

# 第一章：引言

在过去的五年里，Web 开发人员的可用工具实现了跨越式地增长。当技术专家不断推动极限，使 Web 应用无处不在时，我们也不得不升级我们的工具、创建框架以保证构建更好的应用。我们希望能够使用新的工具，方便我们写出更加整洁、可维护的代码，使部署到世界各地的用户时拥有高效的可扩展性。

这就让我们谈论到 Tornado，一个编写易创建、扩展和部署的强力 Web 应用的梦幻选择。我们三个都因为 Tornado 的速度、简单和可扩展性而深深地爱上了它，在一些个人项目中尝试之后，我们将其运用到日常工作中。我们已经看到，Tornado 在很多大型或小型的项目中提升了开发者的速度（和乐趣！），同时，其鲁棒性和轻量级也给开发者一次又一次留下了深刻的印象。

本书的目的是对 Tornado Web 服务器进行一个概述，通过框架基础、一些示例应用和真实世界使用的最佳实践来引导读者。我们将使用示例来详细讲解 Tornado 如何工作，你可以用它做什么，以及在构建自己第一个应用时要避免什么。

在本书中，我们假定你对 Python 已经有了粗略的了解，知道 Web 服务如何运作，对数据库有一定的熟悉。有一些不错的书籍可以为你深入了解这些提供参考（比如 Learning Python, Restful Web Service 和 MongoDB: The Definitive Guide）。

你可以在 [Github](#) 上获得本书中示例的代码。如果你有关于这些示例或其他方面的任何思想，欢迎在那里告诉我们。

所以，事不宜迟，让我们开始深入了解吧！

## 1.1 Tornado 是什么？

Tornado 是使用 Python 编写的一个强大的、可扩展的 Web 服务器。它在处理严峻的网络流量时表现得足够强健，但却在创建和编写时有着足够的轻量级，并能够被用在大量的应用和工具中。

我们现在所知道的 Tornado 是基于 Bret Taylor 和其他人员为 FriendFeed 所开发的网络服务框架，当 FriendFeed 被 Facebook 收购后得以开源。不同于那些最多只能达到 10,000 个并发连接的传统网络服务器，Tornado 在设计之初就考虑到了性能因素，旨在解决 C10K 问题，这样的设计使得其成为一个拥有非常高性能的框架。此外，它还拥有处理安全性、用户验证、社交网络以及与外部服务（如数据库和网站 API）进行异步交互的工具。

## 延伸阅读：C10K 问题

基于线程的服务器，如 Apache，为了传入的连接，维护了一个操作系统的线程池。Apache 会为每个 HTTP 连接分配线程池中的一个线程，如果所有的线程都处于被占用的状态并且尚有内存可用时，则生成一个新的线程。尽管不同的操作系统会有不同的设置，大多数 Linux 发布版中都是默认线程堆大小为 8MB。Apache 的架构在大负载下变得不可预测，为每个打开的连接维护一个大的线程池等待数据极易迅速耗尽服务器的内存资源。

大多数社交网络应用都会展示实时更新来提醒新消息、状态变化以及用户通知，这就要求客户端需要保持一个打开的连接来等待服务器端的任何响应。这些长连接或推送请求使得 Apache 的最大线程池迅速饱和。一旦线程池的资源耗尽，服务器将不能再响应新的请求。

异步服务器在这一场景中的应用相对较新，但他们正是被设计用来减轻基于线程的服务器的限制的。当负载增加时，诸如 Node.js, lighttpd 和 Tornado 这样的服务器使用协作的多任务的方式进行优雅的扩展。也就是说，如果当前请求正在等待来自其他资源的数据（比如数据库查询或 HTTP 请求）时，一个异步服务器可以明确地控制以挂起请求。异步服务器用来恢复暂停的操作的一个常见模式是当合适的数据准备好时调用回调函数。我们将会在[第五章](#)讲解回调函数模式以及一系列 Tornado 异步功能的应用。

自从 2009 年 9 月 10 日发布以来，Tornado 已经获得了很多社区的支持，并且在一系列不同的场合得到应用。除 FriendFeed 和 Facebook 外，还有很多公司在生产上转向 Tornado，包括 Quora、Turntable.fm、Bit.ly、Hipmunk 以及 MyYearbook 等。

总之，如果你在寻找你那庞大的 CMS 或一体化开发框架的替代品，Tornado 可能并不是一个好的选择。Tornado 并不需要你拥有庞大的模型建立特殊的方式，或以某种确定的形式处理表单，或其他类似的事情。它所做的是让你能够快速简单地编写高速的 Web 应用。如果你想编写一个可扩展的社交应用、实时分析引擎，或 RESTful API，那么简单而强大的 Python，以及 Tornado（和这本书）正是为你准备的！

### 1.1.1 Tornado 入门

在大部分 \*nix 系统中安装 Tornado 非常容易——你既可以从 PyPI 获取（并使用 easy\_install 或 pip 安装），也可以从 Github 上下载源码编译安装，如下所示 [1]：

```
$ curl -L -O https://github.com/facebook/tornado/archive/v3.1.0.tar.gz
$ tar xvzf v3.1.0.tar.gz
$ cd tornado-3.1.0
$ python setup.py build
$ sudo python setup.py install
```

Tornado 官方并不支持 Windows，但你可以通过 ActivePython 的 PyPM 包管理器进行安装，类似如下所示：

```
C:\> pypm install tornado
```

一旦 Tornado 在你的机器上安装好，你就可以很好的开始了！压缩包中包含很多 demo，比如建立博客、整合 Facebook、运行聊天服务等示例代码。我们稍后会在本书中通过一些示例应用逐步讲解，不过你也应该看看这些官方 demo。

本书中的代码假定你使用的是基于 Unix 的系统，并且使用的是 Python 2.6 或 2.7 版本。如果是这样，你就不需要任何除了 Python 标准库之外的东西。如果你的 Python 版本是 2.5 或更低，在安装 *pycURL*、*simpleJSON* 和 Python 开发头文件后可以运行 Tornado。[2]

### 1.1.2 社区和支持

对于问题、示例和一般的指南，Tornado 官方文档是个不错的选择。在 [tornadoweb.org](http://tornadoweb.org) 上有大量的例子和功能缺陷，更多细节和变更可以在 [Tornado 在 Github 上的版本库](#) 中看到。而对于更具体的问题，可以到 [Tornado 的 Google Group](#) 中咨询，那里有很多活跃的日常工作使用 Tornado 的开发者。

## 1.2 简单的 Web 服务

既然我们已经知道了 Tornado 是什么了，现在让我们看看它能做什么吧。我们首先从使用 Tornado 编写一个简单的 Web 应用开始。

### 1.2.1 Hello Tornado

Tornado 是一个编写对 HTTP 请求响应的框架。作为程序员，你的工作是编写响应特定条件 HTTP 请求的响应的 handler。下面是一个全功能的 Tornado 应用的基础示例：

代码清单 1-1 基础：hello.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', friendly user!')

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

编写一个 Tornado 应用中最多的工作是定义类继承 Tornado 的 *RequestHandler* 类。在这个例子中，我们创建了一个简单的应用，在给定的端口监听请求，并在根目录（"/"）响应请求。

你可以在命令行里尝试运行这个程序以测试输出：

```
$ python hello.py --port=8000
```

现在你可以在浏览器中打开 <http://localhost:8000/>，或者打开另一个终端窗口使用 curl 测试我们的应用：

```
$ curl http://localhost:8000/
Hello, friendly user!
$ curl http://localhost:8000/?greeting=Salutations
Salutations, friendly user!
```

让我们把这个例子分成小块，逐步分析它们：

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
```

在程序的最顶部，我们导入了一些 Tornado 模块。虽然 Tornado 还有另外一些有用的模块，但在这个例子中我们必须至少包含这四个模块。

```
from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)
```

Tornado 包括了一个有用的模块 (*tornado.options*) 来从命令行中读取设置。我们在这里使用这个模块指定我们的应用监听 HTTP 请求的端口。它的工作流程如下：如果一个与 *define* 语句中同名的设置在命令行中被给出，那么它将成为全局 *options* 的一个属性。如果用户运行程序时使用了一个 *help* 选项，程序将打印出所有你定义的选项以及你在 *define* 函数的 *help* 参数中指定的文本。如果用户没有为这个选项指定值，则使用 *default* 的值进行代替。Tornado 使用 *type* 参数进行基本的参数类型验证，当不合适的类型被给出时抛出一个异常。因此，我们允许一个整数的 *port* 参数作为 *options.port* 来访问程序。如果用户没有指定值，则默认为 8000。

```
class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', friendly user!')
```

这是 Tornado 的请求处理函数类。当处理一个请求时，Tornado 将这个类实例化，并调用与 HTTP 请求方法所对应的方法。在这个例子中，我们只定义了一个 *get* 方法，也就是说这个处理函数将对 HTTP 的 *GET* 请求作出响应。我们稍后将看到实现不止一个 HTTP 方法的处理函数。

```
greeting = self.get_argument('greeting', 'Hello')
```

Tornado 的 *RequestHandler* 类有一系列有用的内建方法，包括 *get\_argument*，我们在这里从一个查询字符串中取得参数 *greeting* 的值。（如果这个参数没有出现在查询字符串中，Tornado 将使用 *get\_argument* 的第二个参数作为默认值。）

```
self.write(greeting + ', friendly user!')
```

*RequestHandler* 的另一个有用的方法是 *write*，它以一个字符串作为函数的参数，并将其写入到 HTTP 响应中。在这里，我们使用请求中 *greeting* 参数提供的值插入到 *greeting* 中，并写回到响应中。

```
if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
```

这是真正使得 Tornado 运转起来的语句。首先，我们使用 Tornado 的 *options* 模块来解析命令行。然后我们创建了一个 Tornado 的 *Application* 类的实例。传递给 *Application* 类 *\_\_init\_\_* 方法的最重要的参数是 *handlers*。它告诉 Tornado 应该用哪个类来响应请求。马上我们讲解更多相关知识。

```
http_server = tornado.httpserver.HTTPServer(app)
http_server.listen(options.port)
```

```
tornado.ioloop.IOLoop.instance().start()
```

从这里开始的代码将会被反复使用：一旦 *Application* 对象被创建，我们可以将其传递给 Tornado 的 *HTTPServer* 对象，然后使用我们在命令行指定的端口进行监听（通过 *options* 对象取出。）最后，在程序准备好接收 HTTP 请求后，我们创建一个 Tornado 的 *IOLoop* 的实例。

### 1.2.1.1 参数 *handlers*

让我们再看一眼 *hello.py* 示例中的这一行：

```
app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
```

这里的参数 *handlers* 非常重要，值得我们更加深入的研究。它应该是一个元组组成的列表，其中每个元组的第一个元素是一个用于匹配的正则表达式，第二个元素是一个 *RequestHandler* 类。在 *hello.py* 中，我们只指定了一个正则表达式-*RequestHandler* 对，但你可以按你的需要指定任意多个。

### 1.2.1.2 使用正则表达式指定路径

Tornado 在元组中使用正则表达式来匹配 HTTP 请求的路径。（这个路径是 URL 中主机名后面的部分，不包括查询字符串和碎片。）Tornado 把这些正则表达式看作已经包含了行开始和结束锚点（即，字符串 `"/"` 被看作为 `"/"`）。

如果一个正则表达式包含一个捕获分组（即，正则表达式中的部分被括号括起来），匹配的内容将作为相应 HTTP 请求的参数传到 *RequestHandler* 对象中。我们将在下个例子中看到它的用法。

### 1.2.2 字符串服务

例 1-2 是一个我们目前为止看到的更复杂的例子，它将介绍更多 Tornado 的基本概念。

代码清单 1-2 处理输入：string\_service.py

```
import textwrap

import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class ReverseHandler(tornado.web.RequestHandler):
    def get(self, input):
        self.write(input[::-1])

class WrapHandler(tornado.web.RequestHandler):
    def post(self):
        text = self.get_argument('text')
        width = self.get_argument('width', 40)
```

```

        self.write(textwrap.fill(text, int(width)))

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(
        handlers=[
            (r"/reverse/(\w+)", ReverseHandler),
            (r"/wrap", WrapHandler)
        ]
    )
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

如同运行第一个例子，你可以在命令行中运行这个例子使用如下的命令：

```
$ python string_service.py --port=8000
```

这个程序是一个通用的字符串操作的 Web 服务端基本框架。到目前为止，你可以用它做两件事情。其一，到 `/reverse/string` 的 *GET* 请求将会返回 URL 路径中指定字符串的反转形式。

```
$ curl http://localhost:8000/reverse/stressed
desserts
```

```
$ curl http://localhost:8000/reverse/slipup
pupils
```

其二，到 `/wrap` 的 *POST* 请求将从参数 *text* 中取得指定的文本，并返回按照参数 *width* 指定宽度装饰的文本。下面的请求指定一个没有宽度的字符串，所以它的输出宽度被指定为程序中的 *get\_argument* 的默认值 40 个字符。

```
$ http://localhost:8000/wrap -d
text>Lorem+ipsum+dolor+sit+amet,+consectetuer+adipiscing+elit.
Lorem ipsum dolor sit amet, consectetur
adipiscing elit.
```

字符串服务示例和上一节示例代码中大部分是一样的。让我们关注那些新的代码。首先，让我们看看传递给 *Application* 构造函数的 *handlers* 参数的值：

```

app = tornado.web.Application(handlers=[
    (r"/reverse/(\w+)", ReverseHandler),
    (r"/wrap", WrapHandler)
])

```

在上面的代码中，*Application* 类在 “handlers” 参数中实例化了两个 *RequestHandler* 类对象。第一个引导 Tornado 传递路径匹配下面的正则表达式的请求：

```
/reverse/(\w+)
```

正则表达式告诉 Tornado 匹配任何以字符串 `/reverse/` 开始并紧跟着一个或多个字母的路径。括号的含义是让 Tornado 保存匹配括号里面表达式的字符串，并将其作为请求方法的一个参数传递给 `RequestHandler` 类。让我们检查 `ReverseHandler` 的定义来看看它是如何工作的：

```
class ReverseHandler(tornado.web.RequestHandler):
    def get(self, input):
        self.write(input[::-1])
```

你可以看到这里的 `get` 方法有一个额外的参数 `input`。这个参数将包含匹配处理函数正则表达式第一个括号里的字符串。（如果正则表达式中有一系列额外的括号，匹配的字符串将被按照在正则表达式中出现的顺序作为额外的参数传递进来。）

现在，让我们看一下 `WrapHandler` 的定义：

```
class WrapHandler(tornado.web.RequestHandler):
    def post(self):
        text = self.get_argument('text')
        width = self.get_argument('width', 40)
        self.write(textwrap.fill(text, int(width)))
```

`WrapHandler` 类处理匹配路径为 `/wrap` 的请求。这个处理函数定义了一个 `post` 方法，也就是说它接收 HTTP 的 `POST` 方法的请求。

我们之前使用 `RequestHandler` 对象的 `get_argument` 方法来捕获请求查询字符串的参数。同样，我们也可以使用相同的方法来获得 `POST` 请求传递的参数。（Tornado 可以解析 `URLencoded` 和 `multipart` 结构的 `POST` 请求）。一旦我们从 `POST` 中获得了文本和宽度的参数，我们使用 Python 内建的 `textwrap` 模块来以指定的宽度装饰文本，并将结果字符串写回到 HTTP 响应中。

### 1.2.3 关于 RequestHandler 的更多知识

到目前为止，我们已经了解了 `RequestHandler` 对象的基础：如何从一个传入的 HTTP 请求中获得信息（使用 `get_argument` 和传入到 `get` 和 `post` 的参数）以及写 HTTP 响应（使用 `write` 方法）。除此之外，还有很多需要学习的，我们将在接下来的章节中进行讲解。同时，还有一些关于 `RequestHandler` 和 Tornado 如何使用它的只是需要记住。

#### 1.2.3.1 HTTP 方法

截止到目前讨论的例子，每个 `RequestHandler` 类都只定义了一个 HTTP 方法的行为。但是，在同一个处理函数中定义多个方法是可能的，并且是有用的。把概念相关的功能绑定到同一个类是一个很好的方法。比如，你可能会编写一个处理函数来处理数据库中某个特定 ID 的对象，既使用 `GET` 方法，也使用 `POST` 方法。想象 `GET` 方法来返回这个部件的信息，而 `POST` 方法在数据库中对这个 ID 的部件进行改变：

```
# matched with (r"/widget/(\d+)", WidgetHandler)
class WidgetHandler(tornado.web.RequestHandler):
    def get(self, widget_id):
        widget = retrieve_from_db(widget_id)
        self.write(widget.serialize())
```



```
def post(self, widget_id):
    widget = retrieve_from_db(widget_id)
    widget['foo'] = self.get_argument('foo')
    save_to_db(widget)
```

我们到目前为止只是用了 *GET* 和 *POST* 方法，但 Tornado 支持任何合法的 HTTP 请求（*GET*、*POST*、*PUT*、*DELETE*、*HEAD*、*OPTIONS*）。你可以非常容易地定义上述任一种方法的行为，只需要在 *RequestHandler* 类中使用同名的方法。下面是另一个想象的例子，在这个例子中针对特定 *frob* ID 的 *HEAD* 请求只根据 *frob* 是否存在给出信息，而 *GET* 方法返回整个对象：

```
# matched with (r"/frob/(\d+)"), FrobHandler)
class FrobHandler(tornado.web.RequestHandler):
    def head(self, frob_id):
        frob = retrieve_from_db(frob_id)
        if frob is not None:
            self.set_status(200)
        else:
            self.set_status(404)
    def get(self, frob_id):
        frob = retrieve_from_db(frob_id)
        self.write(frob.serialize())
```

### 1.2.3.2 HTTP 状态码

从上面的代码可以看出，你可以使用 *RequestHandler* 类的 *set\_status()* 方法显式地设置 HTTP 状态码。然而，你需要记住在某些情况下，Tornado 会自动地设置 HTTP 状态码。下面是一个常用情况的纲要：

#### 404 Not Found

Tornado 会在 HTTP 请求的路径无法匹配任何 *RequestHandler* 类相对应的模式时返回 404（Not Found）响应码。

#### 400 Bad Request

如果你调用了一个没有默认值的 *get\_argument* 函数，并且没有发现给定名称的参数，Tornado 将自动返回一个 400（Bad Request）响应码。

#### 405 Method Not Allowed

如果传入的请求使用了 *RequestHandler* 中没有定义的 HTTP 方法（比如，一个 *POST* 请求，但是处理函数中只有定义了 *get* 方法），Tornado 将返回一个 405（Method Not Allowed）响应码。

#### 500 Internal Server Error

当程序遇到任何不能让其退出的错误时，Tornado 将返回 500（Internal Server Error）响应码。你代码中任何没有捕获的异常也会导致 500 响应码。

#### 200 OK



如果响应成功，并且没有其他返回码被设置，Tornado 将默认返回一个 200（OK）响应码。

当上述任何一种错误发生时，Tornado 将默认向客户端发送一个包含状态码和错误信息的简短片段。如果你想使用自己的方法代替默认的错误响应，你可以重写 `write_error` 方法在你的 `RequestHandler` 类中。比如，代码清单 1-3 是 `hello.py` 示例添加了常规的错误消息的版本。

代码清单 1-3 常规错误响应: `hello-errors.py`

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        greeting = self.get_argument('greeting', 'Hello')
        self.write(greeting + ', friendly user!')
    def write_error(self, status_code, **kwargs):
        self.write("Gosh darnit, user! You caused a %d error." % status_code)

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

当我们尝试一个 `POST` 请求时，会得到下面的响应。一般来说，我们应该得到 Tornado 默认的错误响应，但因为我们的覆写了 `write_error`，我们会得到不一样的东西：

```
$ curl -d foo=bar http://localhost:8000/
Gosh darnit, user! You caused a 405 error.
```

## 1.2.4 下一步<sup>[1]</sup>

现在你已经明白了最基本的东西，我们渴望你想了解更多。在接下来的章节，我们将向你展示能够帮助你使用 Tornado 创建成熟的 Web 服务和应用的功能和技术。首先是：Tornado 的模板系统。

[1] 压缩包地址已更新到 Tornado 的最新版本 3.1.0。

[2] 书中原文中关于 Python3.X 版本的兼容性问题目前已不存在，因此省略该部分。

# 第二章：表单和模板<sup>[1]</sup>

在[第一章](#)中，我们学习了使用 Tornado 创建一个 Web 应用的基础知识。包括处理函数、HTTP 方法以及 Tornado 框架的总体结构。在这章中，我们将学习一些你在创建 Web 应用时经常会用到的更强大的功能。

和大多数 Web 框架一样，Tornado 的一个重要目标就是帮助你更快地编写程序，尽可能整洁地复用更多的代码。尽管 Tornado 足够灵活，可以使用几乎所有 Python 支持的模板语言，Tornado 自身也提供了一个轻量级、快速并且灵活的模板语言在 *tornado.template* 模块中。

## 2.1 简单示例：Poem Maker Pro

让我们以一个叫作 Poem Maker Pro 的简单例子开始。Poem Maker Pro 这个 Web 应用有一个让用户填写的 HTML 表单，然后处理表单的结果。代码清单 2-1 是它的 Python 代码。

代码清单 2-1 简单表单和模板：poemmaker.py

```
import os.path

import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('index.html')

class PoemPageHandler(tornado.web.RequestHandler):
    def post(self):
        noun1 = self.get_argument('noun1')
        noun2 = self.get_argument('noun2')
        verb = self.get_argument('verb')
        noun3 = self.get_argument('noun3')
        self.render('poem.html', roads=noun1, wood=noun2, made=verb,
                    difference=noun3)

if __name__ == '__main__':
    tornado.options.parse_command_line()
    app = tornado.web.Application(
        handlers=[(r'/', IndexHandler), (r'/poem', PoemPageHandler)],
        template_path=os.path.join(os.path.dirname(__file__), "templates")
    )
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

除了 poemmaker.py，你还需要将代码清单 2-2 和代码清单 2-3 中的两个文件加入到 templates 子文件夹中。

代码清单 2-2 Poem Maker 表单：index.html

```
<!DOCTYPE html>
<html>
  <head><title>Poem Maker Pro</title></head>
  <body>
    <h1>Enter terms below.</h1>
    <form method="post" action="/poem">
      <p>Plural noun<br><input type="text" name="noun1"></p>
      <p>Singular noun<br><input type="text" name="noun2"></p>
      <p>Verb (past tense)<br><input type="text" name="verb"></p>
      <p>Noun<br><input type="text" name="noun3"></p>
      <input type="submit">
    </form>
  </body>
</html>
```

代码清单 2-3 Poem Maker 模板：poem.html

```
<!DOCTYPE html>
<html>
  <head><title>Poem Maker Pro</title></head>
  <body>
    <h1>Your poem</h1>
    <p>Two {{roads}} diverged in a {{wood}}, and I—<br>
    I took the one less travelled by,<br>
    And that has {{made}} all the {{difference}}.</p>
  </body>
</html>
```

在命令行执行下述命令：

```
$ python poemmaker.py --port=8000
```

现在，在浏览器中打开 <http://localhost:8000>。当浏览器请求根目录（/）时，Tornado 程序将渲染 index.html，展示如图 2-1 所示的简单 HTML 表单。

The screenshot shows a web browser window titled "Poem Maker Pro". The address bar displays "http://localhost:8000/". The main content area has the heading "Enter terms below." followed by four input fields with labels: "Plural noun" (containing "horoscopes"), "Singular noun" (containing "chinchilla"), "Verb (past tense)" (containing "decoupled"), and "Noun" (containing "pudding"). A "Submit" button is located below the "Noun" field.

图 2-1 Poem Maker Pro：输入表单

这个表单包括多个文本域（命名为 *noun1*、*noun2* 等），其中的内容将在用户点击“Submit”按钮时以 *POST* 请求的方式送到 /poem。现在往里面填写东西然后点击提交吧。

为了响应这个 *POST* 请求，Tornado 应用跳转到 poem.html，插入你在表单中填写的值。结果是 Robert Frost 的诗《The Road Not Taken》的轻微修改版本。图 2-2 展示了这个结果。

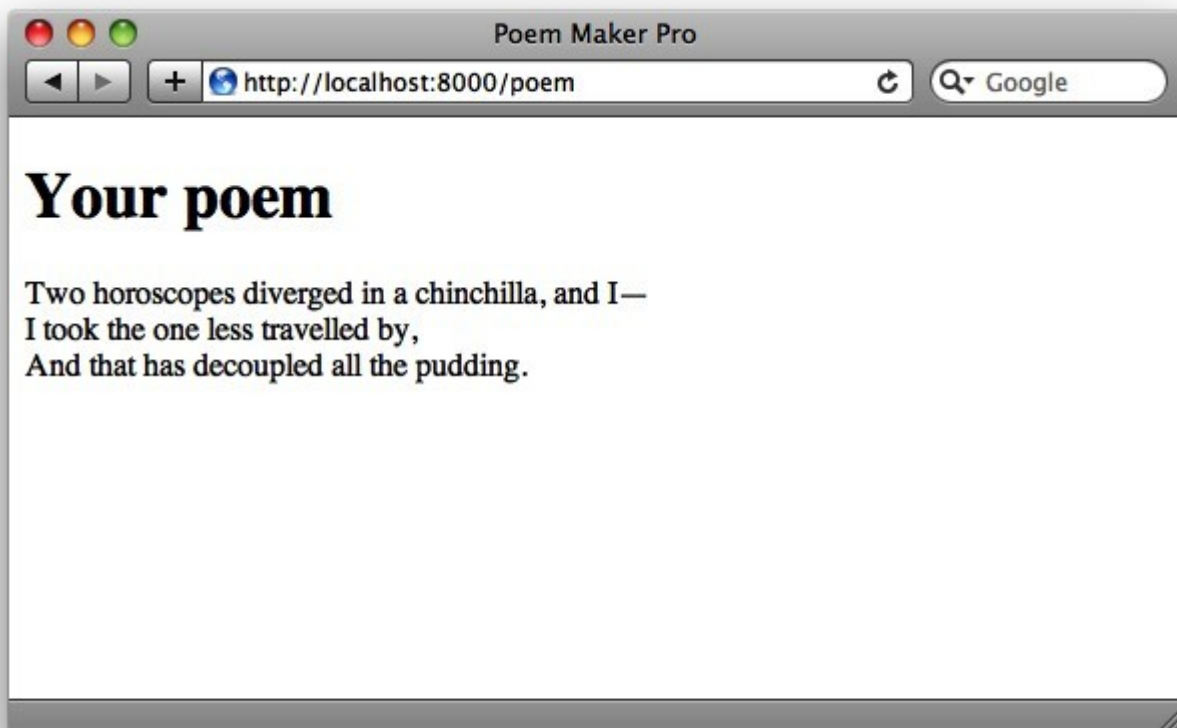


图 2-2 Poem Maker Pro: 输出

### 2.1.1 渲染模板¶

从结构上讲, *poemmaker.py* 和[第一章](#)中的例子很相似。我们定义了几个 *RequestHandler* 子类并把它们传给 *tornado.web.Application* 对象。那么有什么不一样的地方呢? 首先, 我们向 *Application* 对象的 *\_\_init\_\_* 方法传递了一个 *template\_path* 参数。

```
template_path=os.path.join(os.path.dirname(__file__), "templates")
```

*template\_path* 参数告诉 Tornado 在哪里寻找模板文件。我们将在本章和[第三章](#)中讲解其确切性质和语法, 而它的基本要点是: 模板是一个允许你嵌入 Python 代码片段的 HTML 文件。上面的代码告诉 Python 在你 Tornado 应用文件同目录下的 *templates* 文件夹中寻找模板文件。

一旦我们告诉 Tornado 在哪里找到模板, 我们可以使用 *RequestHandler* 类的 *render* 方法来告诉 Tornado 读入模板文件, 插入其中的模版代码, 并返回结果给浏览器。比如, 在 *IndexHandler* 中, 我们发现了下面的语句:

```
self.render('index.html')
```

这段代码告诉 Tornado 在 *templates* 文件夹下找到一个名为 *index.html* 的文件, 读取其中的内容, 并且发送给浏览器。

### 2.1.2 填充¶

实际上 `index.html` 完全不能称之为“模板”，它所包含的完全是已编写好的 HTML 标记。这可以是模板的一个不错的使用方式，但在更通常的情况下我们希望 HTML 输出可以结合我们的程序传入给模板的值。模板 `poem.html` 使用 *PoemPageHandler* 渲染，是这种方式的一个很好的例子。让我们看看它是如何工作的吧。

在 `poem.html` 中，你可以看到模板中有一些被双大括号（`{{和}}`）括起来的字符串，就像这样：

```
<p>Two {{roads}} diverged in a {{wood}}, and I—<br/>
I took the one less travelled by,<br>
And that has {{made}} all the {{difference}}.</p>
```

在双大括号中的单词是占位符，当我们渲染模板时希望以实际值代替。我们可以使用向 *render* 函数中传递关键字参数的方法指定什么值将被填充到 HTML 文件中的对应位置，其中关键字对应模板文件中占位符的名字。下面是在 *PoemPageHandler* 中相应的代码部分：

```
noun1 = self.get_argument('noun1')
noun2 = self.get_argument('noun2')
verb = self.get_argument('verb')
noun3 = self.get_argument('noun3')
self.render('poem.html', roads=noun1, wood=noun2, made=verb, difference=noun3)
```

在这里，我们告诉模板使用变量 *noun1*（该变量是从 *get\_argument* 方法取得的）作为模板中 *roads* 的值，*noun2* 作为模板中 *wood* 的值，依此类推。假设用户在表单中按顺序键入了 pineapples、grandfather clock、irradiated 和 supernovae，那么结果 HTML 将会如下所示：

```
<p>Two pineapples diverged in a grandfather clock, and I—<br>
I took the one less travelled by,<br>
And that has irradiated all the supernovae.</p>
```

## 2.2 模板语法

既然我们已经看到了一个模板在实际应用中的简单例子，那么让我们深入地了解它们是如何工作的吧。Tornado 模板是被 Python 表达式和控制语句标记的简单文本文件。Tornado 的语法非常简单直接。熟悉 Django、Liquid 或其他相似框架的用户会发现它们非常相似，很容易学会。

在 2.1 节中，我们展示了如何在一个 Web 应用中使用 *render* 方法传送 HTML 给浏览器。你可以在 Tornado 应用之外使用 Python 解释器导入模板模块尝试模板系统，此时结果会被直接输出出来。

