第28章 实践：Shoutcast服务器

本章我们来开发基于Web的流式MP3应用的另一个重要部分，也就是实际向诸如iTunes、XMMS[①]和Winamp等客户端发送MP3流的Shoutcast协议。

28.1 Shoutcast协议

Shoutcast协议是由Nullsoft的人发明的，他们也是Winamp MP3软件的开发者。它被设计用来支持Internet音频广播——Shoutcast DJ从他们的个人电脑上将音频数据发送到一个中央Shoutcast服务器上，继而把它以流的形式发送到任何连网听众那里。

你即将构建的服务器实际上只能算半个真正的Shoutcast服务器，尽管你可以使用通常的Shoutcast服务器一样的协议来传送流式MP3给听众，但你的服务器将只能提供那些已经保存在服务器文件系统中的歌曲。

你只需关注Shoutcast协议的两部分：客户端开始接收一个流时所产生的请求，以及回执的格式，其中包括将当前正在播放歌曲的元数据嵌入到流中的机制。

从MP3客户端到Shoutcast服务器的初始请求被格式化成了一个正常的HTTP请求。在回应部分，Shoutcast服务器发送了ICY回执，它看起来就像是一个HTTP回执，只是字符串“ICY”[②]出现在正常HTTP版本字符串的位置上并带有不同的头。在发送了头和一个空行之后，服务器开始流式发送无穷尽的MP3数据。

关于Shoutcast协议唯一的难点是，正在流式发送的歌曲的元数据如何被嵌入到发送给客户端的数据中。Shoutcast设计者面临的问题是要提供一种方式让Shoutcast服务器可以在每次开始播放一首新歌时与客户端沟通新的标题信息，这样客户端就可以将其显示在UI里。（第25章讲过MP3格式本身并不提供对编码元数据的支持。）尽管ID3v2的一个设计目标是使其更适合用在流式MP3中，但Nullsoft的人们还是决定走他们自己的路线，从头发明了一种在服务器和客户端都相当容

① Red Hat 8.0和9.0以及Fedora中附带的XMMS版本已不能如何播放MP3了，这是因为Red Hat的人对MP3相关的解码器存在版权忧虑。为了在这些版本的Linux上得到带有MP3支持的XMMS，你需要从http://www.xmms.org上获取源代码并自行编译它。或者，对于其他可能性可以参见http://www.fedorafaq.org/#xmms-mp3上的信息。

② 更让人困惑的是，还存在另一个称为Icecast的流协议。看起来在Shoutcast和Icecast协议所使用的ICY头之间不存在明显的关联。

易实现的新格式。这对他们来说也是理想的，因为他们也是自己的MP3客户端的开发者。

他们的方法简单地忽略了MP3数据的结构，在每*n*个字节里嵌入一个自分界的元数据片段。客户端有义务分离出这些元数据，使其不被视为MP3数据。由于发送到不支持该格式的客户端的元数据将导致杂音的出现，服务器仅在客户端的原始请求中包含一个特殊的Icy-Metadata头时才发送元数据。并且为了让客户端知道元数据的发送频率，服务器必须发回一个Icy-Metainit头，其值为在每个相邻的元数据片段之间发送的MP3数据的字节数。

元数据的基本内容是一个形如"StreamTitle='title';"的字符串，其中的title是当前歌曲的标题并且不能带有单引号。这一载荷采用定长的字节数组来编码：先发送一个单字节指示接下来有多少个16字节的块，然后再发送这些块。它们含有作为ASCII字符串的字符串载荷，其中最后一个块使用必要的空字节作为补白。

这样，最小的合法元数据片段是单个字节零，代表没有后续块。如果服务器不需要更新元数据，那么它可以发送这样的一个空片段，但它必须至少发送一个字节才能让客户端不会丢掉实际的MP3数据。

28.2 歌曲源

由于Shoutcast服务器必须在客户端连接上的情况下始终保持流式发送歌曲，你需要为服务器提供一个进行操作的歌曲来源。在基于Web的应用中，每个连接的客户端都将拥有一个可以通过Web接口管理的播放列表。但考虑到为了避免过度耦合，应当定义一个接口让Shoutcast服务器用来获得播放的歌曲。现在你可以编写一个该接口的简单实现，然后在第29章里构建一个更复杂的Web应用接口。

