第27章 实践：MP3数据库

本章来回顾在第3章里首次发掘的思想——从基本Lisp数据结构中构建出一个内存数据库。这次你的目标是保存那些你使用第25章的ID3v2库从一组MP3文件中解出的信息，而后在第28章和第29章里把这个数据库作为一个基于Web的流式MP3服务器的一部分。当然，这次你可以使用自从第3章以来所学到的语言特性来构建一个更专业的版本了。

27.1 数据库

第3章的那个数据库的主要问题是它只有一个表，也就是保存在变量*db*中的列表。另一个问题是代码并不清楚保存在不同字段中值的类型。在第3章里你避开了这个问题，通过使用相当通用的**EQUAL**方法来比较从数据库中选出的行的不同列的值，但如果你想要保存一些无法用**EQUAL**来比较的值，或者想要对数据库的行排序的话就会遇到麻烦了，因为不存在像EQUAL那样通用的比较函数。

这次你将定义一个类table来表示单独的数据库表，从而同时解决上述两个问题。每一个table实例都由两个槽构成——一个用来保存表的数据，而另一个保存关于表中各列的信息以供各种数据库操作使用。这个类如下所示：

```
(defclass table ()
  ((rows   :accessor rows   :initarg :rows :initform (make-rows))
   (schema :accessor schema :initarg :schema)))
```

和第3章一样，可以使用plist来表示单独的行，但这次你将创建一层抽象使得以后可以轻松调整实现细节而不会带来太多麻烦。并且这次你将把行保存在一个向量而非列表中，因为你将要支持的特定操作，比如通过数值索引对行进行随机访问，或要排序一个表，使用向量可以更高效地实现。

初始化rows槽的函数make-rows可以简单地封装在**MAKE-ARRAY**之外，从而构建一个空的可调整的带有填充指针的向量。

相关的包

在本章中你将用于开发代码的包如下所示：

```
(defpackage :com.gigamonkeys.mp3-database
```

```
  (:use :common-lisp
        :com.gigamonkeys.pathnames
        :com.gigamonkeys.macro-utilities
        :com.gigamonkeys.id3v2)
  (:export  :*default-table-size*
            :*mp3-schema*
            :*mp3s*
            :column
            :column-value
            :delete-all-rows
            :delete-rows
            :do-rows
            :extract-schema
            :in
            :insert-row
            :load-database
            :make-column
            :make-schema
            :map-rows
            :matching
            :not-nullable
            :nth-row
            :random-selection
            :schema
            :select
            :shuffle-table
            :sort-rows
            :table
            :table-size
            :with-column-values))
```

其中的:use部分可以让你访问那些从第15章、第8章和第25章定义的包中导出的名字所对应的函数和宏，而:export部分导出了该库提供的API，你将在第29章里用到它们。

```
(defparameter *default-table-size* 100)

(defun make-rows (&optional (size *default-table-size*))
  (make-array size :adjustable t :fill-pointer 0))
```

为了表示表的模式（schema），你需要定义另一个类column，其每个实例都含有关于表中一个列的信息：它的名字、如何比较列中值的等价性和顺序、默认值以及在向表中插入数据或查询表时用来正则化列中值的一个函数。schema槽将保存一个column对象的列表。该类的定义如下所示：

```
(defclass column ()
  ((name
    :reader name
    :initarg :name)

   (equality-predicate
    :reader equality-predicate
    :initarg :equality-predicate)

   (comparator
```

```
 :reader comparator
 :initarg :comparator)

(default-value
 :reader default-value
 :initarg :default-value
 :initform nil)

(value-normalizer
 :reader value-normalizer
 :initarg :value-normalizer
 :initform #'(lambda (v column) (declare (ignore column)) v))))
```

column对象的equality-predicate槽和comparator槽保存着几个函数，用来比较给定列中值的等价性和顺序。这样，含有字符串值的列可以使用**STRING=**作为其equality-predicate的值，用**STRING<**作为其comparator，而当列含有数字时可以使用“=”和“<”。

default-value槽和value-normalizer槽用在向数据库中插入行时，其中value-normalizer也用在查询数据库时。当你向数据库中插入新行时，如果没有给特定的列提供值，那么就可以使用保存在column的default-value槽中的值。然后，该值（无论是默认值还是其他值）将连同列对象一起传递给保存在value-normalizer槽中的函数，从而被正则化。你传递整个列以防value-normalizer函数可能需要使用与该列对象相关联的一些数据。（你将在下一节里看到相关的例子。）在将它们与数据库中的值进行比较之前，你也应该正则化传递给查询的值。

