[WebSocket API 学习](https://www.cnblogs.com/xuemanjiangnan/p/7658362.html)

https://abhirockzz.gitbooks.io/java-websocket-api-handbook/content/

◆WebSocket是什么  
简单来说，WebSocket是一个IETF的标准，RFC6455.具体来说，它是一个基于TCP的协议（就像HTTP）。  
你可以把它认为是一个介于长轮询和服务器推送消息之间的解决方案。WebSocket连接通过HTTP来做初始化握手。  
重要：一旦建立了，TCP连接一直打开着。

◆为什么要用WebSocket  
它的关键特性是  
・双向 服务器端和客户端都可以发起连接  
・全双工 一旦WebSocket的session建立后，服务器端和客户端可以同时收发信息。

上述特性使WebSocket非常适合低延迟和高频度消息传送需求的程序。比如，聊天，监控，多人在线游戏，广播实时财务数据等。  
其中一些优势（与其他解决方案相比）包括  
・减少冗余信息（相对于HTTP）  
・更有效率（相对于长轮询）  
・更丰富的语义（相对于服务器消息推送）

◆WebSocket作为一个java标准  
API与实现  
对Java来说，这项技术被定义为JSR 356 ，一个在2013年5月发布第一版本的标准API  
2014年8月发布了一个1.1版本。像其他JSR（java规范需求）定义的API，WebSocket的java api由一个规范  
支持，使它可以有不同的实现而表现一致。  
・Tyrus 一个开源项目，并且是Weblogic and GlassFish的参考实现。  
・Undertow JBoss EAP and Wildfly使用的实现  
・Tomcat 7以上版本 tomcat内部实现

Java EE Platform  
JSR 356也是JavaEE7平台的一部分。任何符合JavaEE7的程序服务将包含一个默认的API实现。就像其他JAVAEE技术一样，  
比如EJB, CDI, Security等。

JSR356也被其他容器和框架支持比如Spring,Jetty等。

◆服务器和客户端模式  
・WebSocket Server endpoint  
一个服务器端组件是  
实现了一些业务逻辑  
公开自己，是自己可以被客户端发现（通过一个主机和端口）  
当有客户端连接到它时触发  
  
・WebSocket Client endpoint  
客户端组件  
实现一些业务逻辑  
连接到存在的WebSocket Server endpoint  
  
Java WebSocket API 提供服务器和客户端组件  
Server javax.websocket.server包  
Client javax.websocket包 里面也包含了服务器客户端共通的内容  
  
客户端组件包括ClientEndpoint, ClientEndpointConfig, ClientEndpointConfig.Builder,  
ClientEndpointConfig.Configurator, WebSocketContainer

支持的消息类型  
WebSocket规范支持两种在线数据格式 - 文本和二进制  
Java WebSocket API支持这些（显然），并增加了使用Java对象以及规范定义的健康检查消息（乒乓）的功能

Text java.lang.String, primitives or their equivalent wrapper classes  
Binary java.nio.ByteBuffer or a byte[] (byte array)  
Java objects API使您可以在代码中使用Java对象，并使用自定义变换器（编码器/解码器）  
将其转换为WebSocket协议允许的兼容在线格式（文本，二进制）  
Ping, Pong javax.websocket.PongMessage是由WebSocket对响应健康检查（ping）请求发送的确认

◆容器抽象  
javax.websocket.WebSocketContainer  
提供容器的高级视图，允许客户端端点激活和连接到现有（WebSocket）服务器，  
设置对服务器和客户端都适用的全局/公共属性（空闲连接超时，消息大小，异步发送超时）  
  
javax.websocket.server.ServerContainer  
用于服务器端的容器类（继承了WebSocketContainer） 增加了部署服务器端endpoint的功能  
  
javax.websocket.ContainerProvider  
提供了对WebSocketContainer的访问功能  
  
WebSocketContainer container = ContainerProvider.getWebSocketContainer();  
container.setDefaultMaxSessionIdleTimeout(60000); //1 min. idle session timeout  
container.connectToServer(ChatClient.class, URI.create("ws://letschat:8080")); //connecting to a websocket server

//fetching the ServerContainer instance (from within a Server endpoint)  
ServerContainer container = (ServerContainer) session.getContainer();

◆RemoteEndpoint  
WebSocket是一个通过两个对等的个体（客户端服务器端）交流的协议。  
RemoteEndpoint接口是配对的另一端的抽象

Synchronous 阻塞API 用javax.websocket.RemoteEndpoint.Basic执行  
Asynchronous 用javax.websocket.RemoteEndpoint.Async控制 调用者不会阻塞，它可以得到一个java.util.concurrent.Future返回  
或者提供一个回调实现。  
  
//Getting a handle to the remote endpoint

RemoteEndpoint.Basic basicPeerConnection = session.getBasicRemote();  
RemoteEndpoint.Async asyncPeerConnection = session.getAsyncRemote();

◆Endpoint  
javax.websocket.Endpoint代表WebSocket endpoint本身，服务器端或者客户端。  
API支持注解或者编程API。  
这个类被设计用于扩展（因为它是抽象的），并且适合于程序化样式优于注释驱动（声明性）实现的场景

