第 2-10 课：使用 Spring Boot WebSocket 创建聊天室

什么是 WebSocket

WebSocket 协议是基于 TCP 的一种网络协议，它实现了浏览器与服务器全双工（Full-duplex）通信——允许服务器主动发送信息给客户端。

以前，很多网站为了实现推送技术，所用的技术都是轮询。轮询是在特定的的时间间隔（如每 1 秒），由浏览器对服务器发出 HTTP 请求，然后由服务器返回最新的数据给客户端的浏览器。这种传统的模式带来很明显的缺点，即浏览器需要不断地向服务器发出请求，然而 HTTP 请求可能包含较长的头部，其中真正有效的数据可能只是很小的一部分，显然这样会浪费很多的带宽等资源。

在这种情况下，HTML 5 定义了 WebSocket 协议，能更好得节省服务器资源和带宽，并且能够更实时地进行通讯。WebSocket 协议在 2008 年诞生，2011 年成为国际标准，现在主流的浏览器都已经支持。

它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

优点

- **较少的控制开销**。在连接创建后，服务器和客户端之间交换数据时，用于协议控制的数据包头部相对较小。在不包含扩展的情况下，对于服务器到客户端的内容，此头部大小只有 2 至 10 字节（和数据包长度有关）；对于客户端到服务器的内容，此头部还需要加上额外的 4 字节的掩码。相对于 HTTP 请求每次都要携带完整的头部，此项开销显著减少了。
- **更强的实时性**。由于协议是全双工的，所以服务器可以随时主动给客户端下发数据。相对于 HTTP 请求需要等待客户端发起请求服务端才能响应，延迟明显更少；即使是和 Comet 等类似的长轮询比较，其也能在短时间内更多次地传递数据。
- **保持连接状态**。与 HTTP 不同的是，Websocket 需要先创建连接，这就使得其成为一种有状态的协议，之后通信时可以省略部分状态信息，而 HTTP 请求可能需要在每个请求都携带状态信息（如身份认证等）。
- **更好的二进制支持**。Websocket 定义了二进制帧，相对 HTTP，可以更轻松地处理二进制内容。 可以支持扩展。Websocket 定义了扩展，用户可以扩展协议、实现部分自定义的子协议。如部分浏览器支持压缩等。
- **更好的压缩效果**。相对于 HTTP 压缩，Websocket 在适当的扩展支持下，可以沿用之前内容的上下文，在传递类似的数据时，可以显著地提高压缩率。

WebSocket 在握手之后便直接基于 TCP 进行消息通信，但 WebSocket 只是 TCP 上面非常轻的一层，它仅仅将 TCP 的字节流转换成消息流（文本或二进制），至于怎么解析这些消息的内容完全依赖于应用本身。

因此为了协助 Client 与 Server 进行消息格式的协商，WebSocket 在握手的时候保留了一个子协议字段。

Stomp 和 WebSocket

STOMP 即 Simple (or Streaming) Text Orientated Messaging Protocol，简单（流）文本定向消息协议，它提供了一个可互操作的连接格式，允许 STOMP 客户端与任意 STOMP 消息代理（Broker）进行交互。STOMP 协议由于

设计简单，易于开发客户端，因此在多种语言和多种平台上得到了广泛的应用。

STOMP 协议并不是为 Websocket 设计的，它是属于消息队列的一种协议，它和 Amqp、Jms 平级。只不过由于它的简单性恰巧可以用于定义 Websocket 的消息体格式。可以这么理解，Websocket 结合 Stomp 子协议段，来让客户端和服务器在通信上定义的消息内容达成一致。

STOMP 协议分为客户端和服务端，具体如下。

STOMP 服务端

STOMP 服务端被设计为客户端可以向其发送消息的一组目标地址。STOMP 协议并没有规定目标地址的格式，它由使用协议的应用自己来定义。例如，/topic/a、/queue/a、queue-a 对于 STOMP 协议来说都是正确的。应用可以自己规定不同的格式以此来表明不同格式代表的含义。比如应用自己可以定义以 /topic 打头的为发布订阅模式，消息会被所有消费者客户端收到，以 /user 开头的为点对点模式，只会被一个消费者客户端收到。

