WebSocket 两个实现方式 (Spring Boot)

初版

开始

众所周知，在Spring Boot下，实现WebSocket共有两种方式。

第一种是使用由 Jakarta EE 规范提供的 API，也就是 `jakarta.websocket` 包下的接口。第二种是使用 Spring 提供的支持，也就是 `spring-websocket` 模块。

1.**使用 Jakarta EE 规范提供的 API**

[jakarta.ee/learn/docs/...](http://cxyroad.com/"https://jakarta.ee/learn/docs/jakartaee-tutorial/current/web/websocket/websocket.html")

2.**使用 spring 提供的支持，也就是 spring-websocket 模块**

[docs.spring.io/spring-fram...](http://cxyroad.com/"https://docs.spring.io/spring-framework/reference/web/websocket.html")

两种方式，喜欢那种就用那种吧，Jakarta 功能更直接些，Spring更好的屏蔽了服务器差异，封装了更多功能，可以直接配置服务器参数，且更支持Spring生态

无论那种方式，我们都需要添加依赖

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

</dependency>

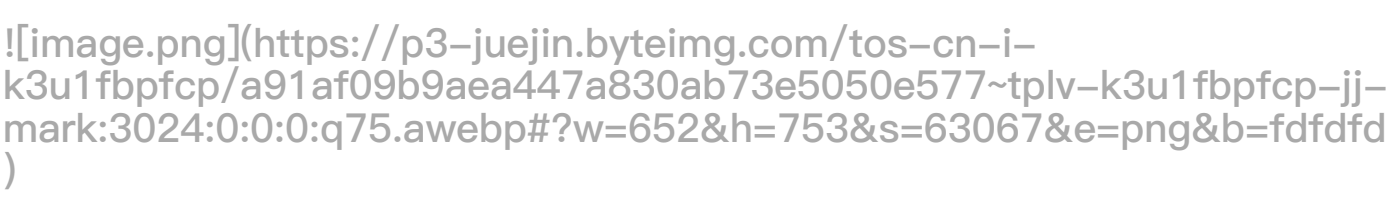
```

jakarta-websocket
=================


配置 ServerEndpointExporter
------------------------


[docs.spring.io/spring-boot...](http://cxyroad.com/
"https://docs.spring.io/spring-boot/how-
to/webserver.html#howto.webserver.create-websocket-endpoints-
using-serverendpoint")

`ServerEndpointExporter` 会尝试去扫描并注册所有标有
 `@ServerEndpoint` 注解的类。这是 Spring Boot 中使用 Jakarta EE
WebSocket 规范的一种方式。



```
import
org.springframework.web.socket.server.standard.ServerEndpointExporter;
@Configuration
public class WebSocketConfig implements WebSocketConfigurer {

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

```


> 上例中显示的 `Bean` 会向底层 `WebSocket` 容器注册任何带有
`@ServerEndpoint` 注释的 `Bean`。当部署到独立的 `servlet` 容器时，**这
一角色由 `servlet` 容器初始化器执行**，而不由`ServerEndpointExporter`


书写 ServerEndpoint
----------------
```

```
@ServerEndpoint(”/websocket”)
@Component  // 交由Spring管理，便于自动注册
public class MyWebSocket implements ApplicationContextAware {

    @OnOpen
    public void onOpen(Session session) throws IOException {
        session.getBasicRemote().sendText(hashCode() + ””);
    }

    @OnMessage
    public void onMessage(String message, Session session) throws
IOException {
        sendMessageToAll(message, rid, session);
    }

    @OnError
    public void onError(Session session, Throwable throwable) {
        System.out.println(”Error”);
        throwable.printStackTrace();
    }

    @OnClose
    public void onClose(Session session, CloseReason reason) throws
IOException {
        session.close();
        System.out.println(”disconnected”);
    }

    private void sendMessageToAll(String message, Session session)
throws IOException {
        for (Session sess : session.getOpenSessions()) {
            if (sess.isOpen() && !sess.equals(session)) {
                sess.getBasicRemote().sendText(message);
            }
        }
    }

    // 无参构造器
    public MyWebSocket() {
        System.out.println(”MyWebSocket NoConstructor”);
    }

    @Override
    public void setApplicationContext(ApplicationContext
applicationContext) throws BeansException {
        System.out.println(”MyWebSocket setApplicationContext”);
    }
```

```
    @PostConstruct
    public void init() {
        System.out.println("MyWebSocket init-method");
    }
}
```



