# 【第12章-WebSocket】

**TODO 程序没有运行起来**

如果你有跟进Web 技术的最新进展，你很可能就遇到过“实时Web”这个短语，而如果你在工程领域中有实时应用程序的实战经验，那么你可能有点怀疑这个术语到底意味着什么。

因此，让我们首先澄清，这里并不是指所谓的硬实时服务质量（QoS），硬实时服务质量是保证计算结果将在指定的时间间隔内被递交。仅HTTP 的请求/响应模式设计就使得其很难被支持，从过去所设计的各种方案中都没有提供一种能够提供令人满意的解决方案的事实中便可见一斑。

虽然已经有了一些关于正式定义实时Web服务语义的学术讨论，但是被普遍接受的定义似乎还未出现。因此现在我们将采纳下面来自维基百科的非权威性描述：

**实时Web 利用技术和实践，使用户在信息的作者发布信息之后就能够立即收到信息，而不需要他们或者他们的软件周期性地检查信息源以获取更新。**

简而言之，虽然全面的实时Web可能并不会马上到来，但是它背后的想法却助长了对于几乎瞬时获得信息的期望。我们将在本章中讨论的WebSocket协议便是在这个方向上迈出的坚实的一步。

## 12.1 WebSocket 简介

WebSocket 协议是完全重新设计的协议，旨在为Web 上的双向数据传输问题提供一个切实可行的解决方案，使得客户端和服务器之间可以在任意时刻传输消息，因此，这也就要求它们异步地处理消息回执。（作为HTML5 客户端API 的一部分，大部分最新的浏览器都已经支持了WebSocket。）

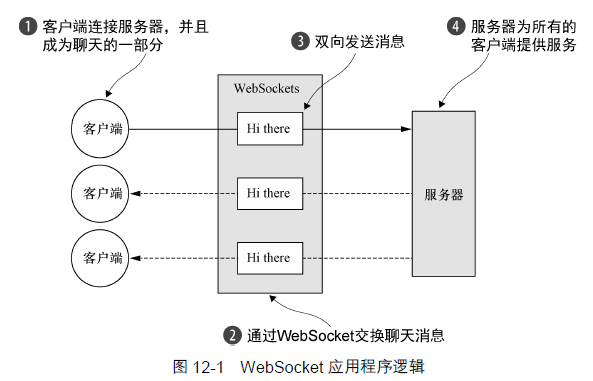
Netty 对于WebSocket 的支持包含了所有正在使用中的主要实现，因此在你的下一个应用程序中采用它将是简单直接的。和往常使用Netty 一样，你可以完全使用该协议，而无需关心它内部的实现细节。我们将通过创建一个基于WebSocket 的实时聊天应用程序来演示这一点。

## 12.2 我们的WebSocket 示例应用程序

为了让示例应用程序展示它的实时功能，我们将通过使用WebSocket 协议来实现一个基于浏览器的聊天应用程序，就像你可能在Facebook 的文本消息功能中见到过的那样。我们将通过使得多个用户之间可以同时进行相互通信，从而更进一步。

图12-1 说明了该应用程序的逻辑：

1. 客户端发送一个消息；
2. 该消息将被广播到所有其他连接的客户端。

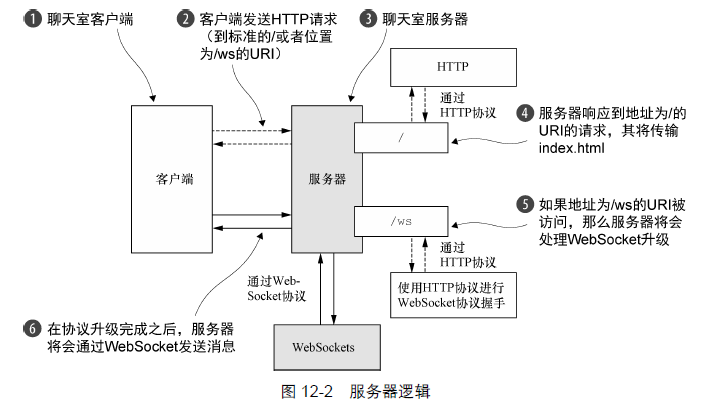


这正如你可能会预期的一个聊天室应当的工作方式：所有的人都可以和其他的人聊天。在示例中，我们将只实现服务器端，而客户端则是通过Web 页面访问该聊天室的浏览器。正如同你将在接下来的几页中所看到的，WebSocket 简化了编写这样的服务器的过程。

