                            new-column)))

(defun crossword-cursor-left (arg)
  "Move ARG cells to the left."
  (interactive "p")
  (crossword-cursor-right (- arg)))
```

类似的纵向移动的命令：

```
(defun crossword-cursor-down (arg)
  "Move ARG cells down."
  (interactive "p")
  (let* ((coords (crossword-cursor-coords))
         (new-row (+ arg (car coords))))
    (if (or (< new-row 0)
```

```
            (>= new-row (crossword-size crossword-grid)))
        (error "Out of bounds"))
    (crossword-place-cursor new-row
                            (cdr coords))))

(defun crossword-cursor-up (arg)
  "Move ARG cells up."
  (interactive "p")
  (crossword-cursor-down (- arg)))
```

现在定义移动到行列头尾的命令。

```
(defun crossword-beginning-of-row ()
  "Move to beginning of current row."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-place-cursor (car coords) 0)))

(defun crossword-end-of-row ()
  "Move to end of current row."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-place-cursor (car coords)
                            (- (crossword-size crossword-grid)
                               1))))

(defun crossword-top-of-column ()
  "Move to top of current column."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-place-cursor 0 (cdr coords))))

(defun crossword-bottom-of-column ()
  "Move to bottom of current row."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (crossword-place-cursor (- (crossword-size crossword-grid)
                               1)
```

```
                              (cdr coords))))
```

最后，移动到网格的首尾的命令。

```
(defun crossword-beginning-of-grid ()
  "Move to beginning of grid."
  (interactive)
  (crossword-place-cursor 0 0))

(defun crossword-end-of-grid ()
  "Move to end of grid."
  (interactive)
  (let ((size (crossword-size crossword-grid)))
    (crossword-place-cursor size size)))
```

又仔细想了想，下面的东西也许会有用：一个用来跳到当前格子的兄弟的命令。

```
(defun crossword-jump-to-cousin ()
  "Move to cousin of current cell."
  (interactive)
  (let* ((coords (crossword-cursor-coords))
         (cousin (crossword-cousin-position crossword-grid
                                            (car coords)
                                            (cdr coords))))
    (crossword-place-cursor (car cousin)
                            (cdr cousin))))
```

10.4　建立模式

有两种情况用户希望进入 Crossword 模式。一种是访问保存着之前填字游戏网格缓存的文件。另一种是创建一个新的网格的时候。

创建一个新的网格需要创建一个新的 buffer 并且使用 `crossword-insert-grid` 填充它。进入主模式不应该变更 buffer 的内容，所以 `crossword-mode` 将只会在一个 buffer 已经包含着填字游戏网格的情况下进入。我们将设计一个单独的命令，crossword，来创建一个新的网格。

```
(defun crossword (size)
  "Create a new buffer with an empty crossword grid."
```

```
(interactive "nGrid size: ")
(let* ((grid (make-crossword size))
       (buffer (generate-new-buffer "*Crossword*")))
  (switch-to-buffer buffer)
  (crossword-insert-grid grid)
  (crossword-place-cursor 0 0) ; 从左上角开始
  ...))
```

我们现在先不完成这个函数，但是在继续之前，让我们看一下这个函数中间的一些有趣的东西：

1. (`interactive "nGrid Size: "`)。字母 `n` 是 Emacs 提示用户输入值的提示符之一。这会允许你提供一个提示字符串，就像我们上面做的那样。这个 `interactive` 声明表示，“用字符串 “Grid size:” 提示用户，然后读入一个数字作为返回。” 但是如果这个命令需要两个参数，一个数字然后一个字符串呢？那么这个 `interactive` 的声明看起来会是什么样呢？Emacs 认为 `n` 后面直到新的一行之间的部分都是提示字符串。所以只要在字符串中嵌入一个换行符，然后再引入另一个提示符就可以了，就像这样：

```
(interactive "nFirst prompt: \nSecond prompt: ")
```

2. 我们使用 `let*` 而不是 `let` 来使得 `grid` 在 buffer 之前创建。这并不是必须的，因为 buffer 的创建并不依赖于 `grid` （例如，`size` 可能是一个不合法的值）。真正的原因是在 Emacs 中创建 buffer 的代价非常高，而且 buffer 并不像其他的变量那样会自己释放（垃圾回收并不会回收它）。一旦一个 buffer 创建了，直到调用 `kill-buffer` 前它一直存在。
3. 新 buffer 的名字是 `*Crossword*` 。通常，没有关联文件的 buffer 的名字一般以星号开头和结尾—例如 `*scratch*` 和 `*Help*` 。当用户编辑了 buffer 之后，他可以将其保存到一个文件里（例如通过 `C-x C-w` ），这时 Emacs 会将 buffer 重命名为对应的文件。

让我们暂时把注意力集中到 `crossword-mode` 命令上。就像我们之前已经决定的，它只应用于已经包含填字游戏网格的 buffer。它会解析这个 buffer。也就是说会根据 buffer 里的文字构建一个新的网格对象出来。解析过的网格需要赋值给 `crossword-grid` 。下面是根据第九章中讲到的主模式而编写的：

```
(defun crossword-mode ()
  "Major mode for editing crossword puzzles.
Special commands:
\\{crossword-mode-map}"
```

```
  (interactive)
  (kill-all-local-variables)
  (setq major-mode 'crossword-mode)
  (setq mode-name "Crossword")
  (use-local-map crossword-mode-map)
  (make-local-variable 'crossword-grid)
  (setq crossword-grid (crossword-parse-buffer))
  (crossword-place-cursor 0 0) ; 从左上角开始
  (run-hooks 'crossword-mode-hook))
```

后面我们再定义 crossword-mode-map 和 crossword-parse-buffer 。

现在让我们来看一下 crossword 命令。在将一个网格放置到一个空的 buffer 中后，这个 buffer 必须进入 Crossword 模式。怎么做呢？最明显的答案是调用 crossword-mode ：

```
(defun crossword (size)
  "Create a new buffer with an empty crossword grid."
  (interactive "nGrid size: ")
  (let* ((grid (make-crossword size))
         (buffer (generate-new-buffer "*Crossword*")))
    (switch-to-buffer buffer)
    (crossword-insert-grid grid)
    (crossword-place-cursor 0 0) ; 从左上角开始
    (crossword-mode)))
```

这看起来不错，但是有一些效率问题。crossword-mode 会调用 crossword-parse-buffer 来创建一个 crossword 数据结构，即使 crossword 之前已经建立好了。如果能够保持一份 crossword 的拷贝的话就可以跳过解析这一步。

这么做的最好的方式是创建另一个被 crossword 和 crossword-mode 同时使用的函数，它负责进入 Crossword 模式时两边相同的处理。

```
(defun crossword--mode-setup (grid)
  "Auxiliary function to set up crossword mode."
  (kill-all-local-variables)
  (setq major-mode 'crossword-mode)
  (setq mode-name "Crossword")
  (use-local-map crossword-mode-map)
  (make-local-variable 'crossword-grid)
```

```
  (setq crossword-grid grid)
  (crossword-place-cursor 0 0)
  (run-hooks 'crossword-mode-hook))
```

我们让 crossword--mode-setup 使用网格作为参数。所以 crossword 应该用自己构建的网格去调用它：

```
(defun crossword (size)
  "Create a new buffer with an empty crossword grid."
  (interactive "nGrid size: ")
  (let* ((grid (make-crossword size))
         (buffer (generate-new-buffer "*Crossword*")))
    (switch-to-buffer buffer)
    (crossword-insert-grid grid)
    (crossword--mode-setup grid)))
```

而 crossword-mode 则应该使用解析 buffer 的结果来调用它：

```
(defun crossword-mode ()
  "Major mode for editing crossword puzzles.
Special commands:
\\{crossword-mode-map}"
  (interactive)
  (crossword--mode-setup (crossword-parse-buffer)))
```

10.4.1　按键绑定

在前面我们定义了几个用户命令，例如 crossword-erase-command 和 crossword-block-command 。现在让我们定义 crossword-mode-map 并且为这些命令选择对应的按键绑定。

```
(defvar crossword-mode-map nil
  "Keymap for Crossword mode.")
(if crossword-mode-map
    nil
  (setq crossword-mode-map (make-keymap))
  ...)
```

这些命令大部分都与通常的 Emacs 命令很类似。例如，crossword-beginning-of-row 和 crossword-end-of-row 与 beginning-of-line 和 end-of-line 很相似，而

它们分别绑定到了 C-a 和 C-e 。这是不是表示我们应该把它们做出类似的绑定呢?

```
(define-key crossword-mode-map "\C-a"
  crossword-beginning-of-row)
(define-key crossword-mode-map "\C-e"
  crossword-end-of-row)
```

也许吧。但是假如用户并不使用 C-a 来调用 beginning-of-line 呢?这样的话,C-a 就不是个正确的选择。因为它们的相似性,用户会希望 crossword-beginning-of-row 与 beginning-of-line 具有相同的快捷键。最好的选择是我们能够找到用户对于 beginning-of-line 的绑定然后将 crossword-beginning-of-row 绑定到上面。而这就是 substitute-key-definition 的功能。

```
(substitute-key-definition 'beginning-of-line
                           'crossword-beginning-of-row
                           crossword-mode-map
                           (current-global-map))
```

这表示，“获取当前 begining-of-line 在全局键位表中的绑定，然后在 crossword-mode-map 中为 crossword-beginning-of-row 创建一个相同的绑定。”

我们可以使用多个对于 substitute-key-definition 的调用来建立 crossword-mode-map ；或者更准确的说，使用一个循环。

```
(let ((equivs
       ((forward-char . crossword-cursor-right)
        (backward-char . crossword-cursor-left)
        (previous-line . crossword-cursor-up)
        (next-line . crossword-cursor-down)
        (beginning-of-line . crossword-beginning-of-row)
        (end-of-line . crossword-end-of-row)
        (beginning-of-buffer . crossword-beginning-of-grid)
        (end-of-buffer . crossword-end-of-grid))))
  (while equivs
    (substitute-key-definition (car (car equivs))
                               (cdr (car equivs))
                               crossword-mode-map
```

```
                                (current-global-map))
    (setq equivs (cdr equivs))))
```

我们创建了一个包含着“等价对”的 list 变量 equivs 。每次遍历循环的时候，(car equivs) 会取到一个等价对，例如 (next-line . crossword-cursor-down) 。这样，(car (car equivs)) 就是要在全局键位表里要查找的命令（例如 next-line ）而 (cdr (car equivs)) 则是对应的要放到 crossword-mode-map 中的命令（例如 crossword-cursor-down ）。

现在我们必须将字母键绑定到 crossword-self-insert 上。

```
(let ((letters
       '(?A ?B ?C ?D ?E ?F ?G ?H ?I ?J ?K ?L ?M
         ?N ?O ?P ?Q ?R ?S ?T ?U ?V ?W ?X ?Y ?Z
         ?a ?b ?c ?d ?e ?f ?g ?h ?i ?j ?k ?l ?m
         ?n ?o ?p ?q ?r ?s ?t ?u ?v ?w ?x ?y ?z)))
  (while letters
    (define-key crossword-mode-map
      (char-to-string (car letters))
      'crossword-self-insert)
    (setq letters (cdr letters))))
```

这样我们就只有 crossword-erase-command , crossword-block-command ,crossword-top-of-column ,crossword-bottom-of-column 和 crossword-jump-to-cousin 没有绑定了（因为它们在通常的编辑模式中并没有对应的操作）。让我们先绑定前两个：

```
(define-key crossword-mode-map " " crossword-erase-command)
(define-key crossword-mode-map "#" crossword-block-command)
```

因为看起来对于清空格子和插入障碍操作来说这很直观。对于剩下的三个，让我们使用以 C-c 开头的两次按键来绑定。前面我们说过，C-c 是模式相关的绑定的前缀。

```
(define-key crossword-mode-map "\C-ct"
  'crossword-top-of-column)
(define-key crossword-mode-map "\C-cb"
  'crossword-bottom-of-column)
```

