这章介绍了 Java WebSocket API,并给出了它的功能的高层次之旅.本章将深入示例应用程序,这个应用程序的服务器端简单地回显由它的客户端发送的消息.我们将使用它说明 Java WebSocket API 的主要特性.这样做,本章将建立本书剩余部分的主要特性的一个基础.如果你需要复习下 WebSocket 协议的主要概念,请在阅读之前看下它的介绍.

创建你的第一个 WebSocket 应用程序

因为 WebSocket 协议的核心就是允许在两端之间进行消息传递,我们将以一个最简单的示例开始 Java WebSocket API: `EchoServer` 应用程序.

`EchoServer` 是一个 客户端/服务器 应用程序.它的客户端是一小段 JavaScript 代码运行在浏览器中.服务器是一个 WebSocket 端点使用 Java WebSocket API 编写,作为一个 web 应用程序的一部分部署到应用程序服务器中.当用户加载 web 网页到浏览器中,在他/她的命令中,JavaScript 代码片段被执行.这段代码要做的第一件事就是连接到应用程序服务器中的 Java EchoServer WebSocket 端点.然后立刻发送一条消息.Java `EchoServer` 端点运行于应用程序服务器中等待传入的连接,当它从它的连接的客户端中的某一个接收到一条消息,它立即回复确定.

我们将使用这个应用程序说明我们创建的 WebSocket 端点的主要概念.

这个示例通常有两步设置--首先,最重要的,我们将编写 Java EchoServer 端点并部署到服务器上.接着,包含一个 JavaScript WebSocket 客户端的 web 页面将运行于一个浏览器中.图 1-1 展示了我们将创建示例的设置的通用图示.

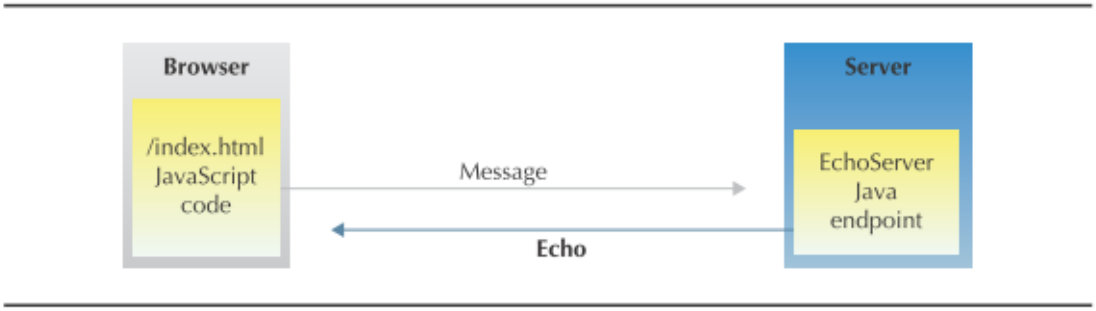


FIGURE 1-1. *The Echo sample application*

创建一个 WebSocket 端点

创建第一个 WebSocket 端点相当简单,需要了解的就是 Java WebSocket API 定义的一些 annotations.

首先,创建一个名为 `EchoServer` 的类,添加一个有一个 `String` 参数的方法保存任何从客户端传入的消息.方法的返回值也是一个 `String` .

清单 一个简单的 Java 对象(POJO)

```
1.  public class EchoServer {
2.      public String echo(String incomingMessage) {
3.              return "I got this (" + incomingMessage + ")" + " so I am sending it back !";
4.          }
5.  }
```

接下来要做的就是使用 Java WebSocket API 将这个 POJO 变为 WebSocket.我们只需要 Java WebSocket API 中的两个注解: `@ServerEndpoint` 和 `@OnMessage` .

`@ServerEndpoint` 是一个类级别的注解用来告诉 Java 平台这个类将作为一个 WebSocket 端点.这个注解只有一个强制的参数,就是相对的 URI,开发者用来使得此端点可用.这就像给某些人电话号码,这些人可以呼叫.在

这个例子中,我们保持一切简单,使用 URI `/echo` 来发布我们创建的这个新端点.
所以, `EchoServer` 类使用下面的注解:

清单 一个 POJO 变为一个 WebSocket 端点

```
1.  import javax.websocket.server.ServerEndpoint;
2.  @ServerEndpoint("/echo")
3.  public class EchoServer {
4.      public String echo(String incomingMessage) {
5.             return "I got this (" + incomingMessage + ")" + " so I am sending it back !";
6.      }
7.  }
```

这个相对的 URI `/echo` 是相对于一个 web 应用程序的 context root.为了能够使得实现的方法可以用来处理任意传入的消息,你需要使用方法级别的 `@OnMessage` 注解.这里只有一个方法.然而在一个复杂的类中,不只是只有这一个方法,还有其他的方法用来处理其它的事件.

清单:一个 WebSocket 端点

```
1.  import javax.websocket.server.ServerEndpoint;
2.  import javax.websocket.OnMessage;
3.  @ServerEndpoint("/echo")
4.  public class EchoServer {
5.       @OnMessage
6.       public String echo(String incomingMessage) {
7.              return "I got this (" + incomingMessage + ")" + " so I am sending it back !";
8.       }
9.  }
```

部署端点

将项目部署到服务器中,web 容器会自动探测是否会有 WebSocket 端点,然后会做一些必要的步骤来部署它.

