第29章 实践：MP3浏览器

构建MP3流应用的最后一步是编写一个Web接口，从而允许用户查找他们想听的歌曲，并且将它们添加到一个播放列表中，这样当用户的MP3客户端请求该流的URL时，Shoutcast服务器将会播放指定的歌曲。为了开发应用的这个组件，你需要把一些前面几章的代码整合起来：MP3数据库、第26章的define-url-function宏，当然还有Shoutcast服务器本身。

29.1 播放列表

接口背后的基本思想是每个MP3客户端连接到Shoutcast服务器上，获取它们自己的播放列表(playlist)，将其作为Shoutcast服务器所需的歌曲源。播放列表还将提供超出Shoutcast服务器需要之外的功能：用户将通过Web接口来向播放列表中添加歌曲，删除已在播放列表中的歌曲以及通过排序和乱序来重新调整播放列表。

你可以像下面这样来定义一个表示播放列表的类：

```
(defclass playlist ()
  ((id           :accessor id           :initarg :id)
   (songs-table  :accessor songs-table  :initform (make-playlist-table))
   (current-song :accessor current-song :initform *empty-playlist-song*)
   (current-idx  :accessor current-idx  :initform 0)
   (ordering     :accessor ordering     :initform :album)
   (shuffle      :accessor shuffle      :initform :none)
   (repeat       :accessor repeat       :initform :none)
   (user-agent   :accessor user-agent   :initform "Unknown")
   (lock         :reader   lock         :initform (make-process-lock))))
```

播放列表的id是其关键字，你从请求对象中解出它并传递给find-song-source来查询一个播放列表。你实际上并不需要将其保存在playlist对象中，但如果可以从一个任意的播放列表对象中找出它的id是什么的话，这会使调试更加方便。

播放列表的核心是songs-table槽，它用来保存一个table对象。用于这个表的模式将和用于主MP3数据库的模式相同。用来初始化songs-table的函数make-playlist-table十分简单：

```
(defun make-playlist-table ()
  (make-instance 'table :schema *mp3-schema*))
```

包定义

可以使用下列DEFPACKAGE来定义用于本章中代码的包：

```
(defpackage :com.gigamonkeys.mp3-browser
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.html
        :com.gigamonkeys.shoutcast
        :com.gigamonkeys.url-function
        :com.gigamonkeys.mp3-database
        :com.gigamonkeys.id3v2)
  (:import-from :acl-socket
                :ipaddr-to-dotted
                :remote-host)
  (:import-from :multiprocessing
                :make-process-lock
                :with-process-lock)
  (:export :start-mp3-browser))
```

由于这是一个高阶应用，它用到了许多底层包。它还从ACL-SOCKET包中导入了三个符号，并从MULTIPROCESSING包中导入了其余两个，这是因为它只需要这两个包中导出的5个符号而不需要其他的139个符号。

通过将歌曲的列表保存在一个表中，可以使用第27章的数据库函数来操作播放列表：你可以用insert-row向播放列表中添加歌曲，用delete-rows删除歌曲以及用sort-rows和shuffle-table重排播放列表。

current-song和current-idx槽用来跟踪当前正在播放哪首歌曲：current-song是实际的song对象，而current-idx是song-table中代表当前歌曲的行的索引。你将在29.3节中看到如何在每当current-idx改变时确保更新current-song。

ordering和shuffle槽保存关于songs-table中的歌曲顺序的信息。其中ordering槽保存一个关键字来告诉songs-table在其不是乱序时应当怎样排序。合法的值包括:genre、:artist、:album和:song。shuffle槽保存下列关键字之一：:none、:song或:album。它指定了songs-table应当如何被乱序。

repeat槽也保存一个关键字，:none、:song或:all之一，指定了播放列表的重复模式。如果:repeat是:none，那么在songs-table的最后一首歌播放完以后，current-song回滚到一个默认的MP3上；当:repeat为:song时，播放列表将不断地返回到相同的current-song上；而如果是:all的话，在最后一首歌结束以后current-song将回到第一首歌上。

user-agent槽保存MP3客户端在其对流的请求中发送的User-Agent头。你纯粹是为了Web接口才保存这个值的——User-Agent头标识了产生请求的程序，因此可以将该值显示在列出所有播放列表的页面上，从而容易看出当有多个用户连接时播放列表与连接的对应关系。

最后，lock槽中保存了一个由函数make-process-lock创建的“进程锁”，该函数是Allegro的MULTIPROCESSING包的一部分。你将需要在操作playlist对象的特定函数中用到这个锁，以

确保每次只有一个线程在操作给定的播放列表对象。可以定义下面的宏来包装一组需要在保持一个播放列表锁的情况下进行处理的代码，该宏是从来自MULTIPROCESSING的with-process-lock宏构建出来的：

```
(defmacro with-playlist-locked ((playlist) &body body)
  `(with-process-lock ((lock ,playlist))
     ,@body))
```

其中的with-process-lock宏要求获得对给定进程锁的排他访问，然后再执行其主体Lisp形式，最后再释放锁。在默认情况下with-process-lock允许递归加锁，这意味着同一个线程可以安全地对同一个锁对象加锁多次。

29.2 作为歌曲源的播放列表

为了将playlist用作Shoutcast服务器的歌曲源，需要在第28章的广义函数find-song-source上实现一个方法。由于你打算拥有多个播放列表，需要一种方式来为连接到服务器的每个客户端找出那个正确的播放列表来。具体的映射部分很简单——你可以定义一个变量来保存EQUAL哈希表，并用它来将一些标识符映射到playlist对象上。

```
(defvar *playlists* (make-hash-table :test #'equal))
```

你还需要定义一个进程锁来保护对这个哈希表的访问，如下所示：

```
(defparameter *playlists-lock* (make-process-lock :name "playlists-lock"))
```

