# **[websocket之三：Tomcat的WebSocket实现](https://www.cnblogs.com/duanxz/p/5041110.html)**

Tomcat自7.0.5版本开始支持WebSocket，并且实现了Java WebSocket规范（JSR356 ），而在7.0.5版本之前（7.0.2版本之后）则采用自定义API，即WebSocketServlet。本节我们仅介绍Tomcat针对规范的实现。

根据JSR356的规定，Java WebSocket应用由一系列的WebSocket Endpoint组成。Endpoint是一个Java对象，代表WebSocket链接的一端，对于服务端，我们可以视为处理具体WebSocket消息的接口，就像Servlet之于HTTP请求一样（不同之处在于Endpoint每个链接一个实例）。

我们可以通过两种方式定义Endpoint，第一种是编程式，即继承类javax.websocket.Endpoint并实现其方法。第二种是注解式，即定义一个POJO对象，为其添加Endpoint相关的注解。

Endpoint实例在WebSocket握手时创建，并在客户端与服务端链接过程中有效，最后在链接关闭时结束。Endpoint接口明确定义了与其生命周期相关的方法，规范实现者确保在生命周期的各个阶段调用实例的相关方法。

Endpoint的生命周期方法如下：

* onOpen：当开启一个新的会话时调用。这是客户端与服务器握手成功后调用的方法。等同于注解@OnOpen。
* onClose：当会话关闭时调用。等同于注解@OnClose。
* onError：当链接过程中异常时调用。等同于注解@OnError。

当客户端链接到一个Endpoint时，服务器端会为其创建一个唯一的会话（javax.websocket.Session）。会话在WebSocket握手之后创建，并在链接关闭时结束。当生命周期中触发各个事件时，都会将当前会话传给Endpoint。

我们通过为Session添加MessageHandler消息处理器来接收消息。当采用注解方式定义Endpoint时，我们还可以通过@OnMessage指定接收消息的方法。发送消息则由RemoteEndpoint完成，其实例由Session维护，根据使用情况，我们可以通过Session.getBasicRemote获取同步消息发送的实例或者通过Session.getAsyncRemote获取异步消息发送的实例。

WebSocket通过javax.websocket.WebSocketContainer接口维护应用中定义的所有Endpoint。它在每个Web应用中只有一个实例，类似于传统Web应用中的ServletContext。

最后，WebSocket规范提供了一个接口javax.websocket.server.ServerApplicationConfig，通过它，我们可以为编程式的Endpoint创建配置（如指定请求地址），还可以过滤只有符合条件的Endpoint提供服务。该接口的实现同样通过SCI机制加载。

介绍完WebSocket规范中的基本概念，我们看一下Tomcat的具体实现。接下来会涉及到Tomcat链接器（Cotyte）和Web应用加载的知识，如不清楚可以阅读Tomcat官方文档。

## **WebSocket加载**

Tomcat提供了一个javax.servlet.ServletContainerInitializer的实现类org.apache.tomcat.websocket.server.WsSci。因此Tomcat的WebSocket加载是通过SCI机制完成的。WsSci可以处理的类型有三种：添加了注解@ServerEndpoint的类、Endpoint的子类以及ServerApplicationConfig的实现类。

Web应用启动时，通过WsSci.onStartup方法完成WebSocket的初始化：

* 构造WebSocketContainer实例，Tomcat提供的实现类为WsServerContainer。在WsServerContainer构造方法中，Tomcat除了初始化配置外，还会为ServletContext添加一个过滤器org.apache.tomcat.websocket.server.WsFilter，它用于判断当前请求是否为WebSocket请求，以便完成握手。
* 对于扫描到的Endpoint子类和添加了注解@ServerEndpoint的类，如果当前应用存在ServerApplicationConfig实现，则通过ServerApplicationConfig获取Endpoint子类的配置（ServerEndpointConfig实例，包含了请求路径等信息）和符合条件的注解类，将结果注册到WebSocketContainer上，用于处理WebSocket请求。
* 通过ServerApplicationConfig接口我们以编程的方式确定只有符合一定规则的Endpoint可以注册到WebSocketContainer，而非所有。规范通过这种方式为我们提供了一种定制化机制。
* 如果当前应用没有定义ServerApplicationConfig的实现类，那么WsSci默认只将所有扫描到的注解式Endpoint注册到WebSocketContainer。因此，如果采用可编程方式定义Endpoint，那么必须添加ServerApplicationConfig实现。