## 12.3 添加WebSocket 支持

在从标准的HTTP或者HTTPS协议切换到WebSocket时，将会使用一种称为升级握手的机制。因此，使用WebSocket的应用程序将始终以HTTP/S作为开始，然后再执行升级。这个升级动作发生的确切时刻特定于应用程序；它可能会发生在启动时，也可能会发生在请求了某个特定的URL之后。

我们的应用程序将采用下面的约定：如果被请求的URL 以/ws 结尾，那么我们将会把该协议升级为WebSocket；否则，服务器将使用基本的HTTP/S。在连接已经升级完成之后，所有数据都将会使用WebSocket 进行传输。图12-2 说明了该服务器逻辑，一如在Netty 中一样，它由一组ChannelHandler 实现。我们将会在下一节中，解释用于处理HTTP 以及WebSocket 协议的技术时，描述它们。



### 12.3.1 处理HTTP 请求

首先，我们将实现该处理HTTP 请求的组件。这个组件将提供用于访问聊天室并显示由连接的客户端发送的消息的网页。代码清单12-1 给出了这个HttpRequestHandler 对应的代码，其扩展了SimpleChannelInboundHandler 以处理FullHttpRequest 消息。需要注意的是，channelRead0()方法的实现是如何转发任何目标URI 为/ws 的请求的。

|  |
| --- |
| *// 代码清单12-1 HTTPRequestHandler*  *// 扩展SimpleChannelInboundHandler 以处理FullHttpRequest 消息* **public class** HttpRequestHandler **extends** SimpleChannelInboundHandler<FullHttpRequest> {  **private final** String **wsUri**;  **private static final** File ***INDEX***;   **static** {  URL location = HttpRequestHandler.**class**.getProtectionDomain().getCodeSource().getLocation();  **try** {  String path = location.toURI() + **"index.html"**;  path = !path.contains(**"file:"**) ? path : path.substring(**5**);  ***INDEX*** = **new** File(path);  } **catch** (URISyntaxException e) {  **throw new** IllegalStateException(**"Unable to locate index.html"**, e);  }  }   **public** HttpRequestHandler(String wsUri) {  **this**.**wsUri** = wsUri;  }   @Override  **public void** channelRead0(ChannelHandlerContext ctx,  FullHttpRequest request) **throws** Exception {  *// 如果请求了WebSocket协议升级，则增加引用计数（调用retain()方法），  // 并将它传递给下一个ChannelInboundHandler* **if** (**wsUri**.equalsIgnoreCase(request.getUri())) {  ctx.fireChannelRead(request.retain());  } **else** {   *// 处理100 Continue请求以符合HTTP 1.1 规范* **if** (HttpHeaders.*is100ContinueExpected*(request)) {  *send100Continue*(ctx);  }  *// 读取index.html* RandomAccessFile file = **new** RandomAccessFile(***INDEX***, **"r"**);  HttpResponse response = **new** DefaultHttpResponse(request.getProtocolVersion(), HttpResponseStatus.***OK***);  response.headers().set(HttpHeaders.Names.***CONTENT\_TYPE***, **"text/html; charset=UTF-8"**);  **boolean** keepAlive = HttpHeaders.*isKeepAlive*(request);  *// 如果请求了keep-alive，则添加所需要的HTTP头信息* **if** (keepAlive) {  response.headers().set(HttpHeaders.Names.***CONTENT\_LENGTH***, file.length());  response.headers().set(HttpHeaders.Names.***CONNECTION***, HttpHeaders.Values.***KEEP\_ALIVE***);  }  *// 将HttpResponse写到客户端* ctx.write(response);  *// 将index.html写到客户端* **if** (ctx.pipeline().get(SslHandler.**class**) == **null**) {  ctx.write(**new** DefaultFileRegion(file.getChannel(), **0**, file.length()));  } **else** {  ctx.write(**new** ChunkedNioFile(file.getChannel()));  }   *// 写LastHttpContent并冲刷至客户端* ChannelFuture future = ctx.writeAndFlush(LastHttpContent.***EMPTY\_LAST\_CONTENT***);  *// 如果没有请求keep-alive，则在写操作完成后关闭Channel* **if** (!keepAlive) {  future.addListener(ChannelFutureListener.***CLOSE***);  }  }  }   **private static void** send100Continue(ChannelHandlerContext ctx) {  FullHttpResponse response = **new** DefaultFullHttpResponse(  HttpVersion.***HTTP\_1\_1***, HttpResponseStatus.***CONTINUE***);  ctx.writeAndFlush(response);  }   @Override  **public void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  **throws** Exception {  cause.printStackTrace();  ctx.close();  } } |