这样，value-normalizer的职责主要是返回一个可被安全和正确地传递给equality-predicate和comparator函数的值。如果value-normalizer不能找出一个适当的返回值，那么它将报错。

向数据库中保存值之前将其正则化也是为了节省内存和CPU时钟周期。举个例子，如果你有一个即将包含字符串值的列，但将会保存在该列中的不同字符串的数量较少，例如MP3数据库中的风格列，那么你可以通过使用value-normalizer来保留这些字符串（将所有**STRING=**的值都转化成单个字符串对象）。这样，无论表中有多少行，你只需要保存相当于所有不同的值那么多的字符串，并且你可以使用**EQL**而非相对缓慢的**STRING=**来比较列中的这些值。[①]

27.2 定义模式

这样，为了生成table实例，你需要构建一个column对象的列表。你可以使用**LIST**和**MAKE-INSTANCE**来手工构建这个列表。但你很快就会注意到你经常在生成许多带有相同比较器和等价谓词组合的列对象。这是因为比较器和等价谓词的组合本质上定义了一个列类型。如果可以有一种方式，只需给出这些类型的名字就可以让你表达出给定列是字符串列，而无需每次指定**STRING<**作为其比较符和**STRING=**作为其等价谓词，那就太好了。一种方式是定义一个广义函数

① 保留对象的一般理论是，如果你打算多次比较一个特定的值，那么值得花时间先保留它。value-normalizer在你将一个值插入到表中时运行一次，然后，如同你将要看到的，它会在每个查询的开始运行一次。由于查询可能涉及对表中的每一行都调用一次equality-predicate，因此保留这些值的摊余成本将快速地收敛到零。

make-column，如下所示：

```
(defgeneric make-column (name type &optional default-value))
```

现在，你可以在这个广义函数上实现通过EQL特化符特化在type上的方法，并返回填充了适当值的column对象。下面是为类型名string和number定义列类型的广义函数和方法：

```
(defmethod make-column (name (type (eql 'string)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'string<
   :equality-predicate #'string=
   :default-value default-value
   :value-normalizer #'not-nullable))

(defmethod make-column (name (type (eql 'number)) &optional default-value)
  (make-instance
   'column
   :name name
   :comparator #'<
   :equality-predicate #'=
   :default-value default-value))
```

下面的函数not-nullable用作string列的value-normalizer，它简单地返回给定值，除非它为NIL而会直接报错：

```
(defun not-nullable (value column)
  (or value (error "Column ~a can't be null" (name column))))
```

这很重要，因为如果STRING<和STRING=在NIL上被调用的话将会报错。在有问题的值进入表之前将其捕捉到，比等到你要使用它们时再报错要好很多。[①]

你在MP3数据库中用到的另一个列类型是interned-string，如同之前讨论的那样，它的值是被保留下来的。由于你需要一个哈希表来保留这些值，你应当定义一个column的子类interned-values-column，它增加了一个槽以保存那个用来做intern操作的哈希表。

为了实现实际的保留过程，你还需要为value-normalizer提供一个:initform以传递函数用来保留该列的interned-values哈希表中的值。并且由于保留这些值的一个主要原因是允许你使用EQL作为等价谓词，你还应该为equality-predicate添加一个值为#'eql的:initform。

```
(defclass interned-values-column (column)
  ((interned-values
    :reader interned-values
    :initform (make-hash-table :test #'equal))
   (equality-predicate :initform #'eql)
```

① 和通常一样，编程书籍中简要讲解的最大牺牲品是正确的错误处理。在产品代码中你可能想要定义你自己的错误类型，例如下面这样的，然后在报错时使用它：

```
(error 'illegal-column-value :value value :column column)
```

接下来，你可能会考虑在哪里放置再启动以便可以从这个状况中恢复。并且在最终给定的应用中，你可以在这些再启动中建立一些状况处理器来进行选择。

```
    (value-normalizer   :initform #'intern-for-column)))

(defun intern-for-column (value column)
  (let ((hash (interned-values column)))
    (or (gethash (not-nullable value column) hash)
        (setf (gethash value hash) value))))
```