```
>>> from tornado.template import Template
>>> content = Template("<html><body><h1>{{ header }}</h1></body></html>")
>>> print content.generate(header="Welcome!")
<html><body><h1>Welcome!</h1></body></html>
```

### 2.2.1 填充表达式

在代码清单 2-1 中，我们演示了填充 Python 变量的值到模板的双大括号中的使用。实际上，你可以将任何 Python 表达式放在双大括号中。Tornado 将插入一个包含任何表达式计算结果值的字符串到输出中。下面是几个可能的例子：

```
>>> from tornado.template import Template
>>> print Template("{} 1+1 {}").generate()
2
>>> print Template("{} 'scrambled eggs' [-4:] {}").generate()
eggs
>>> print Template("{} ', '.join([str(x*x) for x in range(10)]) {}").generate()
0, 1, 4, 9, 16, 25, 36, 49, 64, 81
```

## 2.2.2 控制流语句

你同样可以在 Tornado 模板中使用 Python 条件和循环语句。控制语句以 {%和%} 包围，并以类似下面的形式被使用：

```
{% if page is None %}
```

或

```
{% if len(entries) == 3 %}
```

控制语句的大部分就像对应的 Python 语句一样工作，支持 *if*、*for*、*while* 和 *try*。在这些情况下，语句块以 {%开始，并以%} 结束。

所以这个模板：

```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <h1>{{ header }}</h1>
    <ul>
      {% for book in books %}
        <li>{{ book }}</li>
      {% end %}
    </ul>
  </body>
</html>
```

当被下面这个处理函数调用时：

```
class BookHandler(tornado.web.RequestHandler):
    def get(self):
        self.render(
            "book.html",
            title="Home Page",
            header="Books that are great",
            books=[
                "Learning Python",
                "Programming Collective Intelligence",
```



```

        "Restful Web Services"
    ]
)

```

将会渲染得到下面的输出：

```

<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Books that are great</h1>
    <ul>
      <li>Learning Python</li>
      <li>Programming Collective Intelligence</li>
      <li>Restful Web Services</li>
    </ul>
  </body>
</html>

```

不像许多其他的 Python 模板系统，Tornado 模板语言的一个最好的东西是在 *if* 和 *for* 语句块中可以使用的表达式没有限制。因此，你可以在你的模板中执行所有的 Python 代码。

同样，你也可以在你的控制语句块中间使用 `{% set foo = 'bar' %}` 来设置变量。你还有很多可以在控制语句块中做的事情，但是在大多数情况下，你最好使用 UI 模块来做更复杂的划分。我们稍后会更详细的看到这一点。

### 2.2.3 在模板中使用函数

Tornado 在所有模板中默认提供了一些便利的函数。它们包括：

***escape(s)***

替换字符串 *s* 中的 `&`、`<`、`>` 为他们对应的 HTML 字符。

***url\_escape(s)***

使用 `urllib.quote_plus` 替换字符串 *s* 中的字符为 URL 编码形式。

***json\_encode(val)***

将 *val* 编码成 JSON 格式。（在系统底层，这是一个对 *json* 库的 *dumps* 函数的调用。查阅相关的文档以获得更多关于该函数接收和返回参数的信息。）

***squeeze(s)***

过滤字符串 *s*，把连续的多个空白字符替换成一个空格。

在 Tornado 1.x 中，模板不是被自动转义的。在 Tornado 2.0 中，模板被默认为自动转义（并且可以在 *Application* 构造函数中使用 *autoscaping=None* 关闭）。在不同版本的迁移时要注意向后兼容。

在模板中使用一个你自己编写的函数也是很简单的：只需要将函数名作为模板的参数传递即可，就像其他变量一样。

```
>>> from tornado.template import Template
>>> def disemvowel(s):
...     return ''.join([x for x in s if x not in 'aeiou'])
...
>>> disemvowel("george")
'grg'
>>> print Template("my name is {{d('mortimer')}}").generate(d=disemvowel)
my name is mrtmr
```

## 2.3 复杂示例：The Alpha Munger

在代码清单 2-4 中，我们把在这一章中谈论过的所有东西都放了进来。这个应用被称为 The Alpha Munger。用户输入两个文本：一个“源”文本和一个“替代”文本。应用会返回替代文本的一个副本，并将其中每个单词替换成源文本中首字母相同的某个单词。图 2-3 展示了要填的表单，图 2-4 展示了结果文本。

这个应用包括四个文件：main.py（Tornado 程序）、style.css（CSS 样式表文件）、index.html 和 munged.html（Tornado 模板）。让我们看看代码吧：

代码清单 2-4 复杂表单和模板：main.py

```
import os.path
import random

import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('index.html')

class MungedPageHandler(tornado.web.RequestHandler):
    def map_by_first_letter(self, text):
        mapped = dict()
        for line in text.split('\r\n'):
            for word in [x for x in line.split(' ') if len(x) > 0]:
                if word[0] not in mapped: mapped[word[0]] = []
```

```

        mapped[word[0]].append(word)
    return mapped

def post(self):
    source_text = self.get_argument('source')
    text_to_change = self.get_argument('change')
    source_map = self.map_by_first_letter(source_text)
    change_lines = text_to_change.split('\r\n')
    self.render('munged.html', source_map=source_map, change_lines=change_lines,
               choice=random.choice())

if __name__ == '__main__':
    tornado.options.parse_command_line()
    app = tornado.web.Application(
        handlers=[(r'/', IndexHandler), (r'/poem', MungedPageHandler)],
        template_path=os.path.join(os.path.dirname(__file__), "templates"),
        static_path=os.path.join(os.path.dirname(__file__), "static"),
        debug=True
    )
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

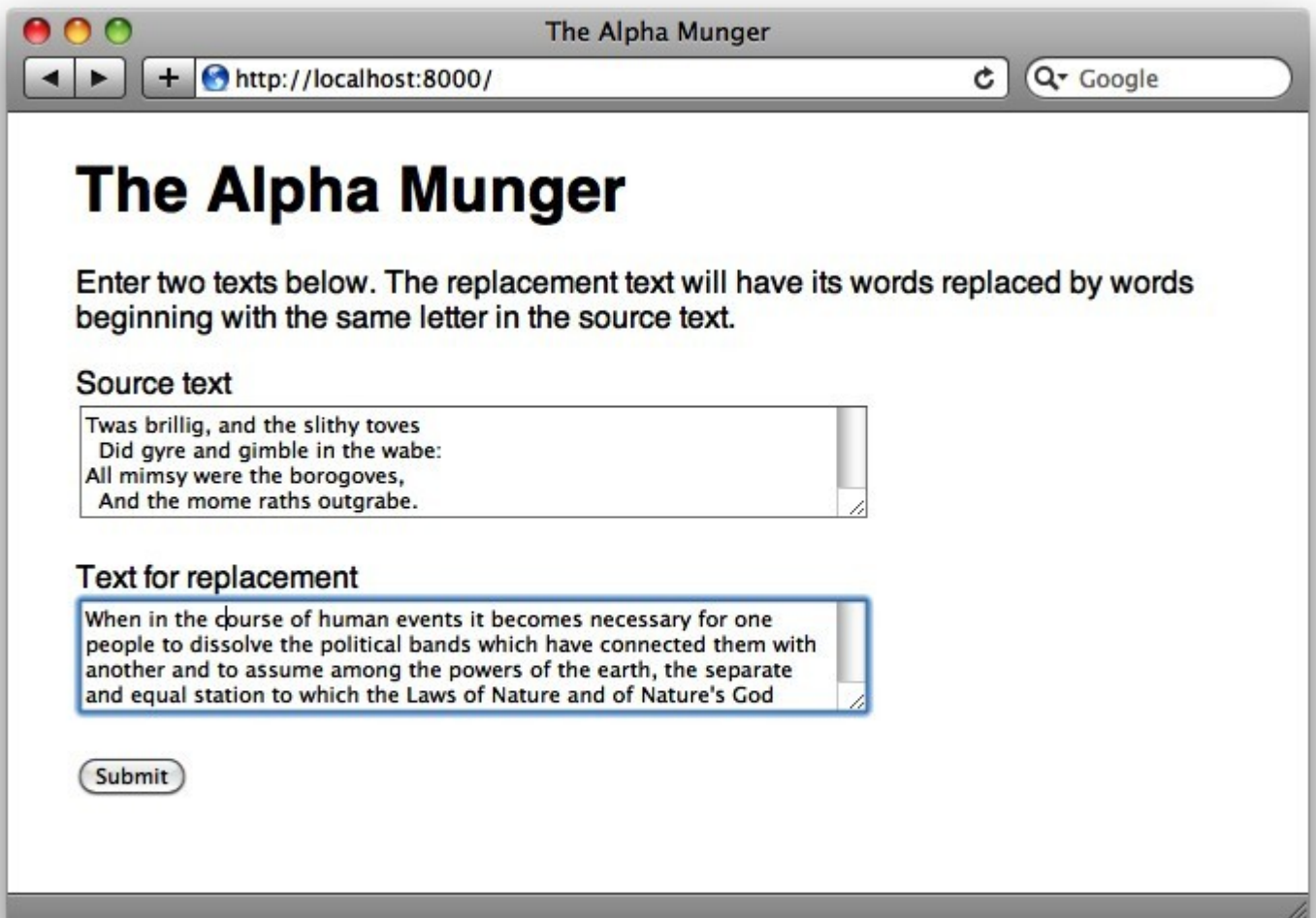


图 2-3 Alpha Munger: 输入表单

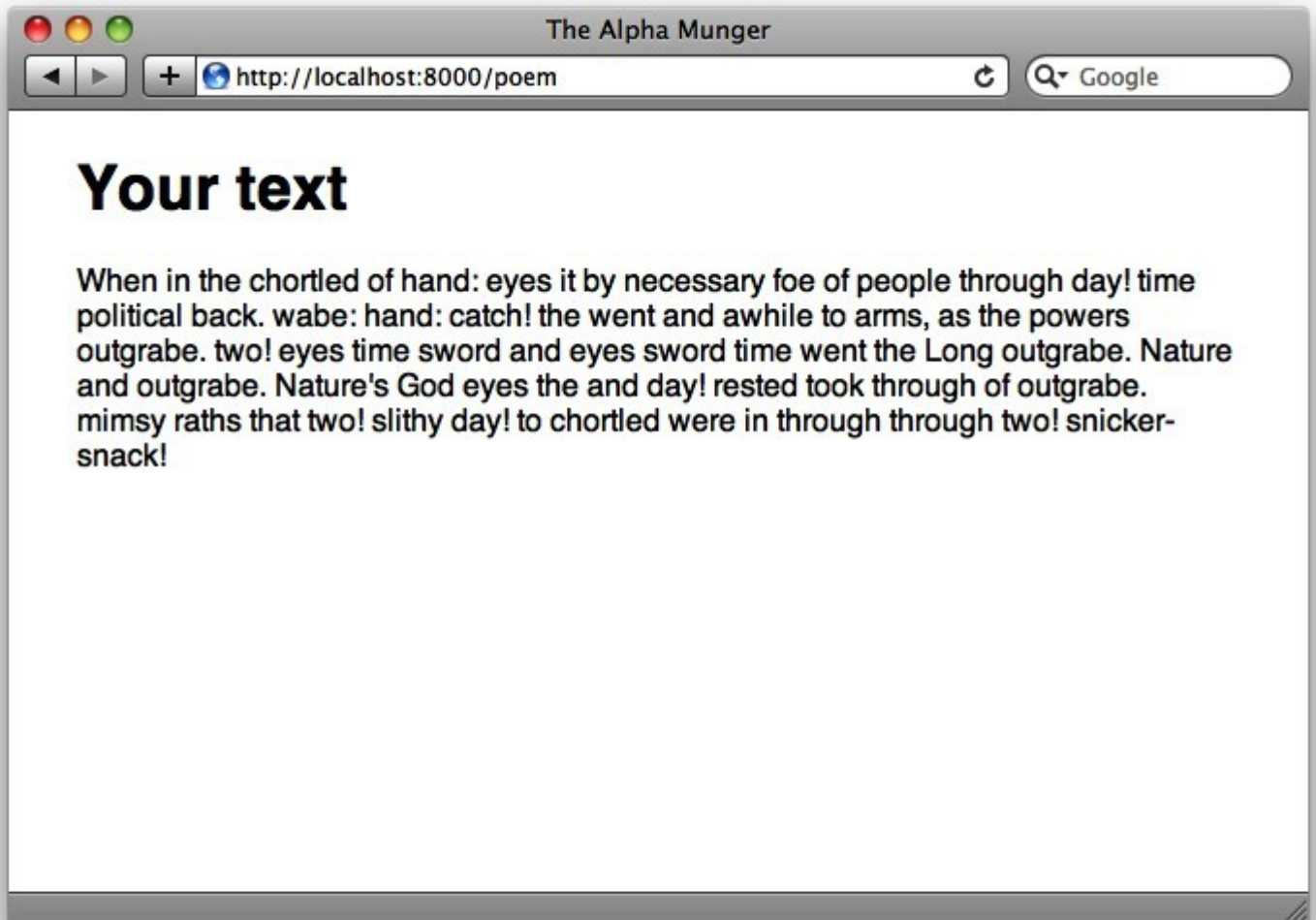


图 2-4 Alpha Munger: 输出

记住 *Application* 构造函数中的 *static\_path* 参数。我们将在下面进行详细的介绍，但是现在你所需要知道的就是 *static\_path* 参数指定了你应用程序放置静态资源（如图像、CSS 文件、JavaScript 文件等）的目录。另外，你还需要在 *templates* 文件夹下添加 *index.html* 和 *munged.html* 这两个文件。

代码清单 2-5 Alpha Munger 表单: *index.html*

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="{{ static_url("style.css") }}">
    <title>The Alpha Munger</title>
  </head>
  <body>
    <h1>The Alpha Munger</h1>
    <p>Enter two texts below. The replacement text will have its words
      replaced by words beginning with the same letter in the source text.</p>
    <form method="post" action="/poem">
    <p>Source text<br>
```

```

        <textarea rows=4 cols=55 name="source"></textarea></p>
    <p>Text for replacement<br>
        <textarea rows=4 cols=55 name="change"></textarea></p>
    <input type="submit">
</form>
</body>
</html>

```

代码清单 2-6 Alpha Munger 模板: munged.html

```

<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" href="{{ static_url("style.css") }}">
        <title>The Alpha Munger</title>
    </head>
    <body>
        <h1>Your text</h1>
        <p>
{% for line in change_lines %}
    {% for word in line.split(' ') %}
        {% if len(word) > 0 and word[0] in source_map %}
            <span class="replaced"
                title="{{word}}">{{ choice(source_map[word[0]]) }}</span>
        {% else %}
            <span class="unchanged" title="unchanged">{{word}}</span>
        {% end %}
    {% end %}
    <br>
{% end %}
        </p>
    </body>
</html>

```

最后, 将代码清单 2-7 中的内容写到 static 子目录下的 style.css 文件中。

代码清单 2-7 Alpha Munger 样式表: style.css

```

body {
    font-family: Helvetica, Arial, sans-serif;
    width: 600px;
    margin: 0 auto;
}
.replaced:hover { color: #00f; }

```

### 2.3.1 它如何工作

这个 Tornado 应用定义了两个请求处理类: *IndexHandler* 和 *MungedPageHandler*。 *IndexHandler* 类简单地渲染了 index.html 中的模板, 其中包括一个允许用户 *POST* 一个源文本 (在 *source* 域中) 和一个替换文本 (在 *change* 域中) 到 /poem 的表单。

*MungedPageHandler* 类用于处理到 /poem 的 *POST* 请求。当一个请求到达时，它对传入的数据进行一些基本的处理，然后为浏览器渲染模板。*map\_by\_first\_letter* 方法将传入的文本（从 *source* 域）分割成单词，然后创建一个字典，其中每个字母表中的字母对应文本中所有以其开头的单词（我们将其放入一个叫作 *source\_map* 的变量）。再把这个字典和用户替代文本（表单的 *change* 域）中指定的内容一起传给模板文件 *munged.html*。此外，我们还将 Python 标准库的 *random.choice* 函数传入模板，这个函数以一个列表作为输入，返回列表中的任一元素。

在 *munged.html* 中，我们迭代替代文本中的每行，再迭代每行中的每个单词。如果当前单词的第一个字母是 *source\_map* 字典的一个键，我们使用 *random.choice* 函数从字典的值中随机选择一个单词并展示它。如果字典的键中没有这个字母，我们展示源文本中的原始单词。每个单词包括一个 *span* 标签，其中的 *class* 属性指定这个单词是替换后的（*class="replaced"*）还是原始的（*class="unchanged"*）。（我们还将原始单词放到了 *span* 标签的 *title* 属性中，以便于用户在鼠标经过单词时可以查看是什么单词被替代了。你可以在图 2-5 中看到这个动作。）

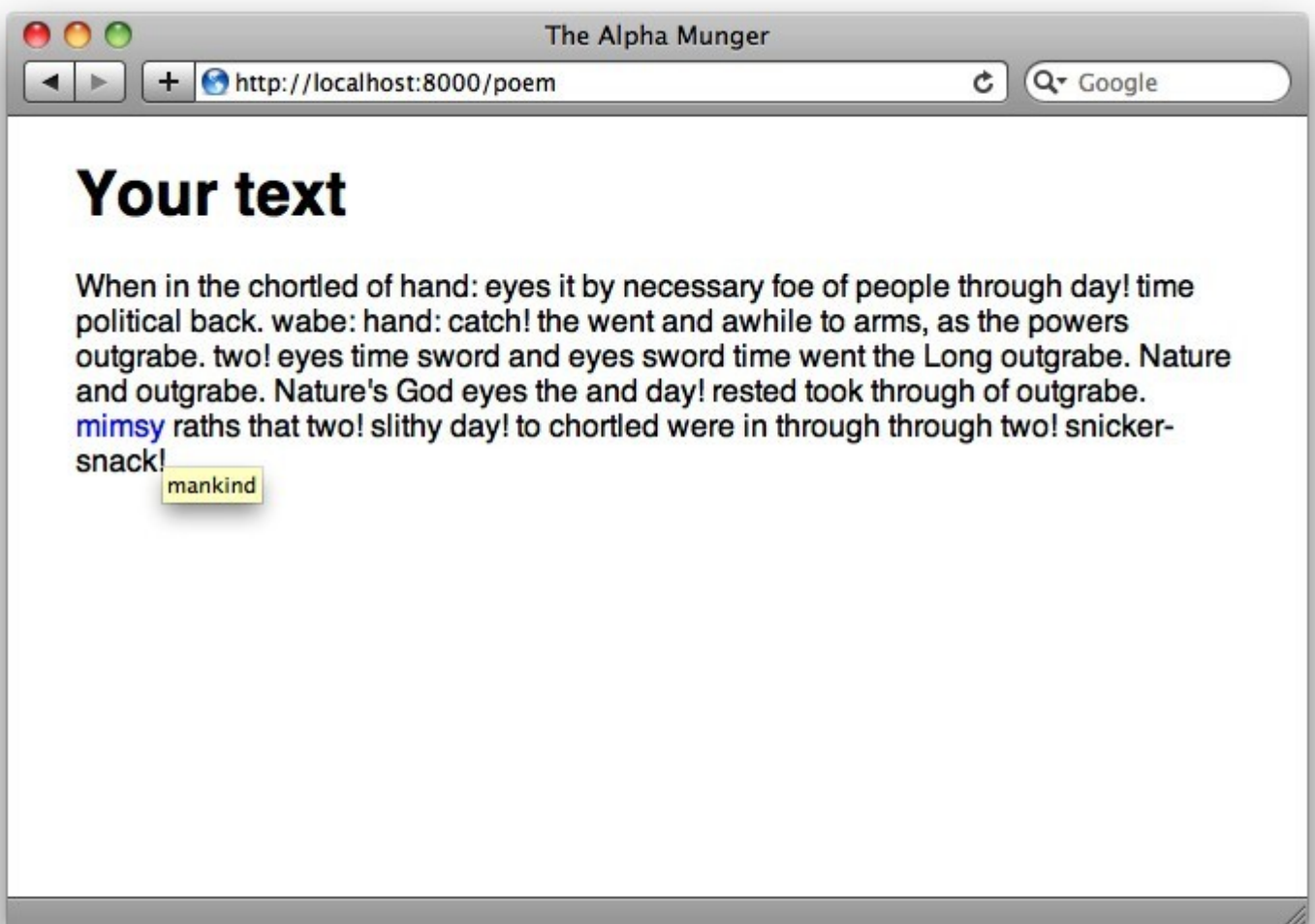


图 2-5 含有被替换单词提示的 Alpha Munger

在这个例子中，你可能注意到了 *debug=True* 的使用。它调用了一个便利的测试模式：*tornado.autoreload* 模块，此时，一旦主要的 Python 文件被修改，Tornado 将会尝试重启服务器，并且在模板改变时会进行刷新。对于快速改变和实时更新这非常棒，但不要再生产上使用它，因为它将防止 Tornado 缓存模板！

### 2.3.2 提供静态文件



当编写 Web 应用时，你总希望提供像样式表、JavaScript 文件和图像这样不需要为每个文件编写独立处理函数的“静态内容”。Tornado 提供了几个有用的捷径来使其变得容易。

### 2.3.2.1 设置静态路径

你可以通过向 *Application* 类的构造函数传递一个名为 *static\_path* 的参数来告诉 Tornado 从文件系统的一个特定位置提供静态文件。Alpha Munger 中的相关代码片段如下：

```
app = tornado.web.Application(
    handlers=[(r'/', IndexHandler), (r'/poem', MungedPageHandler)],
    template_path=os.path.join(os.path.dirname(__file__), "templates"),
    static_path=os.path.join(os.path.dirname(__file__), "static"),
    debug=True
)
```

在这里，我们设置了一个当前应用目录下名为 *static* 的子目录作为 *static\_path* 的参数。现在应用将以读取 *static* 目录下的 *filename.ext* 来响应诸如 */static/filename.ext* 的请求，并在响应的主体中返回。

### 2.3.2.2 使用 *static\_url* 生成静态 URL

Tornado 模板模块提供了一个叫作 *static\_url* 的函数来生成 *static* 目录下文件的 URL。让我们来看看在 *index.html* 中 *static\_url* 的调用的示例代码：

```
<link rel="stylesheet" href="{{ static_url("style.css") }}">
```

这个对 *static\_url* 的调用生成了 URL 的值，并渲染输出类似下面的代码：

```
<link rel="stylesheet" href="/static/style.css?v=ab12">
```

那么为什么使用 *static\_url* 而不是在你的模板中硬编码呢？有如下几个原因。其一，*static\_url* 函数创建了一个基于文件内容的 hash 值，并将其添加到 URL 末尾（查询字符串的参数 *v*）。这个 hash 值确保浏览器总是加载一个文件的最新版而不是之前的缓存版本。无论是在你应用的开发阶段，还是在部署到生产环境使用时，都非常有用，因为你的用户不必再为了看到你的静态内容而清除浏览器缓存了。

另一个好处是你可以改变你应用 URL 的结构，而不需要改变模板中的代码。例如，你可以配置 Tornado 响应来自像路径 */s/filename.ext* 的请求时提供静态内容，而不是默认的 */static* 路径。如果你使用 *static\_url* 而不是硬编码的话，你的代码不需要改变。比如说，你想把静态资源从我们刚才使用的 */static* 目录移到新的 */s* 目录。你可以简单地改变静态路径由 *static* 变为 *s*，然后每个使用 *static\_url* 包裹的引用都会被自动更新。如果你在每个引用静态资源的文件中硬编码静态路径部分，你将不得不手动修改每个模板。

### 2.3.3 模板的下一步

到目前为止，你已经能够处理 Tornado 模板系统的简单功能了。对于像 Alpha Munger 这样简单的 Web 应用而言，基础的功能对你而言足够用了。但是我们在模板部分的学习并没有结束。Tornado 在块和模块的形式上仍然有一些技巧，这两个功能使得编写和维护复杂的 Web 应用更加简单。我们将在[第三章](#)中看到这些功能。



## 第三章：模板扩展

在[第二章](#)中，我们看到了 Tornado 模板系统如何简单地传递信息给网页，使你在插入动态数据时保持网页标记的整洁。然而，大多数站点希望复用像 header、footer 和布局网格这样的内容。在这一章中，我们将看到如何使用扩展 Tornado 模板或 UI 模块完成这一工作。

### 3.1 块和替换

当你花时间为你的 Web 应用建立和制定模板时，希望像你的后端 Python 代码一样重用你的前端代码似乎只是合逻辑的，不是吗？幸运的是，Tornado 可以让你做到这一点。Tornado 通过 *extends* 和 *block* 语句支持模板继承，这就让你拥有了编写能够在合适的地方复用的流体模板的控制权和灵活性。

为了扩展一个已经存在的模板，你只需要在新的模板文件的顶部放上一句 `{% extends "filename.html" %}`。比如，为了在新模板中扩展一个父模板（在这里假设为 `main.html`），你可以这样使用：

```
{% extends "main.html" %}
```

这就使得新文件继承 `main.html` 的所有标签，并且覆写为期望的内容。

#### 3.1.1 块基础

扩展一个模板使你复用之前写过的代码更加简单，但是这并不会为你提供所有的东西，除非你可以适应并改变那些之前的模板。所以，*block* 语句出现了。

一个块语句压缩了一些当你扩展时可能想要改变的模板元素。比如，为了使用一个能够根据不同页覆写的动态 header 块，你可以在父模板 `main.html` 中添加如下代码：

```
<header>
    {% block header %} {% end %}
</header>
```

然后，为了在子模板 `index.html` 中覆写 `{% block header %} {% end %}` 部分，你可以使用块的名字引用，并把任何你想要的内容放到其中。

```
{% block header %} {% end %}

{% block header %}
    <h1>Hello world!</h1>
{% end %}
```

任何继承这个模板的文件都可以包含它自己的 `{% block header %}` 和 `{% end %}`，然后把一些不同的东西加进去。

为了在 Web 应用中调用这个子模板，你可以在你的 Python 脚本中很轻松地渲染它，就像之前你渲染其他模板那样：

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.render("index.html")
```

所以此时，main.html 中的 *body* 块在加载时会被以 index.html 中的信息“Hello world!”填充（参见图 3-1）。

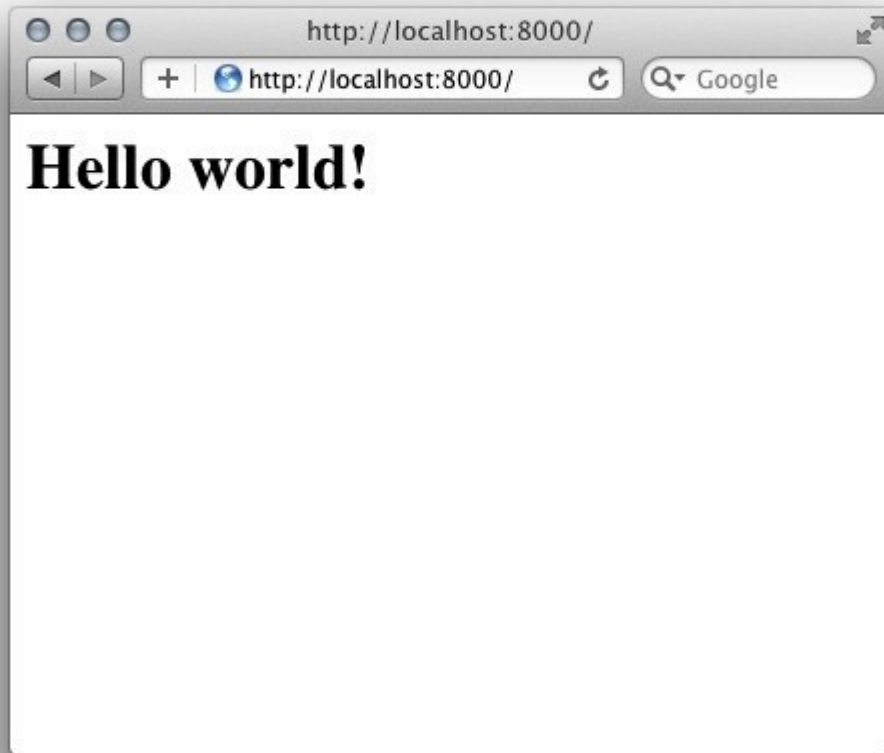


图 3-1 Hello world!