包 定 义

你将在本章里用于开发代码的包如下所示：

```
(defpackage :com.gigamonkeys.shoutcast
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.id3v2)
  (:export :song
           :file
           :title
           :id3-size
           :find-song-source
           :current-song
           :still-current-p
           :maybe-move-to-next-song
           :*song-source-type*))
```

该接口背后的思想是，Shoutcast服务器将根据从AllegroServe请求对象中解出的ID来查找歌曲源。然后它可以对歌曲源做三件事：

- 获得歌曲源中的当前歌曲

- 告诉歌曲源当前歌曲结束
- 询问歌曲源之前给出的某个歌曲是否仍是当前歌曲

最后一个操作是必要的，因为可能存在某种方式（第29章里就会这样做）在Shoutcast服务器之外管理歌曲源。可以用下列广义函数来表达Shoutcast服务器所需的操作：

```
(defgeneric current-song (source)
  (:documentation "Return the currently playing song or NIL."))

(defgeneric maybe-move-to-next-song (song source)
  (:documentation
   "If the given song is still the current one update the value
returned by current-song."))

(defgeneric still-current-p (song source)
  (:documentation
   "Return true if the song given is the same as the current-song."))
```

函数maybe-move-to-next-song如此定义可以允许用单一操作来检查一首歌曲是否为当前歌曲，如果是的话，它就将歌曲源移向下一首歌曲。下一章里当需要实现一个可以安全地从两个不同线程来管理的歌曲源时，这种设计会很重要。[①]

为了表示Shoutcast服务器所需要的关于一首歌曲的信息，你可以定义一个类song。其槽用来保存MP3文件的名字，在Shoutcast元数据中发送的标题，以及使你在发送文件时跳过标签部分的ID3标签的大小。

```
(defclass song ()
  ((file      :reader file      :initarg :file)
   (title     :reader title     :initarg :title)
   (id3-size :reader id3-size :initarg :id3-size)))
```

由current-song（也就是still-current-p和maybe-move-to-next-song的第一个参数）返回的值将是一个song的实例。

此外，你需要定义一个广义函数，以使服务器可以基于想要的歌曲源类型和请求对象来查找一个歌曲源。其方法将特化在type参数上以便返回不同类型的歌曲源，并且从请求对象中将所需的任何信息取出用来检测应返回哪个源。

```
(defgeneric find-song-source (type request)
  (:documentation "Find the song-source of the given type for the given request."))
```

不过，对于本章的目标，可以使用该接口的一个简单实现，让其总是使用相同的对象，即一个可从REPL管理的歌曲对象的简单队列。一开始先定义类simple-song-queue和保存该类的一个实例的全局变量*songs*。

```
(defclass simple-song-queue ()
  ((songs :accessor songs :initform (make-array 10 :adjustable t :fill-pointer 0))
   (index :accessor index :initform 0)))
```

① 从技术上来讲，本章的实现也可以从两个线程来管理，即运行着Shoutcast服务器的AllegroServe线程和REPL线程。但目前你可以接受竞争状况。我将在下一章里讨论如何用锁来确保代码是线程安全的。

```
(defparameter *songs* (make-instance 'simple-song-queue))
```

然后，可以在find-song-source之上定义一个通过符号singleton上的EQL特化符特化在type上的方法，它返回保存在*songs*中的实例。

```
(defmethod find-song-source ((type (eql 'singleton)) request)
  (declare (ignore request))
  *songs*)
```

现在只需实现Shoutcast服务器将会用到的三个广义函数上的方法就可以了。

```
(defmethod current-song ((source simple-song-queue))
  (when (array-in-bounds-p (songs source) (index source))
    (aref (songs source) (index source))))

(defmethod still-current-p (song (source simple-song-queue))
  (eql song (current-song source)))

(defmethod maybe-move-to-next-song (song (source simple-song-queue))
  (when (still-current-p song source)
    (incf (index source))))
```

另外出于测试的目的，可以提供一种方式向队列中添加歌曲。

```
(defun add-file-to-songs (file)
  (vector-push-extend (file->song file) (songs *songs*)))

(defun file->song (file)
  (let ((id3 (read-id3 file)))
    (make-instance
     'song
     :file (namestring (truename file))
     :title (format nil "~a by ~a from ~a" (song id3) (artist id3) (album id3))
     :id3-size (size id3))))
```