然后定义一个函数根据给定ID来查询一个播放列表，如果必要的话就创建一个新的playlist对象，并用with-process-lock来确保每次只有一个线程在操作哈希表。[①]

```
(defun lookup-playlist (id)
  (with-process-lock (*playlists-lock*)
    (or (gethash id *playlists*)
        (setf (gethash id *playlists*) (make-instance 'playlist :id id)))))
```

然后你就可以在该函数和另一个函数playlist-id的基础上实现find-song-source了，它接受AllegroServe请求对象并返回适当的播放列表标识符。find-song-source函数也负责从请求对象中抓取User-Agent字符串并保存在播放列表对象中。

```
(defmethod find-song-source ((type (eql 'playlist)) request)
  (let ((playlist (lookup-playlist (playlist-id request))))
    (with-playlist-locked (playlist)
      (let ((user-agent (header-slot-value request :user-agent)))
```

29

① 并发编程的复杂度超出了本书的讨论范围。基本的思想是，如果你有多个控制线程，就像在当前的应用里这样，一些线程运行shoutcast函数而另一些线程回应浏览器的请求，那么你需要确保每次只有一个线程在操作给定的某个对象，以避免当一个线程工作在该对象时另一个线程看到了不一致的状态。例如，在当前这个函数中，如果两个新的MP3客户端正在同时连接，它们都试图添加一项到*playlists*中，那么这有可能互相影响。with-process-lock确保了每个线程都可以获得对哈希表的排他访问，以便有足够长的时间来完成它们想做的事。

```
        (when user-agent (setf (user-agent playlist) user-agent))))
    playlist))
```

接下来的难点是如何实现playlist-id，即一个从请求对象中解出标识符的函数。你有很多选项，每个分别对应于不同的用户接口实现。你可以从请求对象中取得任何想要的信息，但无论你决定怎样识别一个客户端，都需要一些方式来让Web接口的用户与正确的播放列表关联在一起。

目前你可以采用一个“勉强可用”的方法，这要求每台连接到服务器的机器上只有一个MP3客户端，并且浏览Web接口的用户就来自运行着MP3客户端的那台机器：你将使用客户机的IP地址作为标识符。通过这种方式，可以为一个请求找出正确的播放列表，而不论请求是来自MP3客户端还是一个Web浏览器。尽管如此，你将在Web接口中提供一种方式来从浏览器中选择一个不同的播放列表，因此这一选择在应用上施加的实际约束是每个客户端IP地址上只能有一个连接的MP3客户端。[①] playlist-id的实现如下所示：

```
(defun playlist-id (request)
  (ipaddr-to-dotted (remote-host (request-socket request))))
```

函数request-socket是AllegroServe的一部分，而remote-host和ipaddr-to-dotted都是Allegro的socket库的一部分。

为了创建可被Shoutcast服务器用作歌曲源的播放列表，需要在current-song、still-current-p和maybe-move-to-next-song上定义将source参数特化在playlist上的方法。current-song方法已经准备好了：通过在current-song槽上定义同名的访问函数，会自动地获得了特化在playlist上的可返回该槽的值的current-song方法。不过，为了使对playlist的访问是线程安全的，需要在访问current-song槽之前锁定该playlist。在本例中，最简单的方法是像下面这样定义一个:around方法：

```
(defmethod current-song :around ((playlist playlist))
  (with-playlist-locked (playlist) (call-next-method)))
```

实现still-current-p也很简单，假设你确保只有在当前的歌曲实际发生了改变时current-song才被更新到新的song对象上。你再次需要获取一个进程锁以确保可以对playlist的状态得到一致的视图。

```
(defmethod still-current-p (song (playlist playlist))
  (with-playlist-locked (playlist)
    (eql song (current-song playlist))))
```

剩下的难点是确保current-song槽在正确的时间得到更新。当前的歌曲有几种改变的方式，最明显的一种是当Shoutcast服务器调用maybe-move-to-next-song时。但它还可以在歌曲被添加到播放列表时更新，比如当Shoutcast服务器播完了所有歌曲，或者甚至是在播放列表的重复模式被改变时。

---

① 这种方法也假设了每个客户机都有独立的IP地址。这个假设在所有用户都在同一个LAN下是成立的，但如果用户是来自一个做网络地址转换的防火墙之后的话就不成立了，如果你想要将这个应用部署在更广的因特网上的话，最好对网络有足够的理解从而找出最适合自己的方法。

与其试图编写特定于上述每种情形的代码来检测是否更新current-song，不如定义一个函数update-current-if-necessary，在current-song中的song对象不再匹配current-idx槽对应的当前应播放文件时，它会更新current-song。然后，如果进行了可能导致这两个槽不同步的播放列表操作之后再调用该函数，你就可以确保current-song总是被正确设置了。下面是update-current-if-necessary和它的助手函数：

```
(defun update-current-if-necessary (playlist)
  (unless (equal (file (current-song playlist))
                 (file-for-current-idx playlist))
    (reset-current-song playlist)))

(defun file-for-current-idx (playlist)
  (if (at-end-p playlist)
    nil
    (column-value (nth-row (current-idx playlist) (songs-table playlist)) :file)))

(defun at-end-p (playlist)
  (>= (current-idx playlist) (table-size (songs-table playlist))))
```