我们已经可以看到这种方法在处理整体页面结构和节约多页面网站的开发时间上多么有用。更好的是，你可以为每个页面使用多个块，此时像 header 和 footer 这样的动态元素将会被包含在同一个流程中。

下面是一个在父模板 main.html 中使用多个块的例子：

```
<html>
<body>
    <header>
        {% block header %} {% end %}
    </header>
    <content>
        {% block body %} {% end %}
    </content>
```

```

    <footer>
        {% block footer %} {% end %}
    </footer>
</body>
</html>

```

当我们扩展父模板 main.html 时，可以在子模板 index.html 中引用这些块。

```

{% extends "main.html" %}

{% block header %}
    <h1>{{ header_text }}</h1>
{% end %}

{% block body %}
    <p>Hello from the child template!</p>
{% end %}

{% block footer %}
    <p>{{ footer_text }}</p>
{% end %}

```

用来加载模板的 Python 脚本和上一个例子差不多，不过在这里我们传递了几个字符串变量给模板使用（如图 3-2）：

```

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.render(
            "index.html",
            header_text = "Header goes here",
            footer_text = "Footer goes here"
        )

```

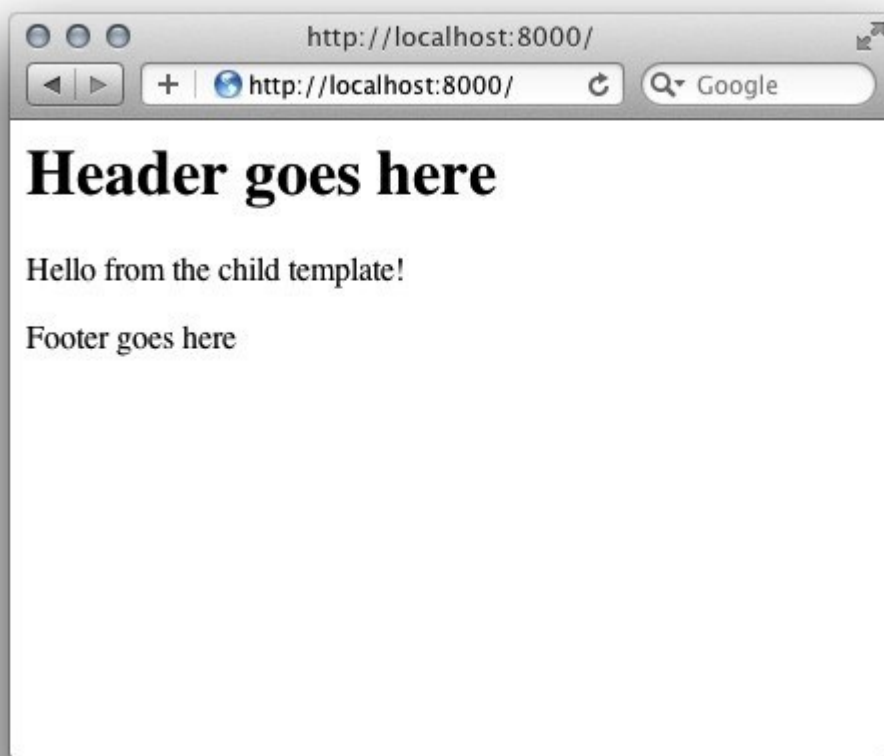
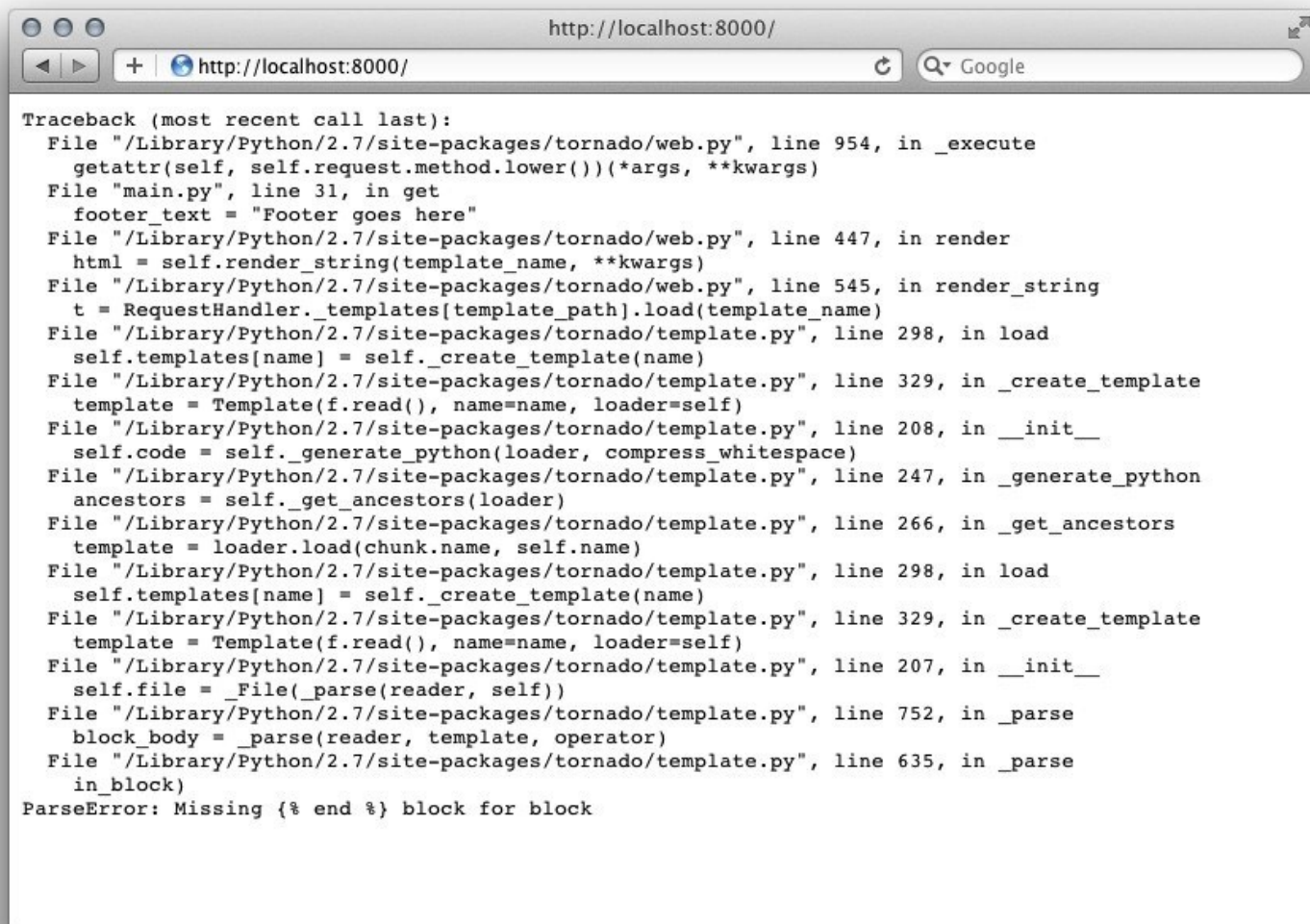


图 3-2 块基础

你也可以保留父模板块语句中的默认文本和标记，就像扩展模板没有指定它自己的块版本一样被渲染。这种情况下，你可以根据某页的情况只替换必须的东西，这在包含或替换脚本、CSS 文件和标记块时非常有用。

正如模板文档所记录的，“错误报告目前...呃...是非常有意思的”。一个语法错误或者没有闭合的 `{% block %}` 语句可以使得浏览器直接显示 500: Internal Server Error（如果你运行在 *debug* 模式下会引发完整的 Python 堆栈跟踪）。如图 3-3 所示。

总之，为了你自己好的话，你需要使自己的模板尽可能的鲁棒，并且在模板被渲染之前发现错误。



```
Traceback (most recent call last):
  File "/Library/Python/2.7/site-packages/tornado/web.py", line 954, in _execute
    getattr(self, self.request.method.lower())(*args, **kwargs)
  File "main.py", line 31, in get
    footer_text = "Footer goes here"
  File "/Library/Python/2.7/site-packages/tornado/web.py", line 447, in render
    html = self.render_string(template_name, **kwargs)
  File "/Library/Python/2.7/site-packages/tornado/web.py", line 545, in render_string
    t = RequestHandler._templates[template_path].load(template_name)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 298, in load
    self.templates[name] = self._create_template(name)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 329, in _create_template
    template = Template(f.read(), name=name, loader=self)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 208, in __init__
    self.code = self._generate_python(loader, compress_whitespace)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 247, in _generate_python
    ancestors = self._get_ancestors(loader)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 266, in _get_ancestors
    template = loader.load(chunk.name, self.name)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 298, in load
    self.templates[name] = self._create_template(name)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 329, in _create_template
    template = Template(f.read(), name=name, loader=self)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 207, in __init__
    self.file = _File(_parse(reader, self))
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 752, in _parse
    block_body = _parse(reader, template, operator)
  File "/Library/Python/2.7/site-packages/tornado/template.py", line 635, in _parse
    in_block)
ParseError: Missing {% end %} block for block
```

图 3-3 块错误

### 3.1.2 模板练习：Burt's Book

所以，你会认为这听起来很有趣，但却不能描绘出在一个标准的 Web 应用中如何使用？那么让我们在这里看一个例子，我们的朋友 Burt 希望运行一个名叫 Burt's Books 的书店。

Burt 通过他的书店卖很多书，他的网站会展示很多不同的内容，比如新品推荐、商店信息等等。Burt 希望有一个固定的外观和感觉的网站，同时也能更简单的更新页面和段落。

为了做到这些，Burt's Book 使用了以 Tornado 为基础的网站，其中包括一个拥有样式、布局和 header/footer 细节的主模版，以及一个处理页面的轻量级的子模板。在这个系统中，Burt 可以把最新发布、员工推荐、即将发行等不同页面编写在一起，共同使用通用的基础属性。

Burt's Book 的网站使用一个叫作 main.html 的主要基础模板，用来包含网站的通用架构，如下面的代码所示：

```
<html>
<head>
  <title>{{ page_title }}</title>
  <link rel="stylesheet" href="{{ static_url("css/style.css") }}" />
```

```

</head>
<body>
    <div id="container">
        <header>
            {% block header %}<h1>Burt's Books</h1>{% end %}
        </header>
        <div id="main">
            <div id="content">
                {% block body %} {% end %}
            </div>
        </div>
        <footer>
            {% block footer %}
                <p>
For more information about our selection, hours or events, please email us at
<a href="mailto:contact@burtsbooks.com">contact@burtsbooks.com</a>.
                </p>
            {% end %}
        </footer>
    </div>
    <script src="{{ static_url("js/script.js") }}"></script>
</body>
</html>

```

这个页面定义了结构，应用了一个 CSS 样式表，并加载了主要的 JavaScript 文件。其他模板可以扩展它，在必要时替换 header、body 和 footer 块。

这个网站的 index 页（index.html）欢迎友好的网站访问者并提供一些商店的信息。通过扩展 main.html，这个文件只需要包括用于替换默认文本的 header 和 body 块的信息。

```

{% extends "main.html" %}

{% block header %}
    <h1>{{ header_text }}</h1>
{% end %}

{% block body %}
    <div id="hello">
        <p>Welcome to Burt's Books!</p>
        <p>...</p>
    </div>
{% end %}

```

在 footer 块中，这个文件使用了 Tornado 模板的默认行为，继承了来自父模板的联系信息。

为了运作网站，传递信息给 index 模板，下面给出 Burt's Book 的 Python 脚本（main.py）：

```

import tornado.web
import tornado.httpserver

```

```

import tornado.ioloop
import tornado.options
import os.path

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [
            (r"/", MainHandler),
        ]
        settings = dict(
            template_path=os.path.join(os.path.dirname(__file__), "templates"),
            static_path=os.path.join(os.path.dirname(__file__), "static"),
            debug=True,
        )
        tornado.web.Application.__init__(self, handlers, **settings)

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.render(
            "index.html",
            page_title = "Burt's Books | Home",
            header_text = "Welcome to Burt's Books!",
        )

if __name__ == "__main__":
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(Application())
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

这个例子的结构和我们之前见到的不太一样，但你一点都不需要害怕。我们不再像之前那样通过使用一个处理类列表和一些其他关键字参数调用 *tornado.web.Application* 的构造函数来创建实例，而是定义了我们自己的 *Application* 子类，在这里我们简单地称之为 *Application*。在我们定义的 *\_\_init\_\_* 方法中，我们创建了处理类列表以及一个设置的字典，然后在初始化子类的调用中传递这些值，就像下面的代码一样：

```
tornado.web.Application.__init__(self, handlers, **settings)
```

所以在这个系统中，Burt's Book 可以很容易地改变 index 页面并保持基础模板在其他页面被使用时完好。此外，他们可以充分利用 Tornado 的真实能量，由 Python 脚本和/或数据库提供动态内容。我们将在之后看到更多相关的内容。

### 3.1.3 自动转义



Tornado 默认会自动转义模板中的内容，把标签转换为相应的 HTML 实体。这样可以防止后端为数据库的网站被 恶意脚本攻击。比如，你的网站中有一个评论部分，用户可以在这里添加任何他们想说的文字进行讨论。虽然一些 HTML 标签在标记和样式冲突时不构成重大威胁（如评论中没有闭 `<h1>` 标签），但 `<script>` 标签会允许攻击者加载其他的 JavaScript 文件，打开通向跨站脚本攻击、XSS 或漏洞之门。

让我们考虑 Burt's Book 网站上的一个用户反馈页面。Melvin，今天感觉特别邪恶，在评论里提交了下面的文字：

```
Totally hacked your site lulz <script>alert('RUNNING EVIL H4CKS AND SPL01TS  
NOW...')</script>
```

当我们在没有转义用户内容的情况下给一个不知情的用户构建页面时，脚本标签被作为一个 HTML 元素解释，并被浏览器执行，所以 Alice 看到了如图 3-4 所示的提示窗口。幸亏 Tornado 会自动转义在双大括号间被渲染的表达式。更早地转义 Melvin 输入的文本不会激活 HTML 标签，并且会渲染为下面的字符串：

```
Totally hacked your site lulz &lt;script&gt;alert('RUNNING EVIL H4CKS AND SPL01TS  
NOW...')&lt;/script&gt;
```

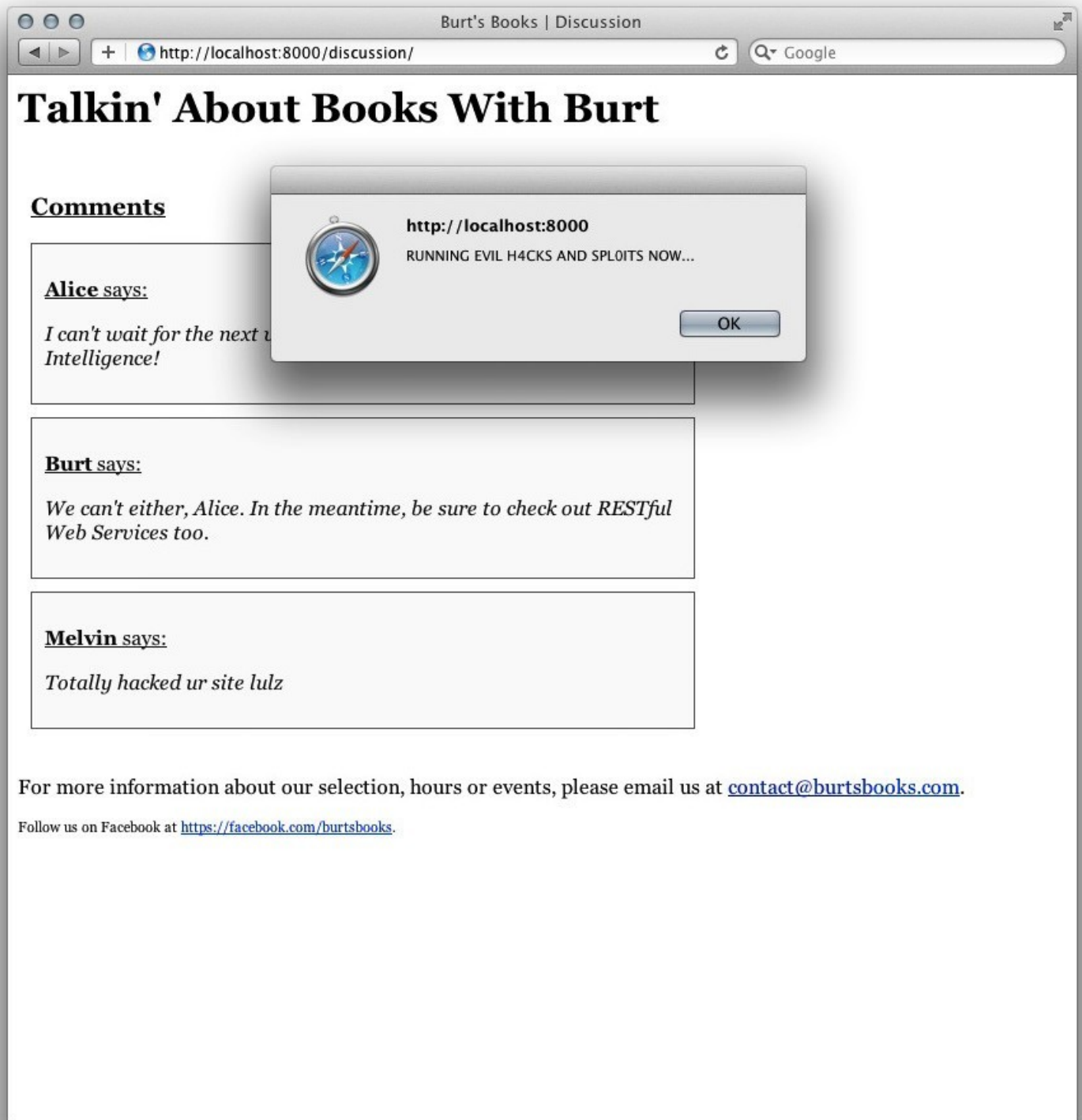


图 3-4 网站漏洞问题

现在当 Alice 访问网站时，没有恶意脚本被执行，所以她看到的页面如图 3-5 所示。

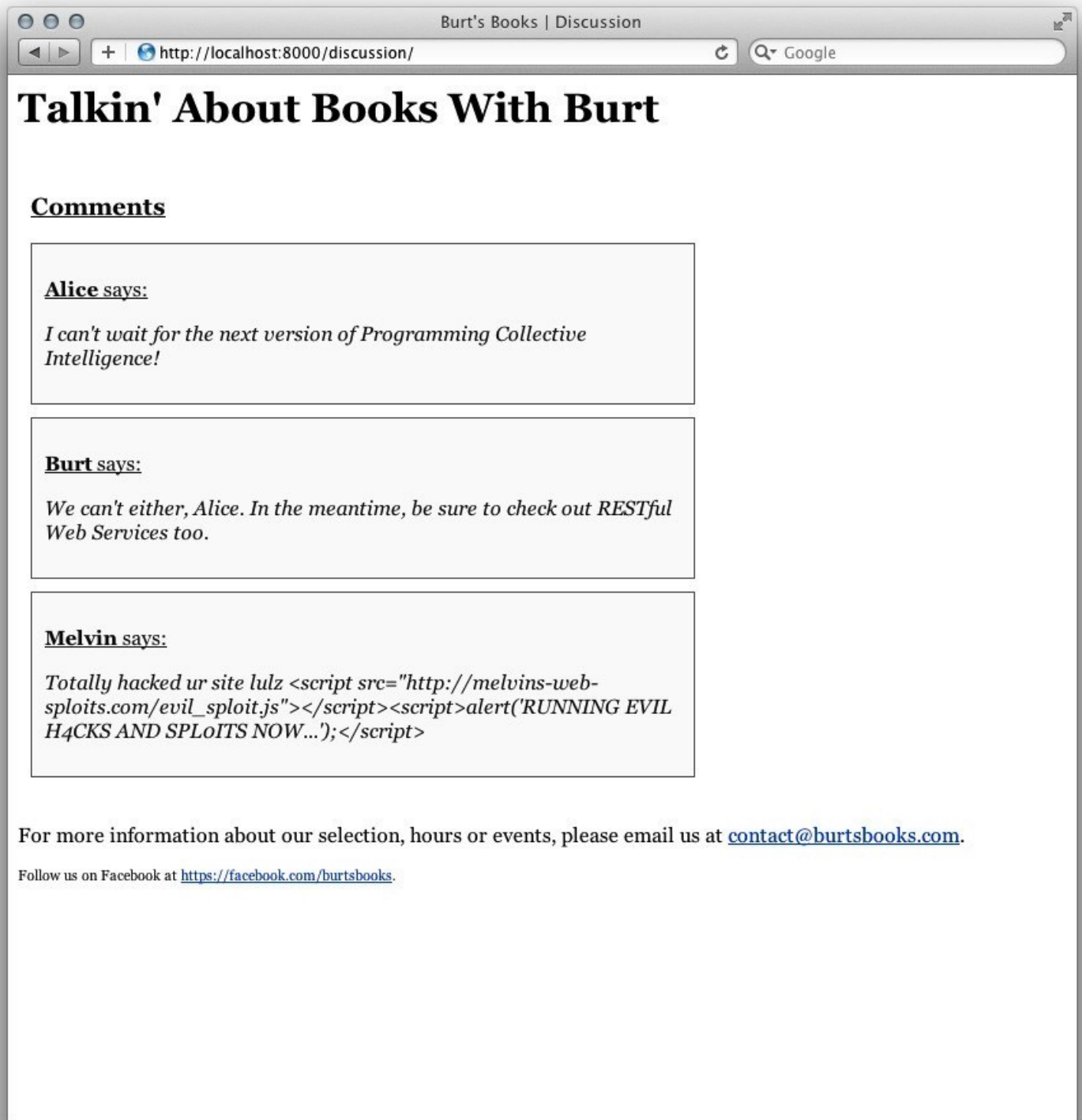


图 3-5 网站漏洞问题——解决

在 Tornado 1.x 版本中，模板没有被自动转义，所以我们之前谈论的防护措施需要显式地在未过滤的用户输入上调用 `escape()` 函数。

所以在这里，我们可以看到自动转义是如何防止你的访客进行恶意攻击的。然而，当通过模板和模块提供 HTML 动态内容时它仍会让你措手不及。

举个例子，如果 Burt 想在 footer 中使用模板变量设置 email 联系链接，他将不会得到期望的 HTML 链接。考虑下面的模板片段：

```
{% set mailLink = "<a href='mailto:contact@burtsbooks.com'>Contact Us</a>" %}
{{ mailLink }}
```

它会在页面源代码中渲染成如下代码：

```
<a href='mailto:contact@burtsbooks.com'>Contact Us</a>
```

此时自动转义被运行了，很明显，这无法让人们联系上 Burt。

为了处理这种情况，你可以禁用自动转义，一种方法是在 Application 构造函数中传递 *autoescape=None*，另一种方法是在每页的基础上修改自动转义行为，如下所示：

```
{% autoescape None %}
{{ mailLink }}
```

这些 *autoescape* 块不需要结束标签，并且可以设置 *xhtml\_escape* 来开启自动转义（默认行为），或 *None* 来关闭。

然而，在理想的情况下，你希望保持自动转义开启以便继续防护你的网站。因此，你可以使用 *{% raw %}* 指令来输出不转义的内容。

```
{% raw mailLink %}
```

需要特别注意的是，当你使用诸如 Tornado 的 *linkify()* 和 *xsrform\_html()* 函数时，自动转义的设置被改变了。所以如果你希望在前面代码的 footer 中使用 *linkify()* 来包含链接，你可以使用一个 *{% raw %}* 块：

```
{% block footer %}
    <p>
        For more information about our selection, hours or events, please email us at
        <a href='mailto:contact@burtsbooks.com'>contact@burtsbooks.com</a>.
    </p>

    <p class='small'>
        Follow us on Facebook at
        {% raw linkify('https://fb.me/burtsbooks', extra_params='ref=website') %}.
    </p>
{% end %}
```

这样，你可以既利用 *linkify()* 简记的好处，又可以保持在其他地方自动转义的好处。

## 3.2 UI 模块

正如前面我们所看到的，模板系统既轻量级又强大。在实践中，我们希望遵循软件工程的谚语，*Don't Repeat Yourself*。为了消除冗余的代码，我们可以使模板部分模块化。比如，展示物品列表的页面可以定位一个单独的模板用来渲染每个物品的标记。另外，一组共用通用导航结构的页面可以从一个共享的模块渲染内容。Tornado 的 UI 模块在这种情况下特别有用

UI 模块是封装模板中包含的标记、样式以及行为的可复用组件。它所定义的元素通常用于多个模板交叉复用或在同一个模板中重复使用。模块本身是一个继承自 Tornado 的 *UIModule* 类的简单 Python 类，并定义了一个 *render* 方法。当一个模板使用 `{% module Foo(...) %}` 标签引用一个模块时，Tornado 的模板引擎调用模块的 *render* 方法，然后返回一个字符串来替换模板中的模块标签。UI 模块也可以在渲染后的页面中嵌入自己的 JavaScript 和 CSS 文件，或指定额外包含的 JavaScript 或 CSS 文件。你可以定义可选的 *embedded\_javascript*、*embedded\_css*、*javascript\_files* 和 *css\_files* 方法来实现这一方法。

### 3.2.1 基础模块使用

为了在你的模板中引用模块，你必须在应用的设置中声明它。*ui\_modules* 参数期望一个模块名为键、类为值的字典输入来渲染它们。考虑代码清单 3-1。

代码清单 3-1 模块基础: `hello_module.py`

```
import tornado.web
import tornado.httpserver
import tornado.ioloop
import tornado.options
import os.path

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class HelloHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('hello.html')

class HelloModule(tornado.web.UIModule):
    def render(self):
        return '<h1>Hello, world!</h1>'

if __name__ == '__main__':
    tornado.options.parse_command_line()
    app = tornado.web.Application(
        handlers=[(r'/', HelloHandler)],
        template_path=os.path.join(os.path.dirname(__file__), 'templates'),
        ui_modules={'Hello': HelloModule}
    )
    server = tornado.httpserver.HTTPServer(app)
    server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

这个例子中 *ui\_module* 字典里只有一项，它把名为 *Hello* 的模块的引用和我们定义的 *HelloModule* 类结合了起来。

现在，当调用 *HelloHandler* 并渲染 `hello.html` 时，我们可以使用 `{% module Hello() %}` 模板标签来包含 *HelloModule* 类中 *render* 方法返回的字符串。

```

<html>
  <head><title>UI Module Example</title></head>
  <body>
    {% module Hello() %}
  </body>
</html>

```

这个 `hello.html` 模板通过在模块标签自身的位置调用 `HelloModule` 返回的字符串进行填充。下一节的例子将会展示如何扩展 UI 模块来渲染它们自己的模板并包含脚本和样式表。

### 3.2.2 模块深入

很多时候，一个非常有用的做法是让模块指向一个模板文件而不是在模块类中直接渲染字符串。这些模板的标记看起来就像我们已经看到过的作为整体的模板。

UI 模块的一个常见应用是迭代数据库或 API 查询中获得的结果，为每个独立项目的数据渲染相同的标记。比如，Burt 想在 Burt's Book 里创建一个推荐阅读部分，他已经创建了一个名为 `recommended.html` 的模板，其代码如下所示。就像前面看到的那样，我们将使用 `{% module Book(book) %}` 标签调用模块。

```

{% extends "main.html" %}

{% block body %}
<h2>Recommended Reading</h2>
  {% for book in books %}
    {% module Book(book) %}
  {% end %}
{% end %}

```

Burt 还创建了一个叫作 `book.html` 的图书模块的模板，并把它放到了 `templates/modules` 目录下。一个简单的图书模板看起来像下面这样：

```

<div class="book">
  <h3 class="book_title">{{ book["title"] }}</h3>
  
</div>

```

现在，当我们定义 `BookModule` 类的时候，我们将调用继承自 `UIModule` 的 `render_string` 方法。这个方法显式地渲染模板文件，当我们返回给调用者时将其关键字参数作为一个字符串。

```

class BookModule(tornado.web.UIModule):
    def render(self, book):
        return self.render_string('modules/book.html', book=book)

```

在完整的例子中，我们将使用下面的模板来格式化每个推荐书籍的所有属性，代替先前的 `book.html`

```

<div class="book">
  <h3 class="book_title">{{ book["title"] }}</h3>

```

```

{% if book["subtitle"] != "" %}
    <h4 class="book_subtitle">{{ book["subtitle"] }}</h4>
{% end %}

<div class="book_details">
    <div class="book_date_released">Released: {{ book["date_released"]}}</div>
    <div class="book_date_added">
        Added: {{ locale.format_date(book["date_added"], relative=False) }}
    </div>
    <h5>Description:</h5>
    <div class="book_body">{% raw book["description"] %}</div>
</div>
</div>

```

使用这个布局，传递给 recommended.html 模板的 *books* 参数的每项都将会调用这个模块。每次使用一个新的 *book* 参数调用 *Book* 模块时，模块（以及 book.html 模板）可以引用 *book* 参数的字典中的项，并以适合的方式格式化数据（如图 3-6）。



Burt's Books | Recommended Reading

← → +

http://localhost:8000/recommended/

↻

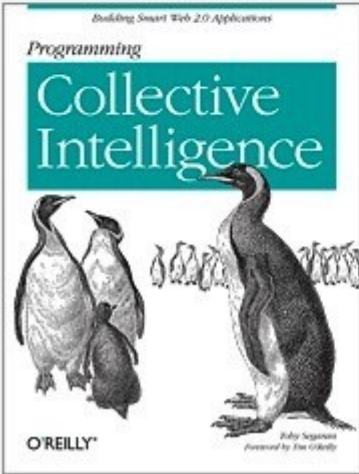
Q Google

# Burt's Books

## Recommended Reading

### Programming Collective Intelligence

#### Building Smart Web 2.0 Applications



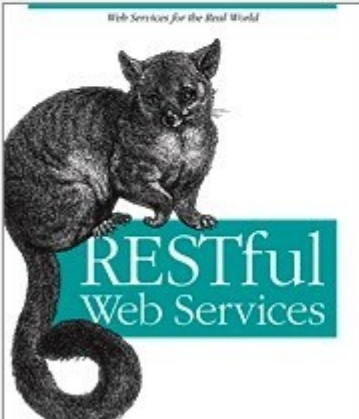
Released: August 2007  
Added: July 9 at 9:47 pm

**Description:**

This fascinating book demonstrates how you can build web applications to mine the enormous amount of data created by people on the Internet. With the sophisticated algorithms in this book, you can write smart programs to access interesting datasets from other web sites, collect data from users of your own applications, and analyze and understand the data once you've found it.