然后你可以定义一个特化在名字interned-string上的make-column方法，来返回一个interned-values-column的实例。

```
(defmethod make-column (name (type (eql 'interned-string)) &optional default-value)
  (make-instance
   'interned-values-column
   :name name
   :comparator #'string<
   :default-value default-value))
```

有了这些定义在make-column上的方法，你现在可以定义一个函数make-schema，它从包括列名、列类型名以及可选默认值的列规范的列表中构建出一个column对象的列表。

```
(defun make-schema (spec)
  (mapcar #'(lambda (column-spec) (apply #'make-column column-spec)) spec))
```

例如，你可以为那个将用来保存从MP3中解出的数据的表定义下面这样的模式：

```
(defparameter *mp3-schema*
  (make-schema
   '((:file      string)
     (:genre     interned-string "Unknown")
     (:artist    interned-string "Unknown")
     (:album     interned-string "Unknown")
     (:song      string)
     (:track     number 0)
     (:year      number 0)
     (:id3-size number))))
```

为了生成一个实际用来保存关于MP3的信息的表，你将*mp3-schema*作为:schema初始化参数传给**MAKE-INSTANCE**。

```
(defparameter *mp3s* (make-instance 'table :schema *mp3-schema*))
```

27.3 插入值

现在你可以开始定义你的第一个表操作insert-row了，它接受由名字和值组成的plist以及一个表，然后在表中添加含有给定值的一行。大量的工作是在一个助手函数normalize-row中完成的，它为每个列构建了一个带有默认值并正则化了的plist，并尽可能使用来自names-and-values的值，否则会使用每个列的default-value。

```
(defun insert-row (names-and-values table)
  (vector-push-extend (normalize-row names-and-values (schema table)) (rows table)))

(defun normalize-row (names-and-values schema)
```

```
  (loop
     for column in schema
     for name  = (name column)
     for value = (or (getf names-and-values name) (default-value column))
     collect name
     collect (normalize-for-column value column)))
```

值得定义一个单独的助手函数normalize-for-column，它接受一个值和一个column对象并返回正则化的值，因为你将需要在查询参数上作同样的正则化处理。

```
(defun normalize-for-column (value column)
  (funcall (value-normalizer column) value column))
```

现在可以将这些数据库代码与前面章节的代码组合起来，从而构建一个从MP3文件中解出的数据的数据库了。你可以定义函数file->row，使用来自ID3v2库的read-id3从一个文件中解出ID3标签并将其转化成可以传给insert-row的plist。

```
(defun file->row (file)
  (let ((id3 (read-id3 file)))
    (list
     :file   (namestring (truename file))
     :genre  (translated-genre id3)
     :artist (artist id3)
     :album  (album id3)
     :song   (song id3)
     :track  (parse-track (track id3))
     :year   (parse-year (year id3))
     :id3-size (size id3))))
```

insert-row可以替你处理这些事情，因此你不必担心值的正则化问题。不过，你确实需要将track和year返回的字符串值转化成数字。ID3标签中的音轨号有时被保存成音轨号的ASCII表示，有时则保存成一个数字后跟左斜杠，然后是该专辑的音轨总数。你只关心实际的音轨号，因此如果有的话，你应当使用**PARSE-INTEGER**的:end参数来指定它只解析到左斜杠之前的位置。[①]

```
(defun parse-track (track)
  (when track (parse-integer track :end (position #\/ track))))

(defun parse-year (year)
  (when year (parse-integer year)))
```

最后，你可以将所有这些函数放在一起，再加上可移植路径名库的walk-directory和ID3v2库的mp3-p函数，从而定义出一个函数：它从给定目录里找出所有MP3文件并解出其中的数据，然后再把这些数据加载到数据库中。

```
(defun load-database (dir db)
  (let ((count 0))
```

① 如果任何MP3文件在音轨和年代帧里数据有格式错误，那么**PARSE-INTEGER**就可能会报错。处理该问题的一种方式是传给**PARSE-INTEGER**一个值为T的:junk-allowed参数，这将使它忽略掉跟在数字后面的任何非数字的垃圾，或是在字符串中没有数字时返回**NIL**。或者，如果你想试用一下状况系统，可以定义一个错误类型并在当数据的格式出现错误时在这些函数中报错，同时也建立一些再启动使这些函数得以恢复。

```
(walk-directory
 dir
 #'(lambda (file)
     (princ #\.)
     (incf count)
     (insert-row (file->row file) db))
 :test #'mp3-p)
(format t "~&Loaded ~d files into database." count)))
```

