第26章

实践：用AllegroServe进行Web编程

本章将利用开源的AllegroServe Web服务器来学习在Common Lisp中开发Web应用的一种方法。这并不意味着本章是对AllegroServe的完整介绍。我只打算介绍关于Web编程这个大型话题的冰山一角。我的目标是涵盖足够多的AllegroServe基本用法，以确保你可以在第29章里用它来开发一个可以浏览MP3文件库，并将它们以流的方式发送到MP3客户端的应用程序。类似地，本章也为初学者提供了一个关于Web编程的简要介绍。

26.1 30秒介绍服务器端Web编程

尽管当今的Web程序开发通常都会用到相当数量的软件框架和不同的协议，但Web编程的核心部分自从它们在20世纪90年代早期被发明以后几乎没有什么变化。对于第29章里将要编写的那些简单应用，你只需理解几个关键的概念就可以了，因此这里将快速地概述一下。有经验的Web程序员可以略读或是干脆跳过本节。[1]

首先，你需要理解Web浏览器和Web服务器在Web编程中的角色。尽管现代浏览器通常带有大量花哨的功能，但Web浏览器的核心功能只是从Web服务器上请求Web页并将它们渲染出来。通常，这些页面是使用超文本标记语言（HTML）来编写的，HTML告诉浏览器如何渲染页面，包括在哪里插入内嵌的图像和指向其他Web页的链接。HTML由带有标签的文本组成，这些标签为文本添加了结构，使浏览器得以渲染页面。一个简单的HTML文档如下所示：

```
<html>
  <head>
  <title>Hello</title>
  </head>
  <body>
  <p>Hello, world!</p>
  <p>This is a picture: <img src="some-image.gif"></p>
```

① Web编程的初学者需要在这篇介绍的基础上补充阅读一两篇更加深入的介绍。你可以在http://www. jmarshall. com/easy/上找到一些很好的在线指导。

```
  <p>This is a <a href="another-page.html">link</a> to another page.</p>
  </body>
</html>
```

图26-1　显示了浏览器是如何渲染这个页面的。

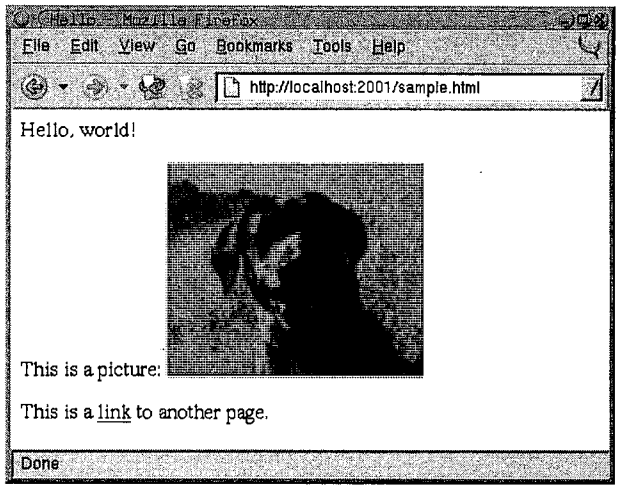

图26-1　网页示例

浏览器和服务器之间使用一种称为超文本传输协议（HTTP）的协议进行通信。尽管你不需要关心该协议的细节，但有必要知道它是由浏览器发起的请求和服务器生成的响应的规范。这就是说，浏览器连接到Web服务器并发送了一个至少包括目标URL和浏览器支持的HTTP版本的请求。浏览器也可以在它的请求中包含数据。这就是浏览器向服务器提交HTML表单的方式。

为了回应一个请求，服务器发送一个包括一系列的头部和一个主体的响应。头部中含有关于主体的信息，诸如数据的类型是什么（比如HTML、纯文本或一个图片），而主体就是数据本身，随后将被浏览器渲染。服务器有时也会发送一个错误响应来告诉浏览器其请求因为某种原因无法正确回应。

情况基本上就是这样。一旦浏览器从服务器那里收到了完整的响应，那么直到下一次浏览器决定从服务器请求一个页面之前，在浏览器和服务器之间将不再有任何通信。[①]这就是Web编程的主要约束所在——无法让运行在服务器上的代码影响用户在浏览器中看到的内容，除非浏览器向服务器发起一个新请求。[②]

有些称为静态页面的Web页只是保存在Web服务器上的HTML文件，在浏览器发出请求时直接被发送出去。另一方面，动态页面是由每次页面被浏览器请求时生成的HTML构成的。一个动

① 加载单个Web页面可能实际上会产生多个请求。为了渲染一个含有内嵌图片的页面的HTML，浏览器必须单独地请求每个图片，再将它们分别插入到渲染后的HTML中的适当位置。

② Web编程的许多复杂性都是试图解决这个基本限制的结果，目标是提供类似桌面应用那样的用户体验。

态页面可能会在查询数据库的基础上生成并构造出HTML来表示查询的结果。[①]

当针对请求生成响应时，服务器端的代码需要处理四种主要信息。第一种信息是被请求的URL。不过，URL通常被Web服务器本身用来决定使用哪些代码来生成响应。接下来，如果URL中含有一个问号，那么问号之后的所有内容将被视为一个查询字符串，后者通常会被Web服务器忽略，除非将它传给用来生成响应的代码。多数时候查询字符串由一组键/值对组成。来自浏览器的请求也可以POST数据，这些数据通常也由键值对构成。POST数据一般用来提交HTML表单。无论是查询字符串中的键值对还是发送数据中的键值对都被统称为查询参数。