### RESTful Web Services

#### Web services for the real world



Released: May 2007  
Added: July 20 at 7:47 am

**Description:**

You've built web sites that can be used by humans. But can you also build web sites that are usable by machines? That's where the future lies, and that's what this book shows you how to do. Today's web service technologies have lost sight of

图 3-6 包含样式数据的图书模块

现在，我们可以定义一个 *RecommendedHandler* 类来渲染模板，就像你通常的操作那样。这个模板可以在渲染推荐书籍列表时引用 *Book* 模块。

```
class RecommendedHandler(tornado.web.RequestHandler):
    def get(self):
        self.render(
            "recommended.html",
            page_title="Burt's Books | Recommended Reading",
            header_text="Recommended Reading",
            books=[
                {
                    "title": "Programming Collective Intelligence",
                    "subtitle": "Building Smart Web 2.0 Applications",
                    "image": "/static/images/collective_intelligence.gif",
                    "author": "Toby Segaran",
                    "date_added": 1310248056,
                    "date_released": "August 2007",
                    "isbn": "978-0-596-52932-1",
                    "description": "<p>This fascinating book demonstrates how you "
                                "can build web applications to mine the enormous amount of data
                                created by people "
                                "on the Internet. With the sophisticated algorithms in this
                                book, you can write "
                                "smart programs to access interesting datasets from other web
                                sites, collect data "
                                "from users of your own applications, and analyze and
                                understand the data once "
                                "you've found it.</p>"
                },
                ...
            ]
        )
```

如果要用更多的模块，只需要简单地在 *ui\_modules* 参数中添加映射值。因为模板可以指向任何定义在 *ui\_modules* 字典中的模块，所以在自己的模块中指定功能非常容易。

在这个例子中，你可能已经注意到了 *locale.format\_date()* 的使用。它调用了 *tornado.locale* 模块提供的日期处理方法，这个模块本身是一组 *i18n* 方法的集合。*format\_date()* 选项默认格式化 GMT Unix 时间戳为 *XX time ago*，并且可以向下面这样使用：

```
{{ locale.format_date(book["date"]) }}
```

*relative=False* 将使其返回一个绝对时间（包含小时和分钟），而 *full\_format=True* 选项将会展示一个包含月、日、年和时间的完整日期（比如，July 9, 2011 at 9:47 pm），当搭配 *shorter=True* 使用时可以隐藏时间，只显示月、日和年。

这个模块在你处理时间和日期时非常有用，并且还提供了处理本地化字符串的支持。

### 3.2.3 嵌入 JavaScript 和 CSS

为了给这些模块提供更高的灵活性，Tornado 允许你使用 *embedded\_css* 和 *embedded\_javascript* 方法嵌入其他的 CSS 和 JavaScript 文件。举个例子，如果你想在调用模块时给 DOM 添加一行文字，你可以通过从模块中嵌入 JavaScript 来做到：

```
class BookModule(tornado.web.UIModule):
    def render(self, book):
        return self.render_string(
            "modules/book.html",
            book=book,
        )

    def embedded_javascript(self):
        return "document.write(\"hi!\")"
```

当调用模块时，*document.write("hi!")* 将被 `<script>` 包围，并被插入到 `<body>` 的闭标签中：

```
<script type="text/javascript">
//
document.write("hi!")
//]]&gt;
&lt;/script&gt;</pre></div><div data-bbox="57 495 937 531" data-label="Text"><p>显然，只是在文档主体中写这些内容并不是世界上最有用的事情，而我们还有另一个给予你极大灵活性的选项，当创建这些模块时，可以在每个模块中包含 JavaScript 文件。</p></div><div data-bbox="57 549 714 566" data-label="Text"><p>类似的，你也可以把只在这些模块被调用时加载的额外的 CSS 规则放进来：</p></div><div data-bbox="57 587 514 622" data-label="Text"><pre>def embedded_css(self):
    return ".book {background-color:#F5F5F5}"</pre></div><div data-bbox="57 637 937 673" data-label="Text"><p>在这种情况下，<i>.book {background-color:#555}</i> 这条 CSS 规则被包裹在 <code>&lt;style&gt;</code> 中，并被直接添加到 <code>&lt;head&gt;</code> 的闭标签之前。</p></div><div data-bbox="57 693 381 747" data-label="Text"><pre>&lt;style type="text/css"&gt;
.book {background-color:#F5F5F5}
&lt;/style&gt;</pre></div><div data-bbox="57 763 932 798" data-label="Text"><p>更加灵活的是，你甚至可以简单地使用 <i>html_body()</i> 来在闭合的 <code>&lt;/body&gt;</code> 标签前添加完整的 HTML 标记：</p></div><div data-bbox="57 818 604 854" data-label="Text"><pre>def html_body(self):
    return "&lt;script&gt;document.write(\"Hello!\")&lt;/script&gt;"</pre></div><div data-bbox="57 869 923 925" data-label="Text"><p>显然，虽然直接内嵌添加脚本和样式表很有用，但是为了更严谨的包含（以及更整洁的代码！），添加样式表和脚本文件会显得更好。他们的工作方式基本相同，所以你可以使用 <i>javascript_files()</i> 和 <i>css_files()</i> 来包含完整的文件，不论是本地的还是外部的。</p></div><div data-bbox="486 953 509 967" data-label="Page-Footer"><p>39</p></div>
```

比如，你可以添加一个额外的本地 CSS 文件如下：

```
def css_files(self):
    return "/static/css/newreleases.css"
```

或者你可以取得一个外部的 JavaScript 文件：

```
def javascript_files(self):
    return "https://ajax.googleapis.com/ajax/libs/jqueryui/1.8.14/jquery-ui.min.js"
```

当一个模块需要额外的库而应用的其他地方不是必需的时候，这种方式非常有用。比如，你有一个使用 JQuery UI 库的模块（而在应用的其他地方都不会被使用），你可以只在这个样本模块中加载 jquery-ui.min.js 文件，减少那些不需要它的页面的加载时间。

因为模块的内嵌 JavaScript 和内嵌 HTML 函数的目标都是紧邻 `</body>` 标签，`html_body()`、`javascript_files()` 和 `embedded_javascript()` 都会将内容渲染后插到页面底部，那么它们出现的顺序正好是你指定它们的顺序的倒序。

如果你有一个模块如下面的代码所示：

```
class SampleModule(tornado.web.UIModule):
    def render(self, sample):
        return self.render_string(
            "modules/sample.html",
            sample=sample
        )

    def html_body(self):
        return "<div class=\"addition\"><p>html_body()</p></div>"

    def embedded_javascript(self):
        return "document.write(\"<p>embedded_javascript()</p>\")"

    def embedded_css(self):
        return ".addition {color: #A1CAF1}"

    def css_files(self):
        return "/static/css/sample.css"

    def javascript_files(self):
        return "/static/js/sample.js"
```

`html_body()` 最先被编写，它紧挨着出现在 `</body>` 标签的上面。`embedded_javascript()` 接着被渲染，最后是 `javascript_files()`。你可以在图 3-7 中看到它是如何工作的。

需要小心的是，你不能包括一个需要其他地方东西的方法（比如依赖其他文件的 JavaScript 函数），因为此时他们可能会按照和你期望不同的顺序进行渲染。

总之，模块允许你在模板中渲染格式化数据时非常灵活，同时也让你能够只在调用模块时包含指定的一些额外的样式和函数规则。

### 3.3 总结¶

正如我们之前看到的，Tornado 使扩展模板更容易，以便你的网站代码可以在整个应用中轻松复用。而使用模块后，你可以在什么文件、样式和脚本动作需要被包括进来这个问题上拥有更细粒度的决策。然而，我们的例子依赖于使用 Python 原生数据结构时是否简单，在你的实际应用中硬编码大数据结构的感觉可不好。下一步，我们将看到如何配合持久化存储来处理存储、提供和编辑动态内容。

## 第四章：数据库¶

在本章中，我们将给出几个使用数据库的 Tornado Web 应用的例子。我们将从一个简单的 RESTful API 例子起步，然后创建 3.1.2 节中的 Burt's Book 网站的完整功能版本。

本章中的例子使用 MongoDB 作为数据库，并通过 pymongo 作为驱动来连接 MongoDB。当然，还有很多数据库系统可以用在 Web 应用中：Redis、CouchDB 和 MySQL 都是一些知名的选择，并且 Tornado 自带处理 MySQL 请求的库。我们选择使用 MongoDB 是因为它的简单性和便捷性：安装简单，并且能够和 Python 代码很好地融合。它结构自然，预定义数据结构不是必需的，很适合原型开发。

在本章中，我们假设你已经在机器上安装了 MongoDB，能够运行示例代码，不过也可以在远程服务器上使用 MongoDB，相关的代码调整也很容易。如果你不想在你的机器上安装 MongoDB，或者没有一个适合你操作系统的 MongoDB 版本，你也可以选择一些 MongoDB 主机服务。我们推荐使用 [MongoHQ](#)。在我们最初的例子中，假设你已经在你的机器上运行了 MongoDB，但使用远程服务器（包括 MongoHQ）运行的 MongoDB 时，调整代码也很简单。

我们同样还假设你已经有一些数据库的经验了，尽管并不一定是特定的 MongoDB 数据库的经验。当然，我们只会使用 MongoDB 的一点皮毛；如果想获得更多信息请查阅 MongoDB 文档（<http://www.mongodb.org/display/DOCS/Home>）让我们开始吧！

### 4.1 使用 PyMongo 进行 MongoDB 基础操作¶

在我们使用 MongoDB 编写 Web 应用之前，我们需要了解如何在 Python 中使用 MongoDB。在这一节，你将学会如何使用 PyMongo 连接 MongoDB 数据库，然后学习如何使用 pymongo 在 MongoDB 集合中创建、取出和更新文档。

PyMongo 是一个简单的包装 MongoDB 客户端 API 的 Python 库。你可以在 <http://api.mongodb.org/python/current/> 下载获得。一旦你安装完成，打开一个 Python 解释器，然后跟随下面的步骤。

#### 4.1.1 创建连接¶

首先，你需要导入 PyMongo 库，并创建一个到 MongoDB 数据库的连接。

```
>>> import pymongo
>>> conn = pymongo.Connection("localhost", 27017)
```

前面的代码向我们展示了如何连接运行在你本地机器上默认端口（27017）上的 MongoDB 服务器。如果你正在使用一个远程 MongoDB 服务器，替换 *localhost* 和 *27017* 为合适的值。你也可以使用 MongoDB URI 来连接 MongoDB，就像下面这样：

```
>>> conn = pymongo.Connection(
... "mongodb://user:password@staff.mongohq.com:10066/your_mongohq_db")
```

前面的代码将连接 MongoHQ 主机上的一个名为 *your\_mongohq\_db* 的数据库，其中 *user* 为用户名，*password* 为密码。你可以在 <http://www.mongodb.org/display/DOCS/Connections> 中了解更多关于 MongoDB URI 的信息。

一个 MongoDB 服务器可以包括任意数量的数据库，而 *Connection* 对象可以让你访问你连接的服务器的任何一个数据库。你可以通过对象属性或像字典一样使用对象来获得代表一个特定数据库的对象。如果数据库不存在，则被自动建立。

```
>>> db = conn.example or: db = conn['example']
```

一个数据库可以拥有任意多个集合。一个集合就是放置一些相关文档的地方。我们使用 MongoDB 执行的大部分操作（查找文档、保存文档、删除文档）都是在一个集合对象上执行的。你可以在数据库对象上调用 *collection\_names* 方法获得数据库中的集合列表。

```
>>> db.collection_names()
[]
```

当然，我们还没有在我们的数据库中添加任何集合，所以这个列表是空的。当我们插入第一个文档时，MongoDB 会自动创建集合。你可以在数据库对象上通过访问集合名字的属性来获得代表集合的对象，然后调用对象的 *insert* 方法指定一个 Python 字典来插入文档。比如，在下面的代码中，我们在集合 *widgets* 中插入了一个文档。因为 *widgets* 集合并不存在，MongoDB 会在文档被添加时自动创建。

```
>>> widgets = db.widgets or: widgets = db['widgets'] (see below)
>>> widgets.insert({"foo": "bar"})
ObjectId('4eada0b5136fc4aa41000000')
>>> db.collection_names()
[u'widgets', u'system.indexes']
```

（*system.indexes* 集合是 MongoDB 内部使用的。处于本章的目的，你可以忽略它。）

在之前展示的代码中，你既可以使用数据库对象的属性访问集合，也可以把数据库对象看作一个字典然后把集合名称作为键来访问。比如，如果 *db* 是一个 pymongo 数据库对象，那么 *db.widgets* 和 *db['widgets']* 同样都可以访问这个集合。

## 4.1.2 处理文档

MongoDB 以文档的形式存储数据，这种形式有着相对自由的数据结构。MongoDB 是一个“无模式”数据库：同一个集合中的文档通常拥有相同的结构，但是 MongoDB 中并不强制要求使用相同结构。在内部，MongoDB 以一种称为 BSON 的类似 JSON 的二进制形式存储文档。PyMongo 允许我们以 Python 字典的形式写和取出文档。

为了在集合中 创建一个新的文档，我们可以使用字典作为参数调用文档的 *insert* 方法。

```
>>> widgets.insert({"name": "flibnip", "description": "grade-A industrial flibnip",
"quantity": 3})
ObjectId('4eada3a4136fc4aa41000001')
```

既然文档在数据库中，我们可以使用集合对象的 *find\_one* 方法来取出文档。你可以通过传递一个键为文档名、值为你想要匹配的表达式字典来告诉 *find\_one* 找到一个特定的文档。比如，我们想要返回文档名域 *name* 的值等于 *flibnip* 的文档（即，我们刚刚创建的文档），可以像下面这样调用 *find\_one* 方法：

```
>>> widgets.find_one({"name": "flibnip"})
{'description': u'grade-A industrial flibnip',
 u'_id': ObjectId('4eada3a4136fc4aa41000001'),
 u'name': u'flibnip', u'quantity': 3}
```

请注意 *\_id* 域。当你创建任何文档时，MongoDB 都会自动添加这个域。它的值是一个 *ObjectID*，一种保证文档唯一的 BSON 对象。你可能已经注意到，当我们使用 *insert* 方法成功创建一个新的文档时，这个 *ObjectID* 同样被返回了。（当你创建文档时，可以通过给 *\_id* 键赋值来覆写自动创建的 *ObjectID* 值。）

*find\_one* 方法返回的值是一个简单的 Python 字典。你可以从中访问独立的项，迭代它的键值对，或者就像使用其他 Python 字典那样修改值。

```
>>> doc = db.widgets.find_one({"name": "flibnip"})
>>> type(doc)
<type 'dict'>
>>> print doc['name']
flibnip
>>> doc['quantity'] = 4
```

然而，字典的改变并不会自动保存到数据库中。如果你希望把字典的改变保存，需要调用集合的 *save* 方法，并将修改后的字典作为参数进行传递：

```
>>> doc['quantity'] = 4
>>> db.widgets.save(doc)
>>> db.widgets.find_one({"name": "flibnip"})
{'_id': ObjectId('4eb12f37136fc4b59d000000'),
 u'description': u'grade-A industrial flibnip',
 u'quantity': 4, u'name': u'flibnip'}
```

让我们在集合中添加更多的文档：

```
>>> widgets.insert({"name": "smorkeg", "description": "for external use only",
"quantity": 4})
ObjectId('4eadaa5c136fc4aa41000002')
>>> widgets.insert({"name": "clobbasker", "description": "properties available on
request", "quantity": 2})
ObjectId('4eadad79136fc4aa41000003')
```



我们可以通过调用集合的 *find* 方法来获得集合中所有文档的列表，然后迭代其结果：

```
>>> for doc in widgets.find():
...     print doc
...
{'u'_id': ObjectId('4eada0b5136fc4aa41000000'), u'foo': u'bar'}
{'u'description': u'grade-A industrial flibnip',
 u'_id': ObjectId('4eada3a4136fc4aa41000001'),
 u'name': u'flibnip', u'quantity': 4}
{'u'description': u'for external use only',
 u'_id': ObjectId('4eadaa5c136fc4aa41000002'),
 u'name': u'smorkeg', u'quantity': 4}
{'u'description': u'properties available on request',
 u'_id': ObjectId('4eadad79136fc4aa41000003'),
 u'name': u'clobbasker',
 u'quantity': 2}
```

如果我们希望获得文档的一个子集，我们可以在 *find* 方法中传递一个字典参数，就像我们在 *find\_one* 中那样。比如，找到那些 *quantity* 键的值为 4 的集合：

```
>>> for doc in widgets.find({"quantity": 4}):
...     print doc
...
{'u'description': u'grade-A industrial flibnip',
 u'_id': ObjectId('4eada3a4136fc4aa41000001'),
 u'name': u'flibnip', u'quantity': 4}
{'u'description': u'for external use only',
 u'_id': ObjectId('4eadaa5c136fc4aa41000002'),
 u'name': u'smorkeg',
 u'quantity': 4}
```

最后，我们可以使用集合的 *remove* 方法从集合中删除一个文档。*remove* 方法和 *find*、*find\_one* 一样，也可以使用一个字典参数来指定哪个文档需要被删除。比如，要删除所有 *name* 键的值为 *flibnip* 的文档，输入：

```
>>> widgets.remove({"name": "flibnip"})
```

列出集合中的所有文档来确认上面的文档已经被删除：

```
>>> for doc in widgets.find():
...     print doc
...
{'u'_id': ObjectId('4eada0b5136fc4aa41000000'),
 u'foo': u'bar'}
{'u'description': u'for external use only',
 u'_id': ObjectId('4eadaa5c136fc4aa41000002'),
 u'name': u'smorkeg', u'quantity': 4}
{'u'description': u'properties available on request',
 u'_id': ObjectId('4eadad79136fc4aa41000003'),
 u'name': u'clobbasker',
```

```
u'quantity': 2}
```

### 4.1.3 MongoDB 文档和 JSON

使用 Web 应用时，你经常会想采用 Python 字典并将其序列化为一个 JSON 对象（比如，作为一个 AJAX 请求的响应）。由于你使用 PyMongo 从 MongoDB 中取出的文档是一个简单的字典，你可能会认为你可以使用 *json* 模块的 *dumps* 函数就可以简单地将其转换为 JSON。但，这还有一个障碍：

```
>>> doc = db.widgets.find_one({"name": "flibnip"})
>>> import json
>>> json.dumps(doc)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    [stack trace omitted]
TypeError: ObjectId('4eb12f37136fc4b59d000000') is not JSON serializable
```

这里的问题是 Python 的 *json* 模块并不知道如何转换 MongoDB 的 *ObjectId* 类型到 JSON。有很多方法可以处理这个问题。其中最简单的方法（也是我们在本章中采用的方法）是在我们序列化之前从字典里简单地删除 *\_id* 键。

```
>>> del doc["_id"]
>>> json.dumps(doc)
'{"description": "grade-A industrial flibnip", "quantity": 4, "name": "flibnip"}
```

一个更复杂的方法是使用 PyMongo 的 *json\_util* 库，它同样可以帮你序列化其他 MongoDB 特定数据类型到 JSON。我们可以在 [http://api.mongodb.org/python/current/api/bson/json\\_util.html](http://api.mongodb.org/python/current/api/bson/json_util.html) 了解更多关于这个库的信息。

## 4.2 一个简单的持久化 Web 服务

现在我们知道编写一个 Web 服务，可以访问 MongoDB 数据库中的数据。首先，我们要编写一个只从 MongoDB 读取数据的 Web 服务。然后，我们写一个可以读写数据的服务。

### 4.2.1 只读字典

我们将要创建的应用是一个基于 Web 的简单字典。你发送一个指定单词的请求，然后返回这个单词的定义。一个典型的交互看起来是下面这样的：

```
$ curl http://localhost:8000/oarlock
{definition: "A device attached to a rowboat to hold the oars in place",
"word": "oarlock"}
```

这个 Web 服务将从 MongoDB 数据库中取得数据。具体来说，我们将根据 *word* 属性查询文档。在我们查看 Web 应用本身的源码之前，先让我们从 Python 解释器中向数据库添加一些单词。

```
>>> import pymongo
>>> conn = pymongo.Connection("localhost", 27017)
```

```
>>> db = conn.example
>>> db.words.insert({"word": "oarlock", "definition": "A device attached to a rowboat
to hold the oars in place"})
ObjectId('4eb1d1f8136fc4be90000000')
>>> db.words.insert({"word": "seminomadic", "definition": "Only partial
ly nomadic"})
ObjectId('4eb1d356136fc4be900000001')
>>> db.words.insert({"word": "perturb", "definition": "Bother, unsettle
, modify"})
ObjectId('4eb1d39d136fc4be900000002')
```

代码清单 4-1 是我们这个词典 Web 服务的源码，在这个代码中我们查询刚才添加的单词然后使用其定义作为响应。

代码清单 4-1 一个词典 Web 服务：definitions\_readonly.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

import pymongo

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [(r"/(\w+)", WordHandler)]
        conn = pymongo.Connection("localhost", 27017)
        self.db = conn["example"]
        tornado.web.Application.__init__(self, handlers, debug=True)

class WordHandler(tornado.web.RequestHandler):
    def get(self, word):
        coll = self.application.db.words
        word_doc = coll.find_one({"word": word})
        if word_doc:
            del word_doc["_id"]
            self.write(word_doc)
        else:
            self.set_status(404)
            self.write({"error": "word not found"})

if __name__ == "__main__":
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(Application())
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()
```

在命令行中像下面这样运行这个程序：

```
$ python definitions_readonly.py
```

现在使用 curl 或者你的浏览器来向应用发送一个请求。

```
$ curl http://localhost:8000/perturb
{"definition": "Bother, unsettle, modify", "word": "perturb"}
```

如果我们请求一个数据库中没有添加的单词，会得到一个 404 错误以及一个错误信息：

```
$ curl http://localhost:8000/snorkle
{"error": "word not found"}
```

那么这个程序是如何工作的呢？让我们看看这个程序的主线。开始，我们在程序的最上面导入了 *import pymongo* 库。然后我们在我们的 *TornadoApplication* 对象的 *\_\_init\_\_* 方法中实例化了一个 pymongo 连接对象。我们在 *Application* 对象中创建了一个 *db* 属性，指向 MongoDB 的 *example* 数据库。下面是相关的代码：

```
conn = pymongo.Connection("localhost", 27017)
self.db = conn["example"]
```

一旦我们在 *Application* 对象中添加了 *db* 属性，我们就可以在任何 *RequestHandler* 对象中使用 *self.application.db* 访问它。实际上，这正是我们为了取出 pymongo 的 *words* 集合对象而在 *WordHandler* 中 *get* 方法所做的事情。

```
def get(self, word):
    coll = self.application.db.words
    word_doc = coll.find_one({"word": word})
    if word_doc:
        del word_doc["_id"]
        self.write(word_doc)
    else:
        self.set_status(404)
        self.write({"error": "word not found"})
```

在我们将集合对象指定给变量 *coll* 后，我们使用用户在 HTTP 路径中请求的单词调用 *find\_one* 方法。如果我们发现这个单词，则从字典中删除 *\_id* 键（以便 Python 的 *json* 库可以将其序列化），然后将其传递给 *RequestHandler* 的 *write* 方法。*write* 方法将会自动序列化字典为 JSON 格式。

如果 *find\_one* 方法没有匹配任何对象，则返回 *None*。在这种情况下，我们将响应状态设置为 404，并且写一个简短的 JSON 来提示用户这个单词在数据库中没有找到。

## 4.2.2 写字典

从字典里查询单词很有趣，但是在交互解释器中添加单词的过程却很麻烦。我们例子的下一步是使 HTTP 请求网站服务时能够创建和修改单词。

它的工作流程是：发出一个特定单词的 *POST* 请求，将根据请求中给出的定义修改已经存在的定义。如果这个单词并不存在，则创建它。例如，创建一个新的单词：

```
$ curl -d definition=a+leg+shirt http://localhost:8000/pants
{"definition": "a leg shirt", "word": "pants"}
```

我们可以使用一个 *GET* 请求来获得已创建单词的定义：

```
$ curl http://localhost:8000/pants
{"definition": "a leg shirt", "word": "pants"}
```

我们可以发出一个带有一个单词定义域的 *POST* 请求来修改一个已经存在的单词（就和我们创建一个新单词时使用的参数一样）：

```
$ curl -d definition=a+boat+wizard http://localhost:8000/oarlock
{"definition": "a boat wizard", "word": "oarlock"}
```

代码清单 4-2 是我们的词典 Web 服务的读写版本的源代码。

代码清单 4-2 一个读写字典服务：definitions\_readwrite.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web

import pymongo

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [(r"/(\w+)", WordHandler)]
        conn = pymongo.Connection("localhost", 27017)
        self.db = conn["definitions"]
        tornado.web.Application.__init__(self, handlers, debug=True)

class WordHandler(tornado.web.RequestHandler):
    def get(self, word):
        coll = self.application.db.words
        word_doc = coll.find_one({"word": word})
        if word_doc:
            del word_doc["_id"]
            self.write(word_doc)
        else:
            self.set_status(404)
    def post(self, word):
        definition = self.get_argument("definition")
        coll = self.application.db.words
        word_doc = coll.find_one({"word": word})
```

```

        if word_doc:
            word_doc['definition'] = definition
            coll.save(word_doc)
        else:
            word_doc = {'word': word, 'definition': definition}
            coll.insert(word_doc)
        del word_doc["_id"]
        self.write(word_doc)

if __name__ == "__main__":
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(Application())
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

除了在 *WordHandler* 中添加了一个 *post* 方法之外，这个源代码和只读服务的版本完全一样。让我们详细看看这个方法吧：

```

def post(self, word):
    definition = self.get_argument("definition")
    coll = self.application.db.words
    word_doc = coll.find_one({"word": word})
    if word_doc:
        word_doc['definition'] = definition
        coll.save(word_doc)
    else:
        word_doc = {'word': word, 'definition': definition}
        coll.insert(word_doc)
    del word_doc["_id"]
    self.write(word_doc)

```

我们首先做的事情是使用 *get\_argument* 方法取得 *POST* 请求中传递的 *definition* 参数。然后，就像在 *get* 方法一样，我们尝试使用 *find\_one* 方法从数据库中加载给定单词的文档。如果发现这个单词的文档，我们将 *definition* 条目的值设置为从 *POST* 参数中取得的值，然后调用集合对象的 *save* 方法将改变写到数据库中。如果没有发现文档，则创建一个新文档，并使用 *insert* 方法将其保存到数据库中。无论上述哪种情况，在数据库操作执行之后，我们在响应中写文档（注意首先要删掉 *\_id* 属性）。

## 4.3 Burt's Books

在[第三章](#)中，我们提出了 Burt's Book 作为使用 Tornado 模板工具构建复杂 Web 应用的例子。在本节中，我们将展示使用 MongoDB 作为数据存储的 Burt's Books 示例版本呢。

### 4.3.1 读取书籍（从数据库）

让我们从一些简单的版本开始：一个从数据库中读取书籍列表的 Burt's Books。首先，我们需要在我们的 MongoDB 服务器上创建一个数据库和一个集合，然后用书籍文档填充它，就像下面这样：

```

>>> import pymongo
>>> conn = pymongo.Connection()
>>> db = conn["bookstore"]
>>> db.books.insert({
...     "title": "Programming Collective Intelligence",
...     "subtitle": "Building Smart Web 2.0 Applications",
...     "image": "/static/images/collective_intelligence.gif",
...     "author": "Toby Segaran",
...     "date_added": 1310248056,
...     "date_released": "August 2007",
...     "isbn": "978-0-596-52932-1",
...     "description": "<p>[...]</p>"
... })
ObjectId('4eb6f1a6136fc42171000000')
>>> db.books.insert({
...     "title": "RESTful Web Services",
...     "subtitle": "Web services for the real world",
...     "image": "/static/images/restful_web_services.gif",
...     "author": "Leonard Richardson, Sam Ruby",
...     "date_added": 1311148056,
...     "date_released": "May 2007",
...     "isbn": "978-0-596-52926-0",
...     "description": "<p>[...]>/p>"
... })
ObjectId('4eb6f1cb136fc42171000001')

```

（我们为了节省空间已经忽略了这些书籍的详细描述。）一旦我们在数据库中有了这些文档，我们就准备好了。代码清单 4-3 展示了 Burt's Books Web 应用修改版本的源代码 `burts_books_db.py`。

代码清单 4-3 读取数据库：burts\_books\_db.py

```

import os.path
import tornado.locale
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
from tornado.options import define, options
import pymongo

define("port", default=8000, help="run on the given port", type=int)

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [
            (r"/", MainHandler),
            (r"/recommended/", RecommendedHandler),
        ]
        settings = dict(
            template_path=os.path.join(os.path.dirname(__file__), "templates"),

```

```

        static_path=os.path.join(os.path.dirname(__file__), "static"),
        ui_modules={"Book": BookModule},
        debug=True,
    )
    conn = pymongo.Connection("localhost", 27017)
    self.db = conn["bookstore"]
    tornado.web.Application.__init__(self, handlers, **settings)

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.render(
            "index.html",
            page_title = "Burt's Books | Home",
            header_text = "Welcome to Burt's Books!",
        )

class RecommendedHandler(tornado.web.RequestHandler):
    def get(self):
        coll = self.application.db.books
        books = coll.find()
        self.render(
            "recommended.html",
            page_title = "Burt's Books | Recommended Reading",
            header_text = "Recommended Reading",
            books = books
        )

class BookModule(tornado.web.UIModule):
    def render(self, book):
        return self.render_string(
            "modules/book.html",
            book=book,
        )
    def css_files(self):
        return "/static/css/recommended.css"
    def javascript_files(self):
        return "/static/js/recommended.js"

if __name__ == "__main__":
    tornado.options.parse_command_line()
    http_server = tornado.httpserver.HTTPServer(Application())
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

正如你看到的，这个程序和[第三章](#)中 Burt's Books Web 应用的原始版本几乎完全相同。它们之间只有两个不同点。其一，我们在我们的 *Application* 中添加了一个 *db* 属性来连接 MongoDB 服务器：

```

conn = pymongo.Connection("localhost", 27017)
self.db = conn["bookstore"]

```



其二，我们使用连接的 *find* 方法来从数据库中取得书籍文档的列表，然后在渲染 *recommended.html* 时将这个列表传递给 *RecommendedHandler* 的 *get* 方法。下面是相关的代码：

```
def get(self):
    coll = self.application.db.books
    books = coll.find()
    self.render(
        "recommended.html",
        page_title = "Burt's Books | Recommended Reading",
        header_text = "Recommended Reading",
        books = books
    )
```

此前，书籍列表是被硬编码在 *get* 方法中的。但是，因为我们在 MongoDB 中添加的文档和原始的硬编码字典拥有相同的域，所以我们之前写的模板代码并不需要修改。

像下面这样运行应用：

```
$ python burts_books_db.py
```

然后让你的浏览器指向 <http://localhost:8000/recommended/>。这次，页面和硬编码版本的 Burt's Books 看起来几乎一样（参见图 3-6）。

