CHAPTER

6 TCP

传输控制协议（TCP）是一个面向连接的协议，它保证了两台计算机之间数据传输的可靠性和顺序。

换句话说，TCP是一种传输层协议，它可以让你将数据从一台计算机完整有序地传输到另一台计算机。

正是由于这些特点，很多我们现在使用的如HTTP这样的协议都是基于TCP协议的。当传输一个页面的HTML文件时，肯定是希望它传输到目的地时能够与传输前一致，要是出什么问题，就应该抛出错误。哪怕有一个字符（字节）传输错位了，浏览器都有可能无法渲染这个页面。

Node.js这个框架的出发点就是为了网络应用开发所设计的。如今，网络应用都是用TCP/IP协议进行通信的。所以，了解TCP/IP协议是如何工作的，以及Node.js是如何通过简单的API对其进行封装的，都是非常有帮助的。

首先要介绍的是该协议的特点。举例来说，使用TCP在两台电脑之间进行数据传输是如何做到的。当传输两条消息时，传输到目的地时还能保持发送前的顺序吗？理解协议本身对于理解使用该协议的软件也是很重要的。比如，大部分时候，连接如MySQL等的数据库以及与数

据库进行通信使用的都是TCP套接字。

Node HTTP服务器是构建于Node TCP服务器之上的。从编程角度来说，也就是Node中的`http.Server`继承自`net.Server`（`net`是TCP模块）。

除了Web浏览器和服务器（HTTP）之外，很多我们日常使用的如邮件客户端（SMTP/IMAP/POP）、聊天程序（IRC/XMPP）以及远程 shell（SSH）等都基于TCP协议。

尽可能多地了解TCP以及如何使用相关的Node.js API对书写和理解网络程序会大有裨益。

70 TCP有哪些特性

若只是使用TCP的话，那无须理解它内部的工作原理，以及其实现机制。

不过，理解这些东西对分析更高层的协议和服务器，如Web服务器、数据库等的问题有很大的帮助。

TCP的首要特性就是它是面向连接的。

面向连接的通信和保证顺序的传递

说到TCP，可以将客户端和服务器端的通信看作是一个连接或者数据流。这对开发面向服务的应用和流应用是很好的抽象，因为TCP协议做基于的IP协议是面向无连接的。

IP是基于数据报的传输。这些数据报是独立进行传输的，送达的顺序也是无序的。

那么TCP又是如何保证这些独立的数据报送达的时候是有序的呢？

使用IP协议意味着数据包送达时是无序的，这些数据包不属于任何的数据流或者连接，那么当使用TCP/IP和服务器建立连接后，是怎样做到让数据包送达时是有序的呢？

要回答上述问题其实就等于在解释为什么会有TCP。当在TCP连接内进行数据传递时，发送的IP数据报包含了标识该连接以及数据流顺序的信息。

假设一条消息分割为四个部分。当服务器从连接A收到第一部分和第四部分后，它就知道还要等待其他数据报中的第二部分和第三部分。

要是用Node来写一个TCP服务器，就完全没必要去考虑这些复杂的内部实现了。只要考虑连接以及往套接字中写数据即可。接收方会按序接收到传输的信息，要是发生网络错误，连接会失效或者终止。

面向字节

TCP对字符以及字符编码是完全无知的。正如第4章介绍的，不同的编码会导致传输的字节数不同。

所以，TCP允许数据以ASCII字符（每个字符一个字节）或者 Unicode（即每个字符四个字节）进行传输。

正是因为对消息格式没有严格的约束，使得TCP有很好的灵活性。 71

可靠性

由于TCP是基于底层不可靠的服务，因此，它必须要基于确认和超时实现一系列的机制来达到可靠性的要求。

当数据发送出去后，发送方就会等待一个确认消息（表示数据包已经收到的简短的确认消息）。如果过了指定的窗口时间，还未收到确认消息，发送方就会对数据进行重发。

这种机制有效地解决了如网络错误或者网络阻塞这样的不可预测的情况。

流控制

要是两台互相通信的计算机中，有一台速度远快于另一台的话，会怎么样呢?

TCP会通过一种叫流控制的方式来确保两点之间传输数据的平衡。

拥堵控制

TCP有一种内置的机制能够控制数据包的延迟率及丢包率不会太高，以此来确保服务的质量（QoS）。

举例来说，和流控制机制能够避免发送方压垮接收方一样，TCP会通过控制数据包的传输速率来避免拥堵的情况。

好了，介绍完TCP的基本工作原理，到实践的时候了。为了测试 TCP服务器，我们可以使用Telnet工具。

Telnet

Telnet是一个早期的网络协议，旨在提供双向的虚拟终端。在SSH出现前，它作为一种控制远程计算机的方式被广泛使用，如远程服务器管理。它是TCP协议上层的协议（不要惊讶）。

尽管从21世纪初就基本不用Telnet了，但如今绝大部分主流的操作系统都内置了telnet客户端（见图6-1）：

```
$ telnet
```

绝大部分Telnet使用的是23号端口。要是通过该端口连接到服务器（telnet host.com 23或者就简单地写成telnet host.com），就说明在通过TCP使用Telnet协议。

72

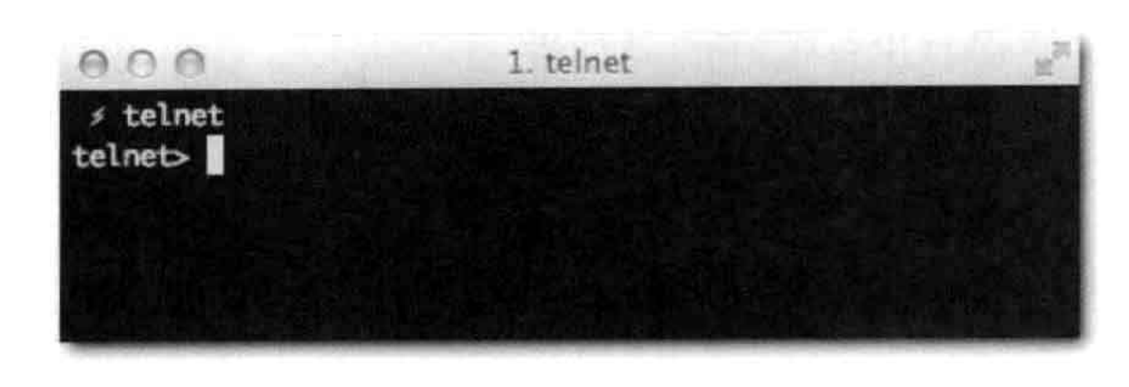


图6-1：运行telnet工具

不过，在本例中，telnet客户端还有更加有意思的功能。发送数据时，客户端要是发现服务器使用的并不是Telnet，这时，它不会关闭连接或者显示错误信息，相反，它会自动降级到低层的纯TCP模式。

所以，要是telnet到Web服务器会怎么样呢？要一探究竟，我们来看下面这个例子。

首先，我们用Node.js来写一个简单的hello world Web服务器，并监听 3000端口：

```
# web-server.js
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<h1>Hello world</h1>');
}).listen(3000);
```

接下来通过node server.js运行上述代码。要确保运行正常，可以打开一个典型的HTTP客户端——浏览器来查看，如图6-2所示。

图6-2：浏览器通过本地3000端口建立了一个TCP连接，然后开始使用HTTP协议进行通信

现在来实现客户端部分。使用telnet来建立一个连接（见图6-3）：

```
$ telnet localhost 3000
```

73

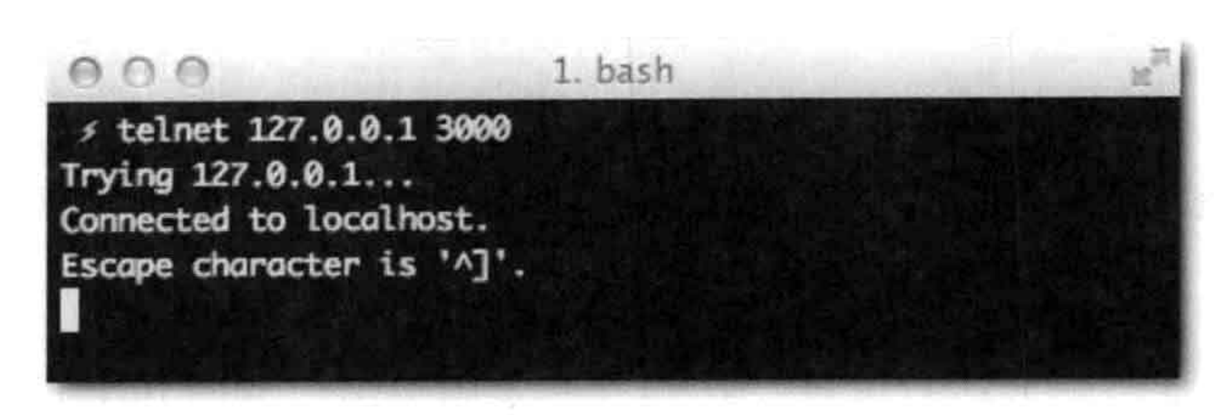


图6-3：telnet允许在终端手动建立一个TCP连接

尽管根据图6-3中所示的结果，看上去已经工作正常了，但是，服务器端的“Hello World”消息并未到客户端这里。原因在于，要往TCP连接中写数据，必须首先创建一个HTTP请求，这就是套接字（socket）。在终端输入GET / HTTP/1.1然后按两下回车键。

如图6-4所示，这个时候，服务器端的响应就出现了！

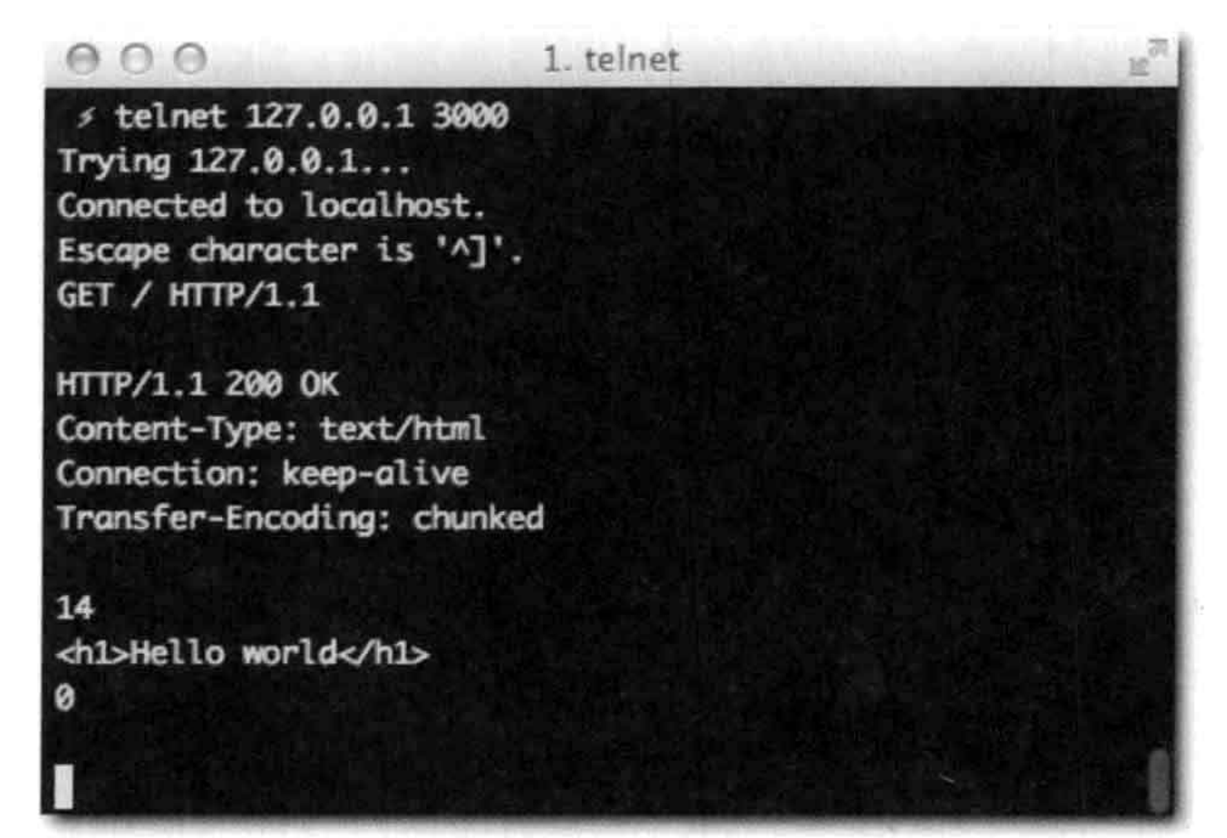


图6-4：Mac上IRC客户端（Textual.app）实践。Textual.app实现了基于TCP套接字的IRC协议

我们来总结一下：

- 成功建立了一个TCP连接。
- 创建了一个HTTP请求。
- 接收到了一个HTTP响应。
- 测试了一些TCP的特性。到达的数据和在Node.js中写的一样：先写了`Content-Type`响应头，然后是响应体，最后所有的信息都按序到达。

74 基于TCP的聊天程序

正如此前介绍的，TCP的主要目的就是为两台计算机通过提供可靠的网络进行通信。

本章选择一个聊天应用作为TCP的“Hello World”程序，因为，它是展示TCP最简单的方式之一。

下面，我们来创建一个基本的TCP服务器，任何人都可以连接到该服务器，无须实现任何协议或者指令：

- 成功连接到服务器后，服务器会显示欢迎信息，并要求输入用户名。同时还会告诉你当前还有多少其他客户端也连接到了该服务器。
- 输入用户名，按下回车键后，就认为成功连接上了。
- 连接后，就可以通过输入信息再按下回车键，来向其他客户端进行消息的收发。

为什么要按下回车键呢？事实上，Telnet中输入的任何信息都会立刻发送到服务器。按下回车键是为了输入\n字符。在Node服务器端，通过\n来判断消息是否已完全到达。所以，这其实是作为一个分隔符在使用。

换句话说，这里按下回车键和输入字符a没有什么区别。

创建模块

按照惯例，我们先来创建一个项目目录和package.json文件：

```
# package.json
{
    "name": "tcp-chat"
  , "description": "Our first TCP server"
  , "version": "0.0.1"
}
```

接着运行npm install测试一下。结果会输出一个空行，因为项目没有任何依赖。

理解NET.SERVER API

接下来，创建一个包含如下代码的index.js文件：

```
/**
 * 模块依赖
 */

var net = require('net')
/**
 * 创建服务器
 */
```

75

```
var server = net.createServer(function (conn) {
  // handle connection
  console.log('\033[90m   new connection!\033[39m');
});

/**
 * 监听
 */

server.listen(3000, function () {
  console.log('\033[96m   server listening on *:3000\033[39m');
});
```

注意，上述代码中为createServer指定了一个回调函数。该函数在每次有新的连接建立时都会被执行。

为了验证，我们来运行上述代码，启动一个TCP服务器。当listen执行时，它会将服务器绑定到3000端口，并最终在终端打印出一段消息。

```
$ node index.js
```

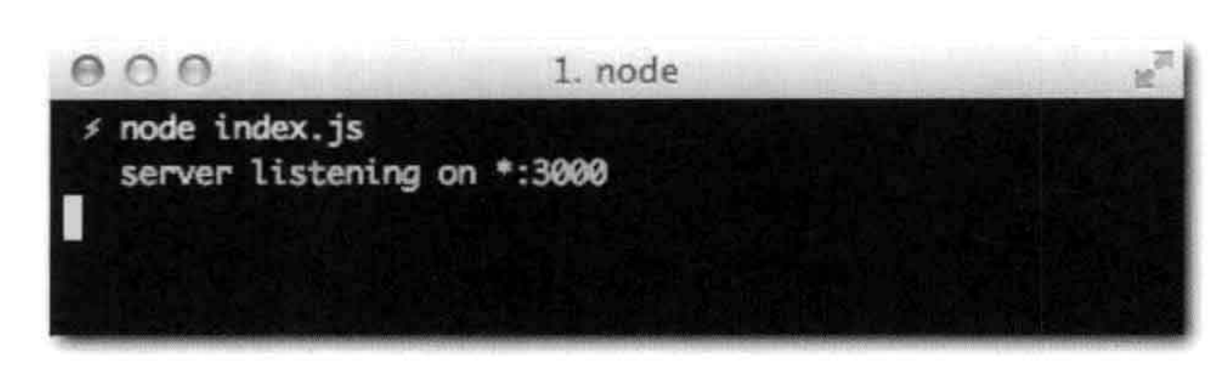


图6-5：服务器会绑定到3000端口，然后显示一段成功启动的信息

现在，我们尝试用telnet来进行连接：

```
$ telnet 127.0.0.1 3000
```

在图6-6中，能看到该命令，而且，“new connection！”消息也会显示出来。

如你所见，这个例子和此前HTTP的hello world程序类似。现在再去理解“HTTP是建立在TCP协议之上的”就不难了吧。不过，本例中，我们会创建自己的协议。

createServer回调函数会接收一个对象，该对象是Node中一个很常见的实例：流（Stream）。本例中，它传递的是net.Stream，该对象通常是既可读又可写的。

最后，还有一个重要的方法就是listen，它可以将服务器绑定要某个端口上。由于该方法也是异步的，所以也接收一个回调函数。

76

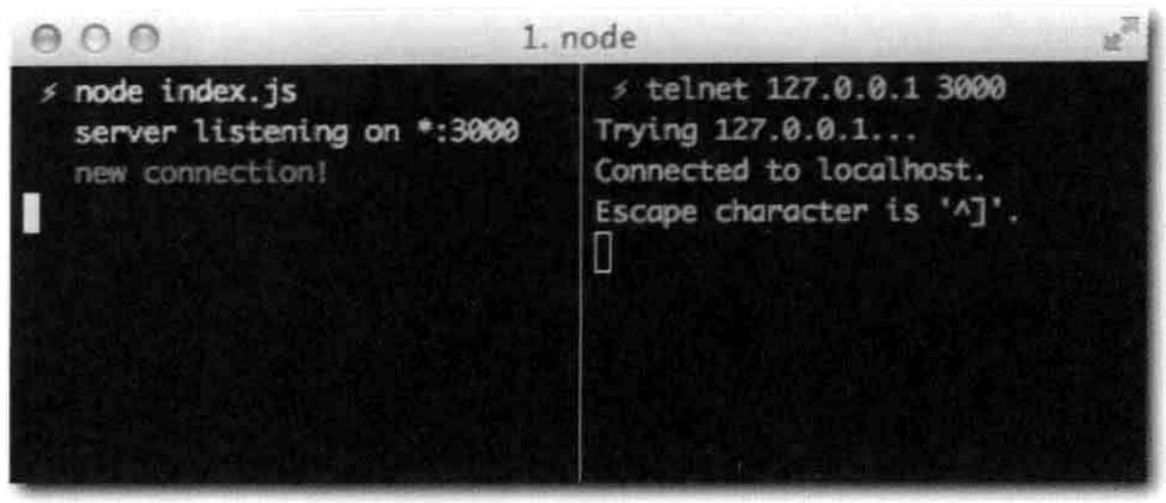


图6-6：左侧显示的是服务器进程的状态。右侧则是客户端，连接到服务器端后，服务器端就输出了“新连接（new connection）！”

接收连接

正如此前项目描述中的，一旦连接建立，就会向客户端回写欢迎语和当前连接数。

我们先在回调函数外部添加一个计数器：

```
/**
 * 追踪连接数
 */

var count = 0;
```

接着，我们需要修改回调函数内容，把计数器递增和打印出欢迎语的逻辑添加上去：

```
var server = net.createServer(function (conn) {
  conn.write(
      '\n > welcome to \033[92mnode-chat\033[39m!'
    + '\n > ' + count + ' other people are connected at this time.'
    + '\n > please write your name and press enter: '
  );
  count++;
});
```

如上述代码所示，我们仍旧使用shell转义码来控制输出文本的颜色。

接着重启服务器进行测试：

```
$ node index
```

再次通过telnet去连接（见图6-7）：

```
$ telnet 127.0.0.1 3000
```

77

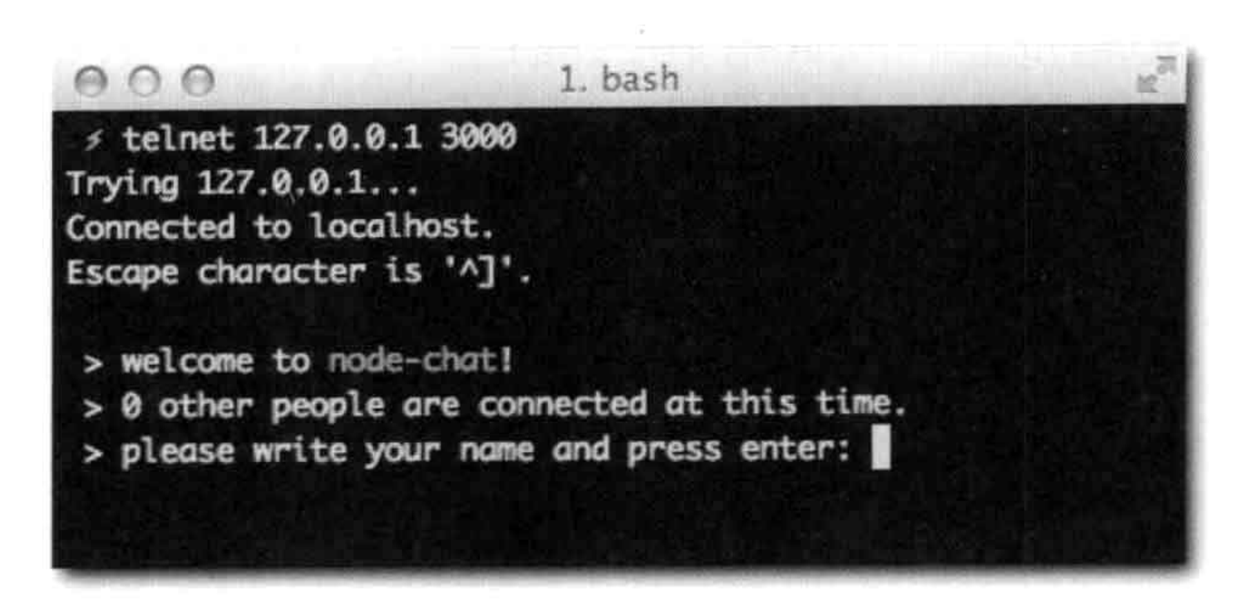


图6-7：现在客户端连接成功后能够接收到更多信息了

如图6-8所示，当第二个客户端连接进去后，计数器就增加了一个！

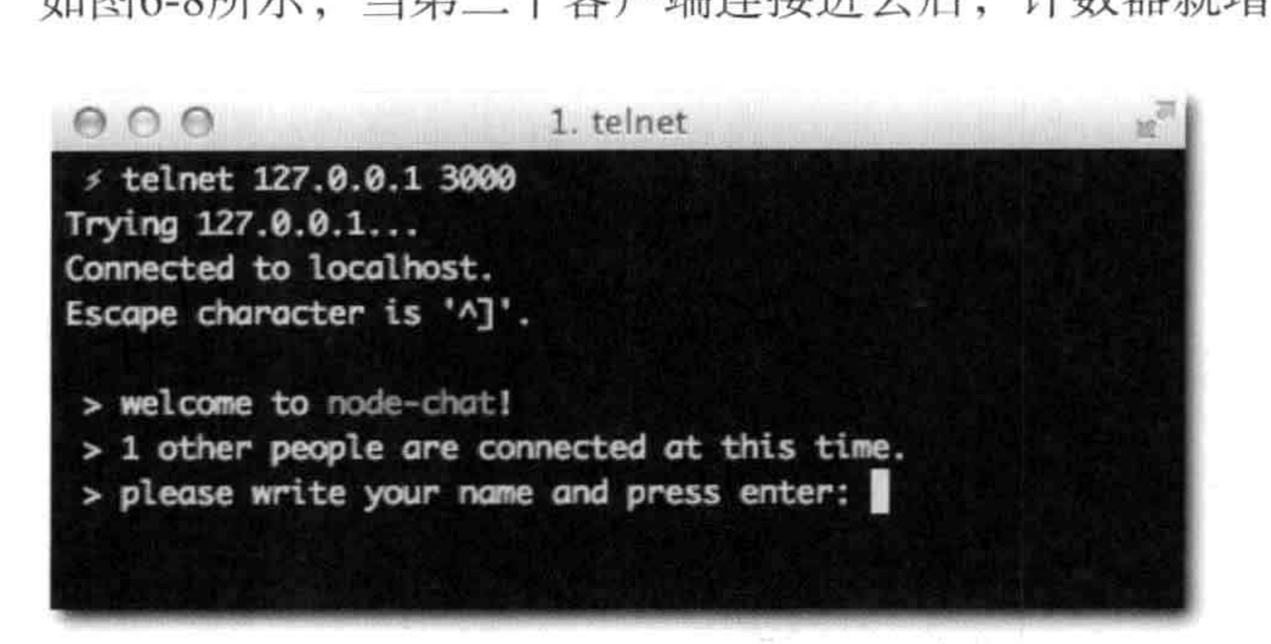


图6-8：连接计数器正确反映了当前连接数

当客户端请求关闭连接时，计数器变量就要进行递减操作：

```
conn.on('close', function () {
  count--;
});
```

当底层套接字关闭时，Node.js会触发close事件。Node.js中有两个和连接终止相关的事件：end和close。前者是当客户端显示关闭TCP连接时触发。比如，当你关闭telnet时，它会发送一个名为“FIN”的包给服务器，意味着要结束连接。

当连接发生错误时（触发error事件），end事件不会触发，因为服务器端并未收到“FIN”包信息。不过这两种情况下，close事件都会被触发，所以，上述例子中使用close事件会比较好。

在Mac上，可以通过按下Alt + [来结束一个telnet连接，Windows上可以用Ctrl +]。

data事件

我们已经能在客户端打印出一些信息了，接着我们来看看如何处理客户端发送的数据。

首先要处理的数据是用户输入的昵称（nickname），所以，我们从监听data事件开始。与

78

其他Node中的API一样，net.Stream同时也是一个EventEmitter。

为了进行测试，我们在服务器端的控制台输出客户端发来的数据：

```
var server = net.createServer(function (conn) {
  conn.write(
      '\n > welcome to \033[92mnode-chat\033[39m!'
    + '\n > ' + count + ' other people are connected at this time.'
    + '\n > please write your name and press enter: '
  );
  count++;

  conn.on('data', function (data) {
    console.log(data);
  });
  conn.on('close', function () {
    count--;
  });
});
```

然后，我们启动该服务器，并使用客户端进行连接。如图6-9所示，在客户端输入一些数据。看左侧部分，当输入数据时，服务器端就直接通过console.log将其打印出来了。