//a bare bone implementation of a programmatic endpoint

public class ProgrammaticEndpointExample extends Endpoint {  
private Session session;  
@Override  
public void onOpen(Session session, EndpointConfig config) {  
this.session = session;  
try {  
//sends back the session ID to the peer  
this.session.getBasicRemote().sendText("Session ID: " + this.session.getId());  
}  
catch (IOException ex) {  
throw new RuntimeException(ex);  
}  
}

RemoteEndpoint 和 Endpoint 代表不同的视角来看一个Endpoint  
RemoteEndpoint也是一个Endpoint  
从服务器端看 自身是EndPoint 客户端是RemoteEndpoint  
从客户端看 反之

◆Session  
WebSocket会话（由javax.websocket.Session表示）的概念与HTTP会话的概念没有很大的不同：  
它封装了两个端点（对等体）之间的交互，两个端点之间的会话由连接初始化，消息交换，连接终止，错误传输等事件组成。  
・每个连接对应一个配对  
・通过javax.websocket.RemoteEndpoint来允许服务器和客户端之间交流

//some of the common methods in the Session interface

Set<Session> openSessions = session.getOpenSessions();  
boolean isOpen = session.isOpen();  
Map<String, String> pathParameters = session.getPathParameters();  
Map<String, Object> userProperties = session.getUserProperties();  
Principal userPrincipal = session.getUserPrincipal();

◆注解声明与编程模式  
注解声明 @ServerEndpoint 和 @ClientEndpoint  
编程 继承javax.websocket.Endpoint

◆编码解码器  
将它们视为由WebSocket API提供的钩子，它允许您插入您的自定义实现，  
该自定义实现负责将Java对象（部分逻辑，域模型等）转换为WebSocket支持的协议即文本和二进制

Encoder javax.websocket.Encoder Java object to text or binary  
Decoder java.websocket.Decoder text or binary to Java object

◆Configuration  
配置类只是一些属性 发布路径，编码器，解码器，扩展，副协议等。  
javax.websocket.EndpointConfig  
javax.websocket.server.ServerEndpointConfig  
javax.websocket.ClientEndpointConfig

◆异步构造  
在配对间发消息不一定是一个阻塞调用。可以选择异步方式。

SendHandler 和 SendResult(javax.websocket) 提供异步消息功能

SendHandler 用于定义消息发送完成或失败后要做的行为的接口  
SendResult 提供消息发送结果的访问

◆WebSocket Extension  
javax.websocket.Extension 代表一个扩展 有名字和参数List属性  
javax.websocket.Extension.Parameter 代表一个扩展参数 有名字和值

◆Exceptions  
DeploymentException 在部署服务器端或者客户端建立连接时发生的异常  
DecodeException 在text/binary转换java对象时发生的异常  
EncodeException 在java对象转换text/binary时发生的异常  
SessionException 表示特定的session发生了异常

◆其他  
・Path parameters  
javax.webocket.PathParam  
注入path上的参数到方法参数的注解  
//Using @PathParam

@ServerEndpoint(value = "/letschat/{login-id}")  
public class WebChatEndpoint {  
@OnOpen  
public void connected(Session session, @PathParam("login-id") String loggedInUser){  
//save the logged in user id  
session.getUserProperties().put("USERNAME", loggedInUser);  
}  
....  
}

・WebSocket connection handshake  
HandshakeRequest和HandshakeResponse的实例（来自javax.websocket pacakge）  
允许访问WebSocket连接的建立（握手）期间的通信状态

·连接终止的详细信息  
javax.websocket.CloseReason 连接终止的详细信息  
javax.websocket.CloseReason.CloseCode 结束code  
javax.websocket.CloseReason.CloseCodes 可以使用的的结束code

//Why did the connection close?

@OnClose  
public void disconnected(Session session, CloseReason reason){  
String peer = (String) session.getUserProperties().get("USERNAME");  
CloseReason.CloseCode closeReasonCode = reason.getCloseCode();  
String closeReasonMsg = reason.getReasonPhrase();  
System.out.println("User "+ peer + " disconnected. Code: "+ closeReasonCode + ", Message: "+ closeReasonMsg);  
}

◆编程模式  
・注解  
・继承

注解方式  
@ServerEndpoint 定义服务器端 指定uri  
@ClientEndpoint 定义客户端  
@OnOpen 定义在连接建立时调用的方法  
@OnMessage 定义一个endpoint接收到消息时调用的方法  
@OnError 定义出现异常时调用的方法  
@OnClose 定义连接关闭时容器调用的方法

//annotated endpoint

@ServerEndpoint("/test/")  
public class AnnotatedEndpoint {  
@OnOpen  
public void onOpenCallback(Session s, EndpointConfig ec){  
...  
}  
@OnMessage  
public void onMessageCallback(String messageFromClient){  
...  
}  
@OnClose  
public void onCloseCallback(Session s, CloseReason cr){  
...  
}  
@OnError  
public void onErrorCallback(Session s, Throwable t){  
...  
}  
}