你不需要为这些函数加锁，因为它们将只在那些预先加锁过播放列表的函数中调用。

函数reset-current-song引入了又一个亮点：由于想要播放列表对客户端提供无穷的MP3流，你不希望current-song被设置成NIL。相反，当一个播放列表没有歌曲可播时，即当songs-table为空或是当repeat设置成:none时最后一首歌已经播完了，你需要将current-song设定在一个其文件为MP3静音[①]的特殊歌曲上，并且其标题要能够解释为何没有歌曲在播放。下面的一些代码定义了两个参数，*empty-playlist-song*和*end-of-playlist- song*，每个都被设置成一个以*silence-mp3*所命名的文件作为文件并带有适当标题的歌曲：

```
(defparameter *silence-mp3* ...)

(defun make-silent-song (title &optional (file *silence-mp3*))
  (make-instance
   'song
   :file file
   :title title
   :id3-size (if (id3-p file) (size (read-id3 file)) 0)))

(defparameter *empty-playlist-song* (make-silent-song "Playlist empty."))

(defparameter *end-of-playlist-song* (make-silent-song "At end of playlist."))
```

reset-current-song会在current-idx没有指向songs-table的任何一行时使用这些参数。否则，它会将current-song设置成代表当前行的一个song对象。

① 不幸的是，由于MP3格式的授权问题，我不太清楚在没有向Fraunhofer IIS付费的情况下提供一个这样的MP3文件是否合法。我的这个MP3来自Slim Devices的Slimp3的配套软件的一部分。你可以通过访问http://svn. slimdevices. com/*checkout*/trunk/server/HTML/EN/html/silentpacket.mp3?rev=2。从他们的Subversion库中获得它。或者购买一个Squeezebox，即Slimp3的新的无线版本，然后作为随机软件的一部分，你将得到silentpacket.mp3。或者你还可以查找John Cage的一个长度为4'33"的MP3文件。

```
(defun reset-current-song (playlist)
  (setf
   (current-song playlist)
   (cond
     ((empty-p playlist) *empty-playlist-song*)
     ((at-end-p playlist) *end-of-playlist-song*)
     (t (row->song (nth-row (current-idx playlist) (songs-table playlist)))))))

(defun row->song (song-db-entry)
  (with-column-values (file song artist album id3-size) song-db-entry
    (make-instance
     'song
     :file file
     :title (format nil "~a by ~a from ~a" song artist album)
     :id3-size id3-size)))

(defun empty-p (playlist)
  (zerop (table-size (songs-table playlist))))
```

现在，你可以实现maybe-move-next-song上的方法了，基于播放列表的重复模式将current-idx移到其下一个值上，然后调用update-current-if-necessary。当current-idx已在播放列表的结尾时，你不需要改变current-idx，因为你希望它保持在当前值上，这样它就可以指向你添加到播放列表的下一首歌。这个函数必须在操作播放列表前锁定它，因为它是由Shoutcast服务器代码调用的，而后者并没有做任何锁定。

```
(defmethod maybe-move-to-next-song (song (playlist playlist))
  (with-playlist-locked (playlist)
    (when (still-current-p song playlist)
      (unless (at-end-p playlist)
        (ecase (repeat playlist)
          (:song) ; nothing changes
          (:none (incf (current-idx playlist)))
          (:all  (setf (current-idx playlist)
                       (mod (1+ (current-idx playlist))
                            (table-size (songs-table playlist)))))))
      (update-current-if-necessary playlist))))
```

29.3　操作播放列表

播放列表代码的其余部分被Web接口用来操作playlist对象，包括添加和删除歌曲、排序和乱序以及设置重复模式。和上一节的那些助手函数一样，你不需要在这些函数中担心锁定问题，因为你将要看到，锁将被调用它们的Web接口函数获取。

添加和删除基本上是一个songs-table的管理问题。你唯一需要做的额外工作是保持current-song和current-idx同步。例如，无论何时播放列表为空，它的current-idx都将是零，而current-song将是*empty-playlist-song*。如果你向空的播放列表里添加一首歌曲，那么这个零索引将在范围内，因此你应该将current-song改变成新添加的歌曲。同样的情况，当你已经播放完一个播放列表中的所有歌曲且current-song为*end-of-playlist-song*

时，添加一首歌会导致current-song被重置。所有这些实际上意味着，你需要在适当时机调用update-current-if-necessary。

Web接口沟通所需添加歌曲的方式，使播放列表添加歌曲的过程有点儿复杂。我由于下一节里讨论的一些原因，Web接口代码无法只是给你一些简单的判定规则来从数据库中选择歌曲，而是给你一个列的名字和一个值的列表，然后让你从主数据库中添加给定列具有值列表中某个值的所有歌曲。这样，为了添加正确的歌曲，你需要首先构造一个含有你想要的值的表对象，然后将它和歌曲数据库上的in查询一起使用。因此，add-songs看起来像下面这样：

```
(defun add-songs (playlist column-name values)
  (let ((table (make-instance
                'table
                :schema (extract-schema (list column-name) (schema *mp3s*)))))
    (dolist (v values) (insert-row (list column-name v) table))
    (do-rows (row (select :from *mp3s* :where (in column-name table)))
      (insert-row row (songs-table playlist))))
  (update-current-if-necessary playlist))
```

删除歌曲会简单一些。你只需从songs-table中删除匹配特定条件的歌曲——无论是一个特定歌曲还是一个特定风格、特定艺术家或来自特定专辑的所有歌曲。因此，你可以编写一个delete-song函数，接受一些键值对，用来构造一个你可以传给delete-rows数据库函数的基于matching的:where子句。

当你删除歌曲时会出现的另一个复杂之处是current-idx可能需要改变。假设当前歌曲并非刚刚删除的歌曲之一，那么你希望它仍旧是当前歌曲。但如果在songs-table中在它之前的歌曲被删除，在删除以后它将处在表中的一个不同的位置上。因此在delete-rows调用之后，需要查看含有当前歌曲的行并重设current-idx。如果当前歌曲本身被删除了，在没有其他法子时，你可以将current-idx重设到零。在更新了current-idx之后，调用update-current-if-necessary将会处理current-song的更新。而如果current-idx改变了却仍然指向了同一首歌，那么current-song将保持不变。

```
(defun delete-songs (playlist &rest names-and-values)
  (delete-rows
   :from (songs-table playlist)
   :where (apply #'matching (songs-table playlist) names-and-values))
  (setf (current-idx playlist) (or (position-of-current playlist) 0))
  (update-current-if-necessary playlist))

(defun position-of-current (playlist)
  (let* ((table (songs-table playlist))
         (matcher (matching table :file (file (current-song playlist))))
         (pos 0))
    (do-rows (row table)
      (when (funcall matcher row)
        (return-from position-of-current pos))
      (incf pos))))
```