### 4.3.2 编辑和添加书籍

我们的下一步是添加一个接口用来编辑已经存在于数据库的书籍以及添加新书籍到数据库中。为此，我们需要一个让用户填写书籍信息的表单，一个服务表单的处理程序，以及一个处理表单结果并将其存入数据库的处理函数。

这个版本的 Burt's Books 和之前给出的代码几乎是一样的，只是增加了下面我们要讨论的一些内容。你可以跟随本书附带的完整代码阅读下面部分，相关的程序名为 *burts\_books\_rwdb.py*。

#### 4.3.2.1 渲染编辑表单

下面是 *BookEditHandler* 的源代码，它完成了两件事情：

1. *GET* 请求渲染一个显示已存在书籍数据的 HTML 表单（在模板 *book\_edit.html* 中）。
2. *POST* 请求从表单中取得数据，更新数据库中已存在的书籍记录或依赖提供的数据添加一个新的书籍。

下面是处理程序的源代码：

```
class BookEditHandler(tornado.web.RequestHandler):
    def get(self, isbn=None):
        book = dict()
        if isbn:
            coll = self.application.db.books
            book = coll.find_one({"isbn": isbn})
        self.render("book_edit.html",
```

```

        page_title="Burt's Books",
        header_text="Edit book",
        book=book)

def post(self, isbn=None):
    import time
    book_fields = ['isbn', 'title', 'subtitle', 'image', 'author',
                   'date_released', 'description']
    coll = self.application.db.books
    book = dict()
    if isbn:
        book = coll.find_one({"isbn": isbn})
    for key in book_fields:
        book[key] = self.get_argument(key, None)

    if isbn:
        coll.save(book)
    else:
        book['date_added'] = int(time.time())
        coll.insert(book)
    self.redirect("/recommended/")

```

我们将在稍后对其进行详细讲解，不过现在先让我们看看如何在 *Application* 类中建立请求到处理程序的路由。下面是 *Application* 的 `__init__` 方法的相关代码部分：

```

handlers = [
    (r"/", MainHandler),
    (r"/recommended/", RecommendedHandler),
    (r"/edit/([0-9Xx\|-]+)", BookEditHandler),
    (r"/add", BookEditHandler)
]

```

正如你所看到的，*BookEditHandler* 处理了两个不同路径模式的请求。其中一个 `/add`，提供不存在信息的编辑表单，因此你可以向数据库中添加一本新的书籍；另一个 `/edit/([0-9Xx\|-]+)`，根据书籍的 ISBN 渲染一个已存在书籍的表单。

#### 4.3.2.2 从数据库中取出书籍信息

让我们看看 *BookEditHandler* 的 `get` 方法是如何工作的：

```

def get(self, isbn=None):
    book = dict()
    if isbn:
        coll = self.application.db.books
        book = coll.find_one({"isbn": isbn})
    self.render("book_edit.html",
               page_title="Burt's Books",
               header_text="Edit book",
               book=book)

```

如果该方法作为到 `/add` 请求的结果被调用，Tornado 将调用一个没有第二个参数的 `get` 方法（因为路径中没有正则表达式的匹配组）。在这种情况下，默认将一个空的 `book` 字典传递给 `book_edit.html` 模板。

如果该方法作为到类似于 `/edit/0-123-456` 请求的结果被调用，那么 `isdb` 参数被设置为 `0-123-456`。在这种情况下，我们从 `Application` 实例中取得 `books` 集合，并用它查询 ISBN 匹配的书籍。然后我们传递结果 `book` 字典给模板。

下面是模板 (`book_edit.html`) 的代码：

```
{% extends "main.html" %}
{% autoescape None %}

{% block body %}
<form method="POST">
  ISBN <input type="text" name="isbn"
    value="{{ book.get('isbn', '') }}"><br>
  Title <input type="text" name="title"
    value="{{ book.get('title', '') }}"><br>
  Subtitle <input type="text" name="subtitle"
    value="{{ book.get('subtitle', '') }}"><br>
  Image <input type="text" name="image"
    value="{{ book.get('image', '') }}"><br>
  Author <input type="text" name="author"
    value="{{ book.get('author', '') }}"><br>
  Date released <input type="text" name="date_released"
    value="{{ book.get('date_released', '') }}"><br>
  Description<br>
  <textarea name="description" rows="5"
    cols="40">{% raw book.get('description', '') %}</textarea><br>
  <input type="submit" value="Save">
</form>
{% end %}
```

这是一个相当常规的 HTML 表单。如果请求处理函数传进来了 `book` 字典，那么我们用它预填充带有已存在书籍数据的表单；如果键不在字典中，我们使用 Python 字典对象的 `get` 方法为其提供默认值。记住 `input` 标签的 `name` 属性被设置为 `book` 字典的对应键；这使得与来自带有我们期望放入数据库数据的表单关联变得简单。

同样还需要记住的是，因为 `form` 标签没有 `action` 属性，因此表单的 `POST` 将会定向到当前 URL，这正是我们想要的（即，如果页面以 `/edit/0-123-456` 加载，`POST` 请求将转向 `/edit/0-123-456`；如果页面以 `/add` 加载，则 `POST` 将转向 `/add`）。图 4-1 所示为该页面渲染后的样子。

**Burt's Books**

ISBN

Title

Subtitle

Image

Author

Date released

Description

For more information about our selection, hours or events, please email us at [contact@burtsbooks.com](mailto:contact@burtsbooks.com).

Follow us on Facebook at <https://facebook.com/burtsbooks>.

图 4-1 Burt's Books: 添加新书的表单

#### 4.3.2.3 保存到数据库中

让我们看看 *BookEditHandler* 的 *post* 方法。这个方法处理书籍编辑表单的请求。下面是源代码：

```
def post(self, isbn=None):
    import time
    book_fields = ['isbn', 'title', 'subtitle', 'image', 'author',
                  'date_released', 'description']
    coll = self.application.db.books
    book = dict()
    if isbn:
        book = coll.find_one({"isbn": isbn})
    for key in book_fields:
        book[key] = self.get_argument(key, None)

    if isbn:
        coll.save(book)
```

```
else:
    book['date_added'] = int(time.time())
    coll.insert(book)
self.redirect("/recommended/")
```

和 *get* 方法一样，*post* 方法也有两个任务：处理编辑已存在文档的请求以及添加新文档的请求。如果有 *isbn* 参数（即，路径的请求类似于 */edit/0-123-456*），我们假定为编辑给定 ISBN 的文档。如果这个参数没有被提供，则假定为添加一个新文档。

我们先设置一个空的字典变量 *book*。如果我们正在编辑一个已存在的书籍，我们使用 *book* 集合的 *find\_one* 方法从数据库中加载和传入的 ISBN 值对应的文档。无论哪种情况，*book\_fields* 列表指定哪些域应该出现在书籍文档中。我们迭代这个列表，使用 *RequestHandler* 对象的 *get\_argument* 方法从 *POST* 请求中抓取对应的值。

此时，我们准备好更新数据库了。如果我们有一个 ISBN 码，那么我们调用集合的 *save* 方法来更新数据库中的书籍文档。如果没有的话，我们调用集合的 *insert* 方法，此时要注意首先要为 *date\_added* 键添加一个值。（我们没有将其包含在我们的域列表中获取传入的请求，因为在图书被添加到数据库之后 *date\_added* 值不应该再被改变。）当我们完成时，使用 *RequestHandler* 类的 *redirect* 方法给用户返回推荐页面。我们所做的任何改变可以立刻显现。图 4-2 所示为更新后的推荐页面。

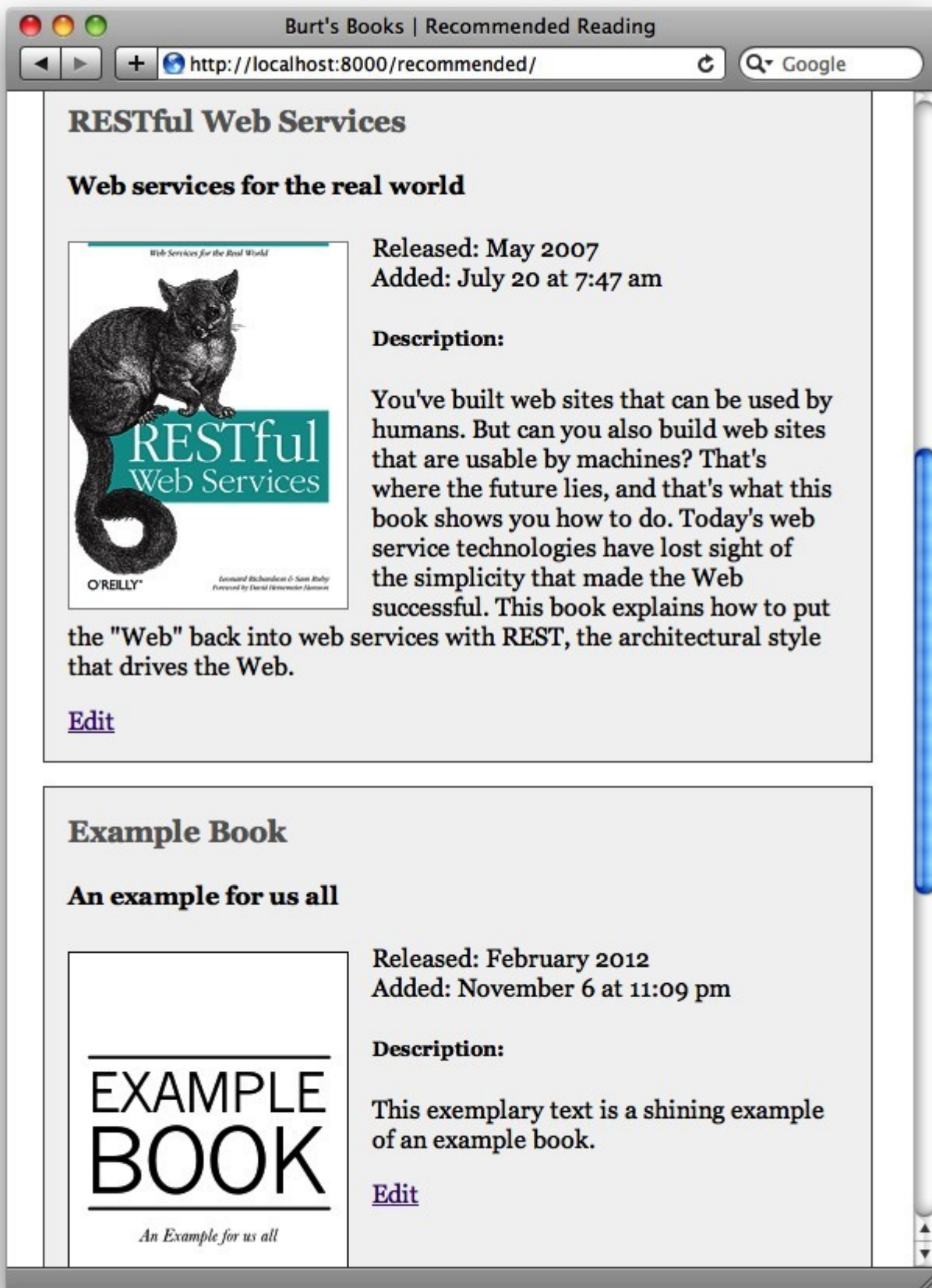


图 4-2 Burt's Books: 带有新添加书籍的推荐列表

你还将注意到我们给每个图书条目添加了一个“Edit”链接，用于链接到列表中每个书籍的编辑表单。下面是修改后的图书模块的源代码：

```

<div class="book" style="overflow: auto">
  <h3 class="book_title">{{ book["title"] }}</h3>
  {% if book["subtitle"] != "" %}
    <h4 class="book_subtitle">{{ book["subtitle"] }}</h4>
  {% end %}
  
  <div class="book_details">
    <div class="book_date_released">Released: {{ book["date_released"] }}</div>
    <div class="book_date_added">Added: {{ locale.format_date(book["date_added"],
relative=False) }}</div>
    <h5>Description:</h5>
    <div class="book_body">{% raw book["description"] %}</div>
    <p><a href="/edit/{{ book['isbn'] }}">Edit</a></p>
  </div>
</div>

```

其中最重要的一行是：

```

<p><a href="/edit/{{ book['isbn'] }}">Edit</a></p>

```

编辑页面的链接是把图书的 *isbn* 键的值添加到字符串 */edit/* 后面组成的。这个链接将会带你进入这本图书的编辑表单。你可以从图 4-3 中看到结果。



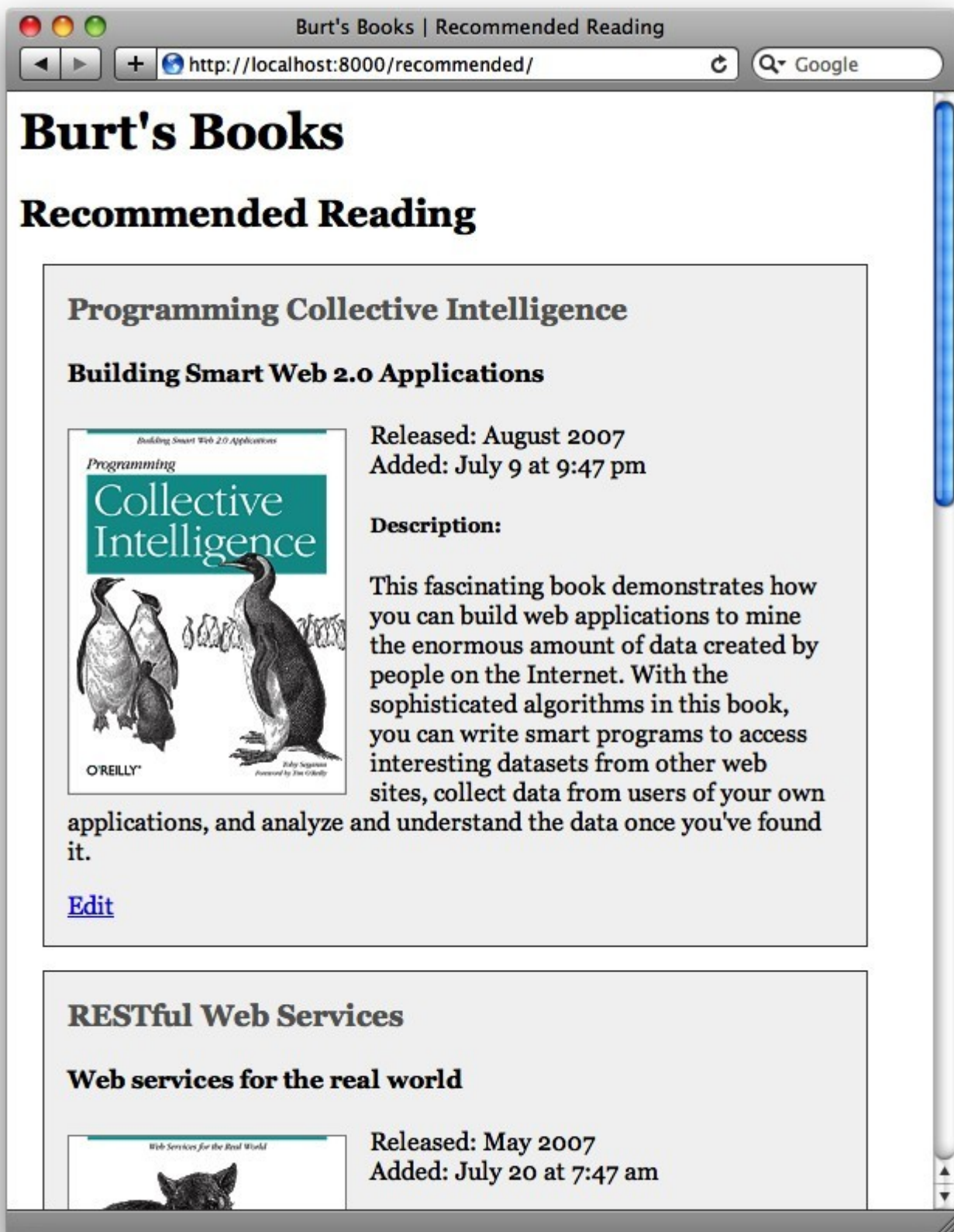


图 4-3 Burt's Books: 带有编辑链接的推荐列表

## 4.4 MongoDB: 下一步



我们在这里只覆盖了 MongoDB 的一些基础知识——仅仅够实现本章中的示例 Web 应用。如果你对于学习更多更用的 PyMongo 和 MongoDB 知识感兴趣的话，PyMongo 教程

(<http://api.mongodb.org/python/2.0.1/tutorial.html>) 和 MongoDB 教程

(<http://www.mongodb.org/display/DOCS/Tutorial>) 是不错的起点。

如果你对使用 Tornado 创建在扩展性方面表现更好的 MongoDB 应用感兴趣的话，你可以自学 `asyncmongo` (<https://github.com/bitly/asyncmongo>)，这是一种异步执行 MongoDB 请求的类似 PyMongo 的库。我们将在第 5 章中讨论什么是异步请求，以及为什么它在 Web 应用中扩展性更好。

## 第五章：异步 Web 服务

到目前为止，我们已经看到了许多使 Tornado 成为一个 Web 应用强有力框架的功能。它的简单性、易用性和便捷性使其有足够的理由成为许多 Web 项目的不错的选择。然而，Tornado 受到最多关注的功能是其异步取得和提供内容的能力，它有着很好的理由：它使得处理非阻塞请求更容易，最终导致更高效的处理以及更好的可扩展性。在本章中，我们将看到 Tornado 异步请求的基础，以及一些推送技术，这种技术可以使你使用更少的资源来提供更多的请求以编写更简单的 Web 应用。

### 5.1 异步 Web 请求

大部分 Web 应用（包括我们之前的例子）都是阻塞性质的，也就是说当一个请求被处理时，这个进程就会被挂起直至请求完成。在大多数情况下，Tornado 处理的 Web 请求完成得足够快使得这个问题并不需要被关注。然而，对于那些需要一些时间来完成的操作（像大数据库的请求或外部 API），这意味着应用程序被有效的锁定直至处理结束，很明显这在可扩展性上出现了问题。

不过，Tornado 给了我们更好的方法来处理这种情况。应用程序在等待第一个处理完成的过程中，让 I/O 循环打开以便服务于其他客户端，直到处理完成时启动一个请求并给予反馈，而不再是等待请求完成的过程中挂起进程。

为了实现 Tornado 的异步功能，我们构建一个向 Twitter 搜索 API 发送 HTTP 请求的简单 Web 应用。这个 Web 应用有一个参数 `q` 作为查询字符串，并确定多久会出现一条符合搜索条件的推文被发布在 Twitter 上（“每秒推数”）。确定这个数值的方法非常粗糙，但足以达到例子的目的。图 5-1 展示了这个应用的界面。

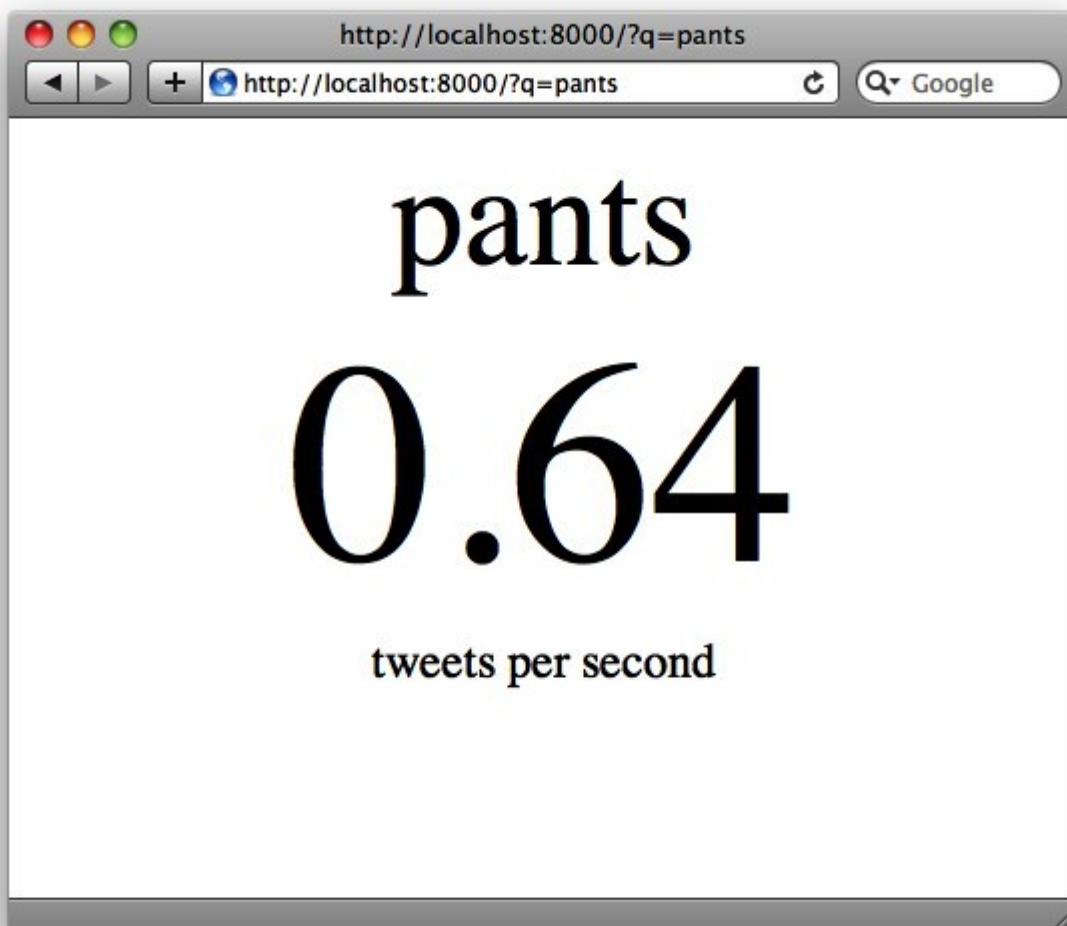


图 5-1 异步 HTTP 示例：推率

我们将展示这个应用的三个不同版本：首先，是一个使用同步 HTTP 请求的版本，然后是一个使用带有回调函数的 Tornado 异步 HTTP 客户端版本。最后，我们将展示如何使用 Tornado 2.1 版本新增的 *gen* 模块来使异步 HTTP 请求更加清晰和易实现。为了理解这些例子，你不需要成为关于 Twitter 搜索 API 的专家，但一定的熟悉不会有害。你可以在 <https://dev.twitter.com/docs/api/1/get/search> 阅读关于搜索 API 的开发者文档。

### 5.1.1 从同步开始

代码清单 5-1 包含我们的推率计算器的同步版本的代码。记住我们在顶部导入了 Tornado 的 *httpclient* 模块：我们将使用这个模块的 *HTTPClient* 类来执行 HTTP 请求。之后，我们将使用这个模块的 *AsyncHTTPClient*。

代码清单 5-1 同步 HTTP 请求：tweet\_rate.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
import tornado.httpclient
```

```

import urllib
import json
import datetime
import time

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        query = self.get_argument('q')
        client = tornado.httpclient.HTTPClient()
        response = client.fetch("http://search.twitter.com/search.json?" + \
                                urllib.urlencode({"q": query, "result_type": "recent", "rpp": 100}))
        body = json.loads(response.body)
        result_count = len(body['results'])
        now = datetime.datetime.utcnow()
        raw_oldest_tweet_at = body['results'][-1]['created_at']
        oldest_tweet_at = datetime.datetime.strptime(raw_oldest_tweet_at,
                                                    "%a, %d %b %Y %H:%M:%S +0000")
        seconds_diff = time.mktime(now.timetuple()) - \
                        time.mktime(oldest_tweet_at.timetuple())
        tweets_per_second = float(result_count) / seconds_diff
        self.write("""
<div style="text-align: center">
    <div style="font-size: 72px">%s</div>
    <div style="font-size: 144px">%.02f</div>
    <div style="font-size: 24px">tweets per second</div>
</div>""" % (query, tweets_per_second))

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

这个程序的结构现在对你而言应该已经很熟悉了：我们有一个 *RequestHandler* 类和一个处理到应用根路径请求的 *IndexHandler*。在 *IndexHandler* 的 *get* 方法中，我们从查询字符串中抓取参数 *q*，然后用它执行一个到 Twitter 搜索 API 的请求。下面是最相关的一部分代码：

```

client = tornado.httpclient.HTTPClient()
response = client.fetch("http://search.twitter.com/search.json?" + \
                        urllib.urlencode({"q": query, "result_type": "recent", "rpp": 100}))
body = json.loads(response.body)

```

这里我们实例化了一个 Tornado 的 *HTTPClient* 类，然后调用结果对象的 *fetch* 方法。*fetch* 方法的同步版本使用要获取的 URL 作为参数。这里，我们构建一个 URL 来抓取 Twitter 搜索 API 的相关搜索结果（*rpp* 参数指定我们想获得搜索结果首页的 100 个推文，而 *result\_type* 参数指定我们只想

获得匹配搜索的最近推文)。 *fetch* 方法会返回一个 *HTTPResponse* 对象，其 *body* 属性包含我们从远端 URL 获取的任何数据。Twitter 将返回一个 JSON 格式的结果，所以我们可以使用 Python 的 *json* 模块来从结果中创建一个 Python 数据结构。

*fetch* 方法返回的 *HTTPResponse* 对象允许你访问 HTTP 响应的任何部分，不只是 *body*。可以在[官方文档](#) [1] 阅读更多相关信息。

处理函数的其余部分关注的是计算每秒推文数。我们使用搜索结果中最旧推文与最新推文时间戳之差来确定搜索覆盖的时间，然后使用这个数值除以搜索取得的推文数来获得我们的最终结果。最后，我们编写了一个拥有这个结果的简单 HTML 页面给浏览器。

### 5.1.2 阻塞的困扰

到目前为止，我们已经编写了一个请求 Twitter API 并向浏览器返回结果的简单 Tornado 应用。尽管应用程序本身响应相当快，但是向 Twitter 发送请求到获得返回的搜索数据之间有相当大的滞后。在同步（到目前为止，我们假定为单线程）应用，这意味着同时只能提供一个请求。所以，如果你的应用涉及一个 2 秒的 API 请求，你将每间隔一秒才能提供（最多！）一个请求。这并不是你所称的高可扩展性应用，即便扩展到多线程和/或多服务器。

为了更具体的看出这个问题，我们对刚编写的例子进行基准测试。你可以使用任何基准测试工具来验证这个应用的性能，不过在这个例子中我们使用优秀的 [Siege utility](#) 工具进行测试。它可以这样使用：

```
$ siege http://localhost:8000/?q=pants -c10 -t10s
```

在这个例子中，Siege 对我们的应用在 10 秒内执行大约 10 个并发请求，输出结果如图 5-2 所示。我们可以很容易看出，这里的问题是无论每个请求自身返回多么快，API 往返都会以至于产生足够大的滞后，因为进程直到请求完成并且数据被处理前都一直处于强制挂起状态。当一两个请求时这还不是一个问题，但达到 100 个（甚至 10 个）用户时，这意味着整体变慢。

```
mjd — bash — bash — 88x47
Science:~ mjd$ siege http://localhost:8000/?q=pants -c10 -t10s
** SIEGE 2.70
** Preparing 10 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.48 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.80 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.07 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.50 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.69 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.92 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.14 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.37 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.61 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.84 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.01 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.06 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.30 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.33 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.33 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.19 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.25 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.92 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.99 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.92 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 2.15 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.28 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.27 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.25 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.18 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.30 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.63 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 3.77 secs: 178 bytes ==> /?q=pants

Lifting the server siege... done.
Transactions: 29 hits
Availability: 100.00 %
Elapsed time: 9.07 secs
Data transferred: 0.00 MB
Response time: 1.99 secs
Transaction rate: 3.20 trans/sec
Throughput: 0.00 MB/sec
Concurrency: 6.37
Successful transactions: 29
Failed transactions: 0
Longest transaction: 3.77
Shortest transaction: 0.24
```

图 5-2 同步推率获取

此时，不到 10 秒时间 10 个相似用户的平均响应时间达到了 1.99 秒，共计 29 次。请记住，这个例子只提供了一个 非常简单的网页。如果你要添加其他 Web 服务或数据库的调用的话，结果会更糟糕。



这种代码如果被 用到网站上，即便是中等强度的流量都会导致请求增长缓慢，甚至发生超时或失败。

### 5.1.3 基础异步调用

幸运的是，Tornado 包含一个 *AsyncHTTPClient* 类，可以执行异步 HTTP 请求。它和代码清单 5-1 的同步客户端实现有一定的相似性，除了一些我们将要讨论的重要区别。代码清单 5-2 是其源代码。

代码清单 5-2 异步 HTTP 请求：tweet\_rate\_async.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
import tornado.httpclient

import urllib
import json
import datetime
import time

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        query = self.get_argument('q')
        client = tornado.httpclient.AsyncHTTPClient()
        client.fetch("http://search.twitter.com/search.json?" + \
            urllib.urlencode({"q": query, "result_type": "recent", "rpp": 100}),
            callback=self.on_response)

    def on_response(self, response):
        body = json.loads(response.body)
        result_count = len(body['results'])
        now = datetime.datetime.utcnow()
        raw_oldest_tweet_at = body['results'][-1]['created_at']
        oldest_tweet_at = datetime.datetime.strptime(raw_oldest_tweet_at,
            "%a, %d %b %Y %H:%M:%S +0000")
        seconds_diff = time.mktime(now.timetuple()) - \
            time.mktime(oldest_tweet_at.timetuple())
        tweets_per_second = float(result_count) / seconds_diff
        self.write("""
<div style="text-align: center">
    <div style="font-size: 72px">%s</div>
    <div style="font-size: 144px">%.02f</div>
    <div style="font-size: 24px">tweets per second</div>
</div>""" % (self.get_argument('q'), tweets_per_second))
```

```

        self.finish()

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

*AsyncHTTPClient* 的 *fetch* 方法并不返回调用的结果。取而代之的是它指定了一个 *callback* 参数；你指定的方法或函数将在 HTTP 请求完成时被调用，并使用 *HTTPResponse* 作为其参数。

```

client = tornado.httpclient.AsyncHTTPClient()
client.fetch("http://search.twitter.com/search.json?" + »
urllib.urlencode({"q": query, "result_type": "recent", "rpp": 100}),
    callback=self.on_response)