Encoder Decoder

Encoder Decoder可以在接收msg前与发送msg后对msg进行一些编码解码操作。详见

[jakarta.ee/learn/docs/...](http://cxyroad.com/"https://jakarta.ee/learn/docs/jakartaee-tutorial/current/web/websocket/websocket.html#_using_encoders_and_decoders")

路径参数

```
@ServerEndpoint("/chatrooms/{room-name}")
public class ChatEndpoint {
   @OnOpen
   public void open(Session session,
                    EndpointConfig c,
                    @PathParam("room-name") String roomName) {
      // Add the client to the chat room of their choice ...
   }
}
```

详见

[jakarta.ee/learn/docs/…](http://cxyroad.com/
”https://jakarta.ee/learn/docs/jakartaee-
tutorial/current/web/websocket/websocket.html#_path_parameters”)

握手阶段拦截

[jakarta.ee/learn/docs/…](http://cxyroad.com/
”https://jakarta.ee/learn/docs/jakartaee-
tutorial/current/web/websocket/websocket.html#_specifying_an_endpoint_configurator_class”)

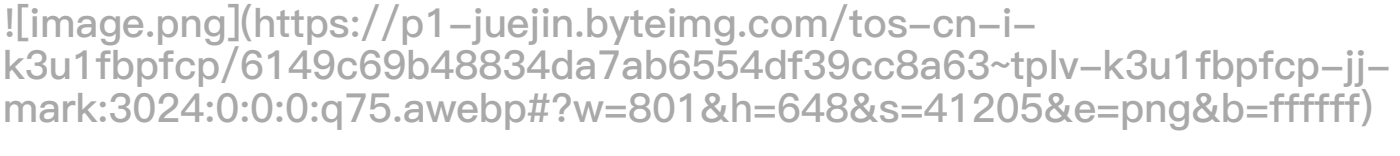
注意
--

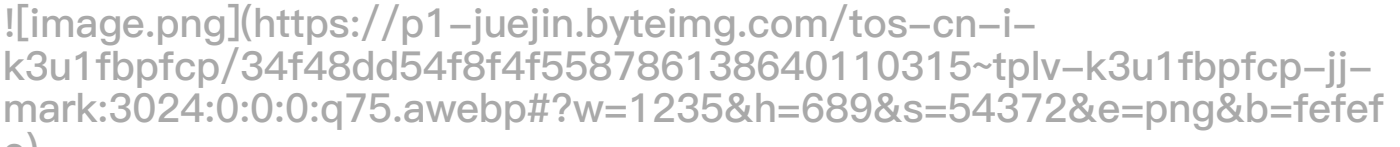
多次实例化

> As opposed to servlets, WebSocket endpoints are instantiated multiple times. The container creates an instance of an endpoint per connection to its deployment URI. Each instance is associated with one and only one connection. This facilitates keeping user state for each connection and makes development easier, because there is only one thread executing the code of an endpoint instance at any given time.

验证

我们通过在每次连接后返回对应实例的`hashcode`来判断是否是同一实例





由此可知，在这种方法下，我们每一次的连接都会创建一个新的`webSocket

endpoints`

注入Bean

> The bean shown in the preceding example registers any `@ServerEndpoint` annotated beans with the underlying WebSocket container. **When deployed to a standalone servlet container, this role is performed by a servlet container initializer**, and the `ServerEndpointExporter` bean is not required.

原因：运行时的 WebSocket 连接对象，也就是端点实例，是由服务器创建，而不是 Spring，所以不能使用自动装配

解决

要求ServerEndpoint由Spring管理 :heavy_exclamation_mark:

```
@ServerEndpoint(value = ”/channel/echo”)
@Component  // 由 spring 扫描管理
public class EchoChannel implements
                ApplicationContextAware { // 实现 ApplicationContextAware 接口， Spring 会在运行时注入 ApplicationContext

    // 全局静态变量，保存 ApplicationContext
    private static ApplicationContext applicationContext;

    // 声明需要的 Bean
    private UserService userService;

    // 保存 Spring 注入的 ApplicationContext 到静态变量
    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        EchoChannel.applicationContext = applicationContext;
    }

    @OnOpen
    public void onOpen(Session session, EndpointConfig endpointConfig){
```

```
        // 保存 session 到对象
        this.session = session;

        // 连接创建的时候，从 ApplicationContext 获取到 Bean 进行初始化
        this.userService =
EchoChannel.applicationContext.getBean(UserService.class);

        // 在业务中使用
        this.userService.foo();
    }
    // ....
}

