springmvc(18)使用WebSocket 和 STOMP 实现消息功能

【0】README

1）本文旨在 介绍如何 利用 WebSocket 和 STOMP 实现消息功能；

2）要知道， WebSocket 是发送和接收消息的 底层API，而SockJS 是在 WebSocket 之上的 API；最后 STOMP（面向消息的简单文本协议）是基于 SockJS 的高级API

（干货——简而言之，WebSocket 是底层协议，SockJS 是WebSocket 的备选方案，也是 底层协议，而 STOMP 是基于 WebSocket（SockJS） 的上层协议）

3）broker==经纪人，代理；

4）当然，你可以直接跳转到 STOMP 知识（章节【3】）；

【1】WebSocket

1）intro：WebSocket 协议提供了 通过一个套接字实现全双工通信的功能。也能够实现 web 浏览器 和 server 间的 异步通信， 全双工意味着 server 与浏览器间 可以发送和接收消息。

【1.1】使用 spring 的低层级 WebSocket API

1）intro：为了在 spring 中 使用较低层级的 API 来处理消息。有如下方案：

> scheme1）我们必须编写一个实现 WebSocketHandler:
>
> scheme2）当然，我们也可以扩展 AbstractWebSocketHandler（更加简单一点）；

对以上代码的分析（Analysis）： 当然了，我们还可以重载其他三个方法：

scheme3）也可以扩展 TextWebSocketHandler（文本 WebSocket 处理器）， 不在扩展AbstractWebSocketHandler ， TextWebSocketHandler 继承 AbstractWebSocketHandler ；

2）你可能会关系建立和关闭连接感兴趣。可以重载 afterConnectionEstablished() and afterConnectionClosed():

```
1.
2.  public void afterConnectionEstablished(WebSocketSession session)
3.  throws Exception {
4.  logger.info("Connection established");
5.  }
```

```
1.
2.  @Override
3.  public void afterConnectionClosed(
4.  WebSocketSession session, CloseStatus status) throws Exception {
5.  logger.info("Connection closed. Status: " + status);
6.  }
```

3）现在已经有了 message handler 类了，下面对其进行配置，配置到 springmvc 的运行环境中。

```
1.  @Configuration
2.  @EnableWebSocket
3.  public class WebSocketConfig implements WebSocketConfigurer{
4.   @Override
5.   public void registerWebSocketHandlers(WebSocketHandlerRegistry registry
    ) {
6.    registry.addHandler(getTextHandler(), "/websocket/p2ptext");
7.   }
8.
```

```
9.  @Bean
10.  public ChatTextHandler getTextHandler() {
11.   return new ChatTextHandler();
12.  }
13. }
```

对上述代码的分析（Analysis）： registerWebSocketHandlers方法 是注册消息处理器的关键： 通过 调用 WebSocketHandlerRegistry .addHandler() 方法 来注册信息处理器；

Attention）server 端的 WebSocket 配置完毕，下面配置客户端；

4）WebSocket 客户端配置

> 4.1）client 发送 一个文本到 server，他监听来自 server 的文本消息。下面代码 展示了 利用 js 开启一个原始的 WebSocket 并使用它来发送消息给 server；
>
> 4.2）代码如下：

error）这样配置后， WebSocket 无法正常运行；

【2】应对不支持 WebSocket 的场景（引入 SockJS）

1）problem+solutions：

> *1.1）problem： 许多浏览器不支持 WebSocket 协议；*
>
> *1.2）solutions： SockJS 是 WebSocket 技术的一种模拟。SockJS 会 尽可能对应 WebSocket API，但如果 WebSocket 技术 不可用的话，就会选择*

另外的 通信方式协议；

2）SockJS 会优先选择 WebSocket 协议，但是如果 WebSocket协议不可用的话，他就会从如下 方案中挑选最优可行方案：

1. XHR streaming
2. XDR streaming
3. iFrame event source
4. iFrame HTML file
5. XHR polling
6. XDR polling
7. iFrame XHR polling
8. JSONP polling

3）如何在 server 端配置 SockJS ： 添加 withSockJS() 方法；

```
1.
2.  @Override
3.  public void registerWebSocketHandlers(WebSocketHandlerRegistry registry
    ) {
4.   registry.addHandler(getTextHandler(), "/websocket/p2ptext").withSockJS()
    ;
5.
6.  }
```

4）客户端配置 SockJS， 想要确保 加载了 SockJS 客户端；

4.1）具体做法是 依赖于 JavaScript 模块加载器（如 require.js or curl.js） 还是简单使用 <script> 标签加载 JavaScript 库。最简单的方法是 使用 <script> 标签从 SockJS CDN 中进行加载，如下所示：

```
1. <script src="http://cdn.sockjs.org/sockjs-0.3.min.js"></script>
```

Attention）用 WebJars 解析 Web资源（可选，有兴趣的童鞋可以尝试下）

A1）在springmvc 配置中搭建一个 资源处理器，让它负责解析路径以 "webjars/**" 开头的请求，这也是 WebJars 的标准路径：

A2）在这个资源处理器 准备就绪后，我们可以在 web 页面中使用 如下的 <script> 标签加载 SockJS 库；

5）处理加载 SockJS 客户端库以外，还要修改 两行代码：

```
1. var url = 'p2ptext';
2. var sock = new SockJS(url);
```

对以上代码的分析（Analysis）：

A1）SockJS 所处理的URL 是 "http://" 或 "https://" 模式，而不是 "ws://" or "wss://" ；

A2）其他的函数如 onopen, onmessage, and onclose ，SockJS 客户端与 WebSocket 一样；

6）SockJS 为 WebSocket 提供了 备选方案。但无论哪种场景，对于实际应用来说，这种通信形式层级过低。下面看一下如何 在 WebSocket 之上使用 STOMP协议，来为浏览器 和 server间的 通信增加适当的消息语义；（干货——引入 STOMP—— Simple Text Oriented Message Protocol——面向消息的简单文本协议）

【3】使用 STOMP消息

1）intro： 如何理解 STOMP 与 WebSocket 的关系：

1.1）假设 HTTP 协议 并不存在，只能使用 TCP 套接字来 编写 web 应用，你可能认为这是一件

疯狂的 事情；

1.2）不过 幸好，我们有 HTTP协议，它解决了 web 浏览器发起请求以及 web 服务器响应请求 的细节；

1.3）直接使用 WebSocket（SockJS） 就很类似于 使用 TCP 套接字来编写 web 应用；因为没有高层协议，因此就需要我们定义应用间所发送消息的语义，还需要确保 连接的两端都能遵循这些语义；

1.4）同 HTTP 在 TCP 套接字上添加 请求-响应 模型层一样，STOMP 在 WebSocket 之上提供了 一个基于 帧的线路格式层，用来定义消息语义；（干货——STOMP 在 WebSocket 之上提供了一个基于 帧的线路格式层，用来定义消息语义）