创建一个 WebSocket 客户端

本书中,客户端的大部分示例都是使用 JavaScript WebSocket API 的 JavaScript 代码.Java WebSocket API 也可以创建客户端应用程序.不过这将在第3章看到.
在这个例子中,web 客户端是一个 web 页面,有一个按钮,当我们按下的时候,引发 WebSocket 客户端创建一个 WebSocket 连接到我们的 `EchoServer` 端点,然后给它发送消息.无论何时 JavaScript WebSocket 接收到一条消息,它展示在 web 页面给你看,然后关闭连接.我们将在本书的后面使用一个 longer-lived WebSocket 连接.

清单:一个 Echo JavaScript 客户端

```
1.  <!DOCTYPE html>
2.  <html>
3.      <head>
4.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
5.          <title>Web Socket JavaScript Echo Client</title>
6.          <script language="javascript" type="text/javascript">
7.              var echo_websocket;
8.
9.              function init() {
10.                 output = document.getElementById("output");
11.             }
```

```
12.
13.             function send_echo() {
14.                 var wsUri = "ws://localhost:8080/echoserver/echo";
15.                 writeToScreen("Connecting to " + wsUri);
16.                 echo_websocket = new WebSocket(wsUri);
17.                 echo_websocket.onopen = function (evt) {
18.                     writeToScreen("Connected !");
19.                     doSend(textID.value);
20.                 };
21.                 echo_websocket.onmessage = function (evt) {
22.                     writeToScreen("Received message: " + evt.data);
23.                     echo_websocket.close();
24.                 };
25.                 echo_websocket.onerror = function (evt) {
26.                     writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
27.                     echo_websocket.close();
28.                 };
29.             }
30.
31.             function doSend(message) {
32.                 echo_websocket.send(message);
33.                 writeToScreen("Sent message: " + message);
34.             }
35.
36.             function writeToScreen(message) {
37.                 var pre = document.createElement("p");
38.                 pre.style.wordWrap = "break-word";
39.                 pre.innerHTML = message;
40.                 output.appendChild(pre);
41.             }
42.
43.             window.addEventListener("load", init, false);
44.
45.         </script>
46.     </head>
47.     <body>
48.         <h1>Echo Server</h1>
49.
50.         <div style="text-align: left;">
51.             <form action="">
52.                 <input onclick="send_echo()" value="Press to send" type="button">
53.                 <input id="textID" name="message" value="Hello Web Sockets" type="text">
54.                 <br>
55.             </form>
56.         </div>
57.         <div id="output"></div>
58.     </body>
59. </html>
```

如果你运行应用程序,你应该会得到图 1-2 中的输出.