```
(define-key crossword-mode-map "\C-c\C-c"
  'crossword-jump-to-cousin) ; 来自于C-x C-x
```

这些就是目前我们需要的所有按键绑定；但是不幸的是，就像其他所有的键位表，对于未设置的按键都会自动继承全局表。这表示，有一些按键可能会对我们小心构建的填字游戏网格造成破坏。数字键和其他一些可见字还是绑定在 self-insert-command 上；C-w ，C-k 和 C-d 仍然能删除掉 buffer 的一部分；C-y 仍然会在任意点插入任意内容；等等。

这些情况可以使用 suppress-keymap 部分解决，它可以使所有的 self-inserting 按键变为未定义。我们应该在创建键位表之后定义按键之前调用 suppress-keymap 。

```
(if crossword-mode-map
    nil
  (setq crossword-mode-map (make-keymap))
  (suppress-keymap crossword-mode-map)
  ...)
```

这只会保证 self-inserting 的按键行为正确，但是类似 C-w 和 C-y 这样的其他危险按键还潜伏着。一个更完全（更彻底）的方法是在 crossword-mode-map 中截取所有按键绑定：

```
(define-key crossword-mode-map [t] 'undefined)
```

在这个对于 define-key 的调用里，按键参数并不是像我们之前那样是一个字符串；它是一个包含着 t 的向量。我们之前说过向量和字符串是相似的；它们都是数组的一种。实际上，在 define-key 中一个包含着字母的向量和一个包含着字母的字符串作用相同；我们将在下一部分更仔细的观察包含着符号的向量。向量 [t] 表示捕获所有在当前键位表中未绑定的按键。通常，如果当前的局部键位表中未定义一个按键，那么就会去查找全局表。[t] 表示 “在这儿停下”。所以这是禁用所有未显式启用的按键的一种方法。

10.4.2　鼠标命令

当在类似 X 这种图形界面系统下运行 Emacs 的时候，鼠标也可以像按键那样触发操作。实际上，鼠标动作与普通按键的绑定都在同一个键位表里。

键位表数据结构可以是向量，assoc list，或者是它们俩的组合。当你按下一个键，你会产生一个用来索引向量的数值，或者用来搜索 assoc 的键值。当你点击鼠标时，你会产生一个只能用来搜索 assoc 的符号。例如符号 down-mouse-1

表示按下了鼠标键 1（通常是左键），而符号 mouse-1 表示按键 1 松开了。（习惯上按键按下的事件用来获取鼠标指针的位置，而松开用来判断鼠标按下之后是否移动过。）其他的鼠标事件包括 C-down-mouse-2（按住 ctrl 键的同时按下鼠标中键），S-drag-mouse-3（按下 shift 键的同时拖动按键 3），以及 double-mouse-1（双击按键 1）。

鼠标输入与键盘输入的另一个不同是当你按下鼠标键时会带来一些额外的数据：例如，你在窗口中按下按键的位置。按键输入总是发生在“point”上，而鼠标输入则发生在鼠标光标处。因此，鼠标输入被表示为一个称为输入事件（input event）的数据结构。绑定到一个鼠标动作的命令可以通过调用 `last-input-event`，或者在 `interactive` 声明中使用符号 e 来访问到当前的事件。

为了展示这些，让我们为 Crossword 模式定义三个简单的鼠标指令。鼠标按键 1 将会把光标放到一个格子里，鼠标按键 2 将会放入一个障碍，而按键 3 则会清除掉一个格子。

在每个例子里，初始的 down-事件会放置光标并且将位置记录到一个变量 `crossword-mouse-location` 里。当按键松开时，新的位置与之前的位置比较。如果不同的话，什么都不做。

让我们以 `crossword-mouse-set-point` 开始，这个函数会回应鼠标键按下的事件。

```
(defvar crossword-mouse-location nil
  "Location of last mouse-down event, as crossword coords.")
(defun crossword-mouse-set-point (event)
  "Set point with the mouse."
  (interactive "@e")
  (mouse-set-point event)
  (let ((coords (crossword-cursor-coords)))
    (setq crossword-mouse-location coords)
    (crossword-place-cursor (car coords)
                            (cdr coords))))
```

`interactive` 声明中的 @ 表示“在做任何事情之前，找到任何的触发这个命令的鼠标点击（如果有的话）并且选中点击发生的窗口”。code letter e 告诉 `interactive` 将触发这个命令的鼠标事件打包为一个列表并且赋给 event。

我们并不需要从这个事件结构中得到任何信息，但是我们需要将它传递给 `mouse-set-point`，它需要使用 event 当中储存的窗体位置数据来为 point 计算一个新的位置。当 point 放置完成后，我们可以调用 `crossword-cursor-coords` 来计算并且记住所在的网格座标。最后我们调用 `crossword-place-cursor`，因为每个格子都有两列宽而 `mouse-set-point` 可能把光标放到了错误的列上。

下面我们为这三个鼠标按下事件建立绑定：

```
(define-key crossword-mode-map [down-mouse-1]
  'crossword-mouse-set-point)
(define-key crossword-mode-map [down-mouse-2]
  'crossword-mouse-set-point)
(define-key crossword-mode-map [down-mouse-3]
  'crossword-mouse-set-point)
```

现在我们分别来看每个鼠标释放的操作。我们希望释放按键 1 与按下按键 1 做的事情一样，所以简单的把 mouse-1 绑定到 down-mouse-1 所绑定的指令就可以了：

```
(define-key crossword-mode-map [mouse-1]
  'crossword-mouse-set-point)
```

下面是用来放置障碍和擦除格子的命令：

```
(defun crossword-mouse-block (event)
  "Place a block with the mouse."
  (interactive "@e")
  (mouse-set-point event)
  (let ((coords (crossword-cursor-coords)))
    (if (equal coords crossword-mouse-location)
        (crossword-block-command))))

(defun crossword-mouse-erase (event)
  "Erase a cell with the mouse."
  (interactive "@e")
  (mouse-set-point event)
  (let ((coords (crossword-cursor-coords)))
    (if (equal coords crossword-mouse-location)
        (crossword-erase-command))))
```

下面是对于这些命令的绑定：

```
(define-key crossword-mode-map [mouse-2]
  'crossword-mouse-block)
(define-key crossword-mode-map [mouse-3]
  'crossword-mouse-erase)
```

10.4.3　菜单命令

我们还没有用来检测单字母单词的用户命令;但是我们在本章的前面章节单字母单词中定义了一个 crossword-one-letter-p 。让我们用它来定义一个命令，crossword-find-singleton ，用来找到网格中的单字母单词（如果存在的话）并且把光标移动到那里。

```
(defun crossword-find-singleton ()
  "Jump to a one-letter word, if one exists."
  (interactive)
  (let ((row 0)
        (size (crossword-size crossword-grid))
        (result nil))
    (while (and (< row size)
                (null result))
      (let ((column 0))
        (while (and (< column size)
                    (null result))
          (if (crossword-one-letter-p crossword-grid
                                      row column)
              (setq result (cons row column))
            (setq column (+ column 1)))))
      (setq row (+ row 1)))
    (if result
        (crossword-place-cursor (car result)
                                (cdr result))
      (message "No one-letter words."))))
```

这个函数会遍历网格中的所有格子，检测它是不是一个单字母单词，找到第一个的时候停止或者显式信息 “No one-letter words.”

我们可以将其绑定到一个按键上。C-c 1 展示了它的用途。

```
(define-key crossword-mode-map "\C-c1"
  'crossword-find-singleton)
```

但是对于用户来讲检测单字母单词并不像光标操作和其他命令那样是一个常用操作。用户可能并不希望为此记住一个按键绑定。既然它并不会频繁的使用，将它放到菜单上就是一个很好的选择。

定义菜单项是很简单的，这涉及到了键位表的另一个方面。首先我们需要定义一个新的键位表，它需要包含菜单“card”的菜单项。后面我们会为这个菜单添加一个顶级的菜单栏“Crossword”。

```
(defvar crossword-menu-map nil
  "Menu for Crossword mode.")
(if crossword-menu-map
    nil
  (setq crossword-menu-map (make-sparse-keymap "Crossword"))
  (define-key crossword-menu-map [find-singleton]
    '("Find singleton" . crossword-find-singleton)))
```

菜单键位表必须有一个“总提示语”。这也就是 make-sparse-keymap 中的可选参数“Crossword”的意义。

当前我们的菜单只有一个菜单项。它绑定到了一个自定义的事件符号 find-singleton。这个“事件”绑定到了一个包含着字符串“Find singleton”以及符号 crossword-find-singleton 的 cons cell。字符串用于显示菜单项的描述。符号则是选中菜单项时要触发的函数名称。自定义的事件符号 find-singleton 是没有意义的，它只需要跟同一个菜单中的其他符号不重名就行了。

要把这个菜单放到顶级的菜单栏上，我们必须为这个菜单选择另一个代表它的全局符号；这里我们使用 crossword。现在，只需要将菜单键位表绑定到一个自定义的事件序列 [menu-bar crossword] 就可以了。

```
(define-key crossword-mode-map [menu-bar crossword]
  (cons "Crossword" crossword-menu-map))
```

这次，绑定被放到了 crossword-mode-map 里，这会使我们能够访问到 crossword-menu-map 中的菜单项。事件符号 menu-bar 代表了全局的菜单栏。事件序列 [menu-bar crossword] 选中了 Crossword 菜单的键位表，而事件序列 [menu-bar crossword find-singleton] 表示用户通过菜单选中了“Find singleton”菜单项。

10.5　追踪未授权的修改

即使我们竭尽全力的防止用户对于 buffer 的不合法的修改，但是用户总是可以找到什么方法去修改它。这时屏幕上的填字游戏网格就不匹配 crossword-grid 中的数据结构了。我们如何才能恢复它呢?

一种方法是将一个函数添加到每次 buffer 发生改变都会触发的 after-change-functions（参照第四章中聪明的方式一节）里。如果改变是“未授权的”，我们必须将 buffer 与 crossword-grid 数据结构重新同步。

什么是“未授权的”呢？它是“授权”的反义词，所以让我们添加一个方法来“授权”buffer 中的改变。

```
(defvar crossword-changes-authorized nil
  "Are changes currently authorized?")
(make-variable-buffer-local 'crossword-changes-authorized)
(defmacro crossword-authorize (&rest subexprs)
  "Execute subexpressions, authorizing changes."
  '(let ((crossword-changes-authorized t))
     ,@subexprs))
```

这是一个可以在 buffer 发生改变的时候包裹函数体的宏。它将 crossword-changes-authorized 暂时设为 t，执行函数体，然后把 crossword-changes-authorized 重置为之前的值。默认的，改变都是未授权的。所以为了防止用户对 buffer 造成破坏，我们必须重写 crossword-insert-grid 和 crossword-update-display 来授权它们做出的更改：

```
(defun crossword-insert-grid (crossword)
  "Insert CROSSWORD into the current buffer."
  (crossword-authorize
   (mapcar 'crossword-insert-row crossword)))
(defun crossword-update-display (crossword)
  "Called after a change, keeps the display up to date."
  (crossword-authorize
   (let* ((coords (crossword-cursor-coords))
          (cousin-coords (crossword-cousin-position crossword
                                                    (car coords)
                                                    (cdr coords))))
     (save-excursion
       (crossword-place-cursor (car coords)
                               (cdr coords))
       (delete-char 2)
       (crossword-insert-cell (crossword-ref crossword
                                             (car coords)
                                             (cdr coords)))
```

```
        (crossword-place-cursor (car cousin-coords)
                                (cdr cousin-coords))
        (delete-char 2)
        (crossword-insert-cell (crossword-ref crossword
                                              (car cousin-coords)
                                              (cdr cousin-coords)))))))
```

然后我们必须向 after-change-functions 添加一个函数用来检测当 crossword-changes-authorized 不为真的时候发生的改变：

