Netty In Action中文版 - 第十一章：WebSocket

目录 [-]

本章介绍

- WebSocket
- ChannelHandler,Decoder and Encoder
- 引导一个Netty基础程序
- 测试WebSocket

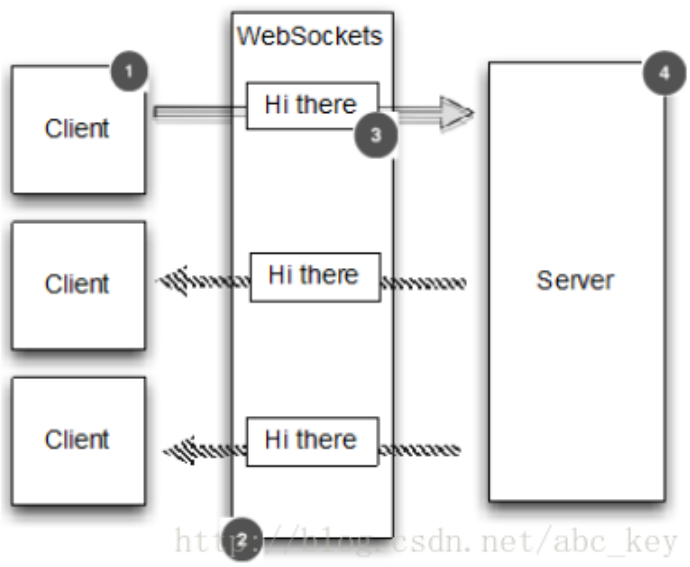
“real-time-web”实时web现在随处可见，很多的用户希望能从web站点实时获取信息。Netty支持WebSocket实现，并包含了不同的版本，我们可以非常容易的实现WebSocket应用。使用Netty附带的WebSocket，我们不需要关注协议内部实现，只需要使用Netty提供的一些简单的方法就可以实现。本章将通过的例子应用帮助你来使用WebSocket并了解它是如何工作。

11.1 WebSockets some background

关于WebSocket的一些概念和背景，可以查询网上相关介绍。这里不赘述。

11.2 面临的挑战

要显示“real-time”支持的WebSocket，应用程序将显示如何使用Netty中的WebSocket实现一个在浏览器中进行聊天的IRC应用程序。你可能知道从Facebook可以发送文本消息到另一个人，在这里，我们将进一步了解其实现。在这个应用程序中，不同的用户可以同时交谈，非常像IRC(Internet Relay Chat，互联网中继聊天)。