```

在这个例子中，我们指定 *on\_response* 方法作为回调函数。我们之前使用期望的输出转化 Twitter 搜索 API 请求到网页中的所有逻辑被搬到了 *on\_response* 函数中。还需要注意的是 *@tornado.web.asynchronous* 装饰器的使用（在 *get* 方法的定义之前）以及在回调方法结尾处调用的 *self.finish()*。我们稍后将简要的讨论他们的细节。

这个版本的应用拥有和之前同步版本相同的外观，但其性能更加优越。有多好呢？让我们看看基准测试的结果吧。

正如你在图 5-3 中所看到的，我们从同步版本的每秒 3.20 个事务提升到了 12.59，在相同的时间内总共提供了 118 次请求。这真是一个非常大的改善！正如你所想象的，当扩展到更多用户和更长时间时，它将能够提供更多连接，并且不会遇到同步版本遭受的变慢的问题。

```
mjd — bash — bash — 88x47
HTTP/1.1 200 0.34 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.46 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 1.72 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.25 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.20 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants

Lifting the server siege... done.
Transactions: 118 hits
Availability: 100.00 %
Elapsed time: 9.37 secs
Data transferred: 0.02 MB
Response time: 0.29 secs
Transaction rate: 12.59 trans/sec
Throughput: 0.00 MB/sec
Concurrency: 3.59
Successful transactions: 118
Failed transactions: 0
Longest transaction: 1.72
Shortest transaction: 0.20
```

图 5-3 异步推率获取

#### 5.1.4 异步装饰器和 finish 方法



Tornado 默认在函数处理返回时关闭客户端的连接。在通常情况下，这正是你想要的。但是当我们处理一个需要回调函数的异步请求时，我们需要连接保持开启状态直到回调函数执行完毕。你可以在你想改变其行为的方法上面使用 `@tornado.web.asynchronous` 装饰器来告诉 Tornado 保持连接开启，正如我们在异步版本的推率例子中 `IndexHandler` 的 `get` 方法中所做的。下面是相关的代码片段：

```
class IndexHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        query = self.get_argument('q')
        [... other request handler code here...]
```

记住当你使用 `@tornado.web.asynchronous` 装饰器时，Tornado 永远不会自己关闭连接。你必须在你的 `RequestHandler` 对象中调用 `finish` 方法来显式地告诉 Tornado 关闭连接。（否则，请求将可能挂起，浏览器可能不会显示我们已经发送给客户端的数据。）在前面的异步示例中，我们在 `on_response` 函数的 `write` 后面调用了 `finish` 方法：

```
        [... other callback code ...]
        self.write("""
<div style="text-align: center">
    <div style="font-size: 72px">%s</div>
    <div style="font-size: 144px">%.02f</div>
    <div style="font-size: 24px">tweets per second</div>
</div>""" % (self.get_argument('q'), tweets_per_second))
        self.finish()
```

### 5.1.5 异步生成器

现在，我们的推率程序的异步版本运转的不错并且性能也很好。不幸的是，它有点麻烦：为了处理请求，我们不得不把我们的代码分割成两个不同的方法。当我们有两个或更多的异步请求要执行的时候，编码和维护都显得非常困难，每个都依赖于前面的调用：不久你就会发现自己调用了一个回调函数的回调函数的回调函数。下面就是一个构想出来的（但不是不可能的）例子：

```
def get(self):
    client = AsyncHTTPClient()
    client.fetch("http://example.com", callback=on_response)

def on_response(self, response):
    client = AsyncHTTPClient()
    client.fetch("http://another.example.com/", callback=on_response2)

def on_response2(self, response):
    client = AsyncHTTPClient()
    client.fetch("http://still.another.example.com/", callback=on_response3)

def on_response3(self, response):
    [..., etc.]
```

幸运的是，Tornado 2.1 版本引入了 *tornado.gen* 模块，可以提供一个更整洁的方式来执行异步请求。代码清单 5-3 就是使用了 *tornado.gen* 版本的推率应用源代码。让我们先来看一下，然后讨论它是如何工作的。

代码清单 5-3 使用生成器模式的异步请求：tweet\_rate\_gen.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
import tornado.httpclient
import tornado.gen

import urllib
import json
import datetime
import time

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class IndexHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self):
        query = self.get_argument('q')
        client = tornado.httpclient.AsyncHTTPClient()
        response = yield tornado.gen.Task(client.fetch,
            "http://search.twitter.com/search.json?" + \
            urllib.urlencode({"q": query, "result_type": "recent", "rpp": 100}))
        body = json.loads(response.body)
        result_count = len(body['results'])
        now = datetime.datetime.utcnow()
        raw_oldest_tweet_at = body['results'][-1]['created_at']
        oldest_tweet_at = datetime.datetime.strptime(raw_oldest_tweet_at,
            "%a, %d %b %Y %H:%M:%S +0000")
        seconds_diff = time.mktime(now.timetuple()) - \
            time.mktime(oldest_tweet_at.timetuple())
        tweets_per_second = float(result_count) / seconds_diff
        self.write("""
<div style="text-align: center">
    <div style="font-size: 72px">%s</div>
    <div style="font-size: 144px">%.02f</div>
    <div style="font-size: 24px">tweets per second</div>
</div>""" % (query, tweets_per_second))
        self.finish()

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[(r"/", IndexHandler)])
```

```
http_server = tornado.httpserver.HTTPServer(app)
http_server.listen(options.port)
tornado.ioloop.IOLoop.instance().start()
```

正如你所看到的，这个代码和前面两个版本的代码非常相似。主要的不同点是我们如何调用 *Asynchronous* 对象的 *fetch* 方法。下面是相关的代码部分：

```
client = tornado.httpclient.AsyncHTTPClient()
response = yield tornado.gen.Task(client.fetch,
    "http://search.twitter.com/search.json?" + \
    urllib.urlencode({"q": query, "result_type": "recent", "rpp": 100}))
body = json.loads(response.body)
```

我们使用 Python 的 *yield* 关键字以及 *tornado.gen.Task* 对象的一个实例，将我们想要的调用和传给该调用函数的参数传递给那个函数。这里，*yield* 的使用返回程序对 Tornado 的控制，允许在 HTTP 请求进行中执行其他任务。当 HTTP 请求完成时，*RequestHandler* 方法在其停止的地方恢复。这种构建的美在于它在请求处理程序中返回 HTTP 响应，而不是回调函数中。因此，代码更易理解：所有请求相关的逻辑位于同一个位置。而 HTTP 请求依然是异步执行的，所以我们使用 *tornado.gen* 可以达到和使用回调函数的异步请求版本相同的性能，正如我们在图 5-4 中所看到的那样。

```
mjd — bash — bash — 88x47
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.29 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.20 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.21 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.24 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.32 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.23 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants
HTTP/1.1 200 0.22 secs: 178 bytes ==> /?q=pants

Lifting the server siege... done.
Transactions: 127 hits
Availability: 100.00 %
Elapsed time: 9.83 secs
Data transferred: 0.02 MB
Response time: 0.26 secs
Transaction rate: 12.92 trans/sec
Throughput: 0.00 MB/sec
Concurrency: 3.30
Successful transactions: 127
Failed transactions: 0
Longest transaction: 0.82
Shortest transaction: 0.20
```

图 5-4 使用 tornado.gen 的异步推率获取

记住 `@tornado.gen.engine` 装饰器的使用需要刚好在 `get` 方法的定义之前；这将提醒 Tornado 这个方法将使用 `tornado.gen.Task` 类。`tornado.gen` 模块还哟一些其他类和函数可以方便 Tornado 的异步编程。查阅一下[文档](#)<sup>[1]</sup>是非常值得的。

## 使一切异步

在本章中我们使用了 Tornado 的异步 HTTP 客户端作为如何执行异步任务的实现。其他开发者也编写了针对其他任务的异步客户端库。志愿者们在 [Tornado wiki](#) 上维护了一个关于这些库的相当完整的列表。

一个重要的例子是 bit.ly 的 [asyncmongo](#)，它可以异步的调用 MongoDB 服务器。这个库是我们的一个非常不错的选择，因为它是专门给 Tornado 开发者开发提供异步数据库访问的，不过对于使用其他数据库的用户而言，在这里也可以找到不错的异步数据存储库的选择。

### 5.1.6 异步操作总结

正如我们在前面的例子中所看到的，Tornado 异步 Web 发服务不仅容易实现也在实践中有着不容小觑的能力。使用异步处理可以让我们的应用在长时间的 API 和数据库请求中免受阻塞之苦，最终更快地提供更多请求。尽管不是所有的处理都能从异步中受益——并且实际上尝试整个程序非阻塞会迅速使事情变得复杂——但 Tornado 的非阻塞功能可以非常方便的创建依赖于缓慢查询或外部服务的 Web 应用。

不过，值得注意的是，这些例子都非常的做作。如果你正在设计一个任何规模下带有该功能的应用，你可能希望客户端浏览器来执行 Twitter 搜索请求（使用 JavaScript），而让 Web 服务器转向提供其他请求。在大多数情况下，你至少希望将结果缓存以便两次相同搜索项的请求不会导致再次向远程 API 执行完整请求。通常，如果你在后端执行 HTTP 请求提供网站内容，你可能希望重新思考如何建立你的应用。

考虑到这一点，在下一组示例中，我们将看看如何在前端使用像 JavaScript 这样的工具处理异步应用，让客户端承担更多工作，以提高你应用的扩展性。

## 5.2 使用 Tornado 进行长轮询

Tornado 异步架构的另一个优势是它能够轻松处理 HTTP 长轮询。这是一个处理实时更新的方法，它既可以应用到简单的数字标记通知，也可以实现复杂的多用户聊天室。

部署提供实时更新的 Web 应用对于 Web 程序员而言是一项长期的挑战。更新用户状态、发送新消息提醒、或者任何一个需要在初始文档完成加载后由服务器向浏览器发送消息方法的全局活动。一个早期的方法是浏览器以一个固定的时间间隔向服务器轮询新请求。这项技术带来了新的挑战：轮询频率必须足够快以便通知是最新的，但又不能太频繁，当成百上千的客户端持续不断的打开新的连接会使 HTTP 请求面临严重的扩展性挑战。频繁的轮询使得 Web 服务器遭受“凌迟”之苦。

所谓的“服务器推送”技术允许 Web 应用实时发布更新，同时保持合理的资源使用以及确保可预知的扩展。对于一个可行的服务器推送技术而言，它必须在现有的浏览器上表现良好。最流行的技术是让浏览器发起连接来模拟服务器推送更新。这种方式的 HTTP 连接被称为长轮询或 Comet 请求。



长轮询意味着浏览器只需启动一个 HTTP 请求，其连接的服务器会有意保持开启。浏览器只需要等待更新可用时服务器“推送”响应。当服务器发送响应并关闭连接后，（或者浏览器端客户请求超时），客户端只需打开一个新的连接并等待下一个更新。

本节将包括一个简单的 HTTP 长轮询实时应用以及证明 Tornado 架构如何使这些应用更简单。

### 5.2.1 长轮询的好处

HTTP 长轮询的主要吸引力在于其极大地减少了 Web 服务器的负载。相对于客户端制造大量的短而频繁的请求（以及每次处理 HTTP 头部产生的开销），服务器端只有当其接收一个初始请求和再次发送响应时处理连接。大部分时间没有新的数据，连接也不会消耗任何处理器资源。

浏览器兼容性是另一个巨大的好处。任何支持 AJAX 请求的浏览器都可以执行推送请求。不需要任何浏览器插件或其他附加组件。对比其他服务器端推送技术，HTTP 长轮询最终成为了被广泛使用的少数几个可行方案之一。

我们已经接触过长轮询的一些使用。实际上，前面提到的状态更新、消息通知以及聊天消息都是目前流行的网站功能。像 Google Docs 这样的站点使用长轮询同步协作，两个人可以同时编辑文档并看到对方的改变。Twitter 使用长轮询指示浏览器在新状态更新可用时展示通知。Facebook 使用这项技术在其聊天功能中。长轮询如此流行的一个原因是它改善了应用的用户体验：访客不再需要不断地刷新页面来获取最新的内容。

### 5.2.2 示例：实时库存报告

这个例子演示了一个根据多个购物者浏览器更新的零售商库存实时计数服务。这个应用提供一个带有“Add to Cart”按钮的 HTML 书籍细节页面，以及书籍剩余库存的计数。一个购物者将书籍添加到购物车之后，其他访问这个站点的访客可以立刻看到库存的减少。

为了提供库存更新，我们需要编写一个在初始化处理方法调用后不会立即关闭 HTTP 连接的 *RequestHandler* 子类。我们使用 Tornado 内建的 *asynchronous* 装饰器完成这项工作，如代码清单 5-4 所示。

代码清单 5-4 长轮询：shopping\_cart.py

```
import tornado.web
import tornado.httpserver
import tornado.ioloop
import tornado.options
from uuid import uuid4

class ShoppingCart(object):
    totalInventory = 10
    callbacks = []
    carts = {}

    def register(self, callback):
        self.callbacks.append(callback)

    def moveItemToCart(self, session):
```

```

        if session in self.carts:
            return

        self.carts[session] = True
        self.notifyCallbacks()

def removeItemFromCart(self, session):
    if session not in self.carts:
        return

    del(self.carts[session])
    self.notifyCallbacks()

def notifyCallbacks(self):
    for c in self.callbacks:
        self.callbackHelper(c)

    self.callbacks = []

def callbackHelper(self, callback):
    callback(self.getInventoryCount())

def getInventoryCount(self):
    return self.totalInventory - len(self.carts)

class DetailHandler(tornado.web.RequestHandler):
    def get(self):
        session = uuid4()
        count = self.application.shoppingCart.getInventoryCount()
        self.render("index.html", session=session, count=count)

class CartHandler(tornado.web.RequestHandler):
    def post(self):
        action = self.get_argument('action')
        session = self.get_argument('session')

        if not session:
            self.set_status(400)
            return

        if action == 'add':
            self.application.shoppingCart.moveToCart(session)
        elif action == 'remove':
            self.application.shoppingCart.removeItemFromCart(session)
        else:
            self.set_status(400)

class StatusHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous

```

```

def get(self):
    self.application.shoppingCart.register(self.async_callback(self.on_message))

def on_message(self, count):
    self.write('{"inventoryCount": "%d"}' % count)
    self.finish()

class Application(tornado.web.Application):
    def __init__(self):
        self.shoppingCart = ShoppingCart()

        handlers = [
            (r'/', DetailHandler),
            (r'/cart', CartHandler),
            (r'/cart/status', StatusHandler)
        ]

        settings = {
            'template_path': 'templates',
            'static_path': 'static'
        }

        tornado.web.Application.__init__(self, handlers, **settings)

if __name__ == '__main__':
    tornado.options.parse_command_line()

    app = Application()
    server = tornado.httpserver.HTTPServer(app)
    server.listen(8000)
    tornado.ioloop.IOLoop.instance().start()

```

让我们在看模板和脚本文件之前先详细看下 `shopping_cart.py`。我们定义了一个 *ShoppingCart* 类来维护我们的库存中商品的数量，以及把商品加入购物车的购物者列表。然后，我们定义了 *DetailHandler* 用于渲染 HTML；*CartHandler* 用于提供操作购物车的接口；*StatusHandler* 用于查询全局库存变化的通知。

*DetailHandler* 为每个页面请求产生一个唯一标识符，在每次请求时提供库存数量，并向浏览器渲染 `index.html` 模板。*CartHandler* 为浏览器提供了一个 API 来请求从访客的购物车中添加或删除物品。浏览器中运行的 JavaScript 提交 *POST* 请求来操作访客的购物车。我们将在下面的 *StatusHandler* 和 *ShoppingCart* 类的讲解中看到这些方法是如何作用域库存数量查询的。

```

class StatusHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        self.application.shoppingCart.register(self.async_callback(self.on_message))

```

关于 *StatusHandler* 首先需要注意的是 *get* 方法上面的 `@tornado.web.asynchronous` 装饰器。这使得 Tornado 在 *get* 方法返回时不会关闭连接。在这个方法中，我们只是注册了一个带有购物车控制



器的回调函数。我们使用 *self.async\_callback* 包住回调函数以确保回调函数中引发的异常不会使 *RequestHandler* 关闭连接。

在 Tornado 1.1 之前的版本中，回调函数必须被包在 *self.async\_callback()* 方法中来捕获被包住的函数可能会产生的异常。不过，在 Tornado 1.1 或更新版本中，这不再是显式必须的了。

```
def on_message(self, count):
    self.write('{"inventoryCount": "%d"}' % count)
    self.finish()
```

每当访客操作购物车，*ShoppingCart* 控制器为每个已注册的回调函数调用 *on\_message* 方法。这个方法将当前库存数量写入客户端并关闭连接。（如果服务器不关闭连接的话，浏览器可能不会知道请求已经被完成，也不会通知脚本有过更新。）既然长轮询连接已经关闭，购物车控制器必须删除已注册的回调函数列表中的回调函数。在这个例子中，我们只需要将回调函数列表替换为一个新的空列表。在请求处理中被调用并完成后删除已注册的回调函数十分重要，因为随后在调用回调函数时将在之前已关闭的连接上调用 *finish()*，这会产生一个错误。

最后，*ShoppingCart* 控制器管理库存分批和状态回调。*StatusHandler* 通过 *register* 方法注册回调函数，即添加这个方法到内部的 *callbacks* 数组。

```
def moveItemToCart(self, session):
    if session in self.carts:
        return

    self.carts[session] = True
    self.notifyCallbacks()

def removeItemFromCart(self, session):
    if session not in self.carts:
        return

    del(self.carts[session])
    self.notifyCallbacks()
```

此外，*ShoppingCart* 控制器还实现了 *Carthandler* 中的 *addItemToCart* 和 *removeItemFromCart*。当 *Carthandler* 调用这些方法，请求页面的唯一标识符（传给这些方法的 *session* 变量）被用于在调用 *notifyCallbacks* 之前标记库存。<sup>[2]</sup>

```
def notifyCallbacks(self):
    for c in self.callbacks:
        self.callbackHelper(c)

    self.callbacks = []

def callbackHelper(self, callback):
    callback(self.getInventoryCount())
```

已注册的回调函数被以当前可用库存数量调用，并且回调函数列表被清空以确保回调函数不会在一个已经关闭的连接上调用。

代码清单 5-5 是展示书籍列表变化的模板。

代码清单 5-5 长轮询: index.html

```
<html>
  <head>
    <title>Burt's Books - Book Detail</title>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"
      type="text/javascript"></script>
    <script src="{{ static_url('scripts/inventory.js') }}"
      type="application/javascript"></script>
  </head>

  <body>
    <div>
      <h1>Burt's Books</h1>

      <hr/>
      <p><h2>The Definitive Guide to the Internet</h2>
        <em>Anonymous</em></p>
    </div>

    <hr />

    <input type="hidden" id="session" value="{{ session }}" />
    <div id="add-to-cart">
      <p><span style="color: red;">Only <span id="count">{{ count }}</span>
        left in stock! Order now!</span></p>
      <p>$20.00 <input type="submit" value="Add to Cart" id="add-button" /></p>
    </div>
    <div id="remove-from-cart" style="display: none;">
      <p><span style="color: green;">One copy is in your cart.</span></p>
      <p><input type="submit" value="Remove from Cart" id="remove-button" /></p>
    </div>
  </body>
</html>
```

当 *DetailHandler* 渲染 index.html 模板时, 我们只是渲染了图书的详细信息并包含了必需的 JavaScript 代码。此外, 我们通过 *session* 变量动态地包含了一个唯一 ID, 并以 *count* 变量保存当前库存值。

最后, 我们将讨论客户端的 JavaScript 代码。由于这是一本关于 Tornado 的书籍, 因此我们直到现在一直使用的是 Python, 而这个例子中的客户端代码是至关重要的, 我们至少要能够理解它的要点。在代码清单 5-6 中, 我们使用了 jQuery 库来协助定义浏览器的页面行为。

代码清单 5-6 长轮询: inventory.js

```
$(document).ready(function() {
```

```

document.session = $('#session').val();

setTimeout(requestInventory, 100);

$('#add-button').click(function(event) {
    jQuery.ajax({
        url: '//localhost:8000/cart',
        type: 'POST',
        data: {
            session: document.session,
            action: 'add'
        },
        dataType: 'json',
        beforeSend: function(xhr, settings) {
            $(event.target).attr('disabled', 'disabled');
        },
        success: function(data, status, xhr) {
            $('#add-to-cart').hide();
            $('#remove-from-cart').show();
            $(event.target).removeAttr('disabled');
        }
    });
});

$('#remove-button').click(function(event) {
    jQuery.ajax({
        url: '//localhost:8000/cart',
        type: 'POST',
        data: {
            session: document.session,
            action: 'remove'
        },
        dataType: 'json',
        beforeSend: function(xhr, settings) {
            $(event.target).attr('disabled', 'disabled');
        },
        success: function(data, status, xhr) {
            $('#remove-from-cart').hide();
            $('#add-to-cart').show();
            $(event.target).removeAttr('disabled');
        }
    });
});

function requestInventory() {
    jQuery.getJSON('//localhost:8000/cart/status', {session: document.session},
        function(data, status, xhr) {
            $('#count').html(data['inventoryCount']);
        }
    );
}

```

```

        setTimeout(requestInventory, 0);
    }
    );
}

```

当文档完成加载时，我们为“Add to Cart”按钮添加了点击事件处理函数，并隐藏了“Remove from Cart”按钮。这些事件处理函数关联服务器的 API 调用，并交换添加到购物车接口和从购物车移除接口。

```

function requestInventory() {
    jQuery.getJSON(' //localhost:8000/cart/status', {session: document.session},
        function(data, status, xhr) {
            $(' #count').html(data[' inventoryCount' ]);
            setTimeout(requestInventory, 0);
        }
    );
}

```

*requestInventory* 函数在页面完成加载后经过一个短暂的延迟再进行调用。在函数主体中，我们通过到 */cart/status* 的 HTTP *GET* 请求初始化一个长轮询。延迟允许在浏览器完成渲染页面时使加载进度指示器完成，并防止 Esc 键或停止按钮中断长轮询请求。当请求成功返回时，*count* 的内容更新为当前的库存量。图 5-5 所示为展示全部库存的两个浏览器窗口。

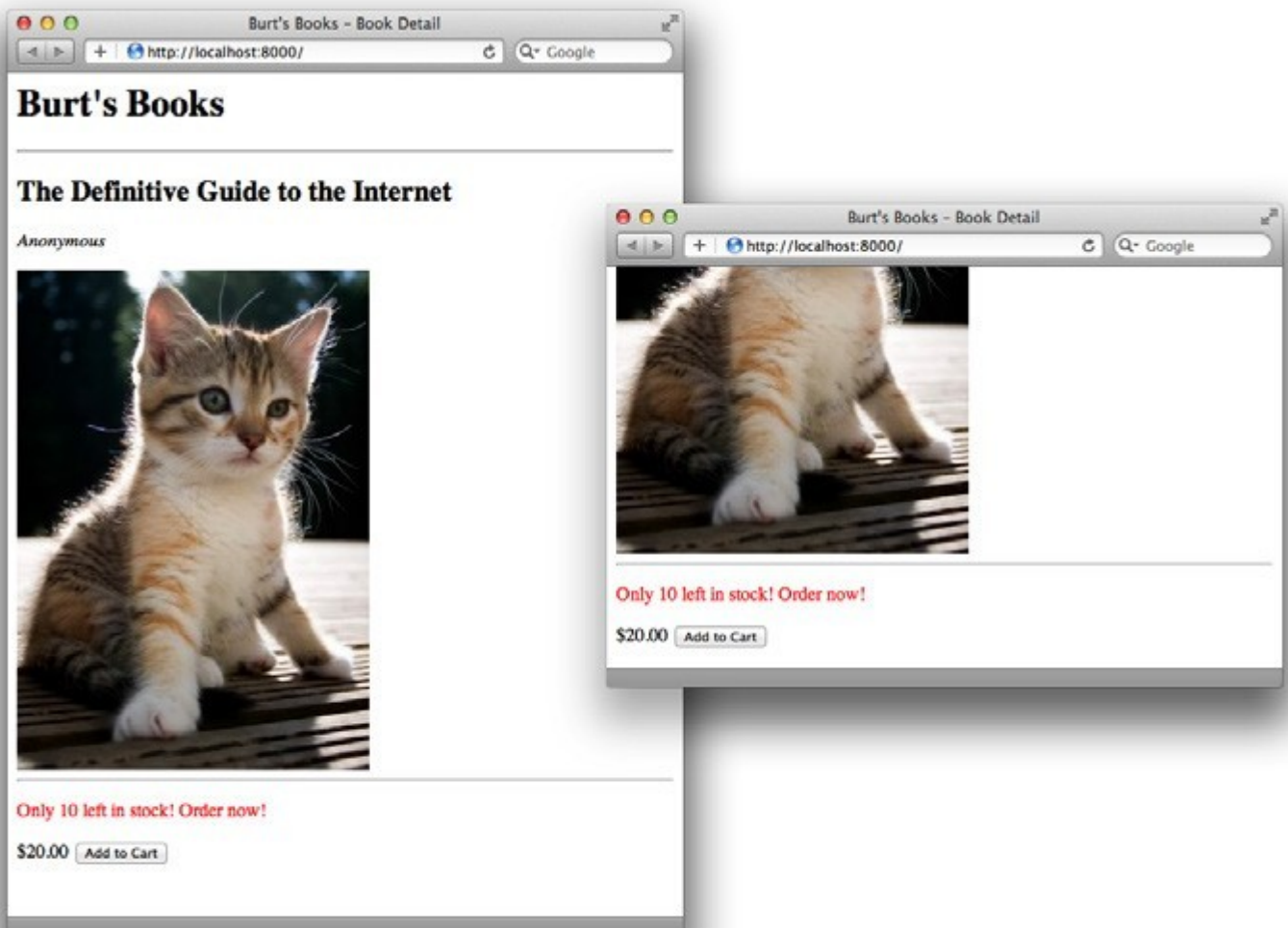


图 5-5 长轮询示例：全部库存

现在，当你运行服务器，你将可以加载根 URL 并看到书籍的当前库存数量。打开多个细节页的浏览器窗口，并在其中一个窗口点击“Add to Cart”按钮。其余窗口的剩余库存数量会立刻更新，如果 5-6 所示。

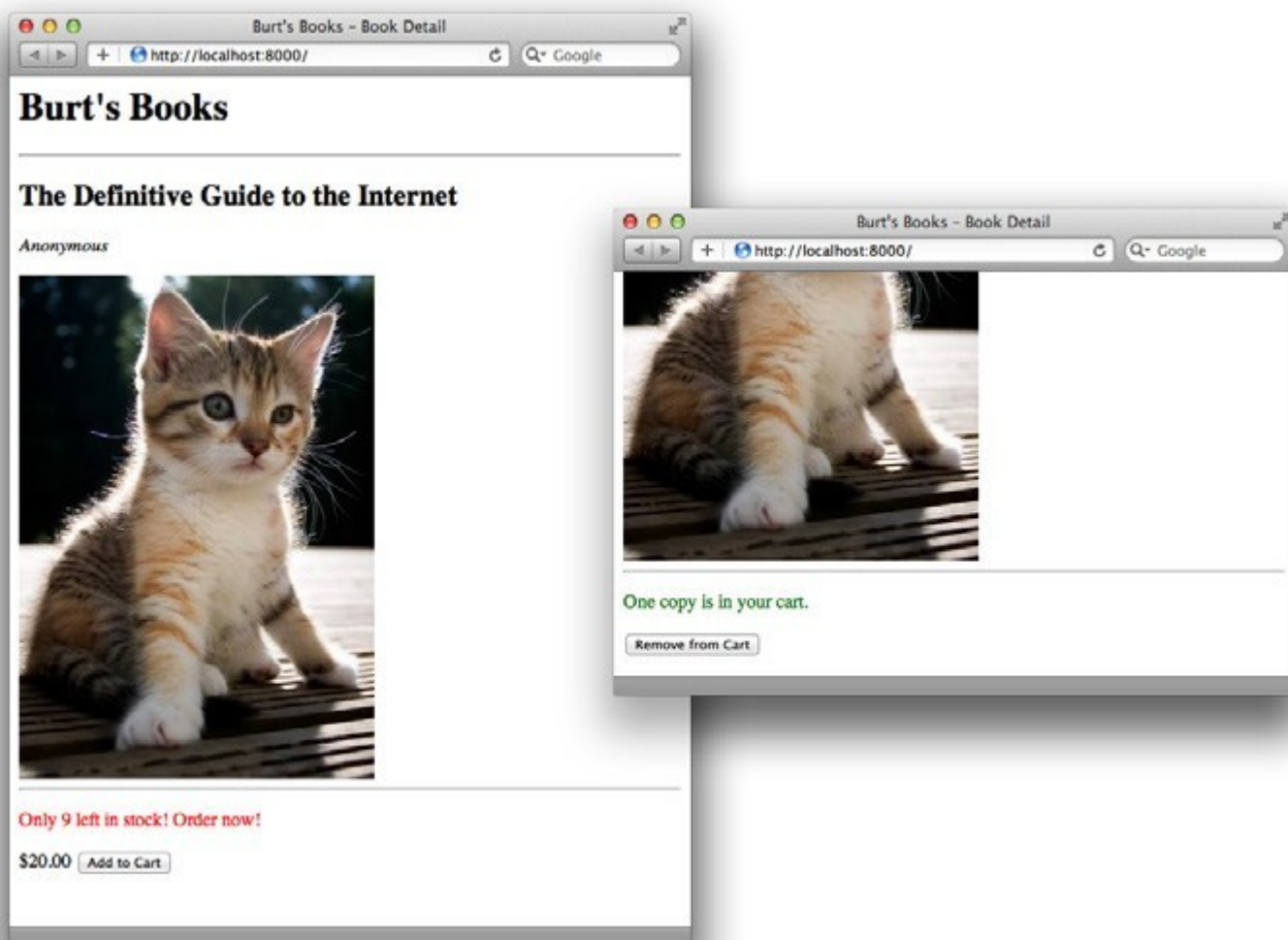


图 5-6 长轮询示例：一个物品在购物车中

这是一个非常简单的购物车实现，可以肯定的是一没有逻辑确保我们不会跌破总库存量，更不用说数据无法在 Tornado 应用的不同调用间或同一服务器并行的应用实例间保留。我们将这些改善作为练习留给读者。

### 5.2.3 长轮询的缺陷

正如我们所看到的，HTTP 长轮询在站点或特定用户状态的高度交互反馈通信中非常有用。但我们也应该知道它的一些缺陷。

当使用长轮询开发应用时，记住对于浏览器请求超时间隔无法控制是非常重要的。由浏览器决定在任何中断情况下重新开启 HTTP 连接。另一个潜在的问题是许多浏览器限制了对于打开的特定主机的并发请求数量。当有一个连接保持空闲时，剩下的用来下载网站内容的请求数量就会 有限制。

此外，你还应该明白请求是怎样影响服务器性能的。再次考虑购物车应用。由于在库存变化时所有的推送请求同时应答和关闭，使得在浏览器重新建立连接时服务器受到了新请求的猛烈冲击。对于像用户间聊天或消息通知这样的应用而言，只有少数用户的连接会同时关闭，这就不再是一个问题了。

## 5.3 Tornado 与 WebSockets

WebSockets 是 HTML5 规范中新提出的客户-服务器通讯协议。这个协议目前仍是草案，只有最新的一些浏览器可以支持它。但是，它的好处是显而易见的，随着支持它的浏览器越来越多，我们将看到它越来越流行。（和以往的 Web 开发一样，必须谨慎地坚持依赖可用的新功能并能在必要时回滚到旧技术的务实策略。）

WebSocket 协议提供了在客户端和服务端间持久连接的双向通信。协议本身使用新的 *ws://URL* 格式，但它是在标准 HTTP 上实现的。通过使用 HTTP 和 HTTPS 端口，它避免了从 Web 代理后的网络连接站点时引入的各种问题。HTML5 规范不只描述了协议本身，还描述了使用 WebSockets 编写客户端代码所需要的浏览器 API。

由于 WebSocket 已经在一些最新的浏览器中被支持，并且 Tornado 为之提供了一些有用的模块，因此来看看如何使用 WebSockets 实现应用是非常值得的。

### 5.3.1 Tornado 的 WebSocket 模块

Tornado 在 *websocket* 模块中提供了一个 *WebSocketHandler* 类。这个类提供了和已连接的客户端通信的 WebSocket 事件和方法的钩子。当一个新的 WebSocket 连接打开时，*open* 方法被调用，而 *on\_message* 和 *on\_close* 方法分别在连接接收到新的消息和客户端关闭时被调用。

此外，*WebSocketHandler* 类还提供了 *write\_message* 方法用于向客户端发送消息，*close* 方法用于关闭连接。