你还可以编写函数来完全清空播放列表，它使用delete-all-rows并且不再需要查找当前歌

曲，因为当前歌曲明显也要被删除。对update-current-if-necessary的调用将使得current-song设置到empty-playlist-song上。

```
(defun clear-playlist (playlist)
  (delete-all-rows (songs-table playlist))
  (setf (current-idx playlist) 0)
  (update-current-if-necessary playlist))
```

排序和乱序播放列表是彼此相关的操作，因为播放列表总是要么排序的要么乱序的。shuffle槽表明播放列表是否应当被乱序，以及如果是的话该怎样做。如果它被设置为:none，那么播放列表将按照ordering槽的值来排序。当shuffle是:song时，播放列表将被随机地调整顺序。而当它被设置成:album时，专辑的列表将被随机调整顺序，但每个专辑中的歌曲仍然以音轨的顺序列出。这样当用户选择一个新的顺序时，Web接口代码调用的sort-playlist函数，就需要在调用实际完成排序工作的order-playlist之前，将ordering设置成你想要的顺序，而将shuffle设置成:none。和delete-songs里的情况一样，你需要使用position-of-current来重设current-idx到当前歌曲的新位置。不过，这时你不需要调用update-current-if-necessary，因为你知道当前歌曲仍然在表中。

```
(defun sort-playlist (playlist ordering)
  (setf (ordering playlist) ordering)
  (setf (shuffle playlist) :none)
  (order-playlist playlist)
  (setf (current-idx playlist) (position-of-current playlist)))
```

在order-playlist中，你可以使用数据库函数sort-rows来实际进行排序，基于ordering的值传递一个列的列表来进行排序。

```
(defun order-playlist (playlist)
  (apply #'sort-rows (songs-table playlist)
    (case (ordering playlist)
      (:genre  '(:genre :album :track))
      (:artist '(:artist :album :track))
      (:album  '(:album :track))
      (:song   '(:song)))))
```

当用户选择一个新的乱序模式时，Web接口代码调用的函数shuffle-playlist以类似的方式工作，只是它不需要改变ordering的值。这样，当使用:none的shuffle来调用shuffle-playlist时，播放列表将根据最近一次的排序状态进行排序。按歌曲乱序比较简单——只需在songs-table上调用shuffle-table。按专辑乱序稍微复杂一些，但也没什么大不了的。

```
(defun shuffle-playlist (playlist shuffle)
  (setf (shuffle playlist) shuffle)
  (case shuffle
    (:none (order-playlist playlist))
    (:song (shuffle-by-song playlist))
    (:album (shuffle-by-album playlist)))
  (setf (current-idx playlist) (position-of-current playlist)))

(defun shuffle-by-song (playlist)
```

```
  (shuffle-table (songs-table playlist)))

(defun shuffle-by-album (playlist)
  (let ((new-table (make-playlist-table)))
    (do-rows (album-row (shuffled-album-names playlist))
      (do-rows (song (songs-for-album playlist (column-value album-row :album)))
        (insert-row song new-table)))
    (setf (songs-table playlist) new-table)))

(defun shuffled-album-names (playlist)
  (shuffle-table
   (select
    :columns :album
    :from (songs-table playlist)
    :distinct t)))

(defun songs-for-album (playlist album)
  (select
   :from (songs-table playlist)
   :where (matching (songs-table playlist) :album album)
   :order-by :track))
```

你需要支持的最后一个操作是设置播放列表的重复模式。多数时候，在设置repeat时不需要做任何额外的操作，它的值只在maybe-move-to-next-song中用到。不过，你需要在一种情况下作为改变repeat的结果来更新current-song，即当current-idx位于一个非空播放列表的结尾，同时repeat从:song改变成:all。在这种情况下，你希望可以继续播放，要么重复最后一首歌曲，要么从播放列表的起始处开始。因此，你应该在广义函数(setf repeat)上定义一个:after方法。

```
(defmethod (setf repeat) :after (value (playlist playlist))
  (if (and (at-end-p playlist) (not (empty-p playlist)))
    (ecase value
      (:song (setf (current-idx playlist) (1- (table-size (songs-table playlist)))))
      (:none)
      (:all  (setf (current-idx playlist) 0)))
    (update-current-if-necessary playlist)))
```

现在有了你需要的所有底层支持。其余的代码将只是提供一个基于Web的用户接口来浏览MP3数据库和操作播放列表了。这个接口将由3个通过define-url-function定义的主函数构成：一个用于浏览歌曲数据库，一个用于查看和管理单个播放列表，最后一个用来列出所有可用的播放列表。

但在开始编写这三个函数之前，你还需要先写出它们将用到的一些助手函数和HTML宏。

29.4 查询参数类型

由于将使用define-url-function，你需要在第28章的string->type广义函数上定义一些方法，使得define-url-function可以用来将字符串查询参数转化成Lisp对象。在当前的应用下，你将需要一些方法来将字符串分别转化成整数、关键字符号以及一个值的列表。

前两个方法很简单。

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))