28.3 实现 Shoutcast

现在可以开始实现Shoutcast服务器了。由于Shoutcast协议在很大程度上是基于HTTP的，你可以将该服务器实现成AllegroServe中的一个函数。不过，由于你需要与AllegroServe的一些底层特性交互，就不能使用第26章的define-url-function宏。你需要编写一个像下面这样的正规函数：

```
(defun shoutcast (request entity)
  (with-http-response
      (request entity :content-type "audio/MP3" :timeout *timeout-seconds*)
    (prepare-icy-response request *metadata-interval*)
    (let ((wants-metadata-p (header-slot-value request :icy-metadata)))
      (with-http-body (request entity)
        (play-songs
         (request-socket request)
         (find-song-source *song-source-type* request)
         (if wants-metadata-p *metadata-interval*))))))
```

然后像下面这样将该函数发布在路径/stream.mp3下：[1]

```
(publish :path "/stream.mp3" :function 'shoutcast)
```

在with-http-response调用中，除了通常的request和entity参数以外，你还需要传递:content-type和:timeout参数。其中:content-type参数告诉AllegroServe如何设置它所发送的Content-Type头，而:timeout参数指定了AllegroServe给该函数用来生成回执的秒数。在默认情况下AllegroServe判定每个请求在五分钟后超时。由于你打算通过流发送一个本质上无穷的MP3序列，你需要更多的时间。但我们无法告诉AllegroServer一个请求“永不”超时，所以你应当将这个时间设置成*timeout-seconds*的值，其中你可以定义一些诸如10年的秒数这类相当大的值。

```
(defparameter *timeout-seconds* (* 60 60 24 7 52 10))
```

然后，在with-http-response的主体中导致回执头被发送的with-http-body调用之前，你需要处理AllegroServe将要发送的回执。函数prepare-icy-response封装了必要的处理：将协议字符串从默认的“HTTP”改为“ICY”并添加了Shoutcast特定的头。[2]为了处理iTunes中的一个bug，你还需要告诉AllegroServe不要使用分段传输编码（chunked transfer-encoding）。[3]函数request-reply-protocol-string、request-uri和reply-header-slot-value都是AllegroServe的一部分。

```
(defun prepare-icy-response (request metadata-interval)
  (setf (request-reply-protocol-string request) "ICY")
  (loop for (k v) in (reverse
       `((:|icy-metaint| ,(princ-to-string metadata-interval))
         (:|icy-notice1| "<BR>This stream blah blah blah<BR>")
         (:|icy-notice2| "More blah")
         (:|icy-name|    "MyLispShoutcastServer")
         (:|icy-genre|   "Unknown")
         (:|icy-url|     ,(request-uri request))
         (:|icy-pub|     "1")))
     do (setf (reply-header-slot-value request k) v))
  ;; iTunes, despite claiming to speak HTTP/1.1, doesn't understand
  ;; chunked Transfer-encoding. Grrr. So we just turn it off.
  (turn-off-chunked-transfer-encoding request))

(defun turn-off-chunked-transfer-encoding (request)
```

① 当编写这些代码时，你可能还想对Lisp形式(net.aserve::debug-on :notrap)求值。这告诉AllegroServe不要捕捉由你代码所抛出的异常，从而使你在一个正常的Lisp调试器中调试它们。在SLIME中，和其他任何错误情况一样，这将会弹出一个SLIME调试器。

② Shoutcast头通常以小写字母发送，因此你需要转义那些用来在AllegroServe中标识它们的关键字符号的名字，从而避免让Lisp读取器将他们全部转换成大写。这样，你应当写成:|icy-metaint|而不是:icy-metaint。你还可以写成:\i\c\y-\m\e\t\a\i\n\t，但这样非常难看。

③ 函数turn-off-chunked-transfer-encoding使了一点儿诡计。无法在不指定内容长度的情况下通过AllegroServe的官方API来关闭分段传输编码，因为任何声称支持HTTP/1.1的客户端都应当理解它，包括iTunes在内。但这段代码做到了。

```
  (setf (request-reply-strategy request)
        (remove :chunked (request-reply-strategy request))))
```