继承方式  
Endpoint 端点接口  
MessageHandler 接收消息时的处理实现接口  
MessageHandler.Whole<T> 整条消息接口  
MessageHandler.Partial<T> 分段消息接口

//programmatic endpoint

public class ProgrammaticEndpoint extends Endpoint {  
@Override  
public onOpen(Session session, EndpointConfig config) {  
session.addMessageHandler((String s) -> System.out.println("got msg "+ s));  
....  
}  
}

◆发送消息  
消息的数据类型可以是 1.文本 2.字节 3.自定义java对象 4.ping/pong消息  
消息的发送模式可以分 1.异步 2.同步 3.分段 4.流

同步 Synchronous : 发送消息的客户端被阻塞直到消息发送过程完成或者异常发生  
异步 Asynchronous : 客户端发出消息立刻被释放，通过Future对象或者回掉函数来追踪发送的进程  
分段 Partial : 消息被分成几部分，客户端必须保持对他们的追踪并告诉API当所有的部分都发送完了  
流 Streaming : 用java的字符流发送消息

・发送文本消息  
　　同步  
　　 这是最容易理解的方式，使用RemoteEndpoint.Basic 中的public void sendText(String msg)  
//synchronous delivery

....  
@OnMessage  
public void onReceipt(String msg, Session session){  
session.getBasicRemote().sendText("got your message ");  
}

　　异步  
　　 使用RemoteEndpoint.Async  
　　 1.Future对象返回 public Future<Void> sendText(String msg)  
　　 2.提供回调函数 public void sendText(String msg, SendHandler handler)  
　　  
//asynchronous text message delivery

....  
@OnMessage  
public void onReceipt(String msg, Session session){  
Future<Void> deliveryTracker = session.getAsyncRemote().sendText("got your message ");  
deliveryTracker.isDone(); //blocks  
}

//asynchronous text message delivery using a callback

....  
@OnMessage  
public void onMsg(String msg, Session session){  
session.getAsyncRemote().sendText("got your message ", new SendHandler() {  
@Override  
public void onResult(SendResult result) {  
pushToDB(session.getID(), msg, result.isOK());  
}  
});  
}  
....

//Java 8 lambda style

....  
session.getAsyncRemote()  
.sendText("got your message ",  
(SendResult result) -> {pushToDB(session.getId(),msg, result.isOK());}  
);  
....

　　分段  
　　 通过在RemoteEndpoint.Basic接口中使用sendText方法的重载版本可以部分发送消息。 这个过程本质上是同步的

//partial message delivery

....  
String partialData = fetch(request);  
try {  
session.getBasicRemote().sendText(partialData, false); // boolean isLast  
} catch (IOException ex) {  
throw new RuntimeException(ex);  
}  
...

　　流  
　　 可以将文本（字符）数据流传输到RemoteEndpoint.Basic中的public void getSendWriter（）提供的java.io.Writer。  
　　 可以使用Writer中任何重载的写入方法  
　　 流模式是同步的

//streaming strings

....  
private Session session;

public void broadcast(String msg){  
session.getBasicRemote().getSendWriter().write(msg);  
}  
....

Text message总结  
Synchronous public void sendText(String msg)  
Asynchronous public Future<Void> sendText(String msg), void sendText(String msg, SendHandler callback)  
Partial public void sendText(String part, boolean isLast)  
Streaming public void getSendWriter().write(String msg)

・发送字节消息  
　　处理（发送）就API而言，二进制数据类似于String。唯一明显的区别是数据类型-字节是ByteBuffer而文本是String。  
　　支持的发送模式也相同。  
　　  
　　同步  
//synchronous delivery of an image

....  
public void syncImage(byte[] image, Session session){  
ByteBuffer img = ByteBuffer.wrap(image);  
session.getBasicRemote().sendBinary(img);  
}  
....  
　　  
　　异步  
//asynchronous delivery of an image

....  
public void syncLargeImage(byte[] image, Session session){  
ByteBuffer img = ByteBuffer.wrap(image);  
Future<Void> deliveryProgress = session.getAsyncRemote().sendBinary(img);  
boolean delivered = deliveryProgress.isDone(); //blocks until completion or failure  
}  
....

　　分段  
//partial delivery of binary data

....  
ByteBuffer partialData = fetch(request);  
try {  
session.getBasicRemote().sendBinary(partialData, false);  
} catch (IOException ex) {  
throw new RuntimeException(ex);  
}  
...  
　　  
　　流  
　　 在RemoteEndpoint.Basic上使用getSendStream（）方法获取一个OutputStream，并使用任何重载的写入方法来传输二进制数据  
//binary data - streaming style

....  
ByteBuffer data = fetch(request);  
try {  
session.getBasicRemote().getSendStream().write(data.array());  
} catch (IOException ex) {  
throw new RuntimeException(ex);  
}  
...