上图显示的逻辑很简单：

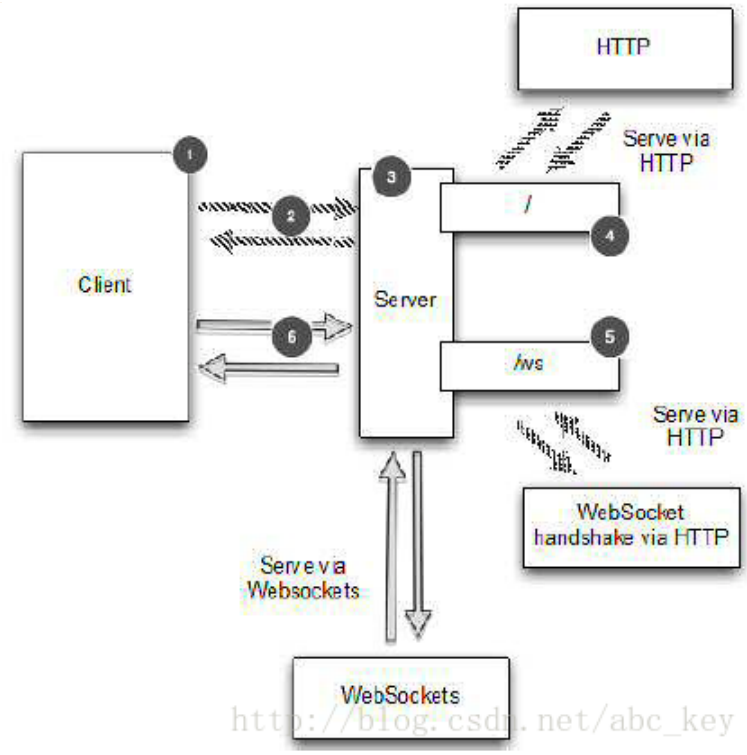
1. 一个客户端发送一条消息
2. 消息被广播到其他已连接的客户端

它的工作原理就像聊天室一样，在这里例子中，我们将编写服务器，然后使用浏览器作为客户端。带着这样的思路，我们将会很简单的实现它。

11.3 实现

WebSocket使用HTTP升级机制从一个普通的HTTP连接WebSocket，因为这个应用程序使用WebSocket总是开始于HTTP(s)，然后再升级。什么时候升级取决于应用程序本身。直接执行升级作为第一个操作一般是使用特定的url请求。

在这里，如果url的结尾以/ws结束，我们将只会升级到WebSocket，否则服务器将发送一个网页给客户端。升级后的连接将通过WebSocket传输所有数据。逻辑图如下：



11.3.1 处理http请求

服务器将作为一种混合式以允许同时处理http和websocket，所以服务器还需要html页面，html用来充当客户端角色，连接服务器并交互消息。因此，如果客户端不发送/ws的uri，我们需要写一个ChannelInboundHandler用来处理FullHttpRequest。看下面代码：

```java
package netty.in.action;

import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.DefaultFileRegion;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.DefaultFullHttpResponse;
import io.netty.handler.codec.http.DefaultHttpResponse;
import io.netty.handler.codec.http.FullHttpRequest;
import io.netty.handler.codec.http.FullHttpResponse;
import io.netty.handler.codec.http.HttpHeaders;
import io.netty.handler.codec.http.HttpResponse;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.netty.handler.codec.http.HttpVersion;
import io.netty.handler.codec.http.LastHttpContent;
import io.netty.handler.ssl.SslHandler;
import io.netty.handler.stream.ChunkedNioFile;

import java.io.RandomAccessFile;

/**
 * WebSocket，处理http请求
 *
 * @author c.k
 *
 */
public class HttpRequestHandler extends
        SimpleChannelInboundHandler<FullHttpRequest> {
    //websocket标识
    private final String wsUri;

    public HttpRequestHandler(String wsUri) {
        this.wsUri = wsUri;
    }

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest msg)
            throws Exception {
        //如果是websocket请求，请求地址uri等于wsuri
        if (wsUri.equalsIgnoreCase(msg.getUri())) {
            //将消息转发到下一个ChannelHandler
            ctx.fireChannelRead(msg.retain());
        } else {//如果不是websocket请求
            if (HttpHeaders.is100ContinueExpected(msg)) {
                //如果HTTP请求头部包含Expect: 100-continue，
                //则响应请求
                FullHttpResponse response = new DefaultFullHttpResponse(
                        HttpVersion.HTTP_1_1, HttpResponseStatus.CONTINUE);
```

```
                ctx.writeAndFlush(response);
            }
            //获取index.html的内容响应给客户端
            RandomAccessFile file = new RandomAccessFile(
                    System.getProperty("user.dir") + "/index.html", "r");
            HttpResponse response = new DefaultHttpResponse(
                    msg.getProtocolVersion(), HttpResponseStatus.OK);
            response.headers().set(HttpHeaders.Names.CONTENT_TYPE,
                    "text/html; charset=UTF-8");
            boolean keepAlive = HttpHeaders.isKeepAlive(msg);
            //如果http请求保持活跃，设置http请求头部信息
            //并响应请求
            if (keepAlive) {
                response.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
                        file.length());
                response.headers().set(HttpHeaders.Names.CONNECTION,
                        HttpHeaders.Values.KEEP_ALIVE);
            }
            ctx.write(response);
            //如果不是https请求，将index.html内容写入通道
            if (ctx.pipeline().get(SslHandler.class) == null) {
                ctx.write(new DefaultFileRegion(file.getChannel(), 0, file
                        .length()));
            } else {
                ctx.write(new ChunkedNioFile(file.getChannel()));
            }
            //标识响应内容结束并刷新通道
            ChannelFuture future = ctx
                    .writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT);
            if (!keepAlive) {
                //如果http请求不活跃，关闭http连接
                future.addListener(ChannelFutureListener.CLOSE);
            }
            file.close();
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
            throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

11.3.2 处理WebSocket框架

WebSocket支持6种不同框架，如下图：

Name	Description
BinaryWebSocketFrame	WebSocketFrame that contains binary data
TextWebSocketFrame	WebSocketFrame that contains text data
ContinuationWebSocketFrame	WebSocketFrame that contains text or binary data that belongs to a previous BinaryWebSocketFrame or TextWebSocketFrame
CloseWebSocketFrame	WebSocketFrame that represent a CLOSE request and contains close status code and a phrase
PingWebSocketFrame	WebSocketFrame which request the send of a PongWebSocketFrame
PongWebSocketFrame	WebSocketFrame which is send as response of a PingWebSocketFrame

我们的程序只需要使用下面4个框架：

- CloseWebSocketFrame
- PingWebSocketFrame
- PongWebSocketFrame
- TextWebSocketFrame

我们只需要显示处理TextWebSocketFrame，其他的会自动由WebSocketServerProtocolHandler处理，看下面代码：