如果该HTTP 请求指向了地址为/ws 的URI，那么HttpRequestHandler 将调用FullHttpRequest 对象上的retain()方法，并通过调用fireChannelRead(msg)方法将它转发给下一个ChannelInboundHandler 。之所以需要调用retain()方法，是因为调用channelRead()方法完成之后，它将调用FullHttpRequest 对象上的release()方法以释放它的资源。（参见我们在第6 章中对于SimpleChannelInboundHandler 的讨论。）

如果客户端发送了HTTP 1.1 的HTTP 头信息Expect: 100-continue，那么HttpRequestHandler 将会发送一个100 Continue 响应。在该HTTP 头信息被设置之后，HttpRequestHandler 将会写回一个HttpResponse 给客户端。这不是一个FullHttpResponse，因为它只是响应的第一个部分。此外，这里也不会调用writeAndFlush()方法，在结束的时候才会调用。

如果不需要加密和压缩，那么可以通过将index.html 的内容存储到DefaultFileRegion 中来达到最佳效率。这将会利用零拷贝特性来进行内容的传输。为此，你可以检查一下，是否有SslHandler 存在于在ChannelPipeline 中。否则，你可以使用ChunkedNioFile。

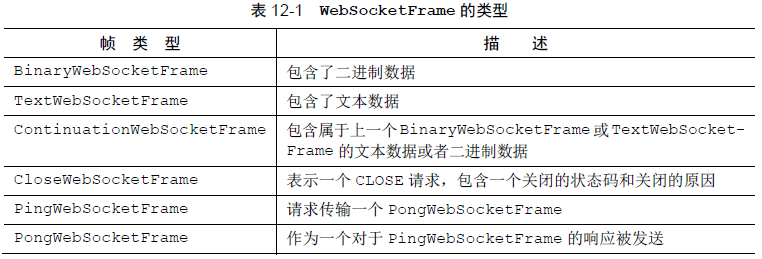
HttpRequestHandler 将写一个LastHttpContent 来标记响应的结束。如果没有请求keep-alive ，那么HttpRequestHandler 将会添加一个ChannelFutureListener到最后一次写出动作的ChannelFuture，并关闭该连接。在这里，你将调用writeAndFlush()方法以冲刷所有之前写入的消息。

这部分代码代表了聊天服务器的第一个部分，它管理纯粹的HTTP 请求和响应。接下来，我们将处理传输实际聊天消息的WebSocket 帧。

**WEBSOCKET 帧 WebSocket 以帧的方式传输数据，每一帧代表消息的一部分。一个完整的消息可能会包含许多帧。**

### 12.3.2 处理WebSocket 帧

由IETF 发布的WebSocket RFC，定义了6 种帧，Netty 为它们每种都提供了一个POJO 实现。表12-1 列出了这些帧类型，并描述了它们的用法。



我们的聊天应用程序将使用下面几种帧类型：

* CloseWebSocketFrame；
* PingWebSocketFrame；
* PongWebSocketFrame；
* TextWebSocketFrame。