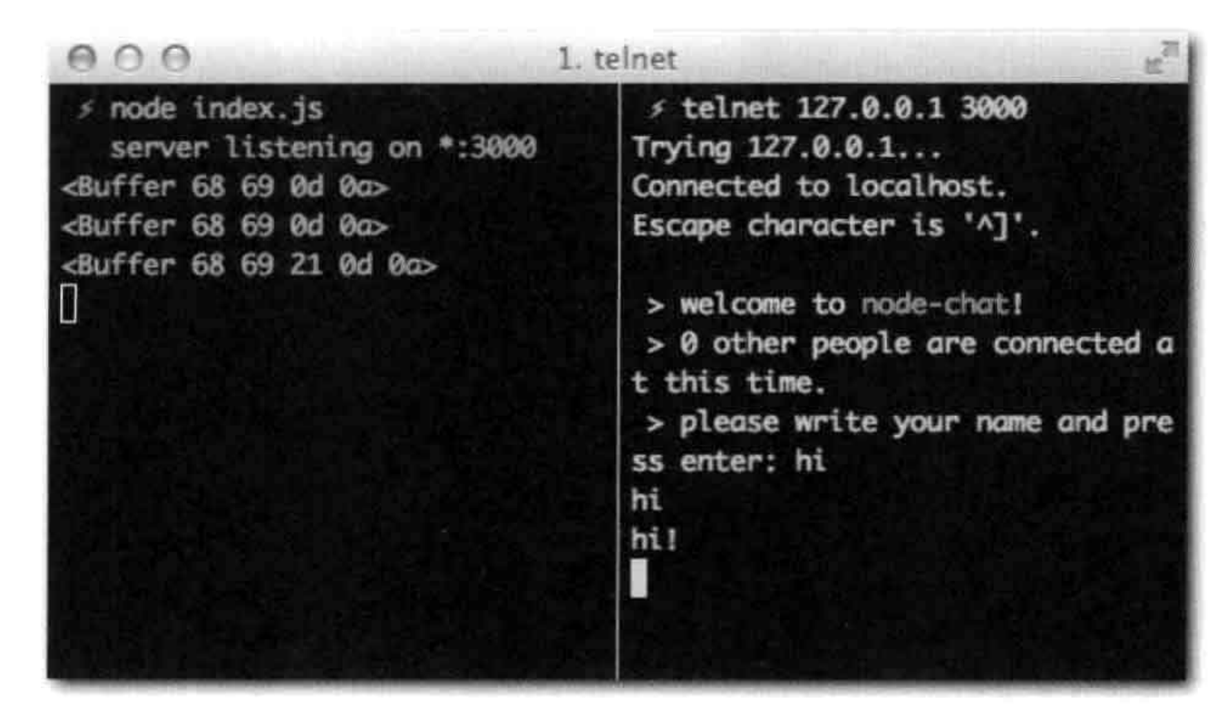


图6-9：图中左侧部分显示了右侧发送过来数据相对应的Buffer对象

如图6-9中所示，我们接收到的数据是一个Buffer。还记得此前介绍的，TCP是面向字节的协议吗？这里可以看得出来Node遵循了TCP的这一习惯！

这个时候，有多种选择可以获取字符串形式的数据。可以通过调用Buffer对象上的.toString('utf8')来获取utf8编码的字符串。

不过，由于我们这里无须获取utf8之外其他编码格式的数据，我们可以通过net.Stream#setEncoding方法来设置编码（如图6-10所示）：

```
# index.js
. . .
conn.setEncoding('utf8');
```

79

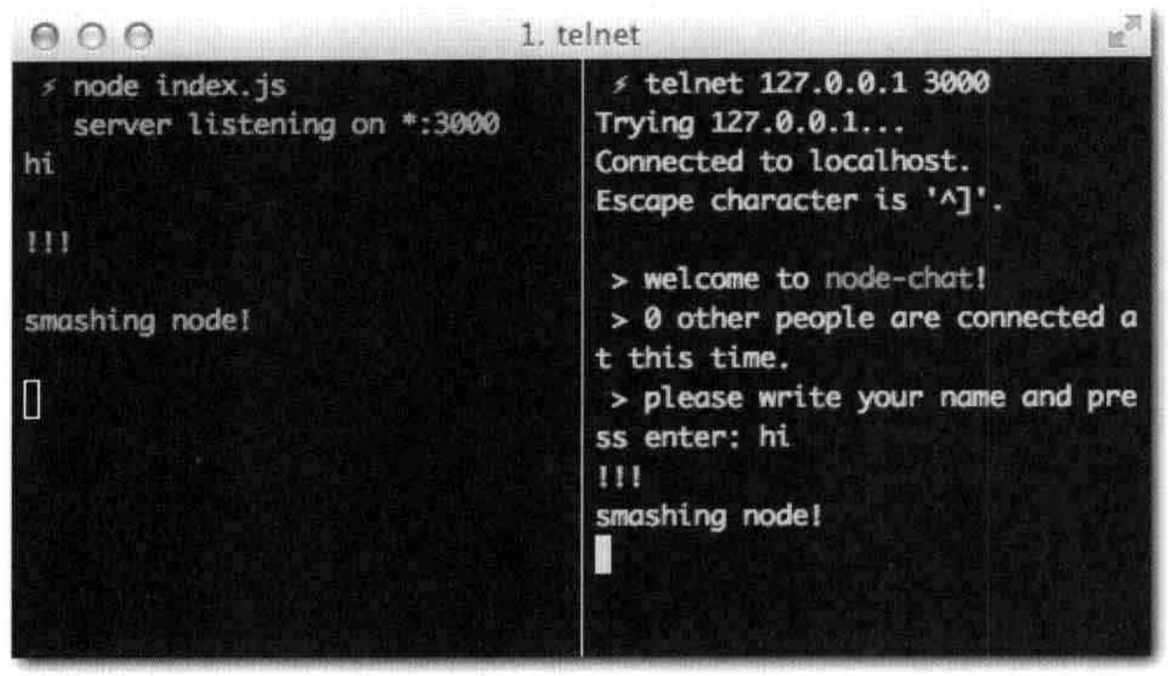


图6-10：聊天信息现在以utf8编码的字符串形式输出在左侧

好了，至此，我们已经可以让客户端和服务器进行交互了，接下来我们要让更多的客户端加入聊天。

状态以及记录连接情况

此前定义的计数器通常称为状态。因为，在本例中，两个不同连接的用户需要修改同一个状态变量，这在Node中称为共享状态的并发。

为了能够向其他连接进来的客户端发送和广播消息，我们需要对该状态进行扩展，来追踪到底谁连接进来了。

当客户端输入了昵称后，就认为该客户端已经连接成功，并可以接收消息了。

首先，我们要记录设置了昵称的用户。为此，我们需要引入一个新的状态变量，users：

```
var count = 0
  , users = {}
```

然后，在每个连接中，再引入一个nickname变量：

```
conn.setEncoding('utf8');

// 代表当前连接的昵称
var nickname;

conn.on('data', function (data) {
```

接收到数据时，确保将\r\n（相当于按下回车键）清除：

80

```
// 删除回车符
data = data.replace('\r\n', '');
```

对于尚未注册的用户，需要进行校验。如果昵称可用，则通知其他客户端当前用户已经连接进来了（见图6-11）：

```
// 接收到的第一份数据应当是用户输入的昵称
if (!nickname) {
  if (users[data]) {
    conn.write('\033[93m> nickname already in use. try again:\033[39m ');
    return;
  } else {
    nickname = data;
    users[nickname] = conn;

    for (var i in users) {
      users[i].write('\033[90m > ' + nickname + ' joined the room\033[39m\n');
    }
  }
}
```

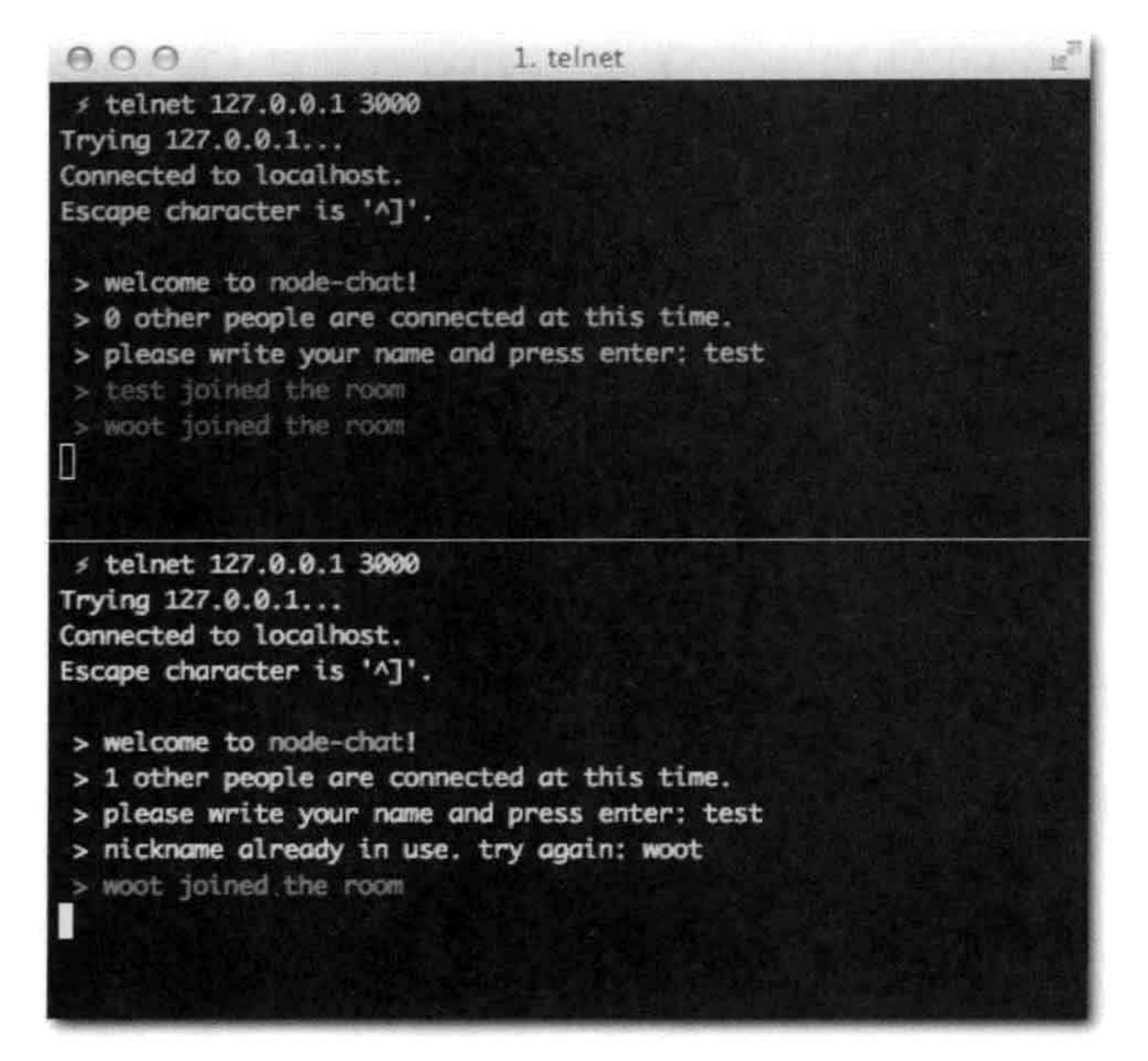


图6-11：消息已经广播给所有加入聊天服务器的客户端了

如果是已经验证通过的用户，那么接下来收到的数据就认为是聊天消息，需要显示给所有其他客户端：

81

```
else {
  // 否则，视为聊天消息
  for (var i in users) {
    if (i != nickname) {
      users[i].write('\033[96m > ' + nickname + ':\033[39m ' + data + '\n');
    }
```

```
  }
}
```

使用`i != nickname`来确保消息只发送给除了自己以外的其他客户端。

图6-12展示了两台客户端连接进来之后，互相聊天的场景。

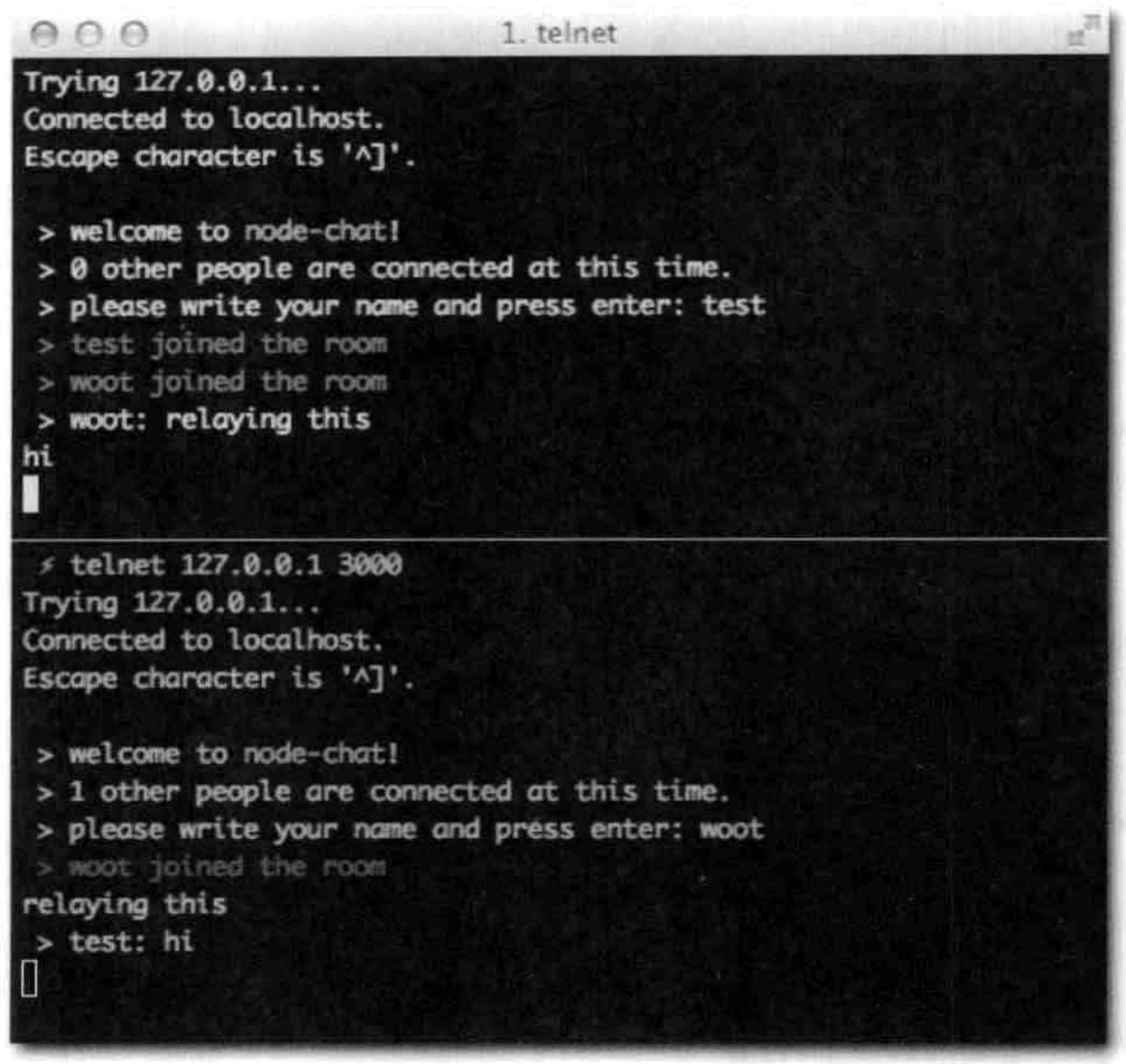


图6-12：展示了两台客户端连进来之后，互相聊天的场景

成功交换聊天消息后，我们来圆满完成这个程序。

圆满完成此程序

当有人断开连接时，我们需要清除users数组中对应的元素：

```
conn.on('close', function () {
  count--;
  delete users[nickname];
});
```

用户断开时通知其他用户也是个不错的想法。由于我们时不时就需要给所有的用户广播消息，所以，把这部分逻辑抽象出来也不错：

```
// . . .
function broadcast (msg, exceptMyself) {
  for (var i in users) {
    if (!exceptMyself || i != nickname) {
      users[i].write(msg);
    }
  }
```

82

```
}

conn.on('data', function (data) {
  // . . .
```

这样一来，就可以重用上面的函数进行消息广播了，简洁易懂，一目了然：

```
broadcast('\033[90m > ' + nickname + ' joined the room\033[39m\n');
// . . .
broadcast('\033[96m > ' + nickname + ':\033[39m ' + data + '\n', true);
```

我们把它加到close处理器中（见图6-13）：

```
conn.on('close', function () {
  // . . .
  broadcast('\033[90m > ' + nickname + ' left the room\033[39m\n');
});
```

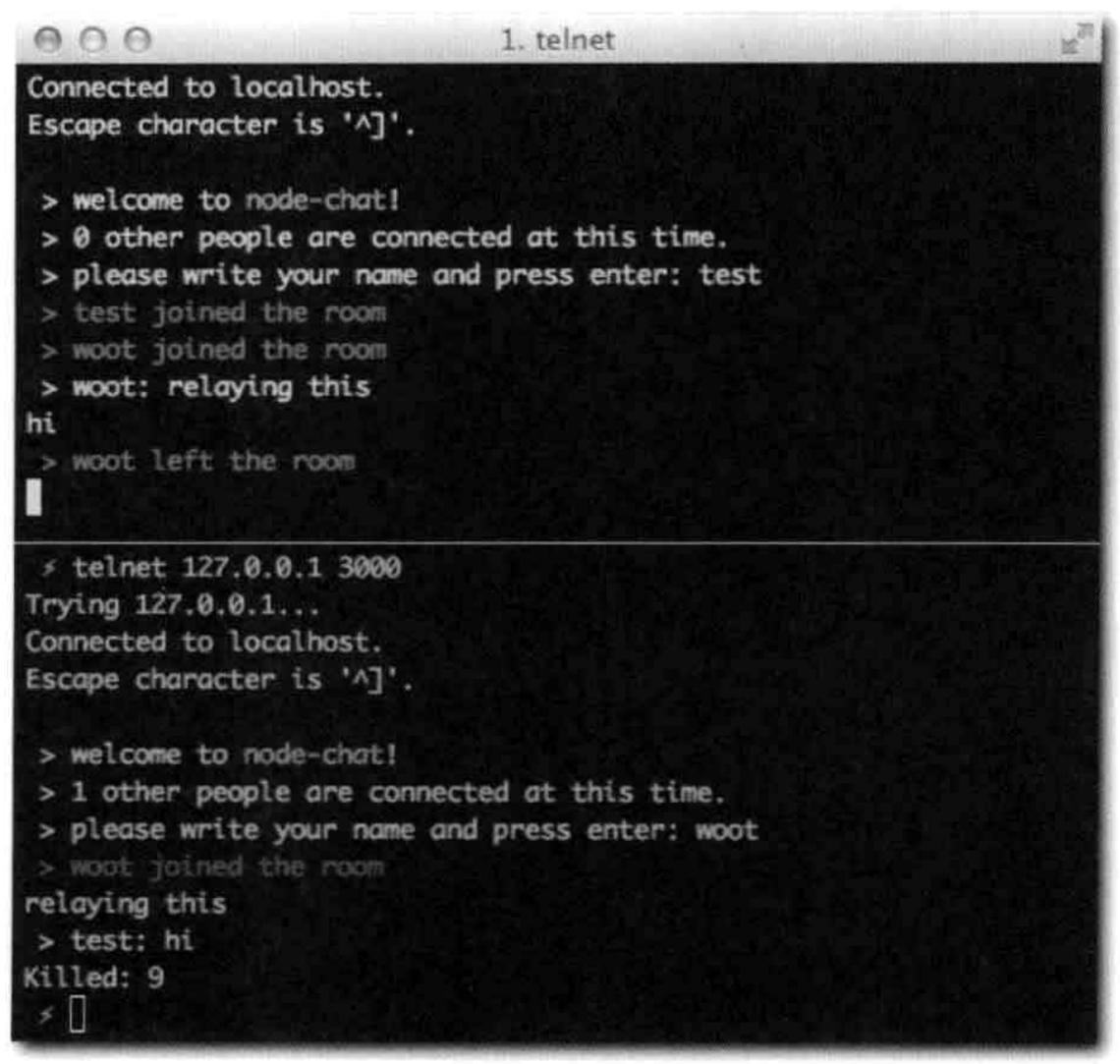


图6-13：在关掉第一个客户端时，会强制关闭该连接，接着其他客户端就会收到当前用户离开的消息

完成！

83

在成功实现了一个TCP服务器后，我们需要进一步学习在Node.js中如何实现一个TCP客户端。

客户端API和其他像Twitter这样用来查询Web服务的HTTP客户端类似，因此，完全明白这些非常重要。

一个IRC客户端程序

IRC是因特网中继聊天（Internet Relay Chat）的缩写，它也是一项常用的基于TCP的协议。它通常被使用在如图6-14所示的客户端程序中，作为连接到IRC服务器的客户端。

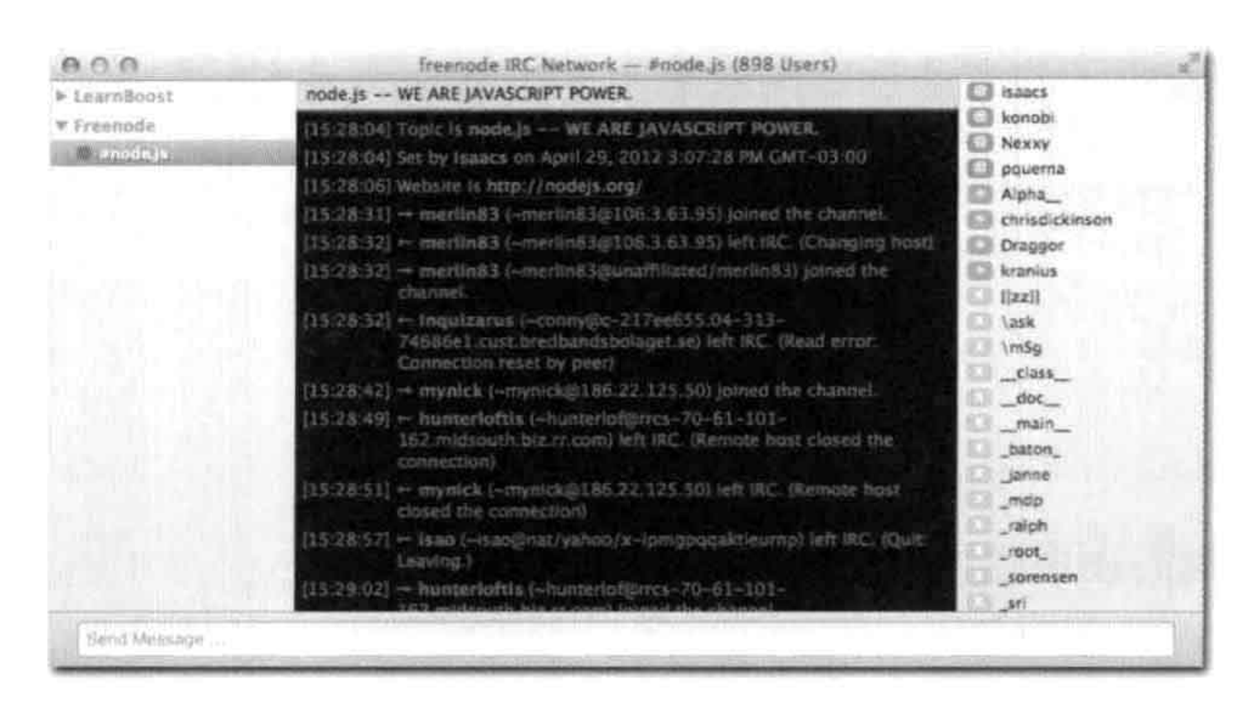


图6-14：正在运行的Mac上一款IRC客户端程序（Linkinus）。Linkinus实现了基于TCP套接字的IRC协议

此前我们成功地构建了一个TCP服务器，这次我们来写一个TCP客户端。

构建一个实现IRC协议的客户端意味着，需要实现通过一组命令来实现与IRC服务器进行“通信”，进行数据的交换。

比如，要设置昵称，客户端可以发送如下指令：

```
NICK mynick
```

IRC是一项非常直观、简单的协议。通过一些简单的命令就可以和现有的应用以及服务器（如图6-14中展示的）进行通信。

下面我们就来学习如何用Node.js书写一个非常基本的客户端，实现加入一个聊天室以及回复消息功能。

创建模块

和往常一样，我们先来创建一个项目目录，并在该目录下创建一个package.json文件：

```
{
    "name": "irc-client"
  , "description": "Our first TCP client"
  , "version": "0.0.1"
}
```

84

接着运行npm install。因为项目没有任何依赖的模块，所以控制台应当会输出一个空行。

理解NET#STREAM API

和createServer一样，net API提供了另外一个名为connect的方法，如下所示：

```
net.connect(port[[, host], callback]])
```

如果提供了回调函数，就等于是监听了该方法返回的对象上的connect事件。

```
var client = net.connect(3000, 'localhost');
client.on('connect', function () {});
```

上述代码和下面的代码是一样的：

```
net.connect(300, 'localhost', function () {});
```

另外，和此前使用的API类似，我们还可以监听data和close事件。

实现部分IRC协议

我们先来初始化IRC客户端。然后尝试去登录irc.freenode.net中的#node.js频道：

```
var client = net.connect(6667, 'irc.freenode.net')
```

设置编码为utf-8：

```
client.setEncoding('utf-8')
```

连接上服务器后，发送自己的昵称。除此之外，还要写上服务器要求的USER命令。采用类似如下所示的方式来发送数据：

```
NICK mynick
USER mynick 0 * :realname
JOIN #node.js
```

85 具体代码如下所示：

```
client.on('connect, function () {
  client.write('NICK mynick\r\n');
  client.write('USER mynick 0 * :realname\r\n');
  client.write('JOIN #node.js\r\n')
});
```

这里要注意，需要在每条命令后加上\r\n分割符。这和此前在Telnet下按下回车键是一样的。\r\n也是HTTP协议用来区分头信息的分隔符。

测试实际的IRC服务器

打开一个IRC客户端（如Windows上的mIRC、Linux上的xChat或者Mac上的Colloquy/Linkinus），并将其指向：

```
irc.freenode.net
#node.js
```

启动后，观察mynick是否连接上了：

```
![](http://f.cl.ly/items/1b3g3i1w120Z2U082I3G/Image%202011.11.07%202:31:35%20AM.png)
```

小结

本章介绍了一个简单的网络客户端的实现，成功使其与一台并非自己实现的TCP服务器进行通信。

作为习题，你可以尝试着去完成如下部分：监听data数据并尝试解析接收到的数据，将其他用户发送到#node.js频道的消息打印出来。你还可以进一步尝试着结合现有的代码去实现一个IRC机器人，来自动对命令做出响应。比如，当某人说日期（data）时（可以在data事件中检测到），你就可以输出new Date()的结果。

紧接着在下一章，你会学到HTTP这种Node.js因此闻名的Web协议。现在你对TCP中数据传递、数据块的构建已经很清楚了，那么接下来学习TCP上层的HTTP API，一定会让你对Node.js中的核心功能有更加深入的了解。

CHAPTER

7 HTTP

超文本传输协议，又称为HTTP，是一种Web协议，它为Web注入了很多强大的功能，正如在第6章中介绍的，它是属于TCP上层的协议。 87

本章会介绍如何使用Node.js服务器端和客户端的API。尽管两者都很容易上手，但是在构建真正的Web网站时，它们还是存在着一些不足的。正因如此，下面的章节还会给大家介绍如何实现HTTP服务器上层的抽象，完成可复用的模块。

注意，由于我们要编写的示例代码一部分属于服务器端，所以每次对文件进行修改，都需要重启Node进程来使其生效。在本章的最后，会给大家介绍如何使用工具来简化这种频繁重启的操作。

下面，就让我们从分析HTTP协议开始吧。

HTTP结构

88

HTTP协议构建在请求和响应的概念上，对应在Node.js中就是由`http.ServerRequest`和`http.ServerResponse`这两个构造器构造出来的对象。