STOMP 客户端

对于 STOMP 协议来说，客户端会扮演下列两种角色的任意一种：

- 作为生产者，通过 SEND 帧发送消息到指定的地址；
- 作为消费者，通过发送 SUBSCRIBE 帧到已知地址来进行消息订阅，而当生产者发送消息到这个订阅地址后，订阅该地址的其他消费者会受到通过 MESSAGE 帧收到该消息。

实际上，WebSocket 结合 STOMP 相当于构建了一个消息分发队列，客户端可以在上述两个角色间转换，订阅机制保证了一个客户端消息可以通过服务器广播到多个其他客户端，作为生产者，又可以通过服务器来发送点对点消息。

STOMP 帧结构

```
COMMAND

header1:value1

header2:value2

Body^@
```

其中，^@ 表示行结束符。

一个 STOMP 帧由三部分组成：命令、Header（头信息）、Body（消息体）。

- 命令使用 UTF-8 编码格式，命令有 SEND、SUBSCRIBE、MESSAGE、CONNECT、CONNECTED 等。
- Header 也使用 UTF-8 编码格式，它类似 HTTP 的 Header，有 content-length、content-type 等。
- Body 可以是二进制也可以是文本，注意 Body 与 Header 间通过一个空行（EOL）来分隔。

来看一个实际的帧例子：

```
SEND
destination:/broker/roomId/1
content-length:57

{“type":"OUT","content":"ossxxxxx-wq-yyyyyyyy"}
```

- 第 1 行：表明此帧为 SEND 帧，是 COMMAND 字段。
- 第 2 行：Header 字段，消息要发送的目的地址，是相对地址。
- 第 3 行：Header 字段，消息体字符长度。
- 第 4 行：空行，间隔 Header 与 Body。
- 第 5 行：消息体，为自定义的 JSON 结构。

更多 STOMP 协议细节，可以参考 STOMP 官网。

WebSocket 事件

Websocket 使用 ws 或 wss 的统一资源标志符，类似于 HTTPS，其中 wss 表示在 TLS 之上的 Websocket。例如：

```
ws://example.com/wsapi
wss://secure.example.com/
```

Websocket 使用和 HTTP 相同的 TCP 端口，可以绕过大多数防火墙的限制。默认情况下，Websocket 协议使用 80 端口；运行在 TLS 之上时，默认使用 443 端口。

事件	事件处理程序	描述
open	Sokcket onopen	连接建立时触发
message	Sokcket onopen	客户端接收服务端数据时触发
error	Sokcket onerror	通讯发生错误时触发
close	Sokcket onclose	链接关闭时触发

下面是一个页面使用 Websocket 的示例：

```
var ws = new WebSocket("ws://localhost:8080");

ws.onopen = function(evt) {
  console.log("Connection open ...");
  ws.send("Hello WebSockets!");
};

ws.onmessage = function(evt) {
  console.log( "Received Message: " + evt.data);
  ws.close();
};

ws.onclose = function(evt) {
  console.log("Connection closed.");
};
```

Spring Boot 提供了 Websocket 组件 spring-boot-starter-websocket，用来支持在 Spring Boot 环境下对 Websocket 的使用。