TextWebSocketFrame 是我们唯一真正需要处理的帧类型。为了符合WebSocket RFC，Netty 提供了WebSocketServerProtocolHandler 来处理其他类型的帧。

代码清单12-2 展示了我们用于处理TextWebSocketFrame 的ChannelInboundHandler，其还将在它的ChannelGroup 中跟踪所有活动的WebSocket 连接。

|  |
| --- |
| *// 代码清单12-2 处理文本帧*  *// 扩展SimpleChannelInboundHandler，并处理TextWebSocketFrame 消息* **public class** TextWebSocketFrameHandler **extends** SimpleChannelInboundHandler<TextWebSocketFrame> {  **private final** ChannelGroup **group**;   **public** TextWebSocketFrameHandler(ChannelGroup group) {  **this**.**group** = group;  }   *// 重写userEventTriggered()方法以处理自定义事件* @Override  **public void** userEventTriggered(ChannelHandlerContext ctx, Object evt) **throws** Exception {  **if** (evt == WebSocketServerProtocolHandler.ServerHandshakeStateEvent.***HANDSHAKE\_COMPLETE***) {  *// 如果该事件表示握手成功，则从该Channelipeline中移除Http  // RequestHandler，因为将不会接收到任何HTTP 消息了* ctx.pipeline().remove(HttpRequestHandler.**class**);  *// 通知所有已经连接的WebSocket 客户端新的客户端已经连接上了* **group**.writeAndFlush(**new** TextWebSocketFrame(**"Client "** + ctx.channel() + **" joined"**));  *// 将新的WebSocket Channel添加到ChannelGroup 中，以便它可以接收到所有的消息* **group**.add(ctx.channel());  } **else** {  **super**.userEventTriggered(ctx, evt);  }  }   @Override  **public void** channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) **throws** Exception {  *// 增加消息的引用计数，并将它写到ChannelGroup 中所有已经连接的客户端* **group**.writeAndFlush(msg.retain());  } } |

TextWebSocketFrameHandler 只有一组非常少量的责任。当和新客户端的WebSocket握手成功完成之后，它将通过把通知消息写到ChannelGroup 中的所有Channel 来通知所有已经连接的客户端，然后它将把这个新Channel 加入到该ChannelGroup 中。

如果接收到了TextWebSocketFrame 消息，TextWebSocketFrameHandler 将调用TextWebSocketFrame 消息上的retain()方法，并使用writeAndFlush()方法来将它传输给ChannelGroup，以便所有已经连接的WebSocket Channel 都将接收到它。

和之前一样，对于retain()方法的调用是必需的，因为当channelRead0()方法返回时，TextWebSocketFrame 的引用计数将会被减少。由于所有的操作都是异步的，因此，writeAndFlush()方法可能会在channelRead0()方法返回之后完成，而且它绝对不能访问一个已经失效的引用。

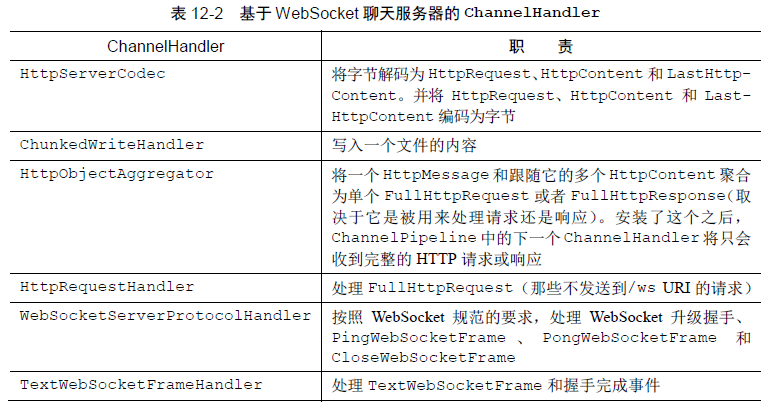
因为Netty 在内部处理了大部分剩下的功能，所以现在剩下唯一需要做的事情就是为每个新创建的Channel 初始化其ChannelPipeline。为此，我们将需要一个ChannelInitializer。