```
(defun crossword-after-change-function (start end len)
  "Recover if this change is not authorized."
  (if crossword-changes-authorized
      nil ; 如果改变是经过授权的则什么都不做
    recover somehow
    ))
(make-local-hook 'after-change-functions)
(add-hook 'after-change-functions
          'crossword-after-change-function)
```

要知道一个用户指令可能会造成很多改变，我们不能每次执行完一个命令就“尝试恢复”几次。这表示在当前的命令执行完成之后（可能发生了许多改变），我们应该检测有哪些未经授权的改变发生了，然后重新同步。因此我们还应该向 post-command-hook 中添加一个函数（在每次执行完一个用户指令之后执行一次）。

我们需要创建一个新的变量，crossword-unauthorized-change ，用来告诉我们当前的指令是否造成了未授权的改变。我们需要修改 crossword-after-change-function 来设置它，然后创建一个新的函数，crossword-post-command-function ，来测试它：

```
(defvar crossword-unauthorized-change nil
  "Did an unauthorized change occur?")
(make-variable-buffer-local 'crossword-unauthorized-change)

(defun crossword-after-change-function (start end len)
  "Recover if this change is not authorized."
  (if crossword-changes-authorized
      nil
```

```
    (setq crossword-unauthorized-change t)))
(defun crossword-post-command-function ()
  "After each command, recover from unauthorized changes."
  (if crossword-unauthorized-change
      resynchronize)
  (setq crossword-unauthorized-change nil))
```

然后把它们添加到 crossword--mode-setup 里：

```
(make-local-hook 'after-change-functions)
(add-hook 'after-change-functions
          'crossword-after-change-function)
(make-local-hook 'post-command-hook)
(add-hook 'post-command-hook
          'crossword-post-command-function)
```

对于重新同步，我们有两种选择：相信 buffer 的内容然后更新 crossword-grid 中的数据结构；或者相信 crossword-grid ，清除 buffer 的内容然后使用 crossword-insert-grid 重新插入网格。

表面来看，没有理由认为 buffer 里的可见内容会比我们内部的数据结构更可信，因为显然 buffer 会比数据结构更容易发生损坏。但是，至少有一大原因使我们至少应该试着去相信 buffer：撤销（undo）命令。如果用户执行了撤销，buffer 将会回滚到最后一个命令执行之前的状态。这是有意义的。但是这并不会回滚 crossword-grid 的状态。因此，我们应该使用“未授权改变鉴别器”重新解析 buffer 中的网格（我们知道我们能做到，因为我们已经约定了 crossword-parse-buffer 的定义）。如果失败了，那么很有可能是因为 buffer 的格式不正确了，那么我们应该擦除掉 buffer 然后插入一个正确的网格。

下面是 crossword-post-command-function 的完成体：

```
(defun crossword-post-command-function ()
  "After each command, recover from unauthorized changes."
  (if crossword-unauthorized-change
      (let ((coords (crossword-cursor-coords)))
        (condition-case nil
            (setq crossword-grid (crossword-parse-buffer))
          (error (erase-buffer)
                 (crossword-insert-grid crossword-grid)))
        (crossword-place-cursor (car coords)
```

```
                                    (cdr coords))))
  (setq crossword-unauthorized-change nil))
```

这个函数使用了 condition-case，一个与 unwind-protect（第八章中优雅的失败章节中第一次引入）类似的特殊结构。unwind-protect 看起来是这样的：

```
(unwind-protect
    body
  unwind ...)
```

它会执行 body，是否完成取决于执行过程中是否有错误产生。不管 body 是否成功完成，unwind 最后都会执行。

condition-case 和 unwind-protect 的区别在于 condition-case 包含着只在错误的时候执行的表达式。它使用起来是这样的：

```
(condition-case var
    body
  (symbol1 handler ...)
  (symbol2 handler ...)
  ...)
```

如果 body 因为“信号条件（signaled condition）”而终止，后面的处理子句之一将会执行来“捕获”那个错误。子句执行的条件是其 symbol 与信号条件相同。在这里，我们只关心称为 error 的信号条件（通过 error 函数发出），所以我们对于 condition-case 的使用看起来是这样的：

```
(condition-case var
    body
  (error handler ...))
```

如果 var 不为空，那么它就是当某个子句执行的时候，Emacs 用来存放当前错误的变量名称—也就是发出这个信号条件的 error 的参数。但是在我们的例子里，由于我们并不需要这个信息，所以 var 是 nil。

我们需要将 crossword-grid 赋值为 crossword-parse-buffer 执行的结果。如果解析失败，crossword-parse-buffer 发出一个错误信号，这会使 condition-case 在替换 crossword-grid 的值之前终止掉。如果这发生了，错误处理子句将会执行，擦除掉 buffer 并且插入正确的 crossword-grid 的拷贝。

不管何种情况，我们最后都会把光标放置到函数开始时记录的网格座标处；但是假如 buffer 的结构已经坏到连获取座标也失败了呢？因此我们应该有两次对于 `condition-case` 的调用：

```
(defun crossword-post-command-function ()
  "After each command, recover from unauthorized changes."
  (if crossword-unauthorized-change
      (condition-case nil
          (let ((coords (crossword-cursor-coords)))
            (condition-case nil
                (setq crossword-grid (crossword-parse-buffer))
              (error (erase-buffer)
                     (crossword-insert-grid crossword-grid)))
            (crossword-place-cursor (car coords)
                                    (cdr coords)))
        (error (erase-buffer)
               (crossword-insert-grid crossword-grid)
               (crossword-place-cursor 0 0))))
  (setq crossword-unauthorized-change nil))
```

外面的那个 `condition-case` 用来处理 `crossword-cursor-coords` 的错误。它会擦除掉 buffer，重新插入网格，然后把光标放置到左上角。里面的 `condition-case` 用来处理 `crossword-parse-buffer` ，擦除并且重新插入网格，然后重置之前记录的光标位置。

既然现在我们可以跟踪并且恢复 buffer 中未授权的修改，我建议从 `crossword-mode-map` 中移除掉对于所有按键绑定的捕获，

```
(define-key crossword-mode-map [t] 'undefined)
```

毕竟这有点太过头了，这会使得我们无法利用类似 `C-k` 和 `C-y` 这样的无害并且有益的命令。

既然填字游戏数据会被存储到文本文件里，那么用户就有可能使用其他编辑器去破坏它，或者使用 Emacs 但是不用 Crossword 模式。这些改变的大多数将会使得 Crossword 模式初始化的时候解析失败。

10.6　解析 Buffer

下面是一个 `crossword-parse-buffer` 的定义：

```
(defun crossword-parse-buffer ()
  "Parse the crossword grid in the current buffer."
  (save-excursion
    (goto-char (point-min))
    (let* ((line (crossword-parse-line))
           (size (length line))
           (result (make-crossword size))
           (row 1))
      (crossword--handle-parsed-line line 0 result)
      (while (< row size)
        (forward-line 1)
        (setq line (crossword-parse-line))
        (if (not (= (length line) size))
            (error "Rows vary in length"))
        (crossword--handle-parsed-line line row result)
        (setq row (+ row 1)))
      result)))
```

这个函数会调用 crossword-parse-line，它会解析一行文本并且转化成列表。列表的长度会告诉我们填字游戏网格的长宽（因为网格是方形的）。然后我们对这个 size 调用 crossword-parse-line，每次一行。每当解析完一行，我们通过调用 crossword--handle-parsed-line 来填充一行 result 中保存的数据结构。它的定义如下：

```
(defun crossword--handle-parsed-line (line row grid)
  "Take LINE and put it in ROW of GRID."
  (let ((column 0))
    (while line
      (cond ((eq (car line) 'block)
             (crossword-store-block grid row column))
            ((eq (car line) nil)
             (crossword-clear-cell grid row column))
            ((numberp (car line))
             (crossword-store-letter grid row column (car line))))
      (setq line (cdr line))
      (setq column (+ column 1)))))
```

下面是 crossword-parse-line，它是 crossword-parse-buffer 的主干：

```
(defun crossword-parse-line ()
  "Parse a line of a Crossword buffer."
  (beginning-of-line)
  (let ((result nil))
    (while (not (eolp))
      (cond ((eq (char-after (point)) ?#)
             (setq result (cons 'block result)))
            ((eq (char-after (point)) ?.)
             (setq result (cons nil result)))
            ((eq (char-after (point)) ??)
             (setq result (cons nil result)))
            ((looking-at "[A-Za-z]")
             (setq result (cons (char-after (point))
                                result)))
            (t (error "Unrecognized character")))
      (forward-char 1)
      (if (eq (char-after (point)) ?\ )
          (forward-char 1)
        (error "Non-blank between columns")))
    (reverse result)))
```

这里每次读取两个字符。第一个字符应该是一个井号（#）、点号（.）、问号（?，与. 一样处理），或者一个字母。cond 表达式告诉我们在每种情况下如何处理。如果这些都不是，则发出一个错误信号– “Unrecognized character”。否则，下一个字符则应该是用来分割网格的列的空格。再一次，如果它不是，那么触发错误。

结果会通过 cons 存储到 result 中，也就是说每一行的第一个元素会出现在列表的最后，第二个会出现在倒数第二，依此类推。所以函数最后要做的是调用 reverse 来得到正确次序的列表。

另一件事：如果一个 Emacs 模式只能用来处理特定格式的文本，那么需要给模式符号设置一个 special 属性：

```
(put 'crossword-mode 'mode-class 'special)
```

这会告诉 Emacs 不要将 Crossword 模式用作其他 buffer 的默认模式，因为它只能处理已经包含可解析的填字游戏网格的 buffer。

10.7　词语查找器

直到目前为止，Crossword 模式并不比一张图片好多少。它会记录你把什么字母放到了什么位置，但是对于谜题的设计者来说它并不会提供什么帮助。设计填字游戏谜题的真正难点并不在于记录每个格子里填写了什么；而在于找到能够与你选择的单词相匹配的词，例如你需要一个五个字母的单词而后三个字母需要是“fas”。

可以使用标准的 UNIX 工具来帮助你寻找合适的单词。UNIX 程序 grep，通过给定一个合适的正则表达式，可以帮助从词语文件里找到匹配的词语。大多数 UNIX 系统都在/usr/dict/words 下或者/usr/lib/dict/words 下有一个词语文件，或者在 GNU 系统里的/usr/local/share/dict/words。

如果词语文件中每个单词一行，那么可以通过下面的 UNIX 命令找到一个五个字母并且以“fas”结尾的单词：

```
grep -i '..fas$' word-file
```

（-i 告诉 grep 大小写敏感。）这个指令会返回我们结果，“sofas”。

如果我们只需要触发一次按键然后让 Emacs 帮助我们构建正确的正则表达式并且运行 grep 不是很棒吗？

下面就是它的工作方式。当光标在一个格子上时，`C-c h` 将会横向搜索适合的单词，`C-c v` 会纵向搜索。在这两种情况下，函数会从左到右，或者从上到下查找最近的障碍。中间的格子被用来构建正则表达式。空格或者“letter”格子变为点号（.）；字母变为它们自己。正则表达式以 `^` 开头，以 `$` 结尾。将这个正则表达式传递给 grep，结果返回到一个临时 buffer 里。

10.7.1　第一次尝试

为了简化，让我们先只设计这个命令的横向版本。让我们称它为 `crossword-hwords`。我们要做的第一件事是得到光标位置并且检测当前格子的类型。

```
(defun crossword-hwords ()
  "Pop up a buffer listing horizontal words for current cell."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (if (eq (crossword-ref crossword-grid
                           (car coords)
                           (cdr coords))
            'block)
        (error "Cannot use this command on a block"))
```

如果当前的格子是个障碍的话则终止。没有单词可以跨越一个障碍（不管是横向还是纵向）。否则的话：

```
(let ((start (- (cdr coords) 1))
      (end (+ (cdr coords) 1)))
```

我们使用 start 和 end 来记录当前格子左面和右面的第一个障碍。

```
(while (not (crossword-block-p crossword-grid
                               (car coords)
                               start))
  (setq start (- start 1)))
```

这会把 start 向左移动直到遇到一个障碍。crossword-block-p 认为网格的边界是由“障碍”围起来的，所以这个循环会保证当遇到网格的边界时停止。

```
(while (not (crossword-block-p crossword-grid
                               (car coords)
                               end))
  (setq end (+ end 1)))
```

end 与之前的 start 一样，只是把向左替换成了向右。

```
(let ((regexp "^")
      (column (+ start 1)))
  (while (< column end)
```

这几行用来准备拼装正则表达式，从 start 后面的格子开始，到 end 前面的格子结束。

