1 使用Spring的低层级WebSocket API

为了在Spring使用底层的API来处理消息，需要编写一个实现WebSockerHandler的类：

```
public interface WebSocketHandler {
    void afterConnectionEstablished(WebSocketSession var1) throws Exception;

    void handleMessage(WebSocketSession var1, WebSocketMessage<?> var2) throws
Exception;

    void handleTransportError(WebSocketSession var1, Throwable var2) throws
Exception;

    void afterConnectionClosed(WebSocketSession var1, CloseStatus var2) throws
Exception;

    boolean supportsPartialMessages();
}
```

需要实现5个方法，更简单的方法是扩展AbstractWebSocketHandler。

```
public class MarcoHandler extends AbstractWebSocketHandler {

    private static final Logger logger =
LoggerFactory.getLogger(MarcoHandler.class);

    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage
message) throws Exception {
        logger.info("Received message: " + message.getPayload());
        Thread.sleep(2000);//模拟时延
        session.sendMessage(new TextMessage("Polo!"));发送文本消息
    }

}
```

MarcoHandler没有重载的方法都由AbstractWebSocketHandler以空操作的方式进行了实现。

此外还可以扩展TextWebSocketHandler，TextWebSocketHandler是AbstractWebSocketHandler的子类，他会拒绝处理二进制消息，如果收到二进制消息，将会关闭连接

```
@Override
public void afterConnectionEstablished(WebSocketSession session) throws
Exception {
    super.afterConnectionEstablished(session);
}

@Override
public void afterConnectionClosed(WebSocketSession session, CloseStatus status)
throws Exception {
    super.afterConnectionClosed(session, status);
}
```

1.1.1 在Java配置中，启用WebSocket并映射消息处理器

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(marcoHandler(), "/marco");
    }

    @Bean
    public MarcoHandler marcoHandler() {
        return new MarcoHandler();
    }
}
```

1.1.2 借助websocket命名空间以XML的方式配置WebSocket

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:websocket="http://www.springframework.org/schema/websocket"
       xsi:schemaLocation="
  http://www.springframework.org/schema/websocket
  http://www.springframework.org/schema/websocket/spring-websocket.xsd
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

    <websocket:handlers>
        <websocket:mapping handler="marcoHandler" path="/marco"/>
    </websocket:handlers>

    <bean id="marcoHandler" class="marcopolo.MarcoHandler"/>

</beans>
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<!--<head>-->
<!--    <title>Home</title>-->
<!--    <script th:src="@{/webjars/sockjs-client/0.3.4/sockjs.min.js}">
</script>-->
<!--    <script th:src="@{/webjars/jquery/2.0.3/jquery.min.js}"></script>-->
<!--</head>-->
<body>
<button id="stop">Stop</button>

<script th:inline="javascript">
    var sock = new WebSocket("ws://localhost:8080/marco")
    // var sock = new SockJS(@{marco});

    sock.onopen = function () {
        console.log('Opening');
        sayMarco();
    }
```

```
    sock.onmessage = function (e) {
        console.log('Received message: ', e.data);
        // $('#output').append('Received "' + e.data + '"<br/>');
        setTimeout(function () {
            sayMarco()
        }, 2000);
    }

    sock.onclose = function () {
        sock.close();
        console.log('Closing');
    }

    function sayMarco() {
        console.log('Sending Marco!');
        // $('#output').append('Sending "Marco!"<br/>');
        sock.send("Marco!");
    }

    // $('#stop').click(function () {
    //     sock.close()
    // });
</script>

<div id="output"></div>
</body>
</html>
```

“ws://” 前缀表明这是一个基本的WebSocket

“wss://” 前缀则是安全WebSocket

2 应对不支持WebSocket的场景

WebSocket备选方案--SocketJS，SocketJS是WebSocket技术的一助攻模拟，表面上它尽可能对应WebSocket API，但是在底层它非常智能，如果WebSOcket技术不可用，他就会选择另外的通讯方式。

2.1.1 启用SocketJS

```
@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(marcoHandler(), "/marco").withSockJS();
}
```

withSockJS方法声明想要是哟共SockJS功能，如果WebSocket不可用，SockJS的备选方案就会发挥作用

```
<websocket:handlers>
    <websocket:mapping handler="marcoHandler" path="/marco"/>
    <websocket:sockjs/>
</websocket:handlers>
```