2）STOMP 帧：该帧由命令，一个或多个 头信息 以及 负载所组成。如下就是发送 数据的一个 STOMP帧： （干货——引入了 STOMP帧格式）

```
1. SEND
2. destination:/app/marco
3. content-length:20
4.
5. {\"message\":\"Marco!\"}
```

对以上代码的分析（Analysis）：

A1）SEND：STOMP命令，表明会发送一些内容；

A2）destination：头信息，用来表示消息发送到哪里；

A3）content-length：头信息，用来表示 负载内容的 大小；

A4）空行：

A5）帧内容（负载）内容：

3）STOMP帧 信息 最有意思的是 destination头信息了： 它表明 STOMP 是一个消息协议，类似于 JMS 或 AMQP。消息会发送到 某个 目的地，这个 目的地实际上可能真的 有消息代理作为 支撑。另一方面，消息处理器 也可以监听这些目的地，接收所发送过来的消息；

【3.1】启用STOMP 消息功能

1）intro：spring 的消息功能是基于消息代理构建的，因此我们必须要配置一个 消息代理 和 其他的一些消息目的地； （干货——spring 的消息功能是基于消息代理构建的）

2）如下代码展现了 如何通过 java配置 启用基于代理的的web 消息功能；

（干货——@EnableWebSocketMessageBroker 注解的作用： 能够在 WebSocket 上启用 STOMP）

```
1. package com.spring.spittr.web;
```

```
2.
3.  import org.springframework.context.annotation.Configuration;
4.  import org.springframework.messaging.handler.annotation.MessageMappin
    g;
5.  import org.springframework.messaging.simp.config.MessageBrokerRegistry;

6.  import org.springframework.web.socket.config.annotation.AbstractWebSock
    etMessageBrokerConfigurer;
7.  import org.springframework.web.socket.config.annotation.EnableWebSocke
    tMessageBroker;
8.  import org.springframework.web.socket.config.annotation.StompEndpointR
    egistry;
9.
10. @Configuration
11. @EnableWebSocketMessageBroker
12. public class WebSocketConfig extends AbstractWebSocketMessageBrokerCo
    nfigurer {
13.     @Override
14.     public void configureMessageBroker(MessageBrokerRegistry config) {
15.         config.enableSimpleBroker("/topic", "/queue");
16.         config.setApplicationDestinationPrefixes("/app");
17.
18.
19.     }
20.
21.     @Override
22.     public void registerStompEndpoints(StompEndpointRegistry registry) {
23.         registry.addEndpoint("/hello").withSockJS();
24.
25.     }
26. }
```

对以上代码的分析（Analysis）：

A1）EnableWebSocketMessageBroker注解表明：这个配置类不仅配置了 WebSocket，还配置了基于代理的 STOMP 消息；

*A2）它重载了 registerStompEndpoints() 方法：将 "/hello" 路径 注册为 STOMP 端点。这个路径与之前发送和接收消息的目的路径有所不同，这是一个端点，客户端在订阅或发布消息到目的地址前，要连接该端点，**即 用户发送请求 url='/server/hello' 与 STOMP server 进行连接，之后再转发到 订阅url；（server== name of your springmvc project ）（干货——端点的作用——客户端在订阅或发布消息 到目的地址前，要连接该端点）***

A3）它重载了 configureMessageBroker() 方法：配置了一个 简单的消息代理。如果不重载，默认case下，会自动配置一个简单的 内存消息代理，用来处理 "/topic" 为前缀的消息。但经过重载后，消息代理将会处理前缀为 "/topic" and "/queue" 消息。

A4）之外：发送应用程序的消息将会带有 "/app" 前缀，下图展现了 这个配置中的 消息流；

Spring简单的 STOMP代理是基于内存的，它模拟了 STOMP 代理的多项功能。

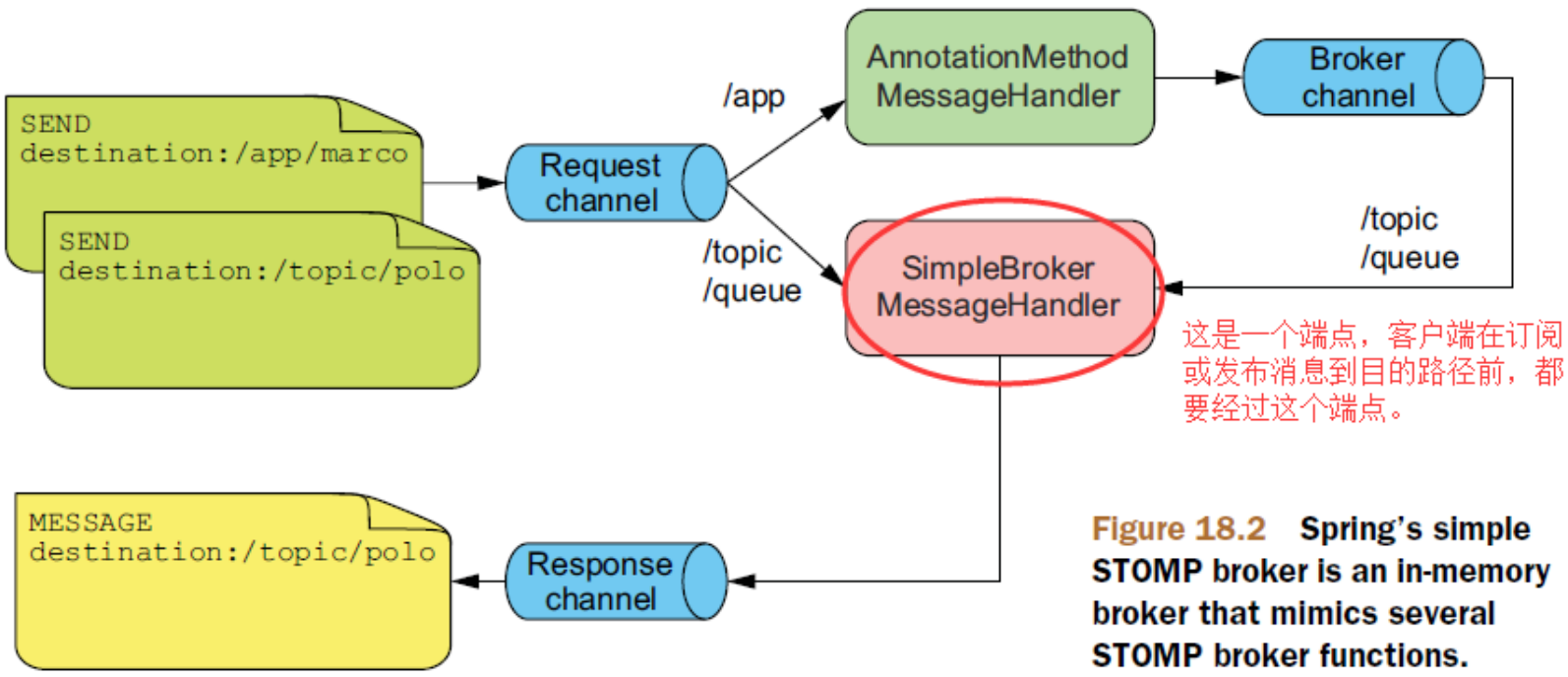


Figure 18.2 Spring's simple STOMP broker is an in-memory broker that mimics several STOMP broker functions.