27.4　查询数据库

一旦你加载了带有数据的数据库，那么你需要一种方式来查询它。对于MP3应用来说，你需要一个比你在第3章里写得更加专业一些的查询函数。这一次你不仅要找出那些匹配特定条件的行，还要将结果限制在一些特定的列上，或是将结果限制在那些唯一的行上，同时还可能会在特定的列上排序这些行。为了保持关系型数据库的精髓，查询的结果将是一个含有你想要的行和列的新对象table。

你即将编写的查询函数select很大程度上出自结构化查询语言（SQL）的SELECT语句。它接受五个关键字参数：:from、:columns、:where、:distinct和:order-by。其中:from参数是你想要查询的table对象。:column参数指定了哪些列应当包含在结果中。其值或者是列名字的列表，或者是单独的列名，或者是默认值T，表示返回所有的列。如果你指定:where参数的话，那它应当是一个函数，其接受一行并在该行应当包含在结果中时返回真。接下来你将编写两个函数matching和in，它们可以返回适用于:where参数的函数。如果你指定:order-by参数的话，那它应当是一个列名的列表，结果将按照命名的列被排序。和:columns参数的情况一样，你可以只用一个名字来指定单一的列，这等价于一个含有同样名字的单元素列表。最后，:distinct参数是一个布尔值，它表明是否需要从结果中清除重复的行。:distinct的默认值为**NIL**。

下面是一些使用select的例子：

```
;; Select all rows where the :artist column is "Green Day"
(select :from *mp3s* :where (matching *mp3s* :artist "Green Day"))

;; Select a sorted list of artists with songs in the genre "Rock"
(select
  :columns :artist
  :from *mp3s*
  :where (matching *mp3s* :genre "Rock")
  :distinct t
  :order-by :artist)
```

select和它直接用到的助手函数的实现如下所示：

```
(defun select (&key (columns t) from where distinct order-by)
  (let ((rows (rows from))
        (schema (schema from)))

    (when where
      (setf rows (restrict-rows rows where)))
```

```
    (unless (eql columns 't)
      (setf schema (extract-schema (mklist columns) schema))
      (setf rows (project-columns rows schema)))

    (when distinct
      (setf rows (distinct-rows rows schema)))

    (when order-by
      (setf rows (sorted-rows rows schema (mklist order-by))))

    (make-instance 'table :rows rows :schema schema)))
(defun mklist (thing)
  (if (listp thing) thing (list thing)))

(defun extract-schema (column-names schema)
  (loop for c in column-names collect (find-column c schema)))

(defun find-column (column-name schema)
  (or (find column-name schema :key #'name)
      (error "No column:~a in schema:~a" column-name schema)))

(defun restrict-rows (rows where)
  (remove-if-not where rows))

(defun project-columns (rows schema)
  (map 'vector (extractor schema) rows))

(defun distinct-rows (rows schema)
  (remove-duplicates rows :test (row-equality-tester schema)))

(defun sorted-rows (rows schema order-by)
  (sort (copy-seq rows) (row-comparator order-by schema)))
```

当然，select中真正有趣的部分在于你如何实现函数extractor、row-equality-tester和row-comparator。

通过它们的用法你可以看出，这些函数中的每一个都必须返回函数。例如，project-columns使用由extractor返回的值作为提供给**MAP**的函数参数。由于project-columns的目标是返回一些仅含有特定列值的行，你可以推断出extractor将返回接受一行作为参数的函数，并返回一个仅含有传递的模式中指定的那些列的新行。下面是一种实现它的方式：

```
(defun extractor (schema)
  (let ((names (mapcar #'name schema)))
    #'(lambda (row)
        (loop for c in names collect c collect (getf row c)))))
```

注意你完成这项工作的方式——在闭包主体之外从模式中解出了所有的名字：由于闭包将被多次调用，你会希望它在每次调用时尽可能地少做事。

函数row-equality-tester和row-comparator的实现方式很相似。为了决定两行是否等价，你需要将用于每一列的相应等价谓词应用在适当的列值上。回顾第22章里的内容，在一对值测试失败以后，LOOP子句always立即返回**NIL**，否则将使整个**LOOP**返回**T**。

```
(defun row-equality-tester (schema)
  (let ((names (mapcar #'name schema))
        (tests (mapcar #'equality-predicate schema)))
    #'(lambda (a b)
        (loop for name in names and test in tests
           always (funcall test (getf a name) (getf b name))))))
```