要在客户端是哟共SockJS，需要确保加载了SockJS客户端库。依赖于是哟共JaaScript模块加载器还是简单地使用标签加载JavaScript库。最简单的是使标签从SockJS CDN中加载：

2.1.2 用WebJars解析web资源

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/webjars/**")
            .addResourceLocations("classpath:/META-INF/resources/webjars/");
}
```

在这个资源处理器准备就绪后，可以在Web页面中使用如下标签加载SocketJS库

```
<script th:src="@{/webjars/sockjs-client/0.3.4/sockjs.min.js}"></script>
```

这个<script标签来源于一个Thymeleaf模板，使用@{...}表达式来为JavaScript文件计算完整的相对于上下文的URL路径

只需要修改两个地方

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Home</title>
    <script th:src="@{/webjars/sockjs-client/1.1.2/sockjs.min.js}"></script>
    <script th:src="@{/webjars/jquery/3.5.1/jquery.min.js}"></script>
</head>
<body>
<button id="stop">Stop</button>

<script th:inline="javascript">
    // var sock = new WebSocket("ws://localhost:8080/marco")
    // var sock = new SockJS(@{marco});

    var url = 'marco';
    var sock = new SockJS(url);

    sock.onopen = function () {
        console.log('Opening');
        sayMarco();
    }

    sock.onmessage = function (e) {
        console.log('Received message: ', e.data);
        // $('#output').append('Received "' + e.data + '"<br/>');
        setTimeout(function () {
            sayMarco()
        }, 2000);
    }

    sock.onclose = function () {
        sock.close();
        console.log('Closing');
    }

    function sayMarco() {
        console.log('Sending Marco!');
        // $('#output').append('Sending "Marco!"<br/>');
        sock.send("Marco!");
    }
```

```
    // $('#stop').click(function () {
    //     sock.close()
    // });
</script>

<div id="output"></div>
</body>
</html>
```

SockJS所处理的URL是"http://" 或"https://"，而不是“ws:”和“wss:”

WebSocket提供了浏览器-服务器之间的通讯方式，当运行环境不支持WebSOcket的时候，SOckJS提供了备用方案。

3 使用STOMP

STOMP(Siplet Text Oriented Messaging Protocol)

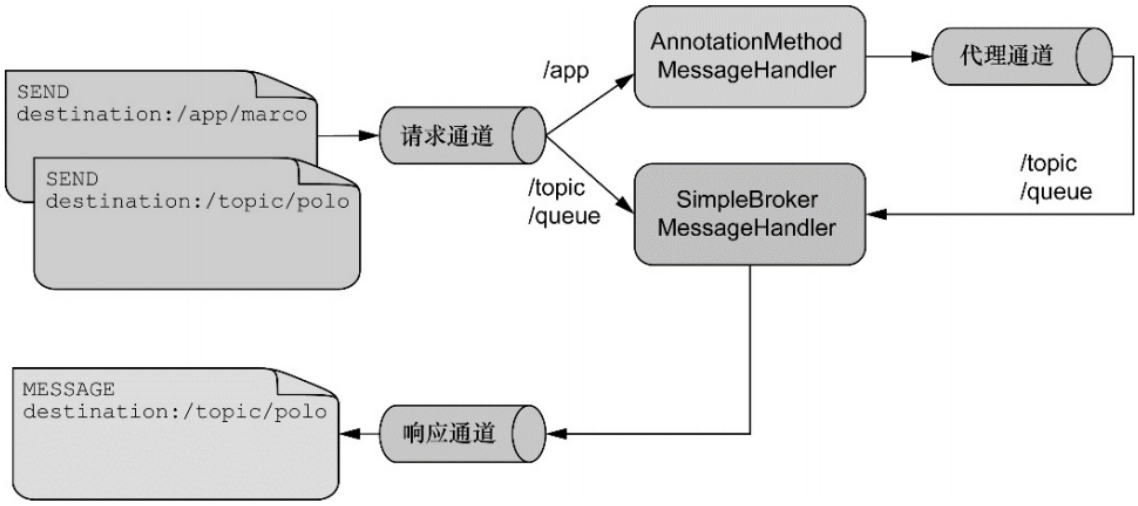
并非要使用一个原生的WebSocket连接，就像HTTP在TCP套接字之上添加了请求-响应模型曾一样，STOMP在WebSocket上提供了一个基于帧的线路格式层，用来定义消息的语义。