对上述处理step的分析（Analysis）：

> *A1）应用程序的目的地 以 "/app" 为前缀，而代理的目的地以 "/topic" 和 "/queue" 作为前缀；*

> *A2）以应用程序为目的地的消息将会直接路由到 带有 @MessageMapping 注解的控制器方法中；（**干货—— @MessageMapping的作用**）*

> *A3）而发送到 代理上的消息，包括 @MessageMapping注解方法的返回值所形成的消息，将会路由到 代理上，并最终发送到 订阅这些目的地客户端；*
>
> *（**干货——client 连接地址和 发送地址是不同的，以本例为例，前者是/server/hello, 后者是/server/app/XX，先连接后发送**）*

【3.1.1】启用 STOMP 代理中继

1）**intro**：在生成环境下，可能会希望使用 真正支持 STOMP 的代理来支持 WebSocket 消息，如RabbitMQ 或 ActiveMQ。这样的代理提供了可扩展性和健壮性更好的消息功能，当然，他们也支持 STOMP 命令；

2）如何 使用 STOMP 代理来替换内存代理，代码如下：

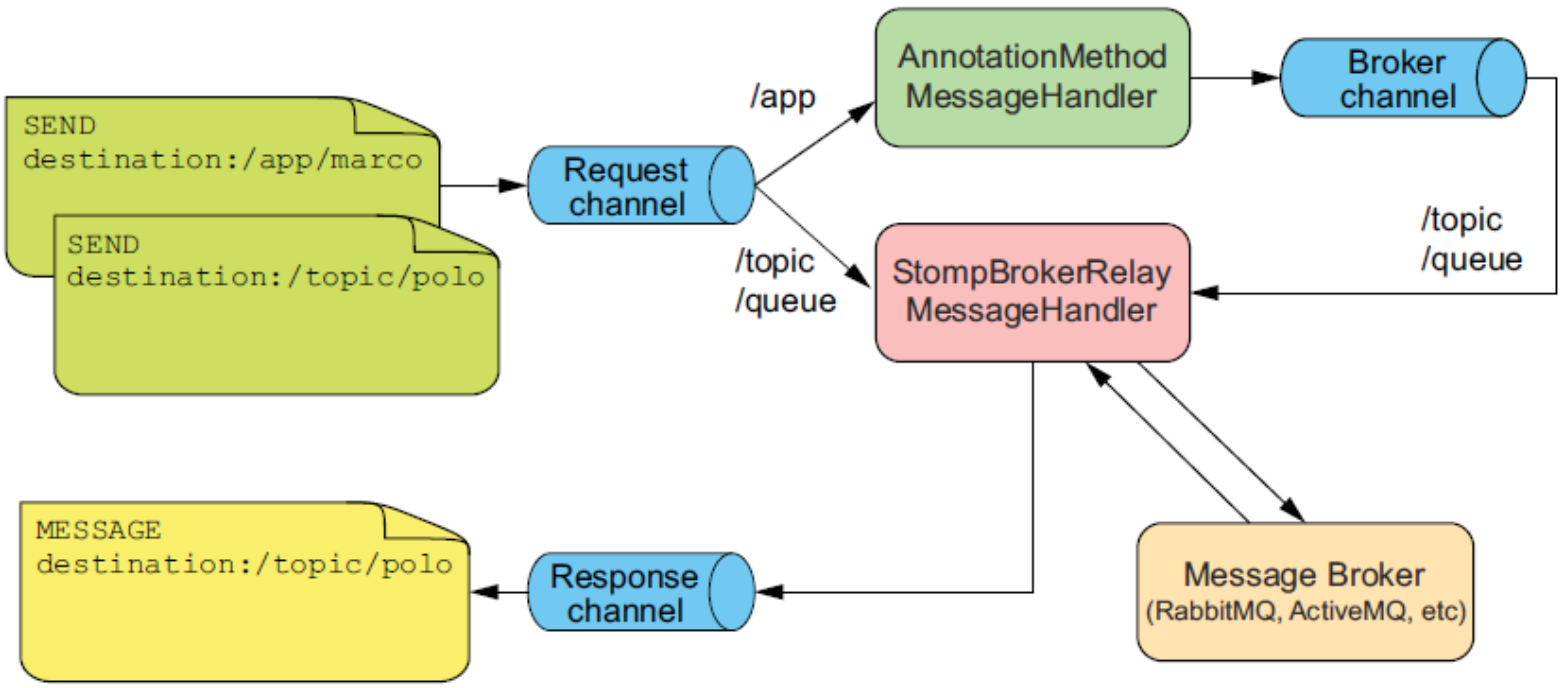
```
1. @Override
2.  public void configureMessageBroker(MessageBrokerRegistry registry) {
3.
4.   registry.enableStompBrokerRelay("/queue", "/topic")
5.    .setRelayPort(62623);
6.   registry.setApplicationDestinationPrefixes("/app");
7.  }
```

对以上代码的分析（Analysis）：（干货——STOMP代理前缀和 应用程序前缀的意义）

A1）方法第一行启用了 STOMP 代理中继功能： *并将其目的地前缀设置为 "/topic" or "/queue"；spring就能知道 所有目的地前缀为 "/topic" or "/queue" 的消息都会发送到 STOMP 代理中；*

A2）方法第二行设置了 应用的前缀为 "app": *所有目的地以 "/app" 打头的消息（**发送消息url not 连接url**）都会路由到 带有 @MessageMapping 注解的方法中，而不会发布到 代理队列或主题中；*

3）下图阐述了 代理中继如何 应用于 spring 的 STOMP 消息处理之中。与 上图的 关键区别在于： 这里不再模拟STOMP 代理的功能，而是由 代理中继将消息传送到一个 真正的消息代理来进行处理；



STOMP 代理中会将 STOMP 消息的处理委托给一个真正的消息代理 来进行处理。

Figure 18.3 The STOMP broker relay delegates to a real message broker for handling STOMP messages.