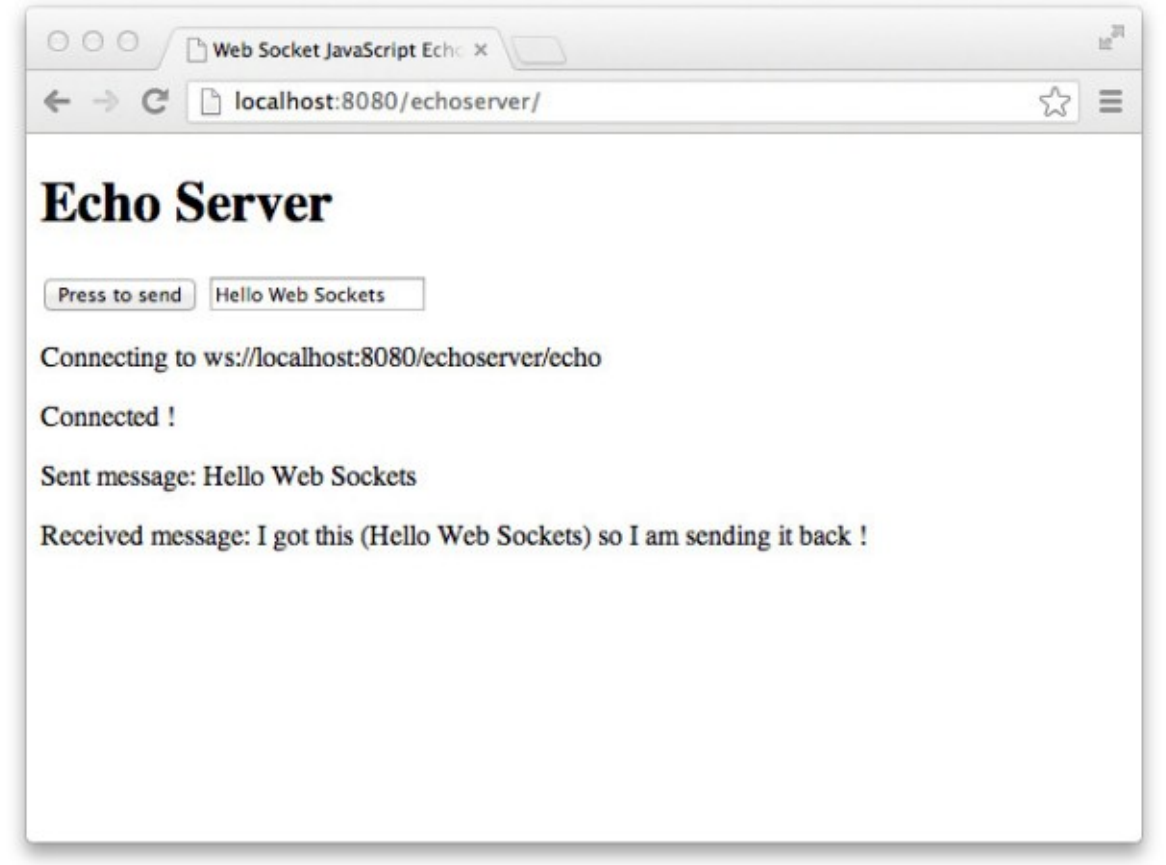
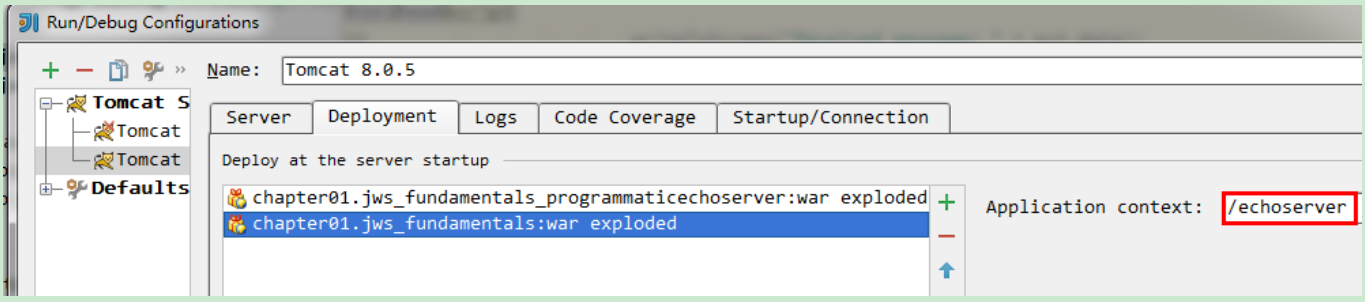


FIGURE 1-2. *Echo sample output*

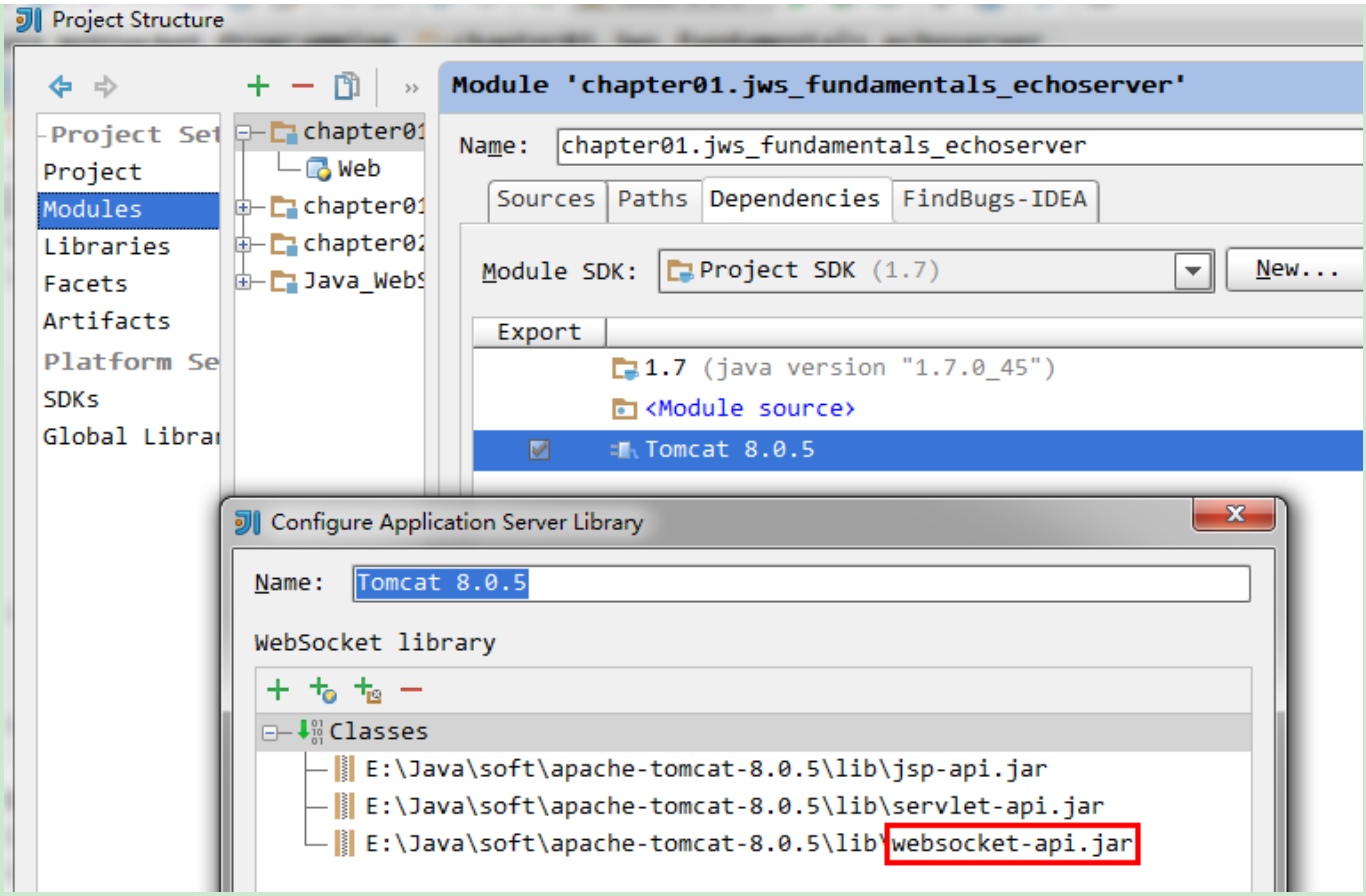
补充：直接运行书中的代码示例,很简单,如果是在 Tomcat 中,只需要进行如下配置：

```
1.    <Context path="/echoserver" docBase="...\java-websocket-code\all samples\chapter1\echoserve
      r\build\web"/>
```

