CHAPTER

10 WebSocket

目前，绝大部分网站和Web应用开发者都习惯了通过发送HTTP请求来和服务器进行通信，随后接收服务器的HTTP响应。 163

正如在上一章中看到的那样，通过指定URL、Content-Type以及设置其他属性来获取资源的模型是最常见的，也是万维网中最常见的方式。Web生来就是用以传递互相关联的文档的。URL之所以含有路径信息是因为文档通常在文件系统中是有层次结构的，并且，每一层的结构都包含了对超链接的索引。

如下述例子：

```
GET /animals/index.html
GET /animals/mammals/index.html
GET /animals/mammals/ferrets.html
```

然而，随着时间的推移，Web变得越来越注重用户体验。如今，特别是随着HTML5以及相关工具的诞生，传统的那种每次需要用户点击之后才能获取文档的方式正在逐步退出历史舞台。现在已经可以创建出如游戏、文本编辑器等这种非常酷的Web应用了，完全可以取代了传统的桌面应用。

Ajax

164

Web 2.0标志着Web应用的崛起。其中一个关键因素就是Ajax，其具体表现在于提高了用

户体验，这背后重要的原因就是：用户再也不用每次都通过交互操作才能从服务器端获取新的HTML文档。

比如，要想在社交应用中更新个人信息，发送一个异步的POST请求，随后会收到一个更新成功的返回消息。接着，使用一个简单易用的 JavaScript框架，更新视图以展现用户行为结果即可。

再比如，当用户点击一张表中的移除按钮时，就可以发送DELETE请求，并移除该行（<tr>）元素即可，无须刷新浏览器，也无须再去获取许多不必要的数据、图片、脚本以及样式文件以及重新渲染整个页面。

Ajax之所以重要，从本质上来说，它避免了许多原本需要在Web应用中处理的数据传递和渲染的开销。

然而，现在许多应用通过传统的HTTP请求+响应模型的方式来发送和接收数据依然会造成很大的开销。就拿本章中要构建的应用来说，我们要想实时地获取每位网站访问者当前鼠标的位置。那么每次当用户移动鼠标时，我们都要将坐标信息发送给服务器。

假设使用jQuery来发送Ajax请求。第一个想到的办法就是：每次mousemove事件触发时，就通过$.post向服务器发送一个POST请求，如下所示：

```
$(document).mousemove(function (ev) {
  $.post('/position', { x: ev.clientX, y: ev.clientY });
});
```

上述代码尽管看起来很直观，但是有个根本性的问题：无法控制服务器接收请求的先后顺序。

当执行代码发出请求时，浏览器会使用可用的socket来进行数据发送，为了提高性能，浏览器会和服务器之间建立多个socket通道。举例来说，在下载图片的时候，还是可以同时发送Ajax请求。要是浏览器只有一个socket通道，那么，网站渲染和使用都会非常慢。

若三个请求分别通过三个不同的socket通道发送，就无法保证服务器端接收的顺序了。所以，要解决这个问题，我们需要调整代码，在服务器接收到一个请求后再接着发送第二个请求，这样就能保证接收的顺序了：

165

```
var sending = false;

$(document).mousemove(function (ev) {
  if (sending) return;
  sending = true;
  $.post('/position', { x: ev.clientX, y: ev.clientY }, function () {
    sending = false;
  });
});
```

作为例子，下面显示了Firefox中TCP传输消息的内容：

Request

```
POST / HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:8.0.1) Gecko/20100101
 Firefox/8.0.1
Accept: */*
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: http://localhost:3000/
Content-Length: 7
Pragma: no-cache
Cache-Control: no-cache

x=6&y=7
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 2
Connection: keep-alive

OK
```

如上述例子所示，除了一小段数据，还包含了很多的文本内容。对于上述例子而言，很多不需要的头信息依然被发送了出去，这些头信息的数据量远远超过了真正要发送数据本身的大小。

尽管可以移除其中一些头信息，但是对于上述例子来说，我们真的需要响应消息吗？在发送一些像鼠标位置这样无关紧要的信息时，其实根本不需要等到响应返回后再接着发送更多的消息。

对于这类Web应用来说，理想解决方案得从TCP而非HTTP入手（就像第6章中的聊天程序那样）。理想状态下，我们更希望直接将数据，另外附加最小的消息窗口（就是与真正发送的数据包裹在一起的数据）通过socket发送。 166

拿telnet举例的话，我们就希望浏览器可以发送如下面这样的数据：

```
x=6&y=7 \n
x=10&y=15 \n
. . .
```

归功于HTML5，现在我们有了这样的解决方案：WebSocket。WebSocket是Web下的TCP，

一个底层的双向socket，允许用户对消息传递进行控制。

HTML5 WebSocket

每次提到WebSocket的时候，其实是在讲两部分内容：一部分是浏览器实现的WebSocket API，另一部分是服务器端实现的WebSocket协议。这两部分是随着HTML5的推动一起被设计和开发的，但是两者都没有成为HTML5标准的一部分。前者被W3C标准化了，而后者被IETF标准化为RFC 6455。

浏览器端实现的API如下：

```
var ws = new WebSocket('ws://host/path');
ws.onopen = function () {
  ws.send('data');
}
ws.onclose = function () {}
ws.ondata = function (ev) {
  alert(ev.data);
}
```

上述简单的API不禁让我们想起第6章中写过的TCP客户端。和XMLHttpRequest（Ajax）不同，它并非面向请求和响应，而是可以直接通过send方法进行消息传递。通过data事件，发送和接收 UTF-8或者二进制编码的消息都非常简单，另外，通过open和close事件能够获知连接打开和关闭的状态。

首先，连接必须通过握手来建立。握手方面和普通的HTTP请求类似，但在服务器端响应后，客户端和服务器端收发数据时，数据本身之外的信息非常少：

167

Request

```
GET /ws HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 6
Sec-WebSocket-Origin: http://pmx
Sec-WebSocket-Extensions: deflate-stream
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
```

Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
```

WebSocket还是建立在HTTP之上的，也就说，对于现有的服务器来说，实现WebSocket协议非常容易。它和HTTP之间主要的区别就是，握手完成后，客户端和服务器端就建立了类似TCP socket这样的通道。

为了更好地理解这部分内容，我们来写个示例应用。

一个ECHO示例

第一个示例包含一个服务器和一个客户端，主要完成互相之间消息的交换并输出接收的消息内容。当客户端发送消息时，记录相应的时间，用于计算服务器处理响应花费了多长时间。

初始化项目

本例中，我们将使用我在LearnBoost工作时开发的`websocket.io`模块。

这里有一点非常重要：`websocket.io`仅处理WebSocket相关的请求。其他请求仍由你的网站或者应用的常规Web服务器来处理，这就是我们在`package.json`文件中加入对`express`的依赖[1]的原因：

---

1 译者注：截止译者翻译期间，本章示例在最新稳定版node.js v0.10.x下运行会有错误抛出。具体原因是websocket.io中代码书写有问题，要解决需要将websocket.io/lib/hybi-16.js文件中如下左侧部分的代码中的`.on('xxx',fn)`，均改为`.onxxx = fn`的形式，如下右侧所示：

```
this.parser
.on('text', function (packet) {

 self.onMessage(packet);
})
.on('binary', function (packet) {

 self.onMessage(packet);
})
.on('ping', function () {

 // version 8 ping => pong
 try {

 self.socket.write('\u008a\u0000');
 }
 catch (e) {
 self.end();
 return;

 }
})
.on('close', function () {

 self.end();
})
.on('error', function (reason) {

 self.log.warn(self.name + 'parser error:'
+ reason);
 self.end();
})
```

```
this.parser.ontext = function (packet) {
 self.onMessage(packet);
};

this.parser.onbinary = function (packet)
{
 self.onMessage(packet);
};

this.parser.onping = function () {
 // version 8 ping => pong
 try {

 self.socket.write('\u008a\u0000');
 }
 catch (e) {

 self.end();
 return;
 }
};

this.parser.onclose = function () {
 self.end();
};

this.parser.onerror = function (reason){
self.log.warn(self.name + 'parser error:'
+ reason);
self.end();
};
```

```
"name": "ws-echo"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.1"
      , "websocket.io": "0.1.6"
  }
}
```

168 服务器的处理就是简单地将浏览器端发送过来的消息再回传回去。浏览器端则计算服务器端处理响应的耗时。

建立服务器

首先要做的就是初始化express并将weksocket.io绑定到express上，这样它就能处理WebSocket的请求了：

```
var express = require('express')
  , wsio = require('websocket.io')

/**
 * Create express app.
 */

var app = express.createServer();

/**
 * Attach websocket server.
 */

var ws = wsio.attach(app);

/**
 * Serve your code
 */

app.use(express.static('public'));

/**
 * Listening on connections
 */

ws.on('connection', function (socket) {
  // . . .
});

/**
 * Listen
 */

app.listen(3000);
```

我们来重点看下connection处理程序。我特意将websocket.io设计为与实现一个net.Server类似。由于我们需要直接将接收到的消息返回给客户端，所以，我们需要做的就是监听message事件，并将其send回去。

```
ws.on('connection', function (socket) {
  socket.on('message', function (msg) {
    console.log(' \033[96mgot:\033[39m ' + msg);
    socket.send('pong');
  });
});
```

169

建立客户端

接着我们在public目录下添加index.html文件，输入如下内容：

index.html

```
<!doctype html>
<html>
  <head>
    <title>WebSocket echo test</title>
    <script>
      var lastMessage;

      window.onload = function () {
        // create socket
        var ws = new WebSocket('ws://localhost:3000');
        ws.onopen = function () {
          // send first ping
          ping();
        }
        ws.onmessage = function (ev) {
          console.log(' got: ' + ev.data);
          // you got echo back, measure latency
          document.getElementById('latency').innerHTML = new Date - lastMessage;
          // ping again
          ping();
        }
        function ping () {
          // record the timestamp
          lastMessage = +new Date;
          // send the message
          ws.send('ping');
              };
    </script>
  </head>
  <body>
    <h1>WebSocket Echo</h1>
    <h2>Latency: <span id="latency"></span>ms</h2>
```

```
</body>
</html>
```

上述HTML代码非常直观。它创建了一个占位符用于显示服务器端处理的延时（消息来回一次所需要的时间，以毫秒为单位）。

170

JavaScript代码相对来说也较直观。首先定义一个存储延时的变量：

```
var lastMessage
```

接着初始化WebSocket并和服务器端建立连接：

```
var ws = new WebSocket('ws://localhost:3000');
```

建立连接后，就向服务器端发送第一条消息：

```
ws.onopen = function () {
  ping();
}
```

收到服务器端响应后计算耗时，并再次发出一条消息：

```
ws.onmessage = function () {
  console.log(' got: ' + ev.data);
  // you got echo back, measure latency
  document.getElementById('latency').innerHTML = new Date - lastMessage;
  // ping again
  ping();
}
```

最后，就是定义ping函数，记录发送消息前的时间戳用于最终计算耗时（也就是此前说的延时），随后发送一条简单的消息给服务器：

```
function ping () {
  // record the timestamp
  lastMessage = +new Date;
  // send the message
  ws.send('ping');
};
```

运行示例程序

输入如下命令来运行服务器端代码：

```
$ node server.js
```

接着，打开浏览器访问http://localhost:3000（见图10-1）。请确保使用的浏览器支持WebSocket，像Chrome 15+或者IE 10+都支持。如果不确定的话，可以访问http://websocket.org，查看网站右上角的“Does your browser support WebSocket”显示框。

好了，至此我们就成功创建了一个单用户的实时应用程序。通过终端的输出和浏览器的控制台能够看到消息传输的日志。在绝大多数现代电脑中，消息传输大概耗时1~5毫秒。作为习题，大家可以尝试使用Ajax配合Express的路由功能书写一个同样功能的应用程序，然后对比一下完成同样一组消息的传递耗时多久。 171

图10-1：一组消息从客户端到服务器端，再返回到客户端所需的时间

接下来，我们要书写一个示例应用，该应用用于将多个用户连接显示到一个屏幕上。

鼠标光标

接下来我们要书写的程序是在一个屏幕上以鼠标光标样式的图片形式展示所有连接过来的用户光标的位置。

通过本例，你将学习到广播的概念，也就是将一个消息发给除了自己以外的所有人。

初始化示例程序

书写本示例程序所需要的模块和上例一样，我们在package.json定义这些依赖：

```
{

    "name": "ws-cursors"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.1"
      , "websocket.io": "0.1.6"
    }

}
```

建立服务器

172

建立服务器的方式和上例类似，使用epxress托管静态HTML文件，同时将websocket.io

绑定到express服务器上：

```
var express = require('express')
  , wsio = require('websocket.io')

/**
 * Create express app.
 */

var app = express.createServer();

/**
 * Attach websocket server.
 */

var ws = wsio.attach(app);

/**
 * Serve your code
 */

app.use(express.static('public'))

/**
 * Listening on connections
 */

ws.on('connection', function (socket) {
  // . . .
});

/**
 * Listen
 */

app.listen(3000);
```

但是，本例中，在用户连接成功后的处理就不同了。我们通过使用一个简单的变量将所有连接过来的用户位置信息记录在内存中。除此之外，我们还记录下连接用户总数，来方便分配给每个用户一个唯一的ID号。这个ID号用来标识positions对象中用户光标位置信息。

```
var positions = {}
  , total = 0

ws.on('connection', function (socket) {
  // . . .
});
```

173 当用户连接过来后，我们首先将其他人的位置信息作为第一条消息内容发送给他。这样一

来，用户就能看到所有连接进来的用户。

最后发送前，我们将positions对象编码为JSON格式：

```
ws.on('connection', function (socket) {
  // you give the socket an id
  socket.id = ++total;

  // you send the positions of everyone else
  socket.send(JSON.stringify(positions));
});
```

当用户发送消息时，我们就假定发送的是JSON格式的位置信息（一个包含x、y坐标的对象），然后将其存储到positions对象中。

```
socket.on('message', function (msg) {
  try {
    var pos = JSON.parse(msg);
  } catch (e) {
    return;
  }

  positions[socket.id] = pos;
});
```

最后，当用户断开连接后，我们就把他的位置信息清除：

```
socket.on('close', function () {
  delete positions[socket.id];
});
```

是不是少了什么？没错！广播！当收到位置信息时，我们需要将其广播给所有其他的人。当socket关闭时，我们得要通知所有其他人并将该光标从屏幕上移除。

我们声明一个broadcast函数，遍历所有其他用户并逐个发送消息给他们。将该函数放在注册connection处理程序的后面：

```
function broadcast (msg) {
  for (var i = 0, l = ws.clients.length; i < l; i++) {
    // you avoid sending a message to the same socket that broadcasts
    if (ws.clients[i] && socket.id != ws.clients[i].id) {
      // you call 'send' on the other clients
      ws.clients[i].send(msg);
    }
  }
}
```

由于我们要发送两类数据，所以，我们在JSON数据包中包含type标示符。 174

发送位置信息时，数据结构如下所示：

```
{
    type: 'position'
  , pos: { x: <x>, y: <y> }
  , id: <socket id>
}
```

当用户断开连接时，发送：

```
{
    type: 'disconnect'
  , id: <socket id>
}
```

因此：

```
socket.on('message', function () {
  // . . .
  broadcast(JSON.stringify({ type: 'position', pos: pos, id: socket.id }));
});
```

并且在关闭时，发送：

```
socket.on('close', function () {
  // . . .
  broadcast(JSON.stringify({ type: 'disconnect', id: socket.id }));
});
```

至此我们完成了服务器部分代码，接着我们开始书写客户端部分。

建立客户端

客户端部分，我们还是从一个简单的HTML开始，监听onload事件：

```
<!doctype html>
<html>
  <head>
    <title>WebSocket cursors</title>
    <script>
      window.onload = function () {
        var ws = new WebSocket('ws://localhost');
        // . . .
      }
    </script>
  </head>
  <body>
    <h1>WebSocket cursors</h1>
  </body>
</html>
```


这次，我们主要集中在两个事件上：open和message。

连接建立后，开始监听mousemove事件，用于将用户鼠标的位置实时发送到服务器端，再由服务器发送到其他客户端。