最后，为了将来自同一个浏览器的一系列请求串接在一起，服务器中运行的代码可以设置cookie，并在浏览器的响应中发送一个特殊的头部，里面含有一些不透明数据。一旦cookie被一个特定的服务器所设置，那么浏览器将在每次向该服务器发送请求时都带上这个cookie。浏览器并不关心cookie中的数据，它只是将其回显给服务器，让服务器端的代码按照它们想要的方式来解释。

以上就是99%的服务器端Web编程所依赖的基础元素。浏览器发起一个请求，服务器查找用来处理该请求的代码并运行它，然后代码使用查询参数和cookie来决定要做什么。

26.2 AllegroServe

有很多种方式可以用Common Lisp来提供Web内容。至少有三种用Common Lisp写的开源Web服务器，还有诸如mod_lisp[②]和Lisplets[③]这类可以允许Apache Web服务器或任何Java Servlet容器将请求代理到运行在独立进程中的Lisp服务器上的系统。

对于本章来说，你将使用开源Web服务器AllegroServe的某个版本，它最初由Franz Inc.的John Foderaro所开发。AllegroServe包含在来自Franz的用于本书的Allegro版本里。如果你没在使用Allegro，那么你可以使用PortableAllegroServe，即一个AllegroServe代码树的友好分支，它还包括一个让PortableAllegroServe得以运行在多数Common Lisp平台上的兼容层。本章和第29章里编写的代码应该可以同时运行在原版的AllegroServe和PortableAllegroServe上。

AllegroServe提供了一个与Java Servlet类似的编程模型。每当浏览器请求一个页面时，AllegroServe会解析请求并查找一个称为实体(entity)的对象来处理该请求。一些作为AllegroServe一部分的实体类知道如何处理静态内容——无论是单独的文件还是一个目录树的内容。而另一些

① 不幸的是，“动态”一词在Web世界中被重载了。术语“动态HTML”指的是含有嵌入式代码的HTML，其代码通常采用JavaScript来编写，JavaScript可在不跟Web服务器进行通信的情况下在浏览器中执行。如果谨慎使用，动态HTML可以改进一个基于Web的应用程序的可用性，因为即便在高速的Internet连接下，向一个Web服务器发出请求、接受响应并渲染新页面也需要花费相当长的时间。更加令人困惑的是，动态生成的页面（换句话说，是在服务器上生成的页面）也可以含有动态HTML（运行在客户端的代码）。对于本书中的应用，你将只是动态地生成简单的非动态HTML。

② http://www.fractalconcept.com/asp/html/mod_lisp.html。

③ http://lisplets.sourceforge.net/。

则是我将用本章的多数篇幅进行讨论的，它们运行任意Lisp代码来生成响应。[1]

但在开始之前，你需要知道如何启动AllegroServe并让它提供一些文件。第一步是将AllegroServe加载到你的Lisp映像中。在Allegro中，可以简单地键入(require :aserve)。在其他Lisp环境（也包括Allegro在内）下，可以通过加载portableaserve目录树顶层的文件INSTALL.lisp来加载PortableAllegroServe。加载AllegroServe会创建三个新包：NET.ASERVE、NET.HTML.GENERATOR和NET.ASERVE.CLIENT。[2]

加载了服务器以后，你可以通过NET.ASERVE包中的函数start来启动它。为了可以方便地访问来自NET.ASERVE、COM.GIGAMONKEYS.HTML（一个即将讨论到的新包）以及Common Lisp其余部分的导出符号，你应该像下面这样创建一个新包：

```
CL-USER> (defpackage :com.gigamonkeys.web
           (:use :cl :net.aserve :com.gigamonkeys.html))
#<The COM.GIGAMONKEYS.WEB package>
```

现在使用下面的IN-PACKAGE表达式切换到该包上：

```
CL-USER> (in-package :com.gigamonkeys.web)
#<The COM.GIGAMONKEYS.WEB package>
WEB>
```

现在你可以无需限定符而使用来自NET.ASERVE的导出符号了。函数start用来启动服务器，它接受相当数量的关键字参数，但你现在唯一需要传递的是:port，即监听的端口。你可能需要使用诸如2001这种较大的端口而不是HTTP服务器的标准端口80。因为在类Unix的操作系统里，只有root用户才能监听1024以下的端口。为了在Unix上运行监听80端口的AllegroServe，你需要以root用户启动Lisp，然后使用:setuid和:setgid参数来告诉start在打开端口以后切换到指定的身份。可以像下面这样启动监听端口2001的服务器：

```
WEB> (start :port 2001)
#<WSERVER port 2001 @ #x72511c72>
```

服务器现在在你的Lisp环境中运行了。在启动服务器时或许会看到类似“port already in use”这样的错误提示。这表明端口2001已经被系统里的其他服务器占用了。在这种情况下，最简单的修复方法是使用一个不同的端口，为start提供一个不同的参数，然后在本章其余部分的URL里始终用该值来代替2001。

你可以继续通过REPL与Lisp环境交互，因为AllegroServe启动了它自己的线程来处理来自浏览器的请求。这意味着你至少可以通过REPL来观察当前运行中的服务器，这使得调试和测试工作比面对一个完全黑箱的服务器要容易得多。

假设你正在运行的Lisp环境与你的浏览器是在同一台机器上，那么你可以通过将浏览器指向http://localhost:2001/来检查服务器是否已经启动并运行了。此刻你应该会在浏览器中得到一个页

① AllegroServe也提供了一个名叫Webactions的框架，其类似于Java中的JSP。这个框架不是编写代码来生成HTML，通过Webactions你可以直接编写本质上是HTML的页面，但其中的某些内容将在页面提供服务时作为代码来运行。在本书里我将不会谈及Webactions。