在函数shoutcast的with-http-body中，实际流出的是MP3数据。函数play-songs接受用来发送数据的流、歌曲源以及应当使用的元数据间隔，或者在客户端不想要元数据时为**NIL**。该流是从请求对象中获取的socket，歌曲源通过调用find-song-source获取到，而元数据间隔则来自全局变量*metadata-interval*。歌曲源的类型由变量*song-source-type*来控制，目前你可以将其设置成singleton以使用你之前实现的simple-song-queue。

```
(defparameter *metadata-interval* (expt 2 12))

(defparameter *song-source-type* 'singleton)
```

函数play-songs本身并不做太多事。它循环调用做所有粗活的函数play-current，包括发送单个MP3文件的内容、跳过ID3标签以及嵌入ICY元数据。唯一的亮点是你需要跟踪何时发送元数据。

由于你必须以特定的间隔来发送元数据片段，而与你何时碰巧从一个MP3文件切换到下一个文件无关，在每次调用play-current时你需要告诉它下一个元数据何时到期，而当它返回时，它必须告诉你相同的事情，这样它才能将该信息传递到下一个play-current调用里。如果play-current从歌曲源里得到了**NIL**，那么它也返回**NIL**，这使play-songs 中的循环得以停下来。

除了处理循环以外，play-songs还提供了一个**HANDLER-CASE**，当MP3客户端从服务器上断开并导致对socket的写入失败时，用它捕捉在play-current中抛出的错误。由于**HANDLER-CASE**在**LOOP**之外，对错误进行处理将中断循环，从而允许play-songs返回。

```
(defun play-songs (stream song-source metadata-interval)
  (handler-case
      (loop
         for next-metadata = metadata-interval
         then (play-current
               stream
               song-source
               next-metadata
               metadata-interval)
         while next-metadata)
    (error (e) (format *trace-output* "Caught error in play-songs: ~a" e))))
```

最终，你可以实现play-current了，它用来实际发送Shoutcast数据。基本思想是，你从歌曲源里得到当前的歌曲，打开该歌曲的文件，然后循环地从文件中读取数据并写入到socket中，直到要么遇到文件结尾，要么当前歌曲不再是当前歌曲了。

这里只有两个复杂之处：一个是需要确保在正确的间隔上发送元数据；另一个是如果文件以ID3标签开始，那么你需要跳过它。如果你不过多地考虑I/O性能，那么可以像下面这样来实现play-current：

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
```

```
    (when song
      (let ((metadata (make-icy-metadata (title song))))
        (with-open-file (mp3 (file song):element-type '(unsigned-byte 8))
          (unless (file-position mp3 (id3-size song))
            (error "Can't skip to position ~d in ~a" (id3-size song) (file song)))
          (loop for byte = (read-byte mp3 nil nil)
             while (and byte (still-current-p song song-source)) do
               (write-byte byte out)
               (decf next-metadata)
             when (and (zerop next-metadata) metadata-interval) do
               (write-sequence metadata out)
               (setf next-metadata metadata-interval))

          (maybe-move-to-next-song song song-source)))
      next-metadata)))
```

该函数从歌曲源中得到当前歌曲，并得到一个缓冲区，含有将要通过传递标题给make-icy-metadata来发送的元数据。接着它打开文件并使用两参数形式的**FILE-POSITION**跳过ID3标签。然后它开始从文件中读取字节并将它们写到请求的流中。[①]

当到达文件的结尾或是当歌曲源的当前歌曲发生改变时，循环就会中断。同时，无论何时next-metadata得到了零（如果允许发送元数据），那么它就将metadata写入到流中并重置next-metadata。一旦它完成了循环，就会检查歌曲是否仍是歌曲源的当前歌曲。如果是的话，那么这意味着它是因为读取了整个文件才跳出循环的，这时它会告诉歌曲源移动到下一首歌上；否则，它跳出循环是因为有人改变了当前正在播放的歌曲，那么函数就只是返回。无论是哪种情况，它都返回在下一个元数据到期前剩余的字节数，以便传给play-current的下一次调用。[②]

函数make-icy-metadata接受当前歌曲的标题，并生成一个含有正确格式化的ICY元数据片段的字节数组，它的实现也是相当直接的。[③]