Websocket 聊天室

Websocket 双相通讯的特性非常适合开发在线聊天室，这里以在线多人聊天室为示例，演示 Spring Boot Websocket 的使用。

首先我们梳理一下聊天室都有什么功能：

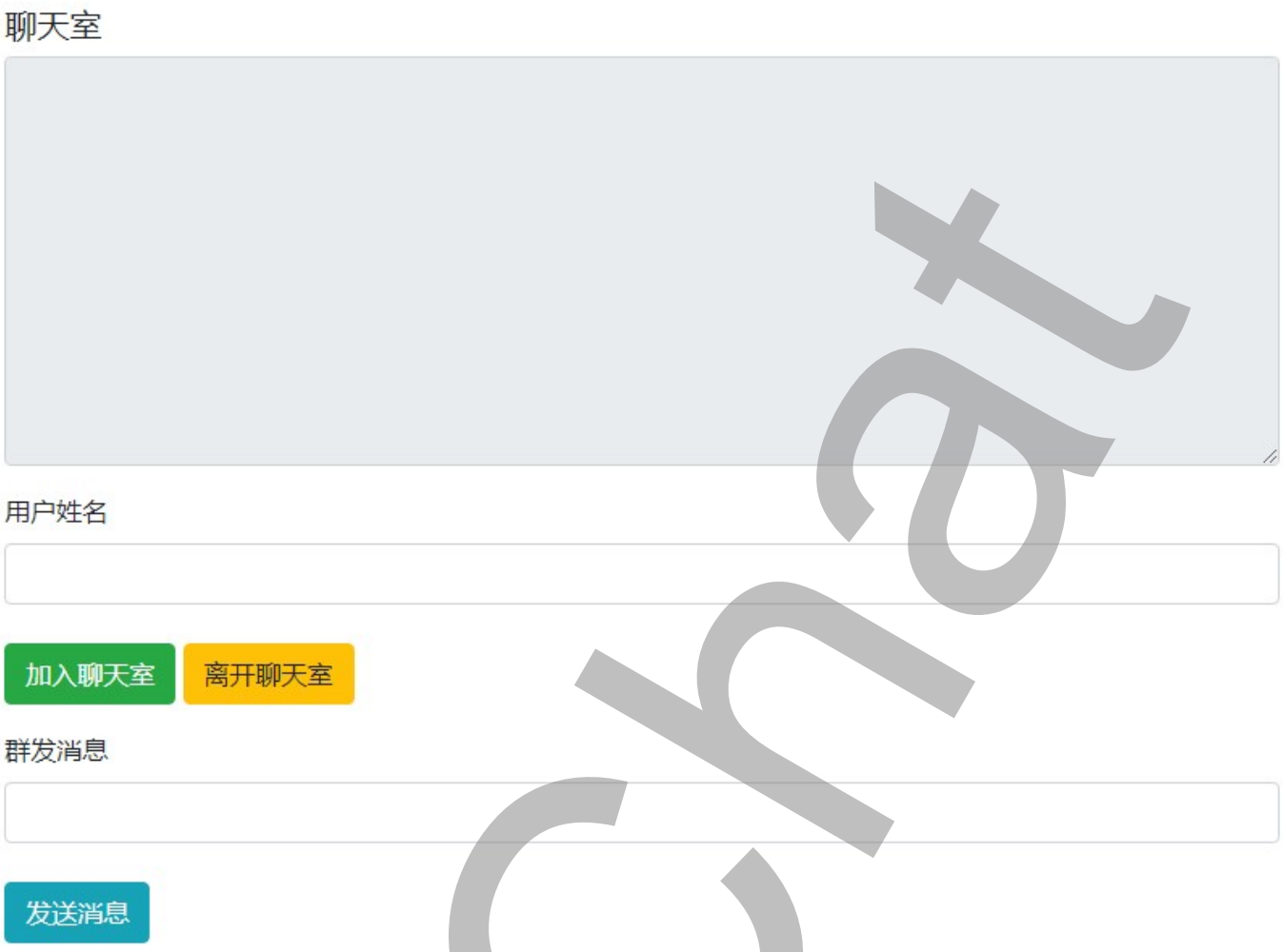
- 支持用户加入聊天室，对应到 Websocket 技术就是建立连接 onopen
- 支持用户退出聊天室，对应到 Websocket 技术就是关闭连接 onclose
- 支持用户在聊天室发送消息，对应到 Websocket 技术就是调用 onmessage 发送消息
- 支持异常时提示，对应到 Websocket 技术 onerror