```
ws.onopen = function () {
  document.onmousemove = function (ev) {
    ws.send(JSON.stringify({ x: ev.clientX, y: ev.clientY }));
  }
}
```

如在此前提到的，接收到的消息无外乎两种，要么是用户光标移动了，要么是用户断开连接了：

```
// we instantiate a variable to keep track of initialization for this client
var initialized;

ws.onmessage = function (ev) {
   var obj = JSON.parse(ev.data);

  // the first message is the position of all existing cursors
  if (!initialized) {
    initialized = true;
    for (var id in obj) {
      move(id, obj[id]);
    }
  } else {
    // other messages can either be a position change or
    // a disconnection
    if ('disconnect' == obj.type) {
      remove(obj.id);
    } else {
      move(obj.id, obj.pos);
    }
  }
}
```

接着声明move和remove函数。

对于move函数，我们首先要确保光标对应的元素是否存在。我们通过查找是否有ID为cursor-{id}的DOM元素来完成这个检查。如果元素不存在，那么就创建一个图片元素，设置图片URL以及浮动的样式。

接着就调整它在屏幕上的位置：

176

```
function move (id, pos) {
  var cursor = document.getElementById('cursor-' + id);

  if (!cursor) {
    cursor = document.createElement('img');
    cursor.id = 'cursor-' + id;
    cursor.src = '/cursor.png';
    cursor.style.position = 'absolute';
    document.body.appendChild(cursor);
```

```
  }

  cursor.style.left = pos.x + 'px';
  cursor.style.top = pos.y + 'px';
}
```

对于remove来说，只要简单地将DOM元素删除就好了：

```
function remove (id) {
  var cursor = document.getElementById('cursor-' + id);
  cursor.parentNode.removeChild(cursor);
}
```

运行示例程序

和此前一样，我们需要做的就是运行服务器程序，并通过浏览器来访问。确保打开了多个浏览器标签页（如图10-2所示），来体验实时交互。

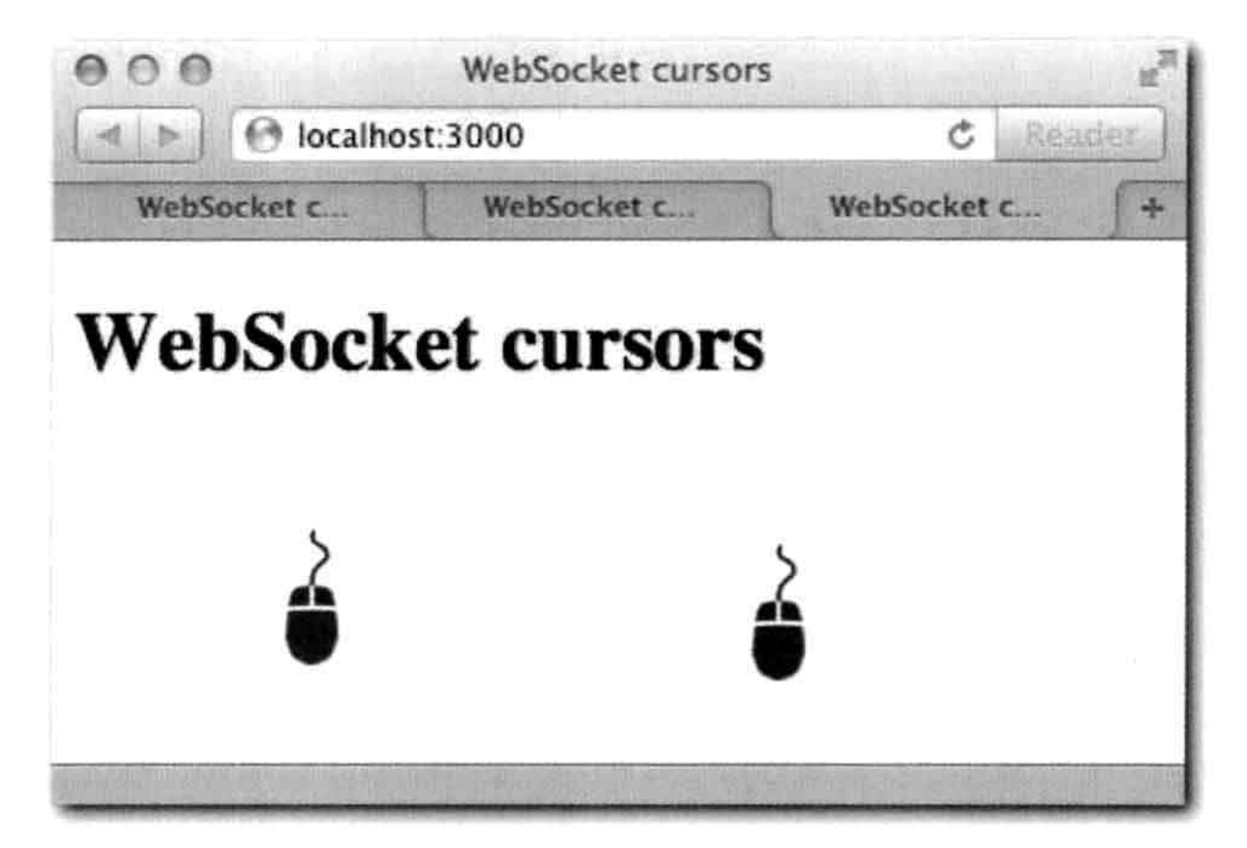


图10-2：移动的鼠标位置实时地反馈到了连入的所有客户端cursor.png图片来自http://thenounproject.com

177

面临一个挑战

尽管本例基本功能都已完备，但是要真的应用到实际产品中还有不少的工作要做。

关闭并不意味着断开连接

当服务器端或者客户端触发close事件时，意味着：TCP连接很可能已经关闭了。而事实上，并非总是如此。电脑有可能会意外关机，网络错误也有可能发生，甚至不小心把一杯水倒在了键盘上都有可能。在类似这样的情况下，close事件可能永远都不会触发！

解决这个问题的方法就是利用超时和心跳检查。要处理这样的情况，我们需要每隔几秒钟

向客户端发送一段消息来确保客户端还“活”着，要是发送失败则认为客户端已经强制断开连接了。

JSON

随着程序复杂度的提升，服务器端和客户端往返的数据量也会变大。

在第二个示例中，会严重依赖JSON进行手动编码和解码工作。因为这部分工作是一个应用中非常常见的任务，所以需要将其抽象出来。

重连

要是客户端临时断开怎么办？大部分应用程序会尝试让用户自动重连。而对于本章中的例子来说，一旦发生断开情况，就只能通过刷新浏览器来重新连接了。

广播

广播对于实时应用来说是非常常规的模式，用于和其他客户端进行交互。对于这部分功能，我们不需要手动去定义自己的广播机制。

WebSocket属于HTML5：早期浏览器不支持

WebSocket是一项新技术。大部分浏览器、代理、防火墙以及杀毒软件都还不完全支持这种新的协议。因此，我们需要一种支持早期浏览器的方案。

解决方案

178

幸运的是，所有这些问题都有解决方案。在下一章中，我们会介绍使用一个名为`socket.io`的模块，它的作用就是在保持简单、加速基于 WebSocket通信的前提下，解决所有上述这些问题。

小结

本章介绍了WebSocket API和WebSocket协议的基础知识，以及如何在Node.js使用WebSocket进行消息的快速传递。同时，通过第一个示例程序，介绍了WebSocket的基本用法。

随后，通过一个多人示例应用展现了WebSocket的威力：它在消息传递时只附带极少的附加数据，使得它能够以尽可能快的速度完成消息传送。

最后，本章提出了WebSocket API并非所有浏览器都支持的弱点，并引出了下一章要介绍的能够解决这一问题的socket.io。

CHAPTER

11 Socket.IO

正如此前提过的，要将WebSocket用到应用中，并非简单地把 WebSocket实现了就可以的。

Socket.IO是我开发的一个项目，用于解决此前提过的使用WebSocket过程中一系列常见问题。它在保留了简洁API的前提下，提供了非常好的灵活性。

Server API

```
io.listen(app);
io.sockets.on('connection', function
(socket) {
  socket.emit('my event', { my: 'object' });
});
```

Browser/Client API

```
var socket = io.connect();
socket.on('my event', function (obj) {
  console.log(obj.my);
});
```

180 传输

Socket.IO最诱人的特性之一就是消息的传递是基于传输的，而非全部依靠WebSocket，也就是说，Socket.IO可以在绝大部分的浏览器和设备上运行，从IE6到iOS都支持。

例如，在使用一项称为long polling技术的时候，就可以通过Ajax来实现实时消息传输。简单来说，这项技术是通过持续发送一系列的Ajax请求来实现的，但是，当服务器端没有数据返回到客户端时，连接还会持续打开20 ~ 50秒，以确保不再有额外的数据通过HTTP请求/响应头传递过来。

Socket.IO会自动使用像long polling这样复杂的技术，但其API保持了与WebSocket一样的简洁。

另外，即使浏览器端支持的WebSocket被代理或者防火墙禁止了，Socket.IO依然能够通过采用别的技术来处理这类问题。

断开 VS 关闭

Socket.IO带来的另一个基础功能就是对超时的支持。正如我们在第6章和第10章中讨论的，在实际情况下，应用不能依赖TCP连接一定能够正常关闭。

本章中，我们使用Socket.IO，监听的是`connect`事件而不是`open`事件，以及`disconnect`事件而不是`close`事件。原因是Socket.IO提供了可靠的事件机制。若客户端停止传输数据，但在一定的时间内又没有正常地关闭连接，Socket.IO就认为它是断开连接了。

这样一来，就能够让你专注在应用逻辑本身，而无须去过多担心网络的各种不确定情况。

当连接丢失时，Socket.IO默认会自动重连。

事件

至此，你看到了，典型的Web通信方式是通过HTTP来收取（发送）文档（资源）的。但是，在实时Web世界中，都是基于事件传输的。

Socket.IO仍然允许你像WebSocket那样传输简单文本信息，除此之外，它还支持通过分发（emit）和监听（listen）事件来进行JSON数据的收发。下面这个例子展示了Socket.IO像WebSocket那样进行消息的收发：

```
io.sockets.on('connection', function (socket) {
  socket.send('a');
  socket.on('message', function (msg) {
    console.log(msg);
  });
});
```

181

回想一下第10章光标的例子，要是用Socket.IO来实现的话，代码可以变得非常简单：

Client code

```
var socket = io.connect();

socket.on('position', move);

socket.on('remove', remove);
```

注意了，使用Socket.IO可以在应用中根据数据的含义进行频道分类，不再需要对单一事件（消息）中收到的事件进行解析。事件可以接收任意数量的JSON编码的参数：Number、Array、String、Object，等等。

命名空间

Socket.IO还提供了另一个强大的特性，它允许在单个连接中利用命名空间来将消息彼此区分开来。

有的时候，应用程序需要进行逻辑拆分，但考虑到性能、速度之类的原因，使用同一个连接还是可以接受的。考虑到我们无法事先获悉客户端的速度的快慢、浏览器的好坏，不依赖同时打开过多的连接通常是个不错的主意。

因此，Socket.IO允许监听多个命名空间中的connection事件。

```
io.sockets.on('connection');
io.of('/some/namespace').on('connection')
io.of('/some/other/namespace').on('connection')
```

尽管当通过如下方式从浏览器中获取连接时，可以获取到不同的连接对象，但是，通常只会使用一个传输通道（像WebSocket连接一样）：

```
var socket = io.connect();
var socket2 = io.connect('/some/namespace');
var socket3 = io.connect('/some/other/namespace');
```

某些场景下，为了更好地抽象，应用程序的部分代码或模块书写的时候完全是互相独立的。部分客户端JavaScript代码可能完全不知道另外一部分并行执行的代码。

比如，构建一个社交网络，在农场游戏旁边展示一个实时聊天程序。尽管，它们可以共享一些如授权用户的信息这样的通用的数据，但书写代码时让它们都能够完全控制一个socket依然是个很好的主意。 182

归功于命名空间（也可以称为多路传输），那样的socket不必非得是自己分配的真正的TCP socket。Socket.IO对同样的资源（为用户选择的传输通道）进行频道切分，并将数据传输给对应的回调函数。

至此，你已经了解了Socket.IO和WebSocket之间主要的不同点，接下来，是时候开始构建第一个示例程序——聊天程序了。

聊天程序

初始化程序

和websocket.io一样，将socket.io绑定到常规的http.Server就可以处理socket.io的请求和响应了：

package.json

```
{
    "name": "chat.io"
  , "version": "0.0.1"
  , "dependencies": {

        "express": "2.5.1"
      , "socket.io": "0.9.2"
    }
}
```

按照惯例，创建完package.json文件后，确保要运行npm install来安装所有的依赖。

构建服务器

和websocket.io一样，现在构建一个带static中间件的普通Express应用[1]：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , sio = require('socket.io')

/**
 * 创建 app
 */

app = express.createServer(
    express.bodyParser()
, express.static('public')
);
```

183

1 译者注：Express3中需要首先用http.createServer来将app绑到server上，需要添加：var server = require("http").createServer(app); sio.listen(server); 。同时，Express3中不能将中间件放在express(…)构造器参数中，需要使用app.use(express.bodyParser()); 。

```
/**
 * 监听
 */

app.listen(3000);
```

接下来，是时候将socket.io绑定到APP上了。和websocket.io一样，调用sio.listen即可：

```
var io = sio.listen(app);
```

接着，设置连接监听器：

```
io.sockets.on('connection', function (socket) {
  console.log('Someone connected');
});
```

好了，现在一旦有连接进来，就会在控制台输出简单的信息了。由于 Socket.IO是自定义的API，所以必须要在浏览器中载入Socket.IO客户端。

构建客户端

因为使用了static中间件，并将public目录设置为了要托管的目录，接下来我们就在该目录下创建一个index.html文件。

这次，为了方便，我们将聊天程序的逻辑部分从HTML代码中分离出来，单独放到chat.js文件中。

Socket.IO中方便的一点在于，当它绑定到http.Server后，所有以/socket.io开始的URL都会被其拦截。

Socket.IO还自带了其浏览器端运用的代码。因此，我们无须担心如何获取和托管客户端代码。

注意，下面这段代码，我们创建了一个<script>标签，并添加对/socket.io/socket.io.js的引用：

184

index.html

```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/chat.js"></script>
    <link href="/chat.css" rel="stylesheet" />
  </head>
  <body>
```

```
    <div id="chat">
      <ul id="messages"></ul>
      <form id="form">
        <input type="text" id="input" />
        <button>Send</button>
      </form>
    </div>
  </body>
</html>
```

现在，chat.js确保客户端载入完成后就开始连接。若一切都正常，就应该能在控制台看到Someone connected这样的输出了。

chat.js

```
window.onload = function () {
  var socket = io.connect();
}
```

所有Socket.IO客户端代码暴露出来的方法和类都在io命名空间中。

io.connect和new WebSocket类似，不过更智能。本例中，因为没有传递参数给它，所以，它会尝试向页面所在的主机发起连接，这也符合本例的要求。

使用如下命令来运行上述应用：

```
$ node server
```

接着，通过浏览器访问http://localhost:3000。应当就能看到 Socket.IO的日志器输出的关于Socket.IO内部发生的情况；比如，从输出结果中能够看到客户端使用的传输方式（见图11-1）。