(defmethod string->type ((type (eql 'keyword)) value)
  (and (plusp (length value)) (intern (string-upcase value) :keyword)))
```

最后一个string->type方法稍微复杂一些。出于我即将谈到的一些原因，你会需要生成页面来显示表单，其中含有一个隐含字段，其值是一个字符串的列表。由于你要负责生成这个隐含字段中的值，并且在它提交回来以后还要解析它，因此可以使用任何你认为方便的编码方式。你可以使用函数**WRITE-TO-STRING**和**READ-FROM-STRING**，它们使用Lisp的打印器和读取器向字符串中写入数据，以及从字符串中读取数据，除非字符串的打印表示中可能含有引号，和其他在嵌入到一个INPUT元素的值属性时可能带来问题的字符。因此，你需要以某种方式转义这些字符。与其试图引入你自己的转义方法，不如直接使用Base 64，它通常是一种用来保护通过电子邮件发送的二进制数据的编码方式。AllegroServe带有两个函数base64-encode和base64-decode，它们可以为你做Base 64的编码和解码，因此你要做的就只是编写一对函数：一个用来编码Lisp对象，先用**WRITE-TO-STRING**将其转化成可读的字符串，然后再对其进行Base 64编码；另一个用来从上述编码的字符串中进行base 64解码，然后再把结果传给**READ-FROM-STRING**。你需要把对**WRITE-TO-STRING**和**READ-FROM-STRING**的调用包装在**WITH-STANDARD-IO-SYNTAX**中，以确保所有可能影响打印器和读取器的变量都被设置在它们的标准值上。不过，由于你打算读取来自网络的数据，必然希望关掉读取器的一个特性，即不需要在读取过程中对任意Lisp代码求值！[①] 可以定义你自己的宏with-safe-io-syntax，它将其主体Lisp形式包装在一个将 ***READ-EVAL*** 绑定到**NIL**的**LET**外围的**WITH-STANDARD-IO-SYNTAX**里。

```
(defmacro with-safe-io-syntax (&body body)
  `(with-standard-io-syntax
     (let ((*read-eval* nil))
       ,@body)))
```

然后编码和解码函数就很容易写了。

```
(defun obj->base64 (obj)
  (base64-encode (with-safe-io-syntax (write-to-string obj))))

(defun base64->obj (string)
  (ignore-errors
    (with-safe-io-syntax (read-from-string (base64-decode string)))))
```

最终，你可以使用这些函数来定义string->type上的方法，为查询参数类型base64-list定义转换方法。

① 读取器支持一种语法“#.”，它使接下来的S-表达式在读取期被求值。这在源代码中偶尔会有用，但显然会在你读取不可信任的数据时打开了一个巨大的安全漏洞。不过，你可以通过将***READ-EVAL***设置为**NIL**来关闭该语法，这样一来读取器在遇到“#.”时就会报错。

```
(defmethod string->type ((type (eql 'base-64-list)) value)
  (let ((obj (base64->obj value)))
    (if (listp obj) obj nil)))
```

29.5 样板 HTML

接下来需要定义一个HTML宏和助手函数，以便让应用中的不同页面获得一致的外观。可以从一个定义了应用中页面基本结构的HTML宏开始。

```
(define-html-macro :mp3-browser-page ((&key title (header title)) &body body)
  `(:html
     (:head
      (:title ,title)
      (:link :rel "stylesheet" :type "text/css" :href "mp3-browser.css"))
     (:body
      (standard-header)
      (when ,header (html (:h1 :class "title" ,header)))
      ,@body
      (standard-footer))))
```

出于两个理由，应该将standard-header和standard-footer定义成单独的函数。首先，在开发过程中可以重定义这些函数并立即观察其效果，而不需要重新编译那些使用了:mp3-browser-page宏的函数。其次，可以看出你以后编写的某个页面将不会使用:mp3-browser-page，但却可能仍然需要标准的页头和页脚。该宏如下所示：

```
(defparameter *r* 25)

(defun standard-header ()
  (html
   ((:p :class "toolbar")
    "[" (:a :href (link "/browse" :what "genre") "All genres") "] "
    "[" (:a :href (link "/browse" :what "genre" :random *r*) "Random genres") "] "
    "[" (:a :href (link "/browse" :what "artist") "All artists") "] "
    "[" (:a :href (link "/browse" :what "artist" :random *r*) "Random artists") "] "
    "[" (:a :href (link "/browse" :what "album") "All albums") "] "
    "[" (:a :href (link "/browse" :what "album" :random *r*) "Random albums") "] "
    "[" (:a :href (link "/browse" :what "song" :random *r*) "Random songs") "] "
    "[" (:a :href (link "/playlist") "Playlist") "] "
    "[" (:a :href (link "/all-playlists") "All playlists") "]")))

(defun standard-footer ()
  (html
   (:hr)
   ((:p :class "footer") "MP3 Browser v" *major-version* "." *minor-version*)))
```

一些较小的HTML宏和助手函数自动化了其他一些常用的模式。HTML宏:table-row可以让生成HTML中表的单行更加容易。它使用了FOO的一个特性（我将在第31章里提到），即一个&attributes参数，它使任何收集到一个列表中并绑定到&attributes参数上的属性可被宏作为正常S-表达式HTML形式来解析。它看起来像下面这样：

```
(define-html-macro :table-row (&attributes attrs &rest values)
  `(:tr ,@attrs ,@(loop for v in values collect `(:td ,v))))
```

另一个link函数用来生成可用作A元素的HREF属性的应用内部URL，它可从一组键值对中构造出一个查询字符串，并确保所有的特殊字符都被正确地转义。例如，你可以将下面的写法

```
(:a :href "browse?what=artist&genre=Rhythm+%26+Blues" "Artists")
```

替换为

```
(:a :href (link "browse" :what "artist" :genre "Rhythm & Blues") "Artists")
```

该函数如下所示：