## **WebSocket请求处理**

当服务器接收到来自客户端的请求时，首先WsFilter会判断该请求是否是一个WebSocket Upgrade请求（即包含Upgrade: websocket头信息）。如果是，则根据请求路径查找对应的Endpoint处理类，并进行协议Upgrade。

在协议Upgrade过程中，除了检测WebSocket扩展、添加相关的转换外，最主要的是添加WebSocket相关的响应头信息、构造Endpoint实例、构造HTTP Upgrade处理类WsHttpUpgradeHandler。

将WsHttpUpgradeHandler传递给具体的Tomcat协议处理器（ProtocolHandler）进行Upgrade。接收到Upgrade的动作后，Tomcat的协议处理器（HTTP协议）不再使用原有的Processor处理请求，而是替换为专门的Upgrade Processor。

根据I/O的不同，Tomcat提供的Upgrade Processor实现如下：

* org.apache.coyote.http11.upgrade.BioProcessor；
* org.apache.coyote.http11.upgrade.NioProcessor；
* org.apache.coyote.http11.upgrade.Nio2Processor；
* org.apache.coyote.http11.upgrade.AprProcessor；

替换成功后，WsHttpUpgradeHandler会对Upgrade Processor进行初始化（按以下顺序）：

* 创建WebSocket会话。
* 为Upgrade Processor的输出流添加写监听器。WebSocket向客户端推送消息具体由org.apache.tomcat.websocket.server.WsRemoteEndpointImplServer完成。
* 构造WebSocket会话，执行当前Endpoint的onOpen方法。
* 为Upgrade Processor的输入流添加读监听器，完成消息读取。WebSocket读取客户端消息具体由org.apache.tomcat.websocket.server.WsFrameServer完成。

通过这种方式，Tomcat实现了WebSocket请求处理与具体I/O方式的解耦。

## **基于编程的示例**

首先，添加一个Endpoint子类，代码如下：

[IMG_256](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

package org.springframework.samples.websocket.demo3;

import java.io.IOException;import java.util.Set;import java.util.concurrent.CopyOnWriteArraySet;

import javax.websocket.CloseReason;import javax.websocket.Endpoint;import javax.websocket.EndpointConfig;import javax.websocket.MessageHandler;import javax.websocket.Session;

public class ChatEndpoint extends Endpoint {

private static final Set<ChatEndpoint> connections = new CopyOnWriteArraySet<>();

private Session session;

private static class ChatMessageHandler implements MessageHandler.Partial<String> {

private Session session;

private ChatMessageHandler(Session session) {

this.session = session;

}

@Override

public void onMessage(String message, boolean last) {

String msg = String.format("%s %s %s", session.getId(), "said:", message);

broadcast(msg);

}

};

@Override

public void onOpen(Session session, EndpointConfig config) {

this.session = session;

connections.add(this);

this.session.addMessageHandler(new ChatMessageHandler(session));

String message = String.format("%s %s", session.getId(), "has joined.");

broadcast(message);

}

@Override

public void onClose(Session session, CloseReason closeReason) {

connections.remove(this);

String message = String.format("%s %s", session.getId(), "has disconnected.");

broadcast(message);

}

@Override

public void onError(Session session, Throwable throwable) {

}

private static void broadcast(String msg) {

for (ChatEndpoint client : connections) {

try {

synchronized (client) {

client.session.getBasicRemote().sendText(msg);