这里最终指向 `\build\web` 目录即可.
如果在 Eclipse 中测试这个示例也比较简单，简单的话只需要项目名称为 `echoserver` 即可.
如果是在 IDEA 中，使用 Tomcat 部署的话需要修改一下配置，指定 `Application context` :



另外就是要包含 `websocket-api.jar` 文件,否则无法编译通过：



WebSocket 端点

我们使用术语 "端点(endpoint)" 代表 WebSocket 通信的一端.当你使用 `@ServerEndpoint` 注解标注 `EchoServer` 类,它将一个 POJO 变为一个逻辑的 WebSocket 端点.当你部署包含 `EchoServer` 类的应用程序,WebSocket 实现会扫描包含 `EchoServer` 类的 WAR 文件,然后找到它里面的一个 WebSocket 端点.接着它注册这个类为一个 WebSocket 端点,这样当一个客户端尝试在 `/echo` URI 上连接到端点,WebSocket 实现创建一个新的 `EchoServer` 类的实例,因为这个 URI 与请求匹配,然后将 `EchoServer` 类的实例放到服务中,用于后面的 WebSocket 通信.这就是当你按下示例 web 页面中按钮的时候,被调用的实例,并且它会一直存活只要 WebSocket 连接处于活动状态.

注意:
端点是 Java WebSocket API 的主要组件模型:每个你创建的 WebSocket 应用程序使用 Java WebSocket API 将会拥有 WebSocket 端点.

我们已经见过你可以使用 Java WebSocket API 创建端点的两种方式中的一种. `EchoServer` 示例向你展示了如何使用 Java WebSocket API 中的注解将一个简单的 Java 类变为一个 WebSocket 端点.第二种方式是子类化 Java WebSocket API 中包含的抽象的 `Endpoint` 类.如果你这么做,你的子类将变为一个 WebSocket 端点,正如我们稍后看到的.

注意
有两种形式的 Java WebSocket 端点:注解的端点(annotated endpoints)和编程式的端点(programmatic endpoints).

在讨论什么情况下你可能想使用注解的端点和使用编程式端点之前,你应该知道,无论采取哪种方法,总的来说,你可以访问 Java WebSocket API 的大多数功能.

编程式端点

这个示例与之前的差不多,但是使用编程式端点而不是带有注解的端点.
首先,编写一个子类化 Java WebSocket API 中的 `Endpoint` 类的 Java 类. `Endpoint` 类需要你实现 `onOpen()` 方法.这个方法的目的是注入端点实例的 life,被创建只要一个客户端连接到它.传递到这个方法中的参数是 Java WebSocket API 中的核心类.在深入细节之前,先看下代码.
如果你检查编程式 `EchoServer` 应用程序的 JavaScript 客户端,你将看到它与注解的 `EchoServer` 几乎一样.有一点不同就是它使用了不同的 URI 连接到编程式 `EchoServer` 端点:

```
ws://localhost:8080/programmaticechoserver/programmaticecho
```

让我们看下编程式的端点:

清单:编程式端点

```
import java.io.IOException;
import javax.websocket.Endpoint;
import javax.websocket.EndpointConfig;
import javax.websocket.MessageHandler;
import javax.websocket.Session;

public class ProgrammaticEchoServer extends Endpoint {
    @Override
    public void onOpen(Session session, EndpointConfig endpointConfig) {
        final Session mySession = session;
        mySession.addMessageHandler(new MessageHandler.Whole<String>() {
            @Override
            public void onMessage(String text) {
                try {
                    mySession.getBasicRemote().sendText("I got this (" +text + ") so I am s
ending it back !");
                } catch (IOException ioe) {
                    System.out.println("oh dear, something went wrong trying to send the me
ssage back: " + ioe.getMessage());
                }
            }
        });
    }
}
```