```java
package netty.in.action;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.channel.group.ChannelGroup;
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;
import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler;

/**
 * WebSocket，处理消息
 * @author c.k
 *
 */
public class TextWebSocketFrameHandler extends
        SimpleChannelInboundHandler<TextWebSocketFrame> {
    private final ChannelGroup group;

    public TextWebSocketFrameHandler(ChannelGroup group) {
        this.group = group;
    }

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt)
            throws Exception {
        //如果WebSocket握手完成
        if (evt == WebSocketServerProtocolHandler.ServerHandshakeStateEvent.HANDSHAKE_COMPLETE) {
            //删除ChannelPipeline中的HttpRequestHandler
            ctx.pipeline().remove(HttpRequestHandler.class);
            //写一个消息到ChannelGroup
            group.writeAndFlush(new TextWebSocketFrame("Client " + ctx.channel()
                    + " joined"));
            //将Channel添加到ChannelGroup
            group.add(ctx.channel());
        }else {
            super.userEventTriggered(ctx, evt);
        }
    }

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg)
            throws Exception {
        //将接收的消息通过ChannelGroup转发到所以已连接的客户端
        group.writeAndFlush(msg.retain());
    }
}
```

11.3.3 初始化ChannelPipeline

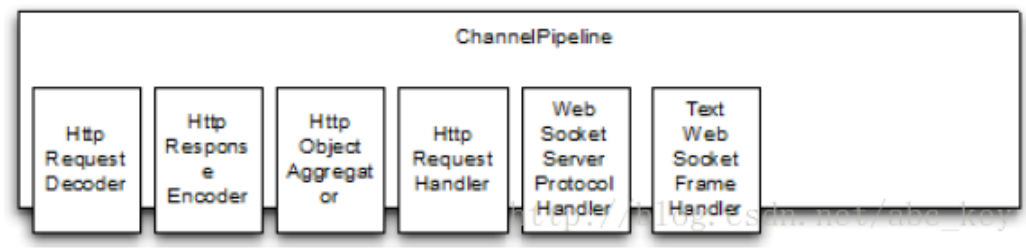
看下面代码：

```java
package netty.in.action;

import io.netty.channel.Channel;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.group.ChannelGroup;
import io.netty.handler.codec.http.HttpObjectAggregator;
import io.netty.handler.codec.http.HttpServerCodec;
import io.netty.handler.codec.http.websocketx.WebSocketServerProtocolHandler;
import io.netty.handler.stream.ChunkedWriteHandler;

/**
 * WebSocket,初始化ChannelHandler
 * @author c.k
 *
 */
public class ChatServerInitializer extends ChannelInitializer<Channel> {
    private final ChannelGroup group;

