**基于Tomcat的Websocket范例及permessage-deflate扩展特性的研究**

[knightingal](https://www.jianshu.com/u/b9c20000587d)关注

2017.02.03 23:23:59字数 3,441阅读 5,285

**0x00 前言**

当前已经成为和空气水食物并列的生存必需品的互联网，其典型的应用大多采用基于HTTP协议的B/S这一基础架构。作为自1994网景发布第一款浏览器以来就存在的这一技术体系，尽管20多年来不断发展，已经非常成熟，却依然有一个尴尬之处随着应用场景的不断丰富而越发明显。那就是作为客户端的浏览器，无法实时的接收来自服务端的信息推送，以至于后来大家想到用js脚本周期性调用ajax轮询数据的方法曲线救国，直到HTML5的广泛应用。

HTML5加入了一个非常重要的特性叫websocket，它的作用是让浏览器开启一个和服务器之间的双向长连接，客户端既可以快速的向服务器发送信息，也可以实时接收来自服务器方向的推送，特别适用与即时通信，股票期货交易，网络游戏这类互性比较强，实时要求比较高的应用场景。

像其他HTML5的新特性一样，当前主流的浏览器均已支持websocket，比如Chrome及其魔改，Firefox，Safari，IE9及以上（包括Edge）。

下面用一个简单的范例来演示如何基于Tomcat实现一个最基本的websocket服务，以便后续的学习和研究。

**0x01 一个简单的范例**

该范例参考tomcat自带的websocket example，这里做进一步的简化。

创建一个普通的java工程，该工程依赖tomcat的lib目录下的两个websocket相关的jar包，tomcat-websoket.jar，websocket-api.jar。

然后创建以下两个类：

newWebsocket.SocketConfig

public class SocketConfig implements ServerApplicationConfig {

@Override

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

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

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

result.add(ServerEndpointConfig.Builder.create(EchoEndpoint.class,

"/websocket/echo").build());

}

return result;

}

@Override

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

Set<Class<?>> results = new HashSet<>();

for (Class<?> clazz : scanned) {

if (clazz.getPackage().getName().startsWith("newWebsocket.")) {

results.add(clazz);

}

}

return results;

}

}

newWebsocket.EchoEndpoint

public class EchoEndpoint extends Endpoint {

@Override

public void onOpen(Session session, EndpointConfig endpointConfig) {

RemoteEndpoint.Basic remoteEndpointBasic = session.getBasicRemote();

session.addMessageHandler(new EchoMessageHandlerText(remoteEndpointBasic));

}

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

private final RemoteEndpoint.Basic remoteEndpointBasic;

private EchoMessageHandlerText(RemoteEndpoint.Basic remoteEndpointBasic) {

this.remoteEndpointBasic = remoteEndpointBasic;

}

@Override

public void onMessage(String message, boolean last) {

try {

if (remoteEndpointBasic != null) {

remoteEndpointBasic.sendText(message, last);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

其中，SocketConfig负责将EchoEndpoint注册到容器中，并且和/websocket/echo这个路径绑定。

EchoEndpoint则是业务逻辑。在onOpen方法中注册了EchoMessageHandlerText这个Handler的实例，EchoMessageHandlerText的onMessage方法用于处理客户端发送过来的信息。

build这个工程，最终会生成这样一些.class文件。

newWebsocket

├── EchoEndpoint$1.class

├── EchoEndpoint.class

├── EchoEndpoint$EchoMessageHandlerText.class

└── SocketConfig.class

将newWebsocket目录复制到tomcat的webapps/examples/WEB-INF/classes，SocketConfig类在tomcat启动时会被扫描到，随后由容器调用SocketConfig实现的方法完成将EchoEndpoint注册至容器的工作。

启动tomcat后，用上述提到的支持websocket特性的浏览器打开[http://127.0.0.1:8080/](https://link.jianshu.com/?t=http://127.0.0.1:8080/)，如果tomcat启动成功，会看到tomcat的欢迎页面。不用管这个，打开开发者调试工具，依次输入以下javascript代码

var ws = new WebSocket("ws://127.0.0.1:8080/examples/websocket/echo");

ws.onmessage = function(event) {console.log(event.data)};

ws.send("111");