发送字节的总结  
Synchronous public void sendBinary(ByteBuffer data)  
Asynchronous public Future<Void> sendBinary(ByteBuffer data) ,  
public void sendBinary(ByteBuffer data, SendHandler callback)  
Partial public void sendBinary(ByteBuffer part, boolean isLast)  
Streaming public void getSendStream().write(byte[] data)

・发送java对象  
　　Synchronous public void sendObject(Object obj)  
　　Asynchronous public Future<Void> sendObject(Object obj),  
　　 public void sendObject(Object obj, SendHandler callback)  
　　  
....  
private Session client;

public void broadcast(String msg) {  
Set<String> subscriptions = (Set<String>) client.getUserProperties().get("TICKER\_SUBSCRIPTIONS");  
StockQuote quote = null; //the Java object  
for (String subscription : subscriptions) {  
try {  
quote = fetchQuote(subscription);  
//sending stock quotes with a Java 8 lambda style callback  
peer.getAsyncRemote().sendObject(quote,  
(SendResult result) -> {audit(session.getId(),quote, result.isOK());}  
);  
}  
catch (Exception e) {  
//log and continue...  
}  
}  
}  
....

Encoder 上层接口  
Encoder.Text<T> java对象T转换成文本 java.lang.String  
Encoder.Binary<T> java对象T转换成字节 java.nio.ByteBuffer

//encoding a 'StockQuote' Java object to a JSON string

public class StockQuoteJSONEncoder implements Encoder.Text<StockQuote> {  
@Override  
public void init(EndpointConfig config) {  
//for custom initialization logic (details omitted)  
}  
@Override  
public String encode(StockQuote stockQuoteObject) throws EncodeException {  
//using the JSON processing API (JSR 353)  
return Json.createObjectBuilder()  
.add("quote", stockQuoteObject.getQuote())  
.add("ticker", stockQuoteObject.getTicker())  
.toString();  
}  
@Override  
public void destroy() {  
//close resources (details omitted)  
}  
}

将java对象转换成流  
Encoder.TextStream<T> 转换java对象并用字符流传输 java.io.Writer  
Encoder.BinaryStream<T> 转换java对象并用字节流传输 java.io.OutputStream

//sending Java objects as character stream

public class StockQuoteJSONEncoder implements Encoder.TextStream<StockQuote> {  
@Override  
public void init(EndpointConfig config) {  
//for custom initialization logic (details omitted)  
}  
@Override  
public void encode(StockQuote stockQuoteObject, Writer writer) throws EncodeException {  
//using the JSON processing API (JSR 353)  
String jsonStockQuote = Json.createObjectBuilder()  
.add("quote", stockQuoteObject.getQuote())  
.add("ticker", stockQuoteObject.getTicker())  
.toString();  
writer.write(jsonStockQuote);  
}  
@Override  
public void destroy() {  
//close resources (details omitted)  
}  
}

所以如果你要用字节或者字符流发送java对象，你需要为这个java类实现并注册一个合适的Encoder，  
websocket运行时会自动调用。

关于基本类型，websocket实现了默认的转换，也可以自己写一个来覆盖它。

・ping-pong  
　　ping是请求消息 没有特定类 byte buffer  
　　pong是响应消息 javax.websocket.PongMessage 它也可以用作单向心跳消息（不需要ping消息）  
　　  
　　他们只是ByteBuffer的字节数据  
　　不能大于125bytes。只能用来检测状态，不应用与业务数据传输。  
　　  
　　发送方法定义在javax.websocket.RemoteEndpoint中 同步异步相同  
Synchronous & Asynchronous  
void sendPing(ByteBuffer ping)  
void sendPong(ByteBuffer pong)  
  
//sending a ping (health check request)  
.....  
private Session s;

public void healthCheck(){  
s.getBasicRemote().sendPing(ByteBuffer.wrap("health-check".getBytes()));  
}  
.....

注意点  
Ping消息只能发送（不能接收）而Pong可以发送和接收  
不需要写逻辑来明确地返回一个乒乓消息来响应一个ping - Java WebSocket API的实现会为你自动的处理  
Pong消息也可以用作自发心跳消息（不仅仅是响应ping）  
  
//sending a pong (as a one-way heart beat)

s.getBasicRemote().sendPong(ByteBuffer.wrap("health-check".getBytes()));

◆异步超时  
首先RemoteEndpoint.Async setSendTimeout  
其次失败结果在Future对象或者SendResult中

用SendResult取失败信息时使用 SendResult.getException()  
//bail out if the message is not sent in 1 second

....  
public void broadcast(Session s, String msg){  
RemoteEndpoint asyncHandle = s.getRemoteAsync();  
asyncHandle.setSendTimeout(1000); //1 second  
asyncHandle.sendText(msg,  
new SendHandler(){  
@Override  
public void onResult(SendResult result) {  
if(!result.isOK()){  
System.out.println("Async send failure: "+ result.getException());  
}  
}  
}); //will timeout after 2 seconds  
}  
....

如果用Future来追踪完成情况 调用get方法会抛出一个 java.util.concurrent.ExecutionException  
//bail out if the message is not sent in 2 seconds

....  
public void broadcast(Session s, String msg){  
RemoteEndpoint asyncHandle = s.getRemoteAsync();  
asyncHandle.setSendTimeout(2000); //2000 ms  
Future<Void> tracker = asyncHandle.sendText(msg); //will timeout after 2 seconds  
tracker.get(); //will throw java.util.ExecutionException if the process had timed out  
}  
....