    public ChatServerInitializer(ChannelGroup group){
        this.group = group;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
```

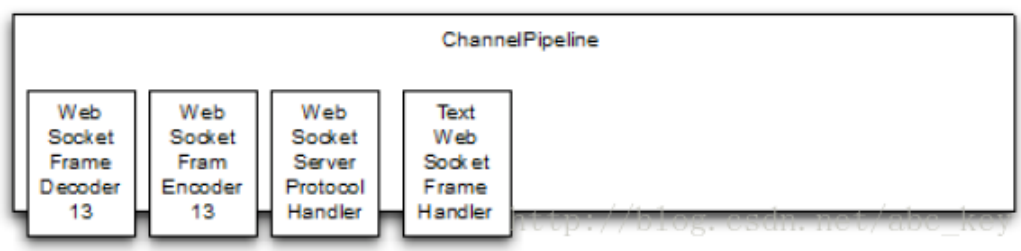
```
27.         //编解码http请求
28.         pipeline.addLast(new HttpServerCodec());
29.         //写文件内容
30.         pipeline.addLast(new ChunkedWriteHandler());
31.         //聚合解码HttpRequest/HttpContent/LastHttpContent到FullHttpRequest
32.         //保证接收的Http请求的完整性
33.         pipeline.addLast(new HttpObjectAggregator(64 * 1024));
34.         //处理FullHttpRequest
35.         pipeline.addLast(new HttpRequestHandler("/ws"));
36.         //处理其他的WebSocketFrame
37.         pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
38.         //处理TextWebSocketFrame
39.         pipeline.addLast(new TextWebSocketFrameHandler(group));
40.     }
41.
42. }
```

WebSocketServerProtcolHandler不仅处理Ping/Pong/CloseWebSocketFrame，还和它自己握手并帮助升级WebSocket。这是执行完成握手和成功修改ChannelPipeline，并且添加需要的编码器/解码器和删除不需要的ChannelHandler。

看下图：



ChannelPipeline通过ChannelInitializer的initChannel(...)方法完成初始化，完成握手后就会更改事情。一旦这样做了，WebSocketServerProtocolHandler将取代HttpRequestDecoder、WebSocketFrameDecoder13和HttpResponseEncoder、WebSocketFrameEncoder13。另外也要删除所有不需要的ChannelHandler已获得最佳性能。这些都是HttpObjectAggregator和HttpRequestHandler。下图显示ChannelPipeline握手完成：



我们甚至没注意到它，因为它是在底层执行的。以非常灵活的方式动态更新ChannelPipeline让单独的任务在不同的ChannelHandler中实现。

11.4 结合在一起使用

一如既往，我们要将它们结合在一起使用。使用Bootstrap引导服务器和设置正确的ChannelInitializer。看下面代码：

```
[java]

01. package netty.in.action;
02.
03. import io.netty.bootstrap.ServerBootstrap;
04. import io.netty.channel.Channel;
05. import io.netty.channel.ChannelFuture;
06. import io.netty.channel.ChannelInitializer;
07. import io.netty.channel.EventLoopGroup;
08. import io.netty.channel.group.ChannelGroup;
09. import io.netty.channel.group.DefaultChannelGroup;
10. import io.netty.channel.nio.NioEventLoopGroup;
11. import io.netty.channel.socket.nio.NioServerSocketChannel;
12. import io.netty.util.concurrent.ImmediateEventExecutor;
13.
14. import java.net.InetSocketAddress;
15.
16. /**
17.  * 访问地址：http://localhost:2048
18.  *
19.  * @author c.k
20.  *
21.  */
22. public class ChatServer {
23.
24.     private final ChannelGroup group = new DefaultChannelGroup(
25.             ImmediateEventExecutor.INSTANCE);
26.     private final EventLoopGroup workerGroup = new NioEventLoopGroup();
27.     private Channel channel;
```

```
28.
29.     public ChannelFuture start(InetSocketAddress address) {
30.         ServerBootstrap b = new ServerBootstrap();
31.         b.group(workerGroup).channel(NioServerSocketChannel.class)
32.                 .childHandler(createInitializer(group));
33.         ChannelFuture f = b.bind(address).syncUninterruptibly();
34.         channel = f.channel();
35.         return f;
36.     }
37.
38.     public void destroy() {
39.         if (channel != null)
40.             channel.close();
41.         group.close();
42.         workerGroup.shutdownGracefully();
43.     }
44.
45.     protected ChannelInitializer<Channel> createInitializer(ChannelGroup group) {
46.         return new ChatServerInitializer(group);
47.     }
48.
49.     public static void main(String[] args) {
50.         final ChatServer server = new ChatServer();
51.         ChannelFuture f = server.start(new InetSocketAddress(2048));
52.         Runtime.getRuntime().addShutdownHook(new Thread() {
53.             @Override
54.             public void run() {
55.                 server.destroy();
56.             }
57.         });
58.         f.channel().closeFuture().syncUninterruptibly();
59.     }
60.
61. }
```

另外，需要将index.html文件放在项目根目录，index.html内容如下：