185

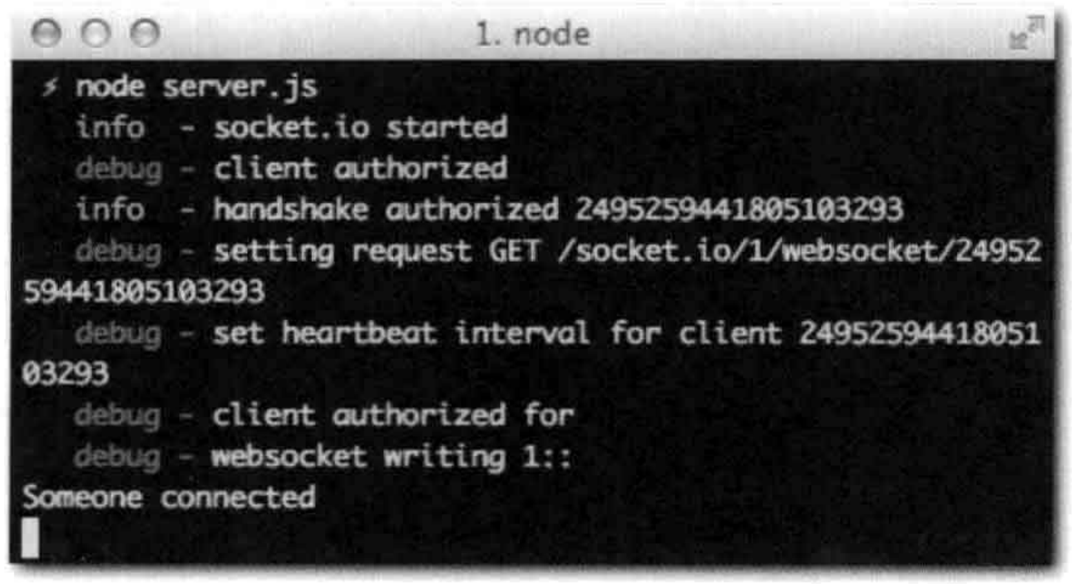


图11-1：socket.io自身的调试信息以及应用通过console.log输出的信息

像本例中那样，如果通过主流浏览器来连接Socket.IO服务器，那么 Socket.IO就会使用WebSocket来通信。

Socket.IO总会尝试选择对用户来说速度最快、对服务器性能来说最好的方法来建立连接，要是条件达不到，那么首先会保证连接正常。

事件和广播

现在已经成功连接上了，那么接下来就应该开始编写Socket.IO服务器端基础代码了。

广播用户加入信息

当有用户连接到服务器时，我们要通知其他人有人连接进来了。由于这属于非用户发送的特殊消息，所以我们称之为通告，并用相应的样式标识出来。

客户端首先要做的就是询问用户的名字。

因为要在用户成功连接后才能聊天，所以我们需要先将聊天窗口隐藏：

chat.css

```
/* … */
#chat { display: none }
```

接着，在用户连接成功后就将聊天窗口显示出来。这里我们需要监听已创建socket上的connect事件（此前定义在window.onload中的函数）：

186

chat.js

```
socket.on('connect', function () {
  // 通过join事件发送昵称
  socket.emit('join', prompt('What is your nickname?'));

  // 显示聊天窗口
  document.getElementById('chat').style.display = 'block';
});
```

服务器端则需要监听join事件，并将收到的消息通知给其他人，告诉他们有新用户连接进来了。我们将此前的io.sockets的connection处理器替换为如下形式即可：

server.js

```
// …
io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
  });
});
```

注意，socket.broadcast.emit中的broadcast是一种标志信息，它改变了emit函数的行为。

要是在上述例子中，我们直接调用socket.emit，那么仅仅是将消息返回给客户端。而我们真正需要的是将消息广播给所有其他的用户，所以这里需要broadcast标志。

再次回到客户端，我们需要监听announcement事件，并在DOM的消息列表中创建一个元素。将下述代码添加到connect处理器的最后：

chat.js

```
socket.on('announcement', function (msg) {
  var li = document.createElement('li');
  li.className = 'announcement';
  li.innerHTML = msg;
  document.getElementById('messages').appendChild(li);
});
```

广播聊天消息

接下来，我们要实现让用户发消息给其他人。

187 当用户在表单中输入消息并提交时，我们需要分发一个text事件，并将输入的消息发送出去：

chat.js

```
var input = document.getElementById('input');
document.getElementById('form').onsubmit = function () {
  socket.emit('text', input.value);

  // 重置输入框
  input.value = '';
  input.focus();

  return false;
}
```

很显然，当用户书写完消息并提交后，肯定不希望服务器再将该消息发回来。所以，在消息发送后，我们就立刻调用addMessage将消息显示出来：

chat.js

```
function addMessage (from, text) {
  var li = document.createElement('li');
  li.className = 'message';
  li.innerHTML = '<b>' + from + '</b>: ' + text;
  document.getElementById('messages').appendChild(li);
```

```
}
document.getElementById('form').onsubmit = function () {
  addMessage('me', input.value);
  // . . .
}
```

在从其他用户处收到消息后，我们也需要做同样的处理。这里，我们可以简单地传递一个对addMessage函数的引用，同时在服务器端，要确保广播消息时参数都正确。

chat.js

```
// …
socket.on('text', addMessage);
```

server.js

```
socket.on('text', function (msg) {
  socket.broadcast.emit('text', socket.nickname, msg);
});
```

至此，客户端和服务器端的代码大致如下所示：

188

chat.js

```
window.onload = function () {
  var socket = io.connect();
  socket.on('connect', function () {
    // 通过join事件发送昵称
    socket.emit('join', prompt('What is your nickname?'));

    // 显示聊天窗口
    document.getElementById('chat').style.display = 'block';

  socket.on('announcement', function (msg) {
    var li = document.createElement('li');
    li.className = 'announcement';
    li.innerHTML = msg;
    document.getElementById('messages').appendChild(li);
  });
});

function addMessage (from, text) {
  var li = document.createElement('li');
  li.className = 'message';
  li.innerHTML = '<b>' + from + '</b>: ' + text;
  document.getElementById('messages').appendChild(li);
}
```

```
  var input = document.getElementById('input');
  document.getElementById('form').onsubmit = function () {
    addMessage('me', input.value);
    socket.emit('text', input.value);

    // 重置输入框
    input.value = '';
    input.focus();

    return false;
  }

  socket.on('text', addMessage);
}
```

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , sio = require('socket.io')

/**
 * 创建app
 */

app = express.createServer(
    express.bodyParser()
  , express.static('public')
);

/**
 * 监听
 */

app.listen(3000);

var io = sio.listen(app);

io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
  });

  socket.on('text', function (msg) {
    socket.broadcast.emit('text', socket.nickname, msg);
```

189

```
  });
});
```

运行server.js应当就能看到一个如图11-2所示的功能完整的实时聊天应用。

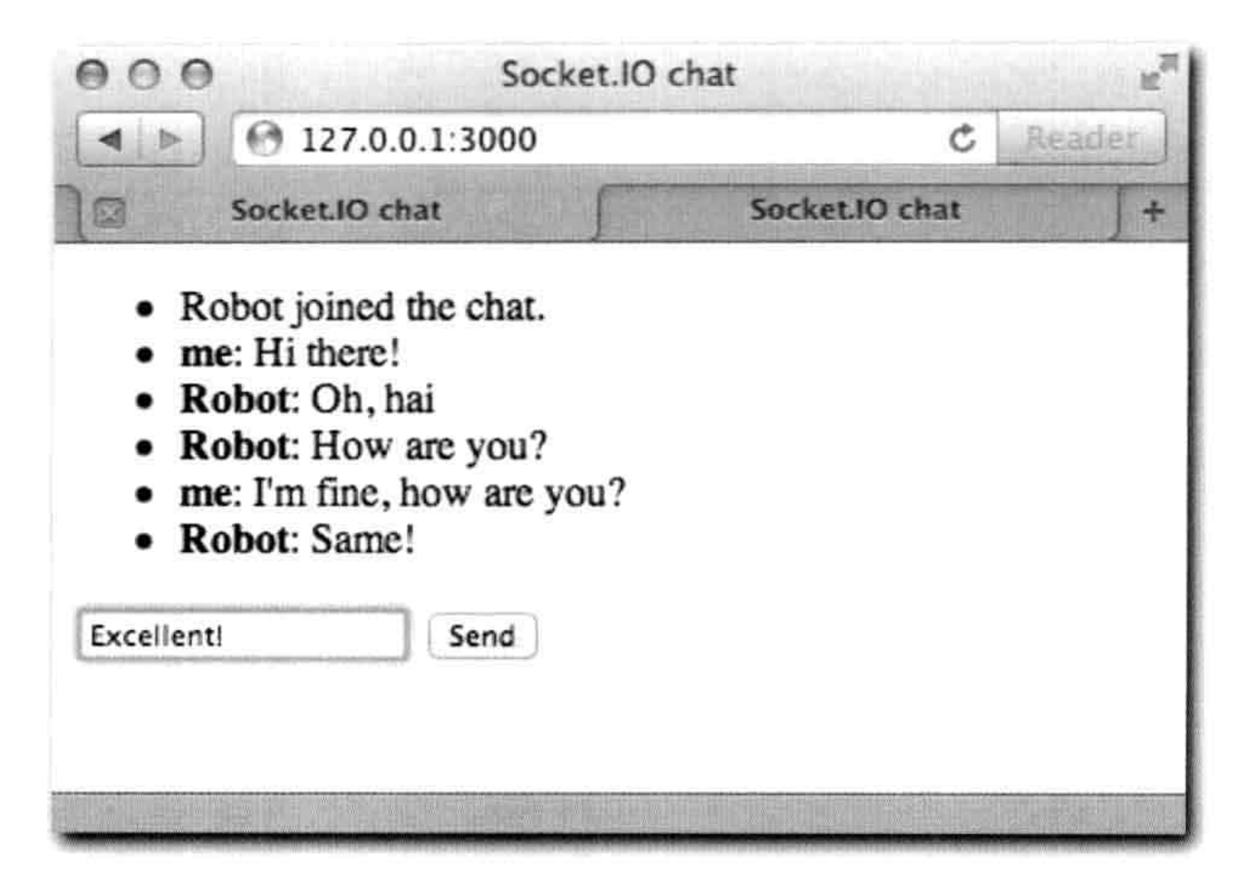


图11-2：聊天程序实战。这里通过多个浏览器标签页来聊天

接下来，我们要了解更多关于事件回调函数的内容，以及如何使用它们实现新特性。 190

消息接收确认

在聊天应用中，我们在用户按下回车键后立刻调用addMessage，这会让人产生一种错觉，感觉消息瞬间就发送成功了。

和WebSocket一样，Socket.IO并不强制对每条发送的消息做回应。不过，有的时候，我们需要确认消息是否达到。Socket.IO把这类确认消息叫做确认（acknowledgment）。

要实现这样的通知响应，我们要做的就是在分发事件的时候提供一个回调函数。

首先，我们要获得通过addMessage函数创建的消息元素，以便于在收到确认响应后，对该消息添加一个CSS类。接着，就可以在该消息后显示一个漂亮的小图标。

/chat.js

```
function addMessage (from, text) {
  // …
  return li;
}
```

接下来，我们添加一个回调函数。Socket.IO也允许在接收确认响应的回调函数中接收数据。本例中，可以在接收到消息后发送一个时间戳：

/chat.js

```
document.getElementById('form').onsubmit = function () {
  var li = addMessage('me', input.value);
  socket.emit('text', input.value, function (date) {
    li.className = 'confirmed';
    li.title = date;
  });
```

在服务器端，Socket.IO会添加一个回调函数作为事件的最后一个参数：

/server.js/

```
// …
socket.on('text', function (msg, fn) {
  // …
  // 确认消息已接收
  fn(Date.now());
});
```

191 现在，当服务器端接收到客户端发送的消息，发送确认响应后，一个CSS类，以及title属性就会添加到消息列表的最后一个元素中。这样做会带来两个好处：用户按下回车键后立刻就看到输入的消息，这使得应用获得最好的响应；另外还能通过CSS给用户反馈（比如：在消息后面显示一个如图11-3所示的对钩图标）。

图11-3：本例中，在确认响应收到后，利用CSS设置一个背景图

一个轮流做DJ的应用

要是把我们的聊天应用扩展为一个DJ的应用，那得有多酷啊？

- 服务器初始选择一名DJ。

- DJ有权利请求查询API，获取查询结果，选择一首歌。然后将这首歌广播给所有其他听众。
- 当DJ离开时，系统就会开放DJ人选给下一个用户。

扩展聊天应用

聊天应用的基础足够强健，可以添加DJ特性。

首先要做的是，初始化的时候选择一名DJ。因为我们还要记录当前播放的歌曲，所以需要申明两个变量：currentSong和dj。

由于DJ是可以更换的，所以，我们定义一个elect函数来执行选择DJ和发布公告的任务。当join事件分发时，DJ就会被选出来，或者（已经有DJ的情况下）将当前播放的歌曲（currentSong）发送给刚加入的用户。后面，实现了搜索之后，currentSong会被包含在一个对象中发送出去。

192

server.js

```
var io = sio.listen(app)
, currentSong
  , dj

function elect (socket) {
  dj = socket;
  io.sockets.emit('announcement', socket.nickname + ' is the new dj');
  socket.emit('elected');
  socket.dj = true;
  socket.on('disconnect', function () {
    dj = null;
    io.sockets.emit('announcement', 'the dj left - next one to join becomes dj');
  });
}

io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
    if (!dj) {
      elect(socket);
    } else {
      socket.emit('song', currentSong);
    }
  });
  // …
});
```

elect函数完成如下几件事情：

1. 将当前用户选为DJ。
2. 分发公告给所有人DJ已经选取完毕。
3. 通过分发elected事件，让DJ知道自己被选中了。
4. 当DJ断开连接时，将DJ的名额留给下一个进来的人。

客户端，将歌曲选择的界面代码添加到聊天表单下面即可：

index.html

```
<div id="playing"></div>
<form id="dj">
  <h3>Search songs</h3>
  <input type="text" id="s" />
  <ul id="results"></ul>
  <button type=submit>Search</button>
</form>
```

193 集成Grooveshark API

Grooveshark（http://grooveshark.com）提供了一个简单易用的API ——TinySong，能满足我们的需求。

TinySong允许如下的查询方式：

```
GET http://tinysong.com/s/Beethoven?key={apiKey}&format=json
```

返回结果如下：

```
[
  {
    "Url": "http:\/\/tinysong.com\/7Wm7",
    "SongID": 8815585,
    "SongName": "Moonlight Sonata",
    "ArtistID": 1833,
    "ArtistName": "Beethoven",
    "AlbumID": 258724,
    "AlbumName": "Beethoven"
  },
  {
    "Url": "http:\/\/tinysong.com\/6Jk3",
    "SongID": 564004,
    "SongName": "Fur Elise",
    "ArtistID": 1833,
    "ArtistName": "Beethoven",
```

```
    "AlbumID": 268605,
    "AlbumName": "Beethoven"
  },
  {
    "Url": "http:\/\/tinysong.com\/8We2",
    "SongID": 269743,
    "SongName": "The Legend Of Lil' Beethoven",
    "ArtistID": 7620,
    "ArtistName": "Sparks",
    "AlbumID": 204019,
    "AlbumName": "Sparks"
  }
]
```

因此，我需要暴露一个叫search的Socket.IO事件，内部使用superagent模块来进行对查询API的调用并返回其结果。

将superagent模块添加到package.json中，并添加模块依赖：

server.js

194

```
var express = require('express')
  , sio = require('socket.io')
  , request = require('superagent')
```

package.json

```
  , "dependencies": {
        "express": "2.5.1"
      , "socket.io": "0.9.2"
      , "superagent": "0.4.0"
    }
```

注意了，在查询URL中必须要包含API key，API key可以去 http://tinysong.com网站申请。

定义apiKey的方式如下：

server.js

```
var io = sio.listen(app)
  , apiKey = '{ your API key }'
  , currentSong
  , dj
```

接着，定义search事件：

```
socket.on('search', function (q, fn) {
    request('http://tinysong.com/s/' + encodeURIComponent(q)
      + '?key=' + apiKey + '&format=json', function (res) {
```

```
    if (200 == res.status) fn(JSON.parse(res.text));
  });});
```

注意，上述代码中需要手动解析JSON返回结果。这是因为TinySong目前没有发送正确的Content-Type响应头信息，导致 superagent无法自动启用JSON解析功能。

接着，我们要将查询功能添加到应用中，但是只能让DJ能够选择歌曲。

在chat.css文件中，添加如下两行代码：