```
(let ((cell (crossword-ref crossword-grid
                           (car coords)
                           column)))
  (if (numberp cell)
      (setq regexp (concat regexp
                           (char-to-string cell)))
    (setq regexp (concat regexp "."))))
```

这会检测 while 循环中的格子是否是一个字母。如果是的话，我们将它添加到正则表达式中；否则添加一个点号（.）。

（我们使用 char-to-string 来将一个字母 ?a 转变为“a”，因为只有字符串才能被传递给 concat 。）

然后我们递增 column 来进行下一次循环：

```
(setq column (+ column 1)))
```

在循环退出时，我们在正则表达式的最后添加一个 $：

```
(setq regexp (concat regexp "$"))
```

然后，我们创建一个 buffer 来获取 grep 的结果：

```
(let ((buffer (get-buffer-create "*Crossword words*")))
```

函数 get-buffer-create 会使用指定的名字返回一个 buffer 对象。如果已经存在叫这个名字的 buffer，则返回这个 buffer，否则创建一个。（如果你不希望重用旧的 buffer，你可以使用 generate-new-buffer 来直接创建一个。）

```
(set-buffer buffer)
```

我们暂时的选中 *Crossword words* buffer，使它成为“当前的”。set-buffer 的作用范围只到当前命令的结束，而并不会影响用户所认为的当前 buffer。（如果需要的话，我们可以调用 switch-to-buffer 。）

```
(erase-buffer)
```

这会清空 buffer，避免我们重用之前调用 crossword-hwords 遗留下来的 buffer。

现在调用 call-process 来执行 grep 程序：

```
(call-process "grep"
              nil t nil
              "-i" regexp
              "/usr/local/share/dict/words")
```

我们不应该直接通过名称 “grep” 来使用这个程序，更好的方式是通过一个变量—例如，crossword-grep-program —在上面的调用中替代 “grep”。如果另一个 grep 程序更适合，用户可以更改这个变量。我们可以对词语文件做同样的处理，使用一个变量 crossword-words-file 来代替直接的命名 /usr/local/share/dict/words 。

call-process 中的参数 nil ， t 和 nil 表示：

1. “这个程序不需要‘标准输入’。” 它的输入来自于后面命令行参数中的文件名。如果使用了一个非 nil 的参数，这个字符串需要指向一个作为输入的文件名。如果为 t，当前的 buffer 会被用作程序的输入。
2. “将输出发送到当前的 buffer”（例如，*Crossword words* buffer）。nil 表示 “丢弃输出”。0 表示 “丢弃输出并且马上返回（不等待程序执行完成）”。buffer 对象表示将输出发送到哪个 buffer。参数也可以是一个包含两个元素的列表，而每个元素都是我们刚才描述的参数之一。列表中的第一个参数表示在哪里存放程序的 “标准输出”。第二个元素表示在哪里存放程序的 “标准错误”。3. “不要在数据到来的时候马上刷新 buffer”（这会减慢程序的执行）。Emacs 会在程序结束之后再在 *Crossword words* buffer 中显式所有的输出。

call-processs 剩下的参数作为命令行参数传递给 grep：-i表示关闭大小写敏感；regexp，包含着我们之前拼接的正则表达式；而 /usr/local/share/dict/words 则是 grep 用来搜索的文件。

crossword-hwords 要做的最后一件事是展示 *Crossword words* buffer 所包含的 grep 的输出。这通过 display-buffer 来实现：

```
(display-buffer buffer))))))
```

这就是我们第一个版本的 crossword-hwords 。

如果你只希望查找在两个已经存在的障碍之间填充的单词的话，这个版本的 crossword-hwords 是不错的；但是有时你可能会根据需要寻找一些更短的单词并且插入一些障碍。例如，如果你有一个看起来这样的行：

```
. . . . . a d a c . . . .
```

而你按下了 C-c h ，你会得到一个提示 “asclepiadaceous”。但是你可能希望把这行变成这样：

```
. . . . # h e a d a c h e # #
```

问题是，crossword-hwords 只会计算正则表达式 “^.......adac....$ ”，而 “headache” 并不符合这个正则。

我们可以尝试移除掉正则中的 ^ 和 $ ，以及前面和后面的点，这样就剩下了 adac。如果把这个正则传递给 grep，它将会找到 “headache”。但是它也会找到 “tetracadactylity”，而它的长度比需要的长 1（而 adac 的位置无论如何也不对）。

10.7.2　第二次尝试

一种不错的方式是构建一个这样的正则：“^.?.?.?.?.?.?.?adac.?.?.?.?$ ”。每个 .? 代表了 0 个或 1 个字符；所以整个正则表达式会匹配 0-7 个字符，然后是 “adac”，后面跟着 0-4 个字符。这个表达式会包含 “headache” 而剔除 “tetracadactylity”。

让我们再尝试一次：

```
(defun crossword-hwords ()
  "Pop up a buffer listing horizontal words for current cell."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (if (eq (crossword-ref crossword-grid
                           (car coords)
                           (cdr coords))
            'block)
        (error "Cannot use this command on a block"))
    (let ((start (- (cdr coords) 1))
          (end (+ (cdr coords) 1)))
      (while (not (crossword-block-p crossword-grid
                                     (car coords)
                                     start))
        (setq start (- start 1)))
      (while (not (crossword-block-p crossword-grid
                                     (car coords)
                                     end))
        (setq end (+ end 1)))
```

直到现在，这与之前一样：start 和 end 指向两边的障碍。

现在让我们给这个函数引入一个新的概念：也就是正则表达式的核心（core）。我们将使用这个概念来匹配每个字母都必须一致的部分。

前面和后面的障碍并不是必须匹配的；它们是可选的。但是从第一个字母到最后一个字母必须匹配，即使中间有障碍。所以当我们构建匹配下面这行的正则时：

```
. . . bar . foo . . . .
```

“核心”是 bar.foo，而整个正则表达式前面有三个可选字符而后面有五个：^.?.?.?bar.foo.?.?.?.?.?$ 就是我们想要的。

这表示我们必须找到填字游戏网格的核心。在正则表达式中任何核心外的空白都需要转化成 .? 。任何核心内的空白都转化成 . （点号）。

我们将以 start 和 end 来开始我们的改进：

```
(let ((corestart (+ start 1))
      (coreend (- end 1)))
  (while (null (crossword-ref crossword-grid
                              (car coords)
                              corestart))
    (setq corestart (+ corestart 1)))
  (while (null (crossword-ref crossword-grid
                              (car coords)
                              coreend))
    (setq coreend (- coreend 1)))
```

这会把 corestart 向右移动而 coreend 向左移动来跳过空白格子。注意 start 和 end 间可能并没有“核心”的存在。在这个例子里，corestart 向 end 移动而 coreend 向 start 移动。这没有问题，因为下面这段代码中我们使用 corestart 和 coreend 的方式对这个特性是不敏感的：

```
(let ((regexp "^")
      (column (+ start 1)))
  (while (< column end)
    (if (or (< column corestart)
            (> column coreend))
        (setq regexp
              (concat regexp ".?"))
```

这里，如果我们还没有找到核心，或者我们已经过去了，我们会向正则中添加 .? 。注意如果没有核心的话，我们总是添加 .? 。[5]

[5]在没有“核心”的情况下触发 crossword-hwords 并不算是一个错误，但是应该提示用户这个

如果我们在核心中，我们的处理与之前一样—除了我们现在调用 egrep 而不是 grep，因为 grep 不理解? 语法而 egrep 理解：

```
(let ((cell (crossword-ref crossword-grid
                           (car coords)
                           column)))
  (if (numberp cell)
      (setq regexp (concat regexp
                           (char-to-string cell)))
    (setq regexp (concat regexp "."))))
   (setq column (+ column 1)))
   (setq regexp (concat regexp "$"))
   (let ((buffer (get-buffer-create "*Crossword words*")))
  (set-buffer buffer)
  (erase-buffer)
  (call-process "egrep"
  nil t nil
  "-i" regexp
  "/usr/local/share/dict/words")
  (display-buffer buffer)))))))
```

再次的，你可能会希望使用 crossword-egrep-program 和 crossword-words-file 来代替 egrep 和 /usr/local/share/dict/words 。实际上，本章剩下的部分将会采取这种方式。

命令 crossword-vwords--crossword-hwords 的纵向版本—大体上与 crossword-hwords 相同。定义并且抽取出两个函数公用代码的工作就留给读者自己做了。

10.7.3　异步 egrep

刚刚编写的 crossword-hwords 调用了 egrep，等待它的完成，然后显示出运行结果。但是假设你使用了不是 egrep 的其他程序；或者假设你将 crossword-words-file 设置为一个访问缓慢的网络上的地址。crossword-hwords 因此可能会需要一段时间来执行，而 Emacs 在这段时间里都是不可用的。

如果 crossword-hwords 只是启动 egrep 的执行，然后让它 “在后台运行”，而让用户能够继续与 Emacs 交互就好多了。为此，我们可以使用 Emacs 的异步进程（asynchronous process）对象。

情况，因为生成的正则会匹配字典中小于等于给定长度的所有单词—而这可能并不是用户希望看到的!

异步进程对象是一个用来表示在你的电脑上运行的另一个程序的 Lisp 数据结构。新的进程通过 start-process 创建，它与 call-process 很像（在前面章节中我们见过）。但是不像 call-process ，start-process 并不会等待程序执行完成。相反，它会返回一个进程对象。

对于进程对象我们能做很多事。你可以发送输入给正在运行的进程；你可以发送信号；你可以杀掉进程。你可以问进程查询状态（例如查询它正在运行还是已经退出了）。你可以将这个进程绑定到一个 Emacs buffer。

让我们使用 start-process 来重写 crossword-hwords 。为了节省空间，我们只关注 crossword-hwords 的结尾。下面是之前的版本：

```
(let ((buffer (get-buffer-create "*Crossword words*")))
  (set-buffer buffer)
  (erase-buffer)
  (call-process crossword-egrep-program
                nil t nil
                "-i" regexp
                crossword-words-file)
  (display-buffer buffer)))))))
```

下面是使用 start-process 的版本：

```
(let ((buffer (get-buffer-create "*Crossword words*")))
  (set-buffer buffer)
  (erase-buffer)
  (start-process "egrep"
                 buffer
                 crossword-egrep-program
                 "-i" regexp
                 crossword-words-file)
  (display-buffer buffer)))))))
```

不同之处只是我们改用了 start-process ,然后调整了参数的顺序。start-process 的第一个参数（例子中的 “egrep”）是 Emacs 内部一个用来引用进程的名称。（它并不必须是要运行的程序名称。）下一个是要接收输出的 buffer，如果存在的话；然后是要运行的程序，以及它的参数。

在程序运行之后，start-process 马上返回，也就是说 display-buffer 会马上执行。但是我们可能并不希望 *Crossword words* buffer 马上显示。我们希望它在 egrep 运行结束之后再显示。所以我们需要一个方法来得到进程什么时候退出。当那发生的时候，我们才希望调用 display-buffer 。

为此，我们需要对这个进程对象添加一个 sentinel。sentinel 是一个当进程状态改变时会调用的 Lisp 函数。我们对程序退出时的状态改变感兴趣；但是状态改变在进程收到信号之后也会发生。

下面是一个调用 start-process ,然后添加了当进程退出的时候显示 buffer 的 sentinel 的版本。为了安装 sentinel，我们必须保存 start-process 返回的进程对象然后把它传递给 set-process-sentinel ：

```
(let ((buffer (get-buffer-create "*Crossword words*")))
  (set-buffer buffer)
  (erase-buffer)
  (let ((process
         (start-process "egrep"
                        buffer
                        crossword-egrep-program
                        "-i" regexp crossword-words-file)))
    (set-process-sentinel process
                          'crossword--egrep-sentinel))))))))
```

我们可以这样定义 crossword--egrep-sentinel ：