页面开发

利用前端框架 Bootstrap 渲染页面，使用 HTML 搭建页面结构，完整页面内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>chat room websocket</title>
    <link rel="stylesheet" href="bootstrap.min.css">
    <script src="jquery-3.2.1.min.js" ></script>
</head>
<body class="container" style="width: 60%">
<div class="form-group" ></br>
    <h5>聊天室</h5>
    <textarea id="message_content"  class="form-control"  readonly="readonly" cols="50"
 rows="10"></textarea>
</div>
<div class="form-group" >
    <label for="in_user_name">用户姓名  </label>
    <input id="in_user_name" value="" class="form-control" /></br>
    <button id="user_join" class="btn btn-success" >加入聊天室</button>
    <button id="user_exit" class="btn btn-warning" >离开聊天室</button>
</div>
<div class="form-group" >
    <label for="in_room_msg" >群发消息  </label>
    <input id="in_room_msg" value="" class="form-control" /></br>
    <button id="user_send_all" class="btn btn-info" >发送消息</button>
</div>
</body>
</html>
```

最上面使用 textarea 画一个对话框，用来显示聊天室的内容；中间部分添加用户加入聊天室和离开聊天室的按钮，按钮上面是输入用户名的入口；页面最下面添加发送消息的入口，页面显示效果如下：



接下来在页面添加 WebSocket 通讯代码：

```
<script type="text/javascript">
    $(document).ready(function(){
        var urlPrefix ='ws://localhost:8080/chat-room/';
        var ws = null;
        $('#user_join').click(function(){
            var username = $('#in_user_name').val();
            var url = urlPrefix + username;
            ws = new WebSocket(url);
            ws.onopen = function () {
                console.log("建立 websocket 连接...");
            };
            ws.onmessage = function(event){
                //服务端发送的消息
                $('#message_content').append(event.data+'\n');
            };
            ws.onclose = function(){
                $('#message_content').append('用户['+username+'] 已经离开聊天室!');
                console.log("关闭 websocket 连接...");
            }
        });
        //客户端发送消息到服务器
        $('#user_send_all').click(function(){
            var msg = $('#in_room_msg').val();
            if(ws){
                ws.send(msg);
            }
        });
        // 退出聊天室
        $('#user_exit').click(function(){
            if(ws){
                ws.close();
            }
        });
    })
</script>
```

这段代码的功能主要是监听三个按钮的点击事件，当用户登录、离开、发送消息是调用对应的 WebSocket 事件，将信息传送给服务端。同时打开页面时创建了 WebSocket 对象，页面会监控 WebSocket 事件，如果后端服务和前端通讯室将对应的信息展示在页面。

服务端开发

引入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

主要添加 Web 和 Websocket 组件。

启动类

启动类需要添加 @EnableWebSocket 开启 WebSocket 功能。

```
@EnableWebSocket
@SpringBootApplication
public class WebSocketApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebSocketApplication.class, args);
    }

    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }
}
```

请求接收

在创建服务端消息接收功能之前，我们先创建一个 WebSocketUtils 工具类，用来存储聊天室在线的用户信息，以及发送消息的功能。首先定义一个全局变量 ONLINE_USER_SESSIONS 用来存储在线用户，使用 ConcurrentHashMap 提升高并发时效率。

```
public static final Map<String, Session> ONLINE_USER_SESSIONS = new ConcurrentHashMap<>
();
```

封装消息发送方法，在发送之前首先判单用户是否存在再进行发送：

```
public static void sendMessage(Session session, String message) {
    if (session == null) {
        return;
    }
    final RemoteEndpoint.Basic basic = session.getBasicRemote();
    if (basic == null) {
        return;
    }
    try {
        basic.sendText(message);
    } catch (IOException e) {
        logger.error("sendMessage IOException ",e);
    }
}
```

聊天室的消息是所有在线用户可见，因此每次消息的触发实际上是遍历所有在线用户，给每个在线用户发送消息。