```
#results a { display: none; }
form.isDJ #results a { display: inline; }
```

195 接着我们要添加查询逻辑，将从Socket.IO回调函数中获取到的查询结果展示出来供其选择。

在chat.js文件中，添加如下内容：

```
// search form
var form = document.getElementById('dj');
var results = document.getElementById('results');
form.style.display = 'block';
form.onsubmit = function () {
  results.innerHTML = '';
  socket.emit('search', document.getElementById('s').value, function (songs) {
    for (var i = 0, l = songs.length; i < l; i++) {
      (function (song) {
        var result = document.createElement('li');
        result.innerHTML = song.ArtistName + ' - <b>' + song.SongName + '</b> ';
        var a = document.createElement('a');
        a.href = '#';
        a.innerHTML = 'Select';
        a.onclick = function () {
          socket.emit('song', song);
          return false;
        }
        result.appendChild(a);
        results.appendChild(result);
      })(songs[i]);
    }
  });
  return false;
};

socket.on('elected', function () {
  form.className = 'isDJ';
});
```

上述代码中大部分都在处理DOM。因为服务器端从TinyURL的API中把所有的歌曲都发送到了客户端，因此可任由客户端对其进行渲染。在本例中，我们首先显示歌手名，紧跟着歌曲

名（对应的是ArtistName和SongName）。

在收到elected事件后，通过改变表单的className来显示每首歌曲的选择链接。

点击Select链接后，客户端发送song事件到服务器端，服务器端要做的就是记录当前歌曲，并广播给所有人。在server.js文件中，添加如下代码：

196

server.js

```
socket.on('song', function (song) {
  if (socket.dj) {
    currentSong = song;
    socket.broadcast.emit('song', song);
  }
});
```

至此，用户已经可以搜索和接收歌曲了，最后就剩下播放歌曲的功能了。 元素就是为此而预留的。

播放歌曲

和addMessage函数一样，我们也需要定义一个函数来标记当前正在播放的歌曲。

将下述代码添加到chat.js文件中。play函数就是简单地将当前播放的歌曲按照歌手、歌名的方式展现出来，与此同时，它还注入了一个 iframe，指向TinySong的Url字段，用来播放歌曲。

```
var playing = document.getElementById('playing');
function play (song) {
  if (!song) return;
  playing.innerHTML = '<hr><b>Now Playing: </b> '
    + song.ArtistName + ' ' + song.SongName + '<br>';

  var iframe = document.createElement('iframe');
  iframe.frameborder = 0;
  iframe.src = song.Url;
  playing.appendChild(iframe);
};
```

与此前一样，该函数用于两个场景：DJ（为自己）选择了一首歌后，以及DJ向其他普通用户发送song事件的时候。

对于第二个场景，我们只需要在chat.js中，将play函数作为回调函数传递给song事件。

```
socket.on('song', play);
```

对于DJ要立刻听到所选歌曲，我们就需要在他选择之后调用play函数。回到onclick处理器，在那里需要分发song事件给服务器并调用play函数，因此，该处理器代码就会是如下

所示的样子：

```
a.onclick = function () {
  socket.emit('song', song);
  play(song);
  return false;
}
```

197

完成！回忆一下一开始服务器端join事件处理器部分代码，要是有currentSong就会分发song事件。也就是说，不仅在DJ选歌曲前加入的用户能播放歌曲，在这之后填写完昵称加入的用户也能播放歌曲（见图11-4）。

聊天+DJ应用完整代码大致如下所示：

server.js

```
var express = require('express')
  , sio = require('socket.io')
  , request = require('superagent')

app = express.createServer(
    express.bodyParser()
  , express.static('public')
);

app.listen(3000);

var io = sio.listen(app)
  , apiKey = '{ your API key }'
  , currentSong
  , dj

function elect (socket) {
  dj = socket;
  io.sockets.emit('announcement', socket.nickname + ' is the new dj');
  socket.emit('elected');
  socket.dj = true;
  socket.on('disconnect', function () {
    dj = null;
    io.sockets.emit('announcement', 'the dj left - next one to join becomes dj');
  });
}

io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
    if (!dj) {
      elect(socket);
```

```
  } else {
    socket.emit('song', currentSong);
  }
});
socket.on('song', function (song) {
  if (socket.dj) {
    currentSong = song;
    socket.broadcast.emit('song', song);
  }
});

socket.on('search', function (q, fn) {
  request('http://tinysong.com/s/' + encodeURIComponent(q)
    + '?key=' + apiKey + '&format=json', function (res) {
      if (200 == res.status) fn(JSON.parse(res.text));
    });
  });

  socket.on('text', function (msg) {
    socket.broadcast.emit('text', socket.nickname, msg);
  });
});
```

198

chat.js

```
window.onload = function () {
  var socket = io.connect();
  socket.on('connect', function () {
    // 通过join事件发送昵称
    socket.emit('join', prompt('What is your nickname?'));

    // 显示聊天窗口
    document.getElementById('chat').style.display = 'block';

    socket.on('announcement', function (msg) {
      var li = document.createElement('li');
      li.className = 'announcement';
      li.innerHTML = msg;
      document.getElementById('messages').appendChild(li);
    });
  });

  function addMessage (from, text) {
    var li = document.createElement('li');
    li.className = 'message';
    li.innerHTML = '<b>' + from + '</b>: ' + text;
    document.getElementById('messages').appendChild(li);
  }
```

199

```
  var input = document.getElementById('input');
  document.getElementById('form').onsubmit = function () {
    addMessage('me', input.value);
    socket.emit('text', input.value);
  // 重置输入框
  input.value = '';
  input.focus();

  return false;
}

socket.on('text', addMessage);

// 播放歌曲
var playing = document.getElementById('playing');
function play (song) {
  if (!song) return;
  playing.innerHTML = '<hr><b>Now Playing: </b> '
    + song.ArtistName + ' ' + song.SongName + '<br>';

  var iframe = document.createElement('iframe');
  iframe.frameborder = 0;
  iframe.src = song.Url;
  playing.appendChild(iframe);
};
socket.on('song', play);

// 查询表单
var form = document.getElementById('dj');
var results = document.getElementById('results');
form.style.display = 'block';
form.onsubmit = function () {
  results.innerHTML = '';
  socket.emit('search', document.getElementById('s').value, function (songs) {
    for (var i = 0, l = songs.length; i < l; i++) {
      (function (song) {
        var result = document.createElement('li');
        result.innerHTML = song.ArtistName + ' - <b>' + song.SongName + '</b> ';
        var a = document.createElement('a');
        a.href = '#';
        a.innerHTML = 'Select';
        a.onclick = function () {
          socket.emit('song', song);
          play(song);
          return false;
        }
        result.appendChild(a);
        results.appendChild(result);
      })(songs[i]);
```

```
      }
    });
    return false;
  };
  socket.on('elected', function () {
    form.className = 'isDJ';
  });
}
```

200

index.html

```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/chat.js"></script>
    <link href="/chat.css" rel="stylesheet" />
  </head>
  <body>
    <div id="chat">
      <ul id="messages"></ul>
      <form id="form">
        <input type="text" id="input" />
        <button>Send</button>
      </form>
      <div id="playing"></div>

      <form id="dj">
        <h3>Search songs</h3>
        <input type="text" id="s" />
        <ul id="results"></ul>
        <button>Search</button>
      </form>
    </div>
  </body>
</html>
```

201

图11-4：DJ+聊天应用实战

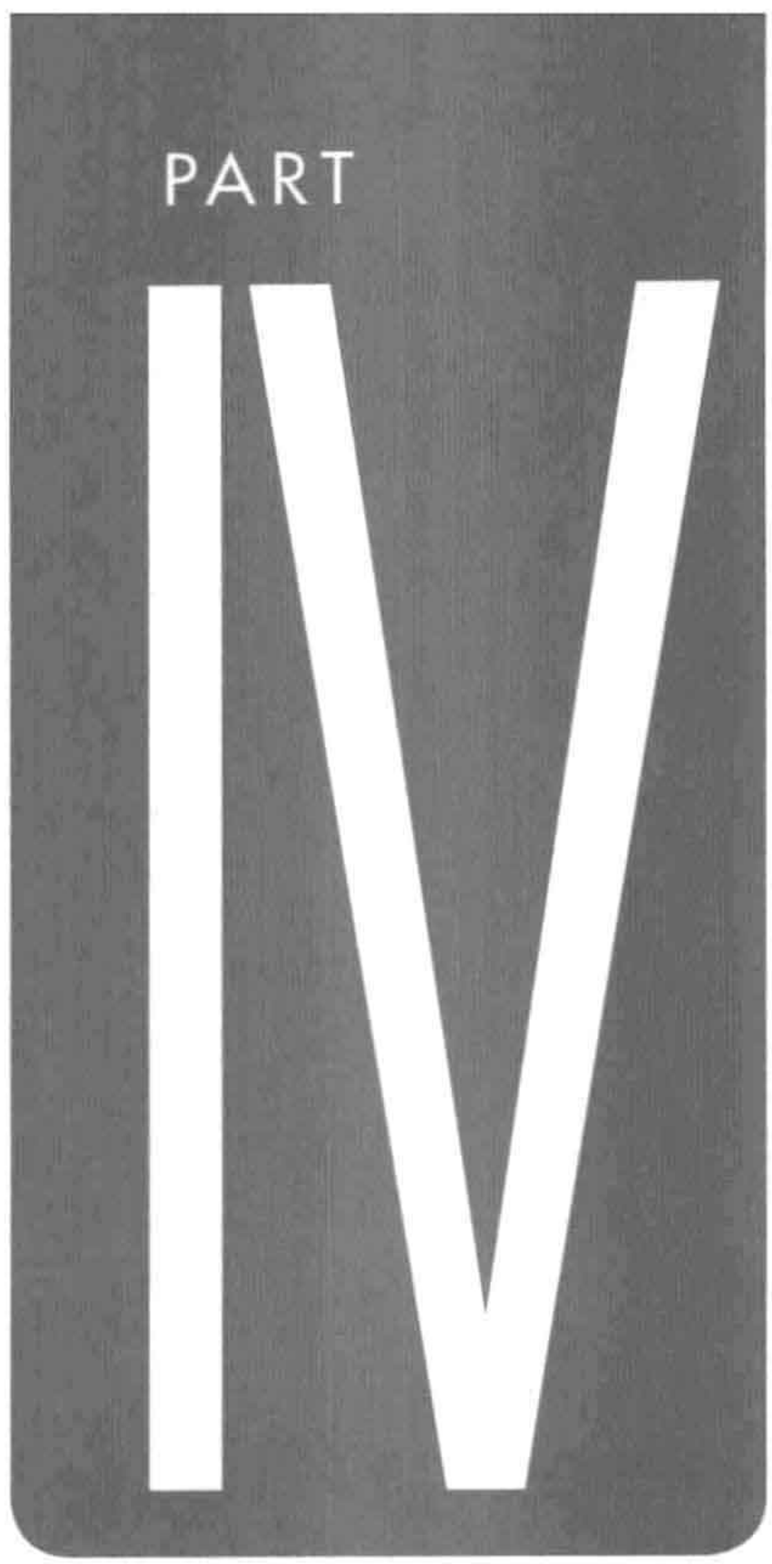
小结

Socket.IO提供了足够简单但却十分强大的API，用于构建实时消息快速通信的应用。Socket.IO不仅保证了消息会尽可能快地进行传输，而且还能在所有的浏览器以及绝大部分移动设备上运行。

本章介绍了如何构建一个简单的应用以及如何使用Socket.IO提供的语法糖。还介绍了通过使用事件的方式来组织用户和服务器端传输的不同类型的数据。

书写实时应用最基础的部分就是广播。本章介绍了如何在服务器端分发事件给所有人，以及如何将一个用户的消息传递给其他人。作为例子，我们使用该技术实现了把DJ挑选的歌曲播放给其他所有的用户。

还有一件值得一提的事情是，绝大部分的功能都在客户端实现：编写代码来根据不同的消息类型进行相应的界面展现。由于本章只关注Socket.IO，所以没有介绍模板引擎以及其他更高层的框架，使用这些可以避免和DOM API直接打交道，不过，在实际情况下，随着应用程序复杂度的提高，这些也会变得更加复杂。



数据库

CHAPTER 12 MongoDB

CHAPTER 13 MySQL

CHAPTER 14 Redis

CHAPTER

12 MongoDB

MongoDB是一个面向文档，schema无关（schema-less）的数据库，它非常适合于Node.js 205
应用以及云端部署。

与MySQL及PostgreSQL是根据固定的结构设计（schema）将数据存储在表中不同，MongoDB可以将任意类型的文档数据存储到集合中（schema无关），这也是MongoDB最有意思的特性之一。

例如，创建下面这张为Web应用保存用户信息的表： 206

First	Last	Email	Twitter
Guillermo	Rauch	rauchg@gmail.com	rauchg

在构建应用时，决定将用户信息按照上面这样的结构设计进行存储。需要如下这些信息：first name、last name、email以及Twitter ID。

随着应用的发展、需求发生了改变，或者随着时间的推移，又有了新的需求，可能需要增加或者删除表中的某些列。

然而，这样一个基础性问题，若要通过传统的（SQL）数据库来实现，从操作上和性能上

来讲都需要耗费非常高的成本来修改表设计。

比如，在MySQL中，每一次修改表的设计结构，都需要运行如下这个命令才能实现添加一个新的列：

```
$ mysql
  > ALTER TABLE profiles ADD COLUMN . . .
```

对于删除一列或多列的情况也是如此。

在MongoDB中，则可以将数据都看作文档，其设计非常灵活。当有数据存储后，这些文档就会以一种非常接近（或者说在绝大多数情况下就是JSON格式）JSON格式的形式存储：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "email": "rauchg@gmail.com"
  , "age": 21
  , "twitter": "rauchg"
}
```

MongoDB还有一个非常重要的特性，能够将其与其他键—值形式的 NoSQL数据库区别开来，就是文档可以是任意深度的。

例如，可以将社交信息以如下结构进行存储，而不是全都将它们直接作为文档的键来存储：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "email": "rauchg@gmail.com"
  , "age": 21
  , "social_networks": {
        "twitter": "rauchg"
      , "facebook": "rauchg@gmail.com"
      , "linkedin": 27760647
    }
}
```

207

如上述代码所示，数据类型可以混用。这里，`twitter`和`facebook`信息都是字符串类型的，而`linkedin`是数字类型。当通过Node.js获取到存储的文档数据后，拿到的数据类型也是和存储时一模一样的。

本章将会介绍MongoDB最常用的功能，以及如何获得最灵活、最高效（通过索引）存储方案的最佳实践。本章还会介绍多种查询文档的方法，以及如何使用Mongoose来简化使用方式，Mongoose是我和 Nathan White一同书写的Node.js模块，它为MongoDB和JavaScript提供了传统关系型数据库ORM（Object-Relational Mapper）的部分功能。对MongoDB来说，这类项目

更贴切的叫法应该是ODM： Object Document Mapper。

安装

记住，本章使用的是MongoDB分支上最新的2.x版本。[1]

通过官网：www.mongodb.org/的下载页面就能获取到MongoDB。与此同时，你或许也会有兴趣看下各平台下MongoDB的快速指南： www.mongodb.org/display/DOCS/Quickstart。

通过执行mongo客户端，看到如图12-1所示的界面就表示安装成功了。

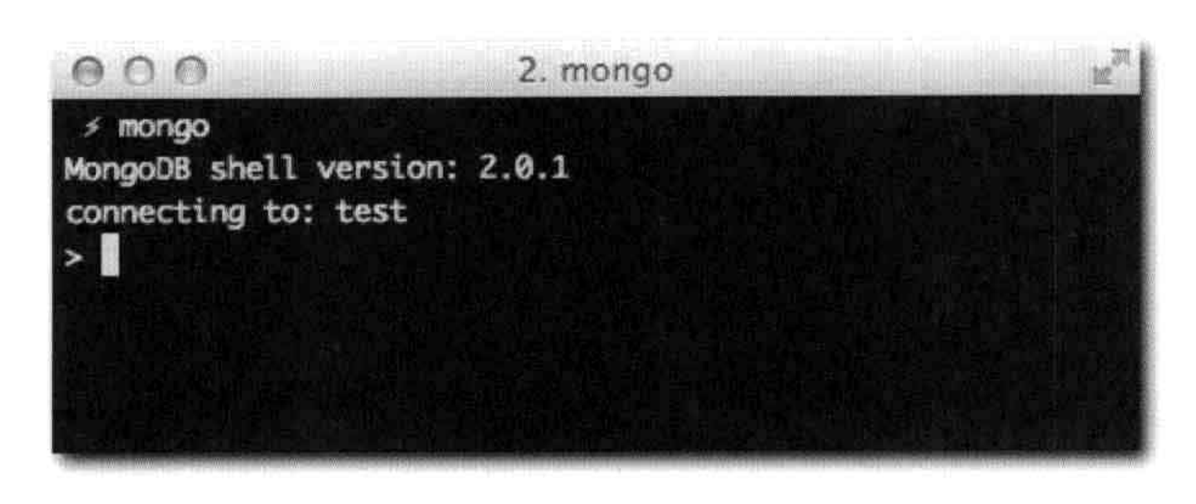


图12-1：MongoDB shell

要是无法连接到MongoDB，再检查下安装过程是否正确，同时通过进程管理器查看确保MongoDB服务器（mongod）正常运行。

使用MongoDB：一个用户认证的例子

208

通过Node.js操作MongoDB文档数据最主要的方式就是通过驱动器（driver）。通常在Node.js中驱动器指的就是一些基本的API，它懂得数据库网络层协议和其通信，并知道如何编码和解码存储的数据。

本例中选择的是由Christian Amor Kvalheim开发的node-mongodb-native。我们可以在Node包管理器（NPM）通过mongodb名字查找到这个驱动器。

第一个例子，我们创建一个简单的Express应用，实现将用户信息存储到MongoDB中，并实现用户注册、登录的功能。

构建应用程序

首先为项目创建package.json文件，声明项目所依赖的包。本例中，我们只需要express和mongodb。除此之外，我们还需要使用jade模板引擎：