```
(defun crossword--egrep-sentinel (process string)
  "When PROCESS exits, display its buffer."
  (if (eq (process-status process)
          'exit)
      (display-buffer (process-buffer process))))
```

调用进程 sentinel 时有两个参数：进程对象，以及一个用来描述状态改变的字符串。我们会忽略掉这个字符串。我们通过检测进程的状态来看它是否已经退出了。如果已经退出了，我们就显示进程 buffer，这可以通过 process-buffer 找到。这个 buffer 我们在调用 start-process 的时候传递过了。

假设我们不希望显示 buffer 等待 egrep 的退出，但是我们也不希望马上显示。相反，我们希望在第一个结果到来之后就显示它。为此，我们需要对进程对象安装一个 filter。

filter 是一个当进程有输出的时候就会调用的函数。当进程没有 filter 的时候，输出会进入到关联的 buffer 里。但是当 filter 存在的时候，filter 函数负责输出的去向。所以让我们稍微修改我们的例子，使用一个 filter 函数来 (a) 将输出放到 buffer 里然后 (b) 显示那个 buffer。

```
(let ((buffer (get-buffer-create "*Crossword words*")))
  (set-buffer buffer)
  (erase-buffer)
  (let ((process
         (start-process "egrep"
                        buffer
                        crossword-egrep-program
                        (set-process-filter process
                                            "-i" regexp
                                            crossword-words-file)))
        (set-process-filter process
                            'crossword--egrep-filter)
        (set-process-sentinel process
                              'crossword--egrep-sentinel))))))))
```

我们保留着 sentinel，这样保证了 egrep 退出之后会显示 buffer，即使并没有输出。

下面是我们对于 `crossword--egrep-filter` 的定义：

```
(defun crossword--egrep-filter (process string)
  "Handle output from PROCESS."
  (let ((buffer (process-buffer process)))
    (save-excursion
      (set-buffer buffer)
      (goto-char (point-max))
      (insert string))
    (display-buffer buffer)))
```

调用 filter 有两个参数：进程对象，以及一个刚刚到来的输出字符串。我们找到进程的 buffer 然后把输出写入到它的最后。然后我们通过调用 `display-buffer` 来确保 buffer 的显示。

因为 filter（以及 sentinel）可能会被调用很多次（这也就是异步编程的本质），我们必须确保这不会造成什么不好的副作用。这表示他们必须额外做一些同步函数不用关心的事。例如，每次命令结束之后，Emacs 会恢复之前选中 buffer 的记录；在命令的执行过程中，函数可能会在不影响用户当前可见 buffer 的情况下调用 `set-buffer` 来做出更改。恢复选中 buffer 只会发生在命令结束之后—而 `post-command-hook` 也差不多这时候执行。因为异步函数可能会在命令结束之后执行，因此任何对于 `set-buffer` 的调用最后可能都不会重置，这会

造成我们不希望看到的结果。这也就是为什么 crossword--egrep-filter 使用了 save-excursion 的原因。

关于 start-process 的另一件事。当 Emacs 创建进程的时候，它会通过 UNIX 的管道或者伪终端（pseudo-ttys，ptys）保持一个对于它的连接（通过它进行输入输出流）。管道对于像 egrep 这种不需要交互的进程来说更合适，而伪终端对于交互程序更合适—例如像 UNIX shell 这样的命令解析器。start-process 创建的连接的种类被变量 process-connection-type 控制— nil 表示使用管道，t 表示伪终端。虽然有点古怪，但是最好每次调用 start-process 的时候都用 let 暂时把 process-connection-type 设置为需要的值，例如：

```
...
(let ((process-connection-type nil))
  (start-process "egrep"
                 buffer
                 crossword-egrep-program
                 "-i" regexp crossword-words-file))
...
```

10.7.4　选择单词

现在让我们添加能够从 *Crossword words* buffer 中选择单词并且自动插入到填字游戏网格中的功能。

我们需要做的第一件事是在 *Crossword words* buffer 中存储一些额外的信息—即存储在 buffer 的局部变量中。如果我们希望在 buffer 中的一个单词上按下 RET 之后这个单词会填写到 Crossword buffer 中正确的位置上，那么 *Crossword words* 必须知道正确的 Crossword buffer 是哪个以及最终摆放在哪里。

下面是两个 buffer 间必须要传递的数据。

1. start + 1 的值—即单词在网格中开始的位置。
2. 当前的单词搜索是横向还是纵向。之前的例子中都限定为横向，但是要记住实际上是有两个方向的。
3. 正则表达式的“核心”的相关信息。要解释为什么这是必须的，让我们考虑一下我们前面的例子：网格的行看起来是这样的：

```
. . . . . a d a c . . . .
```

crossword-hwords 对这行生成的正则表达式是“^.?.?.?.?.?.?adac.?.?.?.?$”。“核心”是 adac，“前缀”是“.?.?.?.?.?.?.?”而“后缀”是“.?.?.?.?

”。当用户选择的时候，例如，从 *Crossword words* buffer 中选择了 adactyl，它应该放置到行的什么位置呢？它应该这样放置吗：

```
a d a c t y l a d a c . . . .
```

当然不是；它应该这样放：

```
. . . . . . . a d a c t y l .
```

为了在行中正确的放置单词，就很有必要知道前缀长 7 个字符，而正则的“核心”在单词 adactyl 的位置 0 处。通常，如果前缀有 p 个字符长，而核心能够在选择的单词的位置 m 处找到，那么我们在摆放单词的时候就应该跳过 p-m 个字符。

为了把这些存储在 *Crossword words* buffer 的局部变量里，以及使 RET 表示“选中光标所在处的单词”，让我们为这个 buffer 定义一个小的主模式。让我们称它为 crossword-words-mode ，如下所示：

```
(defvar crossword-words-mode-map nil
  "Keymap for crossword-words mode.")
(defvar crossword-words-crossword-buffer nil
  "The associated crossword buffer.")
(defvar crossword-words-core nil
  "The core of the regexp.")
(defvar crossword-words-prefix-len nil
  "Length of the regexp prefix.")
(defvar crossword-words-row nil
  "Row number where the word can start.")
(defvar crossword-words-column nil
  "Column number where the word can start.")
(defvar crossword-words-vertical-p nil
  "Whether the current search is vertical.")
(if crossword-words-mode-map
    nil
  (setq crossword-words-mode-map (make-sparse-keymap))
  (define-key crossword-words-mode-map "\r" 'crossword-words-select))
```

回车键在字符串中写作“\r ”。

```
(defun crossword-words-mode ()
  "Major mode for Crossword word-list buffer."
  (interactive)
  (kill-all-local-variables)
  (setq major-mode 'crossword-words-mode)
  (setq mode-name "Crossword-words")
  (use-local-map crossword-words-mode-map)
  (make-local-variable 'crossword-words-crossword-buffer)
  (make-local-variable 'crossword-words-core)
  (make-local-variable 'crossword-words-prefix-len)
  (make-local-variable 'crossword-words-row)
  (make-local-variable 'crossword-words-column)
  (make-local-variable 'crossword-words-vertical-p)
  (run-hooks 'crossword-words-mode-hook))
```

我们还没定义 `crossword-words-select` 。我们一会再来做。首先，让我们重写 `crossword-hwords` 来做两件事：

- 它必须保存正则的核心信息以及前缀的长度。为了简化，如果没有核心则提示错误并且终止操作。
- 当创建单词列表 buffer 的时候，必须使它进入 Crossword-words 模式然后设置那几个局部变量。

如下所示：

```
(defun crossword-hwords ()
  "Pop up a buffer listing horizontal words for current cell."
  (interactive)
  (let ((coords (crossword-cursor-coords)))
    (if (eq (crossword-ref crossword-grid
                           (car coords)
                           (cdr coords))
            'block)
        (error "Cannot use this command on a block"))
    (let ((start (- (cdr coords) 1))
          (end (+ (cdr coords) 1)))
      (while (not (crossword-block-p crossword-grid
                                     (car coords)
                                     start))
```

```
        (setq start (- start 1)))
      (while (not (crossword-block-p crossword-grid
                                     (car coords)
                                     end))
        (setq end (+ end 1)))
      (let ((corestart (+ start 1))
            (coreend (- end 1)))
        (while (null (crossword-ref crossword-grid
                                    (car coords)
                                    corestart))
          (setq corestart (+ corestart 1)))
```

直到这里，仍然与之前相同。

```
(if (= corestart end)
    (error "No core for regexp"))
```

这次，如果没有核心，则以错误终止。

```
(while (null (crossword-ref crossword-grid
                            (car coords)
                            coreend))
  (setq coreend (- coreend 1)))
(let ((core "")
      (column corestart)
      (regexp "^"))
```

我们这次由内向外构建正则，通过对核心求值开始：

```
(while (<= column coreend)
  (let ((cell (crossword-ref crossword-grid
                             (car coords)
                             column)))
    (if (numberp cell)
        (setq core (concat core
                           (char-to-string cell)))
      (setq core (concat core ".")))
    (setq column (+ column 1)))
```

现在 core 保存着正则的核心了。
然后生成正则的前缀：

```
(setq column (+ start 1))
(while (< column corestart)
  (setq regexp (concat regexp ".?"))
  (setq column (+ column 1)))
```

... 将 core 添加到前缀上：

```
(setq regexp (concat regexp core))
```

... 加上后缀：

```
(setq column (+ coreend 1))
(while (< column end)
  (setq regexp (concat regexp ".?"))
  (setq column (+ column 1)))
(setq regexp (concat regexp "$"))
```

现在让我们移动到单词列表 buffer，但是这次让我们把当前 buffer 记录在 crossword-buffer 中以在后面访问到它：

```
(let ((buffer (get-buffer-create "*Crossword words*"))
      (crossword-buffer (current-buffer)))
  (set-buffer buffer)
```

现在让我们把 *Crossword words* 置入 Crossword-words 模式：

```
(crossword-words-mode)
```

然后设置这些 buffer 局部变量：

```
(setq crossword-words-crossword-buffer
      crossword-buffer)
(setq crossword-words-core core)
(setq crossword-words-prefix-len (- corestart
```

```
                                    (+ start 1)))
(setq crossword-words-row (car coords))
(setq crossword-words-column (+ start 1))
(setq crossword-words-vertical-p nil)
```

剩下的就与之前的一样了。

```
(erase-buffer)
(let ((process
       (let ((process-connection-type nil))
         (start-process "egrep"
                        buffer
                        crossword-egrep-program
                        "-i" regexp
                        crossword-words-file))))
  (set-process-filter process
                      'crossword--egrep-filter)
  (set-process-sentinel process
                        'crossword--egrep-sentinel))))
```

现在所剩下的就是定义 `crossword-words-select` 。它的目的是找出光标所在的单词，找出核心在这个单词中的位置，然后找出这个单词应该摆放在填字游戏网格中的位置，然后将它放到那里。

```
(defun crossword-words-select ()
  (interactive)
  (beginning-of-line)
  (let* ((wordstart (point))
         (word (progn (end-of-line)
                      (buffer-substring wordstart
                                        (point)))))
```

现在 word 保存着选中的行中的单词。

下一步我们使用 `string-match` 来找到核心在 `word` 中的位置。

```
(corematch (string-match crossword-words-core
                         word))
```

现在 corematch 保存着核心在 word 中匹配的位置。

```
(vertical-p crossword-words-vertical-p)
```

这会把 buffer 局部变量 crossword-words-vertical-p 拷贝给临时变量 vertical-p ，因为我们需要在 Crossword buffer 中取回它（那里并没有定义 crossword-words-vertical-p ）。

```
(window (selected-window)))
```

这会记录包含着单词列表 buffer 的窗口。在这个函数的后面，我们会关闭这个窗口（但是并不销毁 buffer），因为用户在选择完单词之后大概就不需要它了。

```
(if (not corematch)
    (error "This word does not fit"))
```

理论上这不可能—除非用户自己修改了单词列表 buffer，所以最好还是检测一下。

```
(let ((row (if vertical-p
               (+ crossword-words-row
                  (- crossword-words-prefix-len corematch))
             crossword-words-row))
      (column (if vertical-p
                  crossword-words-column
                (+ crossword-words-column
                   (- crossword-words-prefix-len corematch)))))
```

现在 row 和 column 指出了在网格中我们应该放置单词的起始处。