```
(defun make-icy-metadata (title)
  (let* ((text (format nil "StreamTitle='~a';" (substitute #\Space #\' title)))
         (blocks (ceiling (length text) 16))
         (buffer (make-array (1+ (* blocks 16))
                             :element-type '(unsigned-byte 8)
                             :initial-element 0)))
    (setf (aref buffer 0) blocks)
    (loop
```

① 多数MP3播放软件都会在用户接口的某个地方显示元数据。不过，Linux上的XMMS程序默认不这样做，为了让XMMS显示Shoutcast元数据，需要按Ctrl+P来打开Preferences面板；接着在Audio I/O Plugins标签栏（在版本1.2.10下是最左边的标签）选择MPEG Layer 1/2/3 Player（libmpg123.so）并按下Configure按钮；然后选择配置窗口中的“流”标签，并在标签底部的SHOUTCAST/Icecast部分里选中“Enable SHOUTCAST/Icecast title streaming”复选框。

② 那些从Scheme迁移到Common Lisp的人们可能想知道为什么play-current不是递归地调用其自身。在Scheme中这确实工作得很好，因为Scheme实现在规范要求下必须支持“无限次活跃尾递归”（unbounded number of active tail calls）。Common Lisp实现也允许带有这一属性，但这不是语言标准要求的。因此，Common Lisp习惯上使用循环构造来编写循环，而不是递归。

③ 和你编写的其他代码一样，这个函数假设你的Lisp实现的内部字符编码方式是ASCII或ASCII的一个超集，因此你可以使用**CHAR-CODE**将Lisp的**CHARACTER**对象转化成ASCII数据的字节。

```
    for char across text
    for i from 1
    do (setf (aref buffer i) (char-code char)))
  buffer))
```

根据你的具体Lisp实现是如何处理它的流的，以及需要一次服务多少个MP3客户端，这个简单版本的play-current可能足够高效，也可能不是。

这个简单实现的潜在问题是，你不得不为你传输的每个字节调用**READ-BYTE**和**WRITE-BYTE**。有可能每个调用都产生成本相对高昂的系统调用来读写一个字节。即便Lisp实现了自己的带有内部缓冲区的流，从而并非每个**READ-BYTE**和**WRITE-BYTE**都产生系统调用，函数调用本身的成本也仍然存在。特别是在使用所谓的Gray Streams提供用户可扩展流的实现里，**READ-BYTE**和**WRITE-BYTE**可能导致对广义函数的调用，该函数在底层派发到流参数的类上。虽说广义函数派发通常足够高效让你不必担心，但它还是比非广义的函数调用成本更高一些，如果能够避免的话，你绝不想在几分钟里对其调用数百万次。

实现play-current的一种更高效但也更复杂的方式是使用**READ-SEQUENCE**和**WRITE-SEQUENCE**来一次性读写多个字节。你也给自己一个机会将文件读取操作与文件系统的自然块大小相匹配，从而为你带来最佳的磁盘吞吐量。当然，无论你使用多大的缓冲区，跟踪何时发送元数据都将变得更复杂一些。一个使用了**READ-SEQUENCE**和**WRITE-SEQUENCE**的更高效的play-current版本如下所示：

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song)))
            (buffer (make-array *block-size*:element-type '(unsigned-byte 8))))
        (with-open-file (mp3 (file song)):element-type '(unsigned-byte 8))
          (labels ((write-buffer (start end)
                     (if metadata-interval
                       (write-buffer-with-metadata start end)
                       (write-sequence buffer out :start start :end end)))

                   (write-buffer-with-metadata (start end)
                     (cond
                       ((> next-metadata (- end start))
                        (write-sequence buffer out :start start :end end)
                        (decf next-metadata (- end start)))
                       (t
                        (let ((middle (+ start next-metadata)))
                          (write-sequence buffer out :start start :end middle)
                          (write-sequence metadata out)
                          (setf next-metadata metadata-interval)
                          (write-buffer-with-metadata middle end))))))

            (multiple-value-bind (skip-blocks skip-bytes)
                (floor (id3-size song) (length buffer))

              (unless (file-position mp3 (* skip-blocks (length buffer)))
                (error "Couldn't skip over ~d ~d byte blocks."
```

```
                    skip-blocks (length buffer)))

            (loop for end = (read-sequence buffer mp3)
               for start = skip-bytes then 0
               do (write-buffer start end)
               while (and (= end (length buffer))
                          (still-current-p song song-source)))

            (maybe-move-to-next-song song song-source)))))
      next-metadata)))
```

现在你可以用所有这些东西来做点什么了。在下一章里，你将编写一个本章所开发的Shoutcast服务器的Web接口，它使用第27章的MP3数据库作为歌曲源。