}

} catch (IOException e) {

connections.remove(client);

try {

client.session.close();

} catch (IOException e1) {

}

String message = String.format("%s %s", client.session.getId(), "has been disconnected.");

broadcast(message);

}

}

}

}

[IMG_257](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

为了方便向客户端推送消息，我们使用一个静态集合作为链接池维护所有Endpoint实例。

在onOpen方法中，首先将当前Endpoint实例添加到链接池，然后为会话添加了一个消息处理器ChatMessageHandler，用于接收消息。当接收到客户端消息后，我们将其推送到所有客户端。最后向所有客户端广播一条上线通知。

在onClose方法中，将当前Endpoint从链接池中移除，向所有客户端广播一条下线通知。

然后定义ServerApplicationConfig实现，代码如下：

[IMG_258](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

package org.springframework.samples.websocket.demo3;

import java.util.HashSet;import java.util.Set;

import javax.websocket.Endpoint;import javax.websocket.server.ServerApplicationConfig;import javax.websocket.server.ServerEndpointConfig;

public class ChatServerApplicationConfig implements ServerApplicationConfig {

@Override

public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned) {

return scanned;

}

@Override

public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends Endpoint>> scanned) {

Set<ServerEndpointConfig> result = new HashSet<>();

if (scanned.contains(ChatEndpoint.class)) {

result.add(ServerEndpointConfig.Builder.create(ChatEndpoint.class, "/program/chat").build());

}

return result;

}

}

[IMG_259](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

在ChatServerApplicationConfig中为ChatEndpoint添加ServerEndpointConfig，其请求链接为“/program/chat”。

最后添加对应的HTML页面，src\main\webapp\chat.html：

[IMG_260](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

<?xml version="1.0" encoding="UTF-8"?>

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

<head>

<script type="application/javascript">

var Chat = {};

Chat.socket = null;

Chat.connect = (function(host) {

if ('WebSocket' in window) {

Chat.socket = new WebSocket(host);

} else if ('MozWebSocket' in window) {

Chat.socket = new MozWebSocket(host);

} else {

Console.log('Error: WebSocket is not supported by this browser.');

return;

}

Chat.socket.onopen = function () {

Console.log('Info: WebSocket connection opened.');

document.getElementById('chat').onkeydown = function(event) {

if (event.keyCode == 13) {

Chat.sendMessage();

}

};

};

Chat.socket.onclose = function () {

document.getElementById('chat').onkeydown = null;

Console.log('Info: WebSocket closed.');

};

Chat.socket.onmessage = function (message) {

Console.log(message.data);

};

});

Chat.initialize = function() {

if (window.location.protocol == 'http:') {

Chat.connect('ws://' + window.location.host + '/spring-websocket-test/program/chat');

} else {

Chat.connect('wss://' + window.location.host + '/spring-websocket-test/program/chat');

}

};

Chat.sendMessage = (function() {

var message = document.getElementById('chat').value;

if (message != '') {

Chat.socket.send(message);

document.getElementById('chat').value = '';

}

});

var Console = {};

Console.log = (function(message) {

var console = document.getElementById('console');

var p = document.createElement('p');

p.style.wordWrap = 'break-word';

p.innerHTML = message;

console.appendChild(p);

while (console.childNodes.length > 25) {

console.removeChild(console.firstChild);

}

console.scrollTop = console.scrollHeight;

});

Chat.initialize();