```
{
    "name": "user-auth-example"
```

1 译者注：截止到本文翻译期间，MongoDB最新版本已经是2.4.6版本了。

```
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.8"
      , "mongodb": "0.9.9"
      , "jade": "0.20.3"
    }
}
```

创建Express App

首先，我们通过使用require引入所需的依赖包：

```
/**
 * 模块依赖
 */

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

由于本应用需要进行表单处理，所以，要用到bodyParser中间件。

因为我们要对用户进行认证，并将该信息保留下来，所以还需要用到 session中间件（它依赖Connect中的cookieParser中间件，在第8章中做过介绍）。

```
/**
 * 构建应用程序
 */

app = express.createServer()

/**
 * 中间件
 */

app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.session({ secret: 'my secret' }));
```

209

因为本例中选用了jade模板引擎，所以还需要设置Express的view engine配置：

```
/**
 * 指定视图选项
 */

app.set('view engine', 'jade');

// 若使用了Express 3，则不需要下面这行代码


app.set('view options', { layout: false });
```

默认情况下，视图查找路径是views/。我们需要创建该目录，并在该目录下创建一个layout.jade文件来嵌入所有其他的视图：

```
doctype 5
html
  head
    title MongoDB example
  body
    h1 My first MongoDB app
    hr
    block body
```

尽管这并不在本章范畴，但是，作为Node.js应用中最流行的模板引擎之一，学习一些关于jade的内容还是很重要的。

- jade使用的是缩进（默认两个空格，应当避免使用tab），而不是复杂的嵌套XML、HTML标签。代码如下所示：

  ```
  p
    span Hello world
  ```

  等效于 Hello world 。

- 使用jade只需输入标签名，后面紧跟内容h1 My First MongoDB app即可，不需要这样完整地书写<h1>My first MongoDB app</h1>。
  这里使用doctype 5自动插入了HTML5的doctype。 210

- 代码中还使用了特殊的关键字block，这样其他视图文件就能嵌入到这个位置。这就是为什么要称该文件为layout的原因。其他还包括if和else这样特殊的关键字。

- 属性的写法看起来就像是HTML和JavaScript代码的混合体，并且非常容易嵌入变量（或者locals，express将从controller中暴露给视图层的变量称为locals）：

  ```
  a(href=#, another=attribute, dynamic=someVariable) My link
  ```

- 可以通过#{}这样的写法来嵌入变量：

  ```
  p Welcome back, #{user.name}
  ```

接着定义路由。我们需要一个主页的路由（/）、注册页面的路由（/signup）以及登录页面的路由（/login）：

```
/**
 * 默认路由
 */

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

```
/**
 * 登录路由
 */

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

/**
 * 注册路由
 */

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

在主页的路由中（/），我们传递了一个值为false的本地变量authenticated。等实现了登录功能后，我们会动态输出该变量。

我们在index模板中需要用到authenticated变量：

index.jade

```
extends layout
block body
if (authenticated)
  p Welcome back, #{me.first}
  a(href="/logout") Logout
else
  p Welcome new visitor!
  ul
    li: a(href="/login") Login
    li: a(href="/signup") Signup
```

211

如图12-2所示，注册和登录视图就是简单的表单视图：

signup.jade

```
extends layout
block body
form(action="/signup", method="POST")
  fieldset
    legend Sign up
    p
      label First
      input(name="user[first]", type="text")
    p
      label Last
```

```
    input(name="user[last]", type="text")
  p
    label Email
    input(name="user[email]", type="text")
  p
    label Password
    input(name="user[password]", type="password")
  p
    button Submit
  p
    a(href="/") Go back
```

login.jade

```
extends layout
block body
form(action="/login", method="POST")
  fieldset
    legend Log in
    p
      label Email
      input(name="user[email]", type="text")
    p
      label Password
      input(name="user[password]", type="password")
    p
      button Submit
    p
      a(href="/") Go back
```

图12-2：/signup路由

212

最后，我们需要让应用程序监听3000端口：

```
/**
 * 监听
 */

app.listen(3000);
```

要是通过浏览器访问，就能够很容易地访问到所有定义好的路由。

连接MongoDB

在查找文档（登录功能所需）和插入文档（注册功能所需）之前，先要连接MongoDB服务器并选择正确的数据库。

我们需要在服务器监听前就连接数据库。因为应用逻辑完全依赖数据库的操作，在还未能查询数据前就接收请求显然是不行的。

由于我们直接使用了MongoDB的驱动器，所以API有些长。不过，我们的目的是要暴露像`app.users`这样MongoDB集合的API，方便在路由定义中轻松对数据库进行操作。

首先通过创建一个`mongodb.Server`并提供IP和端口来初始化服务器：

213

```
/**
 * 连接数据库
 */

var server = new mongodb.Server('127.0.0.1', 27017)
```

接着告诉驱动器去连接数据库。比如，取一个叫`my-website`的数据库名字。在MongoDB中，要是指定的名字不存在，就会创建一个数据库。

```
new mongodb.Db('my-website', server).open(function (err, client) {
```

要是连接数据库失败，我们就得终止进程：

```
// 在有错误的情况下不允许应用程序启动
if (err) throw err;
```

要是连接成功则打印成功的消息：

```
console.log('\033[96m  + \033[39m connected to mongodb');
```

接着，建立集合：

```
// 建立集合快捷方式
app.users = new mongodb.Collection(client, 'users');
```

最后，让Express服务器监听端口准备操作集合：

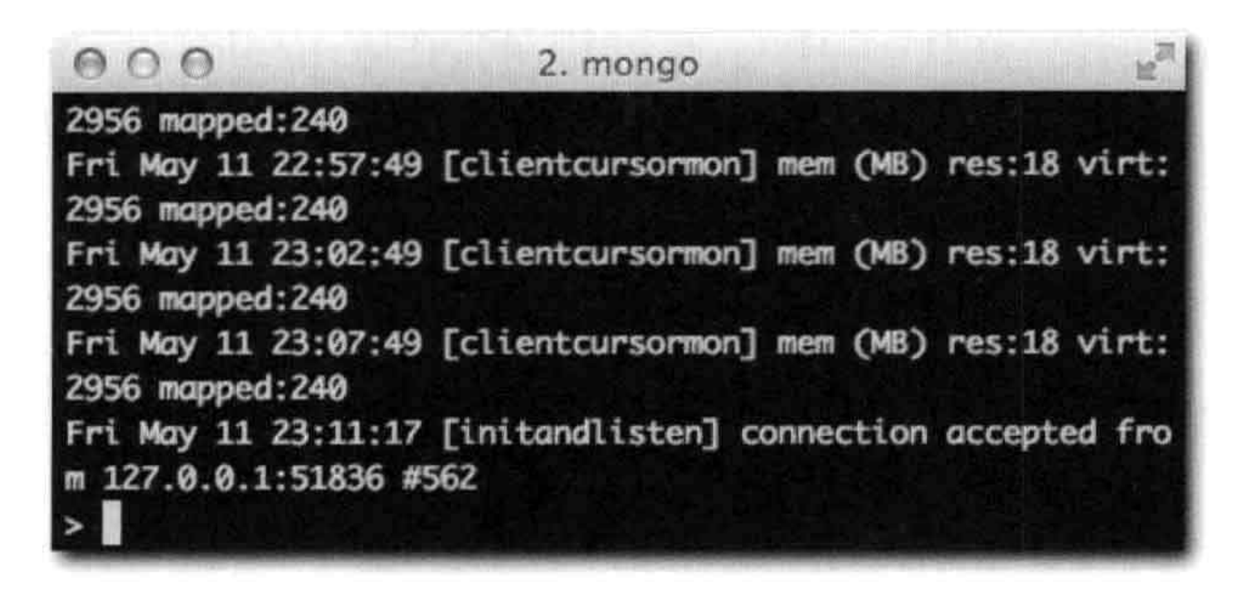
```
  // 监听
  app.listen(3000, function () {
    console.log('\033[96m  + \033[39m app listening on *:3000');
  });
});
```

如果运行应用程序（确保数据库运行正常），大致会输出如下形式的消息：

```
$ node server.js
  +  connected to mongodb
  +  app listening on *:3000
```

要是这个时候通过mongo客户端，运行show log global命令，应当会看到如图12-3所示的连接成功的消息！

```
$ mongo
> show log global;
[. . .]
{date} [initandlisten] connection accepted from 127.0.0.1:53649 #16
```



214

图12-3：如最后一行日志所示，Mongo从你的本地Web服务器获取了连接请求

创建文档

插入文档的API很简单。简单地调用Collection#insert方法，并提供要插入的文档以及一个回调函数就可以了。和绝大多数Node中的回调函数一样，第一个参数是一个错误对象，本例中，第二个参数是一个插入的文档数组：

```
collection.insert({ my: 'document' }, function (err, docs) {
  // . . .
});
```

另外还有一个options对象，是可选的第二个参数，后面会做相应介绍。

要是回头再来看注册表单，就会发现输入框的名字遵循这样的格式：user[field]。例如：

```
input(name="user[name]", type="text")
```

下面会看到，当bodyParser遇到这样的格式，会产生req.body.user.name这样的字段。

这个功能可以很方便地让我们直接将文档插入到MongoDB中。对于本例，我们忽略数据校验（非常重要）。

这样一来，处理注册的路由就变得非常简单：

```
/**
 * 处理注册的路由
 */

app.post('/signup', function (req, res, next) {
  app.users.insert(req.body.user, function (err, doc) {
    if (err) return next(err);
    res.redirect('/login/' + doc[0].email);
  });
});
```

215

若遇到错误，需要调用next，这样就可以显示一个“错误500”页面。尽管错误不会频繁发生，但重要的是必须要对其进行处理。

在处理了错误之后，最常犯的错误就是忘记return。这种错误会在应用程序中产生无法预知的行为。比如，这个时候有错误发生了，doc变量就会是undefined，这样一来代码就会抛出无法捕获的异常。

插入文档成功后，将应用程序重定向到登录路由，并提供email字段。

在登录路由中，我们获取email参数，并将其暴露给视图：

```
/**
 * 登录路由
 */

app.get('/login/:signupEmail', function (req, res) {
  res.render('login', { signupEmail: req.params.signupEmail });
});
```

在视图中，我们显示一个消息：

```
if (signupEmail)
  p Congratulations on signing up! Please login below.
```

然后，将email变量输出到email输入框中：

```
input(name="user[email]", type="text", value=signupEmail)
```

现在启动应用程序验证一下注册功能！这时，若启动mongo客户端，在新创建的集合上运行find命令，应该就能看到刚刚创建的文档内容了：

```
 Brian: in the mongo client things appear like this$ mongo my-website
> db.users.find()
{ "first" : "A", "last" : "B", "email" : "a@b.com", "password" : "d", "_id" : Object
 Id("4ef2cbd77bb50163a7000001") }
```

注意，上述文档看起来和你插入的完全一样，相当直观。除此之外， Mongo自动添加了_id字段，用来唯一确定文档内容。非常方便！

查找文档

至此我们已经创建好了文档，接下来就可以在/login路由中对文档进行查询了。我们要获取的是匹配对应email和password的文档。

在MongoDB中，没有固定的schema能确定一个集合，因此，每次对集合进行特定方式查询时，确保正确地对其进行了索引会是个不错的主意。特别是当某个键是在嵌套结构中时，要是没有对其进行索引，会导致一次扫描整表的查询操作，会导致应用程序性能的下降。 216

MongoDB有一个ensureIndex命令，顾名思义，不管索引是否存在，都可以调用这个命令来确保在查询前建立了索引。我们可以在应用初始化的时候调用它。

在设置了app.users后，我们应当调用两次ensureIndex：

```
client.ensureIndex('users', 'email', function (err) {
  if (err) throw err;
  client.ensureIndex('users', 'password', function (err) {
    if (err) throw err;

    console.log('\033[96m  + \033[39m ensured indexes');

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

重启应用后，会发现有额外日志：

```
$ node server.js
+  connected to mongodb
+  ensured indexes
+  app listening on :3000
```

现在就可以查询了！

```
/**
 * 登录处理路由
 */

app.post('/login', function (req, res) {
  app.users.findOne({ email: req.body.user.email, password: req.body.user.password
 }, function (err, doc) {
    if (err) return next(err);
    if (!doc) return res.send('<p>User not found. Go back and try again</p>');
```

```
    req.session.loggedIn = doc._id.toString();
    res.redirect('/');
  });
});
```

和insert命令一样，findOne命令可以对MongoDB进行查询文档的操作。

217 我们将_id存储为session的一部分，这样可以在用户访问其他路由时，能够获取当前登录用户的信息。注意这里，我们显式地将MongoDB ObjectId存储为字符串类型，其表现形式是十六进制的。

最后，还要实现/logout路由，处理非常简单，清除session就好了。记住，req.session对象可以随意修改，在做出响应后（比如本例中的重定向），Express会自动将其保存下来。

```
/**
 * 登出路由
 */

app.get('/logout', function (req, res) {
  req.session.loggedIn = null;
  res.redirect('/');
});
```

在这个例子中，我们保留了session，并将其ID设置成了null。或者，若想完全清楚session，可以直接调用req.session.regenerate()。

身份验证中间件

我们开发的绝大多数应用貌似在不止一处都需要访问验证后登录用户的信息。

若回过头来再看一下index.jade，我们需要访问me对象来获取匹配登录用户的文档信息，与此同时，还要通过authenticated变量来判断用户是否已经通过了身份验证。

```
if (authenticated)
  p Welcome back, #{me.name}
  a(href="/logout") Logout
```

我们可以定义一个中间件，将这两个变量（authenticated、me）暴露给视图使用。这里需要用到Express的res.local API：

```
/**
 * 身份验证中间件
 */

app.use(function (req, res, next) {
  if (req.session.loggedIn) {
    res.local('authenticated', true);
    app.users.findOne({ _id: { $oid: req.session.loggedIn } }, function (err, doc) {
      if (err) return next(err);
      res.local('me', doc);
```