Attention)

> A1）enableStompBrokerRelay() and setApplicationDestinationPrefixes() 方法都可以接收变长 参数；
>
> A2）默认情况下： STOMP 代理中继会假设 代理监听 localhost 的61613 端口，并且 client 的 username 和password 均为 guest。当然你也可以自行定义；

【3.2】 处理来自客户端的 STOMP 消息

1）借助 @MessageMapping 注解能够 在 控制器中处理 STOMP 消息

```
1. package com.spring.spittr.web;
2.
3. import org.springframework.messaging.handler.annotation.MessageMappin
   g;
4. import org.springframework.messaging.handler.annotation.SendTo;
5. import org.springframework.stereotype.Controller;
6.
7. import com.spring.pojo.Greeting;
```

```
8.  import com.spring.pojo.HelloMessage;
9.
10. @Controller
11. public class GreetingController {
12.
13.     @MessageMapping("/hello")
14.     @SendTo("/topic/greetings")
15.     public Greeting greeting(HelloMessage message) throws Exception {
16.         System.out.println("receiving " + message.getName());
17.         System.out.println("connecting successfully.");
18.         return new Greeting("Hello, " + message.getName() + "!");
19.     }
20. }
```

对以上代码的分析（Analysis）：

A1）@MessageMapping注解：表示 handleShout()方法能够处理 指定目的地上到达的消息；

A2）这个目的地（消息发送目的地url）就是 "/server/app/hello"，其中 "/app" 是 隐含的 ,"/server" 是 springmvc 项目名称；

2）因为我们现在处理的 不是 HTTP，所以无法使用 spring 的 HttpMessageConverter 实现 将负载转换为Shout 对象。Spring 4.0 提供了几个消息转换器如下：（Attention，如果是传输json数据的话，定要添加 Jackson jar 包到你的springmvc 项目中，不然连接不会成功的）

spring 能够使用某个消息转换器将 消息负载转换为 java 类型；

Table 18.1 Spring can convert message payloads to Java types using one of a few message converters.

Message converter	Description
`ByteArrayMessageConverter`	Converts a message with a MIME type of `application/octet-stream` to and from `byte[]`
`MappingJackson2MessageConverter`	Converts a message with a MIME type of `application/json` to and from a Java object
`StringMessageConverter`	Converts a message with a MIME type of `text/plain` to and from `String`

【3.2.1】处理订阅（@SubscribeMapping注解）

1）@SubscribeMapping注解 的方法：当收到 STOMP 订阅消息的时候，带有 @SubscribeMapping 注解 的方法将会触发；其也是通过 AnnotationMethodMessageHandler 来接收消息的；

2）@SubscribeMapping注解的应用场景：实现 请求-回应模式。在请求-回应模式中，客户端订阅一个目的地，然后预期在这个目的地上 获得一个一次性的 响应；（干货——引入了@SubsribeMapping注解实现 请求-回应模式）

2.1）看个荔枝：

```
1. @SubscribeMapping({"/marco"})
2. public Shout handleSubscription() {
3. Shout outgoing = new Shout();
4. outgoing.setMessage("Polo!");
5. return outgoing;
6. }
```

对以上代码的分析（Analysis）：

A1）@SubscribeMapping注解 的方法来处理 对 "/app/macro" 目的地订阅（与 @MessageMapping类似，"/app" 是隐含的）；

A2）请求-回应模式与 HTTP GET 的全球-响应模式差不多：关键区别在于，HTTP GET 请求是同步的，而订阅的全球-回应模式是异步的，这样客户端能够在回应可用时再去处理，而不必等待；（干货——HTTP GET 请求是同步的，而订阅的请求-回应模式是异步的）

【3.2.2】编写 JavaScript 客户端

1）intro：借助 STOMP 库，通过 JavaScript发送消息

```
<script type="text/javascript">
        var stompClient = null;

        function setConnected(connected) {
            document.getElementById('connect').disabled = connected;
            document.getElementById('disconnect').disabled = !connected;
            document.getElementById('conversationDiv').style.visibility = conne
cted ? 'visible' : 'hidden';
            document.getElementById('response').innerHTML = '';
        }

        function connect() {
            var socket = new SockJS("<c:url value='/hello'/>");
            stompClient = Stomp.over(socket);
            stompClient.connect({}, function(frame) {
                setConnected(true);
                console.log('Connected: ' + frame);
                stompClient.subscribe('/topic/greetings', function(greeting){
                    showGreeting(JSON.parse(greeting.body).content);
                });
```

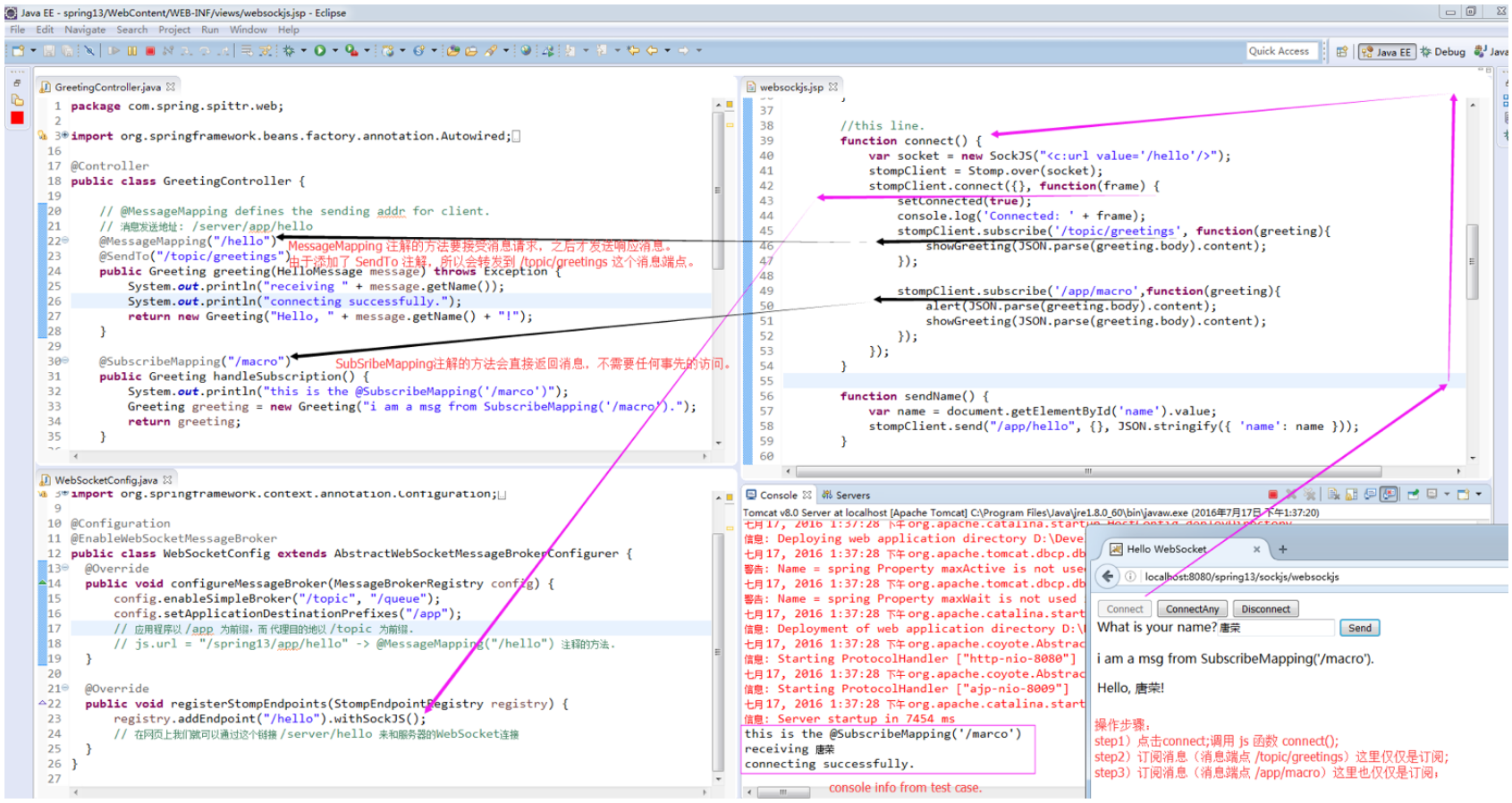
```
20.             });
21.         }
22.
23.         function disconnect() {
24.             if (stompClient != null) {
25.                 stompClient.disconnect();
26.             }
27.             setConnected(false);
28.             console.log("Disconnected");
29.         }
30.
31.         function sendName() {
32.             var name = document.getElementById('name').value;
33.             stompClient.send("/app/hello", {}, JSON.stringify({ 'name': name })
    );
34.         }
35.
36.         function showGreeting(message) {
37.             var response = document.getElementById('response');
38.             var p = document.createElement('p');
39.             p.style.wordWrap = 'break-word';
40.             p.appendChild(document.createTextNode(message));
41.             response.appendChild(p);
42.         }
43.     </script>
```