当用户浏览一个网站时，用户代理（浏览器）会创建一个请求，该请求通过TCP发送给

Web服务器，随后服务器会给出响应。

那么，请求和响应是什么样的呢？我们先用Node创建一个Hello World的HTTP服务器，并监听http://localhost:3000：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello World');

}).listen(3000);
```

接着，建立一个telnet连接，并发送请求：

```
GET / HTTP/1.1
```

输入GET / HTTP/1.1后，按下两次回车键。

如图7-1所示，响应会立刻出现！

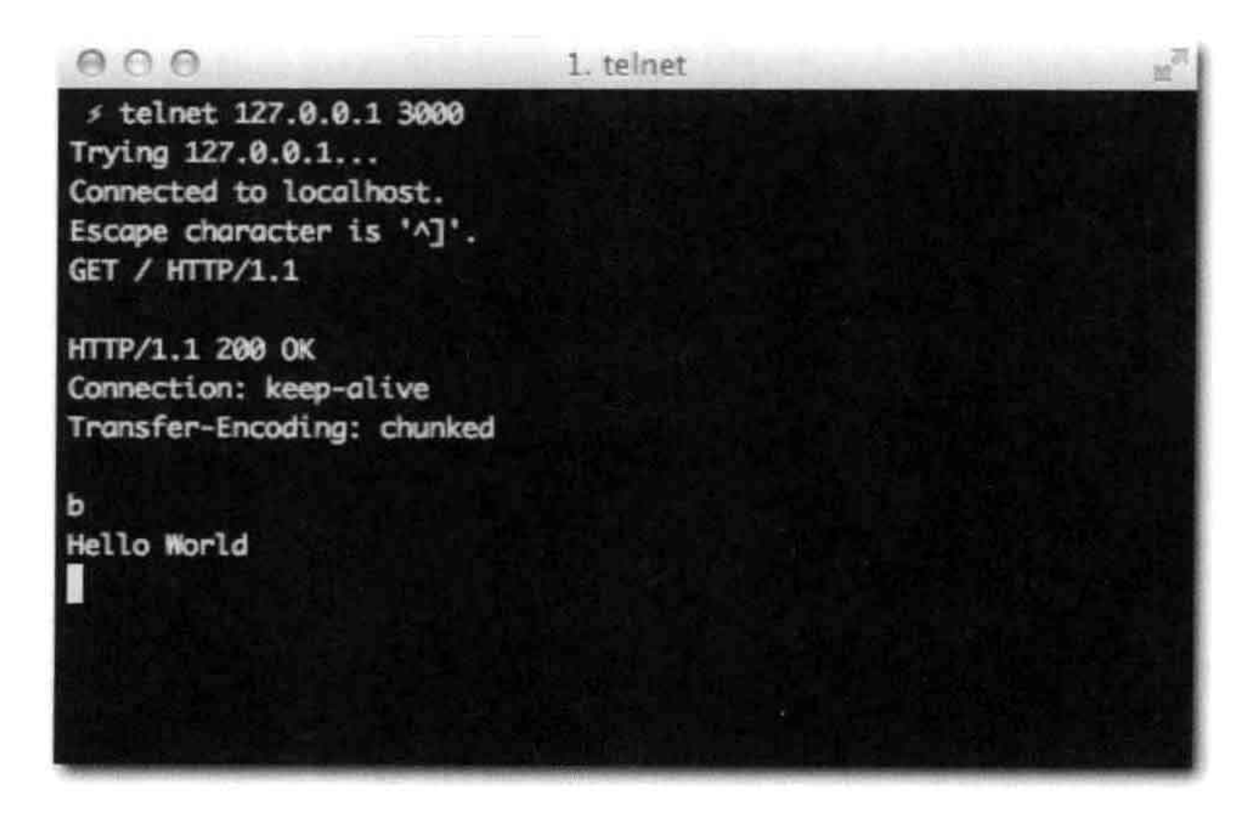


图7-1：HTTP服务器发送的响应

响应的内容如下所示：

```
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

b
Hello World
0
```

89

该响应中第一部分是头信息，下面会对头信息做相关介绍。

头信息

HTTP协议和IRC一样流行，其目的是进行文档交换。它在请求和响应消息前使用头信息

（header）来描述不同的消息内容。

举个例子，Web页面会分发许多不同类型的内容：文本（text）、HTML、XML、JSON、PNG以及JPEG图片，等等。

发送内容的类型（type）就是在著名的Content-Type头信息中标注的。

来看一个实践中的例子。还是回到hello world，不过这次我们在里面加点HTML：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello <b>World</b>');

}).listen(3000);
```

注意，World这个单词放在粗体的标签内。我们再通过简单的TCP客户端来看看效果（见图7-2）。

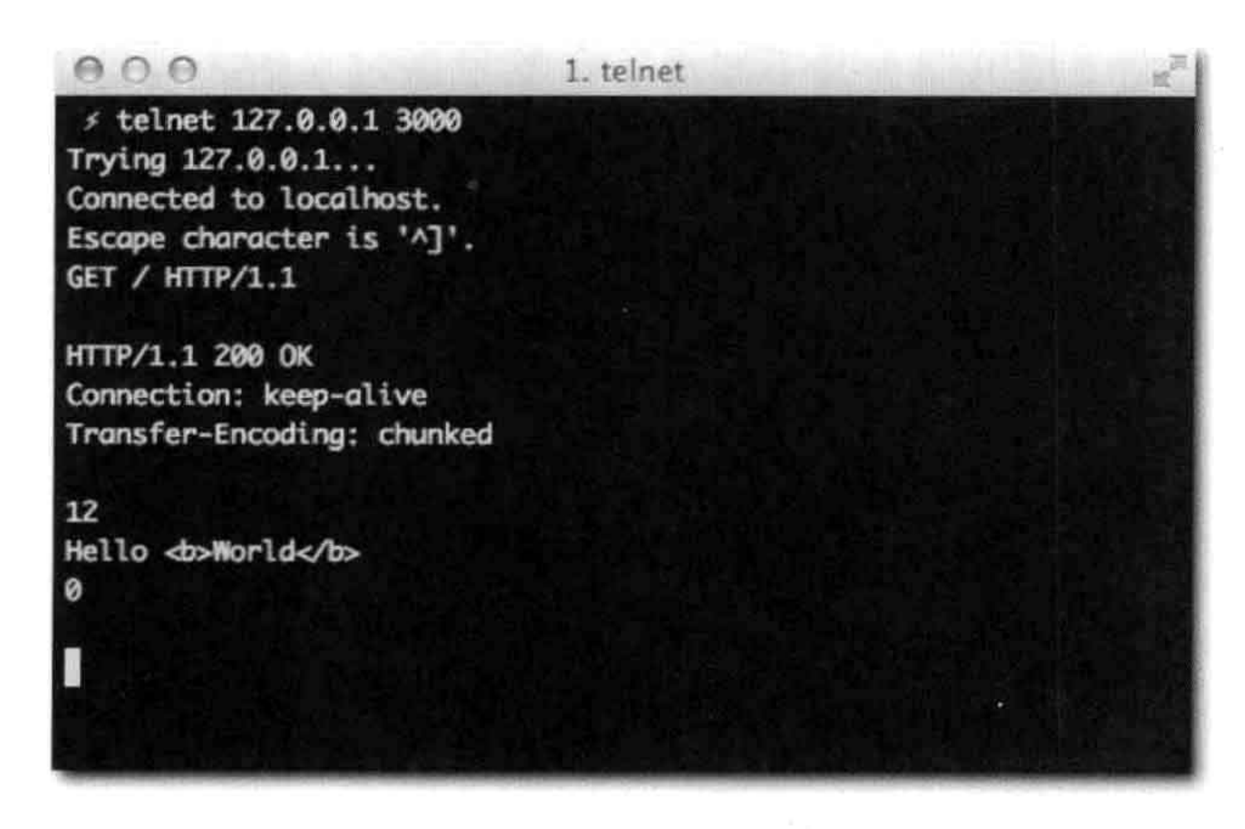


图7-2：Hello World的响应

响应结果和预期的一样：

```
GET / HTTP/1.1

HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

12
Hello <b>World</b>
0
```

90

不过，现在我们再在浏览器中试试，看看结果会如何（见图7-3）。

图7-3：浏览器将响应结果作为纯文本来处理了

可是并没有看到富文本，怎么回事？

之所以会这样，是因为HTTP客户端（浏览器）并不知道服务器发送过来的内容是什么类型，我们没有把这部分信息告诉浏览器。于是，浏览器就认为它看到的内容是text/plain类型，也就是普通文本类型，就不会将它作为HTML来渲染了。

如果我们把代码稍做修改，加入正确的头信息，就能解决这个问题了（见图7-4）：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('Hello <b>World</b>');

}).listen(3000);
```

91

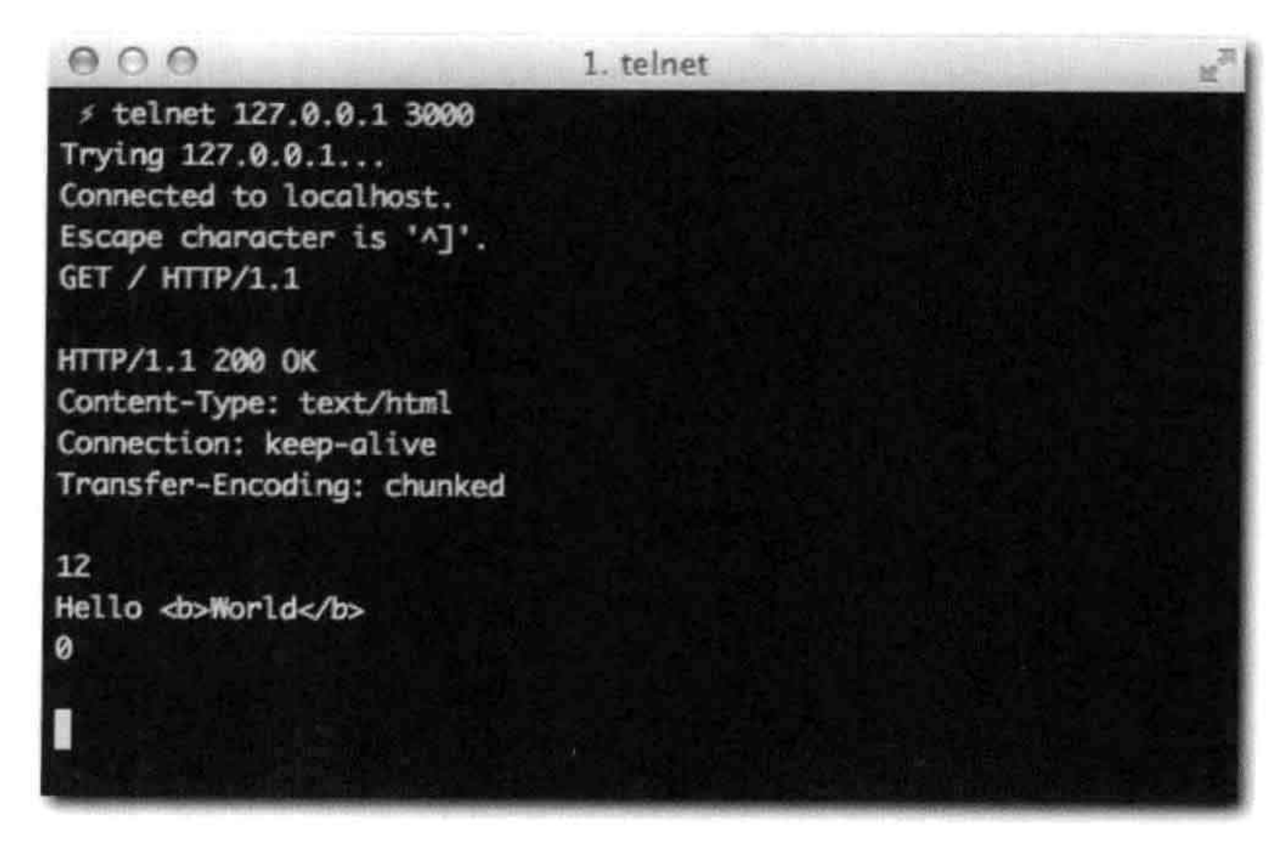


图7-4：带有头信息的响应消息

响应消息如下所示：

```
HTTP/1.1 200 OK

Content-Type: text/html
Connection: keep-alive
Transfer-Encoding: chunked

12
Hello <b>World</b>
0
```

注意，这里头信息包含在响应消息中。浏览器会对其进行解析（见图7-5），这就能够正确地渲染出HTML了。

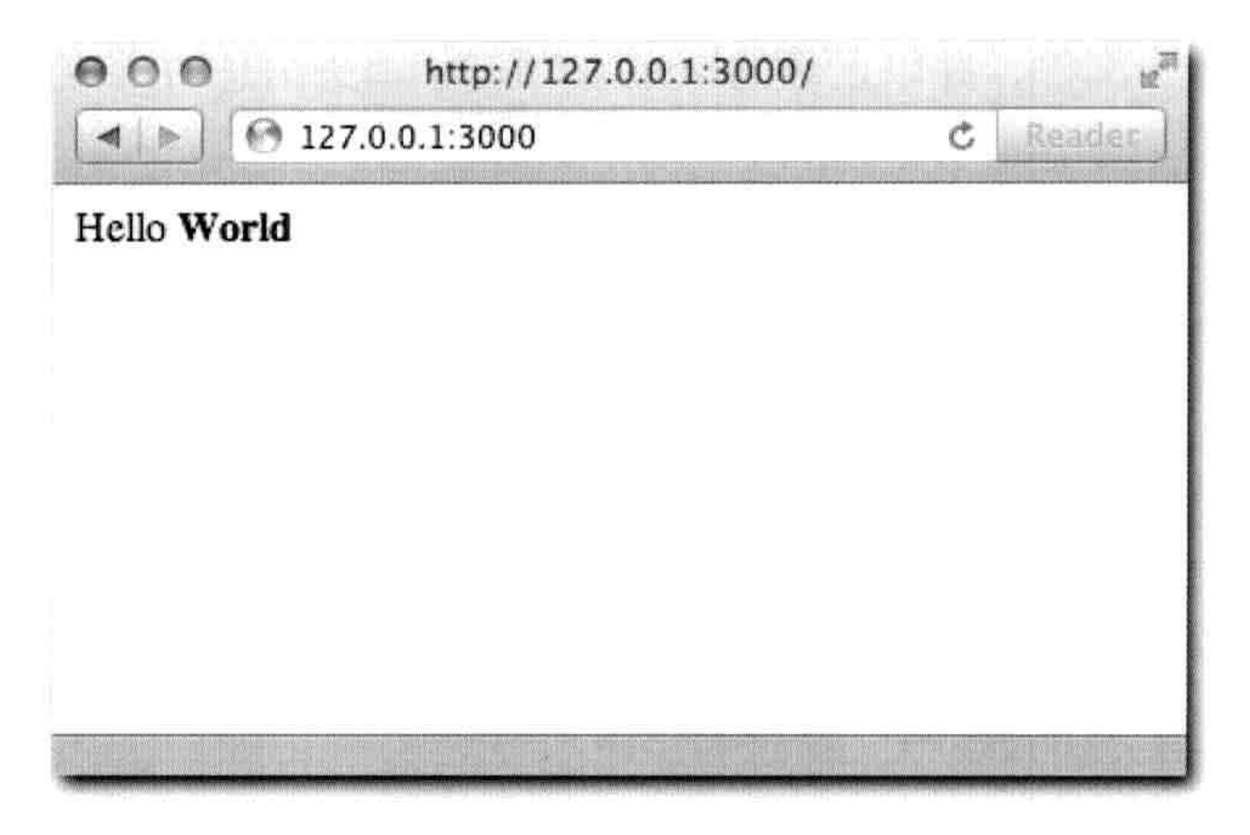


图7-5：现在浏览器将World这个单词展示成粗体了

注意，尽管我们只用`writeHead` API指定了一个头信息，但是，Node还是把另外两个头信息——`Transfer-Encoding`和`Connection`加进去了。 92

`Transfer-Encoding`头信息的默认值是`chunked`，主要的原因是Node天生的异步机制，这样响应就可以逐步产生。

来看下面这个例子：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.write('Hello');

  setTimeout(function () {

    res.end('World');

  }, 500);

}).listen(3000);
```

注意，在调用end前，我们可以多次调用write方法来发送数据。为了尽可能快地响应客户端，在首次调用write时，Node就能把所有的响应头信息以及第一块数据（Hello）发送出去。

随后，在执行setTimeout回调函数时，又写入了另外一块数据。由于这次是使用end而不是write方法，因此，Node会结束响应，并不再允许往这次响应中发送数据了。

发送数据块的方式在涉及文件系统的情况下会非常高效。Web服务器对硬盘上的文件进行托管服务是很常见的。因为Node允许以数据块的形式往响应中写数据，同时它又允许以数据块的形式读取文件，所以我们就可以使用ReadStream文件系统API来实现。

下面这个例子用于读取image.png文件，并以正确的Content-Type头信息做出响应：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'image/png');
  var stream = require('fs').createReadStream('image.png');
  stream.on('data', function (data) {
    res.write(data);
  });
  stream.on('end', function () {
    res.end();
  });
}).listen(3000);
```

93

以一系列数据块的形式来将图片写入到响应中，有如下好处：

- 高效的内存分配。要是对每个请求在写入前都完全把图片信息读取完（通过fs.readFile），在处理大量请求时会消耗大量内存。
- 数据一旦就绪就可以立刻写入了。

另外，这里要注意的是，实际上我们做的就是把一个流（Stream）（文件系统）接（piping）到了另一个流上（一个http.ServerResponse对象）。正如此前介绍的，流是Node.js中非常重要的一种抽象。流的对接是很常见的行为，为此，Node.js提供了一个方法让上述例子代码变得非常简洁：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'image/png');
  require('fs').createReadStream('image.png').pipe(res);
}).listen(3000);
```

好了，现在你应当明白为什么Node默认要用chunked传输编码了，下面我们来看看连接（connection）。

连接

要是比对一下TCP服务器和HTTP服务器的实现，你可能会注意到它们很相似：都调用了`createServer`方法，并且当客户端连入时都会执行一个回调函数。

不过，它们有个本质的区别，即回调函数中对象的类型。在`net`服务器中，是个连接（connection）对象，而在HTTP服务器中，则是请求和响应对象。

之所以会这样，原因有两个。其一，HTTP服务器是更高层的API，提供了控制和HTTP协议相关的一些功能。

比如，当Web浏览器请求服务器时，我们来看下请求对象（例子中的`req`参数）中的`headers`属性（见图7-6）。作为实验代码，我们就用`console.log`将`req.headers`属性值打印出来：

```
require('http').createServer(function (req, res) {
  console.log(req.headers);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('Hello <b>World</b>');

}).listen(3000);
```

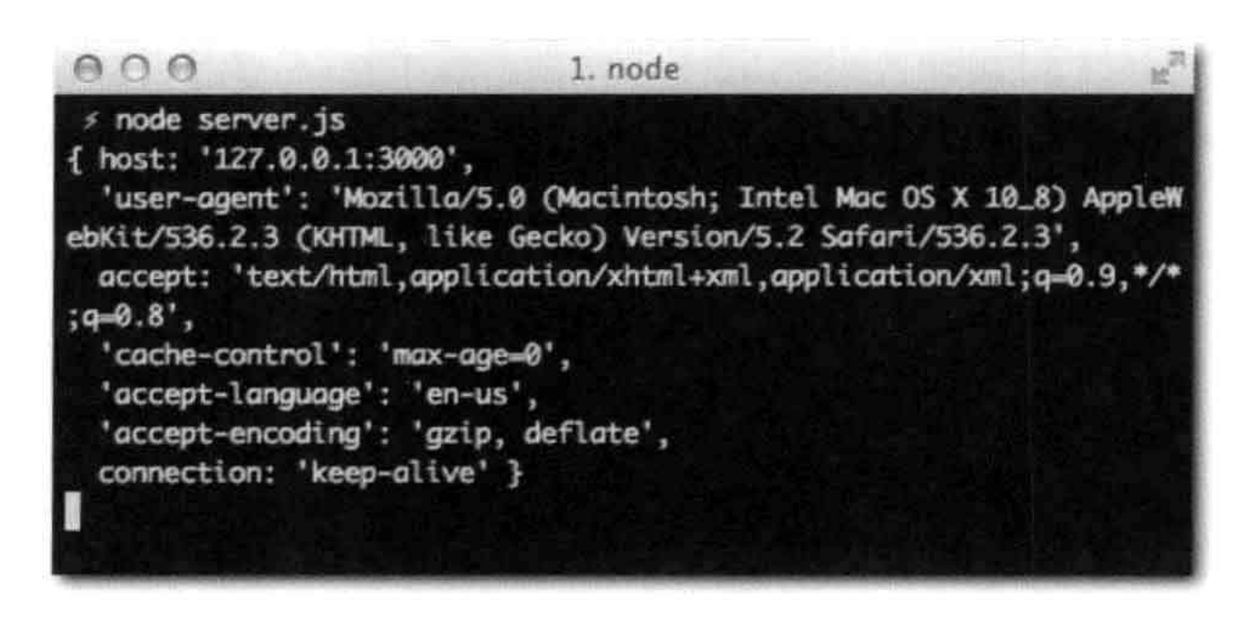


图7-6：利用`console.log`打印出的`ServerRequest headers`属性值

注意了，这里Node在内部做了很多的事情。它拿到浏览器发送的数据后，对其进行分析（解析），然后构造了一个JavaScript对象方便我们在脚本中使用。它甚至还将所有的头信息都变成了小写，这样我们就无须去记忆到底是`Content-type`、`Content-Type`还是`Content-TYPE`了。

其二，同时也是一个更为重要的原因是，浏览器在访问站点时不会就只用一个连接。很多主流的浏览器为了更快地加载网站内容，能向同一个主机打开八个不同的连接，并发送请求。

尽管我们可以通过`req.connection`获取TCP连接对象，Node为了不让我们担心到底这是请求还是连接，为我们提供了请求和响应的抽象。因此，即使你能通过`req.connection`

94

属性获得TCP连接对象，但大多情况下你还是在与请求和响应的抽象打交道。

默认情况下，Node会告诉浏览器始终保持连接，通过它发送更多的请求。这是通过此前我们看到的Connection头信息中的keep-alive值来通知浏览器的。为了提高性能（因为浏览器不想浪费时间去重新建立和关闭TCP连接），这样做通常都是对的，不过，我们也可以调用writeHead方法，传递一个不同的值，如Close，来将其重写掉。

下一个项目，我们会使用Node HTTP API来完成一个切实的任务：处理用户提交的表单。

一个简单的Web服务器

通过本项目，你能够学到如何使用此前列出的一些关键概念，如Content-Type头信息。

你还将学到Web浏览器是如何在表单提交时传递编码过的数据的，以及如何将它们解析为JavaScript中的数据结构。

95 创建模块

按照惯例，我们从创建一个项目目录开始，并在该目录下创建一个package.json文件：

```
{
    "name": "http-form"
  , "description": "An HTTP server that processes forms"
  , "version": "0.0.1"
}
```

接着运行npm install。因为项目没有任何依赖的模块，所以终端应该只会打印出一个空行。

输出表单

此前Hello World例子中，我们输出了一些HTML内容。本例中，我们要输出一个表单。在server.js文件中写入如下内容：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end([
      '<form method="POST" action="/url">'
    , '<h1>My form</h1>'
    ,   '<fieldset>'
    ,   '<label>Personal information</label>'
```

```
    ,   '<p>What is your name?</p>'
    ,   '<input type="text" name="name">'
    ,   '<p><button>Submit</button></p>'
    , '</form>'
  ].join(''')); }).listen(3000);
```

注意，为了让HTML结构更加清楚，我把响应文本内容写在一个数组中，再用数组的join方法将其转为字符串。其他部分和Hello World例子一样。

还要注意的是，<form>标签中有/url以及POST方法。另外，供用户输入的输入框还有个叫name的名字。

下面，我们来运行服务器：

```
$ node server.js
```

然后，通过浏览器进行访问，如图7-7所示，就能看到表单了。 96

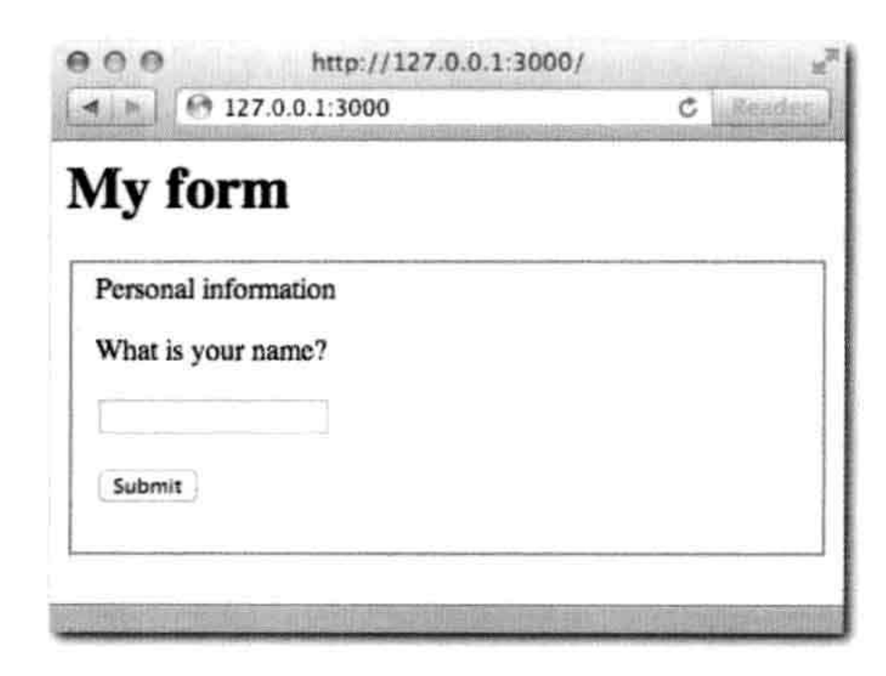


图7-7：渲染出的表单页面

你可以按下回车键试试。浏览器会发出一个新的请求（包含了相应数据），不过，因为现在所有的代码只用于输出表单的HTML，所以结果与图7-7看到的应该是一样的（见图7-8）。输入名字后，然后单击“Submit”（提交）按钮。

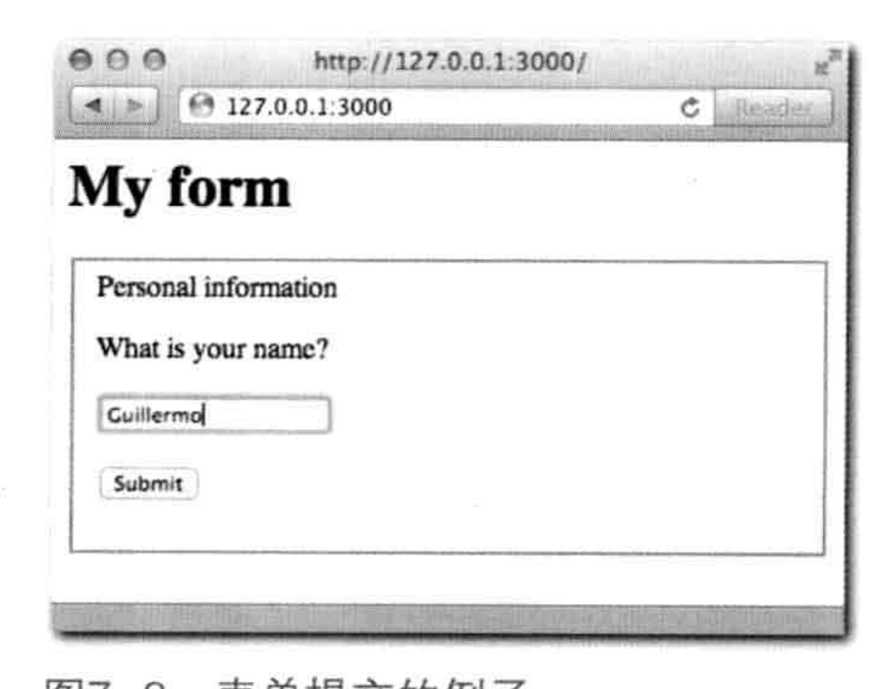


图7-8：表单提交的例子

提交后的结果是，URL变了，不过响应结果还是一样的，如图7-9所示。

97

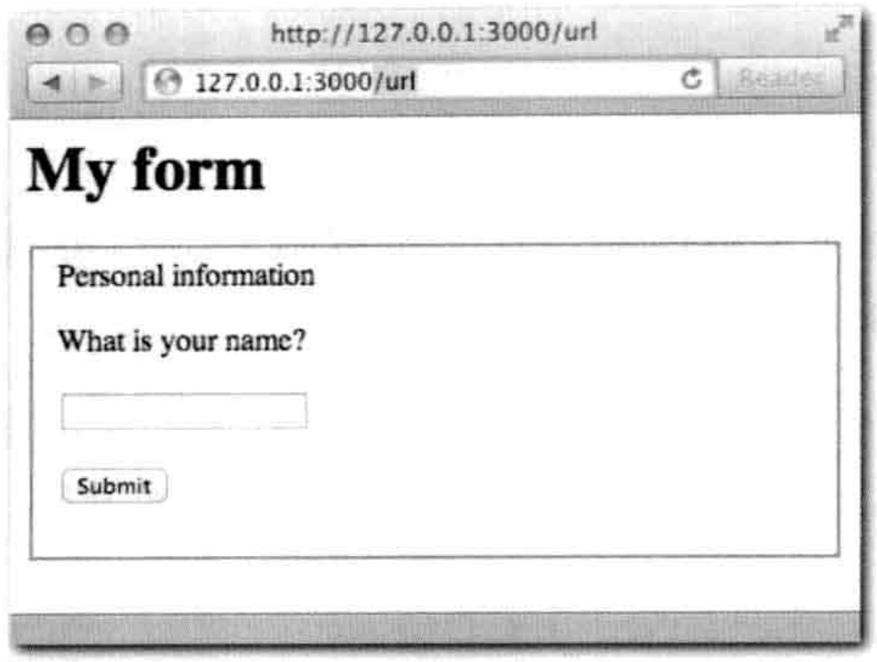


图7-9：尽管表单提交了，但Node还是以同样的方式对请求进行处理，这就是结果还是一样的原因

为了让Node能够对表单提交的请求做出正确的处理，我们需要学习关于检测请求方法和URL的相关内容。

method和URL

显然，在用户按下回车键（提交表单）后，我们得向用户展示些不同的东西，这个时候就需要处理表单。

在代码的最后，需要对请求对象上的`url`属性进行检测。server.js的代码如下所示：