```

流程如下

Spring启动--Spring注册Bean--Spring将ServerEndpoint移交web容器

我们可以在Spring移交前，在注册Bean时，对Bean的**静态变量**进行操作，这样就可以把一下参数带入容器

`ApplicationContextAware`是什么？，参考Bean的生命周期

spring-websocket
================

Spring Framework 提供了 WebSocket API，您可以用它来编写处理 WebSocket 消息的客户端和服务器端应用程序。

配置 handler

你可以通过去实现`WebSocketHandler`, 或扩展 `TextWebSocketHandler` 或 `BinaryWebSocketHandler`

```
public class MyWebSocketHandler implements WebSocketHandler {
    @Override
    public void handleMessage(WebSocketSession session,
WebSocketMessage<?> message) {
        // ...
```

```
    }
}
```

```
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {
    @Override
    public void handleTextMessage(WebSocketSession session,
TextMessage message) {
        // ...
    }
}
```

如下例所示，有专门的 WebSocket Java 配置将 WebSocket 处理程序映射到特定 URL

```
import
org.springframework.web.socket.config.annotation.EnableWebSocket;
import
org.springframework.web.socket.config.annotation.WebSocketConfigurer
;
import
org.springframework.web.socket.config.annotation.WebSocketHandlerRe
gistry;

@Configuration
@EnableWebSocket  // 必须
public class WebSocketConfig implements WebSocketConfigurer {
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry
registry) {
            registry.addHandler(myHandler(), ”/myHandler”);
    }

    @Bean
    public WebSocketHandler myHandler() {
            return new MyHandler();
```

```
    }
}
```

至此，一个简单的websocket编写完毕

握手阶段拦截器

定制初始 HTTP WebSocket 握手请求的最简单方法是通过 `HandshakeInterceptor`，它为握手的 ”前” 和 ”后” 暴露了方法。你可以使用这样的拦截器来处理握手。

```
@Component
public class MyHandshakeInterceptor implements HandshakeInterceptor
{
    @Override
    public boolean beforeHandshake(ServerHttpRequest request,
ServerHttpResponse response, WebSocketHandler wsHandler,
Map<String, Object> attributes) throws Exception {
        System.out.println(”beforeHandshake”);
        return true;
    }

    @Override
    public void afterHandshake(ServerHttpRequest request,
ServerHttpResponse response, WebSocketHandler wsHandler, Exception
exception) {
        System.out.println(”afterHandshake”);
    }
}
```

```
@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry
registry) {
    registry.addHandler(myWebSocketHandler(), ”/ws”)
            .addInterceptors(new MyHandshakeInterceptor())
            .setAllowedOrigins(”*”);
```

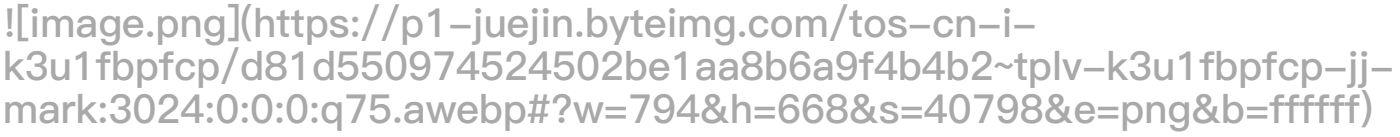
}

```

异常处理器
-----

被修饰的Handler将实现全局异常处理

```
@Bean
public ExceptionWebSocketHandlerDecorator myWebSocketHandler() {
    return new ExceptionWebSocketHandlerDecorator(new
MyWebSocketHandler());
}

```

配置服务器
-----

[docs.spring.io/spring-fram...](http://cxyroad.com/”https://docs.spring.io/spring-framework/reference/web/websocket/server.html#websocket-server-runtime-configuration”)

Encoder Decoder
---------------

没找到。。自己使用AOP增强吧

路径参数
----

不支持

注意
--
```

单次实例化

每次访问同一个路径使用的都是同一个handler对象



> Spring 提供了一个 WebSocketHandlerDecorator 基类，您可以用它来为 WebSocketHandler 装饰附加行为。在使用 WebSocket Java 配置或 XML 命名空间时，默认情况下会提供并添加日志和异常处理实现。ExceptionWebSocketHandlerDecorator 会捕获任何 WebSocketHandler 方法产生的所有未捕获异常，并以 1011 状态关闭 WebSocket 会话，该状态表示服务器出错。

注入Bean，同上面的解决方案

原文链接: https://juejin.cn/post/7383205961195094043