*请注意跨域限制，当前浏览器必须打开127.0.0.1:8080域下的任意一个页面，否则以上javascript代码可能无法成功执行*

这里以firefox为例，如果看到类似下图的调试页面打印出"111"，则表示websocket部署成功，且前端调用也正常。

**0x02 抓包和协议规范对照分析**

依靠纯粹的http协议是无法实现websocket的，所以其背后必然有一套不同于http的应用层协议作为支撑，该协议的标准文档是[RFC6455-The WebSocket Protocol](https://link.jianshu.com/?t=https://tools.ietf.org/html/rfc6455)。

接下来就根据实际抓包和标准文档进行比对来研究一下websocket在网络应用层的实现。由于我们的目的是研究websocket在服务端的底层实现，为了方便，我们直接使用tomcat自带的example来发起websocket请求。

首先是一条由客户端发往/examples/websocket/echoProgrammatic的http GET请求。规范中提到，websocket通信由http请求发起握手。注意该请求中的Connection和Upgrade头域的值，Upgrade头域的值为websocket，表示客户端希望将该次连接升级为一个websocket连接。

GET /examples/websocket/echoProgrammatic HTTP/1.1\r\n

Host: 192.168.0.101:8080\r\n

Connection: Upgrade\r\n

Pragma: no-cache\r\n

Cache-Control: no-cache\r\n

Upgrade: websocket\r\n

Origin: http://192.168.0.101:8080\r\n

Sec-WebSocket-Version: 13\r\n

User-Agent: Mozilla/5.0 (Linux; Android 6.0; H60-L01 Build/HDH60-L01) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.124 Mobile Safari/537.36\r\n

Accept-Encoding: gzip, deflate, sdch\r\n

Accept-Language: zh-CN,zh;q=0.8\r\n

Sec-WebSocket-Key: S4iljLdlI5qk3jpx2fHU4A==\r\n

Sec-WebSocket-Extensions: permessage-deflate; client\_max\_window\_bits\r\n

\r\n

[Full request URI: http://192.168.0.101:8080/examples/websocket/echoProgrammatic]

[HTTP request 1/1]

[Response in frame: 10]

HTTP/1.1 101 \r\n

Server: Apache-Coyote/1.1\r\n

Upgrade: websocket\r\n

Connection: upgrade\r\n

Sec-WebSocket-Accept: xwLDQrb5kzxpZDdeTcUd+7diXXU=\r\n

Sec-WebSocket-Extensions: permessage-deflate;client\_max\_window\_bits=15\r\n

Date: Sun, 09 Oct 2016 23:07:39 GMT\r\n

\r\n

[HTTP response 1/1]

[Time since request: 0.042990000 seconds]

[Request in frame: 9]

服务端随即返回了101响应，并且没有释放该次tcp连接，这表示握手成功，websocket连接已经建立完成。

接下来是数据帧(Data Framing)的抓包

WebSocket

1... .... = Fin: True

.100 .... = Reserved: 0x4

.... 0001 = Opcode: Text (1)

0... .... = Mask: False

.001 0100 = Payload length: 20

Payload

对照文档中对数据帧的格式定义如下：

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-------+-+-------------+-------------------------------+

|F|R|R|R| opcode|M| Payload len | Extended payload length |

|I|S|S|S| (4) |A| (7) | (16/64) |

|N|V|V|V| |S| | (if payload len==126/127) |

| |1|2|3| |K| | |

+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +

| Extended payload length continued, if payload len == 127 |

+ - - - - - - - - - - - - - - - +-------------------------------+

| |Masking-key, if MASK set to 1 |

+-------------------------------+-------------------------------+

| Masking-key (continued) | Payload Data |

+-------------------------------- - - - - - - - - - - - - - - - +

: Payload Data continued ... :