Sending style Text Binary Java object Pong Ping  
Synchronous y y y y y  
Asynchronous y y y y y  
Partial y y n n n  
Streaming y y y n n

◆接收消息  
从API角度来看，发送消息比接收简单多了，因为结构简单。比如javax.websocket.RemoteEndpoint接口  
接收消息也有两种写法，注解和继承  
・注解 在于传合适的参数到@OnMessage的方法  
・继承 为javax.websocket.MessageHandler接口做合适的实现来处理接收消息的逻辑

接收模式  
・完整 整条消息接收  
・分段 这和发送端的分段发送配套使用。如果发送端分段发送，则分段接收。最后一段的标志位last为true  
・流 用Java Readers 或 InputStreams形式接收消息

·接收文本消息  
·注解  
·完整  
...  
@OnMessage  
public void handleChatMsg(String chat) {  
System.out.println("Got message - " + chat);  
}  
...

·分段  
...  
@OnMessage  
public void pushChunk(String partMsg, boolean last) {  
String chunkSeq = last ? "intermediate" : "last" ;  
System.out.println("Got " + chunkSeq + " chunk - "+ partMsg);  
}  
...

·流  
...  
@OnMessage  
public void handleChatMsg(Reader charStream) {  
System.out.println("reading char stream");  
}  
...

·继承  
·完整  
public class WholeTextMsgHandler extends MessageHandler.Whole<String> {  
@Override  
public void onMessage(String chat) {  
System.out.println("Got message - " + chat);  
}  
}

·分段  
public class PartialTextMsgHandler extends MessageHandler.Partial<String> {  
@Override  
public void onMessage(String partMsg, boolean last) {  
String chunkSeq = last ? "intermediate" : "last" ;  
System.out.println("Got " + chunkSeq + " chunk - "+ partMsg);  
}  
}

·流  
public class WholeStreamingTextMsgHandler extends MessageHandler.Whole<Reader> {  
@Override  
public void onMessage(Reader charStream) {  
System.out.println("Got stream message - " + charStream);  
}  
}

·接收字节  
字节数据支持 byte[] (array), java.nio.ByteBuffer 和 java.io.InputStream

·注解  
·完整  
...  
@OnMessage  
public void handleImage(ByteBuffer img) {  
System.out.println("Got message - " + chat);  
}  
...  
  
·分段  
...  
@OnMessage  
public void pushChunk(byte[] audioPart, boolean last) {  
String chunkSeq = last ? "intermediate" : "last" ;  
System.out.println("Got " + chunkSeq + " clip");  
}  
...  
  
·流  
...  
@OnMessage  
public void handleChatMsg(InputStream binaryStream) {  
System.out.println("reading binary stream");  
}  
...

·继承  
·完整  
public class WholeBinaryMsgHandler extends MessageHandler.Whole<byte[]> {  
@Override  
public void onMessage(byte[] image) {  
System.out.println("Got image - " + image.length);  
}  
}  
  
·分段  
public class PartialBinaryMsgHandler extends MessageHandler.Partial<ByteBuffer> {  
@Override  
public void onMessage(ByteBuffer clip, boolean last) {  
String chunkSeq = last ? "intermediate" : "last" ;  
System.out.println("Got " + chunkSeq + " chunk");  
}  
}  
  
·流  
public class WholeStreamingBinaryMsgHandler extends MessageHandler.Whole<InputStream> {  
@Override  
public void onMessage(InputStream binaryStream) {  
System.out.println("Got stream binary message");  
}  
}

◆用java对象接收消息  
使用Decoders 将text/binary数据转换为java对象

Decoder 顶层接口  
Decoder.Text<T> 定义将String转换成T类型java对象的方法  
Decoder.Binary<T> 定义将java.nio.ByteBuffer转换成T类型java对象的方法

public class StockSubscriptionDecoder implements Decoder.Text<Subscription> {

@Override  
public Subscription decode(String subscription){  
//client sends comma seperated list of subscription e.g. appl,goog,orcl  
return new Subscription(Arrarys.asList(subscription.split(",")));  
}  
@Override  
public void willDecode(String subscription){  
return subscription!=null && subscription.split(",").length > 0;  
}

}

・将Stream转换为java对象  
Decoder.TextStream<T> java.io.Reader 转换为 java对象  
Decoder.BinaryStream<T> java.io.InputStream 转换为 java对象

public class ConversationDecoder implements Decoder.TextStream<Conversation> {

@Override  
//handles new-line delimited content  
public Conversation decode(Reader content) {  
Conversation conversation = new Conversation();  
try(LineNumberReader lineByLineReader = new LineNumberReader(content)){  
String line = lineByLineReader.readLine();  
while(line != null) {  
conversation.add(line);  
line = lineByLineReader.readLine();  
}  
}  
return conversation;  
}  
}

◆处理Pong数据  
注解  
//annotated Pong handler  
...  
@OnMessage  
public void healthCheckCallback(PongMessage pong) {  
System.out.println("Pong for Ping! "+ new String(pong.getApplicationData().array());  
}  
...