```
require('http').createServer(function (req, res) {
  if ('/' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form method="POST" action="/url">'
      ,   '<h1>My form</h1>'
      ,   '<fieldset>'
      ,   '<label>Personal information</label>'
      ,   '<p>What is your name?</p>'
      ,   '<input type="text" name="name">'
      ,   '<p><button>Submit</button></p>'
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end('You sent a  <em>' + req.method + '</em> request');
  }
}).listen(3000);
```

98

如果访问/ URL，就会看到如图7-10所示的页面，没有任何变化。

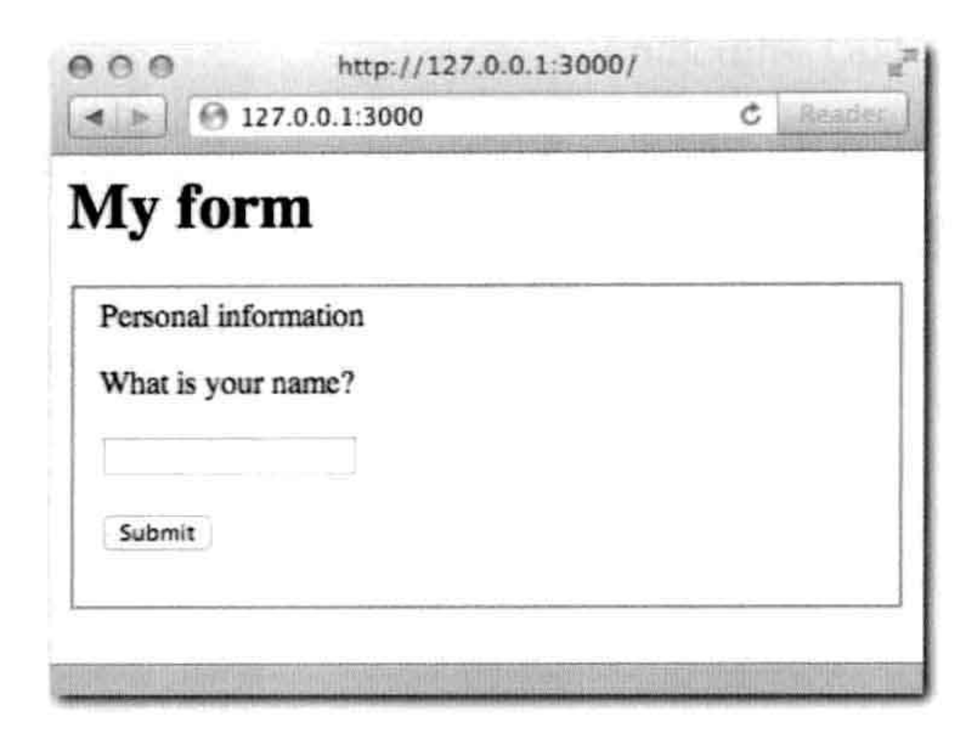


图7-10：当访问URL时，请求处理器仍旧显示同样的HTML内容

要是输入/url，就会看到如图7-11所示的页面。因为该URL匹配到了req.url在else if的这种情况，所以获得了这样的响应结果。

图7-11：当访问/url时，随着req.url值的变化导致了不同的响应结果

但是，当在表单中输入名字并提交后，就会看到如图7-12所示的页面。这是因为浏览器会通过<form>标签中action属性指定的HTTP方法将表单数据发送过去。于是，在本例中req.method值就是POST了，接着就会响应出如图7-12所示的内容了。

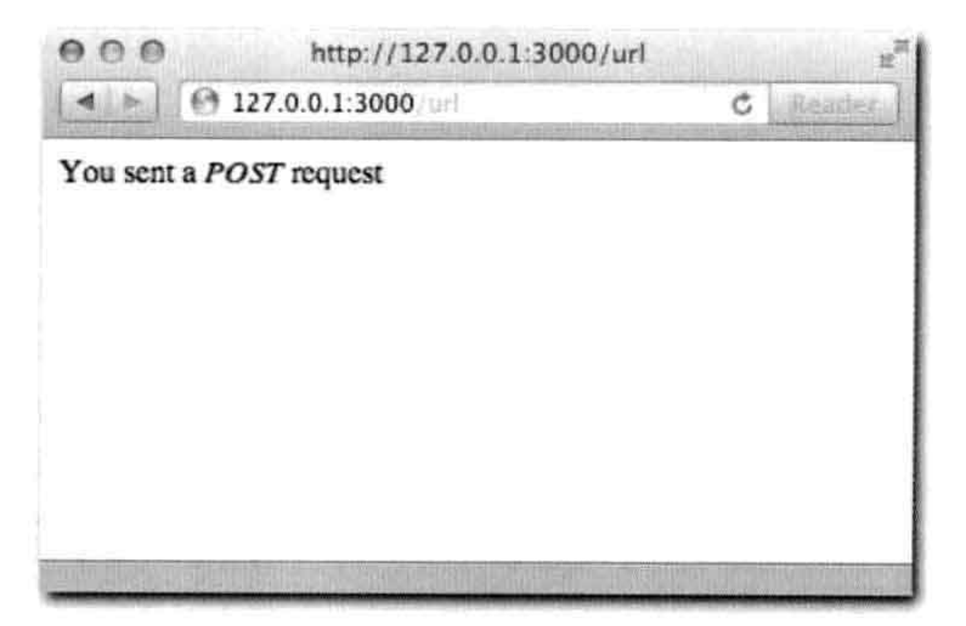


图7-12：本例中，req.method值为POST

如你所见，我们接触了请求对象的两个变量：URL和method。

Node.js会将主机名后所有的内容都放在url属性中。假设访问http://myhost.com/url?this+is+a+long+url，那么url的值就会是/url?this+is+a+long+url。

Web协议HTTP/1.1（你也许还记得第6章中介绍的telnet的例子），为请求定义了以下不同的方法：

- GET（默认）
- POST
- PUT
- DELETE
- PATCH（最新的）

定义这些不同方法的用意在于可以让HTTP客户端选择合适的方法，通过发送请求数据，修改服务器上的资源，该资源通过URL来指定。

数据

当发送HTML时，需要随着响应体定义Content-Type头信息。

100 和响应消息一样，请求消息也可以包含Content-Type头信息。为了更有效地处理表单，这两部分信息都是不可或缺的。就像在没有显式告知浏览器的情况下，浏览器不知道Hello World到底是HTML还是纯文本一样，我们怎样知道用户发送的用户名是JSON格式、XML格式，还是纯文本格式呢？server.js代码如下所示：

```
require('http').createServer(function (req, res) {
  if ('/' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form method="POST" action="/url">'
      ,   '<h1>My form</h1>'
      ,   '<fieldset>'
      ,   '<label>Personal information</label>'
      ,   '<p>What is your name?</p>'
      ,   '<input type="text" name="name">'
      ,   '<p><button>Submit</button></p>'
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url && 'POST' == req.method) {
    var body = '';
    req.on('data', function (chunk) {
      body += chunk;
```

```
    });
    req.on('end', function () {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end('<p>Content-Type: ' + req.headers['content-type'] + '</p>'
        + '<p>Data:</p><pre>' + body + '</pre>');
    });
  }
}).listen(3000);
```

上述代码有什么变化呢？我们监听了data和end事件。创建了一个body字符串用来接收数据块，仅当end事件触发时，我们就知道数据接收完全了。

之所以可以这样逐块接收数据，是因为Node.js允许在数据到达服务器时就可以对其进行处理。因为数据是以不同TCP包到达服务器的，这和现实情况也完全匹配，我们先获取一部分数据，然后在某个时刻再获取其余的数据。

再次提交表单，响应结果如图7-13所示。

图7-13：本例中，我们将Content-Type内容以及请求的数据内容输出在页面上

101

举个例子，当使用Google进行搜索时，URL通常如图7-14所示。

图7-14：在搜索时，URL中高亮的部分为q=<search term>

注意，搜索部分的URL和表单内容一样都是经过编码的。这也解释了为什么Content-Type为urlencoded。

这部分URL片段又被称为查询字符串。

Node.js提供了一个称为querystring的模块，可以方便地对这类字符串进行解析，这样，我们就可以像处理头信息一样对其进行处理。我们来创建一个名为qs-example.js的文件，添加如下内容并运行（见图7-15）。

```
console.log(require('querystring').parse('name=Guillermo')); console.
 log(require('querystring').parse('q=guillermo+rauch'));
```

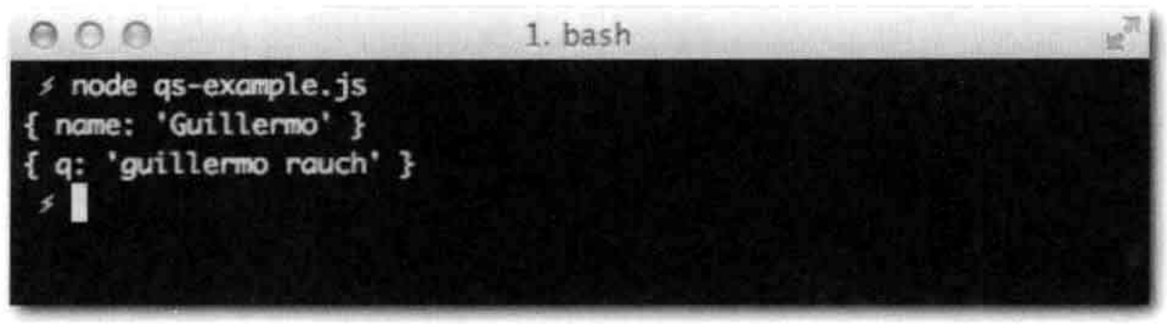



图7-15：调用parse函数后的结果

如图7-15所示，querystring模块将一个字符串解析成一个对象。这个解析处理方式和Node解析headers消息的方式类似，Node将HTTP请求数据中的headers信息从字符串解析成一个方便处理的headers对象。

接下来我们要使用querystring模块来获取提交表单的字段。

整合

现在我们要解析发送进来的数据并展示给用户。将server.js改为如下形式。注意了，这里我们在end事件中，使用querystring parse方法对请求内容进行解析，然后从解析生成的对象中获取name的值，并将其展示给用户。注意，这里的name是指<input>标签中的name值。我们来看一下截止到目前server.js的文件内容：

```
var qs = require('querystring');
require('http').createServer(function (req, res) {
   if ('/' == req.url) {
     res.writeHead(200, { 'Content-Type': 'text/html' });
     res.end([
         '<form method="POST" action="/url">'
       ,   '<h1>My form</h1>'
       ,   '<fieldset>'
       ,   '<label>Personal information</label>'
       ,   '<p>What is your name?</p>'
       ,   '<input type="text" name="name">'
       ,   '<p><button>Submit</button></p>'
```

```
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url && 'POST' == req.method) {
    var body = '';
    req.on('data', function (chunk) {
      body += chunk;
    });
    req.on('end', function () {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end('<p>Your name is <b>' + qs.parse(body).name + '</b></p>');
    });
  }
}).listen(3000);
```

接着打开浏览器访问，看到了吧（见图7-16）！ 103

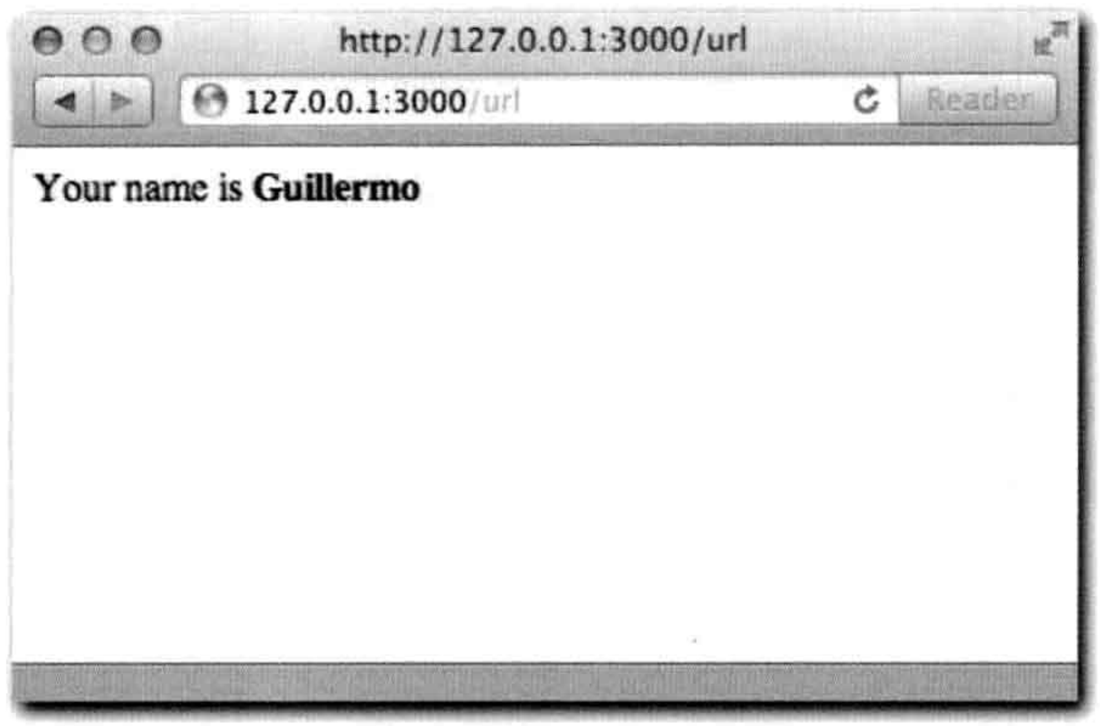


图7-16：name字段值

让程序更健壮

这里还有一个问题：要是URL没有匹配到任何判断条件，怎么办？

要是我们访问/test URL，你会发现服务器端一直都没有响应，浏览器一直都处在挂起的状态。

要解决这个问题，我们可以在服务器不知道如何处理该请求时，发送404(Not Found)状态码给客户端。注意在如下server.js代码中，我们添加了else的逻辑，并调用了writeHead写入404状态码。

```
var qs = require('querystring');
require('http').createServer(function (req, res) {
  if ('/' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form method="POST" action="/url">'
```

```
      ,   '<h1>My form</h1>'
      ,   '<fieldset>'
      ,   '<label>Personal information</label>'
      ,   '<p>What is your name?</p>'
      ,   '<input type="text" name="name">'
      ,   '<p><button>Submit</button></p>'
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url && 'POST' == req.method) {
    var body = '';
    req.on('data', function (chunk) {
      body += chunk;
    });
    req.on('end', function () {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end('<p>Your name is <b>' + qs.parse(body).name + '</b></p>');
    });
  } else {
    res.writeHead(404);
    res.end('Not Found');
  }
}).listen(3000);
```

104

好了，现在我们已经完成了第一个HTTP Web服务器了！尽管代码不是那么的整洁，不过不用担心，接下来的章节中会介绍如何更好地书写更复杂的HTTP服务器。

我们接下来介绍和服务器端API相对的HTTP客户端API。

一个Twitter Web客户端

学习如何使用Node.js向其他Web服务器发送请求是十分重要的。

HTTP已经演变成并非仅用于交换最终渲染、展示给用户的标记文本（如HTML），而且它还是服务器在不同网络环境传递数据的一种方式。同时，JSON因其语法衍生自JavaScrip线性对象，也快速成为了HTTP默认的标准数据格式，这也是Node.js在服务器端的优势之一。

在本例中，会介绍如何查询Twitter API，获取JSON数据，并解码为一种数据结构，以方便对其进行迭代后生成人类可读的形式。

创建模块

按照惯例，我们先从创建一个项目目录，并在该目录下创建一个package.json文件开始：

```
{
    "name": "tweet-client"
  , "description": "An HTTP tweets client"
```

```
  , "version": "0.0.1"
}
```

发送一个简单的HTTP请求

无独有偶，和我们创建的TCP客户端类似，我们通过http模块中的request静态方法创建一个Client对象。

为了让你更加熟悉，我们先回到此前典型的HTTP服务器：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello World');
}).listen(3000);
```

105

然后，写一个客户端来获取该响应，并将其在控制台以彩色的形式打印出来：

```
require('http').request({
    host: '127.0.0.1'
  , port: 3000
  , url: '/'
  , method: 'GET'
}, function (res) {
   var body = '';
   res.setEncoding('utf8');
   res.on('data', function (chunk) {
     body += chunk;
   });
   res.on('end', function () {
     console.log('\n  We got: \033[96m' + body + '\033[39m\n');
   });
 }).end();
```

上述代码中，首先调用了request方法。此方法用于初始化一个新的http.Client Request对象。

注意，我们收集信息块的方式和此前在服务器端收集客户端消息块的方式一样。连接的远程服务器会返回不同的数据块，我们需要将它们全部收集才能得到完整的响应。当然，也有可能所有的数据在一个 data事件中都到达了，不过我们无从得知。

所以，在本例中我们要监听end事件，然后将body数据打印到控制台。

除此之外，我们还通过响应对象上的setEncoding将编码设置为utf8，因为所有要输出到控制台的都是文本。你还可以试试用客户端下载一个PNG图片，这个时候用utf8来输出就不合适了。

下面，我们先运行服务器，然后再运行客户端（见图7-17）：

```
$ node client
```

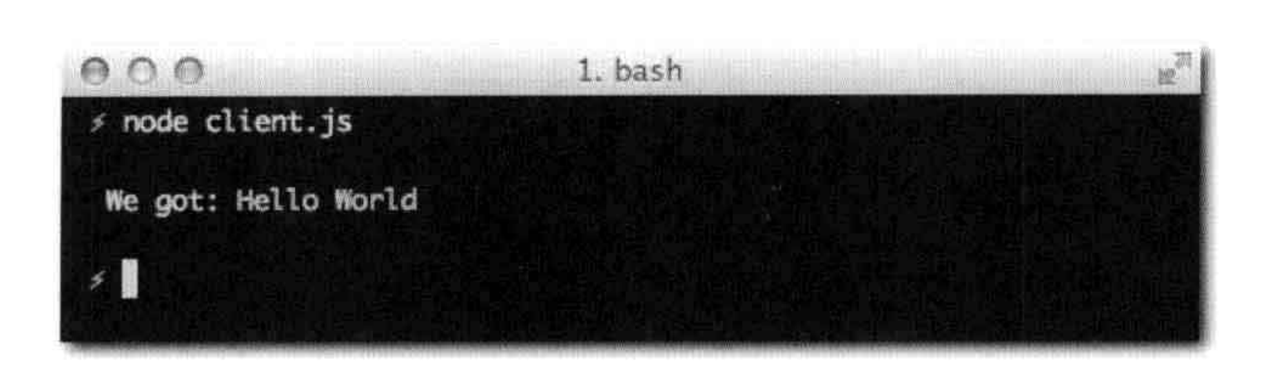


图7-17：从Hello World服务器返回的消息在客户端成功获取后显示了出来

106

下面，我们介绍如何通过请求发送数据。

发送数据

注意，在之前的例子中，调用完request之后，还需要调用end。

原因是在创建完一个请求之后，在发送给服务器前还可以和request对象进行交互。

比如，下面给你展示的就是这样一个发送数据给服务器的例子。

还记得之前我们在浏览器中创建的表单吗？这次仍旧创建该表单，不过不同的是，这次创建表单给客户端、使用Node，并且对于<form>，利用第5章学习的知识——stdin来处理。

服务器端处理表单：

```
var qs = require('querystring');
require('http').createServer(function (req, res) {
  var body = '';
  req.on('data', function (chunk) {
    body += chunk;
  });
  req.on('end', function () {
    res.writeHead(200);
    res.end('Done');
    console.log('\n  got name \033[90m' + qs.parse(body).name + '\033[39m\n');
  });
}).listen(3000);
```

客户端也要做对应的处理。通过使用querystring模块的stringify方法，可以将一个对象转化为url编码过的数据：

```
var http = require('http')
  , qs = require('querystring')

function send (theName) {
  http.request({
    host: '127.0.0.1'     , port: 3000
   , url: '/'
   , method: 'POST'
  }, function (res) {
```

```
      res.setEncoding('utf8');
      res.on('end', function () {
        console.log('\n  \033[90m  request complete!\033[39m');
process.stdout.write('\n  your name: ');
      });
  }).end(qs.stringify({ name: theName }));
}

process.stdout.write('\n  your name: ');
process.stdin.resume(); process.stdin.setEncoding('utf-8');
process.stdin.on('data', function (name) {
  send(name.replace('\n', ''));

});
```

107

注意在上述代码中，数据是通过end方法发送的，和我们在服务器端创建响应消息方式一样。

上述代码中，无须担心从服务器端获取数据块的问题。你知道，当end事件触发时，就可以将完整的请求数据打印出来，然后接着要求用户再次输入数据即可。

图7-18一步步展示了具体操作。在左侧，服务器端会显示由右侧提交的通过stdin输入的名字信息。

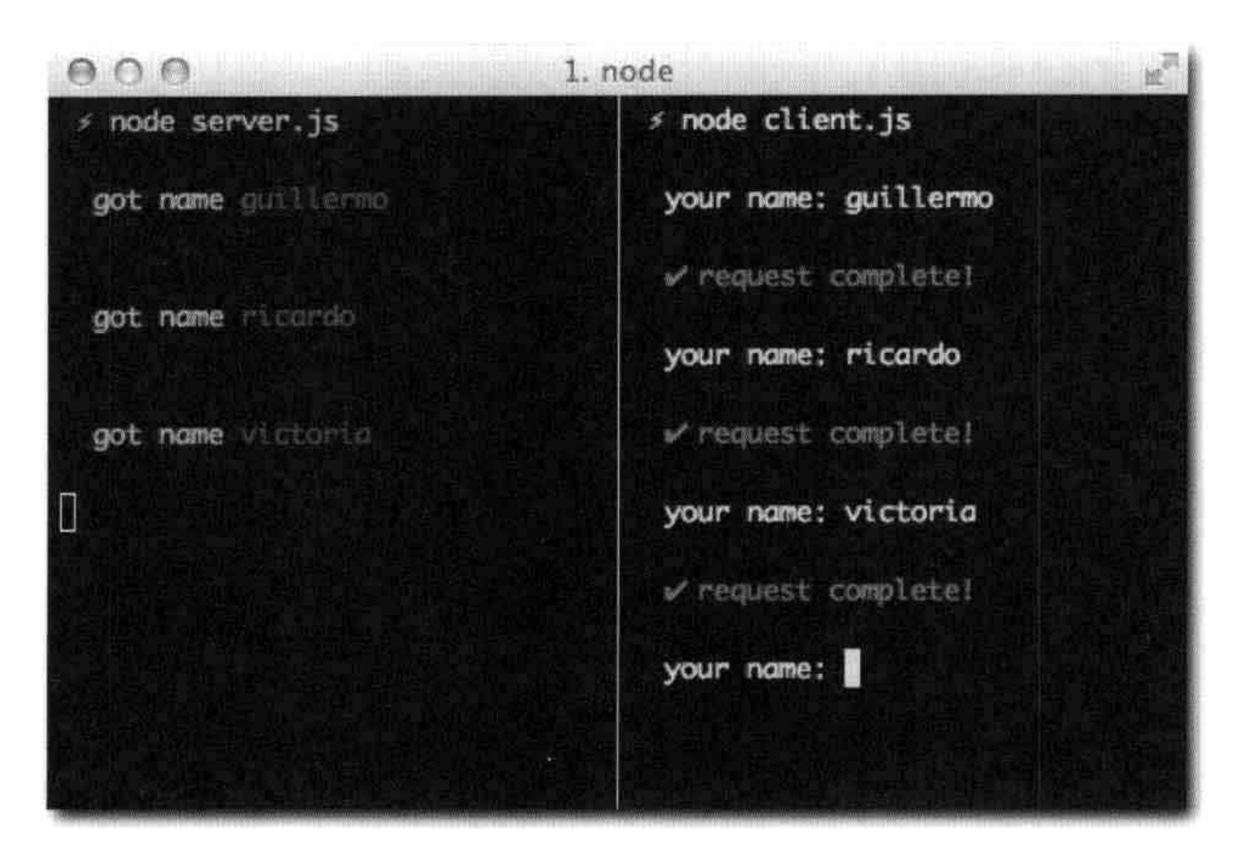


图7-18：当右侧提示输入名字时，我输入之后按下回车键，随后该名字就会在左侧服务器端显示出来

至此，你已经学会了如何通过请求发送数据了，已经了解了几乎全部请求API的用法。下面，让我们继续去完成目标！

获取推文

下面来点实际有用的东西！我们来创建一个tweets命令，该命令接受一个搜索参数，然后将最近相关话题的推文展示出来。

要是你看过Twitter开放的公共搜索API文档，你会发现其URL是这样的：http://search.twitter.com/search.json?q=blue+angels。