对以上代码的 分析（Analysis）： 以上代码连接“/hello” 端点并发送 ”name“；

2）stompClient.send("/app/hello", {}, JSON.stringify({'name':name}))： 第一个参数：json 负载消息发送的 目的地； 第二个参数：是一个头信息的Map，它会包含在 STOMP 帧中；第三个参数：负载消息；

（干货—— stomp client 连接地址 和 发送地址不一样的，连接地址为 <c:url value='/hello'/> ==localhost:8080/springmvc_project_name/hello , 而 发送地

址为 '/app/hello'，这里要当心）



1. <script src="<c:url value="/resources/sockjs-1.1.1.js" />"></script>
2. <script src="<c:url value="/resources/stomp.js" />"></script>

```
1.
2.      function connect() {
3.          var socket = new SockJS("<c:url value='/hello'/>");
4.          stompClient = Stomp.over(socket);
5.          stompClient.connect({}, function(frame) {
6.              setConnected(true);
7.              console.log('Connected: ' + frame);
8.              stompClient.subscribe('/topic/greetings', function(greeting){
9.                  showGreeting(JSON.parse(greeting.body).content);
10.             });
11.
12.             stompClient.subscribe('/app/macro',function(greeting){
13.                 alert(JSON.parse(greeting.body).content);
14.                 showGreeting(JSON.parse(greeting.body).content);
15.             });
16.         });
```

```
17.         }
18.
19.         function sendName() {
20.             var name = document.getElementById('name').value;
21.             stompClient.send("/app/hello", {}, JSON.stringify({ 'name': name })
    );
22.         }
```

```
1.  package com.spring.spittr.web;
2.
3.  import org.springframework.beans.factory.annotation.Autowired;
4.  import org.springframework.messaging.handler.annotation.MessageMappin
    g;
5.  import org.springframework.messaging.handler.annotation.SendTo;
6.  import org.springframework.messaging.simp.SimpMessageSendingOperatio
    ns;
7.  import org.springframework.messaging.simp.SimpMessagingTemplate;
8.  import org.springframework.messaging.simp.annotation.SubscribeMapping;
9.  import org.springframework.stereotype.Controller;
10. import org.springframework.web.bind.annotation.RequestMapping;
11. import org.springframework.web.bind.annotation.RequestMethod;
12. import org.springframework.web.bind.annotation.RequestParam;
13.
14. import com.spring.pojo.Greeting;
15. import com.spring.pojo.HelloMessage;
16.
17. @Controller
18. public class GreetingController {
19.
20.
21.
22.     @MessageMapping("/hello")
23.     @SendTo("/topic/greetings")
24.     public Greeting greeting(HelloMessage message) throws Exception {
```

```
25.         System.out.println("receiving " + message.getName());
26.         System.out.println("connecting successfully.");
27.         return new Greeting("Hello, " + message.getName() + "!");
28.     }
29.
30.     @SubscribeMapping("/macro")
31.     public Greeting handleSubscription() {
32.         System.out.println("this is the @SubscribeMapping('/marco')");
33.         Greeting greeting = new Greeting("i am a msg from SubscribeMapping
    ('/macro').");
34.         return greeting;
35.     }
36.
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.     private SimpMessageSendingOperations  template;
48.
49.     @Autowired
50.     public GreetingController(SimpMessageSendingOperations  template) {
51.         this.template = template;
52.     }
53.
54.     @RequestMapping(path="/feed", method=RequestMethod.POST)
55.     public void greet(
56.             @RequestParam String greeting) {
57.         String text = "you said just now " + greeting;
```

```
58.         this.template.convertAndSend("/topic/feed", text);
59.     }
60. }
```

```
1. package com.spring.spittr.web;
2.
3. import org.springframework.context.annotation.Configuration;
4. import org.springframework.messaging.handler.annotation.MessageMappin
   g;
5. import org.springframework.messaging.simp.config.MessageBrokerRegistry;

6. import org.springframework.web.socket.config.annotation.AbstractWebSock
   etMessageBrokerConfigurer;
7. import org.springframework.web.socket.config.annotation.EnableWebSocke
   tMessageBroker;
8. import org.springframework.web.socket.config.annotation.StompEndpointR
   egistry;
9.
10. @Configuration
11. @EnableWebSocketMessageBroker
12. public class WebSocketConfig extends AbstractWebSocketMessageBrokerCo
    nfigurer {
13.   @Override
14.   public void configureMessageBroker(MessageBrokerRegistry config) {
15.       config.enableSimpleBroker("/topic", "/queue");
16.       config.setApplicationDestinationPrefixes("/app");
17.
18.
19.   }
20.
21.   @Override
22.   public void registerStompEndpoints(StompEndpointRegistry registry) {
23.       registry.addEndpoint("/hello").withSockJS();
24.
25.   }
```

```
}
```

```
package com.spring.spittr.web;

import java.io.IOException;

import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.support.ResourceBundleMessageSource
;
import org.springframework.core.io.FileSystemResource;
import org.springframework.web.multipart.MultipartResolver;
import org.springframework.web.multipart.commons.CommonsMultipartRe
solver;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.DefaultServletH
andlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.ViewControllerR
egistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigur
erAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver
;
import org.springframework.web.servlet.view.tiles3.TilesConfigurer;
import org.springframework.web.servlet.view.tiles3.TilesViewResolver;

@Configuration
@ComponentScan(basePackages = { "com.spring.spittr.web" })
@EnableWebMvc
@Import({WebSocketConfig.class})
```

```
27. public class WebConfig extends WebMvcConfigurerAdapter {
28.
29.     @Bean
30.     public TilesConfigurer tilesConfigurer() {
31.         TilesConfigurer tiles = new TilesConfigurer();
32.         tiles.setDefinitions(new String[] { "/WEB-INF/layout/tiles.xml" });
33.         tiles.setCheckRefresh(true);
34.         return tiles;
35.     }
36.
37.
38.     @Override
39.     public void configureDefaultServletHandling(
40.             DefaultServletHandlerConfigurer configurer) {
41.         configurer.enable();
42.     }
43.
44.
45.     @Bean
46.     public ViewResolver viewResolver1() {
47.         TilesViewResolver resolver = new TilesViewResolver();
48.         return resolver;
49.     }
50.
51.     @Bean
52.     public ViewResolver viewResolver2() {
53.         InternalResourceViewResolver resolver = new InternalResourceViewR
    esolver();
54.         resolver.setPrefix("/WEB-INF/views/");
55.         resolver.setSuffix(".jsp");
56.         resolver.setExposeContextBeansAsAttributes(true);
57.         resolver.setViewClass(org.springframework.web.servlet.view.JstlView.cl
    ass);
58.         return resolver;
```

```
59.     }
60.
61.     @Bean
62.     public MessageSource messageSource() {
63.         ResourceBundleMessageSource messageSource = new ResourceBundle
    MessageSource();
64.         messageSource.setBasename("messages");
65.         return messageSource;
66.     }
67.
68.     @Bean
69.     public MultipartResolver multipartResolver() throws IOException {
70.         CommonsMultipartResolver multipartResolver = new CommonsMulti
    partResolver();
71.         multipartResolver.setUploadTempDir(new FileSystemResource("/WEB
    -INF/tmp/spittr/uploads"));
72.         multipartResolver.setMaxUploadSize(2097152);
73.         multipartResolver.setMaxInMemorySize(0);
74.         return multipartResolver;
75.     }
76. }
```