```
      next();
    });
  } else {
    res.local('authenticated', false);
    next();
  }
});
```

218

注意了，在调用findOne时，我们传递了$oid修饰符。它允许直接传递一个字符串而不用传递一个ObjectId对象。回忆一下，之前我们调用了toString来确保存储的loggedIn是字符串。

记得移除此前在index路由中用来测试的{authenticated:false}，现在这部分代码应该是这样的（见图12-4）：

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

图12-4：用户成功登录后的界面

此前的应用没有考虑实际场景下必要的一些基础特性。接下来的三部分内容会对其进行介绍。

校验

若用户提交的表单太大，该怎么办呢？按照此前例子的处理方式，就会直接向数据库中插入这么大的文档数据了。

除此之外，我们也许还应该在存储数据前确保email字段确实是email字段，密码应该是不少于6个字符的字符串，而不是Date或者是Number。

我们也不想在每次对数据库进行插入、更新、查询操作的时候都要重复上述校验规则。

Mongoose通过允许在应用层定义schema来解决这个问题，它在保持文档灵活性和易改动的前提下，引入了特定的属性对其做一定的约束，称为模型。

219

原子性

假设我们基于Express和MongoDB书写一个博客引擎。可想而知，其中一部分功能会是允许用户修改博文的标题和内容，可能还有一部分功能是允许编辑和删除标签。

面向文档设计的MongoDB非常适合这样的场景。在posts集合中，文档可能会是这样的：

```
{
    "title": "I just bought Smashing Node.JS"
  , "author": "John Ward"
  , "content": "I went to the bookstore and picked up. . ."
  , "tags": ["node.js", "learning", "book"]
}
```

假设，这个时候，有两个人，用户A想要编辑文档的标题，与此同时，用户B想要添加一个标签。

如果两个用户都传递了一份完整的文档拷贝来进行更新操作，那么只有一个会胜出。另外一个将无法成功完成对文档的修改。

要确保某个操作的原子性，MongoDB提供了$set和$push这样不同的操作符：

```
db.blogposts.update({ _id: <id> }, { $set: { title: 'My title' } })
db.blogposts.update({ _id: <id> }, { tags: { $push: "new tag" })
```

Mongoose则是通过检查要对文档做的修改，并只修改受影响的字段来解决这个问题。就算操作的是数组（包括文档数组），原子性依然能够得到保证。

安全模式

此前介绍过，在使用驱动器时，我们可以在操作文档时提供一个可选的options参数：

```
app.users.insert({ }, { <options> })
```

其中一个选项叫safe，它会在对数据库进行修改时启动安全模式。

220 默认情况下，在操作完成后，如果有错误发生，MongoDB不会及时通知你。驱动器需要在操作完成后进行一个特殊的函数调用db.getLastError，来验证数据修改是否成功。

这背后的原因在于对于许多应用来说，相比于要知道某个操作是否失败而言，速度更为重要。比如，丢了某些日志并不是什么世界末日，但是导致性能低下就无法接受了。

Mongoose默认会对所有操作启用安全模式，当然了，你可以关闭这个选项。

Mongoose介绍

按照惯例，在开始使用Mongoose前，先要在package.json文件中定义对Mongoose的依赖，

然后通过require将其引入：

```
var mongoose = require('mongoose')
```

相比原生的驱动器，Mongoose做的第一个简化的事情就是它假定绝大部分的应用程序都是用一个数据库，这大大简化了使用方式。要连接数据库，只需要调用mongoose.connect并提供mongodb://URI即可：

```
mongoose.connect('mongodb://localhost/my_database');
```

另外，使用Mongoose，就无须关心连接是否真的已经建立了，因为，它会先把数据库操作指令缓存起来，在连接上数据库后就会把这些操作发送给MongoDB。这就意味着，我们无须监听connection的回调函数。连接后就可以直接像下面要介绍的这样开始查询数据了。

定义模型

模型是Schema类的简单实例。在指定字段时，简单地使用对应类型的JavaScript原生的构造器即可：

```
var Schema = mongoose.Schema
  , ObjectId = Schema.ObjectId;

var PostSchema = new Schema({
    author     : ObjectId
  , title      : String
  , body       : String
  , date       : Date
});
```

这些类型是：

- Date
- String
- Number
- Array
- Object

除此之外，MongoDB还提供了一种ObjectId类型，这种类型可以通过Schema.ObjectId来获得。

在这个博客例子中，我们可以将创建博文的用户存储为ObjectId类型。

Mongoose还为给定的字段提供了一些不同的选项。在提供选项时，需要在一个对象中引用对应字段类型的构造器。比如，要给字段提供了一个默认值，可以按照如下方式提供

default和type选项：

```
var PostSchema = new Schema({
    author     : ObjectId
  , title      : { type: String, default: 'Untitled' }
  , body       : String
  , date       : Date
});
```

创建好Schema后，通过mongoose来注册一个模型：

```
var Post = mongoose.model('BlogPost', PostSchema);
```

对于本例，Mongoose将集合名字设置为blogposts，除非我们通过第三个参数来指定集合名。Mongoose默认会对模型名字使用小写复数形式。

随后要想获取模型，可以通过调用mongoose.model方法并提供模型名：

```
var Post = mongoose.model('BlogPost');
```

接着就可以操作模型了。要创建一个博客文章，只要使用new操作符就可以了：

```
new Post({ title: 'My title' }).save(function (err) {
  console.log('that was easy!');
});
```

有一点很重要：Schema只是一种简单的抽象，用以描述模型的样子以及它是如何工作的。数据的交互发生在模型上，而不是Schema上。

因此，若要查询，与使用new关键字来初始化博文相对，需要执行静态的Post.find方法（或者其他一些会在下面介绍的方法）。

222 定义嵌套的键

考虑到数据的组织，有的时候以子结构的形式来组织键也非常有帮助：

```
var BlogPost = new Schema({
    author     : ObjectId
  , title      : String
  , body       : String
  , meta       : {
      votes : Number
    , favs  : Number
  }
});
```

在MongoDB中，可以使用点来操作这些属性。比如，要查找拥有指定投票数的博文，可以通过如下方式来实现：

```
db.blogposts.find({ 'meta.votes': 5 })
```

定义嵌套文档

在MongoDB中，文档可以很大，也可以层次很深。也就是说，如果博文有留言的话，可以直接将留言定义在博文中，而不需要将其定义为单独的集合。

```
var Comments = new Schema({
    title     : String
  , body      : String
  , date      : Date
});

var BlogPost = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , buf       : Buffer
  , date      : Date
  , comments  : [Comments]
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

Mongoose还允许为该字段定义想要的类型。

构建索引

正如此前提到过的，索引是在MongoDB数据库中确保快速查询的重要因素。

要对指定的键做索引，需要传递一个index选项，并将值设置为true。 223

比如，要对title键做索引，并将uid键设置为唯一，可以这样：

```
var BlogPost = new Schema({
    author    : ObjectId
  , title     : { type: String, index: true }
  , uid       : { type: Number, unique: true }
});
```

要设置更复杂的索引（如组合索引），可以使用静态的index方法：

```
BlogPost.index({ key: -1, otherKey: 1 });
```

中间件

在相当一部分应用中，有的时候会在不同的地方以不同的方式对同样的数据进行修改。

通过模型接口将这部分对数据库的交互集中处理是避免代码重复很有效的方式。

Mongoose通过引入中间件来实现。Mongoose中间件的工作方式和 Express中间件非常相似。你可以定义一些方法，在某些特定动作前执行：save和remove。

比如，要在博文删除时，发送电子邮件给作者，可以通过下面的方式来实现：

```
Blogpost.pre('remove', function (next) {
  emailAuthor(this.email, 'Blog post removed!);
  next();
});
```

还可以对单个动作定义多次中间件来执行各类操作，特别是异步操作。

探测模型状态

很多时候，我们需要根据要对当前模型做的不同更改进行不同的操作：

```
Blogpost.pre('save', function (next) {
  if (this.isNew) {
    // doSomething
  } else {
    // doSomethingElse
  }
});
```

224

还可以通过this.dirtyPaths来探测什么键被修改了。

查询

在Model实例上暴露的所有常见的操作有：

- find
- findOne
- remove
- update
- count

Mongoose还添加了findById，该方法接受一个ObjectId去匹配文档的_id属性。

扩展查询

如果对某个查询不提供回调函数，那么直到调用run它才会执行：

```
Post.find({ author: '4ef2cbffb1d9807fa7000001' })
  .where('title', 'My title')
  .sort('content', -1)
  .limit(5)
  .run(function(err, post) {
    // . . .
  })
```

排序

要进行排序，只需提供排序的键以及排序的顺序即可：

```
query.sort('key', 1)
query.sort('some.key', -1)
```

选择

若文档很大，而想要的只是部分指定的键，那么就可以调用Query#select方法。

比如，要显示带链接的博文列表，无须获取所有的字段（有些字段可能数据量很大）：

```
Post.find()
  .select('field', 'field2')
```

限制

225

要限制查询结果的数量，可以调用Query#limit方法：

```
query.limit(5)
```

跳过

要跳过指定数量的文档数据，可以通过如下方式：

```
query.skip(10);
```

这个功能结合Model#count对做分页非常有用：

```
Post.count(function (err, totalPosts) {
  var numPages = Math.ceil(totalPosts / 10);
});
```

自动产生键

在BlogPost模型例子中，我们将博文作者的ID存储为author属性。

很多时候，在查询一个博文时，我们还需要获取对应的作者。这个时候，就可以为ObjectId类型提供一个ref属性：

```
var BlogPost = new Schema({
    author    : { type: ObjectId, ref: 'Author' }
  , title     : String
  , body      : String
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

之后，查询文档时就能自动产生作者数据！通过简单地对指定键调用populate方法即可：

```
BlogPost.find({ title: 'My title' })
  .populate('author')
  .run(function (err, doc) {
    console.log(doc.author.email);
  })
```

转换

因为Mongoose提前知道需要什么样的数据类型，所以它总是会尝试去做类型转换。

226

例如，有一个年龄字段，在Schema中描述的是Number类型。如果有人在网站中提交了一个普通表单，那么在没有JSON或者特定逻辑处理的情况下，我们得到的是字符串而不是数字。Mongoose利用了动态语言的特性，所以对它来说在存储前将'21'（字符串）转化为21（数字）就没有什么问题了。

同样的情况还会发生在ObjectId。在此前的例子中，我们不得不使用$oid修饰符来让使用字符串形式ObjectID的查询操作成功执行，然而，这样的方式过于冗长。我们可以直接传递"4ef2cbd77bb50163a7000001"给Mongoose，它会自动将其转化为ObjectId("4ef2cbd77bb50163a7000001")。

除此之外，当类型不匹配并且转换失败时，Mongoose会抛出一个校验错误，并且放弃对文档的存储。这种行为在一致性和文档的整洁性上确保了易用性。

一个使用Mongoose的例子

和此前我们做的一样：先使用node http模块，然后再使用Connect和 Express来对其改进。接下来，我们要用Mongoose对此前的例子进行重构，以此来展现Mongoose在表现形式上对数据表提供的好处。

我们从创建一个新的package.json并添加对mongoose的依赖开始。

构建应用

新的package.json文件如下所示：

```
{
    "name": "mongoose-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "mongoose": "2.5.10"
    }
}
```

和往常一样，运行npm install来安装所需的依赖。接下来，我们重构一下服务器端主要的代码。

重构

我们从此前例子中将server.js文件和views目录复制过来。

首先要重构的就是将对mongodb的依赖替换为mongoose，因为mongodb并未出现在package.json文件中。事实上，mongoose内部就是使用了mongodb。

server.js顶部代码如下所示：

227

```
/**
 * 模块依赖
 */

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

现在，我们要将注意力放在该文件底部代码上，也就是连接数据库的代码：

```
/**
 * 连接数据库
 */

var server = new mongodb.Server('127.0.0.1', 27017)
// . . .
```

正如此前提到的，mongoose大大简化了连接数据库、操作集合、建立索引等的操作。server.js文件最后一部分代码就可以简化为如下形式：

```
/**
 * 连接数据库
 */

mongoose.connect('mongodb://127.0.0.1/my-website');
app.listen(3000, function () {
  console.log('\033[96m  + \033[39m app listening on *:3000');
});
```

接下来，我们要定义模型来取代app.users这种引用方式，并建立此前建立的索引。

建立模型

模型可以在文件任意位置通过Mongoose定义。无所谓Mongoose是否已经与数据库连接。

在文件最后，我们加上对模型的定义：

```
/**
 * 定义模型
```

```
 */

var Schema = mongoose.Schema

var User = mongoose.model('User', new Schema({
    first: String
  , last: String
  , email: { type: String, unique: true }
  , password: { type: String, index: true }
}));
```

228

接下来我们再看用它来替换app.users的地方。第一处就是身份认证中间件。我们用Mongoose提供的便利的findById方法来替换使用$oid的地方：

```
app.use(function (req, res, next) {
  if (req.session.loggedIn) {
    res.local('authenticated', true);
    User.findById(req.session.loggedIn, function (err, doc) {
      if (err) return next(err);
      res.local('me', doc);
      next();
    });
  } else {
    res.local('authenticated', false);
    next();
  }
});
```

在登录的POST路由中，需要再次用到模型方法。这次是findOne：

```
app.post('/login', function (req, res) {
  User.findOne({ email: req.body.user.email, password: req.body.user.password },
function (err, doc) {
    if (err) return next(err);
    if (!doc) return res.send('<p>User not found. Go back and try again');
    req.session.loggedIn = doc._id.toString();
    res.redirect('/');
  });
});
```

正如此前提到过的，模型可以静态方式使用（正如那两个例子所示的），也可以当构造器使用。

注册的POST路由应该重构成如下形式：

```
app.post('/signup', function (req, res, next) {
  var user = new User(req.body.user).save(function (err) {
    if (err) return next(err);
    res.redirect('/login/' + user.email);
  });
});
```

注意，我们不再需要一个包含了文档信息的回调函数。我们只需在回调函数中使用创建的模型实例即可（使用user.email而不是doc[0].email）。 229

重构完成！再次运行server.js，一切都应该运行正常，不过，代码变得更为整洁，更容易理解了。

小结

本章介绍了Node.js世界中最流行的数据库之一：MongoDB。

本章介绍了关于文档数据库是如何工作的基本知识，还介绍了Node.js中 MongoDB的驱动器。

你会注意到，它处理数据的方式非常自然，另外，对数据在浏览器和Web服务器之间传递的映射也非常好。

通过重构第一个例子，现在，你应当能够体会到本章介绍的引入了模型概念以及非常方便的API框架——Mongoose的好处了。

CHAPTER

13 MySQL

尽管现如今NoSQL渐行其道，越来越流行，但是SQL数据库依旧是绝大多数应用的选择。 231

Node.js丰富的生态系统中，有许多模块都是为SQL数据库设计开发的，特别是本章要介绍的：MySQL。

与第12章中介绍MongoDB一样，本章会先介绍一个原生的MySQL驱动器（一个名为node-mysql的项目）。

通过node-mysql，我们可以书写自己的SQL查询语句来操作数据库。

除了驱动器之外，本章还会介绍如何使用MySQL的对象关系映射器（ORM）——node-sequelize。正如上一章介绍的，ORM提供了一个MySQL数据库中数据到JavaScript模型对象的映射，使得操作数据关系、数据处理等变得更加容易。

node-mysql

232

要学习如何使用node-mysql，我们从创建一个简单的购物车应用的模型开始。

初始化项目

按照惯例，我们从添加对`express`、`jade`还有`node-mysql`的依赖开始：

package.json

```
{
    "name": "shopping-cart-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "jade": "0.19.0"
      , "mysql": "0.9.5"
    }
}
```

Express应用

接下来，我们创建一个简单的Express应用，并添加如下路由：

- /：展示所有的商品以及创建商品的表单。
- /item/<id>：展示指定的商品以及用户评价。
- /item//review (POST)：创建一个评价。
- /item/create (POST)：创建一个商品。

