转载请注明出处：http://blog.csdn.net/llew2011/article/details/53056165

做Java开发的小伙伴们应该对Socket比较熟悉，在J2SE的Socket编程这一章节中专门对Socket通信做了详细介绍，当时自学完该章节后只知道Socket是端到端通信的，Server端根据指定端口打开Socket链接，然后等待客户端来连接；客户端根据Server端IP地址和端口创建一个Socket通道，根据该通道和Server端进行通信。后来在工作中有使用Socket通信，使用场景是聊天和推送，当时为了项目进度就在GitHub上找了一个不错的开源库autobahn-java中应用在项目中，功能实现之后并没有继续深入理解WebSocket协议。恰好现在项目又使用到了WebSocket通信，因此决定仔细研究一下Socket通信的底层相关知识并记录下来，希望能给小伙伴们一点帮助，如果你对Socket变成非常熟悉了，请跳过本文(*^__^*) ……

## 背景

以前，很多网站为了实现推动技术，所用的技术都是轮训(例如：AJax)。轮训是在特定的时间间隔(例如每隔5秒)，由浏览器对服务器发送HTTP请求，然后服务器收到浏览器请求后把相关最新数据返回客户端浏览器(注意：也可能没有最新数据)。这种传统的轮训模式带来很明显的缺点，即浏览器需要不断的向服务器发出HTTP请求，然而HTTP请求可能包含较长的头部，其真正有效的数据可能只是很小的一部分，显然这样会浪费很多的宽带等资源；假如某次请求时服务器端并没有最新数据需要返回给客户端，那么这次请求等同于无效的，那么怎么才能实现客户端能及时的收到服务器端最新数据又能占用很少的网络宽带资源呢?

在这种情况下，HTML5定义了WebSocket协议，该协议是一种在单个TCP连接上进行全双工通讯的协议，IETF(Internet Engineering Task Force，互联网工程小组)在2011年把这套协议定为RFC 6455标

准，并被RFC7936所补充规范。因此WebSocket API 也被W3C订为标准。

WebSocket使得客户端和服务器之间的数据交换变得更加简单，允许服务器端主动向客户端推送数据。在WebSocket API中，浏览器和服务器只需要要完成一个握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

根据WebSocket的概念，我们可以总结以下两点：

- TCP基础上
- 全双工通信

那么什么是全双工通信了？

通信方式可以分为两种，一种是半双工通信，一种是全双工通信。半双工通信指的是在某一个时间点上只有发送数据或者接收数据发生这一个行为发生。而全双工通信是相对于半双工通信来讲的，全双工通信允许发送数据和接收数据同时发生。

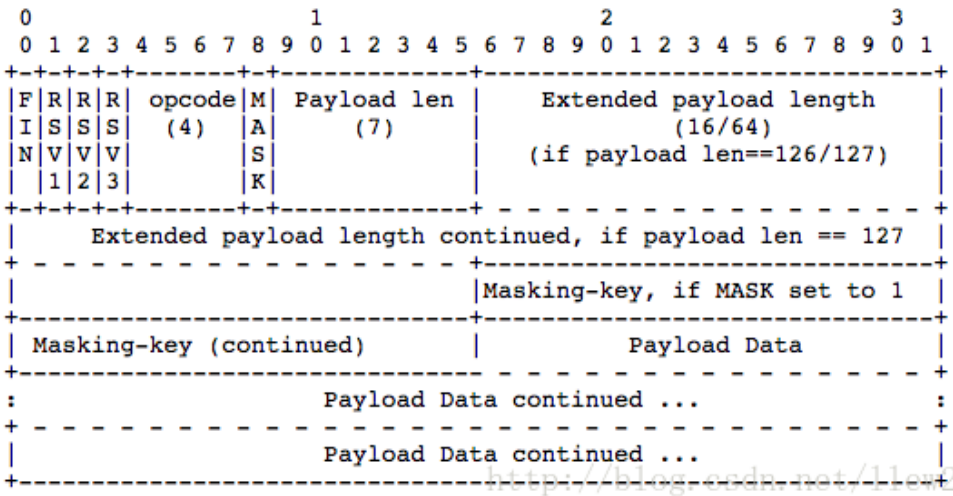
WebSocket使用ws或者是wss的统一资源标识符，类似于HTTP和HTTPS，其中wss表示在TLS之上的WebSocket，如下所示：

ws://*127.0.0.1:80/wsapi*wss://*127.0.0.1:443/wsapi*

WebSocket如果没有运行在TLS之上时默认使用和HTTP相同的80端口，当运行在TLS之上时，默认使用443端口。

WebSocket的协议内容详见The WebSocket Protocol，这是最全面的官方说明，另外维基百科也有对WebSocket做了简单说明。WebSocket的适用场景可以是即时聊天，推送或者是直播中的弹幕等。