```
class EchoHandler(tornado.websocket.WebSocketHandler):
    def open(self):
        self.write_message('connected!')

    def on_message(self, message):
        self.write_message(message)
```

正如你在我们的 *EchoHandler* 实现中所看到的，*open* 方法只是使用 *WebSocketHandler* 基类提供的 *write\_message* 方法向客户端发送字符串“connected!”。每次处理程序从客户端接收到一个新的消息时调用 *on\_message* 方法，我们的实现中将客户端提供的消息原样返回给客户端。这就是全部！让我们通过一个完整的例子看看实现这个协议是如何简单的吧。

### 5.3.2 示例：使用 WebSockets 的实时库存

在本节中，我们可以看到把之前使用 HTTP 长轮询的例子更新为使用 WebSockets 是如何简单。但是，请记住，WebSockets 还是一个新标准，只有最新的浏览器版本可以支持它。Tornado 支持的特定版本的 WebSocket 协议版本只在 Firefox 6.0 或以上、Safari 5.0.1 或以上、Chrome 6 或以上、IE 10 预览版或以上版本的浏览器中可用。

不去管免责声明，让我们先看看源码吧。除了服务器应用需要在 *ShoppingCart* 和 *StatusHandler* 类中做一些修改外，大部分代码保持和之前一样。代码清单 5-7 看起来会很熟悉。

代码清单 5-7 WebSockets: shopping\_cart.py

```
import tornado.web
import tornado.websocket
import tornado.httpserver
import tornado.ioloop
import tornado.options
from uuid import uuid4

class ShoppingCart(object):
    totalInventory = 10
    callbacks = []
    carts = {}

    def register(self, callback):
        self.callbacks.append(callback)

    def unregister(self, callback):
        self.callbacks.remove(callback)

    def moveItemToCart(self, session):
        if session in self.carts:
            return

        self.carts[session] = True
        self.notifyCallbacks()

    def removeItemFromCart(self, session):
        if session not in self.carts:
            return

        del(self.carts[session])
        self.notifyCallbacks()

    def notifyCallbacks(self):
        for callback in self.callbacks:
            callback(self.getInventoryCount())

    def getInventoryCount(self):
        return self.totalInventory - len(self.carts)

class DetailHandler(tornado.web.RequestHandler):
    def get(self):
        session = uuid4()
        count = self.application.shoppingCart.getInventoryCount()
        self.render("index.html", session=session, count=count)
```

```

class CartHandler(tornado.web.RequestHandler):
    def post(self):
        action = self.get_argument('action')
        session = self.get_argument('session')

        if not session:
            self.set_status(400)
            return

        if action == 'add':
            self.application.shoppingCart.moveItemToCart(session)
        elif action == 'remove':
            self.application.shoppingCart.removeItemFromCart(session)
        else:
            self.set_status(400)

class StatusHandler(tornado.websocket.WebSocketHandler):
    def open(self):
        self.application.shoppingCart.register(self.callback)

    def on_close(self):
        self.application.shoppingCart.unregister(self.callback)

    def on_message(self, message):
        pass

    def callback(self, count):
        self.write_message('{"inventoryCount": "%d"}' % count)

class Application(tornado.web.Application):
    def __init__(self):
        self.shoppingCart = ShoppingCart()

        handlers = [
            (r '/', DetailHandler),
            (r '/cart', CartHandler),
            (r '/cart/status', StatusHandler)
        ]

        settings = {
            'template_path': 'templates',
            'static_path': 'static'
        }

        tornado.web.Application.__init__(self, handlers, **settings)

if __name__ == '__main__':
    tornado.options.parse_command_line()

```



```

app = Application()
server = tornado.httpserver.HTTPServer(app)
server.listen(8000)
tornado.ioloop.IOLoop.instance().start()

```

除了额外的导入语句外，我们只需要改变 *ShoppingCart* 和 *StatusHandler* 类。首先需要注意的是，为了获得 *WebSocketHandler* 的功能，需要使用 *tornado.websocket* 模块。

在 *ShoppingCart* 类中，我们只需要在通知回调函数的方式上做一个轻微的改变。因为 WebSockets 在一个消息发送后保持打开状态，我们不需要在它们被通知后移除内部的回调函数列表。我们只需要迭代列表并调用带有当前库存量的回调函数：

```

def notifyCallbacks(self):
    for callback in self.callbacks:
        callback(self.getInventoryCount())

```

另一个改变是添加了 *unregistered* 方法。*StatusHandler* 会在 WebSocket 连接关闭时调用该方法移除一个回调函数。

```

def unregister(self, callback):
    self.callbacks.remove(callback)

```

大部分改变是在继承自 *tornado.websocket.WebSocketHandler* 的 *StatusHandler* 类中的。WebSocket 处理函数实现了 *open* 和 *on\_message* 方法，分别在连接打开和接收到消息时被调用，而不是为每个 HTTP 方法实现处理函数。此外，*on\_close* 方法在连接被远程主机关闭时被调用。

```

class StatusHandler(tornado.websocket.WebSocketHandler):
    def open(self):
        self.application.shoppingCart.register(self.callback)

    def on_close(self):
        self.application.shoppingCart.unregister(self.callback)

    def on_message(self, message):
        pass

    def callback(self, count):
        self.write_message('{"inventoryCount": "%d"}' % count)

```

在实现中，我们在新连接打开时使用 *ShoppingCart* 类注册了 *callback* 方法，并在连接关闭时注销了这个回调函数。因为我们依然使用了 *CartHandler* 类的 HTTP API 调用，因此不需要监听 WebSocket 连接中的新消息，所以 *on\_message* 实现是空的。（我们覆写了 *on\_message* 的默认实现以防止在我们接收消息时 Tornado 抛出 *NotImplementedError* 异常。）最后，*callback* 方法在库存改变时向 WebSocket 连接写消息内容。

这个版本的 JavaScript 代码和之前的非常相似。我们只需要改变其中的 *requestInventory* 函数。我们使用 HTML5 WebSocket API 取代长轮询资源的 AJAX 请求。参见代码清单 5-8。

代码清单 5-8 WebSockets: *inventory.js* 中新的 *requestInventory* 函数

```

function requestInventory() {

```

```

var host = 'ws://localhost:8000/cart/status';

var websocket = new WebSocket(host);

websocket.onopen = function (evt) { };
websocket.onmessage = function(evt) {
    $('#count').html($.parseJSON(evt.data)['inventoryCount']);
};
websocket.onerror = function (evt) { };
}

```

在创建了一个到 `ws://localhost:8000/cart/status` 的 WebSocket 连接后，我们为每个希望响应的事件添加了处理函数。在这个例子中我们唯一关心的事件是 `onmessage`，和之前版本的 `requestInventory` 函数一样更新 `count` 的内容。（轻微的不同是我们必须手工解析服务器送来的 JSON 对象。）

就像前面的例子一样，在购物者添加书籍到购物车时库存量会实时更新。不同之处在于一个持久的 WebSocket 连接取代了每次长轮询更新中重新打开的 HTTP 请求。

### 5.3.3 WebSockets 的未来<sup>[1]</sup>

WebSocket 协议目前仍是草案，在它完成时可能还会修改。然而，因为这个规范已经被提交到 IETF 进行最终审查，相对而言不太可能会再面临重大的改变。正如本节开头所提到的那样，WebSocket 的主要缺陷是目前只支持最新的一些浏览器。

尽管有上述警告，WebSockets 仍然是在浏览器和服务器之间实现双向通信的一个有前途的新方法。当协议得到了广泛的支持后，我们将开始看到更加著名的应用的实现。

[1] 书中网页已不存在，替换为当前网址。

[2] 下面的这组代码书中使用的不是前面的代码，这里为了保持一致修改为和前面的代码一样。

## 第六章：编写安全应用<sup>[1]</sup>

很多时候，安全应用是以牺牲复杂度（以及开发者的头痛）为代价的。Tornado Web 服务器从设计之初就在安全方面有了很多考虑，使其能够更容易地防范那些常见的漏洞。安全 cookies 防止用户的本地状态被其浏览器中的恶意代码暗中修改。此外，浏览器 cookies 可以与 HTTP 请求参数值作比较来防范跨站请求伪造攻击。在本章中，我们将看到使防范这些漏洞更简单的 Tornado 功能，以及使用这些功能的一个用户验证示例。

### 6.1 Cookie 漏洞<sup>[1]</sup>

许多网站使用浏览器 cookies 来存储浏览器会话间的用户标识。这是一个简单而又被广泛兼容的方式来存储跨浏览器会话的持久状态。不幸的是，浏览器 cookies 容易受到一些常见的攻击。本节将展示 Tornado 是如何防止一个恶意脚本来篡改你应用存储的 cookies 的。

### 6.1.1 Cookie 伪造

有很多方式可以在浏览器中截获 cookies。JavaScript 和 Flash 对于它们所执行的页面的域有读写 cookies 的权限。浏览器插件也可由编程方法访问这些数据。跨站脚本攻击可以利用这些访问来修改访客浏览器中 cookies 的值。

### 6.1.2 安全 Cookies

Tornado 的安全 cookies 使用加密签名来验证 cookies 的值没有被服务器软件以外的任何人修改过。因为一个恶意脚本并不知道安全密钥，所以它不能在应用不知情时修改 cookies。

#### 6.1.2.1 使用安全 Cookies

Tornado 的 `set_secure_cookie()` 和 `get_secure_cookie()` 函数发送和取得浏览器的 cookies，以防范浏览器中的恶意修改。为了使用这些函数，你必须在应用的构造函数中指定 `cookie_secret` 参数。让我们来看一个简单的例子。

代码清单 6-1 中的应用将渲染一个统计浏览器中页面被加载次数的页面。如果没有设置 cookie（或者 cookie 已经被篡改了），应用将设置一个值为 1 的新 cookie。否则，应用将从 cookie 中读到的值加 1。

代码清单 6-1 安全 Cookie 示例：cookie\_counter.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.web
import tornado.options

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        cookie = self.get_secure_cookie("count")
        count = int(cookie) + 1 if cookie else 1

        countString = "1 time" if count == 1 else "%d times" % count

        self.set_secure_cookie("count", str(count))

        self.write(
            '<html><head><title>Cookie Counter</title></head>'
            '<body><h1>You' ve viewed this page %s times.</h1>' % countString +
            '</body></html>'
        )

if __name__ == "__main__":
    tornado.options.parse_command_line()
```

```

settings = {
    "cookie_secret": "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E="
}

application = tornado.web.Application([
    (r'/', MainHandler)
], **settings)

http_server = tornado.httpserver.HTTPServer(application)
http_server.listen(options.port)
tornado.ioloop.IOLoop.instance().start()

```

如果你检查浏览器中的 cookie 值，会发现 *count* 储存的值类似于 *MQ==/1310335926/8ef174ecc489ea963c5cdc26ab6d41b49502f2e2*。Tornado 将 cookie 值编码为 Base-64 字符串，并添加了一个时间戳和一个 cookie 内容的 HMAC 签名。如果 cookie 的时间戳太旧（或来自未来），或签名和期望值不匹配，*get\_secure\_cookie()* 函数会认为 cookie 已经被篡改，并返回 *None*，就好像 cookie 从没设置过一样。

传递给 *Application* 构造函数的 *cookie\_secret* 值应该是唯一的随机字符串。在 Python shell 下执行下面的代码片段将产生一个你自己的值：

```

>>> import base64, uuid
>>> base64.b64encode(uuid.uuid4().bytes + uuid.uuid4().bytes)
'bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E='

```

然而，Tornado 的安全 cookies 仍然容易被窃听。攻击者可能会通过脚本或浏览器插件截获 cookies，或者干脆窃听未加密的网络数据。记住 cookie 值是签名的而不是加密的。恶意程序能够读取已存储的 cookies，并且可以传输他们的数据到任意服务器，或者通过发送没有修改的数据给应用伪造请求。因此，避免在浏览器 cookie 中存储敏感的用户数据是非常重要的。

我们还需要注意用户可能修改他自己的 cookies 的可能性，这会导致提权攻击。比如，如果我们在 cookie 中存储了用户已付费的文章剩余的浏览数，我们希望防止用户自己更新其中的数值来获取免费的内容。*httponly* 和 *secure* 属性可以帮助我们防范这种攻击。

#### 6.1.2.2 HTTP-Only 和 SSL Cookies

Tornado 的 cookie 功能依附于 Python 内建的 *Cookie* 模块。因此，我们可以利用它所提供的一些安全功能。这些安全属性是 HTTP cookie 规范的一部分，并在它可能是如何暴露其值给它连接的服务器和它运行的脚本方面给予浏览器指导。比如，我们可以通过只允许 SSL 连接的方式减少 cookie 值在网络中被截获的可能性。我们也可以让浏览器对 JavaScript 隐藏 cookie 值。

为 cookie 设置 *secure* 属性来指示浏览器只通过 SSL 连接传递 cookie。（这可能会产生一些困扰，但这不是 Tornado 的安全 cookies，更精确的说那种方法应该被称为签名 cookies。）从 Python 2.6 版本开始，*Cookie* 对象还提供了一个 *httponly* 属性。包括这个属性指示浏览器对于 JavaScript 不可访问 cookie，这可以防范来自读取 cookie 值的跨站脚本攻击。

为了开启这些功能，你可以向 *set\_cookie* 和 *set\_secure\_cookie* 方法传递关键字参数。比如，一个安全的 HTTP-only cookie（不是 Tornado 的签名 cookie）可以调用 *self.set\_cookie('foo', 'bar', httponly=True, secure=True)* 发送。

既然我们已经探讨了一些保护存储在 cookies 中的持久数据的策略，下面让我们看看另一种常见的攻击载体。下一节我们将看到一种防范向你的应用发送伪造请求的恶意网站。

## 6.2 请求漏洞

任何 Web 应用所面临的一个主要安全漏洞是跨站请求伪造，通常被简写为 CSRF 或 XSRF，发音为“sea surf”。这个漏洞利用了浏览器的一个允许恶意攻击者在受害者网站注入脚本使未授权请求代表一个已登录用户的安全漏洞。让我们看一个例子。

### 6.2.1 剖析一个 XSRF

假设 Alice 是 Burt's Books 的一个普通顾客。当她在这个在线商店登录帐号后，网站使用一个浏览器 cookie 标识她。现在假设一个不择手段的作者，Melvin，想增加他图书的销量。在一个 Alice 经常访问的 Web 论坛中，他发表了一个带有 HTML 图像标签的条目，其源码初始化为在线商店购物的 URL。比如：

```

```

Alice 的浏览器尝试获取这个图像资源，并且在请求中包含一个合法的 cookies，并不知道取代小猫照片的是在线商店的购物 URL。

### 6.2.2 防范请求伪造

有很多预防措施可以防止这种类型的攻击。首先你在开发应用时需要深谋远虑。任何会产生副作用的 HTTP 请求，比如点击购买按钮、编辑账户设置、改变密码或删除文档，都应该使用 HTTP *POST* 方法。无论如何，这是良好的 RESTful 做法，但它也有额外的优势用于防范像我们刚才看到的恶意图像那样琐碎的 XSRF 攻击。但是，这并不足够：一个恶意站点可能会通过其他手段，如 HTML 表单或 XMLHttpRequest API 来向你的应用发送 *POST* 请求。保护 *POST* 请求需要额外的策略。

为了防范伪造 *POST* 请求，我们会要求每个请求包括一个参数值作为令牌来匹配存储在 cookie 中的对应值。我们的应用将通过一个 cookie 头和一个隐藏的 HTML 表单元素向页面提供令牌。当一个合法页面的表单被提交时，它将包括表单值和已存储的 cookie。如果两者匹配，我们的应用认定请求有效。

由于第三方站点没有访问 cookie 数据的权限，他们将不能在请求中包含令牌 cookie。这有效地防止了不可信网站发送未授权的请求。正如我们看到的，Tornado 同样会让这个实现变得简单。

### 6.2.3 使用 Tornado 的 XSRF 保护

你可以通过在应用的构造函数中包含 *xsrp\_cookies* 参数来开启 XSRF 保护：

```
settings = {
    "cookie_secret": "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E=",
    "xsrp_cookies": True
}
application = tornado.web.Application([
```

```
(r'/', MainHandler),
(r' /purchase', PurchaseHandler),
], **settings)
```

当这个应用标识被设置时，Tornado 将拒绝请求参数中不包含正确的 `_xsrf` 值的 `POST`、`PUT` 和 `DELETE` 请求。Tornado 将会在幕后处理 `_xsrf` cookies，但你必须在你的 HTML 表单中包含 XSRF 令牌以确保授权合法请求。要做到这一点，只需要在你的模板中包含一个 `xsrf_form_html` 调用即可：

```
<form action="/purchase" method="POST">
    {% raw xsrf_form_html() %}
    <input type="text" name="title" />
    <input type="text" name="quantity" />
    <input type="submit" value="Check Out" />
</form>
```

### 6.2.3.1 XSRF 令牌和 AJAX 请求

AJAX 请求也需要一个 `_xsrf` 参数，但不是必须显式地在渲染页面时包含一个 `_xsrf` 值，而是通过脚本在客户端查询浏览器获得 cookie 值。下面的两个函数透明地添加令牌值给 AJAX `POST` 请求。第一个函数通过名字获取 cookie，而第二个函数是一个添加 `_xsrf` 参数到传递给 `postJSON` 函数数据对象的便捷函数。

```
function getCookie(name) {
    var c = document.cookie.match("\\b" + name + "=([^;]*)\\b");
    return c ? c[1] : undefined;
}

jQuery.postJSON = function(url, data, callback) {
    data._xsrf = getCookie("_xsrf");
    jQuery.ajax({
        url: url,
        data: jQuery.param(data),
        dataType: "json",
        type: "POST",
        success: callback
    });
}
```

这些预防措施需要思考很多，而 Tornado 的安全 cookies 支持和 XSRF 保护减轻了应用开发者的一些负担。可以肯定的是，内建的安全功能也非常有用，但在思考你应用的安全性方面需要时刻保持警惕。有很多在线 Web 应用安全文献，其中一个更全面的实践对策集合是 Mozilla 的[安全编程指南](#)。

## 6.3 用户验证

既然我们已经看到了如何安全地设置和取得 cookies，并理解了 XSRF 攻击背后的原理，现在就让我们看一个简单用户验证系统的演示示例。在本节中，我们将建立一个应用，询问访客的名字，然后将其存储在安全 cookie 中，以便之后取出。后续的请求将认出回答，并展示给她一个定制的页面。你将学到 `login_url` 参数和 `tornado.web.authenticated` 装饰器的相关知识，这将消除在类似应用中经常会涉及到的一些头疼的问题。



### 6.3.1 示例：欢迎回来

在这个例子中，我们将只通过存储在安全 cookie 里的用户名标识一个人。当某人首次在某浏览器（或 cookie 过期后）访问我们的页面时，我们展示一个登录表单页面。表单作为到 *LoginHandler* 路由的 *POST* 请求被提交。*post* 方法的主体调用 *set\_secure\_cookie()* 来存储 *username* 请求参数中提交的值。

代码清单 6-2 中的 Tornado 应用展示了我们本节要讨论的验证函数。*LoginHandler* 类渲染登录表单并设置 cookie，而 *LogoutHandler* 类删除 cookie。

代码清单 6-2 验证访客：cookies.py

```
import tornado.httpserver
import tornado.ioloop
import tornado.web
import tornado.options
import os.path

from tornado.options import define, options
define("port", default=8000, help="run on the given port", type=int)

class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("username")

class LoginHandler(BaseHandler):
    def get(self):
        self.render('login.html')

    def post(self):
        self.set_secure_cookie("username", self.get_argument("username"))
        self.redirect("/")

class WelcomeHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
        self.render('index.html', user=self.current_user)

class LogoutHandler(BaseHandler):
    def get(self):
        if (self.get_argument("logout", None)):
            self.clear_cookie("username")
            self.redirect("/")

if __name__ == "__main__":
    tornado.options.parse_command_line()

    settings = {
        "template_path": os.path.join(os.path.dirname(__file__), "templates"),
```

```

        "cookie_secret": "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E=",
        "xsrif_cookies": True,
        "login_url": "/login"
    }

    application = tornado.web.Application([
        (r'/', WelcomeHandler),
        (r'/login', LoginHandler),
        (r'/logout', LogoutHandler)
    ], **settings)

    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.instance().start()

```

代码清单 6-3 和 6-4 是应用 templates/ 目录下的文件。

代码清单 6-3 登录表单: login.html

```

<html>
    <head>
        <title>Please Log In</title>
    </head>

    <body>
        <form action="/login" method="POST">
            {% raw xsrf_form_html() %}
            Username: <input type="text" name="username" />
            <input type="submit" value="Log In" />
        </form>
    </body>
</html>

```

代码清单 6-4 欢迎回答: index.html

```

<html>
    <head>
        <title>Welcome Back!</title>
    </head>
    <body>
        <h1>Welcome back, {{ user }}</h1>
    </body>
</html>

```

### 6.3.2 authenticated 装饰器

为了使用 Tornado 的认证功能，我们需要对登录用户标记具体的处理函数。我们可以使用 *@tornado.web.authenticated* 装饰器完成它。当我们使用这个装饰器包裹一个处理方法时，Tornado 将确保这个方法主体只有在合法的用户被发现时才会调用。让我们看看例子中的 *WelcomeHandler* 吧，这个类只对已登录用户渲染 index.html 模板。



```
class WelcomeHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
        self.render('index.html', user=self.current_user)
```

在 *get* 方法被调用之前，*authenticated* 装饰器确保 *current\_user* 属性有值。（我们将简短的讨论这个属性。）如果 *current\_user* 值为假（*None*、*False*、*0*、*""*），任何 *GET* 或 *HEAD* 请求都将把访客重定向到应用设置中 *login\_url* 指定的 URL。此外，非法用户的 *POST* 请求将返回一个带有 403（Forbidden）状态的 HTTP 响应。

如果发现了一个合法的用户，Tornado 将如期调用处理方法。为了实现完整功能，*authenticated* 装饰器依赖于 *current\_user* 属性和 *login\_url* 设置，我们将在下面看到具体讲解。

### 6.3.2.1 current\_user 属性

请求处理类有一个 *current\_user* 属性（同样也在处理程序渲染的任何模板中可用）可以用来存储为当前请求进行用户验证的标识。其默认值为 *None*。为了 *authenticated* 装饰器能够成功标识一个已认证用户，你必须覆写请求处理程序中默认的 *get\_current\_user()* 方法来返回当前用户。

实际的实现由你决定，不过在这个例子中，我们只是从安全 cookie 中取出访客的姓名。很明显，你希望使用一个更加鲁棒的技术，但是出于演示的目的，我们将使用下面的方法：

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("username")
```

尽管这里讨论的例子并没有在存储和取出用户密码或其他凭证上有所深入，但本章中讨论的技术可以以最小的额外努力来扩展到查询数据库中的认证。

### 6.3.2.2 login\_url 设置

让我们简单看看应用的构造函数。记住这里我们传递了一个新的设置给应用：*login\_url* 是应用登录表单的地址。如果 *get\_current\_user* 方法返回了一个假值，带有 *authenticated* 装饰器的处理程序将重定向浏览器的 URL 以便登录。

```
settings = {
    "template_path": os.path.join(os.path.dirname(__file__), "templates"),
    "cookie_secret": "bZJc2sWbQLKos6GkHn/VB9oXwQt8S0R0kRvJ5/xJ89E=",
    "xsrif_cookies": True,
    "login_url": "/login"
}
application = tornado.web.Application([
    (r'/', WelcomeHandler),
    (r'/login', LoginHandler),
    (r'/logout', LogoutHandler)
], **settings)
```

当 Tornado 构建重定向 URL 时，它还会给查询字符串添加一个 *next* 参数，其中包含了发起重定向到登录页面的 URL 资源地址。你可以使用像 *self.redirect(self.get\_argument('next', '/'))* 这样的行来重定向登录后用户回到的页面。

## 6.4 总结¶

我们在本章中看到了两种帮助你的 Tornado 应用安全的技术，以及一个如何使用 `@tornado.web.authenticated` 实现用户认证的例子。在[第七章](#)，我们将看到在那些像 Facebook 和 Twitter 一样需要外部 Web 服务认证的应用中如何扩展我们这里谈论的概念。

## 第七章：外部服务认证¶

[第六章](#)的例子像我们展示了如何使用安全 cookies 和 `tornado.web.authenticated` 装饰器来实现一个简单的用户验证表单。在本章中，我们将着眼于如何对第三方服务进行身份验证。流行的 Web API，比如 Facebook 和 Twitter，使用 OAuth 协议安全验证某人的身份，同时允许他们的用户保持第三方应用访问他们个人信息的控制权。Tornado 提供了一些 Python mix-in 来帮助开发者验证外部服务，既包括显式地支持流行服务，也包括通过通用的 OAuth 支持。在本章中，我们将探讨两个使用 Tornado 的 `auth` 模块的示例应用：一个连接 Twitter，另一个连接 Facebook。

### 7.1 Tornado 的 auth 模块¶

作为一个 Web 应用开发者，你可能想让用户直接通过你的应用在 Twitter 上发表更新或读取最新的 Facebook 状态。大多数社交网络和单一登录的 API 为验证你应用中的用户提供了一个标准的流程。Tornado 的 `auth` 模块为 OpenID、OAuth、OAuth 2.0、Twitter、FriendFeed、Google OpenID、Facebook REST API 和 Facebook Graph API 提供了相应的类。尽管你可以自己实现对于特定外部服务认证过程的处理，不过 Tornado 的 `auth` 模块为连接任何支持的服务开发应用提供了简单的工作流程。

#### 7.1.1 认证流程¶

这些认证方法的工作流程虽然有一些轻微的不同，但对于大多数而言，都使用了 `authorize_redirect` 和 `get_authenticated_user` 方法。`authorize_redirect` 方法用来将一个未授权用户重定向到外部服务的验证页面。在验证页面中，用户登录服务，并让你的应用拥有访问他账户的权限。通常情况下，你会在用户带着一个临时访问码返回你的应用时使用 `get_authenticated_user` 方法。调用 `get_authenticated_user` 方法会把授权跳转过程提供的临时凭证替换成属于用户的长期凭证。Twitter、Facebook、FriendFeed 和 Google 的具体验证类提供了他们自己的函数来使 API 调用它们的服务。

#### 7.1.2 异步请求¶

关于 `auth` 模块需要注意的一件事是它使用了 Tornado 的异步 HTTP 请求。正如我们在[第五章](#)所看到的，异步 HTTP 请求允许 Tornado 服务器在一个挂起的请求等待传出请求返回时处理传入的请求。

我们将简单的看下如何使用异步请求，然后在一个例子中使用它们进行深入。每个发起异步调用的处理方法必须在它前面加上 `@tornado.web.asynchronous` 装饰器。

### 7.2 示例：登录 Twitter¶

让我们来看一个使用 Twitter API 验证用户的例子。这个应用将重定向一个没有登录的用户到 Twitter 的验证页面，提示用户输入用户名和密码。然后 Twitter 会将用户重定向到你在 Twitter 应用设置页指定的 URL。

首先，你必须在 Twitter 注册一个新应用。如果你还没有应用，可以从 [Twitter 开发者网站](#) 的 “Create a new application” 链接开始。一旦你创建了你的 Twitter 应用，你将被指定一个 access token 和一个 secret 来标识你在 Twitter 上的应用。你需要在本节下面代码的合适位置填充那些值。

现在让我们看看代码清单 7-1 中的代码。

代码清单 7-1 查看 Twitter 时间轴：twitter.py