+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +

| Payload Data continued ... |

+---------------------------------------------------------------+

我们观察一下这个帧，wireshark的提示已经很清楚了，和文档中的定义基本都能对应上，唯一值得留意的是rsv1被置为了1，很奇怪，按照RFC6455，除非了其他约定，rsv1~3通常应该置为0，那么这里的其他约定指什么，RFC6455没提，那我们先放着，看一下payload

0000 f2 48 2d 4a 55 c8 2c 56 48 54 c8 4d 2d 2e 4e 4c .H-JU.,VHT.M-.NL

0010 4f 55 04 00 OU..

这里的问题就比较大了，按照文档说的，这里应该是我发送的信息的ascii字节码，比如我发的是"hello"，这里就应该是"0x48 0x65 0x6c 0x6c 0x6f"，但是眼下这个东西，连ascii都不是。

然而无论是作为服务端的tomcat，还是作为客户端的chrome，明显接受了这种奇怪的编码，并且确实无误的得到了我发送的信息。这里一定还是遵循其他的一些我尚未了解到的着某种公共的标准协议。

那么接下来我应该怎么做？像这种奇怪的码流拿去问google肯定也问不出什么名堂，还剩两条路：

* READ THE FUCKING CODE!!!。

于是毫不犹豫选择后者。

**0x03 对照源码寻找解答（虽然已经被标题剧透了，为了便于搜索的无奈选择）**

这里省略下载apache tomcat的源码以及编辑构建的过程，这是个老牌开源项目了。

为了找到以上奇怪码流的成因，跟踪返回消息是个比较容易的切入点，所以将断点定在EchoEndpoint.EchoMessageHandlerText.onMessage方法的入口是个比较好的选择。

于是用debug模式启动tomcat并启动远程调试，使用websocket的example页面发送一条websocket请求，采用单步跟踪，很快定位到WsRemoteEndpointImplBase.sendMessageBlock(byte opCode, ByteBuffer payload, boolean last, long timeoutExpiry)这个方法里的messageParts = transformation.sendMessagePart(messageParts);这行语句，对我们的消息体做了手脚。而transformation这个变量也确实起了一个一看就知道是干这种事情的名字。

transformation他的类型Transformation实际上是一个java接口，单步跟踪后发现实际上进入到PerMessageDeflate这个类的sendMessagePart(List<MessagePart> uncompressedParts)方法中。这个方法实际上做的事情是调用jdk里的Deflater的deflate方法对payload数据做压缩处理。到这里我们就明白了，之所以抓包看到的数据和我们实际发送的不一样，是因为做了deflate压缩。而因为是基于公共的算法，所以在客户端那边，也可以通过同样的算法还原出原信息。

那么下面要解决的问题就是，客户端和服务端之间是如何协商出使用deflate算法对数据进行压缩的。还是从源码中找答案，切入点是WsRemoteEndpointImplBase中transformation这个成员变量什么时候被实例化。经过一番顺藤摸瓜我们发现这个transformation的出生地位于org.apache.tomcat.websocket.server.UpgradeUtil的List<Transformation> createTransformations( List<Extension> negotiatedExtensions)这个方法。仔细观察逻辑发现这个方法构造transformation实例的过程和唯一的入参negotiatedExtensions中的一个叫做permessage-deflate的所谓的name有密切关系。

那么这个negotiatedExtensions是什么东西，他来自哪里？继续往上翻代码实在是有点晕了，猜一下吧，UpgradeUtil这个类名字以及唯一调用这个方法的public static void doUpgrade(WsServerContainer sc, HttpServletRequest req, HttpServletResponse resp, ServerEndpointConfig sec, Map<String,String> pathParams)这个方法名，我猜测这里应该是由第一个http请求发起websocket通信的地方，打个断点验证了一下确实如此，第一个http请求过来的时候即命中了这个方法。“negotiatedExtensions”从名字看是协商扩展的意思，按照以往研究其他协议的经验看，所谓协商通常是在握手的请求和响应过程中完成的，那么permessage-deflate应该是握手请求中的某个参数，扫了一眼抓包信息果然如此，Sec-WebSocket-Extensions: permessage-deflate; client\_max\_window\_bits这个头域里面带的不就是吗？所以我们目前可以这么猜测，客户端和服务端之间就是根据首次http请求中的Sec-WebSocket-Extensions这个头域中的permessage-deflate这个参数来协商是否对传输数据进行deflate压缩的。