```
(i 0))
```

我们使用 i 来遍历 word 的字符，每次向网格中添加一个。

```
(switch-to-buffer crossword-words-crossword-buffer)
```

这里使用了 switch-to-buffer 而不是 set-buffer 切换到 Crossword buffer。这表示在命令结束之后 Crossword buffer 仍然处于选中状态。

```
(while (< i (length word))
  (crossword-store-letter crossword-grid
                          row
                          column
                          (aref word i))
  (crossword-update-display crossword-grid
                            row
                            column)
  (setq i (+ i 1))
  (if vertical-p
      (setq row (+ row 1))
    (setq column (+ column 1)))))
```

这会把每个单词储存到网格里，并且按照需要横向或者纵向排列。在使用 crossword-store-letter 更新数据结构之后，通过调用 crossword-update-display 同步了显示。

当我们调用 crossword-update-display 时，我们并不希望更新包含着光标的格子；我们希望更新 row 和 colum 指向的刚刚存储了一个字母的格子。所以让我们提前约定，crossword-update-display 使用网格座标作为可选参数，并且在它们提供的情况下替换光标所在的位置。我们将在下面修改 crossword-update-display 。

最后，让我们删除 Crossword-words 窗口以使用户专注于 Crossword buffer。

```
(delete-window window)))
```

下面是一个使用了可选网格座标参数的 crossword-update-display 版本，如果可选参数未提供则使用光标所在的位置。

```
(defun crossword-update-display (crossword &optional row column)
  "Called after a change, keeps the display up to date."
  (crossword-authorize
   (if (or (null row)
           (null column))
       (let ((coords (crossword-cursor-coords)))
         (setq row (car coords)
```

```
              column (cdr coords))))
  (let ((cousin-coords (crossword-cousin-position crossword
                                                  row
                                                  column)))
    (save-excursion
      (crossword-place-cursor row
                              column)
      (delete-char 2)
      (crossword-insert-cell (crossword-ref crossword
                                            row
                                            column))
      (crossword-place-cursor (car cousin-coords)
                              (cdr cousin-coords))
      (delete-char 2)
      (crossword-insert-cell (crossword-ref crossword
                                            (car cousin-coords)
                                            (cdr cousin-coords)))))))
```

现在我们只需要再调整一件事：我们必须解决选中的单词的模糊对齐问题。

10.7.5　模糊对齐

假设你的网格中有这么一行：

```
# . . . f . #
```

然后你在这一行中按下了 `C-c h` 。`crossword-hwords` 生成的正则表达式是 `^.?.?.?f.?$` ；它的核心是 `f` 。

单词列表 buffer 会充满包含 “f” 的单词。你选中了 “fluff”。会发生什么呢?

当你选中 “fluff”，`crossword-words-select` 在 “fluff” 的位置 0 处找到了一处对于核心的匹配。这表示它将会试图以 “fluff” 的第一个字母来匹配格子中 “f”，看起来就像这样：

```
# . . . f l #
```

在这个例子里，我们不能使用核心所匹配的第一处。但是我们也不能使用最后一处，因为这会用 “fluff” 的最后一个字母来匹配格子中的 “f”，而这会把太多的字母放到左面：

```
# l u f f . #
```

我们必须使用“fluff”中的第二个“f”来匹配网格中的“f”。我们如何才能正确地做出这个选择呢?

答案是以前缀的长度来找到核心在单词中能够出现的最右的位置。这保证了单词在核心左侧的部分足够短，且最小化了右侧的字母数量。

例如，单词“fluff”包含着三处对于正则核心 f 的匹配。第一个位置为 0，第二个为 3，第三个为 4。正则前缀的长度为 3。所以“fluff”中对于 f 的最右匹配就应该小于等于 3，也就是第二个。

选择尽可能靠右的匹配使得我们在放置单词时能尽可能多的填充前缀。而这又会保证我们不会超出右边界。

因此我们应该将 crossword-words-select 的这一部分:

```
(let* (...
       (corematch (string-match crossword-words-core
                                word))
       ...
```

替换为:

```
(let* (...
       (corematch
        (let ((bestmatch nil)
              (index 0))
          (while (and index (<= index
                                crossword-words-prefix-len))
            (let ((match (string-match crossword-words-core
                                       word
                                       index)))
              (if (and match
                       (<= match crossword-words-prefix-len))
                  (setq bestmatch match
                        index (+ match 1))
                (setq index nil))))
          bestmatch))
       ...
```

下面解释它如何工作:

```
(let ((bestmatch nil)
      (index 0))
```

我们使用 bestmatch 来保存目前找到的最右位置而 index 来指示下一次搜索从哪里开始。循环在 index 为 nil 时终止（这与初始值 0 是不同的）。

```
(while (and index (<= index
                      crossword-words-prefix-len))
```

while 循环一直进行直到我们向右前进的太远了（也就是说与开始搜索的距离超过了 crossword-words-prefix-len ）。

```
(let ((match (string-match crossword-words-core
                           word
                           index)))
```

这里我们使用了 string-match 的可选的第三个参数，也就是从 word 中的何处开始搜索。

```
(if (and match
         (<= match crossword-words-prefix-len))
```

我们必须确保 match 在传递给 <= 之前不为 nil ，因为它只接受数字。

如果已经找到了一个匹配结果，那么记录它然后向右开始下一次搜索；否则，将 index 设为 nil 来退出循环。

```
(setq bestmatch match
      index (+ match 1))
(setq index nil))))
```

最后，将 bestmatch 的值返回给 corematch 。

```
bestmatch)
```

10.8　结语

我们可以继续向 Crossword 模式中添加功能，而且我也很难管住自己的手。例如，一旦网格满了，就计数网格中的方块并且生成横向单词和纵向单词的列表。提供以单词为单位移动光标的命令也是个不错的主意。

但是这就是目前为止我所编写的 Crossword 模式了。我需要面对这本书的时间点，而且，没人喜欢一个不知道该何时放弃一个玩具工程的程序员。

当然对你来说没有任何限制去完善 Crossword 模式。对于 Emacs 的探索同样如此，不管你选择任何方向。

附录 A　总结

现在你可以开始你的 Emacs Lisp 编程生涯了。我花费了多年的经验才学会了本书中讨论的技术和工具。

在我写本书的前言时，我说过本书并不会完全覆盖整个语言。Emacs Lisp 中有很多我们没有涉及的有趣的领域。例如我们还没使用 Emacs 的“可选显示(selective display)”的特性。可选显示允许你隐藏或者显示不同的行和部分。我们也没使用“文本属性（text properties）”。文本属性允许你将颜色和字体甚至 Lisp 动作关联到 buffer 中的文本上。我们也没试着去修改模式栏。我们也几乎没有涉及到 minibuffer 以及多种提示和补全方法。我们甚至没有提到定时器，apply 或者 funcall。我们绕过了 Emacs 整个“撤销（undo）”机制。

我们学到的是 Emacs Lisp 可能做到什么以及它们看起来是怎么工作的。我们研究了很多如何开发 Emacs Lisp 解决方案来应对问题的流程。我们对于如何开始，如何进行，哪里去寻找信息，以及哪些坑需要避免形成了一个良好的、坚实的感觉。

我们从实践中学习。相比于反复学习 Emacs Lisp 的各个方面，我的目标是使你能够快速的开始编写自己的 Lisp 代码并且能够自己去探索 Emacs 其他的部分。如果我成功了的话，Emacs Lisp 中其他仍然在逃的恐怖分子将不会恐吓到你。它们应该会使你胃口大开。

Happy hacking.

附录 B　Lisp 快速参考

在本章：

- 基础
- 数据类型
- 控制结构
- 代码对象

这个附录总结了 Emacs 中常用的 Lisp 语法，以及一些重要的 Lisp 函数。它并不会总结 Emacs 自身的一些特性，例如 buffer，钩子变量，键位表，模式，等等。要完整的查看 Emacs Lisp，请参照 The GNU Emacs Lisp Reference Manual。获取它的细节请参照附录E。

B.1　基础

Lisp 表达式是一个可以被计算的数据单位。表达式可能由子表达式构成，就像列表或者向量。

每个 Lisp 表达式在计算的时候都会返回一个值。大多数表达式都是自运算的，也就是说它们的值就是它们自己。

Lisp 表达式可以被认为是一个字面数据而不必求值。非自运算的表达式必须被引用（quoted）来避免求值而作为字面值使用。

符号 nil 表示否。它与空表()完全相同。其他的 Lisp 对象都为真，而符号 t 则是真的真正表示。

Emacs Lisp（不像其他方言）是大小写敏感的。

B.2　数据类型

B.2.1　数字

Emacs Lisp 支持整数和浮点数。它们的写法与你想的没什么两样：一个 10 进制的有一个可选负号以及一个可选小数点的字符串。有一些可以操作数字的

函数：

- `(numberp x)`：判断 x 是否为数字。
- `(integerp x)`：判断 x 是否为整数。
- `(zerop x)`：判断 x 是否为 0。
- `(= a b)`：判断两个数字是否相等。
- `(+ a b c ...)`：加。
- `(- a b c ...)`：减。

B.2.2　字符

在 Emacs Lisp 中问号用来指代单独的字符。例如，`?a`表示小写的 a。有些特殊字符，特别是用来起始其他 Lisp 表达式种类的那些，需要使用问号-反斜杠来转义，例如`?\"`，`?\(`和`?\)`。有些特殊字符可以使用反斜杠和字符构成。例如，`?\t`表示 tab，而`?\n`表示换行符。

字符的求值结果是它的 ASCII 值。例如，`?a`的计算结果是 97。实际上，整数值可以在任何需要的时候从字符转化；Emacs Lisp 并不区分这两种有什么不同，除了为字符提供了一些方便的方法。

- `(char-equal a b)`：判断两个字符是否相等。如果 case-fold-search 非空则忽略大小写。
- `(char-to-string c)`：创建一个只包含 c 的字符串。

B.2.3　字符串

字符串是一个字符的序列，写法上用双括号包裹起来，`"like this"`。如果字符串中存在双引号或者反斜杠，它必须用反斜杠进行转义，`"\"Like this,\" he said."`。字符串是自运算的。

Emacs，作为一个文本编辑器，有许多操作字符串的函数。下面是一些小例子：

- `(stringp x)`：判断 x 是否是字符串。
- `(string= s1 s2)`：判断两个字符串是否相等。
- `(string-lessp s1 s2)`：检测 s1 在 ASCII 值比较上是否比 s2 更小。
- `(concat a b c ...)`：将所有字符串拼接成一个新字符串。
- `(length s)`：返回字符串 s 的长度。
- `(aref s i)`：返回字符串 s 的第 i 个字符，从 0 开始。
- `(aset s i ch)`：设置字符串 s 的第 i 个字符为 ch。
- `(substring s from [to])`：截取从位置 from 到位置 to 处（如果忽略 to 则到结尾）的 s 的子字符串。

B.2.4　符号

符号是能够有一些与之关联的数据的名称。符号的名字是一个由字符组成的序列，它看起来不能像是数字，字符串，列表，向量，或者其他 Lisp 数据类型。

符号可以用作变量，函数名，或者自身作为一个值存在。符号的求值结果是它的变量值。

- `(symbolp x)`：判断 x 是否是一个符号。
- `(setq sym epr)`：将 sym 用作一个变量：并且将 expr 的值赋给 sym。
- `sym`：计算符号的变量值。
- `(defun sym ...)`：将 sym 用作函数名。
- `(sym arg1 arg2 ...)`：一个以符号开头的列表，表示调用名为 sym 的函数。

每个符号都有与其关联的属性列表。属性列表是一个映射表，它的键是 Lisp 符号而值为任何 Lisp 表达式。

- `(put sym key value)`：在 sym 的属性列表里，将 value 赋给 key。
- `(get sym key)`：从 sym 的属性列表里取得符号 key 对应的值，找不到则为 nil。

符号通常存储在内部的一个符号表里来防止重复符号的出现。可以显式地向符号表里添加条目或者创建但不向符号表中添加（因此可能会出现与其他符号重名的情况）。

- `(intern string)`：从内部符号表中返回一个名为 string 的符号。如果不存在，则创建一个。
- `(make-symbol string)`：创建一个新的名为 string 的符号。符号不会被加入到内部符号表里，并且与其他任何对象都不一样，即使是名字相同的对象。

B.2.5　列表

列表是 Lisp 的基础。一个列表是 0 个或多个其他 Lisp 表达式（也包括其他列表）的序列。列表的写法是用空格分隔它的子表达式；最外面用一个括号括起来。

Lisp 中的函数调用也用列表来表示。当执行的时候，列表中的第一个元素作为函数被调用，其它参数作为参数传入。

列表在内部被实现为一个 cons cells 的链表。因此访问列表中的一个元素需要遍历整个链表直到找到那个元素。

- `(listp x)`：判断 x 是否是列表。

- `(null x)`：判断 x 是否是空列表。
- `(consp x)`：判断 x 是否是非空列表。
- `(car list)`：返回 list 中的第一个元素（或者 cons cell 中的第一个元素）。
- `(cdr list)`：返回 list 剩下的部分（除第一个元素之外的部分，或者 cons cell 中的第二个元素）。
- `(list a b c ...)`：创建一个新列表，以参数作为子元素。
- `(cons a b)`:在列表 b 的开头插入 a(或者创建一个新的 cons cell `(a . b)`)。
- `(append list1 list2 ...)`：去掉每个子列表外面的括号（很高效），把所有元素粘贴在一起，然后再在外面添加一个括号形成一个新的列表。
- `(nth i list)`：返回 list 的第 i 个子表达式，从 0 开始。
- `(nthcdr i list)`：返回对 list 调用 n 次 cdr 的结果。

列表在第六章中有详细的描述。

B.2.6　向量

与列表一样，向量也是由 0 到多个子表达式构成的，写法上把小括号换成中括号。但是不像列表，向量的元素可以随机的访问（不必从头遍历访问内部的数据）。向量是自运算的。

当你编写一个向量时，它的子表达式都自动被引用了。要想先对元素求值再构建向量需要使用 vector 函数。

- `(vectorp x)`：判断 x 是否为向量。
- `(vector a b c ...)`：创建一个新向量，以参数作为子元素。
- `(length vector)`：返回向量的长度。
- `(aref vector i)`：返回向量的第 i 个子表达式，从 0 开始。
- `(aset vector i expr)`：将向量的第 i 个元素设置为 expr。

B.2.7　序列（Sequences）和数组（Arrays）

有些 Emacs Lisp 数据类型是有联系的。字符串和向量都属于数组。数组是一个数据元素的线性集合，它允许对于其元素的随机访问。字符串是字符组成的数组，而向量是由任意表达式组成的数组。函数 aref 和 aset 用来操作数组，这对于字符串和向量都有效。

序列是一个更宽泛的包括了数组和列表的数据结构。序列是一个数据元素的线性集合，句号。函数 length 对于列表，字符串和数组都有效。

- `(arrayp x)`：判断 x 是否为数组。
- `(sequencep x)`：判断 x 是否为序列。
- `(copy-sequence sequence)`：返回列表，字符串或者向量的一个拷贝。

B.3　控制结构

B.3.1　变量

要引用一个变量只需要使用它的名字（一个符号）。要给变量赋值，使用 setq。