3.1 启用STOMP消息功能

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketStompConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/marcopolo").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue", "/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}
```

该配置类不仅配置了WebSocket，还配置了基于代理的STOMP消息，

- registerStompEndpoints注册了一个端点。客户端在订阅或发布消息到目的地路径前，要连接端点
- configureMessageBroker配置了一个简单的消息代理，如果不重载，将会自动配置一个简单的内存消息代理。用它来处理以‘topic’为前缀的消息。本例回自动处理前缀为“/topic”和“queue”的消息，并且发往应用程序的消息会带有“/app”前缀。



上图，应用程序的目的地以/app作为前缀，而代理的目的地以“/topic”和“queue”为前缀。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。而发送到代理上的消息，其中也包括@MessageMapping注解方法的返回值所形成的消息，将会路由到代理商，并最终发送到订阅这些目的地的客户端中

3.1.1 启用STOMP代理

简单的代理有限制，尽管它模拟了STOMP消息大力，但是它只支持STOMP命令的子集，不适合集群。如果是集群的话，每个节点也只能管理自己的代理和自己的那部分消息。

使用STOMP代理来替换内存代理：

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.enableStompBrokerRelay("/queue","/topic");
    registry.setApplicationDestinationPrefixes("/app");
}
```

- enableStompBrokerRelay 启用了STOMP代理中继功能，并将其目的地前缀设置为“topic”和“queue”。这样，Spring就知道所有目的地前缀为“topic”和“queue”的消息都会发送到STOMP代理中
- setApplicationDestinationPrefixes 将应员工的前缀设置为“/app”,所有目的地以“/app”开头的消息都会路由到带有@MessageMaping注解的方法中，而不会发布到代理队列或主题中。

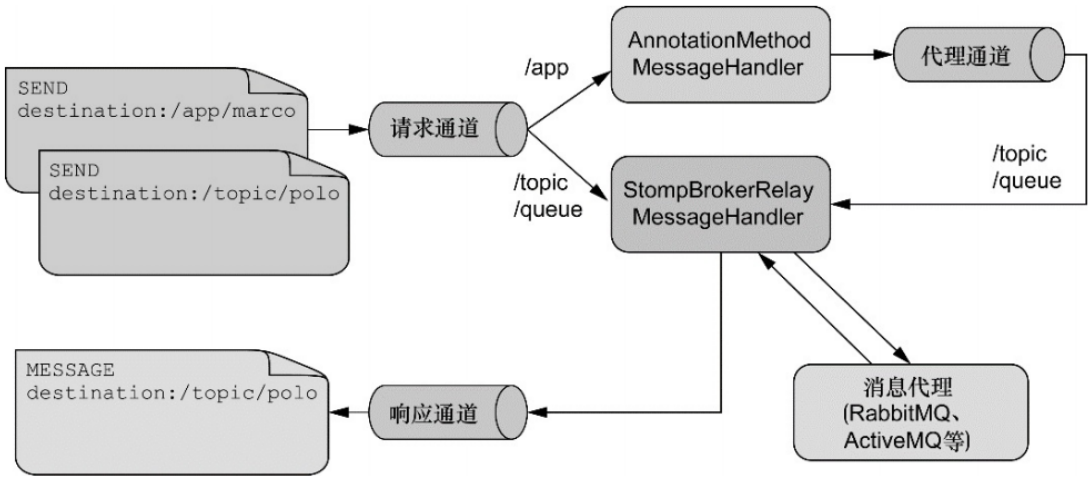


图18.3 STOMP代理中继会将STOMP消息的处理委托给一个真正的消息代理

中继不再使用模拟STOMP代理的功能，而是由代理中继将消息发送到一个真正的消息代理中。

默认情况下，STOMP代理中继会假设代理监听localhost的61613，并且客户端和passward均为“guest” 设置远程连接：

```
@Override
public void configureMessageBroker(MessageBrokerRegistry registry) {
    registry.enableStompBrokerRelay("/queue","/topic")
            .setRelayHost("rabbit.someotherserver")
            .setRelayPort(62623)
            .setClientLogin("marcopolo")
            .setClientPasscode("admin");
    registry.setApplicationDestinationPrefixes("/app");
}
```