下面的问题就是怎么验证我的这猜测了。客户端这边比较头疼，javascript里的WebSocket类并没有更多可供配置的参数，是否支持deflate扩展似乎完全是各家浏览器内部实现自己说了算。尝试了chrome和firefox发现都是默认开启该permessage-deflate，并且没有找到办法关闭，Safari和Edge一个手头没有，另一个被我玩坏了处于罢工状态，最后发现能用上手的只有IE不支持permessage-deflate扩展。被吐槽嫌弃了一万年想不到也有发挥余热的一天————以不支持某一特性这种方式。

MFW

于是用IE发起websocket请求以后我们抓包看的是这样的结果：

GET /examples/websocket/echoProgrammatic HTTP/1.1\r\n

Origin: http://192.168.0.103:18080\r\n

Sec-WebSocket-Key: qd6f1YwxnAfGrkqFIy5kFw==\r\n

Connection: Upgrade\r\n

Upgrade: websocket\r\n

Sec-WebSocket-Version: 13\r\n

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n

Host: 192.168.0.103:18080\r\n

Cache-Control: no-cache\r\n

\r\n

[Full request URI: http://192.168.0.103:18080/examples/websocket/echoProgrammatic]

[HTTP request 1/1]

[Response in frame: 7]

HTTP/1.1 101 \r\n

Upgrade: websocket\r\n

Connection: upgrade\r\n

Sec-WebSocket-Accept: 13FxZb9VlaaWo+9kYEgPZDKfwGg=\r\n

Date: Sat, 14 Jan 2017 15:52:17 GMT\r\n

\r\n

[HTTP response 1/1]

[Time since request: 0.029400000 seconds]

[Request in frame: 5]

可以看到在第一个http请求中没有Sec-WebSocket-Extensions头域，返回的101响应也没有，说明没有对permessage-deflate特性进行协商。  
接下来是数据帧抓包：

WebSocket

1... .... = Fin: True

.000 .... = Reserved: 0x0

.... 0001 = Opcode: Text (1)

0... .... = Mask: False

.001 0010 = Payload length: 18

Payload

PayLoad:

0000 48 65 72 65 20 69 73 20 61 20 6d 65 73 73 61 67 Here is a messag

0010 65 21 e!

果然以ascii码流的形式传输了，并且注意到rsv1标志位也被置0了。

单步跟踪代码的执行过程可以发现，在这样的情况下List<Transformation> createTransformations( List<Extension> negotiatedExtensions)入参是一个空的列表，并且返回的也是一个空的列表，这会导致在后续的一系列的初始化过程当中，transformation被初始化为UnmaskTransformation这类的实例。我们来看看这个类的List<MessagePart> sendMessagePart(List<MessagePart> messageParts)方法的实现：

@Override

public List<MessagePart> sendMessagePart(List<MessagePart> messageParts) {

// NO-OP send so simply return the message unchanged.

return messageParts;

}

呵呵。。。

接下来验证服务端，假设服务端不支持permessage-deflate，即使客户端的http请求里面带了Sec-WebSocket-Extensions头域，扩展协商也会失败。既然有源码，很容易就可以将tomcat改造为我们需要的样子，比如在List<Transformation> createTransformations(List<Extension> negotiatedExtensions)这个方法的开头加入这样一段代码：

for(Extension extension: negotiatedExtensions) {

if (PerMessageDeflate.NAME.equals(extension.getName())) {

negotiatedExtensions.remove(extension);

break;