第一件注意到的事情就是它比注解的端点代码要长!对于一些人来说,这意味着他们通常将选择创建注解的端点,但是其它的开发者通常更熟悉编程式的端点.无论最后你更喜欢哪一种方式,此刻这个例子也是非常有用的,即使它看上去比注解版本的更复杂,因为即使是这样一个简单的例子也可以让我们与 API 中的一些关键对象亲密接触.传递到强制的 `onOpen()` 方法中的两个对象是 `Session` 和 `EndpointConfig` 对象.这些是需要理解的非常重要的对象,即使你总是选择注解的方式而不是编程式方式,当你创建 WebSocket 应用程序的时候,因为你将一次又一次的需要它们.

基础的 Java WebSocket API 对象

Java WebSocket API 中的 `Session` 对象给了开发者一个打开的 WebSocket 连接上的视图.每个连接到 `ProgrammaticEchoServer` 端点的客户端由一个独立的 `Session` 接口的实例代表.它持有调整连接某些属性的方法;另外,可能最重要的是,它给端点提供了一种方式访问 `RemoteEndpoint` 对象. `RemoteEndpoint` 对象代表了 WebSocket 通信的另一端,特别是给开发者提供了可以发送消息给客户端的方法.

`onOpen()` 方法的另一个参数是 `EndpointConfig` 对象. `EndpointConfig` 接口代表了用来配置端点的 WebSocket 实现的信息.这个阶段我们将不会讨论这个接口,因为我们的示例没有使用它.对于一些等不及的读者,API 端点配置的细节在第4章.

回到示例, `onOpen()` 做的第一件事情就是添加一个 `MessageHandler` 到 `session` 中. `MessageHandler` 接口(和它的实现)定义了一个编程式端点可以对接收传入的消息注册对它的兴趣的所有方式.比如,开发者可以使用 `MessageHandler` 接口选择接收文本消息或二进制消息,并且它们可以选择以块的方式接收每条消息,因为它对于应用程序的开发者交换非常大的消息非常有用.在这个例子中,我们已经实现了 `MessageHandler` 接口的最简的实现: `MessageHandler.Whole<String>` 接口.这个接口定义了如何对接收的文本消息注册一个兴趣.它需要开发者实现一个单一的方法,每次一个文本 WebSocket 消息从客户端抵达将被 WebSocket 实现调用.

`public void onMessage(String text)`

在示例中,当 WebSocke 实现传递一个这样的文本消息,我们立即获取 RemoteEndpoint 的一个引用, 我们将使用它来返回一条消息给客户端通过调用 `Session` 对象上的如下方法:

`public RemoteEndpoint.Basic getBasicRemote()`

值得注意的是有两种 `RemoteEndpoint` : `RemoteEndpoint.Basic` 和 `RemoteEndpoint.Async` . `RemoteEndpoint` 的每个子接口提供了一种不同的方式发送消息给客户端. `The RemoteEndpoint.Async` 接口提供了许多方法用来异步发送消息;也就是说,它的方法发送消息,但是在返回之前不等待消息被发送完.这种方式,开发者可以忙于处理应用程序中其它的工作而不是当前的线程阻塞直到消息被实际发送出去.更简单的 `RemoteEndpont.Basic` 接口定义了许多方法同步发送消息给客户端;也就是说,每个发送方法调用返回只有当消息被发送出去的时候.在这个例子中,这种更简单的方式是我们选择使用的方式.这个调用是:

`public void sendText(String text) throws IOException`

在 `RemoteEndpoint` 上发送文本消息给客户端,你将看到它需要开发者处理受检的 `IOException` ,当发送消息的时候如果底层的连接有问题可能会抛出.

现在我们已经看完了代码,当编程式端点被部署的时候发生了什么?那就是的当客户端连接连接到端点,WebSocket 实现调用 `ProgrammaticEchoServer` 类的 `onOpen()` 方法. `onOpen()` 实现创建了 `MessageHandler` 的实现,返回它处理后的文本消息给客户端,添加了 `MessageHandler` 实例到 session 上代表了客户端连接.一旦已经完成,该方法完成,下一次一条文本消息从客户端抵达,它将被 WebSocket 实现路由到这个 `MessageHandler` 的 `onMessage()` 方法.

