第3章 实践：简单的数据库

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

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

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

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

3.1 CD和记录

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

---

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

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

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

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

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

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

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

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

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

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

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

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

3.2　录入 CD

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

来定义。名字中的星号是Lisp的全局变量命名约定。[1]

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

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

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

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

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

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

3.3 查看数据库的内容

无论何时，在REPL里输入*db*都可以看到*db*的当前值。

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

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

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

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

TITLE:     Roses
```

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

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

该函数如下所示。

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

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

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

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

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

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

或是：

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

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

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

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

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

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

你将得到下面的结果（为清楚起见作了折行处理）：

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

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

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

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

这件事究竟算酷还是恐怖，完全看你怎么想。

3.4 改进用户交互

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

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

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

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

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

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

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

里，就像这样：

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

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

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

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

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

来得到一个默认值0。

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

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

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

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

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

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

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

现在可以使用add-cds来添加更多CD到数据库里了。

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

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


3.5 保存和加载数据库

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

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

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

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

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

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

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

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

将数据加载回数据库的函数其形式也差不多。

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

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

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

3.6　查询数据库

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

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

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

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

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

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

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

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

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

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

在这种情况下，谓词是下面这个匿名函数：

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

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

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

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

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

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

假设要将整个表达式包装进一个接受艺术家名字作为参数的函数里，可以写成这样：

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

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

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

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

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

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

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

但这样看起来相当乱。所幸可以将匿名函数的创建过程包装起来。

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

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

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

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

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

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

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

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

是这样的：

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

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

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

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

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

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

前面给出同样的调用将产生下面的结果：

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

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

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

或是这样：

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

该函数看起来是这样的：

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

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

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

3.7　更新已有的记录——WHERE 再战江湖

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

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

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

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

---

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

```
some_var = some_boolean ? value1 : value2;
```

这样则是不合法的：

```
some_var = if (some_boolean) value1; else value2;
```

因为在这些语言里，if是一个语句，而不是一个表达式。

这样就可以了。

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

甚至可以更容易地添加一个函数来从数据库里删除记录。

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

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

3.8 消除重复，获益良多

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

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

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

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

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

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

① 你需要使用delete-rows这个名字而不是更明显的delete，因为已经有一个Common Lisp函数叫做DELETE了。Lisp包系统可以提供你处理这类名字冲突的途径，因此如果你真想要的话，还是可以得到一个叫做delete的函数的。但我还没准备好来解释关于包的事情。

② 如果你担心这些代码产生了内存泄露，那么大可放心：Lisp就是发明垃圾收集（以及相关的堆分配）的那个语言。至少这段代码里没有。*db*的旧值所使用的内存将被自动回收，假设没有其他地方持有对它的引用的话。

③ 我的一个朋友某一次采访一个从事编程工作的工程师，并问了他一个典型的采访问题："你怎样判断一个函数或者方法太大了？""这个嘛，"被采访者说，"我不喜欢任何比我头还大的方法。""你是说那些无法将所有细节都记到脑子里的方法？""不，我是说我把我的头放在显示器上，然后那段代码不应该比我的头还大。"

④ 考虑到检测一个变量是否为NIL是非常省事的，因此很难说检查关键字参数是否被传递的开销对整体性能有可检测到的影响。另一方面，这些由where返回的函数刚好位于任何select、update或delete-rows调用的内循环之中，它们将在数据库每一项上都被调用一次。不管怎么说，出于阐述的目的，我们必须要把它处理掉。

则可以将其改为：

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

可以在REPL里测试它。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

也可以使用“,@”在列表的中间插入东西。

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

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

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

那么变量clauses将包含这个列表。

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

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

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

回其展开式。因此可以像这样检查上一个where调用：

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

看起来不错。现在让我们实际试一下。

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

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

3.9 总结

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

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

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