② 加载PortableAllegroServe将为相关的兼容库创建出其他一些包，但你需要关心的只是那三个包。

面未找到的错误信息，因为你还没有发布任何内容。但这个错误信息来自AllegroServe，你可以从页面的底部看到这点。另一方面，如果浏览器显示了一个错误对话框并提示说“The connection was refused when attempting to contact localhost:2001”，那么这意味着要么服务器没有运行，要么是从不同于2001的端口启动的。

现在你可以发布一些文件了。假设在/tmp/html目录下有一个文件hello.html，其内容如下：

```
<html>
  <head>
  <title>Hello</title>
  </head>
  <body>
  <p>Hello, world!</p>
  </body>
</html>
```

你可以使用publish-file函数单独地发布它。

```
WEB> (publish-file :path "/hello.html" :file "/tmp/html/hello.html")
#<NET.ASERVE::FILE-ENTITY @ #x725eddea>
```

其中的:path参数将出现在浏览器请求的URL中，而:file参数则是文件系统中的文件名。在求值publish-file表达式之后，可以将浏览器指向http://localhost:2001/hello.html，然后它将显示一个类似图26-2这样的页面。

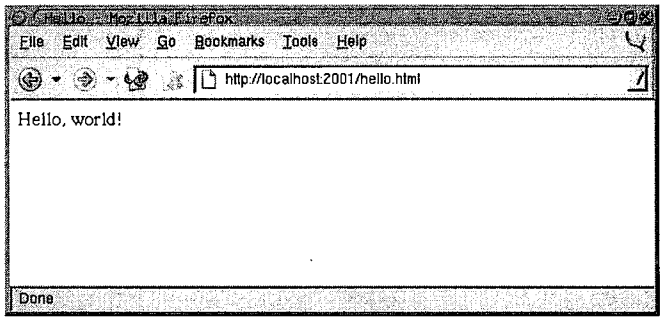

图26-2 http://localhost:2001/hello.html

你也可以使用publish-directory函数来发布整个目录树中的文件。首先让我们使用下面的publish-file调用将已发布的内容清除：

```
WEB> (publish-file :path "/hello.html" :remove t)
NIL
```

现在，你可以使用publish-directory函数发布整个/tmp/html/目录（包括它的所有子目录）了。

```
WEB> (publish-directory :prefix "/" :destination "/tmp/html/")
#<NET.ASERVE::DIRECTORY-ENTITY @ #x72625aa2>
```

在本例中，:prefix参数指定了应由该实体接手的URL路径部分的开始。这样，如果服务器收到了一个来自http://localhost:2001/foo/bar.html的请求，那么其路径部分是/foo/bar.html，以“/”开始。这个路径随后通过将其前缀“/”替换成目标“/tmp/html/”从而变成了一个文件名。同样的道理，http://localhost:2001/hello.html也将被转化成一个对文件/tmp/html/hello.html的请求。

26.3　用AllegroServe生成动态内容

发布生成动态内容的实体几乎和发布静态内容一样简单。函数publish和publish-prefix是publish-file和publish-directory对应的动态版本。这两个函数的基本思想是，你可以发布一个函数，它将被调用来生成一个指定URL或带有给定前缀的任何URL的响应。这个函数将用两个参数来调用：一个代表请求的对象以及一个被发布的实体。多数时候你不需要对那个实体对象做任何操作，除非将它和一些后面即将讨论到的宏一起传递。另一方面，你将使用请求对象来获取由浏览器提交的信息，即包含在URL或使用HTML表单发送的数据中的查询参数。

作为使用函数来生成动态内容的简单示例，让我们编写一个在每次请求时生成一个带有不同随机数的页面的函数。

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (format
       (request-reply-stream request)
       "<html>~@
        <head><title>Random</title></head>~@
        <body>~@
        <p>Random number: ~d</p>~@
        </body>~@
        </html>~@
       "
       (random 1000)))))
```

宏with-http-response和with-http-body是AllegroServe的一部分。前者启动生成一个HTTP响应的过程，并且可以像这样指定诸如返回内容的类型之类的信息。它还可以处理HTTP的其他部分，例如处理If-Modified-Since请求。with-http-body实际发送HTTP回执头并执行其主体，后者应当含有用来生成响应内容的代码。在with-http-response中with-http-body之前的地方，你可以添加或修改在回执中发送的HTTP头。函数request-reply-stream也是AllegroServe的一部分，它返回一个流用来向浏览器中写入想要的输出。

正如该函数显示的，可以只用**FORMAT**将HTML打印到由request-reply-stream返回的流上。在下一节里，我将向你展示更方便的方法来以编程方式生成HTML。[①]

① ~@后接一个新行可以告诉**FORMAT**忽略换行之后的所有空白，这样就可以精美地缩进代码而不会在HTML中增加大量的空白。由于HTML中的空白通常会被忽略，因此这不会影响到浏览器，但它可以让产生的HTML看起来更美观。

现在你可以发布这个函数了。

