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。

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

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: HSmrc0sM1YUkAGmm5OPpG2HaGWk=
```

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。

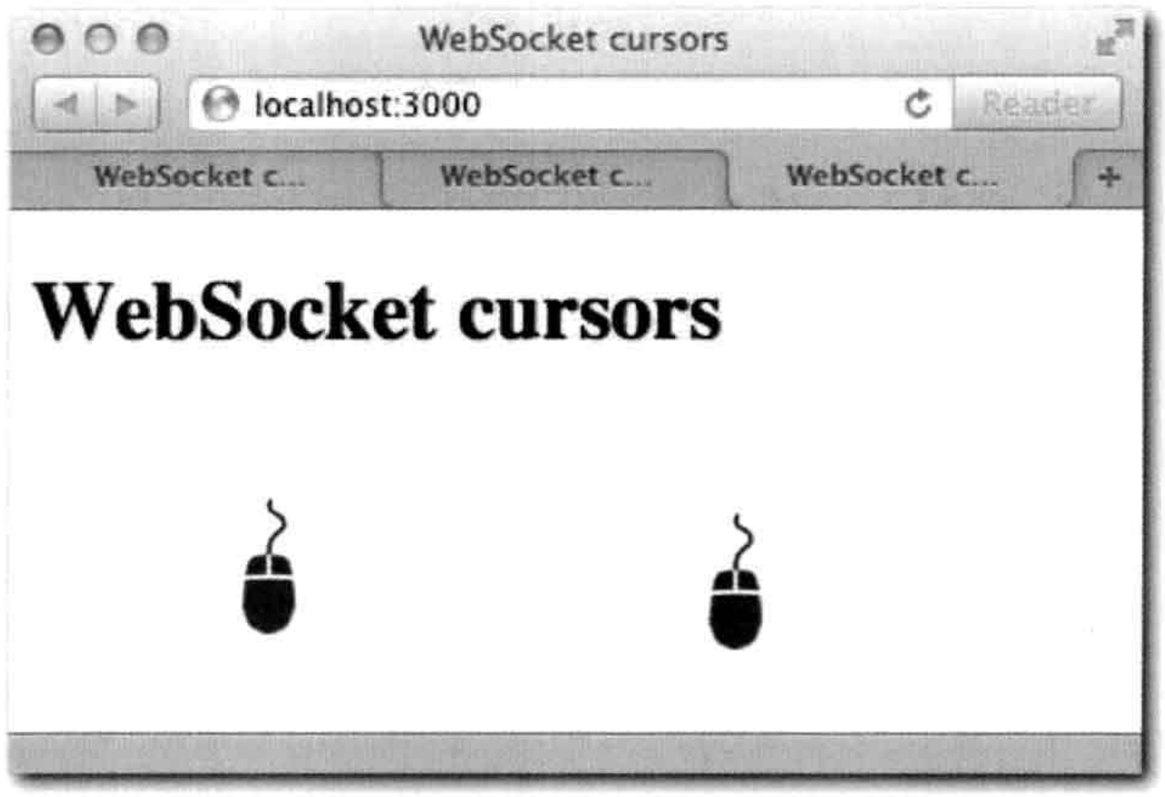

图10-2：移动的鼠标位置实时地反馈到了连入的所有客户端cursor.png图片来自http://thenounproject.com

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。