排序两行稍微复杂一些。在Lisp中，比较操作符当它们的第一个参数应该排在第二个参数的前面时返回真，否则返回假。这样，**NIL**可能意味着第二个参数应该被排在第一个参数的前面，或者说明它俩其实相等。你希望你的行比较器具有相同的行为：当第一个行排在第二个的前面时返回真，否则返回假。

这样，为了比较两个行，你应当比较用于排序的列中的值，其中采用每个列的对应比较器。首先以来自第一个行的值作为第一个参数来调用比较器。如果比较器返回真，那就意味着第一行绝对应该排在第二个行的前面，所以你可以立即返回**T**。

但如果列比较器返回了**NIL**，那么你需要检测这是因为第二个值应当排在第一个值的前面，还是因为它们相等。因此你应当以相反的参数再次调用比较器。如果这次比较器返回真，那就意味着第二个列值排在了第一个列值的前面，并且因此第二个行也应该排在第一个行的前面，所以你可以立即返回**NIL**。否则，两个列值就是等价的，那么你应当继续比较下一个列。如果你通过了所有的列而始终没有遇到某个行的值赢得了比较，那么这两行就是等价的，于是你返回**NIL**。一个实现了该算法的函数如下所示：

```
(defun row-comparator (column-names schema)
  (let ((comparators (mapcar #'comparator (extract-schema column-names schema))))
    #'(lambda (a b)
        (loop
           for name in column-names
           for comparator in comparators
           for a-value = (getf a name)
           for b-value = (getf b name)
           when (funcall comparator a-value b-value) return t
           when (funcall comparator b-value a-value) return nil
           finally (return nil)))))
```

27.5　匹配函数

select的:where参数可以是任何接受行对象并在该行被包括在结果中时返回真的函数。不过在实践中，你很少需要用任意代码来表达查询条件。因此你应当提供两个函数matching和in，它们将用来构造查询函数，从而允许你表达常用类型的查询，并帮你处理每个列的正确等价性谓词和值正则化器的使用。

主要的查询函数构造器是matching，它返回一个函数匹配带有给定列值的行。你在早先的select例子里看到过它的用法。例如，对matching的如下调用：

```
(matching *mp3s* :artist "Green Day")
```

返回一个函数匹配:artist值为“Green Day”的行。你也可以传递多个名字和值，当所有列都匹

配时，返回的函数才算是匹配。例如，下面的例子返回了一个匹配艺术家为“Green Day”和专辑名为“American Idiot”的行的闭包：

```
(matching *mp3s* :artist "Green Day" :album "American Idiot")
```

你必须将整个表对象传给matching，因为它需要访问表的模式，以获得它所要匹配的那些列的等价谓词和值正则化器。

你可以从较小的函数中逐步构造出matching返回的函数，其中每个底层函数负责匹配一个列的值。为了构造出这些函数，你需要定义函数column-matcher，它接受column对象和你想要匹配的未经正则化的值，并返回接受单一行的函数，它在该行的给定列的值匹配给定值的正则化版本时返回真。

```
(defun column-matcher (column value)
  (let ((name (name column))
        (predicate (equality-predicate column))
        (normalized (normalize-for-column value column)))
    #'(lambda (row) (funcall predicate (getf row name) normalized))))
```

然后，你可以使用函数column-matchers为那些你关心的名字和值构造一个列匹配函数的列表：

```
(defun column-matchers (schema names-and-values)
  (loop for (name value) on names-and-values by #'cddr
     when value collect
       (column-matcher (find-column name schema) value)))
```

现在你可以实现matching了。再次注意：你应尽可能多地在闭包之外做事，以确保有些事情只做一次而不用在表的每一行上都做。

```
(defun matching (table &rest names-and-values)
  "Build a where function that matches rows with the given column values."
  (let ((matchers (column-matchers (schema table) names-and-values)))
    #'(lambda (row)
        (every #'(lambda (matcher) (funcall matcher row)) matchers))))
```