【3.3】发送消息到客户端

1）intro：spring提供了两种 发送数据到 client 的方法：

method1）作为处理消息 或处理订阅的附带结果；

method2）使用消息模板；

【3.3.1】在处理消息后，发送消息（server 对 client 请求的 响应消息）

1）intro：如果你想要在接收消息的时候，在响应中发送一条消息，修改方法签名 不是void 类型即可， 如下：

```
1. @MessageMapping("/hello")
2.     @SendTo("/topic/greetings")
3.     public Greeting greeting(HelloMessage message) throws Exception {
4.         System.out.println("receiving " + message.getName());
5.         System.out.println("connecting successfully.");
6.         return new Greeting("Hello, " + message.getName() + "!");
7.     }
```

对以上代码的分析（Analysis）： 返回的对象将会进行转换（通过消息转换器） 并放到 STOMP 帧的负载中，**然后发送给消息代理（消息代理分为STOMP代理中继 和 内存消息代理）**；

2）默认情况下：帧所发往的目的地会与 触发 处理器方法的目的地相同。所以返回的对象 会写入到 STOMP 帧的负载中，并发布到 "/topic/stomp" 目的地。不过，可以通过 @SendTo 注解，重载目的地；（干货——注解 @SendTo 注解的作用）

代码同上。

对以上代码的分析（Analysis）： 消息将会发布到 /topic/hello， 所有订阅这个主题的应用都会收到这条消息；

3）@SubscriptionMapping 注解标注的方式也能发送一条消息，作为订阅的回应。

3.1）看个荔枝： 通过为 控制器添加如下的方法，当客户端订阅的时候，将会发送一条 shout 信息：

```
1. @SubscribeMapping("/macro")
2.     public Greeting handleSubscription() {
3.         System.out.println("this is the @SubscribeMapping('/marco')");
4.         Greeting greeting = new Greeting("i am a msg from SubscribeMapping
   ('/macro').");
5.         return greeting;
```

```
6.      }

1.  function connect() {
2.          var socket = new SockJS("<c:url value='/hello'/>");
3.          stompClient = Stomp.over(socket);
4.          stompClient.connect({}, function(frame) {
5.              setConnected(true);
6.              console.log('Connected: ' + frame);
7.              stompClient.subscribe('/topic/greetings', function(greeting){
8.                  showGreeting(JSON.parse(greeting.body).content);
9.              });
10.
11.             stompClient.subscribe('/app/macro',function(greeting){
12.                 alert(JSON.parse(greeting.body).content);
13.                 showGreeting(JSON.parse(greeting.body).content);
14.             });
15.         });
16.     }
```