一切完全与注解端点示例类似.或则是?缺失的一点是编程式端点没有我们想将端点部署到服务器上的路径.注解的端点示例,路径是一个 `@ServerEndpoint` 上的属性.在这种编程式端点案例中,分配路径有点复杂.为了部署这个例子,我们需要告诉 WebSocket 实现如何部署端点.为了这么做,我们需要提供一个 `ServerApplicationConfig` 接口的实现,将提供 WebSocket 实现需要部署它的缺失部分的信息. `ServerApplicationConfig` 接口定义了两个方法,允许开发者在应用程序中配置端点.让我们先处理第一个方法,因为它非常简单:

`public Set> getAnnotatedEndpointClasses(Set> scanned)`

这个方法被 WebSocket 实现调用当它部署应用程序的时候. `scanned` 参数是一个 `Set` ,包含了使用 `@ServerEndpoint` 注解标记的所有的 Java 类.换句话说,它传递了所有的注解的端点.开发者以这样一种方式实现这个方法,那就是返回他实际想被部署的所有的注解的端点的集合.通常,返回的 scanned 集合将允许实现部署 WAR 文件中所有的注解的端点.比如,在我们的应用程序中,没有注解端点,所以 scanned 集合将是空的. `ServerApplicationConfig` 接口的第二个方法,我们需要实现它为了部署我们的编程式的端点:

`public Set getEndpointConfigs(Set> endpointClasses)`

和注解端点的方法类似,这个方法被调用在应用程序部署阶段.传递到这个方法的 `Set` 是应用程序中继承 `Endpoint` 的所有的类的集合.也就是说,它是应用程序中所有编程式端点的集合.开发者需要实现这个方法,这样它返回 `ServerEndpointConfig` 对象的集合对应他或她希望被部署的 WebSocket 实现的所有的编程式端点.所以在我们的例子中,这个集合中的一个类将被调用: 这个类当然是 `ProgrammaticEchoServer` .我们需要

做的就是创建一个 `ServerEndpointConfig` 对象从这个方法中返回,这样 WebSocket 实现将用来部署端点.让我们看下代码.

清单 Echo 示例的 ServerApplicationConfig

```
1.  import java.util.HashSet;
2.  import java.util.Set;
3.  import javax.websocket.Endpoint;
4.
5.  import javax.websocket.server.ServerApplicationConfiguration;
6.  import javax.websocket.server.ServerEndpointConfiguration;
7.  import javax.websocket.server.ServerEndpointConfiguration.Builder;
8.
9.  public class ProgrammaticEchoServerAppConfig implements ServerApplicationConfiguration {
10.
11.     @Override
12.     public Set<ServerEndpointConfiguration>
13.       getEndpointConfigurations(Set<Class<? extends Endpoint>> endpointClasses) {
14.         Set configs = new HashSet<ServerEndpointConfiguration>();
15.         ServerEndpointConfiguration sec =
16.            ServerEndpointConfiguration.Builder
17.            .create(ProgrammaticEchoServer.class, "/programmaticecho")
18.            .build();
19.         configs.add(sec);
20.         return configs;
21.     }
22.
23.     @Override
24.     public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned){
25.         return scanned;
26.     }
27. }
```

`getAnnotatedEndpointClasses()` 只是简单的返回 scanned 集合,在这个例子中是空的.因为这个例子中没有注解的端点.对于 `getEndpointConfigs()` 调用,我们将创建一个单一的端点配置对象,保存我们想要部署它的路径和我们的端点类.然后我们将它添加到所有的端点配置的集合中,并返回它,这是一条指令,WebSocket 实现部署这个单一的编程式端点到给定的路径 `/programmaticecho`.

然后我们以和之前注解端点例子中相同的方式部署 WAR 文件到应用程序服务器中,我们将得到一个类似的结果当我们运行应用程序的时候.

本书中我们倾向于使用注解方式,但我们也将混合使用编程式方法.本书推荐你学习两种方式的基础,这样你可以打开思路探索哪种方式更加适合于应用程序.

深入 Echo 示例

部署阶段

当我们部署 WAR 文件的时候,它包含了我们的端点到应用程序服务器中,将发生很多事情为了准备应用程序的第一个连接.WebSocket 实现要做的第一件事情就是检测 WAR 文件尝试去定位任何可能需要被部署的端点.首先,检测将定位任何使用 `@ServerEndpoint` 注解的 Java 类,和任何继承 Java WebSocket API 中的 `Endpoint` 类的 Java 类.它也将定义 WAR 文件中任意实现了 `ServerApplicationConfig` 接口的类;这些类将告诉它如何部署端点.

一旦 WebSocket 实现获取了这些信息,它将使用这些信息去构建端点的集合进行部署.在 `EchoServer` 示例中,我们的 WAR 文件简单的持有一个注解的端点.如果在 WAR 文件中没有 `ServerApplicationConfig` 实

现,WebSocket 实现将自动部署所有的注解端点.这个确实是我们的注解的例子的情况,所以这就是端点如何变为被部署的.在编程式 Echo 示例中,WAR 文件包含了 `ServerApplicationConfig` 接口的实现,这样 WebSocket 实现实例化这个类在部署阶段期间,然后查询它的方法获知哪个端点要被部署.
还有其它的开发者可以使用 Java WebSocket API 进行部署的选项,可能在同一个 WAR 中打包多个 `ServerApplicationConfig` 实现.
一旦实现检测到 WAR 文件中要被部署的 WebSocket 端点的集合,大部分实现将在端点上做其它的检测,在进一步处理之前,只会部署一个结构良好的应用程序.不同的 WebSocket 实现可能会捕获和报告配置或编程错误(比如两个端点映射到相同的路径或者 WebSocket 注解用在了错误的语义级别)在部署阶段的不同时刻和以不同的方式,依赖于它们的架构和设计.某些可能在日志文件中报告.某些可能使用图形用户界面工具在屏幕上报告有用的信息在应用程序部署阶段.然而,如果一切正常,并且应用程序是有效的,WebSocket 实现将端点和它们声明的 URIs 进行关联,这样应用程序将准备好接收传入的连接.