搜索结果数据如下所示（注意，这里我把后面的省略了）：

108

```
{
  "completed_in":0.031,
  "max_id":122078461840982016,
  "max_id_str":"122078461840982016",
  "next_page":"?page=2&max_id=122078461840982016&q=blue%20angels&rpp=5",
  "page":1,
  "query":"blue+angels",
  "refresh_url":"?since_id=122078461840982016&q=blue%20angels",
  "results":[     {
// …
```

同样的：搜索关键字也是url编码过的（q=blue+angels），并且结果是JSON数据格式。推文数据在响应结果对象中的results数组中。

由于我们要支持命令接收参数，就像第5章介绍的那样，我们需要访问argv。通过使用querystring模块，可以生成搜索URL，随后发送请求获取响应数据。这里访问资源请求的method就很明显是GET了，端口则是80，不过这些都是默认的，所以可以省略（为了让这首个HTTP客户端例子更清楚，我还是加上了GET选项）。

```
var qs = require('querystring')
  , http = require('http')

var search = process.argv.slice(2).join(' ').trim()

if (!search.length) {
  return console.log('\n  Usage: node tweets <search term>\n');
}
console.log('\n  searching for: \033[96m' + search + '\033[39m\n')
http.request({
    host: 'search.twitter.com'
  , path: '/search.json?' + qs.stringify({ q: search })
}, function (res) {
  var body = '';
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    body += chunk;
  });
  res.on('end', function () {
    var obj = JSON.parse(body);
    obj.results.forEach(function (tweet) {
      console.log('  \033[90m' + tweet.text + '\033[39m');
      console.log('  \033[94m' + tweet.from_user + '\033[39m');
      console.log('--');
```

```
    });
  });
}).end()
```

运行上述代码，会校验process.argv数组，看是否提供了搜索关键字（见图7-19），要是没有提供则会显示帮助信息。

109

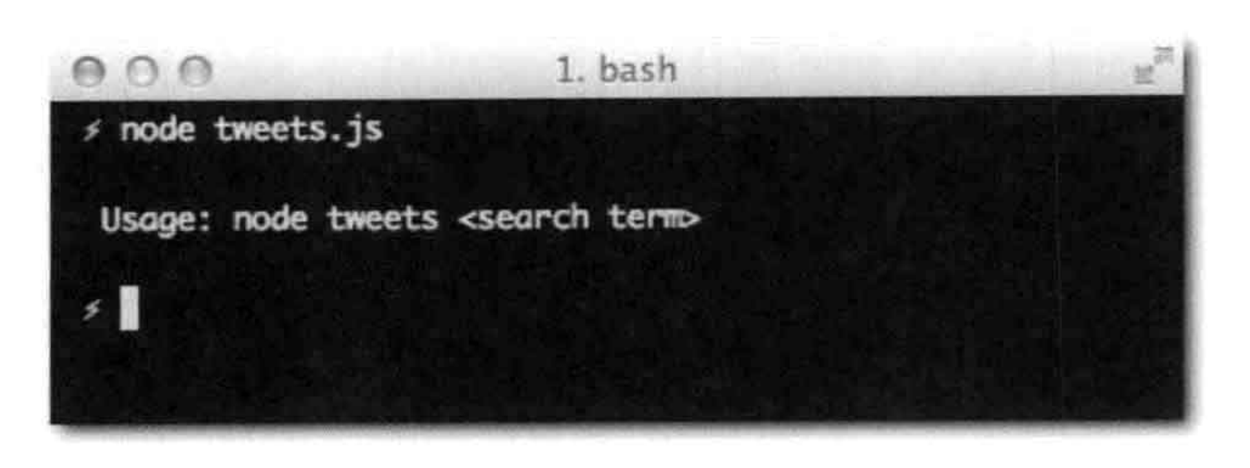


图7-19：不带关键字运行命令

当提供了搜索关键字时，就会执行搜索操作，如图7-20所示。Twitter会响应对应的JSON数据，在end事件处理器中会对该数据进行迭代，并将结果显示给用户。

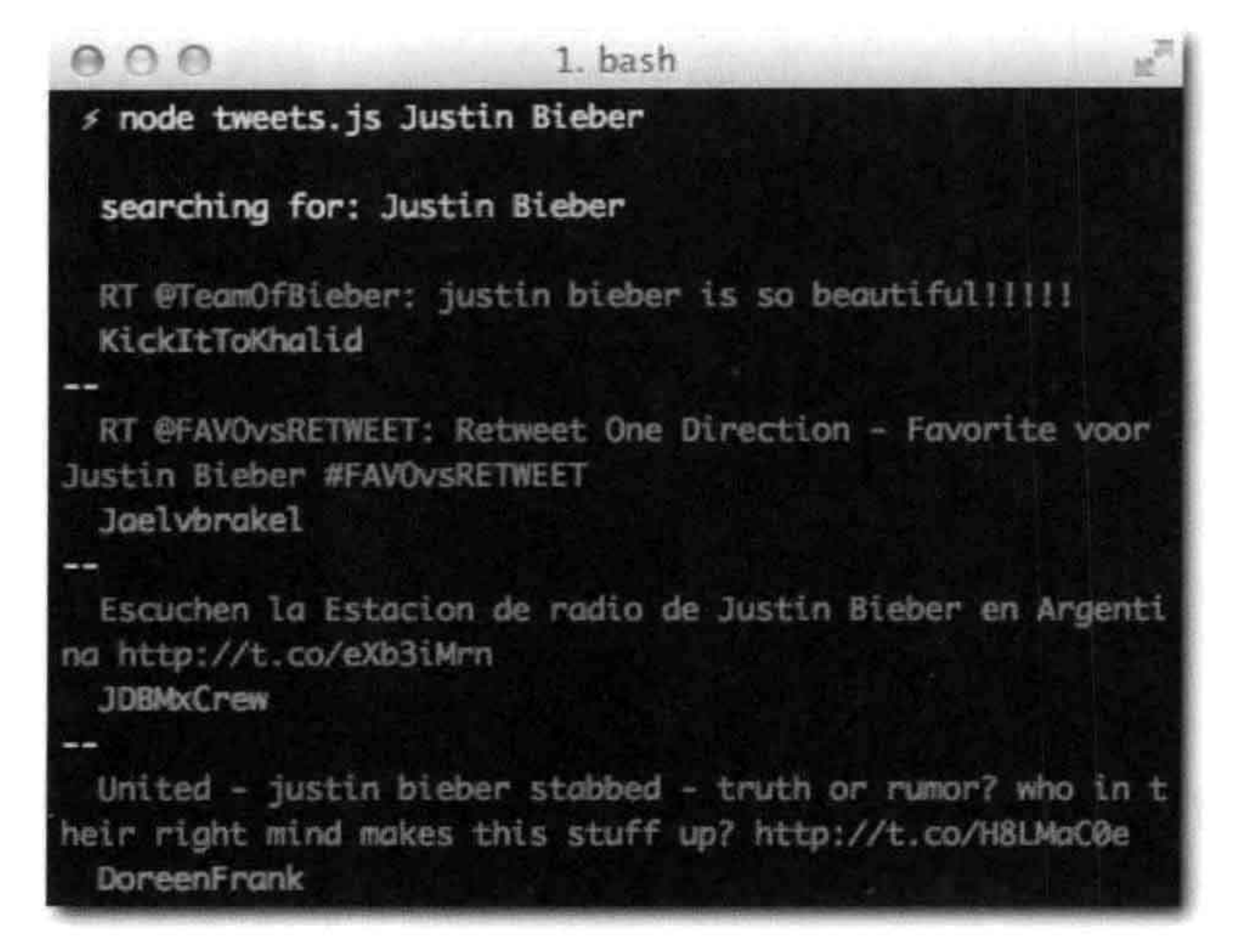


图7-20：本例中，我搜索 Justin Bieber，相关有趣的推文就显示出来了

至此，我们一直在频繁使用http.request，创建了可以说是最常见的GET请求。Web服务通常也会提供相比POST、PUT而言更多的GET服务。随着请求发送数据（一个请求体）也是相对不怎么常见的。

Node.js也为发送最常见的请求提供了便利，它提供了request.get方法。调用Twitter API（使用http.request）的代码可以重写为如下形式：

```
http.get({
    host: 'search.twitter.com'
  , path: '/search.json?' + qs.stringify({ q: search })
}, function (res) {
  var body = '';
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    body += chunk;
  });
  res.on('end', function () {
    var obj = JSON.parse(body);
    obj.results.forEach(function (tweet) {
      console.log('  \033[90m' + tweet.text + '\033[39m');
      console.log('  \033[94m' + tweet.from_user + '\033[39m');
      console.log('--');
    });
  });
})
```

110

唯一本质的不同就是这种方式就无须调用end方法了，并且从语义上更显然能够看出是要获取数据。因为API要接收一个method参数，其默认值是GET，所以这种方式更简单有效。

尽管做了上述改进，但我们还是有很多冗余代码。接下来会介绍一款工具superagent，它是基于HTTP客户端API的更高层封装，可以让上述这些处理变得更加容易。

superagent来拯救

HTTP客户端往往都会有一些共性：获取所有响应数据，根据响应消息的Content-Type值进行数据解析，处理消息数据。

当向服务器发送数据时，情况也是类似的。我们会创建一个POST请求，然后将要发送的数据对象编码为JSON格式。

有一个叫做superagent的模块通过扩展response对象，为其添加一些有用的扩展来解决上述问题，其中部分扩展功能下面会进行介绍。

本例中，我们使用superagent 0.3.0版本。创建一个新的目录，然后本地安装superagent：

```
$ npm install superagent@0.3.0
```

通过request获取数据，如果服务器响应了正确的Content-Type并且表明响应数据是JSON格式，那么superagent就会自动将其进行缓存并解析，然后将数据放在res.body中。我们来创建一个名为tweet.js的文件，并将下述内容添加到该文件中：

```
var request = require('superagent');
request.get('http://twitter.com/search.json')
```

```
  .send({ q: 'justin bieber' })
  .end(function (res) { console.log(res.body); });
```

运行上述文件，就能看到从响应的JSON数据解码成的对象形式的数据。通过res.text还是可以获取到原始的响应文本。

注意，对于查询字符串也无须进行手动编码，因为，superagent知道，当发送一个GET请求，并且要发送数据时，就需要将其编码为查询字符串来作为URL一部分。 111

要设置请求头信息，可以调用set方法。如下例子，我通过set方法设置了请求的Date头信息：

```
var request = require('superagent');
request.get('http://twitter.com/search.json')
  .send({ q: 'justin bieber' })
  .set('Date, new Date)
  .end(function (res) { console.log(res.body); });
```

send和set方法均可被调用多次，并且均为渐进式API：可以进行链式调用，并最后通过end方法来结束。

这类API不仅仅只用于GET请求。类似的，superagent还提供了put、post、head以及del方法。

下面的例子POST一个JSON编码的对象：

```
var request = require('superagent');
request.post('http://example.com/')
  .send({ json: 'encoded' })
  .end();
```

JSON是默认的编码格式。要更改，只需简单地调用set方法来更改请求的Content-Type值即可。

使用up重启HTTP服务器

至此，或许你已经意识到，每次修改服务器端的代码后，为了让通过浏览器访问能够看到修改后的效果，都要手动去重启服务器，是件很让人懊恼的事情。

最直接的解决这个问题的方法就是一旦发现代码有改动就去重启该进程。这在开发模式下的确很好，不过，一旦将Web服务器部署到生产环境之后，就得要确保正在执行的请求（换句话说，即当要重启进程时，还处在处理过程中的请求）不能被杀掉。

为此，我开发了一个名为up的工具，以一种安全可靠的方式来解决这个问题。开发模式下，通过NPM安装即可：

```
$ npm install -g up
```

112 接下来，需要确保代码结构必须要将Node HTTP服务器暴露出来，而不是调用listen来启动。这是因为up会调用listen方法，并且它需要访问服务器实例。举个例子，创建一个新目录，将下述内容添加到server.js文件中：

```
module.exports = require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('Hello <b>World</b>');
});
```

cd到该目录，通过up命令，同时传递--watch和--port选项来运行server.js文件：

```
$ up -watch -port 80 server.js
```

--watch意味着up会通过Node API来监听该目录下所有文件的更改。先试着通过浏览器访问该服务器，然后编辑server.js文件，更改Hello <b>World<b>文本。一旦保存修改的文件后，刷新浏览器就能立刻看到修改后的结果了！

小结

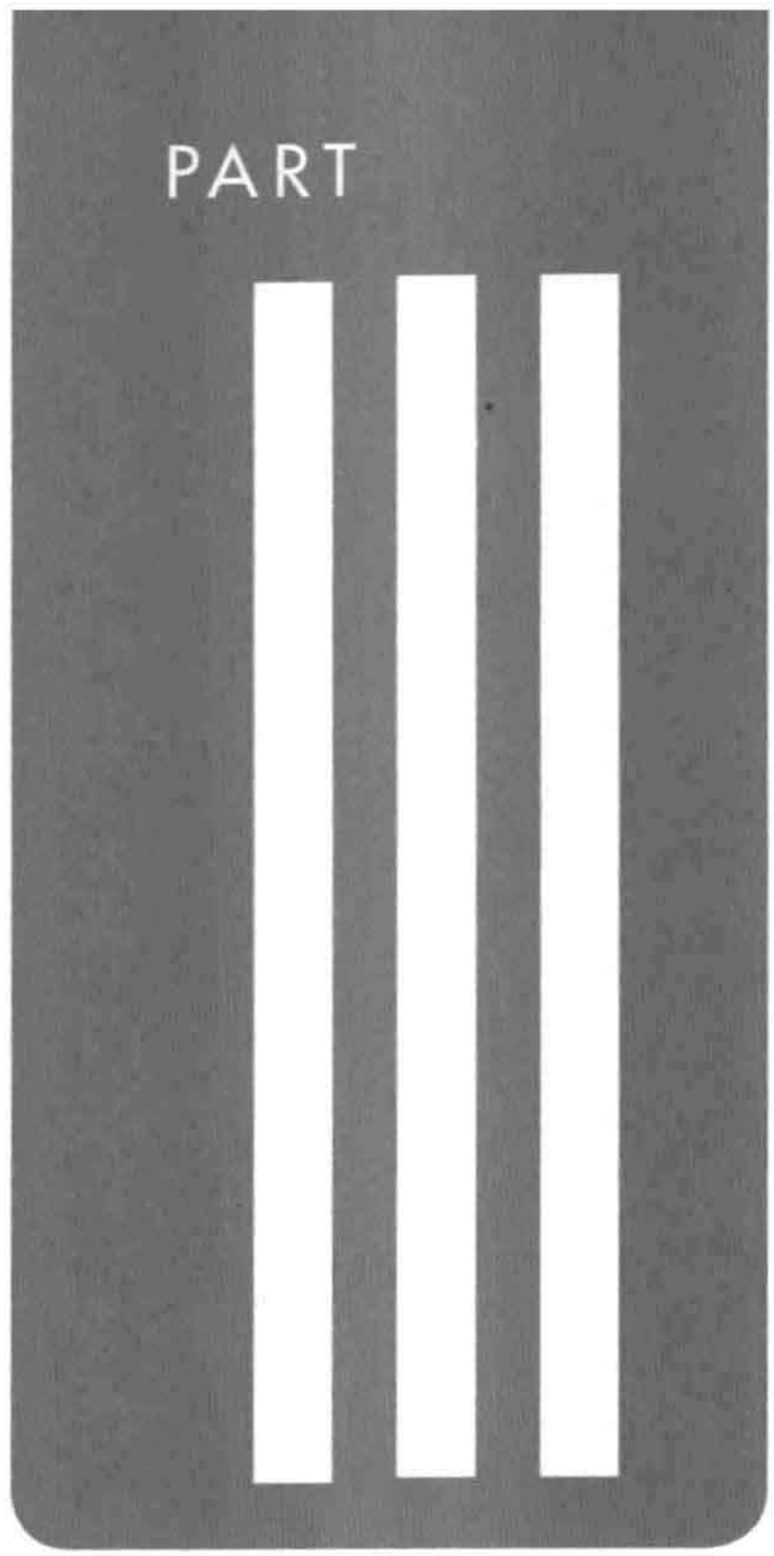
本章介绍了如何使用Node来书写HTTP服务器，本章从介绍基于TCP协议的HTTP协议基础开始。

然后，通过一个Hello World的例子展示了Node.js产生的默认响应结果。其中有默认的头信息，并介绍了为什么会有这些头信息。

之后介绍了HTTP请求中这些头信息的重要性，以及如何在服务器端响应中修改默认的头信息。还介绍了用于浏览器和服务器之间消息传递的编码，以及Node提供了哪些工具来对数据进行解析和处理。

成功完成了一个Web服务器之后，我们还介绍了Node客户端API，这对于在现代Web时代，与Web服务进行交互是非常有用的。再接着，我们在介绍了一些常规的用例后，成功地书写了一个查询Twitter API的客户端，不过在过程中也发现了很多代码都写重复了。为此，我们又介绍了一个核心Node.js之上的API来简化代码的书写。下一章要介绍的Connect模块也是类似的解决方案，它有很多值得学习的地方。

最后，介绍了up，一个命令行工具（同时也提供了JavaScript API），用它能够让开发过程变得更加简单，每次修改代码up就会自动重启服务器进程来让修改立即生效。要谨记：要使用up，必须确保模块要将通过调用createServer返回的http.Server实例暴露出来。



Web开发

CHAPTER 8 Connect

CHAPTER 9 Express

CHAPTER 10 WebSocket

CHAPTER 11 Socket.IO

CHAPTER

8 Connect

Node.js为常规的网络应用提供了基本的API。至此，你也已经了解了其为TCP服务器和基于此的HTTP服务器所提供的基本API了。

然而，实际情况下，绝大部分网络应用都需要完成一系列类似的操作，这些类似的操作你一定不想每次都重复地基于原始的API去实现。

Connect是一个基于HTTP服务器的工具集，它提供了一种新的组织代码的方式来与请求、响应对象进行交互，称为中间件（middleware）。

为了证明通过中间件进行代码复用的好处，假设我们有一个站点，其目录结构如下所示：

```
$ ls website/
index.html  images/
```

在images目录下，有四个图片文件：

```
$ ls website/images/
1.jpg  2.jpg  3.jpg  4.jpg
```

index.html简单地展示了这四张图片，并且可以通过http://localhost来访问（见图8-1）：

```
<h1>My website</h1>

<img src="/images/1.jpg">
<img src="/images/2.jpg">
<img src="/images/3.jpg">
<img src="/images/4.jpg">
```

116

图8-1：一个简单的静态网站，展示了Connect的能力

为了展示Connect为HTTP应用提供的便利，本章会介绍如何使用原生的`http` API书写一个简单的网站，之后再介绍如何使用`connect` API完成同样的事情。

使用HTTP构建一个简单的网站

按照惯例，我们先引入`http`模块用于创建服务器以及`fs`模块，用来读取文件：

```
/**
 * 模块依赖
 */

var http = require('http')
  , fs = require('fs')
```

接着，初始化服务器并处理请求—响应：

```
/**
 * 创建服务器
 */

var server = http.createServer(function (req, res) {
  // …
});
```

最后监听：

```
/**
 * 监听
 */

server.listen(3000);
```

117

回到createServer的回调函数。我们需要检查URL是否和服务器目录下的文件匹配，如果匹配，则读取该文件并展示出来。以后，可能会添加更多的图片，所以要确保足够灵活以支持这种情况。

首先要检查请求方法是GET并且URL以/images开始、.jpg结束。如果URL为'/'的话，则响应index.html（调用后面要实现的serve函数）。否则，发送404 Not Found（404未找到状态码），代码如下所示：

```
if ('GET' == req.method && '/images' == req.url.substr(0, 7)
  && '.jpg' == req.url.substr(-4)) {
  // …
} else if ('GET' == req.method && '/' == req.url) {
  serve(__dirname + '/index.html', 'text/html');
} else {
  res.writeHead(404);
  res.end('Not found');
}
```

接着使用fs.stat来检查文件是否存在。这里使用Node中的全局常量__dirname来获取当前服务器所在的路径。在首个if语句后，添加如下代码：

```
fs.stat(__dirname + req.url, function (err, stat) {

});
```

这里不使用同步版本的fs.stat（fs.statSync）。否则当处理磁盘文件时，会阻塞其他请求的处理，这是处理高并发的服务器的大忌。关于这些，我们早在第3章中就讨论过。

如果检查文件是否存在时发生错误，则终止进程并发送HTTP 404状态码告知无法找到请求的图片。对于stat成功但是路径所表示的并非是文件时，也要做此处理。如下所示的代码在fs.stat回调函数中。

```
if (err || !stat.isFile()) {
  res.writeHead(404);
  res.end('Not Found');
  return;
}
```

否则，就返回图片信息。下面这行代码紧随上述if之后：

118

```
serve(__dirname + req.url, 'application/jpg');
```

最后，我们完成serve方法，也许你已经猜到了，这个方法就是根据文件路径来获取文件内容，并添加必不可少的'Content-Type'头信息，如下所示，该信息告诉浏览器你发送的是什么类型的资源：

```
function serve (path, type) {
  res.writeHead(200, { 'Content-Type': type });
  fs.createReadStream(path).pipe(res);
}
```

还记得第6章中介绍的流（Stream）吗？HTTP响应对象是一个只写流。从文件创建出来的流是只读的。同时，可以将文件系统流接（pipe）到HTTP响应流中！上述简洁的代码其实就等同于如下这段代码：

```
fs.createReadStream(path)
  .on('data, function (data) {
    res.write(data);
  })
  .on('end', function () {
    res.end();
  })
```

这也是最有效的，推荐被用来实现静态文件托管功能的方法。

把所有代码整合起来，就变成了：

```
/**
 * 模块依赖
 */

var http = require('http')
  , fs = require('fs')

/**
 *创建服务器
 */

var server = http.createServer(function (req, res) {
  if ('GET' == req.method && '/images' == req.url.substr(0, 7)
    && '.jpg' == req.url.substr(-4)) {
    fs.stat(__dirname + req.url, function (err, stat) {
      if (err || !stat.isFile()) {
        res.writeHead(404);
        res.end('Not Found');
        return;
      }
      serve(__dirname + req.url, 'application/jpg');
    });
  } else if ('GET' == req.method && '/' == req.url) {
```

```
    serve(__dirname + '/index.html', 'text/html');
  } else {
    res.writeHead(404);
    res.end('Not found');
  }

  function serve (path, type) {
    res.writeHead(200, { 'Content-Type': type });
    fs.createReadStream(path).pipe(res);
  }
});

/**
 * 监听
 */

server.listen(3000);
```

完成！接下来就运行上述代码：

```
$ node server
```

然后通过浏览器访问http://127.0.0.1:3000，就能看到实现的网站了！

通过Connect实现一个简单的网站

接下来这个例子是要实现一个网站，该例子展示了创建网站时一些常见的任务：

- 托管静态文件。
- 处理错误以及损坏或者不存在的URL。
- 处理不同类型的请求。

基于http模块API之上的Connect，提供了一些工具方法能够让这些重复性的处理便于实现，以至于让开发者能够更加专注在应用本身。它很好地体现了DRY模式：不要重复自己（Don't Repeat Yourself）。

归功于Connect，本例实现起来会非常简单。首先在新目录下创建一个package.json文件，并在其中声明对"connect"模块的依赖。

```
{
    "name": "my-website"
  , "version": "0.0.1"
  , "dependencies": {
        "connect": "1.8.7"
    }
}
```

接着，安装依赖：

```
$ npm install
```

然后，引入connect模块：

```
/**
 * 模块依赖
 */

var connect = require('connect')
```

通过Connect创建http.Server：

```
/**
 * 创建服务器
 */

var server = connect.createServer();
```

使用use()方法来添加static中间件。下一部分会介绍中间件的概念，并在下一章会对其做深入讲解。现在，最为重要的是要理解，中间件其实就是一个简单的JavaScript函数。本例中，我们这样配置static中间件——通过传递一些参数给connect.static方法，该方法本身会返回一个方法。

```
/**
 * 处理静态文件
 */

server.use(connect.static(__dirname + '/website'));
```

将index.html以及images目录放到/website下，确保没有将不想托管的文件放进去。

接着，调用listen()方法：

```
/**
 * 监听
 */

server.listen(3000);
```

完成！事实上，Connect还能处理404的情况，你可以通过访问/made-up-url来验证。

121 中间件

为了更好地理解中间件，我们回到node HTTP的例子。移除逻辑，我们来关注如下部分：

```
if ('GET' == req.method && '/images' == req.url.substr(0, 7)) {
  // serve image
} else if ('GET' == req.method && '/' == req.url) {
```

```
  // serve index
} else {
  // display 404
}
```

如上述代码所示，针对每个请求应用都只会做这三件事情中的一件。比如，若还要记录请求日志，就在顶部加上如下代码：

```
console.error(' %s %s ', req.method, req.url);
```

下面，我们来设计一个更大型的应用，它能够根据每个请求的不同情况处理以下这几种不同的任务：

- 记录请求处理时间。
- 托管静态文件。
- 处理授权。

当然，这些任务的处理代码都可以放在一个简单的事件处理器中（createServer的回调函数中），这将会是一个非常复杂的处理过程。

简单来说，中间件由函数组成，它除了处理req和res对象之外，还接收一个next函数来做流控制。

若用中间件模式来书写满足上述要求的应用，会是这样的：

```
server.use(function (req, res, next) {
  //记录日志
  console.error(' %s %s ', req.method, req.url);
  next();
});

server.use(function (req, res, next) {
  if ('GET' == req.method && '/images' == req.url.substr(0, 7)) {
    // 托管图片
  } else {
    // 交给其他的中间件去处理
    next();
  }
});

server.use(function (req, res, next) {
  if ('GET' == req.method && '/' == req.url) {
    // 响应index文件
  } else {
    //交给其他中间件去处理
    next();
  }
```

122

```
});

server.use(function (req, res, next) {
  //最后一个中间件，如果到了这里，就意味着无能为力，只能返回404了
  res.writeHead(404);
  res.end('Not found');
});
```

使用中间件，不仅能够让代码有更强大的表达能力（将应用拆分为更小单元的能力），还能够实现很好的重用性。我们马上就会看到，Connect已经为处理常见的任务提供了对应的中间件。例如，要对请求进行日志记录，就可以简单地通过如下一行代码完成：

```
app.use(connect.logger('dev'))
```

帮你完成日志记录！

下一部分会介绍如何书写一个当请求处理时间过长而进行警告的中间件。

书写可重用的中间件

一个用于当请求时间过长而进行提醒的中间件在很多场景下都非常有用。比如，假设有个页面会向数据库发起一系列的请求。在测试过程中，所有响应都在100毫秒（ms）内完成，但是你要确保能够将响应时间大于100ms的请求记录下来。

为此，我们在一个名为`request-time.js`的独立模块中创建一个中间件。

这个模块暴露一个函数，此函数本身又返回一个函数。这对于可配置的中间件来说是很常见的写法。在前面的例子中，我们调用`connect.logger`时传递了一个参数，然后它自身会返回一个函数，最终用来处理请求。