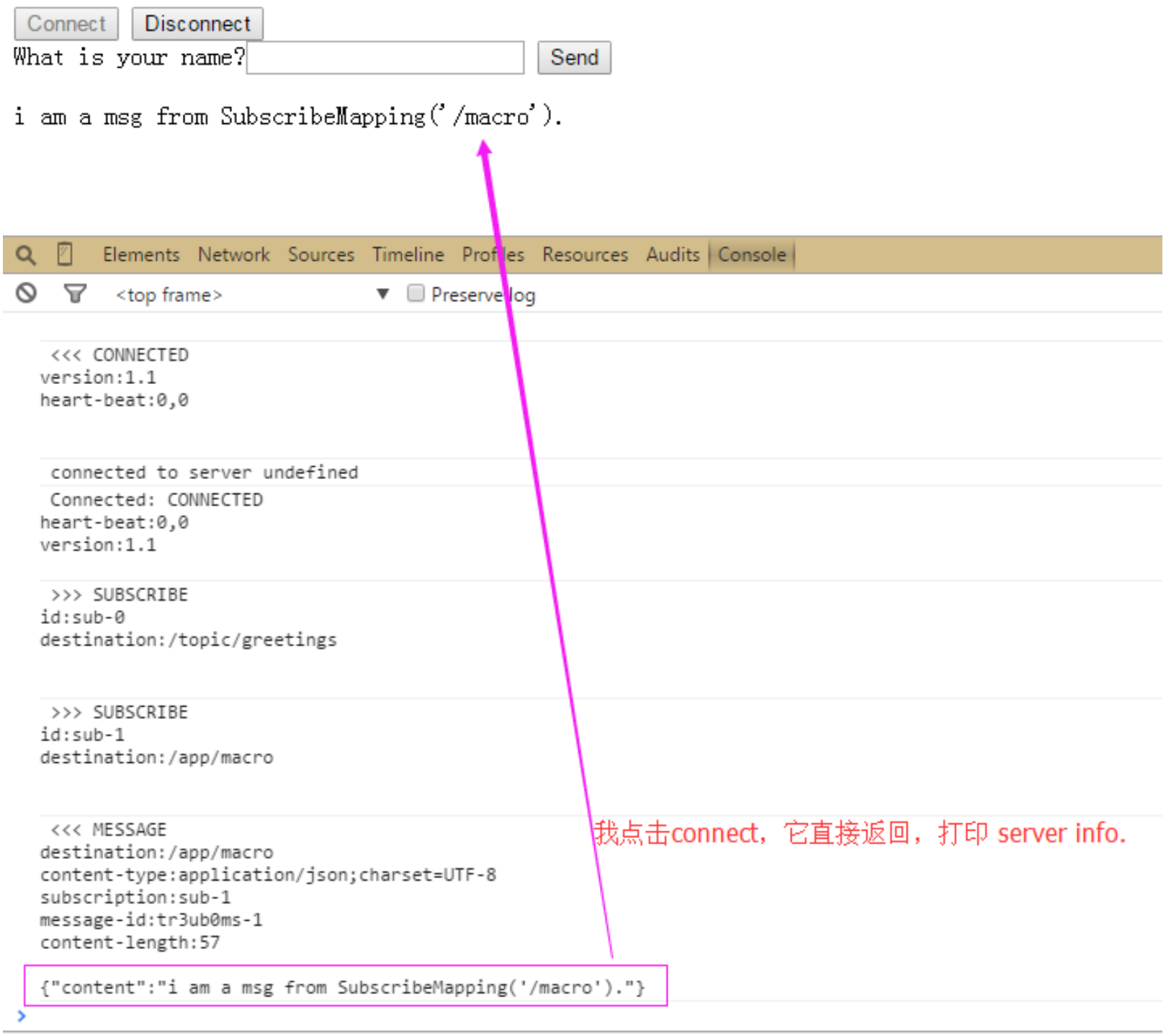
对以上代码的分析（Analysis）：

A0）这个SubscribeMapping annotation标记的方法，是在订阅的时候调用的，也就是说，基本是只执行一次的方法，client 调用定义在server 的 该 Annotation 标注的方法，它就会返回结果，不过经过代理。

*A1）这里的 @SubscribeMapping 注解表明当 客户端订阅 "/app/macro" 主题的时候（**"/app"是应用目的地的前缀，注意，这里没有加 springmvc 项目名称前缀**），将会调用 handleSubscription 方法。它所返回的shout*

对象 将会进行转换 并发送回client；

A2）SubscribeMapping注解的区别在于： *这里的 Shout 消息将会直接发送给 client，不用经过消息代理；但，如果为方法添加 @SendTo 注解的话，那么 消息将会发送到指定的目的地，这样就会经过代理；* ***（干货——SubscribeMapping 注解返回的消息直接发送到 client，不经过代理，而 @SendTo 注解的路径，就会经过代理，然后再发送到 目的地）***



【3.3.2】 在应用的任意地方发送消息

1）**intro**：spring 的 SimpMessagingTemplate 能够在应用的任何地方发送消息，不必以接收一条消息为 前提；

2）**看个荔枝**： 让首页订阅一个 STOMP主题，在 Spittle 创建的时候，该主题能够收到 Spittle 更新时的 feed；

2.1）**JavaScript 代码**：

```
1. <script>
2. var sock = new SockJS('spittr');
3. var stomp = Stomp.over(sock);
4. stomp.connect('guest', 'guest', function(frame) {
5. console.log('Connected');
6. stomp.subscribe("/topic/spittlefeed", handleSpittle);
7. });
8. function handleSpittle(incoming) {
9. var spittle = JSON.parse(incoming.body);
10. console.log('Received: ', spittle);
11. var source = $("#spittle-template").html();
12. var template = Handlebars.compile(source);
13. var spittleHtml = template(spittle);
14. $('.spittleList').prepend(spittleHtml);
15. }
16. </script>
```

对以上代码的分析（Analysis）： 在连接到 STMOP 代理后，我们订阅了 "/topic/spittlefeed" 主题，并指定当消息到达的是，由 handleSpittle()函数来处理 Spittle 更新。

2.2） **server 端代码：使用 SimpMessagingTemplate 将所有新创建的 Spittle 以消息的形式发布到 "/topic/feed" 主题上；**

```
1. @Service
2. public class SpittleFeedServiceImpl implements SpittleFeedService {
3. private SimpMessageSendingOperations messaging;
```

```
4. @Autowired
5. public SpittleFeedServiceImpl(
6. SimpMessageSendingOperations messaging) {
7. this.messaging = messaging;
8. }
9. public void broadcastSpittle(Spittle spittle) {
10. messaging.convertAndSend("/topic/spittlefeed", spittle);
11. }
12. }
```

对以上代码的分析（Analysis）：

A1）配置 spring 支持 stomp 的一个附带功能是 在spring应用上下文中已经包含了 Simple

A2）在发布消息给 STOMP 主题的时候，所有订阅该主题的客户端都会收到消息。但有的时候，我们希望将消息发送给指定用户；

【4】为目标用户发送消息

1）intro：在使用 srping 和 STOMP 消息功能的时候，有三种方式来利用认证用户：

way1）@MessageMapping and @SubscribeMapping 注解标注的方法 能够使用 Principal 来获取认证用户；

way2）@MessageMapping, @SubscribeMapping, and @MessageException 方法返回的值能够以 消息的形式发送给 认证用户；

way3）SimpMessagingTemplate 能够发送消息给

特定用户；

【4.1】在控制器中处理用户的 消息

1）看个荔枝： 编写一个控制器方法，根据传入的消息创建新的Spittle 对象，并发送一个回应，表明 对象创建成功；**（这种 REST也可以实现，不过它是同步的，而这里是异步的）；**

1.1）代码如下：它会处理传入的消息并将其存储我 Spittle：

1.2）该方法最后返回一个 新的 Notificatino，表明对象保存成功；

1.3）该方法使用了 @MessageMapping("/spittle") 注解，所以当有发往 "/app/spittle" 目的地的消息 到达时，该方法就会触发；如果用户已经认证的话，将会根据 STOMP 帧上的头信息得到 Principal 对象；

1.4）@SendToUser注解： 指定了 Notification 要发送的 目的地 "/queue/notifications";

1.5）表明上， "/queue/notifications" 并不会与特定用户相关联，但因为 这里使用的是 @SendToUser注解， 而不是 @SendTo，所以就会发生更多的事情了；

2）看一下针对 控制器方法发布的 Notificatino 对象的目的地，客户端该如何进行订阅。

2.1）看个荔枝：考虑如下的 JavaScript代码，它订阅了一个 用户特定的 目的地：

对以上代码的分析（Analysis）：这个目的地使用了 "/user" 作为前缀，在内部，以"/user" 为前缀的消息将会通过 UserDestinationMessageHandler 进行处理，而不是 AnnotationMethodMessageHandler 或 SimpleBrokerMessageHandler or StompBrokerRelayMessageHandler，如下图所示：