```
(setq x 17) ; 将17赋值给变量x
x -> 17 ; 变量x的值
```

要创建只在一定范围内有效的临时变量，使用 let。

```
(let ((var1 value1)
      (var2 value2)
      ...)
  body1 body2 ...)
```

在 let 中，任何的 vars 赋值之前所有的 value 都会被计算且是无序的。变体`let*`（语法上与 let 相同）会先计算 valuei 并且赋值给 vari，然后再计算`valuei+1`。

B.3.2　顺序

要在只允许一个表达式的地方执行多个表达式，使用 progn。

```
(progn expr1 expr2)
```

顺序执行每个 expr。返回最后一个 expr 的值。
要按顺序执行表达式而返回第一个子表达式的值，使用 progn1。

B.3.3　条件

Emacs Lisp 中有两种条件表达式：if 和 cond。

```
(if test
    then
  else1 else2 ...)
```

对 test 求值。如果结果非 nil，对 then 求值。否则，依次对剩下的每个 else 表达式求值。返回最后一个表达式的求值结果。

```
(cond ((test1 body11 body12 ...)
       (test2 body21 body22 ...)
       ...))
```

对 test1 求值。如果结果非空，则顺序对每个 body1 求值。否则对 test2 求值。如果结果非空，则顺序对每个 body2 求值。依次对于每个 “cond 子句” 执行这个流程。返回最后一个表达式的求值结果。

一个通常的习惯是在最后放置一个截取所有的子句：

```
(cond ((test1 body11 body12 ...)
       (test2 body21 body22 ...)
       ...
       (t bodyn1 bodyn2 ...)))
```

逻辑运算符 and，or 以及 not 通常与条件语句一起工作—有时替代条件语句。

```
(and expr1 expr2 ...)
```

执行每个 expr 直到某个返回 nil（或者执行完所有子表达式），然后返回。返回值是计算的最后一个表达式的值。这是逻辑运算符 “与”，因为 and 只有当所有的子表达式都不为假时才为真。

表达式

```
(if expr1
    (if expr2
        ..
      (if exprn-1 exprn)))
```

和

```
(if (and expr1 expr2 ... exprn-1)
    exprn)
```

通常都会简写作

```
(and expr1 expr2 ... exprn-1 exprn)
```

表达式

```
(or expr1 expr2 ...)
```

执行每个 expr 直到某个返回非 nil（或者执行完所有子表达式），然后返回。返回值是计算的最后一个表达式的值。这是逻辑运算符“或”因为 or 只有当所有的子表达式都为假时才为假。

表达式

```
(if a a b)
```

通常简写作

```
(or a b)
```

最后，

```
(not expr)
```

返回 expr 的逻辑非。如果 expr 为真，返回 nil。如果 expr 为非，返回 t。（有趣的是，not 与 null 是同一个函数。）

B.3.4　循环

Emacs Lisp 有一个循环函数，while。

```
(while test
  body1 body2 ...)
```

对 test 求值。如果结果非 nil，逐次执行每个 body。然后重复。test 为 nil 的时候返回。

B.3.5　函数调用

要调用一个函数就是编写一个列表，它的第一个元素是函数名，剩下的元素是这个函数的参数。

```
(function arg1 arg2 ...)
```

这会使用给定的参数调用 function；返回 function 的结果。

B.3.6　字面数据

要使一个字面数据不受控制结构管辖—例如，避免对它求值—使用’ 引用（quote）它。

```
'expr -> expr
(quote expr) -> expr ; 等价
```

要使一个字面数据的列表中的子表达式能够求值，使用反引用（backquote），然后去引用（unquote）子表达式。

```
'(a b c) -> (a b c)
(backquote (a b c)) -> (a b c) ; 等价
`(a ,b c) -> (a value-of-b c)
```

要去引用一个 list 类型的表达式然后把它拼接到一个包含着反引用的模板里，使用拼接去引用符，“,@”。

```
(setq b '(x y z))
`(a ,@b c) -> (a x y z c)
```

B.4　代码对象

B.4.1　函数

函数是下面这种形式的一个列表：

```
(lambda (parameters ...)
  "documentation string"
  body1 body2 ...)
```

文档字符串 documentation string 是可选的。

当函数执行时，实参会被绑定到参数列表中的 parameters 中。参数列表中的关键字**&optional**表示后面的参数都是可选的。如果函数没有为可选参数指定值则默认为 nil。最后一个参数可能使用**&rest**修饰，表示所有剩下的未分配的参数都被放到一个列表里然后赋给这个参数。

函数执行的结果就是最后一个 body 表达式的结果。

使用 defun 来定义一个带名字的函数。

```
(defun name (parameters ...)
  "documentation string"
  body1 body2 ...)
```

这会创建一个 lambda 表达式然后赋值给符号 name 的函数值(`function value`)。这与符号 name 的变量值不同，所以函数名称与变量名称并不会冲突。

B.4.2　宏函数

宏函数是一个像 lambda 表达式一样的列表，但是并不使用 lambda 创建，而是使用 macro。当宏执行时，它的参数不会被计算。相反的，它们被用作字面形式来计算出一个新的 Lisp 表达式。然后这这个表达式会被求值。

要定义一个带名字的宏，就像 defun 那样使用 defmacro。

附录 C　调试和性能分析

在本章：

- 求值
- 调试器
- Edebug
- 性能分析器

这个附录描述了一些 Emacs 提供的用来测试和调试你的 Lisp 程序的一些工具。

C.1　求值

任何 buffer 中的 Lisp 表达式都可以被执行，只要把光标放到表达式的结尾然后按下C-x C-e (eval-last-sexp)。按键M-: (eval-expression) 会在 minibuffer 中提示你输入一个要求值的 Lisp 表达式。你还可以调用 eval-region 和 eval-current-buffer。

scratch buffer 通常处于 Lisp 交互模式（如果不是的话，你可以通过M-x lisp-interaction-mode RET来切换）。在这个模式里，C-j通常执行 eval-print-last-sexp，它与 eval-last-sexp 很像，除了它还会把结果插入到当前 buffer 里。在这个模式里C-M-x, eval-defun，会执行光标所在处的 “defun”。这里所说的 “defun” 的意义很宽泛；它表示的是一个以括号开头的闭合的 Lisp 表达式（如果存在的话）。最后，Lisp 交互模式还允许你输入部分 Lisp 符号然后使用M-TAB对其进行补全。

Lisp 表达式还能存储在文件里，然后通过 load，load-file，load-library 和 require 进行加载。

C.2　调试器

Emacs Lisp 有一个在指定情况下自动触发的内置调试模式。可以使用下面的手段进入调试器。debug-on-entry 这是一个命令。它会提示（并且补全）你输

入一个函数的名字。每当这个函数执行时，Emacs 将会进入调试器。

- debug-on-error：这是一个变量。如果非空，每当错误发出时，Emacs 将会进入调试器。
- debug-on-next-call：这是一个变量。如果非空，那么 Emacs 会在下一次一个表达式求值的时候进入调试器。
- debug-on-quit：这是一个变量。如果非空，每当“quit”信号发出时（例如用户按下了`C-g`），Emacs 将会进入调试器。

当调试器触发时，一个显示着 Lisp 栈的窗口将会显示出来。这个 buffer 称为`*Backtrace*`，每一行代表了一个挂起的函数调用，最上面表示着最近的调用。你可以看到挂起的 Lisp 表达式，查看不同位置的表达式和变量的值，并且强制函数返回指定的值。

下面是调试模式中的命令。

- c：离开调试器，继续执行被打断的代码。这在由错误触发调试器的时候不可用。
- q：离开调试器，终止挂起的运算。
- d：继续执行直到下一个函数调用，然后返回调试器。
- e：提示输入一个 Lisp 表达式在当前栈最上面的“帧”上执行。
- b：在当前函数返回的时候“break”。如果调试器是在调用一个函数的时候触发的，那么这个命令将会继续执行直到该函数将要返回的时候重新进入调试器。
- r：当要从一个函数返回时，提示你输入一个值作为那个函数的返回值（替换它原来的返回值）。

C.3 Edebug

Edebug 是一个比上一部分讲到的工具更强大的调试环境。它允许你单步调试实际运行的 Lisp 程序。Edebug 是一个完全由 Lisp 完成的惊艳的作品；它既是它的作者 Daniel LaLiberate 天赋的一种证明，也是 Emacs Lisp 强大表现力的一种证明—Emacs Lisp 提供了足够的对于其内部的访问能力来使这么一个工具变得可能。

这一部分只是 Edebug 的一个简要总结。要阅读完整的信息，请查看 The GNU Emacs Lisp Reference Manual 的 Edebug 部分。获取它的细节请参照附录E。

要使用 Edebug，你需要选择你要追踪的函数。每个函数都必须单独插桩（instrument），这表示需要以特殊的方式对其求值。函数 edebug-defun 就是用来执行这个任务的，它的用法与 eval-defun 类似。变量 edebug-all-defs 控制着载入 Edebug 时是否重新定义 eval-开头的那些命令以使它们也执行插桩操作。

在对目标函数插桩之后，你需要把这些定义记录在某个 buffer 中。你可以用通常的形式重新定义它们来去桩（uninstrument）。

Edebug 在任何插桩的函数被调用的时候激活。一个显示着函数定义的窗口会弹出来，最左边显示着一个小箭头表示当前停在哪一行。光标停在下一步要执行的表达式的前面（但是如果你需要的话，你可以移动光标，甚至隐藏这个 buffer，而不影响 Edebug 的操作）。

现在，你处在 Edebug 模式中并且可以执行下面的命令：

- c(Continue)：继续执行。
- q(Quit)：终止执行并且离开 Edebug。
- SPC(Single-step)：单步。如果 Edebug 停在一个变量或者一个常量上，则跳过它并且显示它的值。如果 Edebug 停在一个函数调用的开始处，则会进入该函数调用语句的内部。后续的单步将会经过每个参数，并且显示它们的值。如果所有的参数都求值完毕，那么单步就会用这些参数调用这个函数并且显示结果。如果这个函数也插桩了，则单步会进入该函数。每次单步，光标都会移动到代码中合适的位置。
- n(Next)：下一步。就像单步，但是会对嵌套的、插桩的函数求值而不会进入它们。
- e(Eval)：提示输入一个表达式在当前停止的程序上下文中执行。
- h(Here)：执行到此处。如果你把光标放置在代码中一处你希望停止的位置上，h 将会使程序继续执行直到当前位置。
- d(Display)：显示一个调用栈，就像 Emacs 的`*Backtrace*` buffer 那样（参照上一部分）但是并没有相对应的功能（Edebug 命令继续工作）。
- b(Breakpoint)：在光标处设置一个断点。程序将在执行到断点时停止。
- u(Unset)：去掉一个断点。
- x(Conditional breakpoint)：设置一个条件断点。你会被提示输入一个 Lisp 表达式。每次经过这个断点时，如果表达式为真，则程序停止。

Edebug 还有很多这里没有列出的能力，但是这些是最常用的功能。

C.4 性能分析器

性能分析（Profiling）一个程序是找出它运行每一部分所花时间的过程，一般来说是为了提高效率。Barry Warsaw 写了一个精巧的称为 ELP 的包来分析 Emacs Lisp。

就像 Edebug，ELP 也作用于“插桩”的函数。这通过命令 elp-instrument-function 来实现，它会提示输入一个函数名。还有一个 elp-instrument-package，它提示一个前缀。任何以该前缀开始的函数都会被插桩。

可以通过 elp-restore-function 和 elp-restore-all 来去桩。

要使用 ELP，只需要在插桩之后执行就好了。性能分析数据将会在后台记录。当你需要查看到目前为止的结果时，调用命令 elp-result。一个 buffer 将会弹出来显示每个被分析的函数执行的次数，一共花费了多少时间，以及平均时间。

对一个函数调用 elp-reset-function 来重置其调用的次数以及花费的时间；elp-reset-all 重置所有函数。

附录 D　分享你的代码

在本章：

- 准备源文件
- 文档
- 版权
- 发布

如果你写了一个 buling buling 的新 Emacs 模式，或者一个特性，或者一个游戏，或者其他什么，你应该秉承自由软件的精神与别人分享它，这可以通过向 gnu.emacs.sources 新闻组发布来实现。这个附录会向你描述分享 Emacs Lisp 代码的一些惯例。

D.1　准备源文件

在你向世界分享你的代码之前，最好对代码进行完整的测试，修复所有你遇到过的 bug。通过附录C学习更多关于测试和调试的信息。

在代码按照你的预想工作之后，你应该向每个源文件的头部添加一个注释块来描述这个文件，它的版权（如下所示），它的作者，它的版本信息，以及其他注释。下面是一个典型的开始：