目前，模块就接收一个超时时间阈值选项，该选项用来界定什么时候该将其记录下来。

123

```
/**
 * 请求时间中间件
 *
 * 选项:
 *    - 'time' ('Number'): 超时阈值（默认100）
 *
 * @param {Object} options
 * @api public
 */

module.exports = function (opts) {
  // …
};
```

首先，将默认阈值设为100：

```
var time = opts.time || 100;
```

随后，返回一个中间件函数：

```
return function (req, res, next) {
```

中间件本身创建一个计时器，并在指定时间内触发：

```
var timer = setTimeout(function () {
  console.log(
      '\033[90m%s %s\033[39m \033[91mis taking too long!\033[39m'
    , req.method
    , req.url
  );
}, time);
```

这里要确保当响应时间在100ms以内时要清除（停下来或者取消）计时器。另外一个在中间件中常用的模式叫做重写方法（也叫猴子补丁（monkey-patch）），能够在其他中间件调用它时，执行指定的行为。

在本例中，当响应结束时，我们需要清除计时器：

```
var end = res.end;
res.end = function (chunk, encoding) {
  res.end = end;
  res.end(chunk, encoding);
  clearTimeout(timer);
};
```

首先保持对原始函数的引用（`var end = res.end`）。然后，在重写的函数中，再恢复原始函数，并调用它，最后清除计时器。

最后，总是要让其他中间件能够处理请求，所以得调用`next`。否则，程序不会做任何事情！ 124

```
next();
```

完整版的中间件代码如下所示：

```
/**
 * 请求时间中间件
 *
 * 选项：
 *    - 'time' ('Number'): 超时阈值（默认100）
 *
 * @param {Object} options
 * @api public
 */
module.exports = function (opts) {
  var time = opts.time || 100;
```

```
    return function (req, res, next) {
      var timer = setTimeout(function () {
        console.log(
            '\033[90m%s %s\033[39m \033[91mis taking too long!\033[39m'
          , req.method
          , req.url
        );
      }, time);

      var end = res.end;
      res.end = function (chunk, encoding) {
        res.end = end;
        res.end(chunk, encoding);
        clearTimeout(timer);
      };
      next();
    };
  };
```

为了测试上述例子，我们需要创建一个Connect应用，并创建两条路由：第一条很快得到响应，另外一条1秒后得到响应。

我们先来引入依赖的模块：

```
# sample.js
/**
 * 模块依赖
 */

var connect = require('connect')
  , time = require('./request-time')
```

125

接着，创建服务器：

```
/**
 * 创建服务器
 */

var server = connect.createServer();
```

记录请求情况：

```
/**
 * 记录请求情况
 */

server.use(connect.logger('dev'));
```

实现中间件：

```
/**
 * 实现时间中间件
 */

server.use(time({ time: 500 }));
```

实现快速响应：

```
/**
 * 快速响应
 */

server.use(function (req, res, next) {
  if ('/a' == req.url) {
    res.writeHead(200);
    res.end('Fast!');
  } else {
    next();
  }
});
```

实现模拟的慢速响应：

```
/**
 * 慢速响应
 */

server.use(function (req, res, next) {
  if ('/b' == req.url) {
    setTimeout(function () {
      res.writeHead(200);

      res.end('Slow!');
    }, 1000);
  } else {
    next();
  }
});
```

服务器监听端口：

```
/**
 * 监听
 */

server.listen(3000);
```

运行服务器：

```
$ node server
```

用浏览器访问http://localhost:3000/a（快速响应），如图8-2所示。

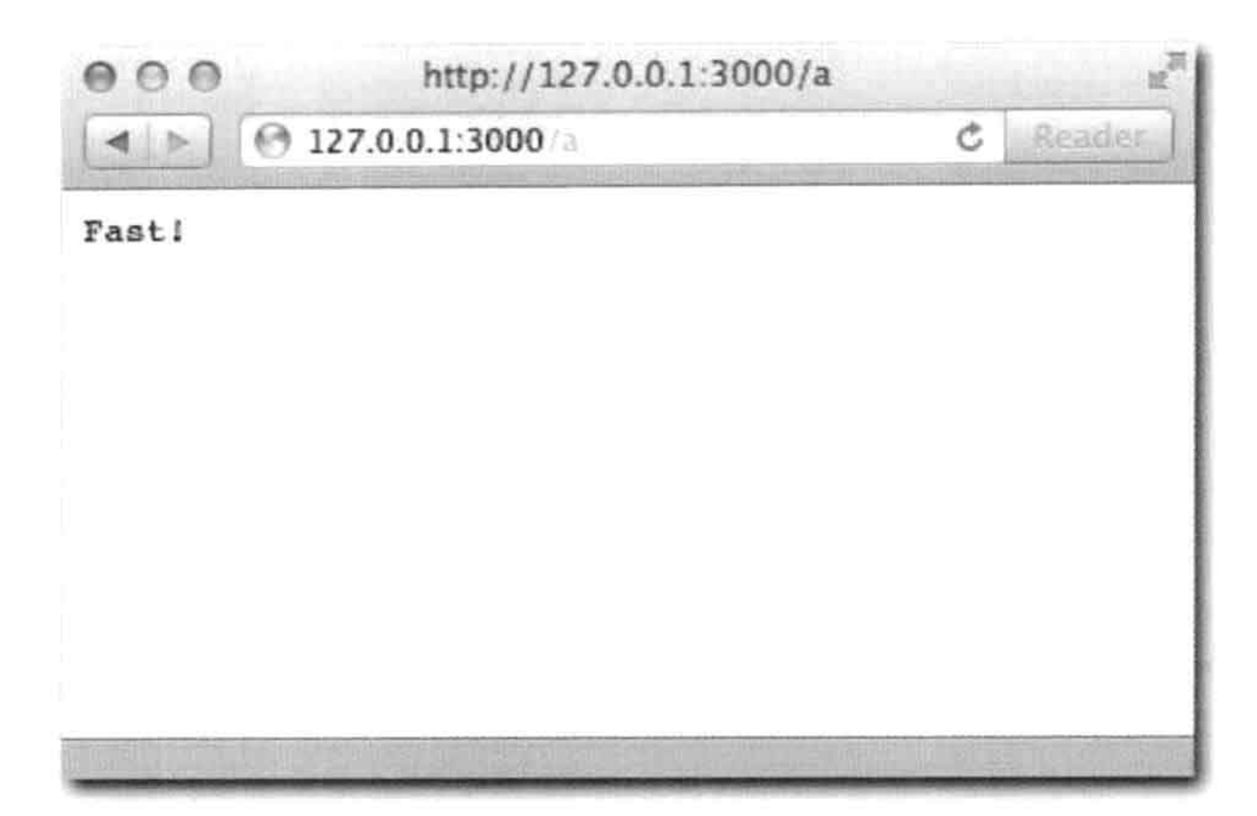


图8-2：一个简单的路由（/a）结果显示在浏览器中

要是看下控制台，就能看到logger中间件在工作了，如图8-3所示。

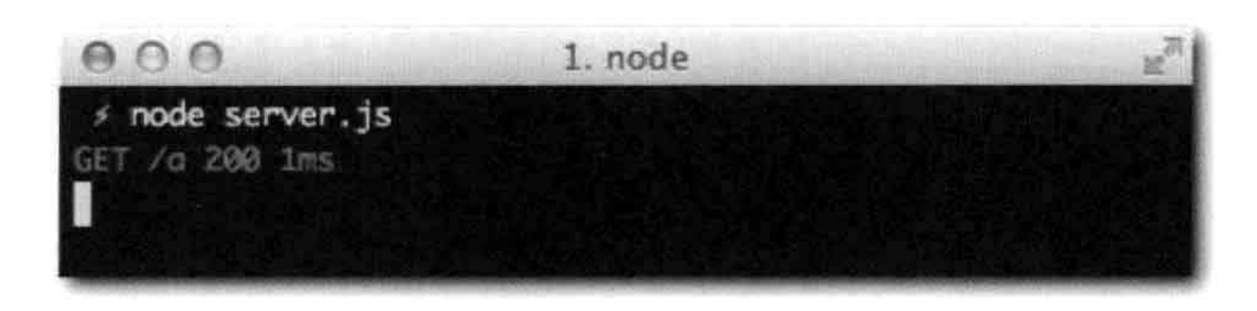


图8-3：在处理路由/a时，显示在控制台的日志

127

图8-4显示了慢速响应的结果（/b）。

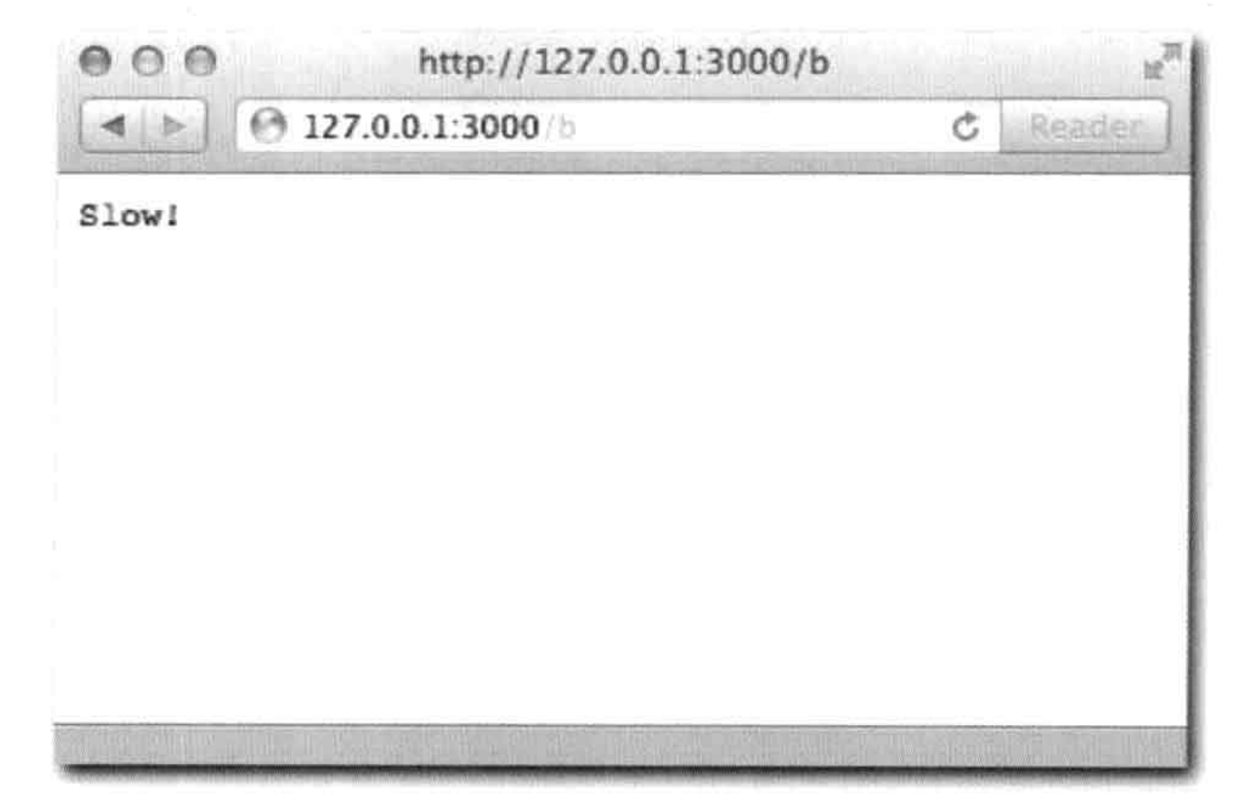


图8-4：慢速响应/b结果显示在浏览器中

图8-5显示了对应的控制台信息。

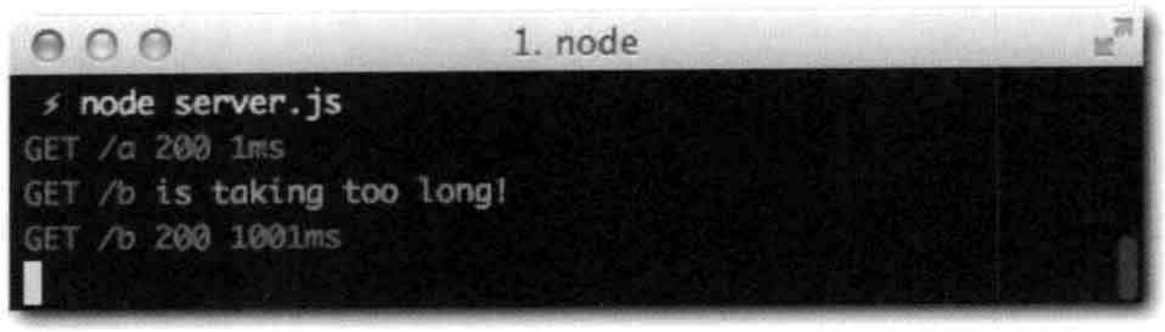


图8-5：处理/b后，控制台输出了警告日志

接下来，我们会介绍Connect内置的一些中间件，这些中间件在Web应用中都是非常常用的。

static中间件

static中间件可以算得上是使用Node开发Web应用时最常用的中间件之一了。

挂载

Connect允许中间件挂载到URL上。比如，static允许将任意一个URL匹配到文件系统中任意一个目录。

举例来说，假设要让/my-images URL和名为/images的目录对应起来，可以通过如下方式进行挂载：

```
server.use('/my-images', connect.static('/path/to/images'));
```

maxAge

128

static中间件接收一个名为maxAge的选项，这个选项代表一个资源在客户端缓存的时间。这非常实用，特别是对一些不经常改动的资源来说，浏览器就无须每次都去请求它了。

比如，一种Web应用常见的实践方式就是将所有的客户端JavaScript文件都合并到一个文件中，并在其文件名中加上修订号。这个时候，就可以设置maxAge选项，让其永远缓存起来。

```
server.use('/js', connect.static('/path/to/bundles', { maxAge: 10000000000000 }));
```

hidden

另一个static接收的参数是hidden。如果该选项为true，Connect就会托管那些文件名以点（.）开始的在UNIX文件系统中被认为是隐藏的文件：

```
server.use(connect.static('/path/to/resources', { hidden: true }));
```

query中间件

有时要发送给服务器的数据会以查询字符串形式，作为URL的一部分。

比如，url /blog-posts?page=5。当在浏览器中访问该URl时，Node会以字符串的形式将该URL存储到req.url变量中：

```
server.use(function (req) {
  // req.url == "/blog-posts?page=5"
});
```

而绝大多数情况下，真正想要获取的其实是查询字符串这部分的数据。

使用query中间件，就能够通过req.query对象自动获取这些数据：

```
server.use(connect.query);
server.use(function (req, res) {
  // req.query.page == "5"
});
```

同样的，解析查询字符串也是应用常见的需求，Connect也对其做了简化。就像刚刚提到的static中间件，你无须关心使用require去引入fs模块的问题一样，你也无须去担心使用querystring模块的问题。

query中间件在Express中默认就是启用的，Express是一个Web框架，我们会在下一章中做介绍。接下来要介绍另一个很实用的中间件——logger。

129 logger中间件

logger中间件是一个对Web应用非常有用的诊断工具。它将发送进来的请求信息和发送出去的响应信息打印在终端。

它提供了以下四种日志格式：

- default
- dev
- short
- tiny

比如，要使用dev日志格式，可以通过如下初始化logger中间件的方式：

```
server.use(connect.logger('dev'));
```

看下面这个例子，一个典型的使用了logger中间件的“Hello World”HTTP 服务器：

```
var connect = require('connect');
connect.createServer(
    connect.logger('dev')
  , function (req, res) {
      res.writeHead(200);
      res.end('Hello world');
    }
).listen(3000);
```

注意，在上述代码中，我传递了一系列中间件函数作为createServer的参数。这其实就等同于初始化Connect服务器时调用use两次。

当通过浏览器访问http://127.0.0.1:3000时，就能看到如下两行输出信息：

```
GET / 200 0ms
GET /favicon.ico 200 2ms
```

上述信息表示浏览器正在请求/favicon.ico和/，connect logger中间件将请求的方法、响应状态码以及处理时间输出。

dev是一种精准简短的日志格式，能够提供行为以及性能方面的信息，方便测试Web应用。

logger中间件还允许自定义日志输出格式。

比如，若只想记录请求方法和IP地址：

130

```
server.use(connect.logger(':method :remote-addr'));
```

另外，还能够通过动态的req和res来记录头信息。要记录响应的content-length和content-type信息，可以通过如下方式：

```
server.use(connect.logger('type is :res[content-type], length is '
  + ':res[content-length] and it took :response-time ms.'));
```

注意：记住，在Node中，请求/响应头都是小写的。

要是把上述代码应用到请求四个图片的例子，就能看到很有趣的输出结果，如图8-6所示。（确保在没有缓存的情况下。）

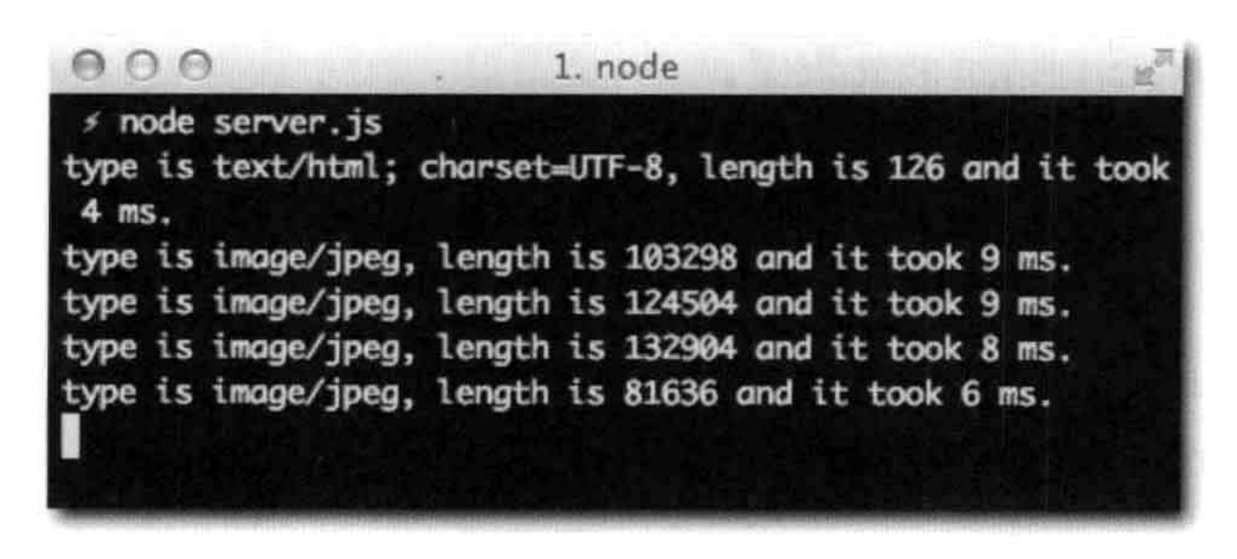


图8-6：自定义日志格式实践

下面是完整的可用token：

- :req[header]（如:req[Accept]）
- :res[header]（如:res[Content-Length]）
- :http-version
- :response-time
- :remote-addr
- :date
- :method
- :url
- :referrer
- :user-agent
- :status

131

logger还能够自定义token。比如，要给请求的Content-Type定义一个简写的:type token，就可以采用如下方式：

```
connect.logger.token('type', function (req, res) {
  return req.headers['content-type'];
});
```

接下来要介绍的是body parse中间件，它可以自动帮你完成另一件开发Web应用常见的任务。

body parser中间件

在一个使用http模块的例子中，我们用了qs模块来解析POST请求的消息体。

Connect也对这部分提供了帮助！它提供了bodyParser中间件：

```
server.use(connect.bodyParser());
```

然后，就能够在req.body中获取POST请求的数据了：

```
server.use(function (req, res) {
  // req.body.myinput
});
```

如果客户端在POST请求中使用了JSON格式，那么req.body也会对应地转换为JSON对象，因为bodyParser会检测Content-Type的值。

处理上传

bodyParser另一个功能就是使用formidable模块，它可以让你处理用户上传的文件。

本例中，我们可以使用createServer的快捷方式来创建一个服务器，并将所有要用的中间件都传递给它：

```
var server = connect(
    connect.bodyParser()
  , connect.static('static')
);
```

在static/文件夹中，我们创建一个包含用来处理文件上传表单的index.html文件：

```
<form action="/" method="POST" enctype="multipart/form-data">
  <input type="file" />
  <button>Send file!</button>
</form>
```

接着，我们可以添加一个简单的中间件来看一下req.body.file值是什么样子的： 132

```
function (req, res, next) {
  if ('POST' == req.method) {
    console.log(req.body.file);
  } else {
    next();
  }
}
```

到测试时间了！如图8-7所示，上传一个Hello.txt文件来测试一下。

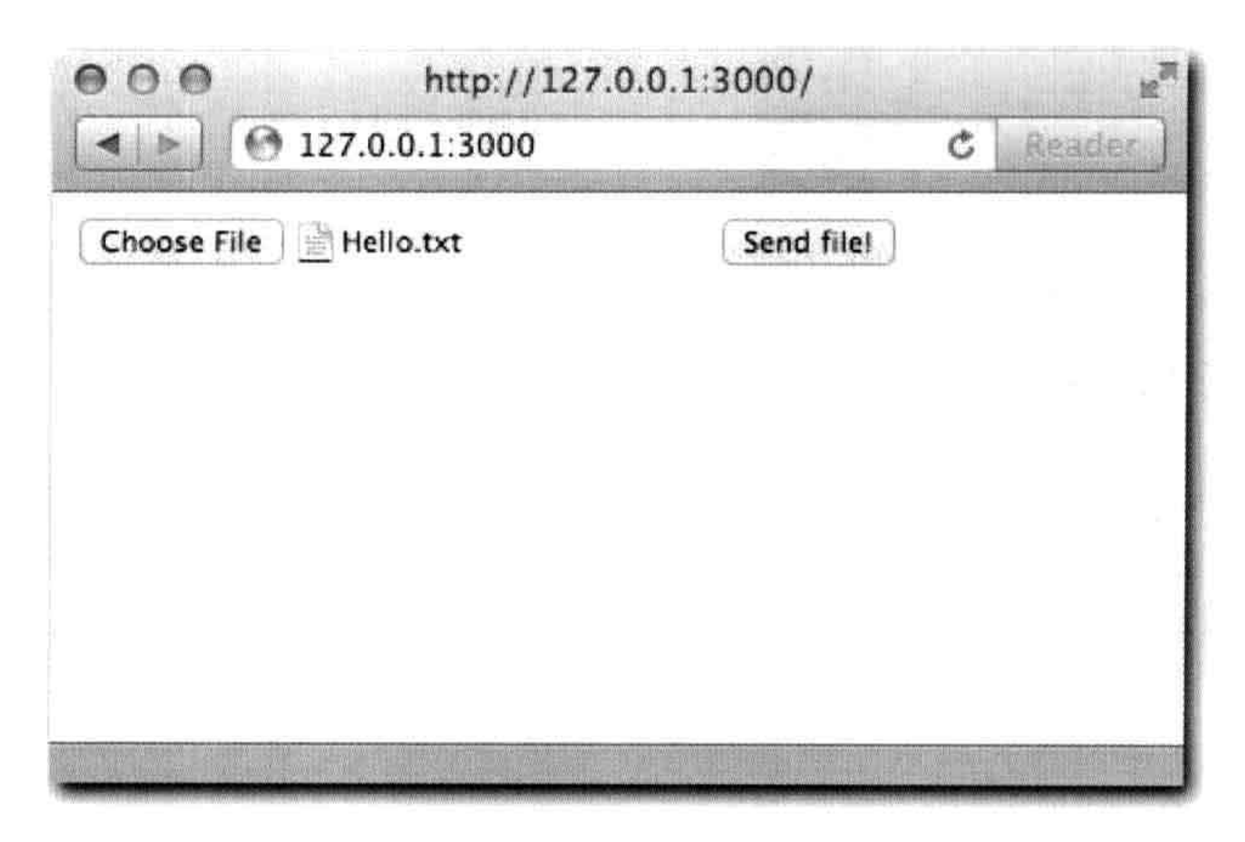


图8-7：从浏览器上传一个示例文本文件

接着，我们看看服务端的输出结果，如图8-8所示。

如图8-8中所示，我们获取到一个包含了许多有用的上传信息的对象。接着，我们把上传

的文件返回给用户：

```
if ('POST' == req.method && req.body.file) {
  fs.readFile(req.body.file.path, 'utf8', function (err, data) {
    if (err) {
      res.writeHead(500);
      res.end('Error!');
      return;
    }

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<h3>File: ' + req.body.file.name + '</h3>'
      , '<h4>Type: ' + req.body.file.type + '</h4>'
      , '<h4>Contents:</h4><pre>' + data + '</pre>'
    ].join(''));

  });
} else {
  next();
}
```

133

```
1. node
⚡ node server.js
{ size: 23,
  path: '/tmp/042e078b76ccd06995fefda8d270c639',
  name: 'Hello.txt',
  type: 'text/plain',
  lastModifiedDate: Mon, 30 Apr 2012 21:06:25 GMT,
  _writeStream:
   { path: '/tmp/042e078b76ccd06995fefda8d270c639',
     fd: 8,
     writable: false,
     flags: 'w',
     encoding: 'binary',
     mode: 438,
     bytesWritten: 23,
     busy: false,
     _queue: [],
     drainable: true },
  length: [Getter],
  filename: [Getter],
  mime: [Getter] }
```

图8-8： 在req.body包含的上传文件对象，显示在控制台中

再次上传该文件来看看返回结果（见图8-9）。

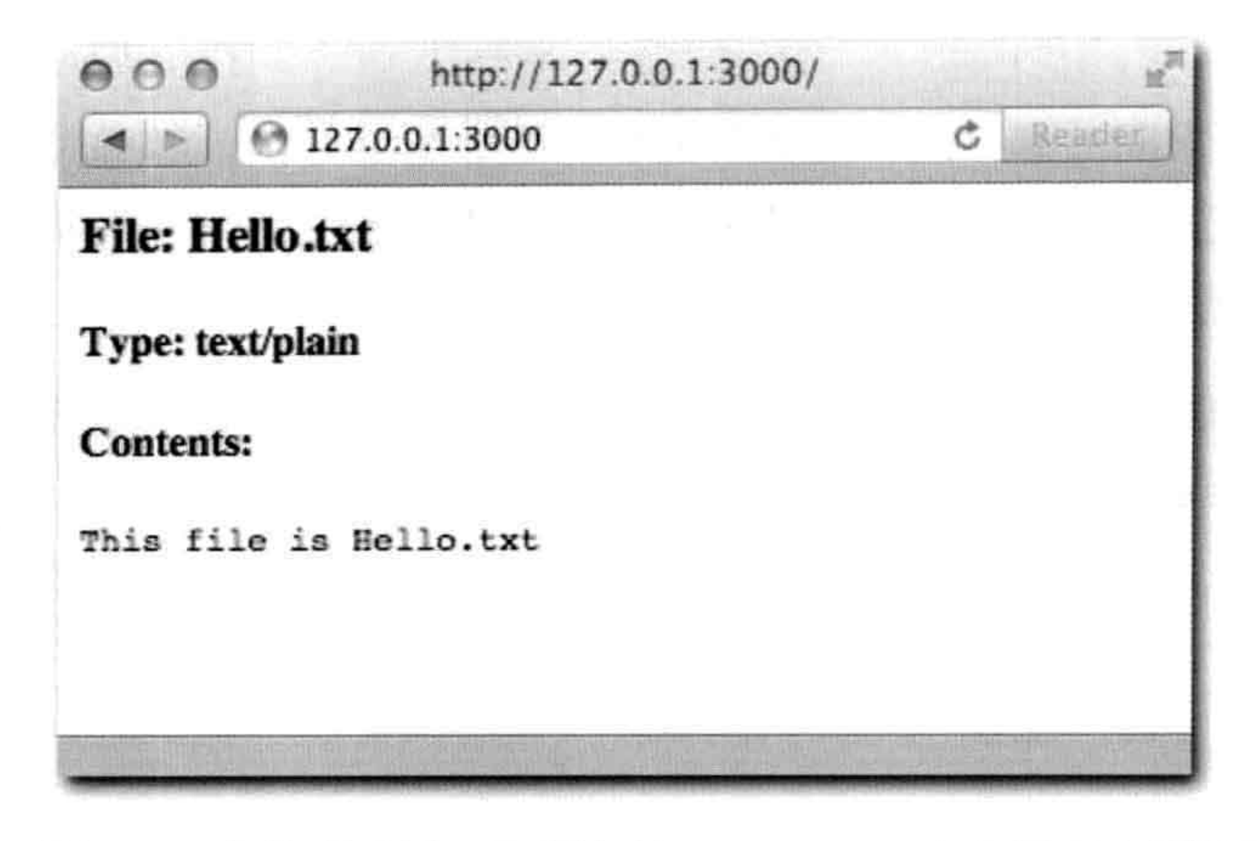


图8-9：上传成功后，示例文件Hello.txt内容显示在浏览器中

多文件上传

134

要处理多文件上传也很简单，只要在input的name属性上加上[]即可：

```
<input type="file" name="files[]" />
<input type="file" name="files[]" />
```

req.body.files就包含了一个数组，数组中的元素就是此前单个Hello.txt示例的对象。

cookie

除了query之外，Connect还为读写cookie提供了便利。

当浏览器发送cookie数据时，会将其写到Cookie头信息中。其数据格式和URL中的查询字符串类似。我们来看下面这个包含了该头信息的请求：

```
GET /secret HTTP/1.1
Host: www.mywebsite.org
Cookie: secret1=value; secret2=value2
Accept: */*
```