3.2 处理来自客户端的STOMP消息

Spring提供了非常类似于Spring MVC的编程模型来处理STOMP消息。**STOMP消息的处理器方法也会包含在带有@Controller注解的类中**

```
@Controller
public class MarcoController {

    private static final Logger logger =
LoggerFactory.getLogger(MarcoController.class);

    @MessageMapping("/marco")
    public void handleShout(Shout incoming) {
        logger.info("Received message: " + incoming.getMessage());
    }
}
```

当 消息抵达某个特定的目的地时，带有@MessageMapping注解的方法能够处理这些消息。本例中，目的地是“/app/marco”

handleShout方法接受一个Shout参数，所以Spring的消息转换器会将STOMP消息转换为Shout对象

```
public class Shout {

    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

}
```

消息转换器

消息转换器	描述
ByteArrayMessageConverter	实现MIME类型为“application/octet-stream”消息与byte[]之间的相互转换
MappingJackson2MessageConverter	实现MIME类型为“application/json”消息与java对象之间的相互转换
StringMessageConverter	实现MIME类型为“text/plain”消息与string之间的相互转换

3.2.1 处理订阅

处理@MessagingMapping注解，Spring还提供了@SubscribeMapping注解。当收到STMP定于消息的时候，@SubscribeMapping注解的方法将会触发。

与@MessagingMapping类似@SubscribeMapping方法也是通过AnnotationMethodMessageHandler接接收消息的。@SubscribeMapping只能接受以“app”为前缀的消息

@SubscribeMapping 主要应用场景是实现请求-应答模式。客户端订阅某一个目的地，然后预期在这个目的地上获得一个一次性的响应

```
@SubscribeMapping("/marco")
public Shout handleSunscripting() {
    Shout shout = new Shout();
    shout.setMessage("polo");
    return shout;
}
```

处理对“/app/marco”目的地的订阅。该方法与HTTP的GET的请求-响应模式没有他打差别。区别在于，订阅-回应模式是异步的。

3.2.2 编写客户端

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">

<head>
    <title>Home</title>
    <script th:src="@{/webjars/sockjs-client/0.3.4/sockjs.min.js}"></script>
    <script th:src="@{/webjars/stomp-websocket/2.3.0/stomp.min.js}"></script>
    <script th:src="@{/webjars/jquery/2.0.3/jquery.min.js}"></script>
</head>
<body>
<button id="stop">Stop</button>

<script th:inline="javascript">

    var url = 'marcopolo';
    var sock = new SockJS(url);
    var stomp = Stomp.over(sock);

    stomp.connect('guest', 'guest', function (frame) {
        console.log('*****  Connected  *****');
        stomp.subscribe("/topic/marco", handlePolo);
        sayMarco();
```

```
    });

    function handleOneTime(message) {
        console.log('Received: ', message);
    }

    function handlePolo(message) {
        console.log('Received: ', message);
        $('#output').append("<b>Received: " +
            JSON.parse(message.body).message + "</b><br/>")
        if (JSON.parse(message.body).message === 'Polo!') {
            setTimeout(function () {
                sayMarco()
            }, 2000);
        }
    }

    function handleErrors(message) {
        console.log('RECEIVED ERROR: ', message);
        $('#output').append("<b>GOT AN ERROR!!!: " +
            JSON.parse(message.body).message + "</b><br/>")
    }

    function sayMarco() {
        console.log('Sending Marco!');
        stomp.send("/app/marco", {},
            JSON.stringify({'message': 'Marco!'}));
        $('#output').append("<b>Send: Marco!</b><br/>")
    }

    $('#stop').click(function () {
        sock.close()
    });
</script>

<div id="output"></div>
</body>
</html>
```

Stomp.over(sock);创建了一个STOMP客户端实例，实际上封装了SockJS，泽阳就能在WebSocket连接上发送STOMP消息。

stomp.send发丝不过都体育HSIB负载的消息到名为“marco”的目目的地。第二个参数是一个头信息的map，会包含在STOMP的帧中，

3.3 发送洗脑洗到客户端

- **作为处理消息或处理订阅的附带结果**
- **使用消息模板**