继承  
//programmatic Pong handler  
public class PongMsgHandler extends MessageHandler.Whole<PongMessage> {  
@Override  
public void onMessage(PongMessage pong) {  
System.out.println("Pong for Ping! "+ new String(pong.getApplicationData().array());  
}  
}

◆使用MessageHandler  
第一步，实现针对特定对象（text/binary whole/partial）的MessageHandler接口  
第二步，用Session#addMessageHandler方法注册到websocket 可以注册多个实现

//attaching message handlers

public class ProgrammaticEndpoint extends Endpoint {

@Override  
public void onOpen(Session session, EndpointConfig config) {  
session.addMessageHandler(new WholeBinaryMsgHandler()); //basic  
session.addMessageHandler(String.class, new WholeTextMsgHandler()); //specify class type for Whole message handler  
session.addMessageHandler(ByteBuffer.class, new PartialBinaryMsgHandler()); //specify class type for Partial message handler  
}  
}

◆@OnMessage方法的其他可用参数  
除了消息内容本身，@OnMessage方法还能接收以下参数，它们将在运行时自动注入  
・带@javax.websocket.PathParam注解的String参数 个数0到多个  
・Session对象  
・EndpointConfig对象 （服务器端或客户端）

public void onMsgCallback(String theMsg, @PathParam("user") String username, Session peer, EndpointConfig condfig){  
System.out.println("I have everything I could possibly receive from the WebSocket implementation !");  
}

◆java基础类型  
基本类型的decoder有默认实现，也可自定义覆盖

◆客户端API  
・注解 使用@ClientEndpoint  
//annotated client endpoint in action

@ClientEndpoint  
public class AnnotatedChatClient {

private ClientEndpointConfig clientConfig;  
private String user;  
@OnOpen  
public void connected(Session session, EndpointConfig clientConfig){  
this.clientConfig = (ClientEndpointConfig) clientConfig;  
this.user = session.getUserPrincipal().getName();  
System.out.println("User " + user + " connected to Chat room");  
}  
@OnMessage  
public void connected(String msg){  
System.out.println("Message from chat server: " + msg);  
}  
@OnClose  
public void disconnected(Session session, CloseReason reason){  
System.out.println("User "+ user + " disconnected as a result of "+ reason.getReasonPhrase());  
}  
@OnError  
public void disconnected(Session session, Throwable error){  
System.out.println("Error communicating with server: " + error.getMessage());  
}  
}

・继承 继承java.websocket.Endpoint类  
//a bare bone implementation of a programmatic endpoint

public class WeatherClient extends Endpoint {  
private Session session;  
@Override  
public void onOpen(Session session, EndpointConfig config) {  
this.session = session;  
try {  
//sends back the session ID to the peer  
this.session.getBasicRemote().sendText("Session ID: " + this.session.getId());  
} catch (IOException ex) {  
throw new RuntimeException(ex);  
}  
}  
}

◆用客户端API连接服务器端  
使用javax.websocket.WebSocketContainer接口中的方法  
・客户端类使用注解写法时  
connectToServer(Class<?> annotatedEndpointClass, URI path)  
connectToServer(Object annotatedEndpointInstance, URI path) //annotatedEndpointInstance不支持注入  
・客户端类使用继承写法时  
connectToServer(Class<? extends Endpoint> endpointClass, ClientEndpointConfig cec, URI path)  
connectToServer(Endpoint endpointInstance, ClientEndpointConfig cec, URI path) //endpointInstance不支持注入  
  
・注解  
用class类型  
WebSockerContainer.connectToServer(AnnotatedChatClient.class, URI.create("ws://javaee-chat.com"));

用客户端实例  
WebSockerContainer.connectToServer(new AnnotatedChatClient(), URI.create("ws://javaee-chat.com"));

・继承  
比注解多需要一个ClientEndpointConfig参数  
用class类型  
WebSockerContainer.connectToServer(WeatherClient.class,  
ClientEndpointConfig.Builder.create().build(), //fluent API  
URI.create("ws://weather-tracker.com"));  
  
用客户端实例  
WebSockerContainer.connectToServer(new WeatherClient(),  
ClientEndpointConfig.Builder.create().build(), //fluent API  
URI.create("ws://weather-tracker.com"));  
  
◆配置器 Configuration  
EndpointConfig接口的实现 EndpointConfig被ServerEndpointConfig 和 ClientEndpointConfig继承

◆服务器端配置  
ServerEndpointConfig  
ServerEndpointConfig.Builder  
ServerEndpointConfig.Configurator

・注解写法的服务器端配置  
通过@ServerEndpoint的属性设置  
//annotated server endpoint with all its configuration elements

@ServerEnpdoint(  
value = "/chat/",  
configurator = ChatEndpointConfigurator.class, //discussed later  
decoders = JSONToChatObjectDecoder.class,  
encoders = ChatObjectToJSONEncoder.class,  
subprotocols = {"chat"}  
)  
public class ChatServer {  
//business logic...  
}

EndpointConfig 实例会自动注入到@OnOpen方法的参数  
//server endpoint configuration in action

@OnOpen  
public void onOpenCallback(Session session, EndpointConfig epConfig){  
ServerEndpointConfig serverConfig = (ServerEndpointConfig) epConfig;  
Map<String, Object> globalPropertiesMap = serverConfig.getUserProperties();  
......  
}