```
WEB> (publish :path "/random-number" :function 'random-number)
#<COMPUTED-ENTITY @ #x7262bab2>
```

参数:path与它在publish-file函数中的用法相同，它指定导致该函数被调用的URL的路径部分。:function参数用来指定函数的名字或实际的函数对象。像这样使用一个函数的名字可以允许你以后重定义该函数而无需重新发布即可令AllegroServe使用新的函数定义。在求值了publish调用以后，可以让浏览器指向http://localhost:2001/random-number来得到一个带有一个随机数的页面，如图26-3所示。

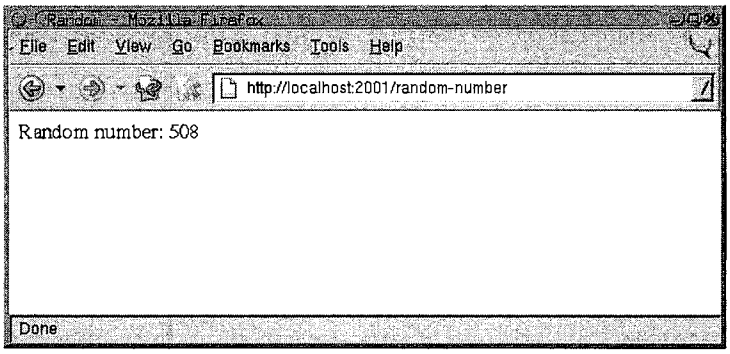

图26-3 http://localhost:2001/random-number

26.4 生成 HTML

尽管使用**FORMAT**来生成HTML对于到目前为止所讨论的简单页面还不错，但如果要构建更复杂的页面，那么如果有一种更简洁的HTML生成方式就更好了。有几个库可以用来从S-表达式形式的表示生成HTML，其中的htmlgen就包含在AllegroServe中。在本章里你将使用一个称为FOO[①]的库，它在很大程度上来自Franz的htmlgen，并且你将在第30章和第31章里看到更多关于它的具体实现的细节。不过目前你只需要知道如何使用FOO。

从Lisp里生成HTML是件相当自然的事情，因为本质上S-表达式跟HTML是同构的。你可以用S-表达式来表示HTML元素，方法是将HTML中的每个元素视为一个以适当头元素“标记”的列表，例如一个与HTML标签同名的关键字符号。这样，HTML foo 就可以用S-表达式(:p "foo")来表示了。由于HTML元素嵌套的方式与S-表达式中的列表嵌套方式相同，因此上述表示法可以扩展到更复杂的HTML。例如，下面的HTML

```
<html>
  <head>
  <title>Hello</title>
  </head>
```

① FOO是来源于FOO Outputs Output的递归伪技术缩略语。

```
  <body>
  <p>Hello, world!</p>
  </body>
</html>
```

可以用下列S-表达式来表示：

```
(:html
  (:head (:title "Hello"))
  (:body (:p "Hello, world!")))
```

带有属性的HTML元素会稍微复杂一些，但也不是无法解决。FOO支持两种在标签中添加属性的方式。一种方式是简单地在列表的第一个元素之后跟上一个键值对。跟在键值对后面的第一个不是关键字符号的元素代表该HTML元素内容的开始。这样，你可以将下面的HTML

```
<a href="foo.html">This is a link</a>
```

用下列S-表达式来表示：

```
(:a :href "foo.html" "This is a link")
```

FOO支持的另一种语法是将标签名和属性组织在它们自己的列表中，如下所示：

```
((:a :href "foo.html") "This is link.")
```

FOO可以通过这两种方式使用S-表达式来表示HTML。函数emit-html接受一个HTML的S-表达式并输出相应的HTML。

```
WEB> (emit-html '(:html (:head (:title "Hello")) (:body (:p "Hello, world!"))))
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
T
```

不过，emit-html并非总是最有效的HTML生成方式，因为其参数必须是想要生成的HTML的完整S-表达式表示。尽管构造这样一个表示很容易，但它却并不总是高效的。例如，假设你想要生成一个含有10 000个随机数的列表的HTML页面。你可以像下面这样使用一个反引用模板来构造S-表达式并将其传给emit-html：

```
(emit-html
  `(:html
     (:head
       (:title "Random numbers"))
     (:body
       (:h1 "Random numbers")
       (:p ,@(loop repeat 10000 collect (random 1000) collect " ")))))
```

不过，这会导致在实际开始生成HTML之前就先要构造出一个含有10 000个元素的列表的树来，而一旦HTML生成出来以后，整个S-表达式就没有任何用处了。为了避免这种低效，FOO还

支持一个宏html，它允许你在一个HTML的S-表达式中嵌入一点儿Lisp代码。

位于html宏的输入中的诸如字符串和数字这样的字面值将被插入到输出的HTML中。同样，符号将被视为对变量的引用，宏所生成的代码会在运行期输出它们的值。这样，下面两个形式

```
(html (:p "foo"))

(let ((x "foo")) (html (:p x)))
```

都将生成下面的代码：

```
<p>foo</p>
```

不以一个关键字符号开始的列表形式会被视为代码，并被嵌入到生成的代码中。被嵌入的代码返回的任何值都将被忽略，但是代码可以通过调用html宏本身来产生更多的HTML。例如，为了在HTML中输出一个列表的内容，你可以写成下面这样

```
(html (:ul (dolist (item (list 1 2 3)) (html (:li item)))))
```

它将产生下面的HTML：

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

如果你想输出一个列表形式的值，必须将其包装在伪标签:print中。这样，表达式

```
(html (:p (+ 1 2)))
```

在计算并丢弃值3以后会生成下面的HTML

```
<p></p>
```

为了输出那个3，必须写成下面这样：

```
(html (:p (:print (+ 1 2))))
```

或者也可以先计算出该值并将其保存在一个html调用之外的变量里，如下所示：

```
(let ((x (+ 1 2))) (html (:p x)))
```

这样，你就可以使用html宏来生成随机数的列表了，如下所示：

```
(html
  (:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p (loop repeat 10 do (html (:print (random 1000)) " "))))))
```

宏版本将比emit-html版本更加高效。你不仅不再需要生成一个代表整个页面的S-表达式，而且emit-html的很多在运行期解释S-表达式的工作现在都可以在宏展开时一次性完成，而不必在每次代码运行时来做了。

你可以通过宏with-html-output来控制由html和emit-html生成的输出被发送到哪里，该宏是FOO库的一部分。这样，你可以使用来自FOO的with-html-output和html宏来重写random-number，如下所示：

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random 1000)))))))))))
```