```
@Controller
public class MarcoController {
    @MessageMapping("/marco")
    public Shout handleShout(Shout incoming) {
        logger.info("Received message: " + incoming.getMessage());

        Shout outgoing = new Shout();
        outgoing.setMessage("Polo!");

        return outgoing;
    }
}
```

在响应中发送一条消息，只需要将内容返回就可以。当@MessageMapping注解标示的方法有返回值的时候，返回的对象将会进行转换并发到STOMP帧的负载中，然后发送给消息代理

默认情况下，帧发往的目的地会与触发处理器方法的目的地相同，只不过会添加上“/topic”。本例中返回的SHout对象会写入到STOMP的负载中，并发布到“/topic/marco”目的地。同时可以通过为方法添加@SendTo，重载目的地：

```
@MessageMapping("/marco")
@SendTo("/topic/shout")
public Shout handleSunscripting() {
    Shout shout = new Shout();
    shout.setMessage("polo");
    return shout;
}
```

类似地，@SubscribeMapping注解标注的方法也能发送一条消息，作为订阅的回应，

```
@SubscribeMapping("/marco")
@SendTo("/topic/shout")
public Shout handleSubscription() {
    Shout shout = new Shout();
    shout.setMessage("polo");
    return shout;
}
```

@SubscribeMapping注解表明，当客户端订阅“/app/marco”目的地的时候，将会调用handleSubscription。@SubscribeMapping的区别在于Shout消息将会直接发送给客户端，而不必经过消息代理，如果添加@SendTo，消息将发送到指定的目的地，这样会经过代理。

3.3.1 在应用的任意地方发送消息

SimpMessagingTemplate能够在应用的任意地方发送消息，也不必以首先接受一条消息做前提。

前端

```
var url = 'marcopolo';
var sock = new SockJS(url);
var stomp = Stomp.over(sock);

stomp.connect('guest', 'guest', function (frame) {
console.log('*****  Connected  *****');
stomp.subscribe("/topic/spittlefeed", handleSpittle);
stomp.subscribe("/user/queue/notifications", handleNotification);
});
```

后端

```
@Service
public class SpittleFeedServiceImpl implements SpittleFeedService {

    private SimpMessagingTemplate messaging;
    private Pattern pattern = Pattern.compile("\\@(\\S+)");

    @Autowired
    public SpittleFeedServiceImpl(SimpMessagingTemplate messaging) {
        this.messaging = messaging;
    }

    @Override
    public void broadcastSpittle(Spittle spittle) {
        messaging.convertAndSend("/topic/spittlefeed", spittle);

        Matcher matcher = pattern.matcher(spittle.getMessage());
        if (matcher.find()) {
            String username = matcher.group(1);
            messaging.convertAndSendToUser(username, "/queue/notifications",
                    new Notification("You just got mentioned!"));
        }
    }

}
```

配置Spring支持STOMP，Spring会在应用上下文中包含SimpMessagingTemplate，因此无需在创建新的实例。

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受到消息。

4 为目标用户发送消息

可以是哟共Spring Security来认证用户，并为目标用户处理消息

- **@MessageMapping和@SubsctibeMapping标注的方法能够Principal来获取认证用户**
- **@MessageMapping、SubsctibeMapping和MessageException方法返回的只能够一消息的形式发送给认证用户**
- **SimpMessagingTemplate额能发送消息个特定用户**

4.1 在控制器中处理用户的消息

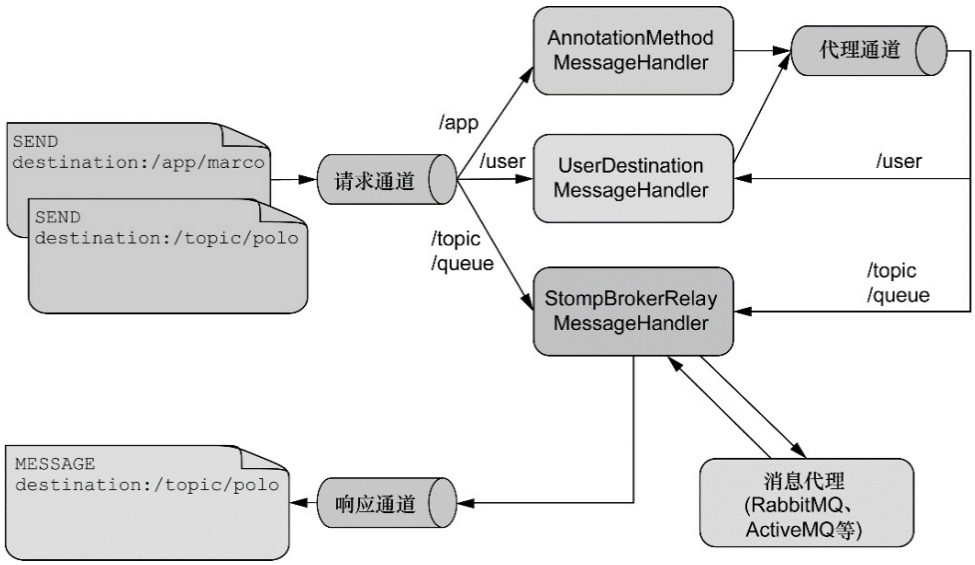
在处理去方法中，通过简单地添加一个Principal参数，这个方法就知道用户是谁并利用该信息关注此用户相关的数据。处理器方法还可以使用@SendToUser注解，表明它的返回值要以消息的形式发送给某个认证用户的客户端。