・继承写法的服务器端配置  
需要显示的编码  
ServerEndpointConfig serverConfig = ServerEndpointConfig.Builder  
.create(StockTrackerEndpoint.class , "/pop-stocks/").  
.configurator(StockTrackerConfigurator.getInstance()) //discussed later  
.decoders(JSONToStockTickerObject.class)  
.encoders(StockTickerObjectToJSON.class)  
.build();

javax.websocket.Endpoint接口的onOpen方法有ServerEndpointConfig实例参数  
public class ProgrammaticChatClient extends Endpoint {  
@Override  
public void onOpen(Session session, EndpointConfig config){  
ServerEndpointConfig serverConfig = (ServerEndpointConfig) epConfig;  
.....  
}  
}

◆客户端配置  
ClientEndPointConfig  
ClientEndpointConfig.Builder  
ClientEndpointConfig.Configurator

・注解写法的客户端配置  
@ClientEndpoint(  
configurator = ChatClientEndpointConfigurator.class, //discussed later  
decoders = JSONToChatObjectDecoder.class,  
encoders = ChatObjectToJSONEncoder.class,  
subprotocols = {"chat"}  
)  
public class ChatClient {  
//business logic...  
}

//server endpoint configuration in action

@OnOpen  
public void onOpenCallback(Session session, EndpointConfig epConfig){  
ClientEndpointConfig clientConfig = (ClientEndpointConfig) epConfig;  
......  
}

・继承写法的客户端配置  
ClientEndpointConfig cec = ClientEndpointConfig.Builder  
.configurator(ChatClientConfigurator.getInstance()) //discussed later  
.decoders(JSONToStockTickerObject.class)  
.encoders(StockTickerObjectToJSON.class)  
.build();  
  
・服务器端配置和客户端配置区别不大 除了服务端多了path或URI  
・EndpointConfig 可以保存实例间共通的属性 通过getUserProperties()方法 它返回一个map

服务器端配置的设定与取得对应关系  
@ServerEndpoint设定属性 ServerEndpointConfig取得方法 ServerEndpointConfig.Builder设定方法  
value getPath() create(Class<?> endpointClass, String path)  
configurator getConfigurator() configurator(ServerEndpointConfig.Configurator sec)  
decoders getDecoders() decoders(List<Class<? extends Decoder>> decoders)  
encoders getEncoders() encoders(List<Class<? extends Encoder>> encoders)  
subprotocols getSubprotocols() subprotocols(List<String> subprotocols)  
客户端配置的设定与取得对应关系  
@ClientEndpoint 设定属性 ClientEndpointConfig取得方法 ClientEndpointConfig.Builder设定方法  
configurator getConfigurator() configurator(ClientEndpointConfig.Configurator cec)  
decoders getDecoders() decoders(List<Class<? extends Decoder>> decoders)  
encoders getEncoders() encoders(List<Class<? extends Encoder>> encoders)  
subprotocols getPreferredSubprotocols() preferredSubprotocols(List<String> preferredSubprotocols)

◆配置器  
可以用于服务器端和客户端。可以拦截连接的握手阶段。它可以做以下的事  
1.自定义WebSocket握手过程  
2.自定义端点实例的创建过程  
3.实现绑定了这个配置器的所有端点的共通逻辑

如果不自定义，会有一个默认的配置器

・服务器端配置器  
ServerEndpointConfig.Configurator

Handshake void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest request, HandshakeResponse response)

endpoint创建 <T> T getEndpointInstance(Class<T> endpointClass)

来源check boolean checkOrigin(String originHeaderValue) // 检查握手过程中的HTTP Origin header 来做安全确认

子协议匹配 List<Extension> getNegotiatedExtensions(List<Extension> installed, List<Extension> requested)  
  
扩展匹配 String getNegotiatedSubprotocol(List<String> supported, List<String> requested)

//custom configurator

public class CustomServerEndpointConfigurator extends ServerEndpointConfig.Configurator {

@Override  
public <T> T getEndpointInstance(Class<T> endpointClass){  
//override the default behavior by providing a 'Singleton'  
return (T) StockTickerEndpoint.getInstance();  
}

@Override  
public boolean checkOrigin(String originHeaderValue){  
//just audit this  
audit(originHeaderValue);  
return true;  
}

private String user;

@Override  
public void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest request, HandshakeResponse response){  
//introspect the request headers  
System.out.println(request);

//the authenticated user  
this.user = request.getUserPrincipal().getName();

}

@Override  
public List<Extension> getNegotiatedExtensions(List<Extension> installed, List<Extension> requested){  
//invoke default implementation  
return super.getNegotiatedExtensions(installed, requested);  
}

@Override  
public String getNegotiatedSubprotocol(List<String> supported, List<String> requested){  
//invoke default implementation  
return super.getNegotiatedSubprotocol(supported, requested);  
}  
}