26.5 HTML 宏

FOO还有一个特性，它允许你定义一种HTML“宏”可将任意形式转化成html宏可理解的HTML S-表达式。例如，假设你经常发现自己会编写下列形式的页面：

```
(:html
  (:head (:title "Some title"))
  (:body
    (:h1 "Some title")
    ... stuff ...))
```

那么你应该定义一个HTML宏来捕捉这个模式，就像这样：

```
(define-html-macro :standard-page ((&key title) &body body)
  `(:html
     (:head (:title ,title))
     (:body
      (:h1 ,title)
      ,@body)))
```

现在，你可以在你的S-表达式HTML中使用“标签”:standard-page了，它将在被解释或编译之前展开。例如，下面的形式

```
(html (:standard-page (:title "Hello") (:p "Hello, world.")))
```

可以生成这样的HTML：

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Hello, world.</p>
  </body>
</html>
```

26.6 查询参数

当然，生成HTML输出还只是Web编程的一半，另一半是得到来自用户的输入。正如26.1节中讨论过的，当浏览器从Web服务器上请求一个页面时，它可以在URL和POST数据中发送查询参数，两者都是向服务器端代码提供输入的途径。

和多数Web编程框架一样，AllegroServe可以帮你解析这两种输入。等到你发布的函数被调用时，所有来自查询字符串和/或POST数据的键/值对都已被解码并放置在一个alist中，后者可以使用函数requiest-query从请求对象中获取alist。下面的函数可以返回一个页面，其中显示了所有它收到的查询参数：

```
(defun show-query-params (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
           (:title "Query Parameters")
           (if (request-query request)
             (html
               (:table :border 1
                       (loop for (k . v) in (request-query request)
                          do (html (:tr (:td k) (:td v))))))
             (html (:p "No query parameters."))))))))))

(publish :path "/show-query-params" :function 'show-query-params)
```

如果你给浏览器一个类似下面这样的带有查询字符串的URL

```
http://localhost:2001/show-query-params?foo=bar&baz=10
```

那么你应该可以得到一个类似图26-4所示的页面。

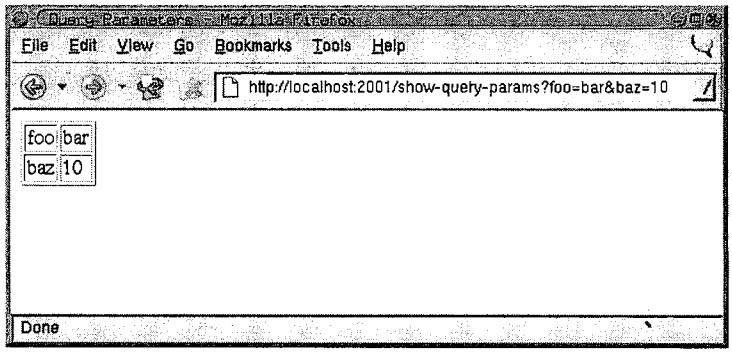

图26-4 http://localhost:2001/show-query-params?foo=bar&baz=10

为了生成POST数据，你需要一个HTML表单。下面的函数可以生成一个简单的表单，其数据将发送到show-query-params：

```
(defun simple-form (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let ((*html-output* (request-reply-stream request)))
        (html
          (:html
            (:head (:title "Simple Form"))
            (:body
             (:form :method "POST" :action "/show-query-params"
               (:table
                (:tr (:td "Foo")
                     (:td (:input :name "foo" :size 20)))
                (:tr (:td "Password")
                     (:td (:input :name "password" :type "password" :size 20))))
               (:p (:input :name "submit" :type "submit" :value "Okay")
                   (:input ::type "reset" :value "Reset")))))))))))

(publish :path "/simple-form" :function 'simple-form)
```

将你的浏览器指向http://localhost:2001/simple-form，然后你应该可以看到一个类似图26-5所示的页面。

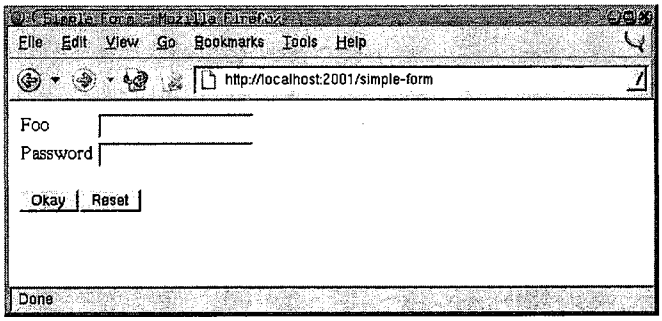

图26-5　http://localhost:2001/simple-form

如果你在表单中填入“abc”和“def”两个值，那么点击Okay按钮应该会把你带到一个类似图26-6所示的页面里。

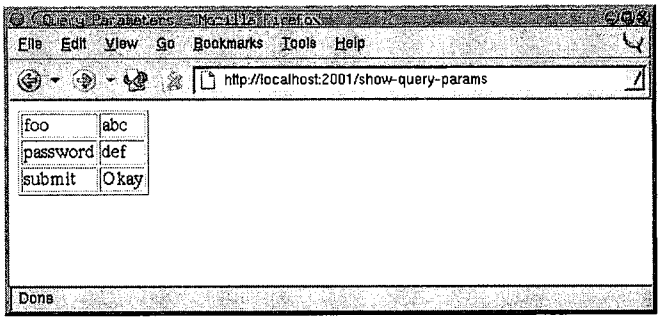

图26-6　提交一个简单表单的结果

尽管如此，多数时候你不需要在所有查询参数上迭代，你只需要提取单独的参数。例如，你可能想要修改random-number，令你传给**RANDOM**的限制值可以通过一个查询参数来提供。在这种情况下，你可以使用函数request-query-value，它接受一个请求对象和你想要查询的参数名并将其值以字符串的形式返回，或者当没有参数时返回**NIL**。一个参数化的random-number版本如下所示：

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let* ((*html-output* (request-reply-stream request))
             (limit-string (or (request-query-value "limit" request) ""))
             (limit (or (parse-integer limit-string :junk-allowed t) 1000)))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random limit)))))))))))
```