```
;;; foretell.el -- predict what the user will do
;;; Copyright 1996 by Mortimer J. Hacker <mjh@mjh.net>
;;; Foretell is free software distributed under the terms
;;; of the GNU General Public License, version 2. For details,
;;; see the file COPYING.
;;; This is version 1.7 of 5 August 1996.
;;; For more information about Foretell, subscribe to the
;;; Foretell mailing list by sending a message to
;;; <foretell-request@mjh.net>.
```

文件的最后应该有这么一行：

```
;;; foretell.el ends here
```

这会在文件用 email 发出的时候帮助找到文件的边界（后面可能跟着签名和其他内容）。

如果你的包含有多个文件，通常要创建一个 README 文件来描述这个包，里面的文件，以及如何进行安装；然后使用 shar 程序把所有的文件整合进一个单独的发布文件里。如果你没有 shar，你可以获取到一个 GNU 版本的，具体参照附录E。

D.2　文档

最起码的，你的源文件应该在头部的注释块中有足够的注释，这样用户就能够明白这个文件的作用。如果你的代码是自解释的就更好了—也就是说，在你的函数和变量定义时恰当的使用了文档字符串。

如果你希望更严肃地编写文档，你可能会考虑为你的包创建一个 Textinfo 手册。Textinfo 是 GNU 系统的标准文档格式。Textinfo 文件可以通过 makeinfo 程序处理生成 Info 文件，这种树形的文件可以通过 Emacs 的 Info 模式来浏览。Textinfo 文件还能通过 TEX 排版系统处理来生成良好格式的可打印手册。

一个很不错的关于如何编写 Textinfo 手册的 Info 手册包含在 GNU textinfo 包里，其中还包含着 makeinfo。关于如何获取它以及 TEX，参照附录E。

D.3　版权

你可以自由的对你的代码设定任何你想要的版权条款，当然要合法。大多数 Emacs Lisp 包的作者选择让他们的软件遵循 GNU General Public License，一种由自由软件基金会（Free Software Foudation）发起的特殊种类的版权以使它们的软件“自由（free，在权限上，而不是价格上）”。遵循 GPL 的软件会被确保自由访问，但这对于某些情况并不合适，例如你要把你的软件发布到公共领域（这时，其他人可以合法的拷贝你的软件，修改它，声明这是它自己的，贩售它，并且拒绝继续分发代码）。

如果你希望让你的软件基于 GPL（一种幽默的称之为使你的软件“copy-lefting”的过程），你需要把 GPL 的条款放入到你的源文件里，或者在一个单独的文件里（通常是 COPYING）然后在你的每个源文件里引用它（参照本附录开头处的例子）。在 Emacs 中你可以通过按下`M-x` describe-copying `RET`来查看 GPL。

D.4　发布

当你的软件已经添加完版权，做好了文档，测试并且调试过，你就可以使用你喜欢的新闻阅读器将你的软件发布到 gnu.emacs.sources 新闻组里了。确保为你的帖子在 Subject: 域添加一行有用的描述，并且确保新闻组的读者在有问题或者建议的时候知道如何联系你。注意，向 gnu.emacs.sources 发布任何除 Emacs Lisp 代码之外的东西都非常糟糕。对于非代码的发布，使用 gnu.misc.discuss。

附录 E　获取以及编译 Emacs

在本章：

- 获取包
- 解包，编译，以及安装 Emacs

E.1　获取包

本书中描述的所有软件包，除了 TEX，都是自由软件基金会的 GNU 软件。它们的软件和其他包都可以通过匿名 FTP 从 ftp.gnu.ai.mit.edu 下的/pub/gnu 路径获取。有很多镜像网站，相关信息在 GNUinfo/FTP 下。

如果你不能从网上下载到，或者你希望有更简单的方式，你可以从自由软件基金会订购软件的发布版本。它们有磁盘，磁带，以及光盘版本。你还可以订购许多纸质版本的 GNU 手册，包括一些关于 Emacs 的，还有附录D中提到的 Textinfo 手册。要知道更多信息，包括价格，可以联系 FSF：

```
Free Software Foundation, Inc.
59 Temple Place - Suite 330
Boston, MA 02111-1307 USA
Telephone: +1-617-542-5942
Fax: +1-617-542-2652
Email: gnu@prep.ai.mit.ed
```

本书中提到的 FSF 提供的包有：

Emacs 编辑器本身，包括很多 Lisp 扩展。源文件以 emacs-x.y.tar.gz 的形式存在，其中 x 和 y 分别是最新版本的主版本和小版本号（当前是 19.34）。

Textinfo GNU 文档系统，包括 makeinfo 和一个关于如何编写 Textinfo 文档的手册。需要 TEX 来生成可打印的手册。形式为 textinfo-x.y.gz。当前版本为 3.7。

Emacs Lisp Reference Manual The Emacs Lisp Reference Manual 的 Textinfo 文档的源文件以 elisp-manual-19-x.y.tar.gz 的形式存在。（19 是

Emacs 的主版本。）当前手册的版本是 2.4。对于 Emacs Lisp 程序员来说，一个在线版本的手册是不可或缺的。

Shar untilities 包括 shar 和 unshar，用来创建和解包软件发布版本。形式为 sharutils-x.y.gz。当前版本是 4.2。shar 文件不需要使用 unshar 解包；只需要使用标准的 UNIX sh 命令就可以了。

Gzip Compression and decompression package，形式为 gzip-x.y.shar。当前版本为 1.2.4。

Tar 另一个用来创建和解包软件发布版本的程序，形式为 tar-x.y.shar.gz。当前版本为 1.11.8。注意不像大多数其他的实现，GNU tar 可以处理.tar.gz 文件而并不需要 gzip 的存在。

The Jargon File 在线 Hacker Jargon File（在前言中引用过）在 FSF 中也有提供，文件为 jargversion.txt.gz，当前版本为 400。也有 Info 格式的版本，jargversion.info.gz。它是黑客知识的宝库，会以书籍的形式 The New Hacker's Dictionary 定期发布。

你可以通过 TEX 用户组获取到 TEX。

```
TEX Users Group
1850 Union Street—Suite 1637
San Francisco, CA 94123 USA
Home page: http://www.tug.org/
Email: tug@tug.org
Fax: +1-415-982-8559
```

E.2 解包，编译，以及安装 Emacs

就像大多数 GNU 软件，Emacs 的解包，编译和安装也很简单。实际上，下面的说明对于几乎所有 GNU 软件包都有效，而不仅仅是 Emacs。

E.2.1 解包

如果你有一个压缩的 tar 文件（文件名以.tar.gz，.tar.Z，或者.tgz）并且你拥有 GNU tar，执行：

```
tar zxvf file
```

如果你没有 GNU tar，使用这个：

```
zcat file | tar xvf -
```

（你可以在 Gzip 包中找到 zcat。）如果你使用的 tar 来自于 SVR4 的衍生版本，你可能需要把 xvf 替换为 xvof。o 使你成为展开的文件的拥有者。（否则，拥有者为 tar 文件的发布者—你的电脑可能并不认识他。）

如果你有一个 shar 文件（文件名以.sh 或者.shar 结尾），执行

```
unshar file
```

或者简单的

```
sh file
```

如果 shar 文件本身压缩了（.Z 或者.gz），首先用`gzip -d`解压它。

E.2.2 编译和安装

首先在要编译的软件包的顶层目录，通过运行 configure 脚本来配置软件。

不同的软件包拥有不同的配置选项。使用`./configure --help`来查看这个包的选项。Emacs 的选项有：

`--with-gcc` 使用 GNU C 编译器来编译 Emacs。

`--with-pop` 编译支持 Post Office Protocol（POP），有时用来收取 email（对于用 Emacs 读取邮件的用户）

`--with-kerberos` 对 POP 使用 Kerberos 鉴权扩展。

`--with-hesiod` 使用 Hesiod 寻找 POP 服务器。

`--with-x` 支持 X 窗口。

`--with-x-toolkit` 更好的 X 窗口支持；使用工具箱部件。默认使用 Toolkit，但是–with-x-toolkit=motif 会使用 Motif 工具箱替代它。

你还可能希望查看 configure 脚本在执行的时候做了什么—这可能会有一会儿—所以你还可以使用`--verbose`选项。下面就是我经常执行的 configure：

```
./configure --verbose --with-x --with-x-toolkit
```

在配置完包之后，执行 make。这将会编译程序并且会花相当长的时间。

下一步，执行 make check。这会执行包里提供的自检程序。

假设软件正确编译并且通过了测试，使用 make install 来安装它。

表格清单

插图清单