这个函数就像一个闭包的迷宫，但值得花点儿时间来思考一下作为第一类对象的函数会给编程带来多少灵活性。

matching的职责是返回一个函数，它将在表的每一行上被调用来检测其是否应被包含在新表中。因此，matching返回带有单个参数row的闭包。

现在回想一下，函数**EVERY**可以接受谓词函数作为其第一个参数，并当且仅当该函数应用在作为**EVERY**第二个参数传递进来的列表的每一个元素上均为真时才返回真。不过在本例中，你传递给**EVERY**的列表本身是一个由函数组成的列表，即列匹配器。你想要知道的是，每个列匹配器在你当前测试的行上被调用时，是否均返回真。因此，作为**EVERY**的谓词参数，你传递了另一个闭包给它，该闭包向列匹配器传递当前行的**FUNCALL**调用。

另一个你偶尔会觉得有用的匹配函数是in，它返回一个函数匹配那些特定列从给定的值集合中取值的行。你将定义in来接受两个参数：一个列名和一个含有你想要匹配的那些值的表。例如，假设你想要在MP3数据库中找出所有与Dixie Chicks的歌曲同名的歌曲。你可以像下面这样通过

in和一个子select写出这个where字句：[1]

```
(select
  :columns '(:artist :song)
  :from *mp3s*
  :where (in :song
             (select
               :columns :song
               :from *mp3s*
               :where (matching *mp3s* :artist "Dixie Chicks"))))
```

尽管查询更复杂了，但in本身的定义却比matching要简单得多。

```
(defun in (column-name table)
  (let ((test (equality-predicate (find-column column-name (schema table))))
        (values (map 'list #'(lambda (r) (getf r column-name)) (rows table))))
    #'(lambda (row)
        (member (getf row column-name) values :test test))))
```

27.6　获取结果

select返回了另一个table，因此你需要思考一下如何才能得到表中单独的行和列值。如果你确定将永不改变在表中表达数据的方式，那么就可以直接把表结构作为API的一部分，也就是说，该table中带有一个rows槽，类型为plist构成的向量，然后使用所有正常的Common Lisp函数操作向量和plist来得到表中的值。但这些表示可能确实是你以后会改变的内部细节。另外，你也不希望其他代码可以直接操作这些数据结构，例如，你不希望任何人使用SETF在行中放置一个未经正则化的列值。因此，定义一些抽象来提供你想要支持的操作就会是个好主意了。如果你决定以后改变表的内部表示，就只需要改变这些函数和宏的实现。尽管Common Lisp并不能使你绝对避免人们获得“内部”数据，但通过提供一个官方API你至少可以清楚地表明边界在哪里。

你需要对查询结果做的最常见的事情，也许就是在各个行上迭代并解出特定列的值。因此，你需要提供一种方式来同时做到这两件事，而无需直接用rows向量或是GETF来获取一个行中的列值。

目前这些操作都很容易实现，它们几乎就是包装在没有这些抽象时你编写的代码之上的。你

---

① 这个查询也会返回所有Dixie Chicks的歌曲。如果你想把查询限制在除Dixie Chicks之外的其他艺术家的歌曲上，那么就需要一个更复杂的:where函数。由于:where参数可以是任何函数，所以这确实是可能的。你可以通过下列查询移除Dixie Chicks自己的歌曲：

```
(let* ((dixie-chicks (matching *mp3s* :artist "Dixie Chicks"))
       (same-song
        (in :song (select :columns :song :from *mp3s* :where dixie-chicks)))
       (query
        #'(lambda (row)
            (and (not (funcall dixie-chicks row)) (funcall same-song row)))))
  (select :columns '(:artist :song) :from *mp3s* :where query))
```

这样显然不是很方便。如果你打算编写一个需要做很多复杂查询的应用程序，那么你会考虑设计更加复杂的查询语言。

可以提供两种方式在一个表的行上迭代：宏do-rows用来提供基本的循环构造，函数map-rows可以构造出一个列表，含有将一个函数应用在表的每一行时所得到的结果。[1]

```
(defmacro do-rows ((row table) &body body)
  `(loop for ,row across (rows ,table) do ,@body))

(defun map-rows (fn table)
  (loop for row across (rows table) collect (funcall fn row)))
```

为了得到一行中各列的值，你应该编写一个函数column-value，它接受一个行和一个列名并返回对应的值。再一次，这只是对你本该自行编写的代码的简单封装。但如果你以后改变了表的内部表示，那么column-value的用户可以不必受到影响。

```
(defun column-value (row column-name)
  (getf row column-name))
```