```
(defun link (target &rest attributes)
  (html
    (:attribute
     (:format "~a~@[?~{~(~a~)=~a~^&~}~]" target (mapcar #'urlencode attributes)))))
```

为了编码用于URL的键和值，你用到了助手函数urlencode，这是一个包装在函数encode-form-urlencoded上的函数，函数encode-form-urlencoded是来自AllegroServe的一个非公开的函数。一方面，这种做法并不是很好。由于名字encode-form-urlencoded并非是NET.ASERVE导出的名字，encode-form-urlencoded将来有可能消失或在你不知道的情况下被重命名。另一方面，使用这个没有导出的函数可以让你立刻完成手头的工作。通过将encode-form-urlencoede封装在你自己的函数中，就将有风险的代码隔离在了一个函数里，将来如果需要的话还可以重写它。

```
(defun urlencode (string)
  (net.aserve::encode-form-urlencoded string))
```

最后，你需要:mp3-browser-page用到的CSS样式表mp3-browser.css。由于它并非是动态的，最简单的方法就是用publish-file发布一个静态文件。

```
(publish-file :path "/mp3-browser.css"  :file filename :content-type "text/css")
```

一个示例样式表在本书Web站点上与本章配套的源代码放在了一起。你将在本章结尾处定义一个函数来启动这个MP3浏览器应用。它将在完成其他工作的同时顺便发布这个文件。

29.6 浏览页

第一个URL函数将生成一个用来浏览MP3数据库的页面。查询参数将告诉它用户正在浏览什么类型的东西，并提供他们感兴趣的数据库元素的查询条件。它将给你一种方式来查询匹配一个特定风格、艺术家或专辑的数据库项。为了增加奇遇的可能性，你还可以提供一种方式来选择匹配项的一个随机子集。当用户在单个歌曲的层面上浏览时，歌曲的标题是一个添加该歌曲到播放列表的链接。否则，每个项都带有链接，可以让用户浏览其他分类所列出的项。例如，如果用户正在浏览风格，其中的项“Blues”包含的链接可浏览所有带有Blues风格的专辑、艺术家和歌曲。而且，浏览页面里还带有一个“Add all”按钮，可将匹配页面中所给条件的每一首歌曲都添加到该用户的播放列表中。该函数如下所示：

```
(define-url-function browse
    (request (what keyword :genre) genre artist album (random integer))

  (let* ((values (values-for-page what genre artist album random))
         (title (browse-page-title what random genre artist album))
         (single-column (if (eql what :song) :file what))
         (values-string (values->base-64 single-column values)))
    (html
     (:mp3-browser-page
      (:title title)
      ((:form :method "POST" :action "playlist")
       (:input :name "values" :type "hidden" :value values-string)
       (:input :name "what" :type "hidden" :value single-column)
       (:input :name "action" :type "hidden" :value :add-songs)
       (:input :name "submit" :type "submit" :value "Add all"))
      (:ul (do-rows (row values) (list-item-for-page what row)))))))
```

这个函数首先使用函数values-for-page来获得一个含有它需要表示的值的列表。当用户按歌曲来浏览时，这时what参数为:song，你需要从数据库中选择完成的行。但是当用户按风格、艺术家或专辑名来浏览时，你将只想选择给定分类中不同的值。数据库函数select完成了几乎所有的重活儿，其中values-for-page多数时候负责根据what的值来向select传递正确的参数。这也是在必要时你选择匹配行的一个随机子集的地方。

```
(defun values-for-page (what genre artist album random)
  (let ((values
         (select
          :from *mp3s*
          :columns (if (eql what :song) t what)
          :where (matching *mp3s* :genre genre :artist artist :album album)
          :distinct (not (eql what :song))
          :order-by (if (eql what :song) '(:album :track) what))))
    (if random (random-selection values random) values)))
```

为了生成浏览页的标题，可以将浏览条件传递给下列函数browse-page-title：

```
(defun browse-page-title (what random genre artist album)
  (with-output-to-string (s)
    (when random (format s "~:(~r~) Random " random))
    (format s "~:(~a~p~)" what random)
    (when (or genre artist album)
      (when (not (eql what :song)) (princ " with songs" s))
      (when genre  (format s " in genre ~a" genre))
      (when artist (format s " by artist ~a" artist))
      (when album  (format s " on album ~a" album)))))
```

一旦有了想要表示的那些值，就需要对它们做两件事。当然，主要的任务是将它们表示出来，这是由do-rows循环来做的，然后把每行的渲染工作交给list-item-for-page。该函数以一种方式来渲染用于:song的行，而用另一种方式来渲染其他类型的行。

```
(defun list-item-for-page (what row)
  (if (eql what :song)
      (with-column-values (song file album artist genre) row
```

```
        (html
         (:li
          (:a :href (link "playlist" :file file :action "add-songs") (:b song))
          " from "
          (:a :href (link "browse"  :what :song :album  album) album)
          " by "
          (:a :href (link "browse" :what :song :artist artist) artist)
          " in genre "
          (:a :href (link "browse"  :what :song :genre  genre) genre))))
      (let ((value (column-value row what)))
        (html
         (:li value " - "
              (browse-link :genre  what value)
              (browse-link :artist what value)
              (browse-link :album  what value)
              (browse-link :song   what value))))))

(defun browse-link (new-what what value)
  (unless (eql new-what what)
    (html
     "["
     (:a :href (link "browse" :what new-what what value)
         (:format "~(~as~)" new-what))
     "] ")))
```

在browse页上你要做的另一件事是，编写一个带有几个隐含INPUT字段的表单和一个“Add all”提交按钮。你需要使用HTML表单而不是正常的链接来确保应用的无状态性，从而确保拥有所需的信息来回应一个来自该请求本身的请求。由于浏览页中的结果可能是部分随机的，因此你需要向服务器提交相当多的数据才能重构添加到播放列表的歌曲列表。如果你没有允许浏览页返回随机结果的话，就不需要太多的数据，你只需提交一个添加歌曲的请求，采用浏览页使用的任何搜索条件均可。但如果你以这种方式提交一个含有random参数的条件来添加歌曲的话，那么最终你所添加的歌曲将与用户点击“Add all”按钮时在页面中看到的歌曲属于不同的随机集合。

你将使用的解决方案是发回一个含有足够多信息的表单，其中带有一个隐含的INPUT元素，允许服务器可以重构匹配浏览页条件的歌曲列表。该信息就是由values-for-page所返回的值列表以及what参数的值。这就是你用到base64-list参数类型的地方。函数values->base64从values-for-page返回的表中将一个指定列的值解出来并放在一个列表中，然后再从该列表中生成一个base 64编码的字符串嵌入到表单里。