//declaring the custom configuration

@ServerEndpoint(value = "/letschat" , configurator = CustomServerEndpointConfigurator.class)  
public class AnnotatedServerEndpointExample {  
//call back life cycle method(s) implementation...  
}

・客户端配置器  
拦截Handshake void afterResponse(HandshakeResponse hr)  
void beforeRequest(Map<String,List<String>> headers)  
//custom configurator

public class CustomClientEndpointConfigurator extends ClientEndpointConfig.Configurator {

@Override  
public void beforeRequest(Map<String,List<String>> headers){  
//mutate the header  
String token = ...;  
headers.put("X-token" , Arrays.asList(token));  
}

@Override  
public void afterResponse(HandshakeResponse hr){  
//introspect the handshake response  
System.out.println(hr.getHeaders());  
}  
}

//declaring the client configuration

@ClientEndpoint(configurator = CustomClientEndpointConfigurator.class)  
public class AnnotatedClientEndpointExample {  
//call back life cycle method(s) implementation...  
}

◆发布  
・注解方式的服务器端会自动发布

・继承方式的服务器端需要用javax.websocket.server.ServerApplicationConfig类编写代码  
//A 'chat' club using programmatic web socket endpoint style

public class ChatClub extends Endpoint {  
....  
@Override  
public void onOpen(Session joinee, EndpointConfig config) {  
System.out.println("Peer " + joinee.getId() + " connected");  
joinee.getRemoteBasic().sendText("Welcome to the Chat Club. The first rule of Chat Club is ..... don't talk, just type");  
joinee.addMessageHandler(new MessageHandler.Whole<String>() {  
@Override  
public void onMessage(String message) {  
try {  
joinee.getBasicRemote().sendText("You sent \n"+message+"\nRead the rulez.. again!");  
} catch (IOException ex) {  
throw new RuntimeException(ex);  
}  
}  
});  
}  
....  
}

//custom implementation to guide the deployment of our programmatic endpoint

public class CustomServerAppConfigProvider implements ServerApplicationConfig {

@Override  
public Set<ServerEndpointConfig> getEndpointConfigs(Set<Class<? extends Endpoint>> endpointClasses) {  
Set<ServerEndpointConfig> result = new HashSet<>();  
for (Class epClass : endpointClasses) {  
if (epClass.equals(ChatClub.class)) {  
ServerEndpointConfig sec = ServerEndpointConfig.Builder.create(epClass, "/chatclub").build();  
result.add(sec);  
}  
}  
return result;  
}

@Override  
public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned) {  
// we do not have annotated endpoints. if we did, they will not be deployed !  
return Collections.emptySet();  
}  
}

当容器发现有ServerApplicationConfig接口的实现，就会调用它

・要发布继承方式的端点，必须使用ServerApplicationConfig  
・getEndpointConfigs 是发布继承方式的端点  
・getAnnotatedEndpointClasses 是发布注解方式的端点  
・上面代码中getAnnotatedEndpointClasses返回Collections.emptySet()因为我们假设没有注解方式的端点

只有全部使用注解方式时才不要实现getAnnotatedEndpointClasses

・同时发布注解和继承方式的服务器端点  
getAnnotatedEndpointClasses可以改成  
//return ALL the auto-detected (scanned) annotated endpoints which the container will deploy

....  
@Override  
public Set<Class<?>> getAnnotatedEndpointClasses(Set<Class<?>> scanned) {  
return scanned;  
}  
....  
这个方法可以控制发布的端点 并不是所有检测到的端点类都需要发布

◆编程方式的发布API  
javax.websocket.server.ServerContainer接口  
void addEndpoint(Class<?> endpointClass) // 追加注解的  
void addEndpoint(ServerEndpointConfig serverConfig) // 追加继承的  
  
ServerContainer的实例以不同的方式获取，具体取决于应用程序是在Servlet（Web容器）还是独立模式中执行

・Servlet容器中的用法  
从javax.servlet.ServletContext的名为"javax.websocket.server.ServerContainer"的属性来取得ServerContainer的实例  
@WebListener  
public class ServletCtxBasedDeploymentStrategy implements ServletContextListener {

@Override  
public void contextInitialized(ServletContextEvent sce){  
//obtain the instance  
ServerContainer sc = (ServerContainer) sce.getServletContext().getAttribute("javax.websocket.server.ServerContainer");

//trigger endpoint deployment  
deployAnnotatedEndpoint(sc);  
deployProgEndpoint(sc);  
}

private void deployAnnotatedEndpoint(ServerContainer container) {  
container.addEndpoint(StockTicker.class);  
container.addEndpoint(WeatherTracker.class);  
}

private void deployProgEndpoint(ServerContainer container) {  
container.addEndpoint(ServerEndpointConfig.Builder.create(ChatClub.class, "/chatclub").build());  
container.addEndpoint(ServerEndpointConfig.Builder.create(RealTimeLocationTracker.class, "/location").build());  
}

@Override  
public void contextDestroyed(ServletContextEvent sce){}  
}

・独立模式中的用法  
ServerContainer的取得取决于特定的容器是如何取得实例的  
不能与ServerApplicationConfig的方式混合使用  
将忽略重复的端点（要各自的部署技术实现），这是规范要求的