</script>

</head>

<body>

<div>

<p>

<input type="text" placeholder="type and press enter to chat" id="chat" />

</p>

<div id="console-container">

<div id="console"/>

</div>

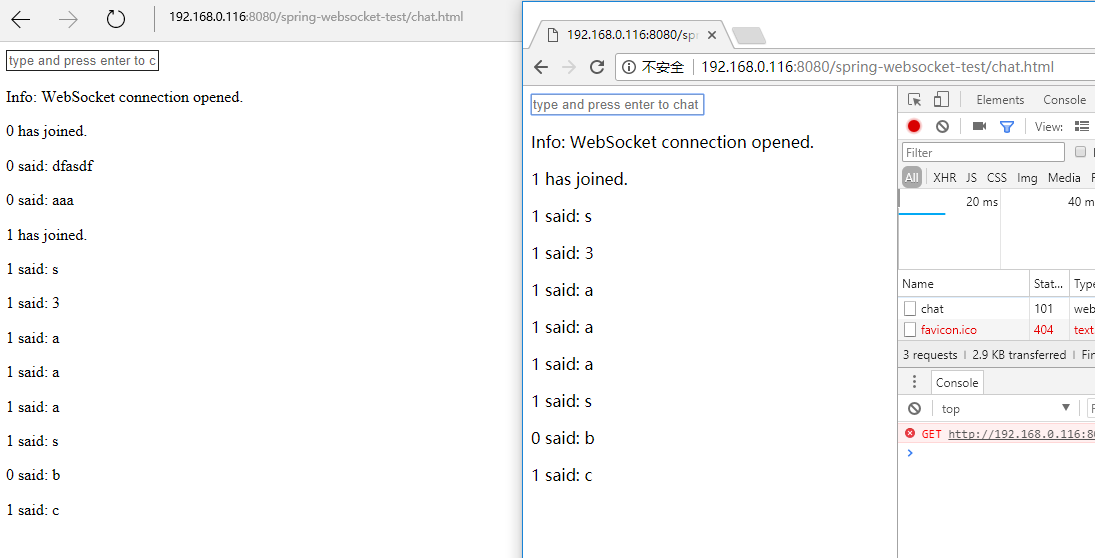
</div>

</body>

</html>

[IMG_261](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

客户端实现并不复杂，只是要注意浏览器的区别。在添加完所有配置后，可以将应用部署到Tomcat查看效果，与Comet类似，我们可以同时开启两个客户端查看消息推送效果。



## **基于注解的示例**

基于注解的定义要比编程式简单一些，首先定义一个POJO对象，并添加相关注解：

[IMG_263](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

package org.springframework.samples.websocket.demo3;

import java.io.IOException;import java.util.Set;import java.util.concurrent.CopyOnWriteArraySet;

import javax.websocket.OnClose;import javax.websocket.OnError;import javax.websocket.OnMessage;import javax.websocket.OnOpen;import javax.websocket.Session;import javax.websocket.server.ServerEndpoint;

@ServerEndpoint(value = "/anno/chat")public class ChatAnnotation {

private static final Set<ChatAnnotation> connections = new CopyOnWriteArraySet<>();

private Session session;

@OnOpen

public void start(Session session) {

this.session = session;

connections.add(this);

String message = String.format("%s %s", session.getId(), "has joined.");

broadcast(message);

}

@OnClose

public void end() {

connections.remove(this);

String message = String.format("%s %s", session.getId(), "has disconnected.");

broadcast(message);

}

@OnMessage

public void incoming(String message) {

String msg = String.format("%s %s %s", session.getId(), "said:", message);

broadcast(msg);

}

@OnError

public void onError(Throwable t) throws Throwable {

}

private static void broadcast(String msg) {

for (ChatAnnotation client : connections) {

try {

synchronized (client) {

client.session.getBasicRemote().sendText(msg);

}

} catch (IOException e) {

connections.remove(client);

try {

client.session.close();

} catch (IOException e1) {

}

String message = String.format("%s %s", client.session.getId(), "has been disconnected.");

broadcast(message);

}

}

}

}

[IMG_264](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

@ServerEndpoint注解声明该类是一个Endpoint，并指定了请求的地址。

@OnOpen注解的方法在会话打开时调用，与ChatEndpoint类似，将当前实例添加到链接池。@OnClose注解的方法在会话关闭时调用。@OnError注解的方法在链接异常时调用。@OnMessage注解的方法用于接收消息。