```
public static void sendMessageAll(String message) {
    ONLINE_USER_SESSIONS.forEach((sessionId, session) -> sendMessage(session, message))
;
}
```

其中，`ONLINE_USER_SESSIONS.forEach((sessionId, session) -> sendMessage(session, message))` 是 JDK 1.8 forEach 的简洁写法。

这样我们在创建 ChatRoomServerEndpoint 类的时候就可以直接将工具类的方法和全局变量导入：

```
import static com.neo.utils.WebSocketUtils.ONLINE_USER_SESSIONS;
import static com.neo.utils.WebSocketUtils.sendMessageAll;
```

接收类上需要添加 @ServerEndpoint("url") 代表监听此地址的 WebSocket 信息。

```
@RestController
@ServerEndpoint("/chat-room/{username}")
public class ChatRoomServerEndpoint {
}
```

用户登录聊天室时，将用户信息添加到 ONLINE_USER_SESSIONS 中，同时通知聊天室中的人。

```
@OnOpen
public void openSession(@PathParam("username") String username, Session session) {
    ONLINE_USER_SESSIONS.put(username, session);
    String message = "欢迎用户[" + username + "] 来到聊天室！";
    logger.info("用户登录："+message);
    sendMessageAll(message);
}
```

其中，@OnOpen 注解和前端的 onopen 事件一致，表示用户建立连接时触发。

当聊天室某个用户发送消息时，将此消息同步给聊天室所有人。

```
@OnMessage
public void onMessage(@PathParam("username") String username, String message) {
    logger.info("发送消息：“+message);
    sendMessageAll("用户[" + username + "] : " + message);
}
```

其中，@OnMessage 监听发送消息的事件。

当用户离开聊天室后，需要将用户信息从 ONLINE_USER_SESSIONS 移除，并且通知到在线的其他用户：

```
@OnClose
public void onClose(@PathParam("username") String username, Session session) {
    //当前的Session 移除
    ONLINE_USER_SESSIONS.remove(username);
    //并且通知其他人当前用户已经离开聊天室了
    sendMessageAll("用户[" + username + "] 已经离开聊天室了！");
    try {
        session.close();
    } catch (IOException e) {
        logger.error("onClose error",e);
    }
}
```

其中，@OnClose 监听用户断开连接事件。

当 WebSocket 连接出现异常时，出触发 @OnError 事件，可以在此方法内记录下错误的异常信息，并关闭用户连接。