为了清楚WebSocket协议内容，先上一张图：




上图是一个数据帧，WebSocket就是按照该协议规则进行传输数据的。

- FIN

占 **1** 位，该标记表示消息时候是最后一帧，一个消息由1个或者多个数据帧组成，若消息只有一帧，那么起始帧就是结束帧。

- RSV1，RSV2，RSV3

各占**1**位，预留位，用于自定义扩展。如果没有扩展，各位值都是0；如果定义了扩展，则值为非0。如果接受到的帧中此处值为非0，但是扩展中却没有该值的定义，那么关闭连接。

- OPCODE

占 **4** 位，表示数据帧的类型，帧类型分为控制帧和非控制帧。如果接收到未知帧，接收端必须关闭连接。已定义的帧类型如下图所示：

| Opcode | Meaning | Reference |
|---|---|---|
| 0 | Continuation Frame | RFC 6455 |
| 1 | Text Frame | RFC 6455 |
| 2 | Binary Frame | RFC 6455 |
| 8 | Connection Close Frame | RFC 6455 |
| 9 | Ping Frame | RFC 6455 |
| 10 | Pong Frame | RFC 6455 |

上图中0-7表示非空指针，8-15表示控制帧；而3-7暂时没有进行定义，为以后的非控制帧做保留。同样11-15也暂时没有定义，为以后的控制帧保留。

消息的分片，一般来说，对于一个长度较小的消息，可以使用1帧完成消息的发送，比如说文本消息，则FIN值为1，表示消息结束，此时OPCODE只不能为0，0表示后续还有数据帧会发送过来。而对于一些较长的消息，则需要将消息进行分片发送。比如语音消息，这时候起始帧的FIN值为0，OPCODE值非0，接着是若干帧(FIN值为0，OPCODE值为0)，最后结束帧FIN值为1，OPCODE值为0.

WebSocket的控制帧有三种：关闭帧，Ping帧，Pong帧。关闭帧很好理解，客户端如果收到关闭帧则直接关闭连接即可，当然客户端也可以发送关闭帧给服务器端。二Ping帧和Pong帧则是WebSocket的心跳检测帧，用来确保客户端是在线的，一般来说，只有服务器端给客户端发送Ping帧，然后客户端发送Pong帧进行回应，表示自己还在线，可以进行后续通信。

- MASK

占 **1** 位，掩码位，表示帧中的数据是否经过加密，客户端发出的数据帧需要进行掩码处理，这个值都是1。如果值是1，那么Masking-key域的数据帧就是掩码密钥，用来解码PlayloadData数据，否则Masking-key长度是0。

【注意：】WebSocket协议规定数据通过帧序列传输，客户端必须对其发送到服务端的所有帧进行掩码处理。服务器端一旦收到无掩码帧则关闭连接并向客户端发送一个状态码为1002(表示协议错误)的Close帧。服务器端发送给客户端的数据帧不错掩码处理，一旦客户端发现接受到的数据帧经过了掩码处理，将关闭连接并发送给服务端状态码为1002的Close帧。更多状态码如下图所示：

| Status Code | Meaning | Contact | Reference |
|---|---|---|---|
| 1000 | Normal Closure | hybi@ietf.org | RFC 6455 |
| 1001 | Going Away | hybi@ietf.org | RFC 6455 |
| 1002 | Protocol error | hybi@ietf.org | RFC 6455 |
| 1003 | Unsupported Data | hybi@ietf.org | RFC 6455 |
| 1004 | ---Reserved---- | hybi@ietf.org | RFC 6455 |
| 1005 | No Status Rcvd | hybi@ietf.org | RFC 6455 |
| 1006 | Abnormal Closure | hybi@ietf.org | RFC 6455 |
| 1007 | Invalid frame payload data | hybi@ietf.org | RFC 6455 |
| 1008 | Policy Violation | hybi@ietf.org | RFC 6455 |
| 1009 | Message Too Big | hybi@ietf.org | RFC 6455 |
| 1010 | Mandatory Ext. | hybi@ietf.org | RFC 6455 |
| 1011 | Internal Server Error | hybi@ietf.org | RFC 6455 |
| 1015 | TLS handshake | hybi@ietf.org | RFC 6455 |