```
(defun values->base-64 (column values-table)
  (flet ((value (r) (column-value r column)))
    (obj->base64 (map-rows #'value values-table))))
```

当该参数以values查询参数的值的形式回到一个将values声明为类型base-64-list的URL函数中时，它将被自动转换回一个列表。后面你将很快看到，该列表可被用来构造返回正确

的歌曲列表的查询。[1]当你正在按:song浏览时，你使用来自:file列的值，因为它们可以唯一地识别实际的歌曲，而通过歌曲名可能不行。

29.7 播放列表

本节将我们带到了下一个URL函数playlist。这是三个页面中最复杂的一个，它负责显示用户播放列表的当前内容，同时提供操作播放列表的接口。但在大部分繁文缛节都由define-url-function处理的背景下，不难看出函数playlist是如何工作的。下面是其定义的开始部分，只给出了参数列表：

```
(define-url-function playlist
    (request
     (playlist-id string (playlist-id request) :package)
     (action keyword)       ; Playlist manipulation action
     (what keyword :file)   ; for :add-songs action
     (values base-64-list) ;              "
     file                   ; for :add-songs and :delete-songs actions
     genre                  ; for :delete-songs action
     artist                 ;              "
     album                  ;              "
     (order-by keyword)     ; for :sort action
     (shuffle keyword)      ; for :shuffle action
     (repeat keyword))      ; for :set-repeat action
```

除了强制出现的request参数以外，playlist还接受大量的查询参数。从某种程度来讲最重要的是playlist-id，它标识了页面应显示和管理的那个playlist对象。对于这个参数，你可以利用define-url-function的“粘滞参数”特性。正常情况下playlist-id无需显式提供，默认为playlist-id函数所返回的值，也就是浏览器所在客户机的IP地址。不过，通过允许显式地指定该值，用户也可以从未运行他们的MP3客户端的其他机器上管理其播放列表。并且如果该参数被指定过一次，那么define-url-function将通过在浏览器中设置一个cookie来使其成为“粘滞的”。随后你将定义一个URL函数来生成全部已有播放列表的列表，其中用户可以选取一个与他们正在机器上浏览的不同的播放列表。

参数action指定了一些在用户的播放列表对象上所做的操作。该参数的值将会自动地转化成一个关键字符号，包括：:add-songs、:delete-songs、:clear、:sort、:shuffle、或:set-repeat。其中:add-songs操作用于浏览页中的“Add all”按钮，也用于那些用来添加单独歌曲的链接。其他的操作都用于播放列表页面本身的链接。

29

① 这个解决方案也有其负面效果——如果一个浏览页返回了许多结果，那么大量的数据将在底层来回发送。另外，数据库查询也未必是最有效的。但它确实可以保证应用的无状态性。一个替代的方法是反过来在服务器端保存由browse返回的结果，然后当一个添加歌曲的请求进来时，查找适当的一点儿信息以重建正确的歌曲集。例如，你可以只是将这个值列表保存下来，而不必将它放在表单里发回服务器。或者你可以在生成浏览结果之前将RANDOM-STATE复制下来，以便后面可以重建出同样的“随机”结果。但这个思路也有它自己的问题。你永远不知道用户何时可能点击了它们浏览器的回退按钮，从而返回到一个旧的浏览页然后再点击那个“Add all”按钮。总之，欢迎来到丰富多彩的Web编程世界。

参数file、what和values与:add-songs操作配合使用。通过将values声明为类型base-64-list，define-url-function底层将负责解码由“Add all”形式提交的值。其余的参数按照注释里所描述的方式分别用于其他操作。

现在让我们来查看playlist的函数体。你需要做的第一件事是使用playlist-id来查找一个队列对象，并使用下面的两行来获取该播放列表的锁：

```
(let ((playlist (lookup-playlist playlist-id)))
  (with-playlist-locked (playlist)
```

由于lookup-playlist将在必要时创建一个新的播放列表，它将总是返回一个playlist对象。然后你进行必要的队列处理，派发action参数的值以便调用一个playlist系列的函数。

```
(case action
  (:add-songs      (add-songs playlist what (or values (list file))))
  (:delete-songs   (delete-songs
                    playlist
                    :file file :genre genre
                    :artist artist :album album))
  (:clear          (clear-playlist playlist))
  (:sort           (sort-playlist playlist order-by))
  (:shuffle        (shuffle-playlist playlist shuffle))
  (:set-repeat     (setf (repeat playlist) repeat)))
```

函数playlist中其余的部分就是实际的HTML生成了。你可以再次使用:mp3-browser-page这个HTML宏来确保页面的基本样式匹配应用程序中的其他页面，尽管这一次你要向:header参数传递NIL以避免生成那个H1头。下面是该函数的其余部分：