```
import tornado.web
import tornado.httpserver
import tornado.auth
import tornado.ioloop

class TwitterHandler(tornado.web.RequestHandler, tornado.auth.TwitterMixin):
    @tornado.web.asynchronous
    def get(self):
        oAuthToken = self.get_secure_cookie('oauth_token')
        oAuthSecret = self.get_secure_cookie('oauth_secret')
        userID = self.get_secure_cookie('user_id')

        if self.get_argument('oauth_token', None):
            self.get_authenticated_user(self.async_callback(self._twitter_on_auth))
            return

        elif oAuthToken and oAuthSecret:
            accessToken = {
                'key': oAuthToken,
                'secret': oAuthSecret
            }
            self.twitter_request('/users/show',
                                access_token=accessToken,
                                user_id=userID,
                                callback=self.async_callback(self._twitter_on_user)
            )
            return

        self.authorize_redirect()

    def _twitter_on_auth(self, user):
        if not user:
            self.clear_all_cookies()
            raise tornado.web.HTTPError(500, 'Twitter authentication failed')

        self.set_secure_cookie('user_id', str(user['id']))
        self.set_secure_cookie('oauth_token', user['access_token']['key'])
```

```

        self.set_secure_cookie('oauth_secret', user['access_token']['secret'])

        self.redirect('/')

    def _twitter_on_user(self, user):
        if not user:
            self.clear_all_cookies()
            raise tornado.web.HTTPError(500, "Couldn't retrieve user information")

        self.render('home.html', user=user)

class LogoutHandler(tornado.web.RequestHandler):
    def get(self):
        self.clear_all_cookies()
        self.render('logout.html')

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [
            (r'/', TwitterHandler),
            (r'/logout', LogoutHandler)
        ]

        settings = {
            'twitter_consumer_key': 'cWc3 ... d3yg',
            'twitter_consumer_secret': 'nEoT ... cCXB4',
            'cookie_secret': 'NTli0TY5NzJkYTVlMTU00TAwMTdlNjgzMTA5M2U3OGQ5NDIxZmU3Mg==',
            'template_path': 'templates',
        }

        tornado.web.Application.__init__(self, handlers, **settings)

if __name__ == '__main__':
    app = Application()
    server = tornado.httpserver.HTTPServer(app)
    server.listen(8000)
    tornado.ioloop.IOLoop.instance().start()

```

代码清单 7-2 和 7-3 的模板文件应该被放在应用的 templates 目录下。

代码清单 7-2 Twitter 时间轴: home.html

```

<html>
  <head>
    <title>{{ user['name'] }} ({{ user['screen_name'] }}) on Twitter</title>
  </head>

  <body>
    <div>
      <a href="/logout">Sign out</a>
    </div>
  </body>
</html>

```

```

</div>
<div>
    
    <h2>About @{{ user['screen_name'] }}</h2>
    <p style="clear:both"><em>{{ user['description'] }}</em></p>
</div>
<div>
    <ul>
        <li>{{ user['statuses_count'] }} tweets.</li>
        <li>{{ user['followers_count'] }} followers.</li>
        <li>Following {{ user['friends_count'] }} users.</li>
    </ul>
</div>
{% if 'status' in user %}
    <hr />
    <div>
        <p>
            <strong>{{ user['screen_name'] }}</strong>
            <em>on {{ ' '.join(user['status']['created_at'].split()[:2]) }}
                at {{ user['status']['created_at'].split()[3] }}</em>
        </p>
        <p>{{ user['status']['text'] }}</p>
    </div>
{% end %}
</body>
</html>

```

代码清单 7-3 Twitter 时间轴: logout.html

```

<html>
    <head>
        <title>Tornadoes on Twitter</title>
    </head>

    <body>
        <div>
            <h2>You have successfully signed out.</h2>
            <a href="/">Sign in</a>
        </div>
    </body>
</html>

```

让我们分块进行分析，首先从 `twitter.py` 开始。在 `Application` 类的 `__init__` 方法中，你将注意到有两个新的键出现在设置字典中：`twitter_consumer_key` 和 `twitter_consumer_secret`。它们需要被设置为你的 Twitter 应用详细设置页面中列出的值。同样，你还会注意到我们声明了两个处理程序：`TwitterHandler` 和 `LogoutHandler`。让我们立刻看看这两个类吧。

`TwitterHandler` 类包含我们应用逻辑的主要部分。有两件事情需要立刻引起我们的注意，其一是这个类继承自能给我们提供 Twitter 功能的 `tornado.auth.TwitterMixin` 类，其二是 `get` 方法使用了我们在[第五章](#)中讨论的 `@tornado.web.asynchronous` 装饰器。现在让我们看看第一个异步调用：

```

if self.get_argument('oauth_token', None):
    self.get_authenticated_user(self.async_callback(self._twitter_on_auth))
    return

```

当一个用户请求我们应用的根目录时，我们首先检查请求是否包括一个 *oauth\_token* 查询字符串参数。如果有，我们把这个请求看作是一个来自 Twitter 验证过程的回调。

然后，我们使用 *auth* 模块的 *get\_authenticated* 方法把给我们的临时令牌换为用户的访问令牌。这个方法期待一个回调函数作为参数，在这里是 *self.\_twitter\_on\_auth* 方法。当到 Twitter 的 API 请求返回时，执行回调函数，我们在代码更靠下的地方对其进行了定义。

如果 *oauth\_token* 参数没有被发现，我们继续测试是否之前已经看到过这个特定用户了。

```

elif OAuthToken and OAuthSecret:
    accessToken = {
        'key': OAuthToken,
        'secret': OAuthSecret
    }
    self.twitter_request('/users/show',
        access_token=accessToken,
        user_id=userID,
        callback=self.async_callback(self._twitter_on_user)
    )
    return

```

这段代码片段寻找我们应用在 Twitter 给定一个合法用户时设置的 *access\_key* 和 *access\_secret* cookies。如何这个值被设置了，我们就用 *key* 和 *secret* 组装访问令牌，然后使用 *self.twitter\_request* 方法来向 Twitter API 的 */users/show* 发出请求。在这里，你会再一次看到异步回调函数，这次是我们稍后将要定义的 *self.\_twitter\_on\_user* 方法。

*twitter\_quest* 方法期待一个路径地址作为它的第一个参数，另外还有一些可选的关键字参数，如 *access\_token*、*post\_args* 和 *callback*。*access\_token* 参数应该是一个字典，包括用户 OAuth 访问令牌的 *key* 键，和用户 OAuth secret 的 *secret* 键。

如果 API 调用使用了 *POST* 方法，请求参数需要绑定一个传递 *post\_args* 参数的字典。查询字符串参数在方法调用时只需指定为一个额外的关键字参数。在 */users/show* API 调用时，我们使用了 HTTP *GET* 请求，所以这里不需要 *post\_args* 参数，而所需的 *user\_id* API 参数被作为关键字参数传递进来。

如果上面我们讨论的情况都没有发生，这说明用户是首次访问我们的应用（或者已经注销或删除了 cookies），此时我们想将其重定向到 Twitter 的验证页面。调用 *self.authorize\_redirect()* 来完成这项工作。

```

def _twitter_on_auth(self, user):
    if not user:
        self.clear_all_cookies()
        raise tornado.web.HTTPError(500, 'Twitter authentication failed')

    self.set_secure_cookie('user_id', str(user['id']))
    self.set_secure_cookie('oauth_token', user['access_token']['key'])

```

```
self.set_secure_cookie('oauth_secret', user['access_token']['secret'])
```

```
self.redirect('/')
```

我们的 Twitter 请求的回调方法非常的直接。`_twitter_on_auth` 使用一个 `user` 参数进行调用，这个参数是已授权用户的用户数据字典。我们的方法实现只需要验证我们接收到的用户是否合法，并设置应有的 cookies。一旦 cookies 被设置好，我们将用户重定向到根目录，即我们之前谈论的发起请求到 `/users/show` API 方法。

```
def _twitter_on_user(self, user):
    if not user:
        self.clear_all_cookies()
        raise tornado.web.HTTPError(500, "Couldn't retrieve user information")

    self.render('home.html', user=user)
```

`_twitter_on_user` 方法是在 `twitter_request` 方法中指定调用的回调函数。当 Twitter 响应用户的个人信思时，我们的回调函数使用响应的数据渲染 `home.html` 模板。这个模板展示了用户的个人图像、用户名、详细信息、一些关注和粉丝的统计信息以及用户最新的状态更新。

`LogoutHandler` 方法只是清除了我们为应用用户存储的 cookies。它渲染了 `logout.html` 模板，来给用户反馈，并跳转到 Twitter 验证页面允许其重新登录。就是这些！

我们刚才看到的 Twitter 应用只是为一个授权用户展示了用户信息，但它同时也说明了 Tornado 的 `auth` 模块是如何使开发社交应用更简单的。创建一个在 Twitter 上发表状态的应用作为一个练习留给读者。

## 7.3 示例：Facebook 认证和 Graph API

Facebook 的这个例子在结构上和刚才看到的 Twitter 的例子非常相似。Facebook 有两种不同的 API 标准，原始的 REST API 和 Facebook Graph API。目前两种 API 都被支持，但 Graph API 被推荐作为开发新 Facebook 应用的方式。Tornado 在 `auth` 模块中支持这两种 API，但在这个例子中我们将关注 Graph API。

为了开始这个例子，你需要登录到 Facebook 的[开发者网站](https://developers.facebook.com)，并创建一个新的应用。你将需要填写应用的名称，并证明你不是一个机器人。为了从你自己的域名中验证用户，你还需要指定你应用的域名。然后点击“Select how your app integrates with Facebook”下的“Website”。同时你需要输入你网站的 URL。要获得完整的创建 Facebook 应用的手册，可以从<https://developers.facebook.com/docs/guides/web/>开始。

你的应用建立好之后，你将使用基本设置页面的应用 ID 和 secret 来连接 Facebook Graph API。

回想一下上一节的提到的单一登录工作流程，它将引导用户到 Facebook 平台验证应用，Facebook 将使用一个 HTTP 重定向将一个带有验证码的用户返回给你的服务器。一旦你接收到含有这个验证码的请求，你必须请求用于标识 API 请求用户身份的验证令牌。

这个例子将渲染用户的时间轴，并允许用户通过我们的接口更新她的 Facebook 状态。让我们看下代码清单 7-4。



代码清单 7-4 Facebook 验证: facebook.py

```
import tornado.web
import tornado.httpserver
import tornado.auth
import tornado.ioloop
import tornado.options
from datetime import datetime

class FeedHandler(tornado.web.RequestHandler, tornado.auth.FacebookGraphMixin):
    @tornado.web.asynchronous
    def get(self):
        accessToken = self.get_secure_cookie('access_token')
        if not accessToken:
            self.redirect('/auth/login')
            return

        self.facebook_request(
            "/me/feed",
            access_token=accessToken,
            callback=self.async_callback(self._on_facebook_user_feed))

    def _on_facebook_user_feed(self, response):
        name = self.get_secure_cookie('user_name')
        self.render('home.html', feed=response['data'] if response else [], name=name)

    @tornado.web.asynchronous
    def post(self):
        accessToken = self.get_secure_cookie('access_token')
        if not accessToken:
            self.redirect('/auth/login')

        userInput = self.get_argument('message')

        self.facebook_request(
            "/me/feed",
            post_args={'message': userInput},
            access_token=accessToken,
            callback=self.async_callback(self._on_facebook_post_status))

    def _on_facebook_post_status(self, response):
        self.redirect('/')

class LoginHandler(tornado.web.RequestHandler, tornado.auth.FacebookGraphMixin):
    @tornado.web.asynchronous
    def get(self):
        userID = self.get_secure_cookie('user_id')

        if self.get_argument('code', None):
            self.get_authenticated_user(
```

```

        redirect_uri='http://example.com/auth/login',
        client_id=self.settings['facebook_api_key'],
        client_secret=self.settings['facebook_secret'],
        code=self.get_argument('code'),
        callback=self.async_callback(self._on_facebook_login))
    return
elif self.get_secure_cookie('access_token'):
    self.redirect('/')
    return

self.authorize_redirect(
    redirect_uri='http://example.com/auth/login',
    client_id=self.settings['facebook_api_key'],
    extra_params={'scope': 'read_stream,publish_stream'}
)

def _on_facebook_login(self, user):
    if not user:
        self.clear_all_cookies()
        raise tornado.web.HTTPError(500, 'Facebook authentication failed')

    self.set_secure_cookie('user_id', str(user['id']))
    self.set_secure_cookie('user_name', str(user['name']))
    self.set_secure_cookie('access_token', str(user['access_token']))
    self.redirect('/')

class LogoutHandler(tornado.web.RequestHandler):
    def get(self):
        self.clear_all_cookies()
        self.render('logout.html')

class FeedListItem(tornado.web.UIModule):
    def render(self, statusItem):
        dateFormatter = lambda x: datetime.
        strftime(x, '%Y-%m-%dT%H:%M:%S+0000').strftime('%c')
        return self.render_string('entry.html', item=statusItem, format=dateFormatter)

class Application(tornado.web.Application):
    def __init__(self):
        handlers = [
            (r'/', FeedHandler),
            (r'/auth/login', LoginHandler),
            (r'/auth/logout', LogoutHandler)
        ]

        settings = {
            'facebook_api_key': '2040 ... 8759',
            'facebook_secret': 'eae0 ... 2f08',

```

```

        'cookie_secret':
'NTli0TY5NzJkYTVlMTU00TAwMTdlNjgzMTA5M2U3OGQ5NDIxZmU3Mg==',
        'template_path': 'templates',
        'ui_modules': {'FeedListItem': FeedListItem}
    }

    tornado.web.Application.__init__(self, handlers, **settings)

if __name__ == '__main__':
    tornado.options.parse_command_line()

    app = Application()
    server = tornado.httpserver.HTTPServer(app)
    server.listen(8000)
    tornado.ioloop.IOLoop.instance().start()

```

我们将按照访客与应用交互的顺序来讲解这些处理。当请求根 URL 时，*FeedHandler* 将寻找 *access\_token* cookie。如果这个 cookie 不存在，用户会被重定向到 */auth/login* URL。

登录页面使用了 *authorize\_redirect* 方法来讲用户重定向到 Facebook 的验证对话框，如果需要的话，用户在这里登录 Facebook，审查应用程序请求的权限，并批准应用。在点击“Approve”之后，她将被跳转回应用在 *authorize\_redirect* 调用中 *redirect\_uri* 指定的 URL。

当从 Facebook 验证页面返回后，到 */auth/login* 的请求将包括一个 *code* 参数作为查询字符串参数。这个码是一个用于换取永久凭证的临时令牌。如果发现了 *code* 参数，应用将发出一个 Facebook Graph API 请求来取得认证的用户，并存储她的用户 ID、全名和访问令牌，以便在应用发起 Graph API 调用时标识该用户。

存储了这些值之后，用户被重定向到根 URL。用户这次回到根页面时，将取得最新 Facebook 消息列表。应用查看 *access\_cookie* 是否被设置，并使用 *facebook\_request* 方法向 Graph API 请求用户订阅。我们把 OAuth 令牌传递给 *facebook\_request* 方法，此外，这个方法还需要一个回调函数参数——在代码清单 7-4 中，它是 *\_on\_facebook\_user\_feed* 方法。

代码清单 7-5 Facebook 验证：home.html

```

<html>
  <head>
    <title>{{ name }} on Facebook</title>
  </head>

  <body>
    <div>
      <a href="/auth/logout">Sign out</a>
      <h1>{{ name }}</h1>
    </div>
    <div>
      <form action="/facebook/" method="POST">
        <textarea rows="3" cols="50" name="message"></textarea>
        <input type="submit" value="Update Status" />
      </form>
    </div>
  </body>
</html>

```

```

</div>
<hr />
{% for item in feed %}
    {% module FeedListItem(item) %}
{% end %}
</body>
</html>

```

当包含来自 Facebook 的用户订阅消息的响应的回调函数被调用时，应用渲染 `home.html` 模板，其中使用了 `FeedListItem` 这个 UI 模块来渲染列表中的每个条目。在模板开始处，我们渲染了一个表单，可以用 `message` 参数 `post` 到我们服务器的 `/resource`。应用发送这个调用给 Graph API 来发表一个更新。

为了发表更新，我们再次使用了 `facebook_request` 方法。这次，除了 `access_token` 参数之外，我们还包括了一个 `post_args` 参数，这个参数是一个成为 Graph 请求 `post` 主体的参数字典。当调用成功时，我们将用户重定向回首页，并请求更新后的时间轴。

正如你所看到的，Tornado 的 `auth` 模块提供的 Facebook 验证类包括很多构建 Facebook 应用时非常有用的功能。这不仅在原型设计中是一笔巨大的财富，同时也非常适合是生产中的应用。

## 第八章：部署 Tornado

到目前为止，为了简单起见，在我们的例子中都是使用单一的 Tornado 进程运行的。这使得测试应用和快速变更非常简单，但是这不是一个合适的部署策略。部署一个应用到生产环境面临着新的挑战，既包括最优化性能，也包括管理独立进程。本章将介绍强化你的 Tornado 应用、增加请求吞吐量的策略，以及使得部署 Tornado 服务器更容易的工具。

### 8.1 运行多个 Tornado 实例的原因

在大多数情况下，组合一个网页不是一个特别的计算密集型处理。服务器需要解析请求，取得适当的数据，以及将多个组件组装起来进行响应。如果你的应用使用阻塞的调用查询数据库或访问文件系统，那么服务器将不会在等待调用完成时响应传入的请求。在这些情况下，服务器硬件有剩余的 CPU 时间来等待 I/O 操作完成。

鉴于响应一个 HTTP 请求的时间大部分都花费在 CPU 空闲状态下，我们希望利用这个停工时间，最大化给定时间内我们可以处理的请求数量。也就是说，我们希望服务器能够在处理已打开的请求等待数据的过程中接收尽可能多的新请求。

正如我们在[第五章](#)讨论的异步 HTTP 请求中所看到的，Tornado 的非阻塞架构在解决这类问题上大有帮助。回想一下，异步请求允许 Tornado 进程在等待出站请求返回时执行传入的请求。然而，我们碰到的问题是当同步函数调用块时。设想在一个 Tornado 执行的数据库查询或磁盘访问块中，进程不允许回应新的请求。这个问题最简单的解决方法是运行多个解释器的实例。通常情况下，你会使用一个反向代理，比如 Nginx，来非配多个 Tornado 实例的加载。

### 8.2 使用 Nginx 作为反向代理

一个代理服务器是一台中转客户端资源请求到适当的服务器的机器。一些网络安装使用代理服务器过滤或缓存本地网络机器到 Internet 的 HTTP 请求。因为我们将运行一些在不同 TCP 端口上的 Tornado 实例，因此我们将使用反向代理服务器：客户端通过 Internet 连接一个反向代理服务器，然后反向代理服务器发送请求到代理后端的 Tornado 服务器池中的任何一个主机。代理服务器被设置为对客户端透明的，但它会向上游的 Tornado 节点传递一些有用信息，比如原始客户端 IP 地址和 TCP 格式。

我们的服务器配置如图 8-1 所示。反向代理接收所有传入的 HTTP 请求，然后把它们分配给独立的 Tornado 实例。

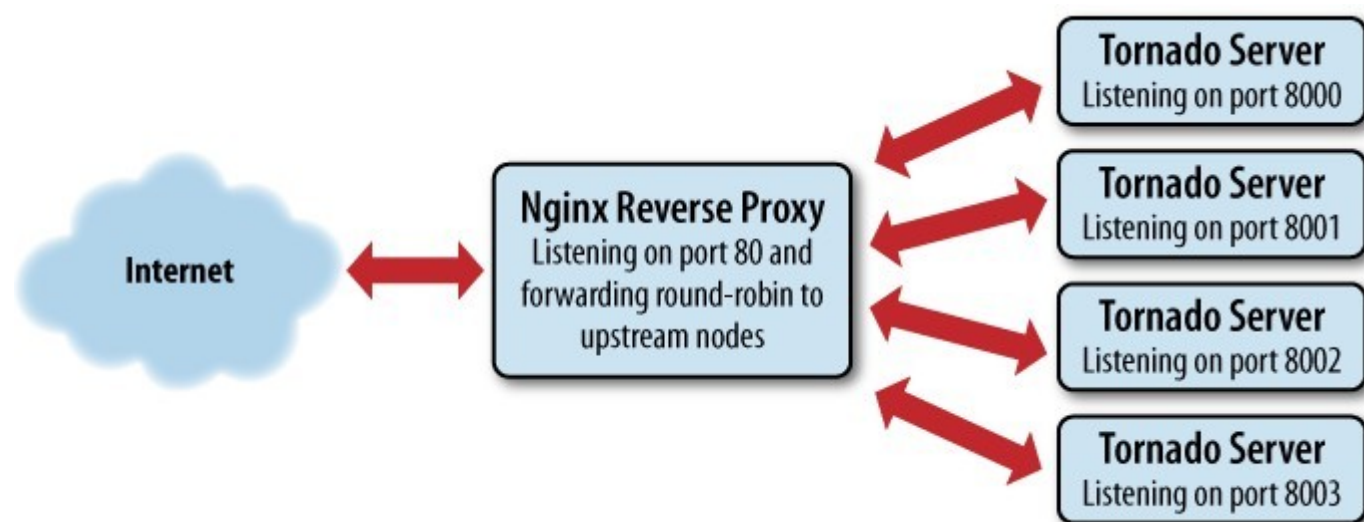


图 8-1 反向代理服务器后端的 Tornado 实例

### 8.2.1 Nginx 基本配置

代码清单 8-1 中的列表是一个 Nginx 配置的示例。Nginx 启动后监听来自 80 端口的连接，然后分配这些请求到配置文件中列出的上游主机。在这种情况下，我们假定上游主机监听来自他们自己的环回接口上的端口的连接。

代码清单 8-1 一个简单的 Nginx 代理配置

```
user nginx;
worker_processes 5;

error_log /var/log/nginx/error.log;

pid /var/run/nginx.pid;

events {
    worker_connections 1024;
    use epoll;
}

proxy_next_upstream error;
```

```

upstream tornadoes {
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
    server 127.0.0.1:8002;
    server 127.0.0.1:8003;
}

server {
    listen 80;
    server_name www.example.org *.example.org;

    location /static/ {
        root /var/www/static;
        if ($query_string) {
            expires max;
        }
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_pass http://tornadoes;
    }
}

```

这个配置示例假定你的系统使用了 `epoll`。在不同的 UNIX 发行版本中经常会有轻微的不同。一些系统可能使用了 `poll`、`/dev/poll` 或 `kqueue` 代替。

按顺序来看匹配 `location /static/` 或 `location /` 的请求可能会很有帮助。Nginx 把位于 `location` 指令中的字符串看作是一个以行起始锚点开始、任何字母重复结束的正则表达式。所以 `/` 被看作是表达式 `^/.*`。当 Nginx 匹配这些字符串时，像 `/static` 这样更加特殊的字符串在像 `/` 这样的更加的通用的字符串之前被检查。Nginx 文档中详细解释了匹配的顺序。

除了一些标准样板外，这个配置文件最重要的部分是 `upstream` 指令和服务器配置中的 `proxy` 指令。Nginx 服务器在 80 端口监听连接，然后分配这种请求给 `upstream` 服务器组中列出的 Tornado 实例。`proxy_pass` 指令指定接收转发请求的服务器 URI。你可以在 `proxy_pass` URI 中的主机部分引用 `upstream` 服务器组的名字。

Nginx 默认以循环的方式分配请求。此外，你也可以选择基于客户端的 IP 地址分配请求，这种情况下（除非连接中断）可以确保来自同一 IP 地址的请求总是被分配到同一个上游节点。你可以在 [HTTPUpstreamModule 文档](#) 中了解更多关于这个选项的知识。

还需要注意的是 `location /static/` 指令，它告诉 Nginx 直接提供静态目录的文件，而不再代理请求到 Tornado。Nginx 可以比 Tornado 更高效地提供静态文件，所以减少 Tornado 进程中不必要的加载是非常有意义的。

## 8.2.2 Nginx 的 SSL 解密

应用的开发者在浏览器和客户端之间传输个人信息时需要特别注意保护信息不要落入坏人之手。在不安全的 WiFi 接入 中，用户很容易受到 cookie 劫持攻击，从而威胁他们在流行的社交网站上的账户。对此，大部分主要的社交网络应用都默认或作为用户可配置选项使用安全协 议。同时，我们使用 Nginx 解密传入的 SSL 加密请求，然后把解码后的 HTTP 请求分配给上游服务器。

代码清单 8-2 展示了一个用于解密传入的 HTTPS 请求的 *server* 块，并使用代码清单 8-1 中我们使用过的代理指令转发解密后的通信。

代码清单 8-2 使用 SSL 的 *server* 块

```
server {
    listen 443;
    ssl on;
    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/cert.key;

    default_type application/octet-stream;

    location /static/ {
        root /var/www/static;
        if ($query_string) {
            expires max;
        }
    }

    location = /favicon.ico {
        rewrite (.* ) /static/favicon.ico;
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_pass http://tornadoes;
    }
}
```

这段代码和上面的配置非常相似，除了 Nginx 将在标准 HTTPS 的 443 端口监听安全 Web 请求外。如果你想强制使用 SSL 连接，你可以在 *server* 块中包含一个 *rewrite* 指令来监听 80 端口的 HTTP 连接。代码清单 8-3 是这种重定向的一个例子。

代码清单 8-3 用于重定向 HTTP 请求到安全渠道的 *server* 块

```
server {
    listen 80;
    server_name example.com;

    rewrite /(.* ) https://$http_host/$1 redirect;
}
```



Nginx 是一个非常鲁棒的工具，我们在这里仅仅接触到帮助 Tornado 部署的一些简单的配置选项。Nginx 的 [wiki 文档](#) 是获得安装和配置这个强有力的工具额外信息的一个非常好的资源。

## 8.3 使用 Supervisor 监控 Tornado 进程

正如 8.2 节中埋下的伏笔，我们将在我们的 Tornado 应用中运行多个实例以充分利用现代的多处理器和多核服务器架构。开发团队大多传闻每个核运行一个 Tornado 进程。但是，正如我们所知道的，大量的传闻并不代表事实，所以你的结果可能不同。在本节中，我们将讨论在 UNIX 系统中管理多个 Tornado 实例的策略。

到目前为止，我们都是命令行中运行 Tornado 服务器的，就像 `$ python main.py --port=8000`。但是，再擦河南刮起的生产部署中，这是不可管理的。因为我们为每个 CPU 核心运行一个独立的 Tornado 进程，因此有很多进程需要监控和控制。supervisor 守护进程可以帮助我们完成这个任务。

Supervisor 的设计是每次开机时启动其配置文件中列出的进程。这里，我们将看到管理我们在 Nginx 配置文件中作为上游主机提到的四个 Tornado 实例的 Supervisor 配置。典型的 `supervisord.conf` 文件中包含了全局的配置指令，并加载 `conf.d` 目录下的其他配置文件。代码清单 8-4 展示了我们想启动的 Tornado 进程的配置文件。

代码清单 8-4 tornado.conf

```
[group:tornadoes]
programs=tornado-8000, tornado-8001, tornado-8002, tornado-8003

[program:tornado-8000]
command=python /var/www/main.py --port=8000
directory=/var/www
user=www-data
autorestart=true
redirect_stderr=true
stdout_logfile=/var/log/tornado.log
loglevel=info

[program:tornado-8001]
command=python /var/www/main.py --port=8001
directory=/var/www
user=www-data
autorestart=true
redirect_stderr=true
stdout_logfile=/var/log/tornado.log
loglevel=info

[program:tornado-8002]
command=python /var/www/main.py --port=8002
directory=/var/www
user=www-data
autorestart=true
redirect_stderr=true
```

```
stdout_logfile=/var/log/tornado.log
loglevel=info

[program:tornado-8003]
command=python /var/www/main.py --port=8003
directory=/var/www
user=www-data
autorestart=true
redirect_stderr=true
stdout_logfile=/var/log/tornado.log
loglevel=info
```

为了 Supervisor 有意义，你需要至少包含一个 *program* 部分。在代码清单 8-4 中，我们定义了四个程序，分别命名为 *tornado-8000* 到 *tornado-8003*。program 部分定义了 Supervisor 将要运行的每个命令的参数。*command* 的值是必须的，通常是带有我们希望监听的 *port* 参数的 Tornado 应用。我们还为每个程序的工作目录、有效用户和日志文件定义了额外的设置；而把 *autorestart* 和 *redirect\_stderr* 设置为 *true* 是非常有用的。

为了一起管理所有的 Tornado 进程，创建一个组是很有必要的。在这个例子的顶部，我们声明了一个叫作 *tornadoes* 的组，并在其中列出了每个程序。现在，当我们想要管理我们的 Tornado 应用时，我们可以通过带有通配符的组名引用所有的组成程序。比如，要重启应用时，我们只需要在 `supervisorctl` 工具中使用命令 `restart tornadoes:*`。

一旦你安装和配置好 Supervisor，你就可以使用 `supervisorctl` 来管理 `supervisord` 进程。为了启动你的 Web 应用，你可以让 Supervisor 重新读取配置，然后任何配置改变的程序或程序组将被重启。你同样可以手动启动、停止和重启被管理的程序或检查整个系统的状态。

```
supervisor> update
tornadoes: stopped
tornadoes: updated process group
supervisor> status
tornadoes:tornado-8000 RUNNING pid 32091, uptime 00:00:02
tornadoes:tornado-8001 RUNNING pid 32092, uptime 00:00:02
tornadoes:tornado-8002 RUNNING pid 32093, uptime 00:00:02
tornadoes:tornado-8003 RUNNING pid 32094, uptime 00:00:02
```

Supervisor 和你系统的初始化进程一起工作，并且它应该在系统启动时自动注册守护进程。当 supervisor 启动后，程序组会自动在线。默认情况下，Supervisor 会监控子进程，并在任何程序意外终止时重生。如果你想不管错误码，重启被管理的进程，你可以设置 *autorestart* 为 *true*。

Supervisor 不只可以使管理多个 Tornado 实例更容易，还能让你在 Tornado 服务器遇到意外的服务中断后重新上线时泰然处之。