```
@Controller
public class SpittrMessageController {

    private SpittleRepository spittleRepo;
    private SpittleFeedService feedService;

    @Autowired
    public SpittrMessageController(SpittleRepository spittleRepo,
SpittleFeedService feedService) {
        this.spittleRepo = spittleRepo;
        this.feedService = feedService;
    }

    @MessageMapping("/spittle")
    @SendToUser("/queue/notifications")
    public Notification handleSpittle(Principal principal, SpittleForm form) {
        Spittle spittle = new Spittle(principal.getName(), form.getText(), new
Date());
        spittleRepo.save(spittle);
        feedService.broadcastSpittle(spittle);
        return new Notification("Saved Spittle for user: " +
principal.getName());
    }

}
```

@SendToUser指定返回的Notification以消息的形式发送到"/queue/notifications"的目的地上，即：/user/queue/notifications

stomp.subscribe("/user/queue/notifications", handleNotification);

以"/user"作为目的地将会以特殊的方式进行处理。这种消息不会通过AnnotationMethodMessageHandler(像应用消息那样)来处理，也不会通过SimpBrokerMessageHandler或StompBrokerRelayMessageHandler(像代理消息那样)来处理。将会通过UserDestinationMessageHandler处理



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

在处理订阅的时候，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。例如：对"/user/queue/notifications"的订阅最后可能路由到名为"/queue/notification-user6hr83v6t"的目的地上

@SendToUser("/queue/notifications")，根据配置，这是StompBrokerRelayMessageHandler(或SimpBrokerMessageHandler)要处理的前缀，所有消息接下来会到达这里。由于客户端会订阅这个目的地，因此客户端会受到Notification消息

4.2 为指定用户发送消息

```
@Service
public class SpittleFeedServiceImpl implements SpittleFeedService {

    private SimpMessagingTemplate messaging;
    private Pattern pattern = Pattern.compile("\\@(\\S+)");//实现用户体积功能的正则表达式

    @Autowired
    public SpittleFeedServiceImpl(SimpMessagingTemplate messaging) {
        this.messaging = messaging;
    }

    @Override
    public void broadcastSpittle(Spittle spittle) {
        messaging.convertAndSend("/topic/spittlefeed", spittle);

        Matcher matcher = pattern.matcher(spittle.getMessage());
        if (matcher.find()) {
            String username = matcher.group(1);//发送提醒给用户
            messaging.convertAndSendToUser(username, "/queue/notifications",
                    new Notification("You just got mentioned!"));
        }
    }

}
```

如果给定Spittle对象的消息中共包含了类似于用户名的内容(以”@“开头的文本)，那么一个新的Notification将会发送到"/queue/notifications"

5 处理消息异常

可以在控制器方法上添加@MessageExceptionHandler注解，来处理@MessageMapping方法抛出的异常

```
@MessageExceptionHandler
public void handleException(Throwable throwable) {
    logger.error(throwable.getMessage());
}
```

可以以参数的形式声明它所能处理的异常

```
@MessageExceptionHandler({SpittleException.clss})
public void handleException(Throwable throwable) {
    logger.error(throwable.getMessage());
}
```