}

}

即将请求中的permessage-deflate扩展参数移除掉。重新编译tomcat后重启服务，用chrome发起websocket通信，抓包如下：

GET /examples/websocket/echoProgrammatic HTTP/1.1\r\n

Host: 192.168.163.128:18080\r\n

Connection: Upgrade\r\n

Pragma: no-cache\r\n

Cache-Control: no-cache\r\n

Upgrade: websocket\r\n

Origin: http://192.168.163.128:18080\r\n

Sec-WebSocket-Version: 13\r\n

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87 Safari/537.36\r\n

Accept-Encoding: gzip, deflate, sdch\r\n

Accept-Language: zh-CN,zh;q=0.8,en;q=0.6\r\n

Sec-WebSocket-Key: N+GWswsViw18TfSpryLcVw==\r\n

Sec-WebSocket-Extensions: permessage-deflate; client\_max\_window\_bits\r\n

\r\n

[Full request URI: http://192.168.163.128:18080/examples/websocket/echoProgrammatic]

[HTTP request 1/1]

[Response in frame: 52]

HTTP/1.1 101 \r\n

Upgrade: websocket\r\n

Connection: upgrade\r\n

Sec-WebSocket-Accept: 4tRMuDpE6WErH7Gc0XqTBmfN/7U=\r\n

Date: Mon, 16 Jan 2017 16:10:14 GMT\r\n

\r\n

[HTTP response 1/1]

[Time since request: 0.380323000 seconds]

[Request in frame: 49]

我们看到服务器无视了请求中的Sec-WebSocket-Extensions头域，假装不支持permessage-deflate特性一样返回了和ie浏览器类似的101响应，这意味着permessage-deflate特性在协议层面协商失败。于是即使客户端是chrome，大家也还是用ascii码流的形式传输数据

1... .... = Fin: True

.000 .... = Reserved: 0x0

.... 0001 = Opcode: Text (1)

0... .... = Mask: False

.001 0010 = Payload length: 18

Payload

0000 48 65 72 65 20 69 73 20 61 20 6d 65 73 73 61 67 Here is a messag

0010 65 21 e!

在源码实现层面上，我们了解了上文中最初的WebSocket的payload没有采用ascii编码的原因：是因为http握手过程中客户端和服务端对permessage-deflate扩展特性协商采用了deflate对payload做了压缩编码导致的。

**0x04 相关标准**

剖析完了代码，我们最后还需要再找到相应的标准才能完成闭环。RFC6455当中并没有提及这个permessage-deflate，搜了一下发现相关标准位于[RFC7692](https://link.jianshu.com/?t=https://tools.ietf.org/html/rfc7692)，一份对websocket的扩展协议。在该协议的第7节专门对permessage-deflate扩展做了规定，包括握手请求和响应中应用Sec-WebSocket-Extensions头域对permessage-deflate相关参数的协商，以及规定，一旦采用permessage-deflate扩展，则rsv1标志位必须置为1。

The "Per-Message Compressed" bit, which indicates whether or not

the message is compressed. RSV1 is set for compressed messages

and unset for uncompressed messages.

至此，tomcat源码实现，实际抓包结果，和标准规范已经完全能够对应上了。

**0x05 后记**

目前为止，我们了解到了websocket相关标准中的一些扩展特性，以及tomcat对这些特性的实现方面的一些细节，学习到了一些很有趣的课外知识。那么这些知识有什么实际用处呢？当然有，比如某一天如果我们的实际业务涉及到websocket的应用，在调试的过程中我们如何观察websocket接口的数据流？由于permessage-deflate扩展的影响，从抓包上几乎无法观察数据流，Chrome的调试工具能够展现解码后的ascii编码的字符串

但是遇上二进制的码流也是无能为力，用IE系列倒是可以规避permessage-deflate的影响，但用IE不觉得跌份吗？

现在有了新的选择，我们可以定制自己的tomcat，在服务端屏蔽掉permessage-deflate扩展，任意一个浏览器都可以进行调试，用抓包工具就可以观察码流。可以，这很GEEK！