尽管column-value对于获取列的值来说已经足矣，但你会经常想要一次性得到多个列的值。因此你可以提供一点儿语法糖，即宏with-column-values，它将一组变量绑定到通过适当的关键字名从一个行中解出的值上。这样，你可以将下面的写法：

```
(do-rows (row table)
   (let ((song (column-value row :song))
         (artist (column-value row :artist))
         (album (column-value row :album)))
     (format t "~a by~a from~a~%" song artist album)))
```

简单替换为：

```
(do-rows (row table)
  (with-column-values (song artist album) row
    (format t "~a by ~a from ~a~%" song artist album)))
```

再一次，如果你使用第8章的once-only宏，实际的实现并不复杂。

```
(defmacro with-column-values ((&rest vars) row &body body)
  (once-only (row)
    `(let ,(column-bindings vars row) ,@body)))

(defun column-bindings (vars row)
  (loop for v in vars collect `(,v (column-value ,row ,(as-keyword v)))))

(defun as-keyword (symbol)
  (intern (symbol-name symbol) :keyword))
```

最后，你应当提供一种抽象方法来获取一个表中所有行的个数，并通过数值索引来访问指定行。

```
(defun table-size (table)
  (length (rows table)))
```

[1] 在Common Lisp被标准化以前，MIT.实现的LOOP版本含有一种机制来扩展LOOP语法以支持在新数据结构上的迭代。一些从该代码树上继承了LOOP实现的Common Lisp实现可能仍然支持这一功能，从而使do-rows和map-rows变得不再是必需的了。

```
(defun nth-row (n table)
  (aref (rows table) n))
```

27.7 其他数据库操作

最后，我们来实现其他一些将在第29章里用到的数据库操作。前两个类似于SQL DELETE语句。函数delete-rows用来从一个表中删除匹配特定条件的行。和select一样，它接受:from和:where关键字参数。和select不同的是，它并不返回新表——它实际修改了作为:from参数传递的表。

```
(defun delete-rows (&key from where)
  (loop
     with rows = (rows from)
     with store-idx = 0
     for read-idx from 0
     for row across rows
     do (setf (aref rows read-idx) nil)
     unless (funcall where row) do
       (setf (aref rows store-idx) row)
       (incf store-idx)
     finally (setf (fill-pointer rows) store-idx)))
```

出于对效率的兴趣，你可能想要编写一个单独的函数来从表中删除所有的行。

```
(defun delete-all-rows (table)
  (setf (rows table) (make-rows *default-table-size*)))
```

其余的表操作并没有映射到正常的关系型数据库操作中，但它在MP3浏览器应用中非常有用。首先是一个在表中直接排序所有行的函数。

```
(defun sort-rows (table &rest column-names)
  (setf (rows table)
        (sort (rows table) (row-comparator column-names (schema table))))
  table)
```

另一方面，在MP3浏览器应用中，你需要一个直接在表中打乱所有的行的函数，它用到了第23章的nshuffle-vector。

```
(defun shuffle-table (table)
  (nshuffle-vector (rows table))
  table)
```

最后，再一次为了MP3浏览器，你应当编写一个函数来选择*n*个随机行，然后作为新表返回。它也用到了nshuffle-vector，另外还有一个版本的random-sample，后者基于我在第20章里讨论过的Donald Knuth的《计算机程序设计艺术，卷2：半数值算法（第3版）》中的算法S。

```
(defun random-selection (table n)
  (make-instance
   'table
   :schema (schema table)
   :rows (nshuffle-vector (random-sample (rows table) n))))
```

```
(defun random-sample (vector n)
  "Based on Algorithm S from Knuth. TAOCP, vol. 2. p. 142"
  (loop with selected = (make-array n :fill-pointer 0)
     for idx from 0
     do
       (loop
          with to-select = (- n (length selected))
          for remaining = (- (length vector) idx)
          while (>= (* remaining (random 1.0)) to-select)
          do (incf idx))
       (vector-push (aref vector idx) selected)
     when (= (length selected) n) return selected))
```

有了这些代码，你就可以在第29章里构建一个用于浏览MP3文件集合的Web接口了。但在此之前，你还需要实现服务器中使用Shoutcast协议流式播放MP3的部分，这正是下一章的主题。