要访问这些值（secret1和secret2），但又不想手动去解析，也不想使用正则表达式去抽取的话，就可以使用cookieParser中间件：

```
server.use(connect.cookieParser())
```

然后，就可以通过req.cookies对象来访问这些cookie数据了：

```
server.use(function (req, res, next) {
  // req.cookies.secret1 = "value"
  // req.cookies.secret2 = "value2"
})
```

会话（session）

在绝大多数Web应用中，多个请求间共享“用户会话”的概念是非常必要的。“登录”一个网站时，多多少少会使用某种形式的会话系统，它主要通过在浏览器中设置cookie来实现，该cookie信息会在随后所有的请求头信息中被带回到服务器。

Connect为此也提供了简单的实现方式。作为例子，我们创建了一个简单的登录系统。我们把用户凭证信息存放在一个名为users.json的文件中：

135

```
{
    "tobi": {
        "password": "ferret"
      , "name": "Tobi Holowaychuk"
    }
}
```

首先，通过require载入Connect和users文件：

```
/**
 * 模块依赖
 */

var connect = require('connect')
  , users = require('./users')
```

注意，这里直接require了JSON文件！当你只是要对外暴露数据时，就不需要加上module.exports，直接把数据文件以JSON形式暴露出来就好了。

接着，我们加上logger、bodyParser和session中间件。由于session中间件需要操作cookie，所以在这之前先要引入cookieParser中间件：

```
var server = connect(
    connect.logger('dev')
  , connect.bodyParser()
  , connect.cookieParser()
  , connect.session({ secret: 'my app secret' })
```

出于安全性的考虑，在初始化session中间件的时候需要提供secret选项。

接下来，我们首先检查用户是否已经登录。如果没有登录，则交给其他中间件处理：

```
, function (req, res, next) {
    if ('/' == req.url && req.session.logged_in) {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(
          'Welcome back, <b>' + req.session.name + '</b>. '
        + '<a href="/logout">Logout</a>'
      );
    } else {
```

```
      next();
    }
  }
```

紧接着的中间件，展示一个登录表单：

```
, function (req, res, next) {
    if ('/' == req.url && 'GET' == req.method) {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end([
          '<form action="/login" method="POST">'
        ,   '<fieldset>'
        ,     '<legend>Please log in</legend>'
        ,     '<p>User: <input type="text" name="user"></p>'
        ,     '<p>Password: <input type="password" name="password"></p>'
        ,     '<button>Submit</button>'
        ,   '</fieldset>'
        , '</form>'
      ].join(''));
    } else {
      next();
    }
  }
```

136

再后面的中间件检查登录表单的信息是否与用户凭证匹配，匹配则认为登录成功：

```
, function (req, res, next) {
    if ('/login' == req.url && 'POST' == req.method) {
      res.writeHead(200);
      if (!users[req.body.user] || req.body.password != users[req.body.user].
 password) {
        res.end('Bad username/password');
      } else {
        req.session.logged_in = true;
        req.session.name = users[req.body.user].name;
        res.end('Authenticated!');
      }
    } else {
      next();
    }
  }
```

注意在上述代码中，修改了一个名为req.session的对象。该对象在响应发送出去时就会自动保存，无须我们手动处理。上述代码中将在session对象上存储name，以及将logged_in标记为true。

最后处理登出：

```
, function (req, res, next) {
    if ('/logout' == req.url) {
      req.session.logged_in = false;
```

```
      res.writeHead(200);
      res.end('Logged out!');
    } else {
      next();
    }
  }
```

137 完整的代码如下所示：

```
/**
 * Module dependencies
 */

var connect = require('connect')
  , users = require('./users')

/**
 * Create server
 */

var server = connect(
    connect.logger('dev')
  , connect.bodyParser()
  , connect.cookieParser()
  , connect.session({ secret: 'my app secret' })
  , function (req, res, next) {
      if ('/' == req.url && req.session.logged_in) {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end(
            'Welcome back, <b>' + req.session.name + '</b>. '
          + '<a href="/logout">Logout</a>'
        );
      } else {
        next();
      }
    }
  , function (req, res, next) {
      if ('/' == req.url && 'GET' == req.method) {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end([
            '<form action="/login" method="POST">'
          ,   '<fieldset>'
          ,     '<legend>Please log in</legend>'
          ,     '<p>User: <input type="text" name="user"></p>'
          ,     '<p>Password: <input type="password" name="password"></p>'
          ,     '<button>Submit</button>'
          ,   '</fieldset>'
          , '</form>'
        ].join(''));
      } else {
```

```
      next();
    }
  }
, function (req, res, next) {
    if ('/login' == req.url && 'POST' == req.method) {
      res.writeHead(200);
      if (!users[req.body.user] || req.body.password != users[req.body.user].
password) {
        res.end('Bad username/password');
      } else {
        req.session.logged_in = true;
        req.session.name = users[req.body.user].name;
        res.end('Authenticated!');
      }
    } else {
      next();
    }
  }
, function (req, res, next) {
    if ('/logout' == req.url) {
      req.session.logged_in = false;
      res.writeHead(200);
      res.end('Logged out!');
    } else {
      next();
    }
  }
);

/**
 * Listen.
 */

server.listen(3000);
```

138

下面来试试这个简单的登录系统。首先来测试一下基本的认证功能是否正常工作，如图8-10和图8-11所示。

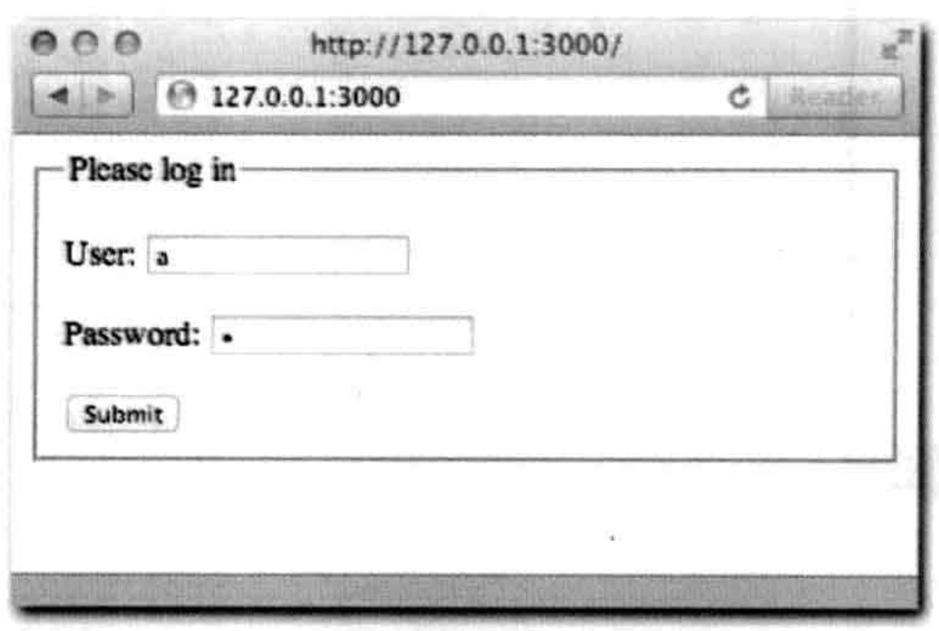


图8-10：使用错误的认证信息登录

139

图8-11：结果显示登录失败

下面，用正确的信息登录，如图8-12所示。

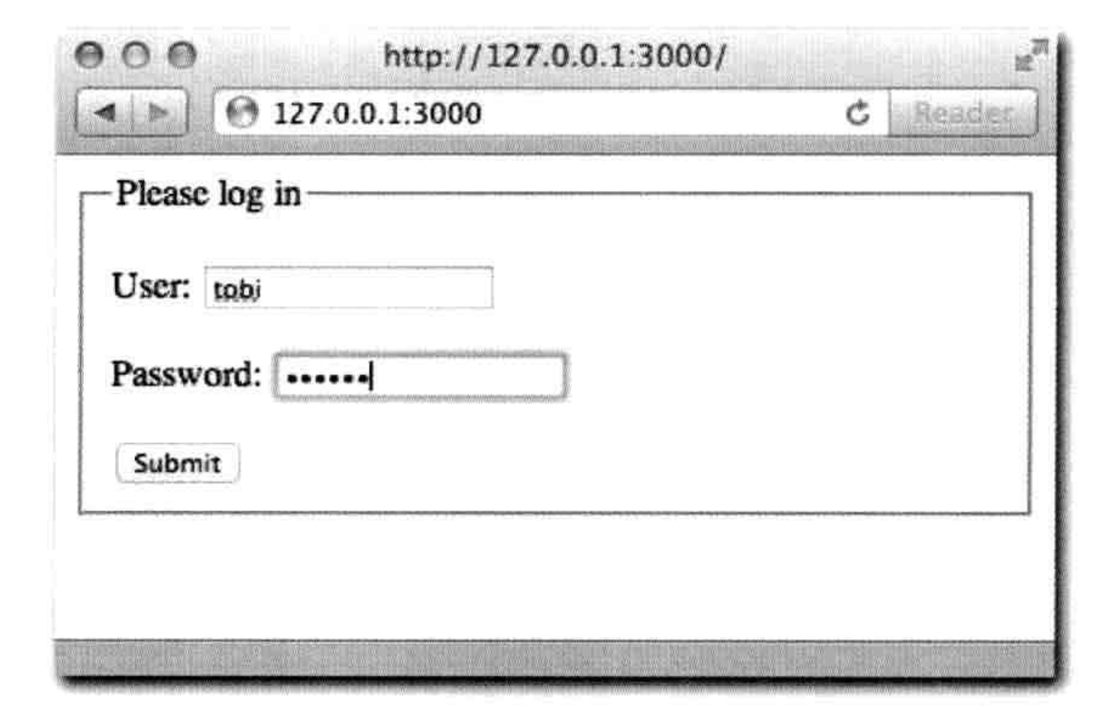


图8-12：有效用户登录

图8-13展示了成功登录的情况。

图8-13：登录成功

如图8-14所示，登录成功后，返回首页。 140

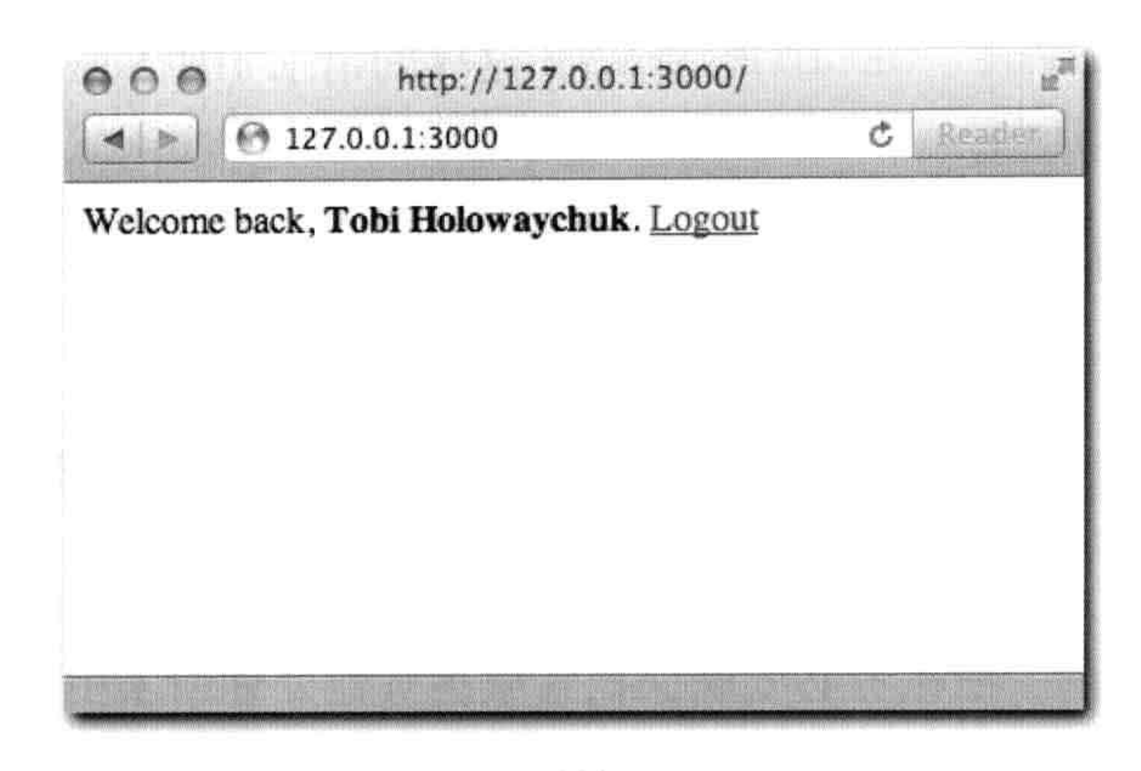


图8-14：登录成功后回到首页

为了让senssion能够在生产环境中也正常工作，我们需要学习如何通过Redis来实现一个持久化的session。

Redis session

尝试这样一件事情：登录后，重启node服务器，然后刷新浏览器。注意到没有，session丢了！

原因就在于session默认的存储方式是在内存。这就意味着session数据都是存储在内存中的，当进程退出后，session数据自然也就丢失了。

这对于开发过程并不算什么坏事，但是对于线上的应用来说，简直就是不能忍受的。生产环境中，需要使用一种当应用重启后，还能够将session信息持久化存储下来的机制，如Redis（第12章会详细介绍 Redis）。

Redis是一个既小又快的数据库，有一个`connect-redis`模块使用Redis来持久化session数据，这样就让session驻扎到了Node进程之外。

通过如下方式来使用该模块（必须要安装好Redis）：

```
var connect = require('connect')
  , RedisStore = require('connect-redis')(connect);
```

然后，将其作为`store`选项的值传递给`session`中间件：

```
server.use(connect.session({ store: new RedisStore, secret: 'my secret' }))
```

完成！现在session已经脱离Node进程了。

141

methodOverride中间件

一些比较早期的浏览器并不支持创建如PUT、DELETE、PATCH这样的请求（如Ajax）。常见的解决方案就是在GET或者POST请求中加上一个_method变量来模拟上述这些请求。

举例来说，要想在IE中发送一个PUT请求，可以采用如下方式：

```
POST /url?_method=PUT HTTP/1.1
```

为了让后台的处理程序能够觉得这是一个PUT请求，可以使用methodOverride中间件：

```
server.use(connect.methodOverride())
```

这里要记住的是，中间件是串行执行的，所以务必要确保把它放在其他处理请求中间件之前。

basicAuth中间件

有些项目需要对客户端进行基本的身份验证（见图8-15）。

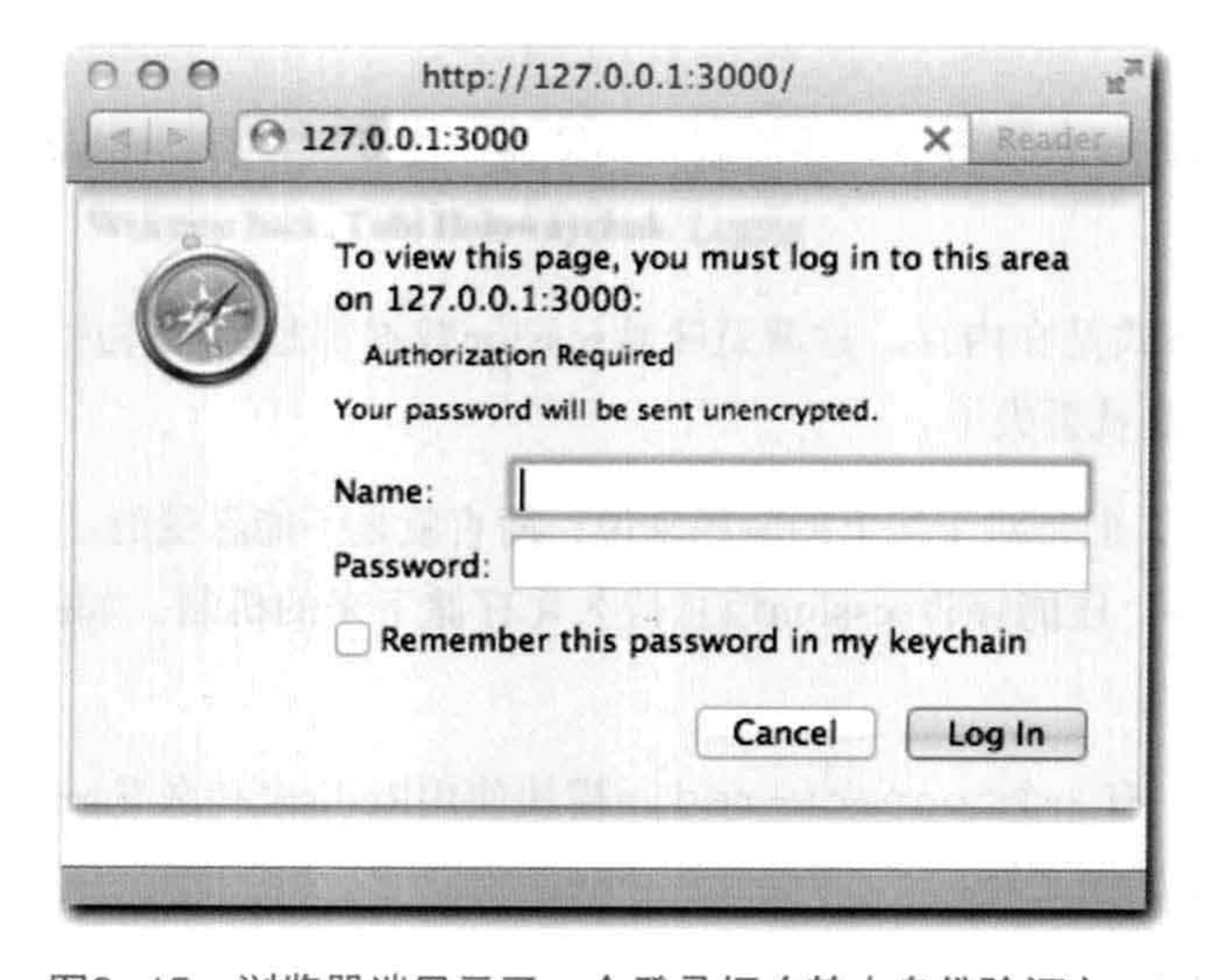


图8-15：浏览器端显示了一个登录框（基本身份验证）

142

为此，Connect也提供了一个名为basicAuth的中间件。

作为例子，我们来创建一个简单的身份验证系统，并且通过命令行对用户进行验证操作。

首先实现用户输入：

```
process.stdin.resume();
process.stdin.setEncoding('ascii');
```

接着，添加basicAuth中间件：

```
connect.basicAuth(function (user, pass, fn) {
  process.stdout.write('Allow user \033[96m' + user + '\033[39m '
    + 'with pass \033[90m' + pass + '\033[39m ? [y/n]: ');
  process.stdin.once('data', function (data) {
    if (data[0] == 'y') {
      fn(null, { username: user });
    }
    else fn(new Error('Unauthorized'));
  });
})
```

注意了，上述代码在stdin EventEmitter上使用了once方法，这是因为我们只需要对每个请求获取一次数据。

basicAuth使用起来非常简单。它提供了user和pass作为参数，同时还提供了一个回调函数来告诉你验证成功还是失败。

如果验证通过，就传递null作为第一个参数（反之，则传递一个Error对象），以及user对象用来生成req.remoteUser对象（供后续的中间件使用）。

然后，继续声明下一个中间件，它会在验证成功后执行。

```
, function (req, res) { res.writeHead(200); res.end('Welcome to the protected area,
  ' + req.remoteUser.username); }
```

好了，运行此例，浏览器会要求输入验证信息（见图8-16）。

图8-16：用示例的验证信息填写登录对话框

最后，通过命令行来对验证进行处理（不安全），如图8-17所示。

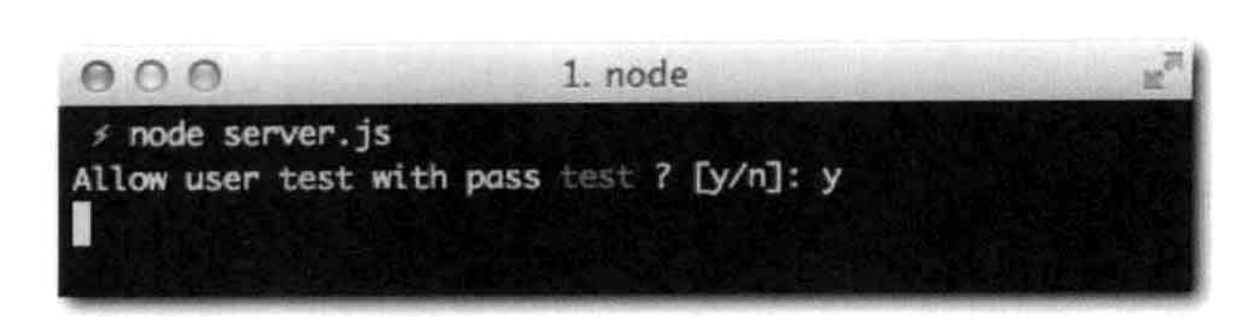


图8-17：服务器将用户输入的验证信息显示在命令行中供管理员处理

图8-18展示了验证通过后的Web站点。

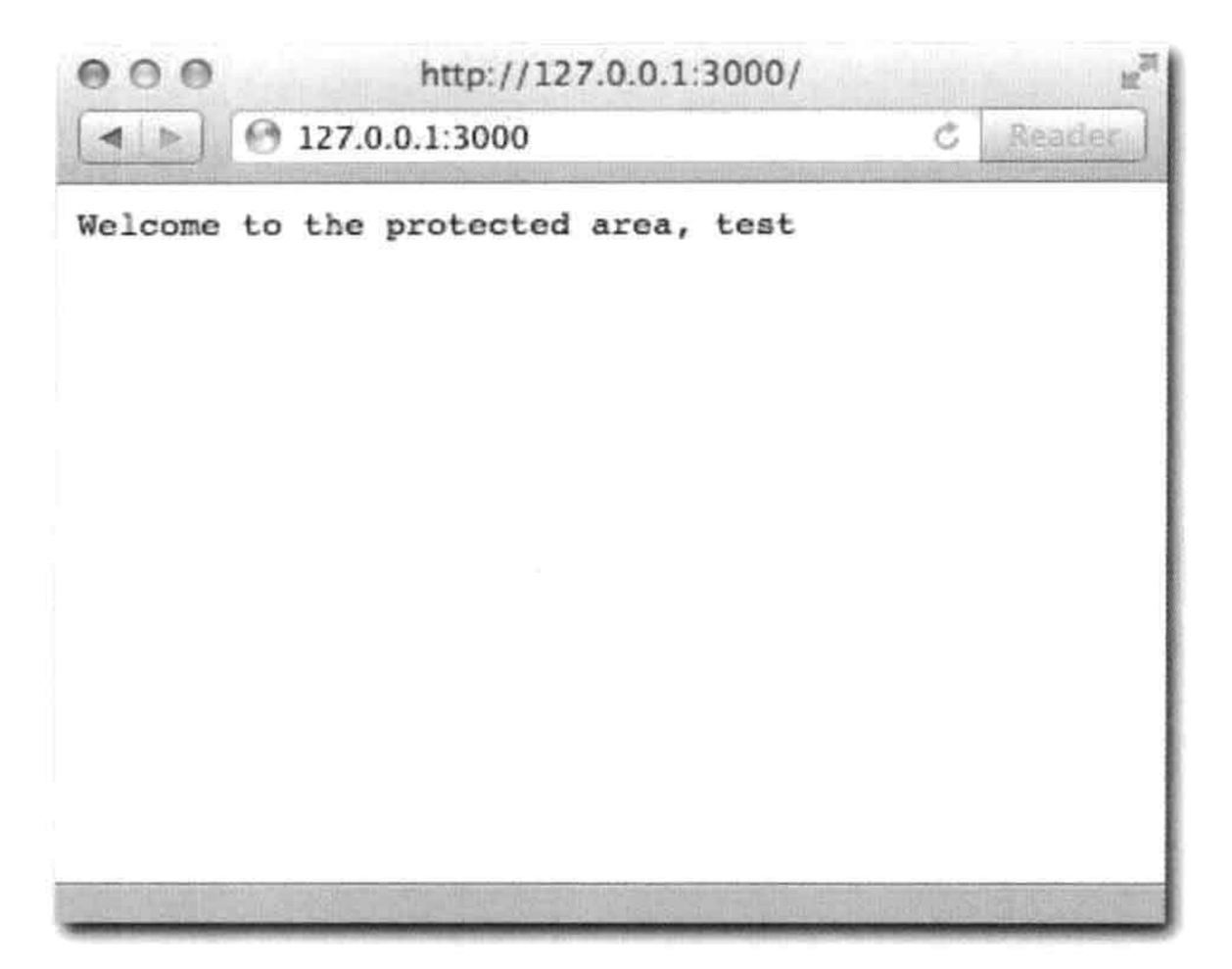


图8-18：在用户验证通过后，请求也验证通过了

144

小结

本章介绍了使用中间件带来的好处：代码能以此为构建单元进行组织，并且能够获得高复用性。Connect是一个实现这种思路的模块，它为构建更具表达力的中间件提供了基础架构。

紧接着，本章又对同一个例子的两种实现——单个请求处理器实现以及拆分成更小构建单元的中间件实现，进行了对比。

最后，又对Connect自带的一些Web应用开发过程中实用的中间件一一进行了介绍。至此，你应当已经学会了如何自定义中间件，以及如何将中间件在Node.js模块系统中进行重用了。