使用注解方式定义Endpoint时，ServerApplicationConfig不是必须的，此时直接默认加载所有的@ServerEndpoin注解POJO。

我们可以直接将编程式示例中HTML页面src\main\webapp\chatanno.html中的链接地址改为“/anno/chat”查看效果。

[IMG_265](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/1999/xhtml"><head>

<script type="application/javascript">

var Chat = {};

Chat.socket = null;

Chat.connect = (function(host) {

if ('WebSocket' in window) {

Chat.socket = new WebSocket(host);

} else if ('MozWebSocket' in window) {

Chat.socket = new MozWebSocket(host);

} else {

Console.log('Error: WebSocket is not supported by this browser.');

return;

}

Chat.socket.onopen = function () {

Console.log('Info: WebSocket connection opened.');

document.getElementById('chat').onkeydown = function(event) {

if (event.keyCode == 13) {

Chat.sendMessage();

}

};

};

Chat.socket.onclose = function () {

document.getElementById('chat').onkeydown = null;

Console.log('Info: WebSocket closed.');

};

Chat.socket.onmessage = function (message) {

Console.log(message.data);

};

});

Chat.initialize = function() {

if (window.location.protocol == 'http:') {

Chat.connect('ws://' + window.location.host + '/spring-websocket-test/anno/chat');

} else {

Chat.connect('wss://' + window.location.host + '/spring-websocket-test/anno/chat');

}

};

Chat.sendMessage = (function() {

var message = document.getElementById('chat').value;

if (message != '') {

Chat.socket.send(message);

document.getElementById('chat').value = '';

}

});

var Console = {};

Console.log = (function(message) {

var console = document.getElementById('console');

var p = document.createElement('p');

p.style.wordWrap = 'break-word';

p.innerHTML = message;

console.appendChild(p);

while (console.childNodes.length > 25) {

console.removeChild(console.firstChild);

}

console.scrollTop = console.scrollHeight;

});

Chat.initialize();

</script></head><body><div>

<p>

<input type="text" placeholder="type and press enter to chat" id="chat" />

</p>

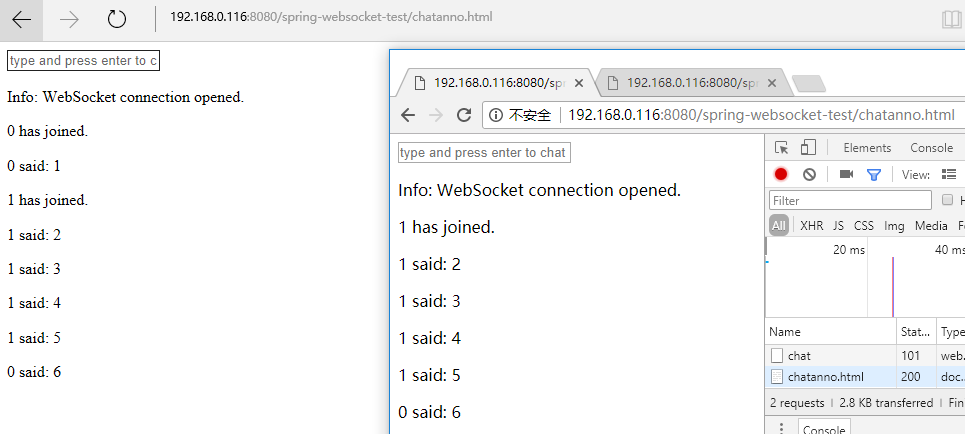
<div id="console-container">

<div id="console"/>

</div></div></body></html>

[IMG_266](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

结果：



tomcat是怎么加载ServerApplicationConfig的配置的，我想做嵌入式tomcat开发，请问您这边清楚的吗？

回复xiaospace1028：通过org.apache.tomcat.websocket.server.WsSci类，这是一个ServletContainerInitializer，容器启动时会自动加载这个类，执行onStartup方法