```
[html]

01. <html>
02. <head>
03. <title>Web Socket Test</title>
04. </head>
05. <body>
06. <script type="text/javascript">
07. var socket;
08. if (!window.WebSocket) {
09.   window.WebSocket = window.MozWebSocket;
10. }
11. if (window.WebSocket) {
12.   socket = new WebSocket("ws://localhost:2048/ws");
13.   socket.onmessage = function(event) {
14.     var ta = document.getElementById('responseText');
15.     ta.value = ta.value + '\n' + event.data
16.   };
17.   socket.onopen = function(event) {
18.     var ta = document.getElementById('responseText');
19.     ta.value = "Web Socket opened!";
20.   };
21.   socket.onclose = function(event) {
22.     var ta = document.getElementById('responseText');
23.     ta.value = ta.value + "Web Socket closed";
24.   };
25. } else {
26.   alert("Your browser does not support Web Socket.");
27. }
28.
29. function send(message) {
30.   if (!window.WebSocket) { return; }
31.   if (socket.readyState == WebSocket.OPEN) {
32.     socket.send(message);
33.   } else {
34.     alert("The socket is not open.");
35.   }
36. }
37. </script>
```

```
    <form onsubmit="return false;">
        <input type="text" name="message" value="Hello, World!"><input
            type="button" value="Send Web Socket Data"
            onclick="send(this.form.message.value)">
        <h3>Output</h3>
        <textarea id="responseText" style="width: 500px; height: 300px;"></textarea>
    </form>
</body>
</html>
```

最后在浏览器中输入：http://localhost:2048，多开几个窗口就可以聊天了。

11.5 给WebSocket加密

上面的应用程序虽然工作的很好，但是在网络上收发消息存在很大的安全隐患，所以有必要对消息进行加密。添加这样一个加密的功能一般比较复杂，需要对代码有较大的改动。但是使用Netty就可以很容易的添加这样的功能，只需要将SslHandler加入到ChannelPipeline中就可以了。实际上还需要添加SslContext，但这不在本例子范围内。

首先我们创建一个用于添加加密Handler的handler初始化类，看下面代码：

[java]

```
package netty.in.action;

import io.netty.channel.Channel;
import io.netty.channel.group.ChannelGroup;
import io.netty.handler.ssl.SslHandler;

import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLEngine;

public class SecureChatServerIntializer extends ChatServerInitializer {
    private final SSLContext context;

    public SecureChatServerIntializer(ChannelGroup group,SSLContext context) {
        super(group);
        this.context = context;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        super.initChannel(ch);
        SSLEngine engine = context.createSSLEngine();
        engine.setUseClientMode(false);
        ch.pipeline().addFirst(new SslHandler(engine));
    }
}
```

最后我们创建一个用于引导配置的类，看下面代码：

[java]

```
package netty.in.action;

import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.group.ChannelGroup;
import java.net.InetSocketAddress;
import javax.net.ssl.SSLContext;

/**
 * 访问地址：https://localhost:4096
 *
 * @author c.k
 *
 */
public class SecureChatServer extends ChatServer {
    private final SSLContext context;

    public SecureChatServer(SSLContext context) {
        this.context = context;
    }

    @Override
    protected ChannelInitializer<Channel> createInitializer(ChannelGroup group) {
        return new SecureChatServerIntializer(group, context);
    }

```

```
28.     /**
29.      * 获取SSLContext需要相关的keystore文件，这里没有 关于HTTPS可以查阅相关资料，这里只介绍在Netty中如何使用
30.      *
31.      * @return
32.      */
33.     private static SSLContext getSslContext() {
34.         return null;
35.     }
36.
37.     public static void main(String[] args) {
38.         SSLContext context = getSslContext();
39.         final SecureChatServer server = new SecureChatServer(context);
40.         ChannelFuture future = server.start(new InetSocketAddress(4096));
41.         Runtime.getRuntime().addShutdownHook(new Thread() {
42.             @Override
43.             public void run() {
44.                 server.destroy();
45.             }
46.         });
47.         future.channel().closeFuture().syncUninterruptibly();
48.     }
49. }
```

11.6 Summary