CHAPTER

Express

鉴于Connect基于HTTP模块提供了开发Web应用常用的基础功能，Express基于Connect为构建整个网站以及Web应用提供了更为方便的 API。 145

纵观第8章中的例子，你可能已经发现了，绝大部分Web服务器和浏览器之间交互的任务都是通过URL和method来完成的。这两者的组合有时又称为路由，通过Express创建的应用中的一个基础概念。

Express基于Connect构建而成，因此，它也保持了重用中间件来完成基础任务的想法。这就意味着，通过Express的API方便地构建Web应用的同时，又不失构建于HTTP模块之上高可用中间件的生态系统。

接下来，我们通过Express构建一个查询Twitter API的小应用，来了解Express的用法和功能。

一个小型Express应用

146

应用虽小但五脏俱全。在用户通过查询关键词查询“推文”时，需要返回包含查询结果的HTML页面。另外，我们这次不在请求处理器中采用字符串拼接的方式来返回HTML页面内

容，而是使用一个简单的模板语言来实现，并将逻辑从视图层分离到控制层。

那么第一步就是要确保将所需的模块引入进来。

创建模块

按照惯例，先创建一个package.json文件，这次需要多添加两个额外的依赖：ejs。本例中要使用的模板引擎，以及superagent来简化向 Twitter发送HTTP查询请求的实现。

```
{
    "name": "express-tweet"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.9"
      , "ejs": "0.4.2"
      , "superagent": "0.3.0"
    }
}
```

这里请注意，尽管本例中我们使用的是Express 2，但代码应该是能和 Express 3完全兼容的（截止本书撰写期间Express 3还处在开发阶段）。

定义好项目的元数据之后，接下来创建HTML模板。

HTML

和上一个应用不同的是，为了避免将HTML代码嵌入到应用逻辑（通常叫做处理器或者路由处理器）中，这次我们要使用一个简单的模板语言来处理。模板语言的名字是EJS（内嵌（embedded）的js），和在HTML中内嵌PHP类似。

从在views/文件夹中创建一个index.ejs文件开始。事实上，模板可以放在任何一个地方，不过考虑到本例的项目结构，我们就将其单独放到views目录下。

首个模板对应默认的路由路径（首页）。它提供一个入口让用户提交搜索推文的关键字：

```
<h1>Twitter app</h1>
<p>Please enter your search term:</p>
<form action="/search" method="GET">
  <input type="text" name="q">
  <button>Search</button>
</form>
```

147

另一个模板search.ejs用于展示查询结果。高亮查询关键词并将查询结果逐一显示出来（如果有的话），否则就显示一条消息：

```
<h1>Tweet results for <%= search %></h1>
<% if (results.length) { %>
  <ul>
```

```
    <% for (var i = 0; i < results.length; i++) { %>
    <li><%= results[i].text %> - <em><%= results[i].from_user %></li>
    <% } %>
  </ul>
<% } else { %>
  <p>No results</p>
<% } %>
```

如上面所示，我们将JavaScript代码嵌在<%和%> EJS标签中。另外，我们通过在<%之后加入“=”符号将变量值打印出来。

SETUP

我们现在server.js文件中引入依赖的模块：

```
var express = require('express')
```

引入Express之后，需要用它来初始化Web服务器。和Connect类似，Express提供了一个createServer快捷方法返回一个Express HTTP服务器。就像这样：

```
var app = express.createServer()
```

和其他流行的Web框架不同，Express并不要求任何必要的配置，也不对文件结构有特殊的要求。它足够灵活，允许你对每一个单独的功能点进行自定义。

本例中，我们需要指定使用的模板引擎（这样就不需要每次载入视图时都去引入）以及视图文件（模板）所在的目录。刚刚介绍的express.createServer方法返回的HTTP服务器自带配置系统。我们可以通过调用该对象上的set方法来修改默认的配置项。添加如下代码：

```
app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });
```

第三个view options参数所定义的选项，在渲染视图时，会传递到每个模板中。这里layout的值设置为false，是为了匹配Express 3中的默认值。

要获取配置信息，可以调用app.set方法并传递对应要获取配置的标志。比如，要获取views的配置信息，可以通过如下方式： 148

```
console.log(app.set('views'));
```

接下来，我们会使用Express提供的方法来定义路由，关于路由的有关内容，我们其实在第7章和第8章就已经接触过了。

定义路由

通过使用Express来定义路由就无须手动地每次去检查method和url属性，只需调用

Express提供的对应HTTP method的方法，并将URL和对应的处理中间件传递进去就可以了。

Express支持的方法有get、put、post、del、patch以及head，分别对应HTTP的GET、PUT、POST、DELETE、PATCH以及HEAD。下面是定义路由的例子：

```
app.get('/', function (req, res, next) {});
app.put('/post/:name', function (req, res, next) {});
app.post('/signup', function (req, res, next) {});
app.del('/user/:id', function (req, res, next) {});
app.patch('/user/:id', function (req, res, next) {});
app.head('/user/:id', function (req, res, next) {});
```

第一个参数是路由地址，第二个参数是路由处理程序。路由处理程序就和中间件一样。

注意了，路由部分还可以通过特殊格式来定义变量。如上述例子中的/user/:id，哪怕id值不同，路由也能匹配到：如/user/2、/user/3，等等。下一章会对这部分内容做详细介绍。

下面我们来定义首页的路由：添加如下代码到server.js文件中：

```
app.get('/', function (req, res) {
  res.render('index');
});
```

到目前为止，完整代码如下所示：

```
/**
 * 模块依赖
 */
var express = require('express')
  , search = require('./search')

/**
 * 创建app
 */

var app = express.createServer();

/**
 * 配置
 */

app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });

/**
 * 路由
 */

app.get('/', function (req, res) {
```

149

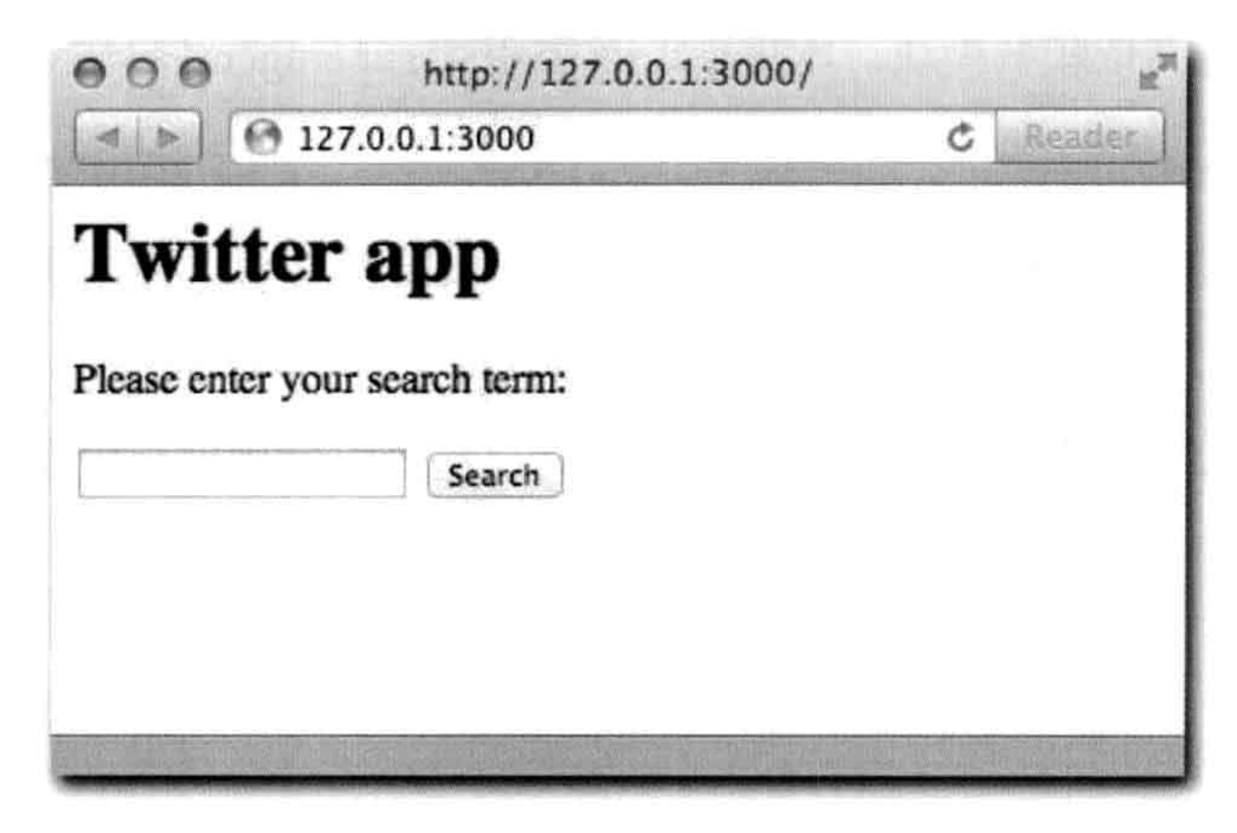
```
  res.render('index');
});

/**
 * 监听
 */
app.listen(3000);
```

Express为response对象提供了render方法。该方法完成如下三件事：

1. 初始化模板引擎。
2. 读取视图文件并将其传递给模板引擎。
3. 获取解析后的HTML页面并作为响应发送给客户端。

由于在前一步中将ejs指定为视图引擎，因此无须显式地指明index.ejs了。

运行之后结果如图9-1所示（别忘记调用listen方法）。



150

图9-1：路由处理器/渲染index视图

第二个路由中，我们调用一个名为search的方法（将在另外一个模块中定义）：

```
app.get('/search', function (req, res, next) {
  search(req.query.q, function (err, tweets) {
    if (err) return next(err);
    res.render('search', { results: tweets, search: req.query.q });
  });
});
```

接着，在express之后添加对search模块的依赖：

```
var express = require('express')
  , search = require('./search')
```

这里要注意，如果search方法回调函数中接收到错误对象，那么我们就直接把它传递给

next。后面章节会介绍错误处理的相关内容，到时你就能明白为什么要这么做了，目前，我们就假设Express会将错误信息通知给用户。

这个路由处理器中，我们也调用了render方法，不过不同的是，这次我们传递了一个对象作为第二个参数。该对象的内容会暴露给视图。注意这里我们是如何把tweets和search传递过去的，这两个变量可以在search.ejs视图中直接使用。这类对象称为本地变量，因为它的内容只对其传递的视图可见。

查询

查询模块暴露一个简单的方法提供对推文的查询，其内部调用了Twitter的查询API。本例中，我们将查询模块search.js文件和server.js文件放到同一目录中。

上述调用search方法的代码中，传递了一个查询关键字以及回调函数，回调函数接收两个参数，其一是错误对象（如果有的话），其二是一个包含了查询到的推文的数组。

151

要写这样一个模块，我们先来定义依赖的模块。本例中，只需用到 superagent：

```
var request = require('superagent')
```

由于向Twitter的Web服务发送HTTP请求是本例中重要的功能，我们需要确保查询模块正确地进行了错误处理。

比如，要是Twitter API服务宕机了，我们就发送一个错误对象，来给用户显示一个错误页面（如：显示HTTP错误状态码500）。

```
/**
 * Search function.
 *
 * @param {String} search query
 * @param {Function} callback
 * @api public
 */

module.exports = function search (query, fn) {
  request.get('http://search.twitter.com/search.json')
    .data({ q: query })
    .end(function (res) {
      if (res.body && Array.isArray(res.body.results)) {
        return fn(null, res.body.results);
      }
      fn(new Error('Bad twitter response');
    });
};
```

和其他superagnet例子类似，我们发送一个GET请求，将查询关键字作为数据字段q的值

以查询字符串的形式发送出去。以hello world为查询关键字的URL类似`http://search.twitter.com/search.json? q=hello+world`。

在响应处理程序中，我们确保请求发送成功并且完全符合我们的预期。相比查看HTTP响应码是否为200，我们采用更加聪明的方式：直接查看响应结果是否包含含有推文内容的数组。

第7章中我们就介绍过，如果superagent获取到一个JSON形式的响应消息，它会自动进行解码，并将解码后的内容放到res.body变量中。由于Twitter API会返回一个JSON对象，对象中的results字段内容就是包含了推文的数组，因此，下述代码段对于进行错误处理的判断来说足矣：

```
if (res.body && Array.isArray(res.body.results)) {
  return fn(null, res.body.results);
}
```

运行

152

运行上述服务器代码，并通过浏览器访问`http://localhost:3000`（见图9-2），试着输入推文关键字（见图9-3）。

图9-2：输入查询推文关键字并提交的例子

图9-3：查询结果

成功！向Twitter查询推文后，我们获得了一个包含推文的数组。最终它会显示在根据推文数组动态生成的search.ejs模板中。

HTML页面由Express的渲染引擎产生，/search路由成功地将完整的页面显示给用户，如图9-3所示。

在这个简单的示例后，接下来该是深入研究Express特性的时候了，我们来看一些新的功能。

153 设置

Express提供了最有意思的特性之一，同时也是对任意类型Web应用都不可或缺的，就是对环境设置的管理能力。

比如，在生产环境下，为了提高性能，我们可以让express将模板缓存起来，这样可以加快响应速度。然而，若在开发模式下开启这个功能，每次对模板稍做改动就需要重启Node服务程序才能生效。

通过如下方式就可以达到让Express对模板进行缓存的效果：

```
app.configure('production', function () {
  app.enable('view cache');
});
```

在上述代码中，app.enable就等同于此前例子中为了配置views config标志所用到的app.set。

```
app.set('view cache', true);
```

要查看配置标志是否启用，也可以调用app.enabled。对应的还有app.disable和app.disabled方法。

当环境变量NODE_ENV设置为production时，我们在app.configure定义的对应的回调函数就会被执行。

要测试上述代码，运行如下命令：

```
$ NODE_ENV=production node server
```

要是NODE_ENV没有设置，则默认会调用development的配置：

```
app.configure('development', function () {
  // gets called in the absence of NODE_ENV too!
});
```

下面还有一些内置的实用设置：

- 大小写敏感的路由：启用大小写敏感的路由。默认情况下，当定义如下路由：

```
app.get('/my/route', function (req, res) {}
```

  Express能匹配到/my/route和/MY/ROUTE。开启之后，路由只会匹配定义的路由，上述例子中就只会匹配/my/route。

- 严格路由：启用后，最后的斜杠就不会自动忽略。什么意思呢？比如，此前例子中能匹配/my/route和/my/route/，而一旦开启严格路由模式，只会匹配到/my/route，因为路由就是这样在app.set中定义的。 154
- jsonp回调：启用res.send() / res.json()对jsonp的支持。JSONP是一项解决跨域JSON请求的技术，它将响应结果包裹在一个用户指定的回调函数中。
- 当有JSONP请求时，其URL类似这样：/my/route?callback=myCallback。Express会自动检测callback参数，并将相应结果包裹在myCallback文本中。要启用该功能，可以调用app.enable('jsonp callback')。这里要注意，这只作用于res.send和res.json，后面的章节会有相关介绍。

模板引擎

要像此前例子那样使用ejs，必须完成以下两个步骤：

1. 通过NPM安装ejs模块。
2. 声明view engine为ejs。

如下的其他模板引擎也可以在Express中使用：

- Haml
- Jade
- CoffeeKup
- jQuery Templates for node

Express会试着以模板文件扩展名或者以配置的view engine的值为名去调用require方法。比如，可以调用：

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

这时，Express会尝试着去require html引擎。找不到该引擎，就会报错。

也可以通过app.register API将扩展名匹配到指定的模板引擎。比如，通过如下方式

就可以将html扩展名匹配到jade模板引擎：[1]

```
app.register('.html', require('jade'));
```

Jade是最流行的模板语言之一，绝对值得一学。要想了解更多关于Jade的内容可以访问其官方网站http://jade-lang.org。

155 错误处理

在Node中，将错误对象作为非阻塞I/O回调函数的第一个参数是很常见的。在本章的例子中，我们也有可能在调用Twitter API进行查询时接收到错误对象。

面对这种情况，在Express中常规的做法就是将该错误对象传递给next函数。默认情况下，Express会展示一个错误页面并发送500状态码。

绝大多数Web应用都会自定义错误页面，或者甚至自建一套后台的错误处理机制。

我们可以通过app.error方法定义一个特殊的错误处理器作为错误处理的中间件：

```
app.error(function (err, req, res, next) {
  if ('Bad twitter response' == err.message) {
    res.render('twitter-error');
  } else {
    next();
  }
});
```

注意在上述代码中，通过检测错误消息内容来判断是否要对错误进行处理，要是检测结果不匹配就直接调用next。

还可以设置多个.error处理器来执行不同的处理。比如，最后一个错误处理器就可以发送500 Internal Server Error并展示一个通用的错误页面。

```
app.error(function (err, req, res) {
  res.render('error', { status: 500 });
});
```

当调用next并且对应的处理器无法找到时，默认的Express错误处理器就会触发。

快捷方法

Express为Node中的Request和Response对象提供了一系列扩展来简化开发。

Request对象上的扩展如下。

1 译者注：Express 3.x中需要使用app.engine方法。

- header：此扩展能够让程序以函数的方式获取头信息，并且还是大小写不敏感的。

```
req.header('Host')
req.header('host')
```

- accepts：此扩展会分析请求中的Accept头信息，并根据提供的值返回true或者false。 156

```
req.accepts('html')
req.accepts('text/html')
```

- is：此扩展和accepts类似，但它检查Content-Type头信息。

```
req.is('json')
req.is('text/html')
```

Response对象上的扩展如下。

- header：此扩展接收一个参数来检查对应的头信息是否已经在 response上设置了。

```
res.header('content-type')
```

或设置header的两个参数：

```
res.header('content-type', 'application/json')
```

- render：对render相信你已经了解不少了。不过，在此前的例子中，你也许注意到了我们传递了status值。这是一个特殊的类型，设置了它就等于为response响应对象设置了状态码。
  - 除此之外，还可以提供第三个参数给render方法来获取 HTML内容而不是直接将其作为响应消息自动传递出去。

```
res.render('template', function (err, html) {
  // 处理收到的 html
});
```

- send：此扩展很奇特。它会根据提供参数的类型执行响应的行为。
  - Number：发送状态码。

```
res.send(500);
```

  - String：发送HTML内容。

```
res.send('<p>html</p>');
```

  - Object/Array：序列化成JSON对象，并设置相应的Content-Type头信息。

```
res.send({ hello: 'world' }); res.send([1,2,3]);
```

- json：此扩展在绝大多数场景下和send类似。只是它会显式地将内容作为JSON对象发送。

```
res.json(5);
```

在发送值类型未知的情况下可以使用此方法。res.send会判断发送值的类型，并且依据判断结果来选择是否调用JSON.stringify方法。如果是数字类型，那么会认为发送的是状态码，并直接结束响应。而res.json会把数字类型也进行JSON.stringify转换。

由于绝大部分情况下我们会对发送对象进行编码，所以，res.send还是很不错的选择。

157

- redirect：redirect等效于发送302（暂时移除）状态码以及 Location头信息。如下所示。

```
res.redirect('/some/other/url')
```

就等效于：

```
res.header('Location', '/some/other/url');
res.send(302);
```

上述代码在Node.js内部其实是这样处理的：

```
res.writeHead(302, { 'Location': '/some/other/url' });
```

- redirect还可以接收自定义的状态码作为第二个参数。假设你不想发送302而是发送表示永久性移除的301状态码，可以采取如下方式。

```
res.redirect('/some/other/url', 301)
```

- sendfile：此扩展和Connect中的static中间件类似，不同之处在于它用于单个文件。

```
res.sendfile('image.jpg')
```

路由曾在我们的示例应用中有所应用，事实上，路由还有很多内容有待我们了解，它们对大型Web应用都非常有帮助。

路由

在定义路由时，可以使用自定义参数：

```
app.get('/post/:name', function () {
  // . . .
})
```

上述代码中，name变量值会注入到req.params对象上。比如，当通过浏览器访问'/post/hello-world'时，req.params对象会变为如下形式：

```
app.get('/post/:name', function () {
  // req.params.name == "hello-world"
})
```

还可以通过在变量后添加问号（？）来表示该变量是可选的。在此前的路由示例中，要是通过浏览器访问/post，那么该路由是不会匹配到的。要匹配到可以采用如下方式：

```
app.get('/post/:name?', function (req, res, next) {
  // this will match for /post and /post/a-post-here
})
```

像这样定义了参数的路由，内部会当正则表达式处理。也就是说，定义路由时也可以直接使用RegExp对象。比如，只想匹配字母、数字以及中划线的话，可以这样： 158

```
app.get(/^\/post\/([a-z\d\-]*)/, function (req, res, next) {
  // req.params contains the matches set by the RegExp capture groups
})
```

和中间件一样，在路由处理程序中也可以使用next。即使当一个路由匹配并得到处理，还是可以强制Express去继续匹配其他路由的。

比如，让路由只接受以'h'开头的参数：

```
app.get('/post/:name', function (req, res, next) {
  if ('h' != req.params.name[0]) return next();
  // . . .
});
```

就是因为Express提供了灵活的路由，才能实现像上述这样的处理代码来解决多变的情况。

举例来说，许多Web应用允许如/home、/about这样的路由，但它们同时又希望能够让动态的内容也能有永久的URL。

在定义好所有的路由之后，可以再定义一个路由来获取用户名，并进行数据库调用。如果该用户未能找到，就调用next并发送404，否则就渲染该用户个人页面。

```
app.get('/home', function (req, res) {
  // . . .
});

app.get('/:username', function (req, res, next) {
  // if you got here, no prior application routes matched
  getUser(req.params.username, function (err, user) {
    if (err) return next(err);

    if (exists) {
      res.render('profile')
    } else {
```

```
      next();
    }
  });
});
```

如你所知，Express也沿袭了中间件的概念。下面我们就来深入了解一下。

159 中间件

由于Express是构建于Connect之上的，所以，当创建Express服务器时可以使用Connect兼容的中间件。比如，要托管images/目录下的图片，就可以像这样使用static中间件：

```
app.use(express.static(__dirname + '/images'));
```

或者，要想使用connect的session，也很容易，像这样就可以了：

```
app.use(express.cookieParser());
app.use(express.session());
```

注意了，在引入了Express之后就可以直接使用Connect的中间件了。不需要require('connect')或者把connect作为项目依赖添加到package.json文件中。

中间件是易理解的。

更有意思的是，和全局中间件（针对每个请求）不同，Express还允许只在特定匹配到的路由中才使用中间件。

想象一下这样一个场景：你需要检查用户是否已经登录，并且这部分检查只在特定受保护的路由中进行。这个时候，就可以定义一个secure中间件，判断若req.session.logged_in不为true时就发送403 Not Authorized状态码。

```
function secure (req, res, next) {
  if (!req.session.logged_in) {
    return res.send(403);
  }

  next();
}
```

然后，将它应用到对应的路由中：

```
app.get('/home', function () {
  // accessible to everyone
});

app.get('/financials', secure, function () {
  // secure!
});

app.get('/about', function () {
  // accessible to everyone
```

```
});

app.get('/roadmap', secure, function () {
  // secure!
});
```

还可以像这样为路由定义多个中间件：

160

```
''app.post('/route', a, b, c, function () { });
```

有的时候，在中间件中调用next就可以跳过该路由的其他中间件，这样Express就会紧接着在下一个路由中做相应处理。

比如，若相比发送403，你更希望Express去检查其他的路由，那么就可以采用如下方式来实现：

```
function secure (req, res, next) {
  if (!req.session.logged_in) {
    return next('route');
  }

  next();
}
```

通过调用next('route')，就能确保当前路由会被跳过。

随着项目规模的扩大，路由和中间件的数量也会越来越多，因此，如何更好地组织代码就变得非常有用。下面我们就来介绍这部分的内容。

代码组织策略

对于任意一个Node.js应用（包括Express Web应用）来说，第一条准则都是模块化。Node.js通过提供一个简单的require API来提供一个强大的代码组织策略。

比如，一个应用包含三块独立的内容：/blog、/pages以及/tags。每块都包含各自的路由。例如：/blog/search、/pages/list以及tagst/cloud。

好的代码组织方式应当是维护一个server.js文件，该文件中包含了路由表，同时将每一部分的路由处理器都通过模块化的方式来引入，如blog.js、pages.js以及tags.js。首先，定义依赖的模块并初始化 app、引入中间件等：

```
var express = require('express')
  , blog = require('./blog')
  , pages = require('./pages')
  , tags = require('./tags')

// initialize app
```

```
var app = express.createServer();

// here you would include middleware, settings, etc
```

161 接着定义之前提到的路由表，这里简单地将所有的路由信息都罗列出来放在一个地方：

```
// blog routes
app.get('/blog', blog.home);
app.get('/blog/search', blog.search);
app.post('/blog/create', blog.create);

// pages routes
app.get('/pages', pages.home);
app.get('/pages/list', pages.list);

// tags routes

app.get('/tags', tags.home);
app.get('/tags/search', tags.search);
```

然后，针对每个模块需要使用exports函数。以blog.js为例：

```
exports.home = function (req, res, next) {
  // home
};

exports.search = function (req, res, next) {
  // search functionality
};
```

模块化提供了很强大的扩展性。上述代码还可以根据http方法进行进一步扩展。例如：

```
exports.get = {};
exports.get.home = function (req, res, next) {})
exports.post = {};
exports.post.create = function (req, res, next) {})
```

另外一种对应用进行解耦的方式叫做app挂载。你可以将整个Express app作为一个模块（模块中依然可以使用NPM模块），并将其挂载到现有的应用中，这样就能够让路由无缝对接起来。

考虑这样一个博客的应用。你可以将一个博客相关的路由都通过/来定义，/categories以及/search，然后将其以blog.js文件暴露出来：

```
var app = module.exports = express.createServer();
app.get('/', function (req, res, next) {});
app.get('/categories', function (req, res, next) {});
app.get('/search', function (req, res, next) {});
```

注意，上述路由都是通过绝对路径来定义的，不包含任何前缀。接着，在主程序中，要做的就是将其引入并传递给app.use方法：

```
app.use(require('./blog'));
```

162

这样一来，所有博客相关的路由就都可以使用了。除此之外，你还可以为它们定义一个前缀：

```
app.use ('/blog', require('./blog'));
```

现在，/blog/、/blog/categories以及/blog/search以后都可以直接给其他express应用使用，并且它自身拥有完全独立的依赖、中间件、配置，等等。

小结

本章介绍了如何使用最流行的Node.js Web框架Express。

使用Express最大的益处就在于，它简洁、配置少但却又不失灵活，同时它和Connect一样构建于测试全面、抽象简洁的基础之上。

和其他Web框架或者类库不同，Express非常容易进行模块化来满足不同的需求、结构以及模式。本章介绍了如何使用Express以最少的代码来构建首个示例应用，就像Node.js Hello World程序那样。

事实上，你或许已经注意到了，与重新在Node.js core API基础上构建一套新的方式不同，Express尝试与core API靠拢，并扩展它。这就是路由处理器可以直接接收原生的Node request和response对象的原因，就像我们在首个HTTP服务器应用中那样。本章介绍了一些Express实用的扩展，比如，如何使用res.send来以JSON的形式进行响应等。

本章最后介绍了如何将不同功能的代码组织起来创建一个可维护的应用。介绍了最好的方式就是通过Node.js核心的API：通过使用require就是其中一个最强大的工具，来进行更上层的代码组织。