对于首页以及商品的路由，我们将渲染简单的模板。注意了，我们配置了Express的view options，将模板布局取消了，这符合Express 3的行为。模板布局将直接通过jade来实现。

server.js

```
/**
 * 模块依赖
 */

var express = require('express')

/**
 * 创建应用
 */
app = express.createServer();

/**
 * 配置应用
 */

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

```
/**
 * 首页路由
 */

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

/**
 * 创建商品路由
 */

app.post('/create', function (req, res, next) {
});

/**
 * 查看商品路由
 */

app.get('/item/:id', function (req, res, next) {
  res.render('item');
});

/**
 * 创建商品评价路由
 */

app.post('/item/:id/review', function (req, res, next) {
});

/**
 * 监听
 */

app.listen(3000, function () {
  console.log(' - listening on http://*:3000');
});
```

连接MySQL

234

下一步是添加对node-mysql的依赖：

server.js

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

要初始化连接，和创建net客户端所使用的Node API方式一样，我们通过调用createClient

方法来实现。

和Mongoose对mongodb的处理方式一样，node-mysql在真正连接到MySQL前就可以接收指令，并将它们缓存起来（也就是将它们存储在内存中），当连接建立后，就一次性将它们全部发送给MySQL。

所以，我们无须监听connection事件或者提供回调函数，只需简单地初始化客户端，提供相应的设置即可。将下列代码添加到应用配置部分的下方：

server.js

```
/**
 * 连接MySQL
 */

var db = mysql.createClient({
    host: 'localhost'
  , database: 'cart-example'
});
```

若要设置数据库的用户名和密码，就在调用`createClient`方法时在参数中传递`user`和`password`选项。要了解更多关于node-mysql的用法，可以查看其官方文档http://github.com/felixge/node-mysql。

初始化脚本

在应用程序中使用SQL数据库前，我们总是先要创建必要的数据库和表。

为了让代码重用，我们创建一个名为`setup.js`的简单node脚本来运行必要的`CREATE TABLE`命令。

由于连接数据库所需参数和此前书写在应用中的是一样的，所以我们先将这些参数的配置抽象到一个`config.json`文件中：

235

config.json

```
{
    "host": "localhost"
  , "database": "cart-example"
}
```

注意了，有效的JavaScript代码未必一定是有效的JSON。本例中，我们将所有的键都用引号括起来，并且要确保所有的值都使用的是双引号而不是单引号。

从Node 0.6开始，就可以直接使用`require`来引入JSON文件，而无须再用JSON.parse和

fs#readFileSync了。接下来，编辑修改依赖的模块：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , mysql = require('mysql')
  , config = require('./config')

// . . .
```

在连接数据库的代码部分，使用config来替换原先的参数对象：

server.js

```
var db = mysql.createClient(config);
```

接下来准备创建启动脚本。该脚本只依赖mysql和config，因为它是从命令行直接运行的。

setup.js

```
/**
 * 模块依赖
 */

var mysql = require('mysql')
  , config = require('./config')
```

下面初始化客户端，由于数据库还未创建好，所以需要将config中的database字段删除：

236

setup.js

```
/**
 * 初始化客户端
 */
delete config.database;
var db = mysql.createClient(config);
```

node-mysql提供的执行查询语句的API非常简单：client.query(<sql>, <callback>)。关闭连接的API是client.end。

由于我们使用单个TCP连接，所以服务器接收到的指令顺序和我们书写的顺序是一致的。也就意味着，不需要为了确保执行顺序而嵌套回调函数：

```
// 没有必要这么做!
db.query('CREATE TABLE. . .', function (err) {
  db.query('CREATE TABLE. . .', function (err) {
    db.query('CREATE TABLE. . .', function (err) { });
  });
});
```

为了确保能够将错误报告给用户，还需要监听db error事件：

```
db.on('error', function () {
  // handle error
});
```

对于中断程序而言，当错误发生时，最好的处理方式是将错误和调用堆都展示给用户并终止程序。你或许还记得第4章中介绍的，当错误对象通过EventEmitter分发出来，但又没有对应的监听器时（也就是说事件未处理），Node会将错误抛出，确保开发者能够意识到错误的发生，而不是将错误“吞”掉。因此，事实上，对于本例，我们无须专门添加一个错误处理器，因为Node会将未处理的错误直接抛出。

首先，我们需要创建数据库并告诉MySQL要使用该数据库：

setup.js

```
/**
 * 创建数据库
 */

db.query('CREATE DATABASE IF NOT EXISTS `cart-example`');
 db.query('USE `cart-example`');
```

237

setup.js

```
/**
 * 创建表
 */

db.query('DROP TABLE IF EXISTS item');
 db.query('CREATE TABLE item (' +
  'id INT(11) AUTO_INCREMENT,' +
  'title VARCHAR(255),' +
  'description TEXT,' +
  'created DATETIME,' +
  'PRIMARY KEY (id))');
db.query('DROP TABLE IF EXISTS review');
db.query('CREATE TABLE review (' +
  'id INT(11) AUTO_INCREMENT,' +
  'item_id INT(11),' +
  'text TEXT,' +
```

```
  'stars INT(1),' +
  'created DATETIME,' +
  'PRIMARY KEY (id))');
```

setup.js

```
/**
 * 关闭客户端
 */

db.end();
```

正如我们在第3章中介绍的，当事件轮询中没有任务要处理时，Node就会退出该进程。在连接MySQL服务器的过程中，我们打开了一个文件描述符，于是Node事件轮询机制就会开始监听。当调用结束客户端时，文件描述符也会被关闭，因此，也要结束程序。

代码如下所示：

setup.js

```
db.end(function () {
  process.exit();
});
```

接下来测试刚刚书写的脚本：

```
$ node setup.js
```

然后，我们可以用mysql客户端确认下，数据库和表都已经创建好了（见图13-1）：

238

```
1. mysql
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| cart-example       |
| mysql              |
| performance_schema |
| test               |
| todo-example       |
+--------------------+
6 rows in set (0.00 sec)

mysql> USE `cart-example`;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+------------------------+
| Tables_in_cart-example |
+------------------------+
| item                   |
| review                 |
+------------------------+
```

图13-1：通过MySQL命令行客户端来查看利用setup.js创建的数据库和表是否创建成功了

```
$ mysql
> show databases;
. . .
> use cart-examples;
. . .
> SHOW TABLES;
. . .
```

创建数据

接下来，我们在views目录下创建一个简单的布局。如下所示，该文件包含一个特殊的jade `block body`声明，用于将其他视图嵌入：

views/layout.jade

```
doctype 5
html
  head
    title My shopping cart
  body
    h1 My shopping cart
    #cart

block body
```

index文件展示了一个包含所有商品的列表，以及用于创建新商品的表单：

239

views/index.jade

```
extends layout
block body
h2 All items
if (items.length)
  ul
    each item in items
      li
        h3: a(href="/item/#{item.id}")= item.title
        = item.description
else
  p No items to show

h2 Create new item

form(action="/create", method="POST")
  p
    label Title
    input(type="text", name="title")
  p
    label Description
```

```
    textarea(name="description")
  p
    button Submit
```

因为在上述代码中，通过检查length属性来展示items数组项，所以，我们暂且先确保在/路由中传递一个空数组，如下所示。当然了，真正的数据稍后肯定是从数据库中获得的。

server.js

```
app.get('/', function (req, res, next) {
  res.render('index', { items: [] });
});. . .
```

对于商品查看页面，我们需要商品本身、关于它的评价以及创建新评价的表单。

views/item.jade

```
extends layout

block body
  a(href="/") Go back

  h2= item.title
  p= item.description
  h3 User reviews

  if (reviews.length)
    each review in reviews
      .review
        b #{review.stars} stars
        p= review.text
      hr
  else
    p No reviews to show. Write one!

  form(action="/item/#{item.id}/review", method="POST")
    fieldset
      legend Create review
      p
        label Stars
        select(name="stars")
          option 1
          option 2
          option 3
          option 4
          option 5
      p
        label Review
```

240

```
    textarea(name="text")
  p
    button(type="submit") Send
```

注意在上述代码中，表单的action属性部分使用了jade的插补特性。通过使用#{}以一种安全的方式（HTML的实体会被转义）来引入变量。如果想要引入不需要转义的字符串变量，可以使用!{}。

在开始获取数据前，我们需要先创建数据来做简单的测试。

在项目配置代码下方，添加bodyParser中间件来处理POST请求：

server.js

```
/**
 * 中间件
 */

app.use(express.bodyParser());
```

241

接着，完成/create路由：

server.js

```
/**
 * 创建商品路由
 */

app.post('/create', function (req, res, next) {
  db.query('INSERT INTO item SET title = ?, description = ?',
  [req.body.title, req.body.description], function (err, info) {
    if (err) return next(err);
    console.log(' - item created with id %s', info.insertId);
    res.redirect('/');
  });
});
```

上述代码有两部分非常意思。第一部分是db.query允许用后面提供的数据替换?记号。通过这种替换记号的方式，可以有效地避免字符串的拼接，从而避免SQL注入攻击。如果在查询语句中包含了?记号，那么需要提供一个包含要替换数据的数组作为第二个参数。

另外一部分有意思的是info对象。本例中，我们通过insertId来获取创建商品的ID。只要不发生错误，这个属性一直都在。如果有错误发生，我们就终止处理，并调用next方法。

创建评价的路由也类似：

```
/**
 * 创建商品评价路由
 */

app.post('/item/:id/review', function (req, res, next) {
  db.query('INSERT INTO review SET item_id = ?, stars = ?, text = ?',
  [req.params.id, req.body.stars, req.body.text], function (err, info) {
    if (err) return next(err);
    console.log(' - review created with id %s', info.insertId);
    res.redirect('/item/' + req.params.id);
  });
});
```

通过运行上述应用并创建一个商品来做测试（见图13-2）。

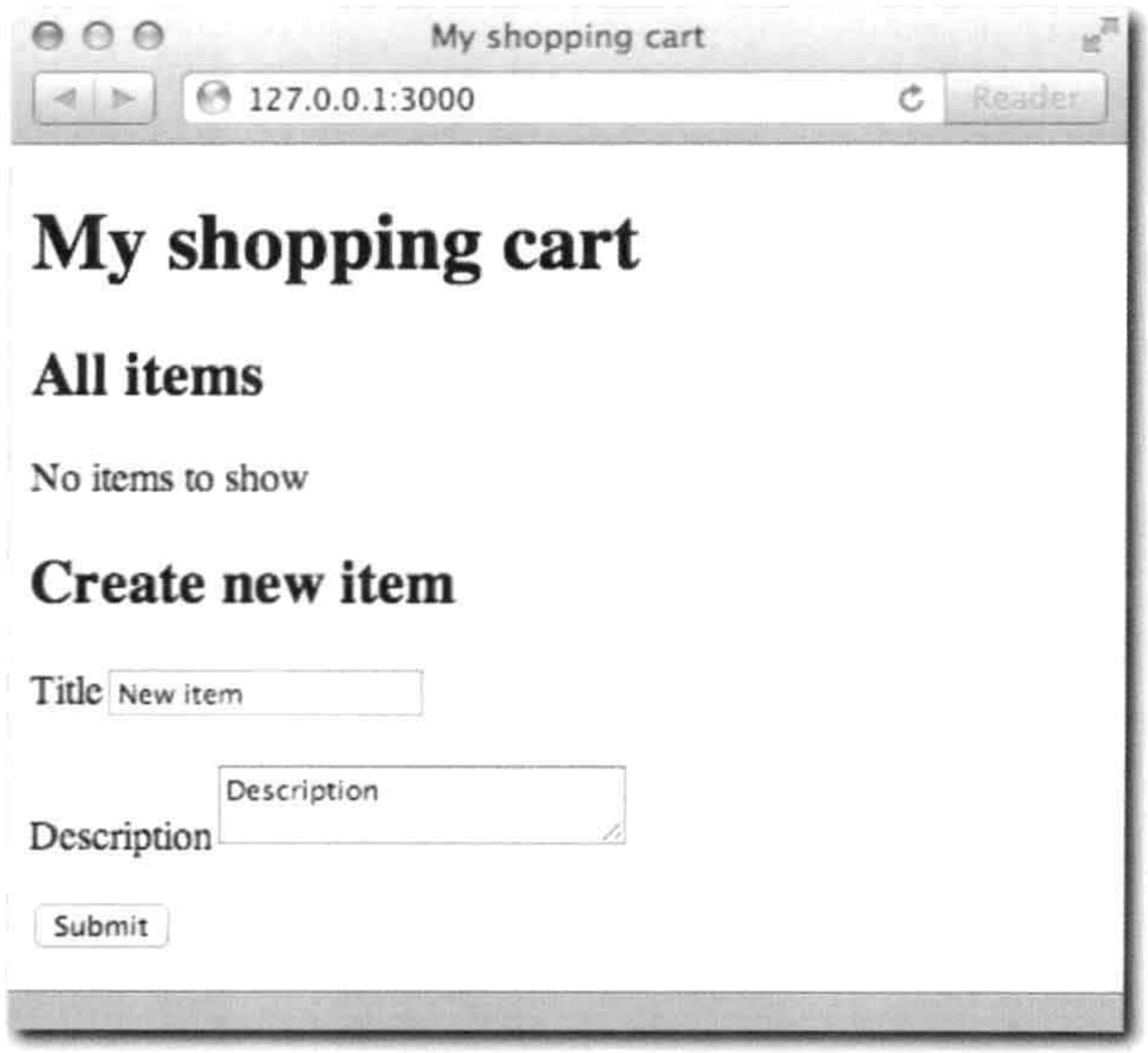


图13-2：在首页路由中，填写创建商品的表单

之后，就能在控制台看到如图13-3所示的内容了。

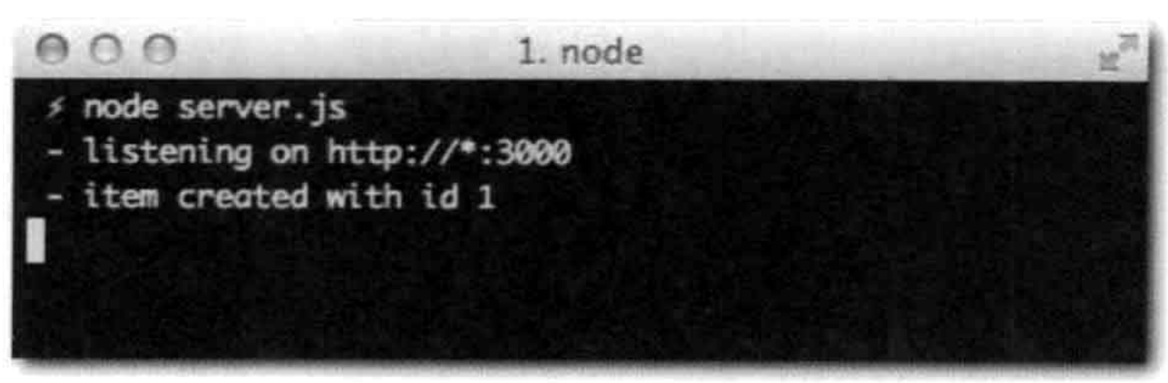


图13-3：商品创建成功后，在控制台显示了其id

获取数据

通过node-mysql从MySQL中获取数据是非常简单直观的。当执行的命令是SELECT时，回调函数中会接收一个包含查询结果对象的数组。数组中的对象包含了指定返回的字段。根据本章所属范畴，我们将只讨论SELECT指令。

回调函数中接收到的是数组，这其实和我们在index.jade模板中想要的数据结构是一致的，模板中，我们需要遍历数组，并显示其id、title以及description属性。

因此，我们要做的就是检查是否有错误发生，如果没有，就把查询结果传递给视图：

```
/**
 * 首页路由
 */
243 app.get('/', function (req, res, next) {
  db.query('SELECT id, title, description FROM item', function (err, results) {
res.render('index', { items: results });
  });
});
```