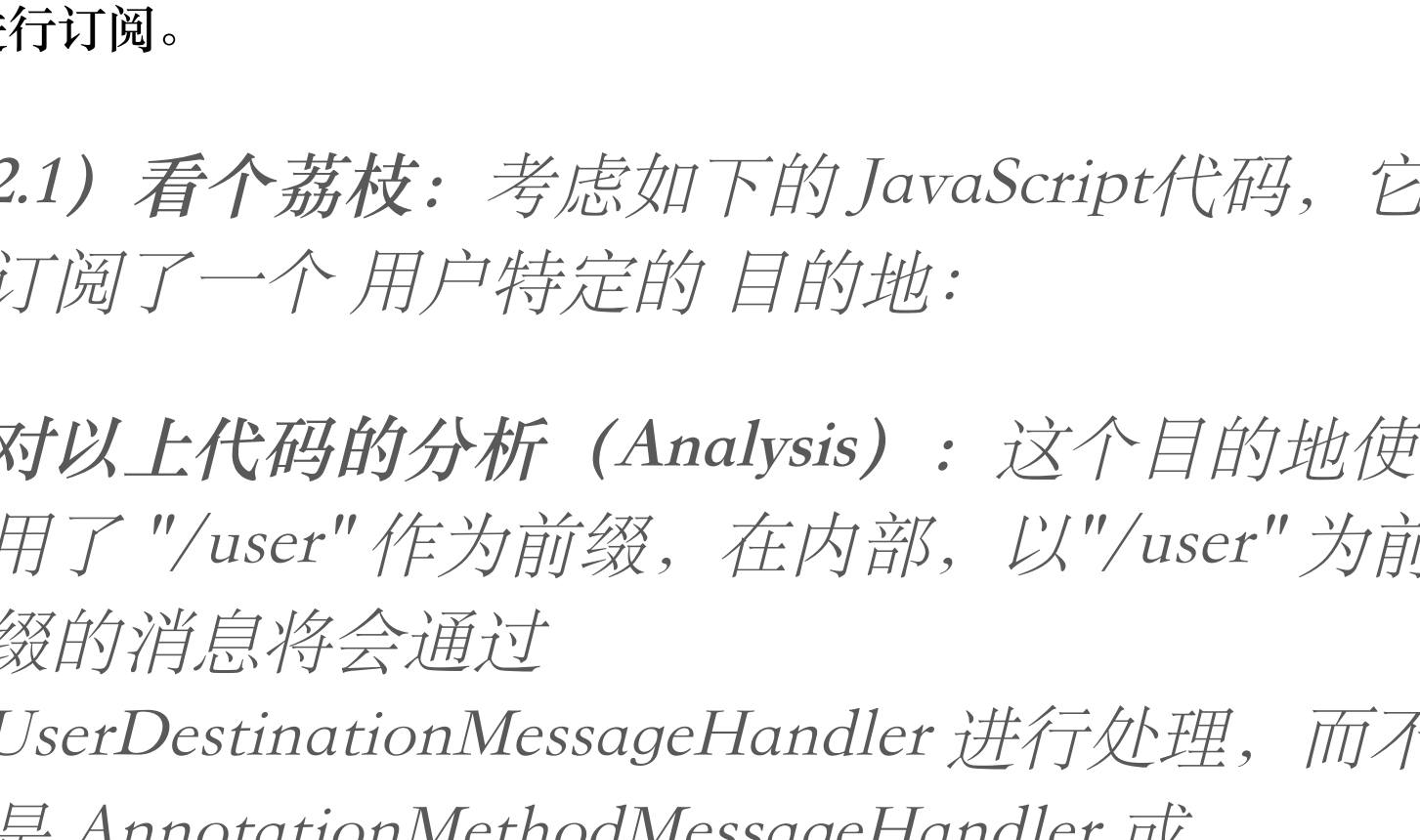


Figure 18.4 User messages flow through `UserDestinationMessageHandler`, which reroutes them to a destination that's unique to a user.

用户消息流会通过 UserDestinationMessageHandler 进行处理，它会将消息重路由到某个用户独有的目的地上。

Attention） UserDestinationMessageHandler 的主

要任务：是 将用户消息重新路由到 某个用户独有的目的地上。在处理订阅的时候，它会将目标地址中的 "/user" 前缀去掉，并基于用户 的会话添加一个后缀。如，对 "/user/queue/notifications" 的订阅最后可能路由到 名为 "/queue/notifacations-user65a4sdfa" 目的地上；

【4.2】为指定用户发送消息

1）intro：SimpMessagingTemplate还提供了 convertAndSendToUser() 方法，该方法能够让 我们给特定用户发送消息；

2）我们在 web 应用上添加一个特性：当其他用户提交的 Spittle 提到某个用户时，将会提醒该用户（干货——这难道不是 微博的 @ 功能吗）

2.1）看个荔枝：如果Spittle 文本中包含 "@tangrong"，那么我们就应该发送一条消息给 使用 tangrong 用户名登录的client，代码实例如下：

```
1.  @Service
2.  public class SpittleFeedServiceImpl implements SpittleFeedService {
3.  private SimpMessagingTemplate messaging;
4.
5.  private Pattern pattern = Pattern.compile("\\@(\\S+)");
6.
7.  @Autowired
8.  public SpittleFeedServiceImpl(SimpMessagingTemplate messaging) {
9.  this.messaging = messaging;
10. }
11. public void broadcastSpittle(Spittle spittle) {
12. messaging.convertAndSend("/topic/spittlefeed", spittle);
13. Matcher matcher = pattern.matcher(spittle.getMessage());
```

```
14. if (matcher.find()) {
15. String username = matcher.group(1);
16.
17. messaging.convertAndSendToUser(
18. username, "/queue/notifications",
19. new Notification("You just got mentioned!"));
20. }
21. }
22. }
```

【5】处理消息异常

1）intro：我们也可以在 控制器方法上添加 @MessageExceptionHandler 注解，让它来处理 @MessageMapping 方法所抛出的异常；

2）看个荔枝：它会处理 消息方法所抛出的异常；

```
1. @MessageExceptionHandler
2. public void handleExceptions(Throwable t) {
3. logger.error("Error handling message: " + t.getMessage());
4. }
```

3）我们也可以以 参数的形式声明它所能处理的异常；

```
1. @MessageExceptionHandler(SpittleException.class)
2. public void handleExceptions(Throwable t) {
3. logger.error("Error handling message: " + t.getMessage());
4. }
5.
6. @MessageExceptionHandler( {SpittleException.class, DatabaseException.class})
7. public void handleExceptions(Throwable t) {
8. logger.error("Error handling message: " + t.getMessage());
9. }
```

4）该方法还可以回应一个错误：

```
1. @MessageExceptionHandler(SpittleException.class)
2. @SendToUser("/queue/errors")
3. public SpittleException handleExceptions(SpittleException e) {
4. logger.error("Error handling message: " + e.getMessage());
5. return e;
6. }
7.
8.
```