### 12.3.3 初始化ChannelPipeline

正如你已经学习到的，为了将ChannelHandler 安装到ChannelPipeline 中，你扩展了ChannelInitializer，并实现了initChannel()方法。代码清单12-3 展示了由此生成的ChatServerInitializer 的代码。

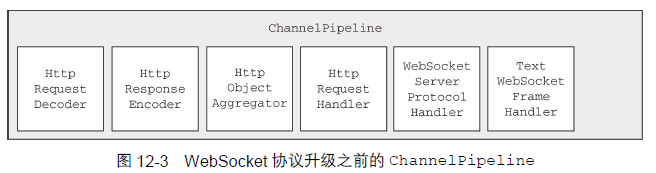
|  |
| --- |
| *// 代码清单12-3 初始化ChannelPipeline*  *// 扩展了ChannelInitializer* **public class** ChatServerInitializer **extends** ChannelInitializer<Channel> {  **private final** ChannelGroup **group**;   **public** ChatServerInitializer(ChannelGroup group) {  **this**.**group** = group;  }   *// 将所有需要的ChannelHandler 添加到ChannelPipeline 中* @Override  **protected void** initChannel(Channel ch) **throws** Exception {  ChannelPipeline pipeline = ch.pipeline();  pipeline.addLast(**new** HttpServerCodec());  pipeline.addLast(**new** ChunkedWriteHandler());  pipeline.addLast(**new** HttpObjectAggregator(**64** \* **1024**));  pipeline.addLast(**new** HttpRequestHandler(**"/ws"**));  pipeline.addLast(**new** WebSocketServerProtocolHandler(**"/ws"**));  pipeline.addLast(**new** TextWebSocketFrameHandler(**group**));  } } |

对于initChannel()方法的调用，通过安装所有必需的ChannelHandler 来设置该新注册的Channel 的ChannelPipeline。这些ChannelHandler 以及它们各自的职责都被总结在了表12-2 中。



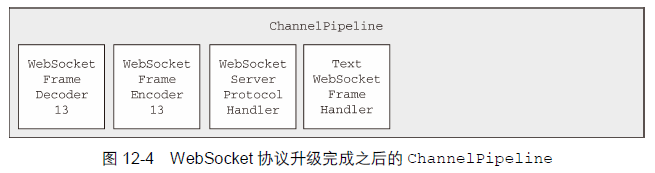
Netty 的WebSocketServerProtocolHandler 处理了所有委托管理的WebSocket 帧类型以及升级握手本身。如果握手成功，那么所需的ChannelHandler 将会被添加到ChannelPipeline中，而那些不再需要的ChannelHandler 则将会被移除。

WebSocket 协议升级之前的ChannelPipeline 的状态如图12-3 所示。这代表了刚刚被ChatServerInitializer 初始化之后的ChannelPipeline。



当WebSocket 协议升级完成之后，WebSocketServerProtocolHandler 将会把HttpRequestDecoder 替换为WebSocketFrameDecoder，把HttpResponseEncoder 替换为WebSocketFrameEncoder。为了性能最大化，它将移除任何不再被WebSocket 连接所需要的ChannelHandler。这也包括了图12-3 所示的HttpObjectAggregator 和HttpRequestHandler。

图12-4 展示了这些操作完成之后的ChannelPipeline。需要注意的是，Netty目前支持4个版本的WebSocket协议，它们每个都具有自己的实现类。Netty将会根据客户端（这里指浏览器）所支持的版本，自动地选择正确版本的WebSocketFrameDecoder和WebSocketFrameEncoder。