由于request-query-value可能返回**NIL**或一个空字符串，在把参数解析成一个用来传给**RANDOM**的数字时需要同时考虑这两种情况。你可以在绑定limit-string时当没有"limit"查询参数的情况下将它绑定到空字符串""，从而处理**NIL**的情形。然后，你可以使用带有:junk-allowed参数的**PARSE-INTEGER**来确保它要么返回**NIL**（如果不能从给定字符串中解析出整数的话）要么返回一个整数。在26.8节中，你将开发一些宏来使查询参数的提取和到多种类型的转换工作变得更加容易。

26.7 cookie

在AllegroServe中你可以发送一个Set-Cookie头来告诉浏览器保存一个cookie并将其随着后续请求一起发送，具体方法是，在with-http-response的主体中调用with-http-body之前调用函数set-cookie-header。该函数的第一个参数是请求对象，其余参数都是用来设定cookie中不同属性的关键字参数。其中两个你必须传递的是:name和:value参数，两者都应该是字符串。其他可能影响发送到浏览器的cookie的参数包括:expires、:path、:domain和:secure。

当然，你只需要担心:expires，它控制浏览器应该保存cookie多久。如果:expires是**NIL**（默认值），那么浏览器只把cookie保存到它被关闭时。其他可能的值是:never，这意味着cookie应当被永远保存下去，或者在一个由**GET-UNIVERSAL-TIME**或**ENCODE-UNIVERSAL-TIME**返回的全局时间里被保存。一个值为零的:expire参数告诉客户端立即丢弃已有的cookie。[①]

在设置了一个cookie以后，可以使用函数get-cookie-values得到一个alist，其中含有由浏览器发送的每个cookie对应的键值对。从这个alist中，可以使用**ASSOC**和**CDR**来提取单独的cookie值。

下面的函数可以显示出浏览器所发送的所有cookie的名字和值：

① 关于其他参数的含义，可参见AllegroServe文档和RFC 2109，这些文档里描述了cookie机制。

```
(defun show-cookies (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
           (:title "Cookies")
           (if (null (get-cookie-values request))
             (html (:p "No cookies."))
             (html
               (:table
                 (loop for (key . value) in (get-cookie-values request)
                    do (html (:tr (:td key) (:td value))))))))))))

(publish :path "/show-cookies" :function 'show-cookies)
```

第一次加载页面http://localhost:2001/show-cookies时，它应该会说“No cookies”，如图26-7所示，因为你还没有设置任何cookie。

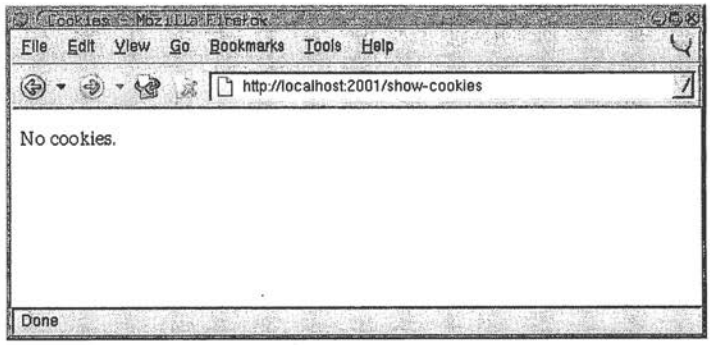

图26-7　没有cookies的http://localhost:2001/show-cookies

为了设置cookie，你需要另外一个函数，例如：

```
(defun set-cookie (request entity)
  (with-http-response (request entity :content-type "text/html")
    (set-cookie-header request :name "MyCookie" :value "A cookie value")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
           (:title "Set Cookie")
           (:p "Cookie set.")
           (:p (:a :href "/show-cookies" "Look at cookie jar."))))))))

(publish :path "/set-cookie" :function 'set-cookie)
```

如果你输入URL http://localhost:2001/set-cookie，那么你的浏览器应该会显示如图26-8所示的页面。同时，服务器将发送一个Set-Cookie头部，其中带有一个名为“MyCookie”值为“A cookie value”的cookie。如果你点击链接Look at cookie jar，那么你将被带到/show-cookies页面，在