- Payload len

占 7 位，或者 7 + 16 位，或者 7 + 64 位，表示数据帧大小，这里分以下几种情况：

a）如果值在0 - 125 之间，那么该值就表示Playload Data的真实长度。

b）如果值在为126，那么该7位后面紧跟着的2个字节就是Playload Data的真实长度。

c）如果值为127，那么该7位后面紧跟着的8个字节就是Playload Data的真实长度。

【注意：】0-125之间，必须用7位表示；不允许将这7位表示成126或者127，然后后面用2个字节或者8个字节表示124，这样就违反了原则。

- Masking-key

占0个字节，或者4个字节；当MASK位设置为0时则该字段缺失；若MASK位为1时，则该字段占4个字节；那么发出去的数据必须经过掩码处理，掩码流程如下：

```
a. voidmask(byte[] origin, byte[] maskKey){
b. if (null == origin || null == maskKey) return;
c. int length = origin.length;
d. for (int i = 0; i < length; i++) {
```

```
e.          origin[i] = (byte) (origin[i] ^ maskKey[i % 4]); // 按位异或运
算，相同为0，不同为1
f.      }
g. }
```

- Payload Data

占(x+y)个字节，x表示Extension Data，即扩展数据；y表示Application Data，即程序数据。

【注意：】扩展数据可能为0，双方必须提前进行协商，规定其长度，否则就是不合法的数据帧。

-

以上就是WebSocket协议规定的数据传输的帧内容，大致了解一下即可。除此之外，WebSocket协议还有一个握手的过程，通过握手发送一个HTTP请求来完成，这里基本和HTTP2.0有点类似，客户端发送一个请求协议升级的get请求给服务端，服务端支持的话会返回HTTP Code 为101的状态码，表示可以切换到对应的协议，大致流程如下：

客户端发送Get请求：

```
GET /chat HTTP/1.1Host: example.comUpgrade: websocketConnection:
UpgradeSec-WebSocket-key: Sec-WebSocket-Version: 13
```

客户端给服务端发送握手协议包，包的报文格式必须符合HTTP报文规范，其中：

- 请求方法必须为GET方法
- HTTP版本不能低于1.1
- 必须包含Upgrade头部，值必须为websocket，表示请求升级协议，把协议升级为websocket协议
- 必须包含Connection头部，值必须为Upgrade，表示请求连接为协议升级连接
- 必须包含Sec-WebSocket-Key头部，值是一个Base64编码的16字节随机字符串
- 必须包含Sec-WebSocket-Version头部，值必须为13，表示websocket协

议的版本号
服务端响应请求：

HTTP/1.1101"Switching Protocols"Upgrade: websocketConnection: UpgradeSec-WebSocket-Accept:

服务器端收到客户端的请求后，返回给客户端的报文必须满足一下规范：

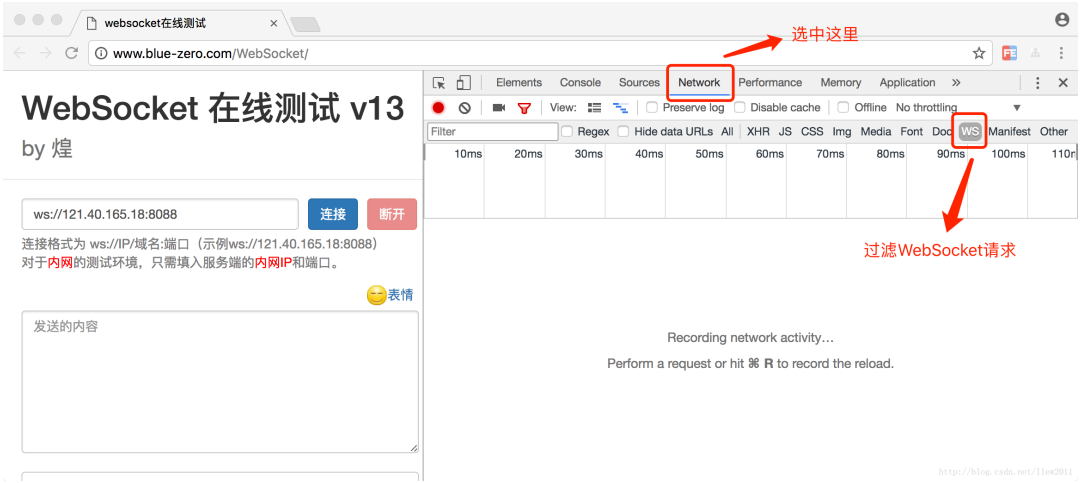
- 若服务器支持指定版本的websocket协议，则返回101的状态码并有描述，一般为"Switching Protocols"
- 头部必须包含Upgrade字段，其值必须为websocket
- 头部必须包含Connection字段，其值必须为Upgrade
- 头部必须包含Sec-WebSocket-Accept字段，其值算法如下：

1、把接收到的Sec-WebSocket-Key值后拼接固定字符串"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
2、把拼接后的值进行一次SHA-1计算
3、把计算后的值进行一次Base64编码

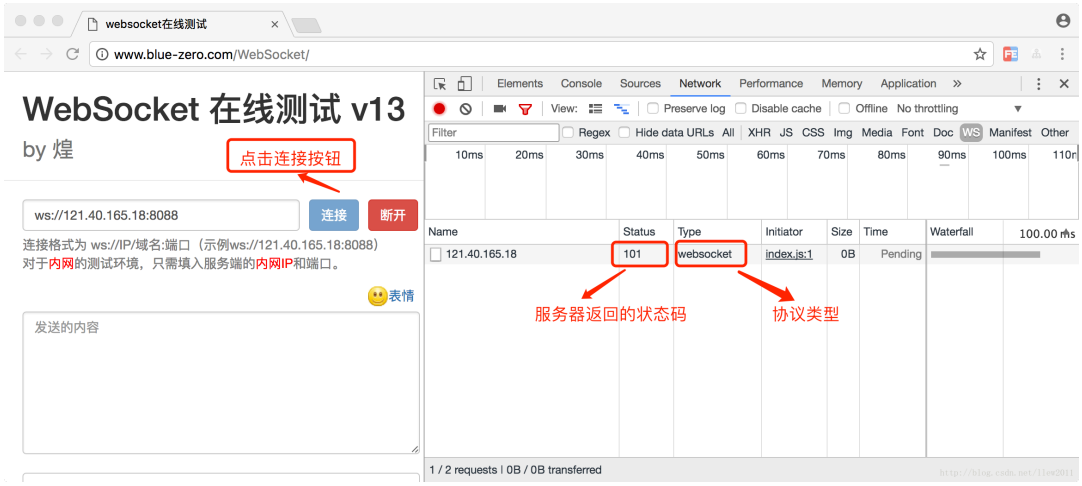
在客户端收到服务器端发送的报文后，首先验证报文格式是否符合规范，如果符合规范则继续验证Sec-WebSocket-Accept的值，当Sec-WebSocket-Accept的值验证通过则WebSocket连接建议，否则只要其中任何一步验证不通过就不能建立WebSocket连接。

为了更清楚的了解WebSocket请求流程，网上有许多在线的WebSocket测试网站，百度一下"WebSocket在线测试"就会有许多测试网站，我们选择第一个测试即可，使用Chrome浏览器(注意：没有安装Chrome浏览器使用其他浏览器也可，只要浏览器上带有开发者工具选项即可)，打开该页面，如下所示：