/路由完成后，就可以重启应用，查看所有商品的列表了。

对于列表中包含的查看商品路由，我们需要获取商品数据，确保它存在，并获取相关的评价。如果商品不存在，就返回404状态码。

为了让代码更可读，我们将逻辑按照执行顺序拆分到几个函数中：

server.js

```
/**
 * 查看商品路由
 */

app.get('/item/:id', function (req, res, next) {
  function getItem (fn) {
    db.query('SELECT id, title, description FROM item WHERE id = ? LIMIT 1',
    [req.params.id], function (err, results) {
      if (err) return next(err);
      if (!results[0]) return res.send(404);
      fn(results[0]);
    });
  }

  function getReviews (item_id, fn) {
    db.query('SELECT text, stars FROM review WHERE item_id = ?',
    [item_id], function (err, results) {
      if (err) return next(err);
      fn(results);
    });
```

```
  }

  getItem(function (item) {
    getReviews(item.id, function (reviews) {
      res.render('item', { item: item, reviews: reviews });
    });
  });
});
```

图13-4展示了完成后的应用。现在可以浏览商品、提交评价以及在界面上看到它们的信息了。

244

图13-4：完整应用展示

sequelize

在此前的例子中，直接操作SQL数据库的方式多少有些问题。

第一个问题就是建表的过程是手动的（耗时），而且表的定义并非项目本身的一部分。应用程序根本无法得知商品的title属性只能允许最多255个字符。如果能够知道，那么应用程序就可以对用户的输入做校验，并显示错误信息。

解决这个问题的方式就是使用sequelize：通过它可以定义schema和模型，同时还可以使用其同步的特性，根据那些定义来创建要使用的数据库表。这样一来，此前例子中的`setup.js`就不再需要了。

因为schema也是应用程序的一部分，我们可以使用它们来做类型转化。要存储包含指定数据的商品，我们可以直接传递一个JavaScript的Date对象，而无须手动构造MySQL需要的日期格式。

最后一点，同时也是很重要的一点就是联合。在此前的例子中，我们是手动去获取商品的评价信息的，而通过sequelize，可以自动获取这部分数据。

245

我们通过创建一个简单的任务列表应用，来运用sequelize为数据库表带来的不同的概念和特性。任务可以根据项目来分组。我们可以添加、创建和删除项目，同时也可以在指定项目中，添加、创建和删除任务。

初始化sequelize

由于sequelize内部使用了node-mysql驱动器，所以依赖列表就可以是如下形式：

package.json

```
{
    "name": "todo-list-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "jade": "0.19.0"
      , "sequelize": "1.3.7"
    }
}
```

初始化Express应用

这部分应用将与以往传统的应用有所不同，这次在创建和删除商品时将采用Ajax的方式。我们可以通过使用DELETE方法让应用程序更加RESTful。若你对REST还不熟悉，那么简单来说它就是一系列准则，引入了一种HTTP协议更宽泛的使用方式，使得像HTTP的PATCH、DELETE以及平时不常使用的状态码为我们所用。

定义的路由如下所示：

- / (GET)：获取所有项目。
- /projects (POST)：创建项目。
- /project/:id (DELETE)：删除项目。
- /project/:id/tasks (GET)：获取任务。
- /project/:id/tasks (POST)：添加任务。
- /task/:id (DELETE)：删除任务。

```
/**
 * 模块依赖
 */

var express = require('express')

/**
 * 创建应用
 */

app = express.createServer();

/**
 * 配置应用
 */

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

/**
 * 首页路由
 */

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

/**
 * 删除项目路由
 */

app.del('/project/:id', function (req, res, next) {
});

/**
 * 创建项目路由
 */

app.post('/projects', function (req, res, next) {
});

/**
 * 展示指定项目中的任务
 */

app.get('/project/:id/tasks, function (req, res, next) {
});

/**
```

246

```
 * 为指定项目添加任务
 */

app.post('/project/:id/tasks, function (req, res, next) {
});

/**
 * 删除任务路由
 */

app.del('/task/:id', function (req, res, next) {
});
/**
 * 监听
 */

app.listen(3000, function () {
  console.log(' - listening on http://*:3000');
});
```

247

我们再定义一个简单的布局，这次使用jQuery来更容易地发送Ajax请求：

views/layout.jade

```
doctype 5
html
  head
    title TODO list app
    script(src="http://code.jquery.com/jquery-1.7.2.js")
    script(src="/js/main.js")
  body
    h1 TODO list app
    #todo
      block body
```

注意了，在上述代码中，载入了main.js文件，该文件将包含所有的客户端逻辑（如：发送Ajax请求来提交表单）。

项目和任务列表展示方式一样，因为它们都有添加和删除操作：

views/index.jade

```
extends layout

block body
  h2 Projects

  #list
    ul#projects-list
```

```
    each project in projects
      li
        a(href="/project/#{project.id}/items")= project.title
        a.delete(href="/project/#{project.id}") x

  form#add(action="/projects", method="POST")
    input(type="text", name="title")
```

248

button(type="submit") Addviews/tasks.jade

```
h2 Tasks for project #{project.title}

#list
  ul#tasks-list
    each task in tasks
      li
        span= task.title
        a.delete(href="/task/#{task.id} ") x

  form#add(action="/project/#{project.id}/tasks", method="POST")
    input(type="text", name="title")
    button Add
```

连接sequelize

现在，我们需要将sequelize添加到模块依赖中：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , Sequelize = require('sequelize')
```

接下来初始化主类。我们可以直接在模块依赖后做这个初始化工作，也可以出于对程序结构清晰的考虑，在应用设置后做。

server.js

```
/**
 * 初始化 sequelize
 */

var sequelize = new Sequelize('todo-example', 'root')
```

Sequelize构造器接收如下参数：

- database (String)
- username (String) - 必要
- password (String) - 可选
- other options (Object) - 可选
  - host (String)
  - port (Number)

249

可以使用如下命令行来创建数据库。记住将root替换为在Sequelize构造器中使用的MySQL用户名：

```
$ mysqldmin -u root -p create todo-exmaple
```

定义模型和同步

要定义模型，需要调用sequelize.define方法。我们可以在引入sequelize后直接调用该方法。该方法第一个参数是模型名，第二个参数是包含了属性的对象。

server.js

```
var Project = sequelize.define('Project', {
    title: Sequelize.STRING
  , description: Sequelize.TEXT
  , created: Sequelize.DATE
});
```

在上述代码中，属性的类型对应如下sequelize中的类型。接下来，每种类型都对应MySQL中的类型：

- Sequelize.STRING // VARCHAR(255)
- Sequelize.BOOLEAN // TINYINT(1)
- Sequelize.TEXT // TEXT
- Sequelize.DATE // DATETIME
- Sequelize.INTEGER // INT

除了传递类型之外，还可以传递一个包含选项的对象，比如，要设置默认值，就可以传递：

```
title: { type: Sequelize.STRING, defaultValue: 'No title' }
```

接下来定义任务模型：

server.js

```
/**
 * 定义任务模型
 */

var Task = sequelize.define('Task', {
    title: Sequelize.STRING
});
```

最后，设置一个hasMany的联合：

250

server.js

```
/**
 * 设置联合
 */

Task.belongsTo(Project);
 Project.hasMany(Task);
```

为了设置联合，Sequelize会处理构建响应的列、主键以及索引的任务。

belongsTo联合意味着每个Task都有一个指向它所属项目的字段。另外，每个任务模型都会有一个名为getProject的方法来获取其所属的项目。

对于hasMany而言，调用find查询到项目后，它们都会有一个名为getTasks的方法来获取项目中的任务。

除此之外，sequelize还支持另一种关系：hasOne。不过本例中用不到这种联合，它与belongsTo是相对的。

最后，我们要确保schema都同步到数据库中了，并且不需要手动运行CREATE TABLE命令：

```
/**
 * 同步
 */

sequelize.sync();
```

在开发阶段，我们会经常对数据库表做修改操作。因此，我们可以在调用sync方法时，传递一个{force: true}参数来让sequelize始终先删除已有的表，再重新创建，以确保数据变化总能同步到数据库中。

server.js

```
sequelize.sync();
```

创建数据

对于项目和任务列表，我们都需要在表单提交时，绑定一个jQuery的监听器。

当发送Ajax调用时，我们希望返回的是JSON格式的模型实例数据。

251

获取数据后，我们就将其添加到DOM中。

在这之前，我们需要添加一个static中间件来托管public/js目录（也需要创建好）。因为还需要使用jQuery来POST数据，所以还需要bodyParser中间件。

server.js

```
/**
 * 中间件
 */

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

public/js/main.js

```
$(function () {
  $('form').submit(function (ev) {
    ev.preventDefault();
    var form = $(this);
    $.ajax({
        url: form.attr('action')
      , type: 'POST'
      , data: form.serialize()
      , success: function (obj) {
          var el = $('<li>');
          if ($('#projects-list').length) {
            el
              .append($('<a>').attr('href', '/project/' + obj.id
+ '/tasks').text(obj.title + ' '))
              .append($('<a>').attr('href', '/project/' + obj.id)
.attr('class', 'delete').text('x'));
          } else {
            el
              .append($('<span>').text(obj.title + ' '))
              .append($('<a>').attr('href', '/task/' + obj.id)
.attr('class', 'delete').text('x'));
          }
          $('ul').append(el);
        }
    });
form.find('input').val(''); // clear the input
  });
});
```

代码很简单。捕获到网站中所有的表单提交，并利用Ajax的方式来提交：

1. 捕获表单提交。
2. 通过调用preventDefault方法来阻止默认行为。也就是说，阻止浏览器试图自动POST表单，因为我们想要用Ajax的方式来提交。 252
3. 调用jQuery的$.ajax方法来提交一个POST请求，同时将表单数据序列化成查询字符串发送过去（通过form.serialize序列化数据，并将其作为data属性值）。

JSON数据返回后，我们重新构造该数据项，并将其添加到项目列表或者任务列表中。如果该数据项是项目，那么我们将其链接添加到任务列表中，同时再添加一个删除链接。若是任务，则简单地添加一个span和删除链接。

现在我们来实现Express应用中的.post路由。这里我们使用模型上的.build方法：

server.js

```
/**
 * 创建项目路由
 */

app.post('/projects', function (req, res, next) {
  Project.build(req.body).save()
    .success(function (obj) {
       res.send(obj);
     })
    ..error(next)
});

/**
 * 为项目添加任务
 */

app.post('/project/:id/tasks', function (req, res, next) {
   res.body.ProjectId = req.params.id;
  Task.build(req.body).save()
    .success(function (obj) {
       res.send(obj);
     })
    .error(next)
});
```

需要记住很重要的一点：像本例这样，如果用户可以设置数据库中的字段，并且没有安全隐患的情况下，我们只需传递整个请求体即可（如：传递req.body）。哪怕我们在表单中只创建了一些输入项，但是不要忘记，用户是可以手动伪造任意请求类型的。

253 如第9章中介绍的，在Express中使用res.send方法可以很容易地发送 JSON数据。

如下所示，当调用模型实例上的.save方法时，sequelize会分发一个success事件并传递构建好的对象，或者一个failure事件并传递一个错误对象。在sequelize中，如下代码是有效的：

```
Task.build(req.body).save()
    .on('success', function (obj) {
      res.send(obj);
     })
    .on('failure', next)
```

可以使用success和error方法来更容易地添加事件处理器：

```
Task.build(req.body).save()
  .success(function (obj) {
    res.send(obj);
  })
  .error(next)
```

注意了，为了确保任务和项目之前的关系能够保留，我们在使用Task.build创建任务时，在Task对象上添加了ProjectId字段。回过头来，当我们在模型上设置了belongsTo关系后，sequelize会自动在schema定义中添加ProjectId字段。

获取数据

每个sequelize模型都有简单的方法可以获取指定数据表中单个或多个实例。

调用Model#find方法时，可以直接提供一个主键，并监听success和failure事件：

```
/**
 * 首页路由
 */

app.get('/', function (req, res, next) {
  Project.findAll()
    .success(function (projects) {
      res.render('index', { projects: projects });
    })
    .error(next);
});
```

254 由于此前建立了项目——任务的联合，在/project/:id/items路由，我们可以使用getTasks方法将项目和任务都传递给视图层：

server.js

```
app.get('/project/:id/tasks', function (req, res, next) {
  Project.find(Number(req.params.id))
    .success(function (project) {
```

```
      project.getTasks().on('success', function (tasks) {
        res.render('tasks', { project: project, tasks: tasks });
      })
    })
    .error(next)
});
```

另外，还要注意的是，当使用模型实例的find方法时，需要将参数转化为Number类型。这很重要，因为这样sequelize才能知道这里是用主键去查询。

接下来，我们实现剩下的路由：删除项目和删除任务。

删除数据

接下来，我们利用事件委派来捕获所有delete类的链接，并发送DELETE请求。将如下代码添加到$(form).submit处理器中：

public/js/main.js

```
$('ul').delegate('a.delete', 'click', function (ev) {
  ev.preventDefault();
  var li = $(this).closest('li');
  $.ajax({
      url: $(this).attr('href')
    , type: 'DELETE'
    , success: function () {
        li.remove();
      }
  });
});
```

jQuery的delegate方法允许捕获任意包含了delete类的超链接，不管它是本来就在DOM中的还是后台动态加进去的。

注意，在点击了含有delete类的超链接后，我们查找它的父元素li，并在Ajax请求成功后，将其删除。

接着，定义删除路由：

255

```
/**
 * 删除项目路由
 */

app.del('/project/:id', function (req, res, next) {
  Project.find(Number(req.params.id)).success(function (proj) {
    proj.destroy()
      .success(function () {
```

```
            res.send(200);
          })
          .error(next);
    }).error(next);
  });

  /**
   * 删除任务路由
   */

  app.del('/task/:id', function (req, res, next) {
    Task.find(Number(req.params.id)).success(function (task) {
      task.destroy()
        .success(function () {
            res.send(200);
          })
          .error(next)
    }).error(next);
  });
```

如上述代码所示，首先获取任务或者项目的实例，然后，调用destroy将其删除。当destroy指令成功后，发送200状态码给浏览器。

同样的，要想修改获取到的数据项的属性，可以调用updateAttributes。下述代码修改指定任务实例的标题：

```
task.updateAttributes({
   title: 'a new title'
});
```

图13-5展示了完整的应用。可以浏览项目和任务，异步地对它们进行添加和删除。

256

图13-5：为某个项目创建一个新任务

完整地完成应用

sequelize还有许多功能。就像Mongoose操作MongoDB那样，Sequelize可以在MySQL和模

型数据之间添加一层验证层，除了定义类型之外，这个功能也非常有用。

在模型中定义字段时，可以通过传递validate选项来设置验证机制。类型则定义在type中。

比如，要想任务标题只允许大写字母，那么模型定义可以采用如下方式：

```
var Task = sequelize.define('Task', {
    title: { type: Sequelize.STRING, isUppercase: true }
});
```

要设置自定义验证，只需传递任意的函数名和验证函数。要想查看Sequelize自带的完整验证函数列表，可以前往其官方文档查看：http://sequelizejs.com/?active=validations#validations。

还可以通过自定义的类和实例方法来扩展模型：

```
var Task = sequelize.define('Task', {
    title: { type: Sequelize.STRING, isUppercase: true }
  , classMethods: {
      staticMethod: function(){}
    }
  , instanceMethods: {
      instanceMethod: function(){}
    }
});
```

上述例子中的staticMethod方法可以通过如下方式进行调用：

257

```
Task.staticMethod()
```

instanceMethod则可以在查询到的实例上使用：

```
Task.find(4).success(function (task) {
  task.instanceMethod();
});
```

小结

MySQL仍旧是最流行、最可靠的开源数据库之一。无论新的趋势如何，毋庸置疑，MySQL依然是构建各类应用不错的选择。

本章介绍了一个优秀的MySQL Node.js驱动器。不过，需要手动书写 SQL语句来创建数据库、表才能查询。

对于开发Web应用，ORM通常是一种非常有用的武器。在本章第二个例子中，得益于Sequelize，我们无须再手写查询语句，而是可以直接通过模型类和实例来操作数据了。

尽管总要在选择正确的工具这件事情上面多加谨慎，不过，现在你应该懂得了在Node.js中什么样的项目适合使用MySQL。