那里你将看到新的cookie，如图26-9所示。由于你并未指定一个:expires参数，浏览器将继续在每个请求中发送该cookie，直到你关闭了浏览器。

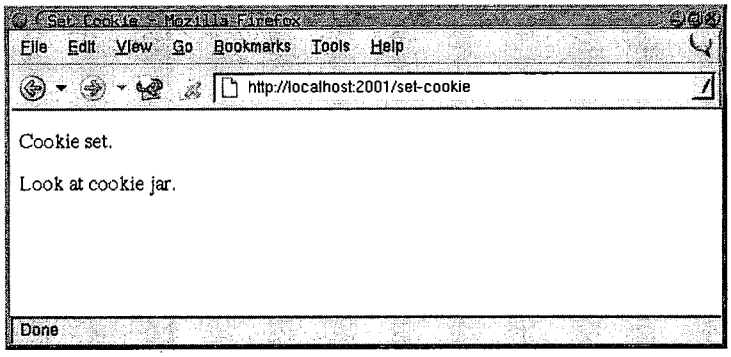

图26-8. http://localhost:2001/set-cookie

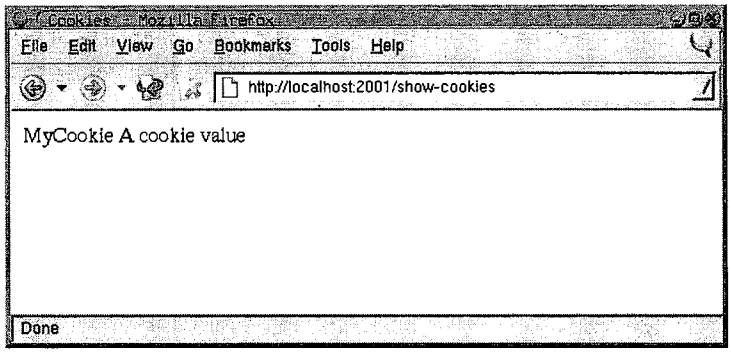

图26-9 设置cookie之后的http://localhost:2001/show-cookies

26.8 小型应用框架

尽管AllegroServe为你提供了用来编写服务端Web代码的几乎所有基本功能(访问URL的查询字符串和提交数据中的查询参数，设置和获取cookie值的能力，以及生成发给浏览器响应的能力)，但这需要写很多令人厌烦的重复性代码。

举个例子，你编写的每一个HTML生成函数都需要带有参数request和entity，并且它们都会包含对with-http-response、with-http-body以及（如果你打算使用FOO来生成HTML的话）with-html-output的调用。然后，在需要获取查询参数的函数里，还会有大量的request-query-value调用以及更多的代码来将这些字符串转化成任何你实际需要的类型。最后，你还需要记得publish这些函数。

为了减少样板代码的数量，可以在AllegroServer之上编写一个小型的框架，使其可以更容易地用来定义那些处理特定URL请求的函数。

基本的思路是先定义一个宏define-url-function，再用它来定义可自动通过publish发布的函数。这个宏展开成一个含有适当样板代码的DEFUN，以及在同名的URL下发布该函数的代码。它也负责生成代码来从查询参数和cookie中解出值，并将这些值绑定到声明在函数参数列表中的变量上。这样，define-url-function定义的基本形式如下所示：

```
(define-url-function name (request query-parameter*)
  body)
```

其中body是产生页面HTML的代码，它将被包装在一个对FOO的html宏的调用中，因此对于简单的页面来说，它只含有S-表达式形式的HTML。

在宏的主体中，查询参数变量将被绑定到同名查询参数或来自一个cookie的值上。在最简单的情形下，一个查询参数的值是同名的查询参数或POST数据字段中的字符串。如果查询参数使用列表来指定，还可以指定自动的类型转换、默认值以及是否在cookie中查找并保存该值。*query-parameter*的完整语法如下所示：

```
name | (name type [default-value] [stickiness])
```

其中的*type*必须是一个define-url-function可以识别的名字。我将很快讨论如何定义新的类型。*default-value*必须是该给定类型的一个值。最后，如果有*stickness*，它表示参数的值应当在没有查询参数的情况下从一个适当命名的cookie中获取，并且Set-Cookie头部应当在响应中发送，其中保存了同名cookie的值。这样，在显式地通过一个查询参数的值来指定粘滞参数以后，它将在该页面的后续请求中保持该值，即便没有查询参数被提供。

所使用的cookie名取决于*stickness*的值：使用值:global，cookie将采用与参数相同的命名方式。这样，使用同名的全局粘滞参数的不同函数将共享其值。如果*stickness*是:package，那么cookie的名字将根据参数的名字和函数名所在的包构造出来，这样来自同一个包的函数可以共享一些值，且不必担心被其他包里的函数参数所破坏。最后，一个带有*stickness*值为:local的参数将使用由参数名、函数名所在的包以及函数名构成的cookie，这对该函数而言将是唯一的。

举个例子，你可以使用define-url-function来将之前random-page的17行定义替换成下面的5行：

```
(define-url-function random-number (request (limit integer 1000))
  (:html
    (:head (:title "Random"))
    (:body
      (:p "Random number: " (:print (random limit))))))
```

如果要限制参数为粘滞的，你可以将limit的声明改成(limit integer 1000 :local)。

26.9　上述框架的实现

我将会自顶向下地解释define-url-function的实现。该宏本身如下所示：

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
```

```
`(progn
   (defun ,name (,request ,entity)
     (with-http-response (,request ,entity :content-type "text/html")
       (let* (,@(param-bindings name request params))
         ,@(set-cookies-code name request params)
         (with-http-body (,request ,entity)
           (with-html-output ((request-reply-stream ,request))
             (html ,@body))))))
   (publish :path ,(format nil "/~(~a~)" name) :function ',name)))))
```

让我们一点一点地分析它，首先看最初的几行。

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
```

直到这里你才开始生成代码。你可以用**GENSYM**生成一个符号以便在后面的**DEFUN**里把它作为实体参数的名字来使用。然后正则化所有参数，使用下列函数将简单的符号转化成列表形式：

```
(defun normalize-param (param)
  (etypecase param
    (list param)
    (symbol `(,param string nil nil))))
```