接收第一个连接

当初始化一个 WebSocket 连接的时候第一件发生的事就是一个初始化的 HTTP 请求/响应 交互.交互被称为 WebSocket opening handshake(WebSocket 开始握手).许多 WebSocket 开发者将永远不需要理解这个交互是如何工作的细节,除非你需要了解电话交换或蜂窝网络的机制来完成一个电话呼叫.这个主题将在第4章涵盖.我只想说,此时,客户端想使用完整的 URI 地址连接,包括主机名和端点发布的相对地址,然后它会发出一种特殊的 HTTP 请求到该 URI.此时和 opening handshake 客户端关联的还有其它的参数,这是一个主题将在第 4 章涵盖.当服务器接收到这个 opening handshake 请求.它检查请求,可能在客户端执行许多检查(比如,客户端发出它从哪里来的请求?客户端是否被授权?)如果一切正常,服务器将返回一个特殊格式的 HTTP 响应,将告诉客户端服务器是否希望接收传入的连接.在典型情况下,对于 WebSocket 开发者而言,所有的这些都是在 "幕后" 发生的,尽管在更高级的案例中 Java WebSocket 开发者可能要拦截这个请求和响应交互为了自定义它.我们在图 1-3 中说明了 opening handshake 的概念.
如果一切正常,在服务器中确实有一个 WebSocket 端点注册到了 opening handshake 提供的地址,连接将被建立.WebSocket 实现将创建端点的一个新的实例,无论是注解或编程式,将专用于与现在连接的客户端交互.这就意味着,比如,如果你的 WebSocket 端点与多个客户端连接,WebSocket 实现将实例化你的端点多次,一次为一个新的客户端连接.图1-4 展示了 opening handshake 请求的一个示例.

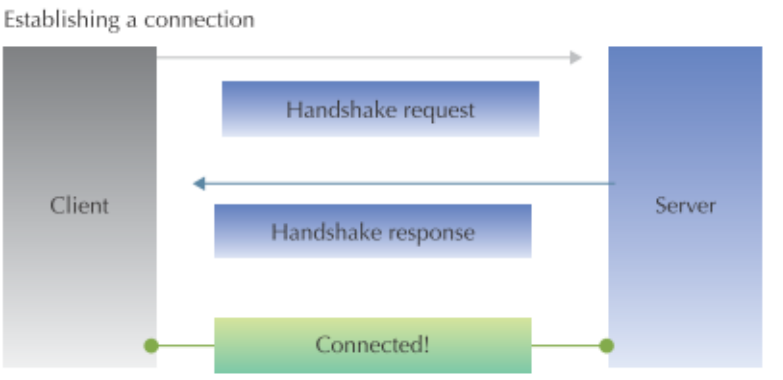


FIGURE 1-3. *The opening handshake interaction*

Handshake Request

```
Http Request
GET /mychat HTTP/1.1
Host:server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: megachat,chat
Sec-WebSocket-Extensions: compress,mux
Sec-WebSocket-Version: 13
Origin:http://example.com
```

FIGURE 1-4. *An opening handshake request*

在这个例子中,客户端请求连接到由 http://server.example.com 服务器托管的一个 WebSocket,URI 相对于服务器的根路径是 `/mychat` .你将注意到 WebSocket 协议使用 HTTP `Upgrade` 机制,被浏览器使用的相同的一般机制是用来升级一个 HTTP 连接为一个安全的 HTTPS 连接,除了在这种情况下,客户端请求的协议当然是 WebSocket 协议.客户端发送一个独立的 token 将用在响应中,你将注意到有其它的 headers 定义了客户端希望使用的 WebSocket 协议版本和一些它希望使用的一些特殊的子协议和扩展.在后面的章节中我们将接触 WebSocket 协议的明确的定义和子协议与扩展的使用.此时,简单地知道它们表示一个特定应用程序调整协议来更好地满足它的需求的方式即可.

最后, opening handshake 请求也同样声明了原始的 header,比如,如果它是由浏览器制定,这通常是服务包含 WebSocket 客户端代码的 web 站点的因特网地址.

图 1-5 展示了一个 opening handshake 响应的例子.在这个例子中,服务器同意 "升级" 连接为 WebSocket 连接.它发送回一个安全的 token,这就是客户端用来验证它接收到的响应来响应它发送相同的请求(which the client uses to verify that the response it received came in response to the same request it sent).这里例子中的服务器决定使用 opening handshake 请求中列出的请求的 chat 子协议.因为它支持 WebSocket 协议的两种扩展(分别称为 compress、mux—compression 和 multiplexing),它同意在客户端和服务器之间已经建立的连接中使用这些扩展作为这个 opening handshake 的结果.