```
@OnError
public void onError(Session session, Throwable throwable) {
    try {
        session.close();
    } catch (IOException e) {
        logger.error("onError excepiton",e);
    }
    logger.info("Throwable msg "+throwable.getMessage());
}
```

到此我们服务端内容就开发完毕了。

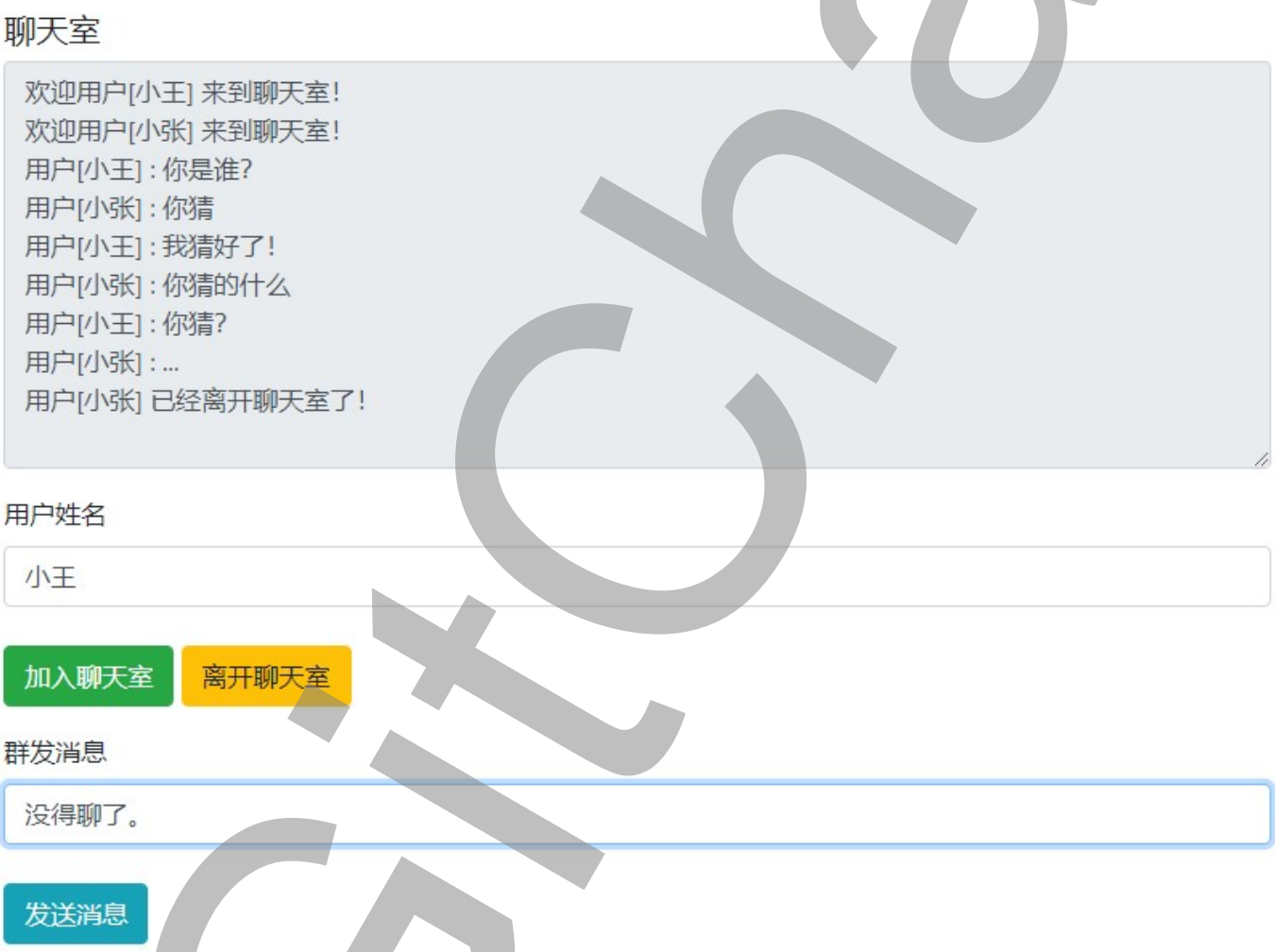
测试

启动 spring-boot-websocket 项目，在浏览器中输入地址 http://localhost:8080/ 打开两个页面进行测试。

在第一个页面中以用户“小王”登录聊天室，第二个页面以“小张”登录聊天室。

> 小王：你是谁?
> 小张：你猜
> 小王：我猜好了!
> 小张：你猜的什么
> 小王：你猜?
> 小张：...
>
> 小张离开聊天室...

大家在两个页面模式小王和小张对话，可以看到两个页面的展示效果，页面都可实时无刷新展示最新聊天内容，页面最终展示效果如下：



总结

这节课首先介绍了 WebSocket，以及 WebSocket 的相关特性和优点，Spring Boot 提供了 WebSocket 对应的组件包，因此很容易让我们集成在项目中。利用 WebSocket 可以双向通讯的特点做了一个简易版的聊天室，来验证 WebSocket 相关特性，通过示例实践发现 WebSocket 双向通讯机制非常高效简洁，特别适合在服务端和客户端通讯较多的场景下使用，相比以前的轮询方式更加优雅易用。

> 点击这里下载源码。