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。 179

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 182
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。

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：

index.html

184

```
<!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。

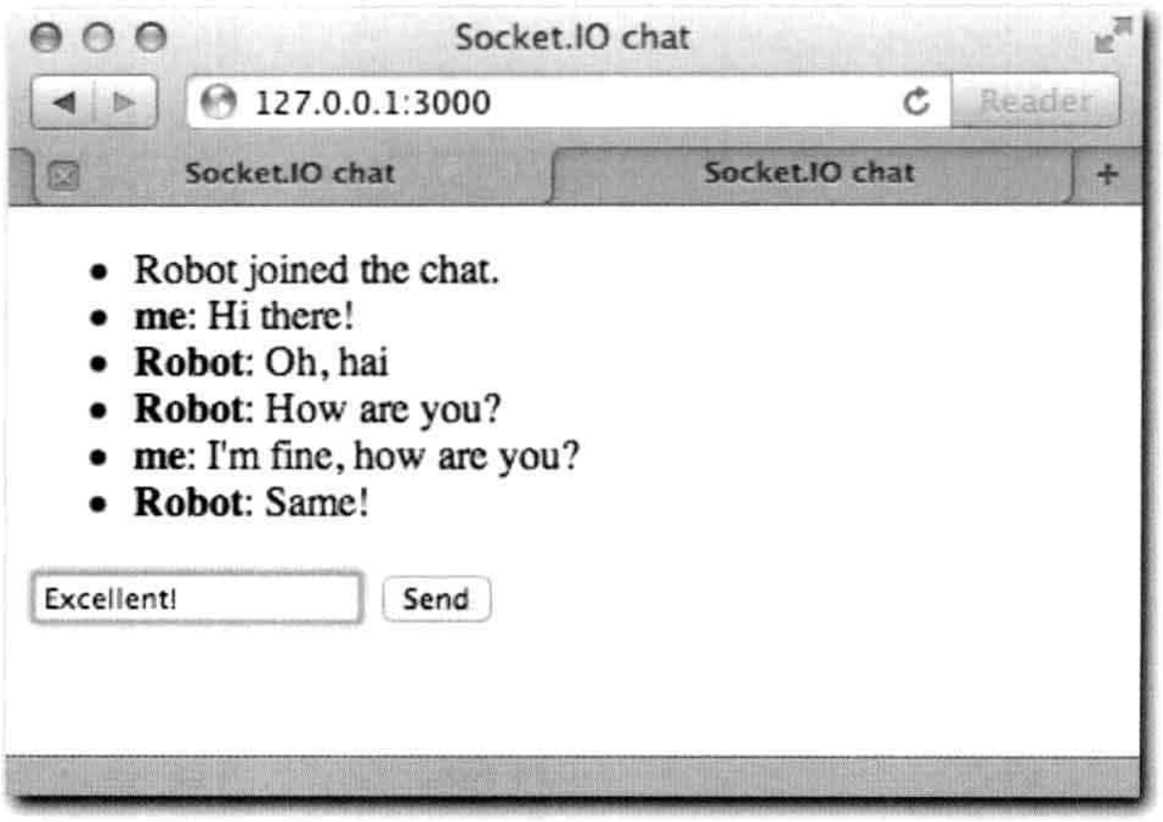

图11-2：聊天程序实战。这里通过多个浏览器标签页来聊天

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);
}
```

```
  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]);
```

199

```
      }
    });
    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。