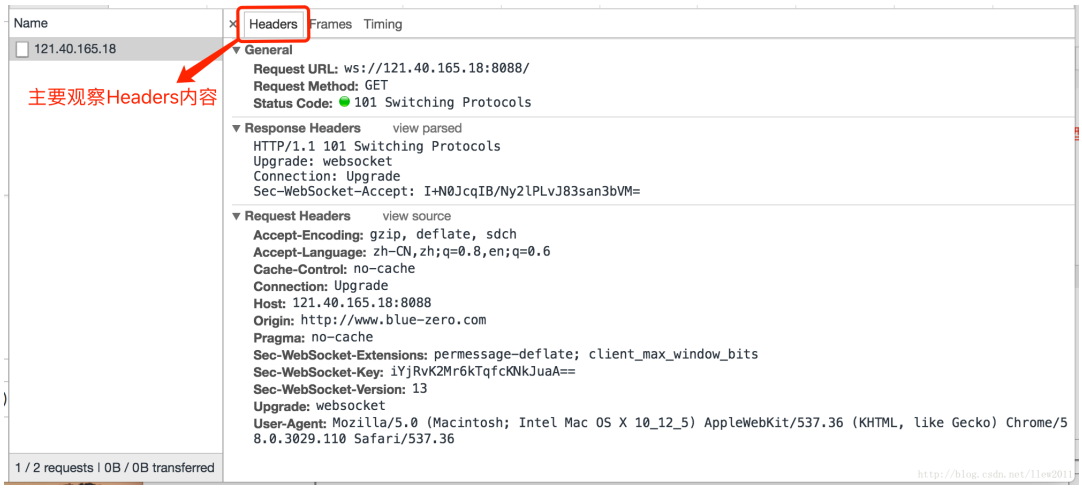

该测试网页有连接和断开的按钮，不用说你也明白这是测试左侧给出的测试地址，当然你也可以输入其他测试地址。这时候打开Chrome的开发者工具，选择Network选项，然后选中下边的WS选项，表示监控所有的WebSocket连接，如下图所示：




打开Chrome的开发者选项之后，这时候你点击左侧的连接按钮，就会有对该此请求的拦截，如下图所示：




然后点击拦截的IP地址，如下图所示：




上图展示了拦截当前请求的Headers的内容，我们先看一下General部分，General部分表示整体的请求和响应流程，详细解释如下所示：

- Request URL：ws://121.40.165.18:8088

表示该次请求地址是：ws://121.40.165.18:8088/

- Request Method：GET

表示该次请求为GET请求，【注意：只能是GET请求】

- Status Code：101 Switching Protocols

表示该次请求服务器返回的状态，101表示协议升级，后边的Switching Protocols是一个文本描述，可以是其他值

了解了Headers的General部分后，我们再来看一下Request

Headers部分，部分说明如下所示：

- Connection：Upgrade

表示当前连接要升级协议

- Upgrade：websocket

表示升级为Websocket协议

- Sec-WebSocket-Key：iYjRvK2Mr6kTqfcKNkJuaA==

客户端随机生成的字符串，为了验证服务器端返回数据的合法性

- Sec-WebSocket-Version：13

表示WebSocket协议的版本号，目前统一订为13

- Sec-WebSocket-Extensions：permessage-deflate; client_max_window_bits

表示一些扩展信息，可以为多值，中间用分号隔开

最后是Response Headers内容，如下所示：

- HTTP/1.1 101 Switch Protocols
- Upgrade：websocket
- Connection：Upgrade
- Sec-WebSocket-Accept：I+N0JcqIB/Ny2lPLvJ83san3bVM=

好了，到这里有关WebSocket协议的核心知识点差不多讲完了，并通过Chrome的开发者工具向小伙伴们清晰明了的展示了WebSocket的请求流程，如果你想更为细致的了解WebSocket协议，请点击这里。了解了WebSocket协议后如果想要通过WebSocket协议进行即时通信，那就意味着你需要写一套该协议的解析实现类，如果你有时间和精力那是最好的，没有也无妨，这个世界上总有热心开源的人，他们已经开源各个平台的源码库，在Android平台著名的WebSocket库有以下几个：

1. okhttp，
2. autobahn-java，
3. Java-WebSocket，

好了，由于篇幅原因，在下篇文章中我计划使用okhttp来实现我们自己的即时聊天小APP，然后带领小伙伴们从源码的角度深入理解一下okhttp里边有关WebSocket的实现原理，敬请期待(*^__^*) ……

【参考文章】

https://tools.ietf.org/html/rfc6455
https://zh.wikipedia.org/wiki/WebSocket

http://blog.csdn.net/sbsujjbcy/article/details/52839540