### 12.3.4 引导

这幅拼图最后的一部分是引导该服务器，并安装ChatServerInitializer 的代码。这将由ChatServer 类处理，如代码清单12-4 所示。

|  |
| --- |
| **// 代码清单12-5 编译并运行ChatServer**  **public class** ChatServer {  *// 创建DefaultChannelGroup，其将保存所有已经连接的WebSocket Channel* **private final** ChannelGroup **channelGroup** =  **new** DefaultChannelGroup(ImmediateEventExecutor.***INSTANCE***);  **private final** EventLoopGroup **group** = **new** NioEventLoopGroup();  **private** Channel **channel**;   **public** ChannelFuture start(InetSocketAddress address) {  *// 引导服务器* ServerBootstrap bootstrap = **new** ServerBootstrap();  bootstrap.group(**group**)  .channel(NioServerSocketChannel.**class**)  .childHandler(createInitializer(**channelGroup**));  ChannelFuture future = bootstrap.bind(address);  future.syncUninterruptibly();  **channel** = future.channel();  **return** future;  }   *// 创建ChatServerInitializer* **protected** ChannelInitializer<Channel> createInitializer(ChannelGroup group) {  **return new** ChatServerInitializer(group);  }   *// 处理服务器关闭，并释放所有的资源* **public void** destroy() {  **if** (**channel** != **null**) {  **channel**.close();  }  **channelGroup**.close();  **group**.shutdownGracefully();  }   **public static void** main(String[] args) **throws** Exception {  **if** (args.**length** != **1**) {  System.***err***.println(**"Please give port as argument"**);  System.*exit*(**1**);  }  **int** port = Integer.*parseInt*(args[**0**]);  **final** ChatServer endpoint = **new** ChatServer();  ChannelFuture future = endpoint.start(  **new** InetSocketAddress(port));  Runtime.*getRuntime*().addShutdownHook(**new** Thread() {  @Override  **public void** run() {  endpoint.destroy();  }  });  future.channel().closeFuture().syncUninterruptibly();  } } |

### 12.3.5 如何进行加密

在真实世界的场景中，你将很快就会被要求向该服务器添加加密。使用Netty，这不过是将一个SslHandler 添加到ChannelPipeline 中，并配置它的问题。代码清单12-6 展示了如何通过扩展我们的ChatServerInitializer 来创建一个SecureChatServerInitializer 以完成这个需求。

|  |
| --- |
| *// 代码清单12-6 为ChannelPipeline 添加加密*  *// 扩展ChatServerInitializer以添加加密* **public class** SecureChatServerInitializer **extends** ChatServerInitializer {  **private final** SslContext **context**;   **public** SecureChatServerInitializer(ChannelGroup group, SslContext context) {  **super**(group);  **this**.**context** = context;  }   @Override  **protected void** initChannel(Channel ch) **throws** Exception {  *// 调用父 类的initChannel()方法* **super**.initChannel(ch);  *// 将SslHandler 添加到ChannelPipeline 中* SSLEngine engine = **context**.newEngine(ch.alloc());  engine.setUseClientMode(**false**);  ch.pipeline().addFirst(**new** SslHandler(engine));  } } |

最后一步是调整ChatServer 以使用SecureChatServerInitializer，以便在Channel-Pipeline 中安装SslHandler。这给了我们代码清单12-7 中所展示的SecureChatServer。

|  |
| --- |
| *// 代码清单12-7 向ChatServer 添加加密*  *// SecureChatServer 扩展ChatServer 以支持加密* **public class** SecureChatServer **extends** ChatServer {  **private final** SslContext **context**;   **public** SecureChatServer(SslContext context) {  **this**.**context** = context;  }   @Override  **protected** ChannelInitializer<Channel> createInitializer(ChannelGroup group) {  *// 返回之前创建的SecureChatServerInitializer 以启用加密* **return new** SecureChatServerInitializer(group, **context**);  }   **public static void** main(String[] args) **throws** Exception {  **if** (args.**length** != **1**) {  System.***err***.println(**"Please give port as argument"**);  System.*exit*(**1**);  }  **int** port = Integer.*parseInt*(args[**0**]);  SelfSignedCertificate cert = **new** SelfSignedCertificate();  SslContext context = SslContext.*newServerContext*(  cert.certificate(), cert.privateKey());  **final** SecureChatServer endpoint = **new** SecureChatServer(context);  ChannelFuture future = endpoint.start(**new** InetSocketAddress(port));  Runtime.*getRuntime*().addShutdownHook(**new** Thread() {  @Override  **public void** run() {  endpoint.destroy();  }  });  future.channel().closeFuture().syncUninterruptibly();  } } |