换句话说，只用一个符号来声明参数等价于声明不带默认值的非粘滞字符串参数。

接下来是**PROGN**。你必须展开成一个**PROGN**，因为你需要生成代码来做两件事：用**DEFUN**定义函数以及调用publish。你应当首先定义该函数，这样如果定义中出现错误，那么该函数将不会被发布。**DEFUN**的前两行只是一些样板代码。

```
(defun ,name (,request ,entity)
  (with-http-response (,request ,entity :content-type "text/html")
```

现在开始做实际工作。接下来两行将为在define-url-function中指定的除request以外的参数生成绑定，以及为粘滞性参数调用set-cookie-header的代码。当然，实际的工作是由你即将看到的助手函数来完成的。[①]

```
(let* (,@(param-bindings name request params))
  ,@(set-cookies-code name request params)
```

其余的就只是些样板代码了，它们将来自define-url-function定义的主体放在适当的with-http-body、with-html-output和html宏的上下文中。然后是对publish的调用。

```
(publish :path ,(format nil "/~(~a~~)" name) :function ',name)
```

表达式(format nil "/~(~a~)" name)在宏展开阶段求值，生成一个由“/”后跟你定义的这个函数名的全小写版本。该字符串随后成为publish的:path参数，而函数名则作为:function参数被插入。

① 你需要使用**LET***而非**LET**来使参数的默认值形式可以引用更早出现在参数列表中的参数。例如，可以写成下面这样：

```
(define-url-function (request (x integer 10) (y integer (* 2 x))) ...)
```

从而允许当没有显式提供y的值时，则使用x值的两倍。

再看一看用来生成**DEFUN**形式的助手函数。为了生成参数绑定，需要在params上循环并收集每个参数的由param-binding生成的代码片段。该片段是一个含有需要绑定的变量名和用来计算该变量值的代码的列表。用来计算值的代码的确切形式取决于参数的类型是否为粘滞的以及其默认值，如果有的话。因为你已经正则化了所有的参数，所以你可以在param-binding中使用**DESTRUCTURING-BIND**来将各部分取出。

```
(defun param-bindings (function-name request params)
  (loop for param in params
     collect (param-binding function-name request param)))

(defun param-binding (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (let ((query-name (symbol->query-name name))
          (cookie-name (symbol->cookie-name function-name name sticky)))
      `(,name (or
               (string->type ',type (request-query-value ,query-name ,request))
               ,@(if cookie-name
                     (list `(string->type ',type
                                          (get-cookie-value ,request ,cookie-name))))
               ,default)))))
```

函数string->type用来将那些从查询参数和cookie中获取的字符串转化成你想要的类型，它是一个下列形式的广义函数：

```
(defgeneric string->type (type value))
```

为了让一个特定的名字可被用作某个查询参数的类型名，只需在string->type上定义一个方法。你需要至少定义一个特化在符号string上的方法，因为这是默认类型。当然，这很容易做到。由于浏览器有时会提交带有空字符串的表单，以表明没有值提供给某个特定的变量，你需要采用如下的方法把空字符串转化成**NIL**：

```
(defmethod string->type ((type (eql 'string)) value)
  (and (plusp (length value)) value))
```

可以为应用程序所需的其他类型添加转换方法。例如，为了使integer成为一个可用的查询参数类型，从而可以处理random-page中的limit参数，你可以定义下列方法：

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

另一个在param-binding生成的代码中用到的助手函数是get-cookie-value，它只是由AllegroServe提供的get-cookie-values函数外围的一点儿语法糖：

```
(defun get-cookie-value (request name)
  (cdr (assoc name (get-cookie-values request) :test #'string=)))
```

类似地，用来计算查询参数和cookie名的函数也相当直接。

```
(defun symbol->query-name (sym)
  (string-downcase sym))

(defun symbol->cookie-name (function-name sym sticky)
```

```
  (let ((package-name (package-name (symbol-package function-name))))
    (when sticky
      (ecase sticky
        (:global
         (string-downcase sym))
        (:package
         (format nil "~(~a:~a~)" package-name sym))
        (:local
         (format nil "~(~a:~a:~a~)" package-name function-name sym))))))
```

为了生成那些为粘滞性参数设置cookie的代码，需要再次循环参数列表，但这一次只收集来自每个粘滞性参数的代码片段。你可以使用when和collect it这两个LOOP形式来只收集那些由set-cookie-code返回的非空值。

```
(defun set-cookies-code (function-name request params)
  (loop for param in params
        when (set-cookie-code function-name request param) collect it))

(defun set-cookie-code (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (declare (ignore type default))
    (if sticky
      `(when ,name
         (set-cookie-header
          ,request
          :name ,(symbol->cookie-name function-name name sticky)
          :value (princ-to-string ,name))))))
```

像这样用助手函数来定义宏的一大优点是，很容易确保生成的代码单独看起来是正确的。例如，你可以检查下面的set-cookie-code：

```
(set-cookie-code 'foo 'request '(x integer 20 :local))
```

是否生成了如下代码：

```
(WHEN X
  (SET-COOKIE-HEADER REQUEST
    :NAME "com.gigamonkeys.web:foo:x"
    :VALUE (PRINC-TO-STRING X)))
```

假设这些代码将会出现在x为变量名的某个上下文中，那么它看起来是正确的。

宏再次使你的代码直击要害，在本例中，就是你想要从请求中解出的数据和你想要生成的HTML。这就是说，该框架并不是一个大而全的Web应用框架，而只是可以让将在第29章编写的简单应用更容易一些的语法糖。

但在此之前，你还需要编写应用程序的功能性部分，相对来说第29章的应用将成为用户界面。下一章，我们要编写一个之前在第3章里编写的数据库的增强版，这一次用它来跟踪从MP3文件中解出的ID3数据。