```
(html
 (:mp3-browser-page
  (:title (:format "Playlist - ~a" (id playlist)) :header nil)
  (playlist-toolbar playlist)
  (if (empty-p playlist)
    (html (:p (:i "Empty.")))
    (html
      ((:table :class "playlist")
       (:table-row "#" "Song" "Album" "Artist" "Genre")
       (let ((idx 0)
             (current-idx (current-idx playlist)))
         (do-rows (row (songs-table playlist))
           (with-column-values (track file song album artist genre) row
             (let ((row-style (if (= idx current-idx) "now-playing" "normal")))
               (html
                 ((:table-row :class row-style)
                  track
                  (:progn song   (delete-songs-link :file file))
                  (:progn album  (delete-songs-link :album album))
                  (:progn artist (delete-songs-link :artist artist))
                  (:progn genre  (delete-songs-link :genre genre)))))
             (incf idx)))))))))))
```

其中的函数playlist-toolbar生成一个含有到playlist页面的链接的工具栏，以进行多种:action操作。而delete-songs-link可以生成一个带有设置为:delete-songs的:action

参数和其他适当参数的playlist链接，可以分别删除单独的文件、专辑里的所有文件、特定艺术家的文件或是指定风格的文件。

```
(defun playlist-toolbar (playlist)
  (let ((current-repeat (repeat playlist))
        (current-sort (ordering playlist))
        (current-shuffle (shuffle playlist)))
    (html
     (:p :class "playlist-toolbar"
         (:i "Sort by:")
         " [ "
         (sort-playlist-button "genre" current-sort) " | "
         (sort-playlist-button "artist" current-sort) " | "
         (sort-playlist-button "album" current-sort) " | "
         (sort-playlist-button "song" current-sort) " ] "
         (:i "Shuffle by:")
         " [ "
         (playlist-shuffle-button "none" current-shuffle) " | "
         (playlist-shuffle-button "song" current-shuffle) " | "
         (playlist-shuffle-button "album" current-shuffle) " ] "
         (:i "Repeat:")
         " [ "
         (playlist-repeat-button "none" current-repeat) " | "
         (playlist-repeat-button "song" current-repeat) " | "
         (playlist-repeat-button "all" current-repeat) " ] "
         "[ " (:a :href (link "playlist" :action "clear") "Clear") " ] "))))

(defun playlist-button (action argument new-value current-value)
  (let ((label (string-capitalize new-value)))
    (if (string-equal new-value current-value)
      (html (:b label))
      (html (:a :href (link "playlist" :action action argument new-value) label)))))

(defun sort-playlist-button (order-by current-sort)
  (playlist-button :sort :order-by order-by current-sort))

(defun playlist-shuffle-button (shuffle current-shuffle)
  (playlist-button :shuffle :shuffle shuffle current-shuffle))

(defun playlist-repeat-button (repeat current-repeat)
  (playlist-button :set-repeat :repeat repeat current-repeat))

(defun delete-songs-link (what value)
  (html " [" (:a :href (link "playlist" :action :delete-songs what value) "x") "]"))
```

29

29.8 查找播放列表

三个URL函数中的最后一个是最简单的。它可以表示一个列出了当前已创建的所有播放列表的表。通常用户不需要用到这个页面，但在开发过程中它可以给你一个有用的系统状态视图。它还提供了一个选择不同的播放列表的机制，即每个播放列表ID都是一个带有显式playlist-id查询参数的playlist页面的链接，该查询参数随后会被playlist URL函数设置成粘滞的。注意，你需要获取*playlists-lock*以确保*playlists*哈希表在你对其迭代时不会发生改变。

```
(define-url-function all-playlists (request)
  (:mp3-browser-page
   (:title "All Playlists")
   ((:table :class "all-playlists")
    (:table-row "Playlist" "# Songs" "Most recent user agent")
    (with-process-lock (*playlists-lock*)
      (loop for playlist being the hash-values of *playlists* do
           (html
             (:table-row
              (:a :href (link "playlist" :playlist-id (id playlist))
                  (:print (id playlist)))
              (:print (table-size (songs-table playlist)))
              (:print (user-agent playlist)))))))))
```

29.9　运行应用程序

这样就完成了。为了使用这个应用程序，只需使用第27章的load-database函数来加载MP3数据库，发布那个CSS样式表，将*song-source-type*设置成playlist以便find-song-source可以使用播放列表来代替前面章节里定义的单一歌曲源，最后启动AllegroServe。下面的函数为你完成了所有这些步骤，你只需在两个参数中填入适当的值就可以了：*mp3-dir*指定了你的MP3集所在的根目录，而*mp3-css*则是CSS样式表的文件名：

```
(defparameter *mp3-dir* ...)

(defparameter *mp3-css* ...)

(defun start-mp3-browser ()
  (load-database *mp3-dir* *mp3s*)
  (publish-file :path "/mp3-browser.css"  :file *mp3-css* :content-type "text/css")
  (setf *song-source-type* 'playlist)
  (net.aserve::debug-on :notrap)
  (net.aserve:start :port 2001))
```

当你调用这个函数时，它将在从你的ID3文件中加载ID3信息时打印出一些点。然后,你可以将你的MP3客户端指向下面的URL：

```
http://localhost:2001/stream.mp3
```

然后再将你的浏览器指向一个好的起始点，比如：

```
http://localhost:2001/browse
```

这可以让你从默认的风格分类开始浏览。在你为播放列表添加了一些歌曲以后，只要点击MP3客户端的播放按钮，然后它就开始播放第一首歌了。

很明显，你可以从几方面来改进用户接口。假如你的库里有许多MP3，那么通过艺术家或专辑名的第一个字母来浏览就会很有用。或者也许你可以为播放列表页面添加一个“Play whole album”按钮，从而立刻把来自同一张专辑的所有歌曲全部放入播放列表的最顶端，并作为当前播放的歌曲。或者你还可以改变playlist类，当没有歌曲在队列中时不再播放静音，而是从数据库中随机选择一首歌曲来播放。但所有这些思路都属于应用设计的范畴，实际上并不是本书的主题。相反，接下来两章将回到软件基础设施的层面上，探索HTML生成库FOO是如何工作的。