Handshake Response

```
Http Response
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Extensions: compress, mux
```

FIGURE 1-5. *The opening handshake response*

WebSocket 消息

现在 TCP 连接建立了.WebSocket 协议在 TCP 之上使用最小帧(minimal framing) 定义了一条消息协议,通过 TCP 连接被向后和向前发送(sent backward and forward)的不同的 WebSocket 协议帧定义了 WebSocket 的生命周期事件,比如,打开和关闭连接,同时定义了应用程序创建的文本和二进制消息如何通过连接被传输.

当 WebSocket 实现接收一个传入的连接,它保证有一个端点的实例能够处理它,它与发起连接的客户端相关联.你将在后面的章节中学习到关于客户端连接和端点实例之间关联的更多信息.此时,无论何时客户端发送消息到端点,端点实例,实现了与客户端接收回调消息的有效负载的方法.端点通过 `Session` 对象获取一个到

`RemoteEndpoint` 对象的引用,唯一的代表客户端连接来发送消息.相同的关联隐式存在于注解端点的消息处理方法返回值的情况中(The same association exists implicitly in the case of a return value from the message-handling method of an annotated endpoint).图 1-6 展示了被设置在典型的服务器部署用于编程式端点的对象模型图.

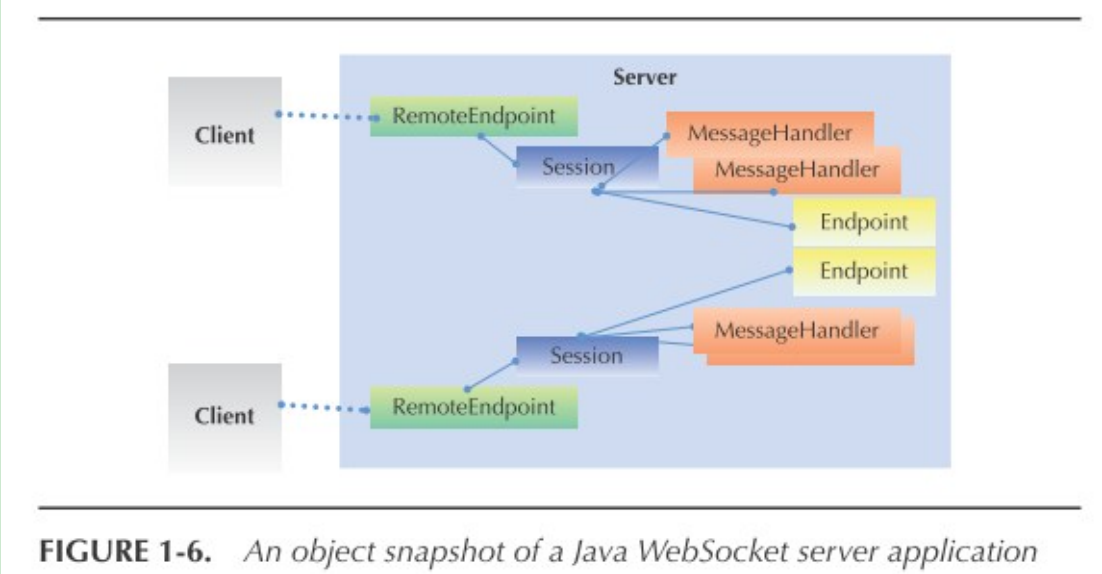


FIGURE 1-6. *An object snapshot of a Java WebSocket server application*

在这个图中,我们可以看到,在服务器上部署了一个逻辑端点,有两个实例由两个 `Endpoint` 方框表示,每种消息处理器来自不同的客户端.端点的每个实例注册了多个 `MessageHandler` 实例去处理传入的消息.你也可以看到每个端点有一个 `Session` 对象的引用代表了到每个客户端的连接.这些 sessions 每个都引用了一个 `RemoteEndpoint` 对象的实例,给了端点发送消息回给客户端的能力.

对于注解端点,对象模型在运行时被设置,大体是相同的, `Session` 和 `RemoteEndpoint` 对象总是对端点可用的.然而有一个简化,那就是注解端点的 `MessageHandler` 由 WebSocket 实现生成;开发者永远不需要创建,注册或引用它们.

当然,这个图只是当连接处于活动状态的对象的快照.作为一个 WebSocket 的端点的全貌(As a complete picture of a WebSocket endpoint, it is only as complete as a single photograph of two people speaking on the phone is a complete description of a phone conversation).为了更全面地了解两个端点之间的完整的 WebSocket 通信,我们需要深入 WebSocket 端点生命周期.这将是下一章的主题.

总结

在这章,你已经看到了如何使用创建一个简单的 客户端/服务器 WebSocket 应用程序,使用 Java WebSocket API 创建服务器,使用 JavaScript API 创建客户端.你也看到了 WebSocket 端点如何被注册通过使用 JavaSocket 中 Java 注解或者继承 Java WebSocket API 中的 `Endpoint` 类.这章接触了 Java WebSocket API 中最重要的对象: `Session` , `RemoteEndpoint` , 和 `MessageHandler` 对象.它以两种方式检测在 WebSocket 端点作为一个标准的 web 应用程序的一部分被部署的时候.