**[WebSocket数据包协议详解](https://www.cnblogs.com/smark/archive/2012/11/26/2789812.html)**

其实我一直想不明白HTML5包装个应用层办议作为Socket通过基础目的是为了什么,其实直接支持Socket tcp相对来说更加简单灵活.既然标准已经制定而浏览器也支持那对于我们开发者来说只能用的分.最新版本的WebSocket协议于2011-12其标准规范已经明确下来,所以现在可以根据这标准进行相应的开发.详细参考http://datatracker.ietf.org/doc/rfc6455/?include\_text=1

WebSocket协议主要分为两部分,第一部分是连接许可验证和验证后的数据交互.连接许可验证比较简单,由Client发送一个类似于HTTP的请求,服务端获取请求后根据请求的KEY生成对应的值并返回.

**连接请求内容:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | GET / HTTP/1.1  Connection:Upgrade  Host:127.0.0.1:8088  Origin:null  Sec-WebSocket-Extensions:x-webkit-deflate-frame  Sec-WebSocket-Key:puVOuWb7rel6z2AVZBKnfw==  Sec-WebSocket-Version:13  Upgrade:websocket |

服务端接收请求后主要是成针对Sec-WebSocket-Key生成对就Sec-WebSocket-Accept 的key,生成Sec-WebSocket-Accept 值比较简单就是Sha1(Sec-WebSocket-Key+258EAFA5-E914-47DA-95CA-C5AB0DC85B11)即可,C#代码如下:

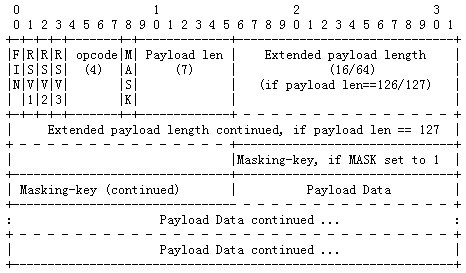
|  |  |
| --- | --- |
| 1  2  3  4  5 | SHA1 sha1 = new SHA1CryptoServiceProvider();  byte[] bytes\_sha1\_in = Encoding.UTF8.GetBytes(request.SecWebSocketKey+ "258EAFA5-E914-47DA-95CA-C5AB0DC85B11");  byte[] bytes\_sha1\_out = sha1.ComputeHash(bytes\_sha1\_in);  string str\_sha1\_out = Convert.ToBase64String(bytes\_sha1\_out);  response.SecWebSocketAccept = str\_sha1\_out; |

**服务端返回内容:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | HTTP/1.1 101 Switching Protocols  Connection:Upgrade  Server:beetle websocket server  Upgrade:WebSocket  Date:Mon, 26 Nov 2012 23:42:44 GMT  Access-Control-Allow-Credentials:true  Access-Control-Allow-Headers:content-type  Sec-WebSocket-Accept:FCKgUr8c7OsDsLFeJTWrJw6WO8Q= |

经过服务器的返回处理后连接握手成功,后面就可以进行TCP通讯.WebSocket在握手后发送数据并象下层TCP协议那样由用户自定义,还是需要遵循对应的应用协议规范...这也是在文章之说没有直接基于Socket tcp方便的原因.

**数据交互协议:**



这图有点难看懂...里面包括几种情况有掩码,数据长度小于126,小于UINT16和小于UINT64等几种情况.后面会慢慢详细说明.整个协议头大概分三部分组成,第一部分是描述消息结束情况和类型,第二部分是描述是否存在掩码长度,第三部分是扩展长度描述和掩码值.

从图中可以看到WebSocket协议数据主要通过头两个字节来描述数据包的情况

**第一个字节**

最高位用于描述消息是否结束,如果为1则该消息为消息尾部,如果为零则还有后续数据包;后面3位是用于扩展定义的,如果没有扩展约定的情况则必须为0.可以通过以下c#代码方式得到相应值

|  |  |
| --- | --- |
| 1 | mDataPackage.IsEof = (data[start] >> 7) > 0; |

最低4位用于描述消息类型,消息类型暂定有15种,其中有几种是预留设置.c#代码可以这样得到消息类型:

|  |  |
| --- | --- |
| 1  2 | int type = data[start] & 0xF;  mDataPackage.Type = (PackageType)type; |

**第二个字节**

消息的第二个字节主要用一描述掩码和消息长度,最高位用0或1来描述是否有掩码处理,可以通过以下c#代码方式得到相应值

|  |  |
| --- | --- |
| 1 | bool hasMask = (data[start] >>7) > 0; |

剩下的后面7位用来描述消息长度,由于7位最多只能描述127所以这个值会代表三种情况,一种是消息内容少于126存储消息长度,如果消息长度少于UINT16的情况此值为126,当消息长度大于UINT16的情况下此值为127;这两种情况的消息长度存储到紧随后面的byte[],分别是UINT16(2位byte)和UINT64(4位byte).可以通过以下c#代码方式得到相应值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | mPackageLength = (uint)(data[start] & 0x7F);  start++;  if (mPackageLength == 126)  {      mPackageLength = BitConverter.ToUInt16(data, start);      start = start + 2;  }  else if (mPackageLength == 127)  {      mPackageLength = BitConverter.ToUInt64(data, start);      start = start + 8;  } |

如果存在掩码的情况下获取4位掩码值:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | if (hasMask)  {      mDataPackage.Masking\_key = new byte[4];      Buffer.BlockCopy(data, start, mDataPackage.Masking\_key, 0, 4);        start = start + 4;      count = count - 4;  } |

**获取消息体**

当得到消息体长度后就可以获取对应长度的byte[],有些消息类型是没有长度的如%x8 denotes a connection close.对于Text类型的消息对应的byte[]是相应字符的UTF8编码.获取消息体还有一个需要注意的地方就是掩码,如果存在掩码的情况下接收的byte[]要做如下转换处理:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if (mDataPackage.Masking\_key != null)      {          int length = mDataPackage.Data.Count;          for (var i = 0; i < length; i++)              mDataPackage.Data.Array[i] = (byte)(mDataPackage.Data.Array[i] ^ mDataPackage.Masking\_key[i % 4]);      } |