# **[websocket之一：websocket简介](https://www.cnblogs.com/duanxz/p/4489313.html)**

**Websocket**

websocket为一次HTTP握手后，后续通讯为tcp协议的通讯方式。

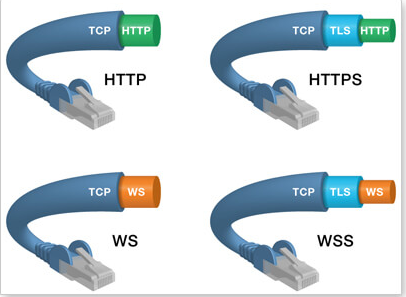
WebSocket 使用一种被称作**“Upgrade handshake（升级握手）”的机制将标准的 HTTP 或HTTPS 协议转为 WebSocket**。因此，使用 WebSocket 的应用程序将始终以 HTTP/S 开始，然后进行升级。这种升级发生在什么时候取决于具体的应用;可以在应用启动的时候，或者当一个特定的 URL 被请求的时候。

在我们的应用中，仅当 URL 请求以“/ws”结束时，我们才升级协议为WebSocket。否则，服务器将使用基本的 HTTP/S。一旦连接升级，之后的数据传输都将使用 WebSocket 。

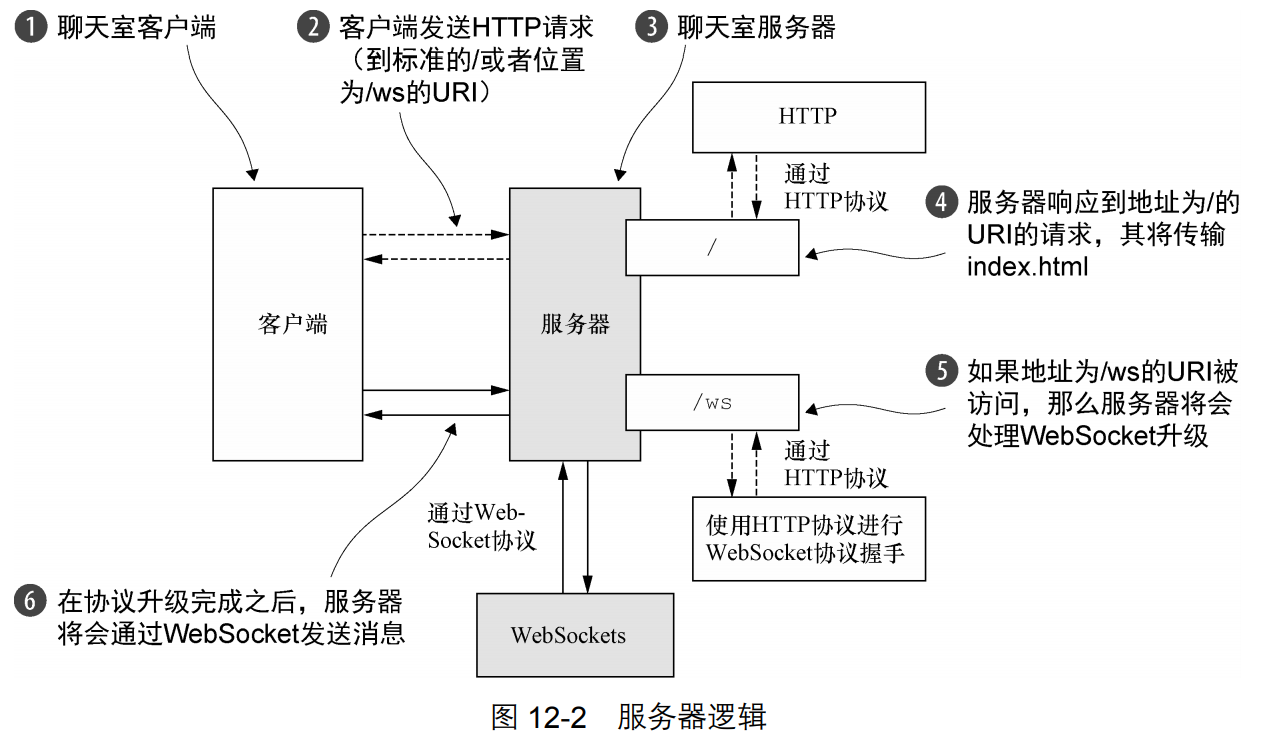
当然，和HTTP一样，websocket也有一些约定的通讯方式，http通讯方式为http开头的方式,e.g. [http://xxx.com/path](http://xxx.com/path" \t "https://www.cnblogs.com/duanxz/p/_blank) ,websocket通讯方式则为ws开头的方式,e.g. ws://xxx.com/path

SSL:

1. HTTP: [https://xxx.com/path](https://xxx.com/path" \t "https://www.cnblogs.com/duanxz/p/_blank)
2. WEBSOCKET: wss://xxx.com/path



下面的是Netty中Websocket的逻辑图：



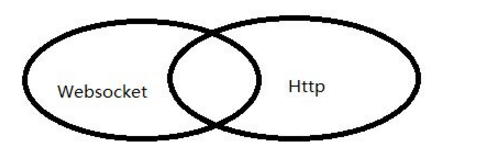
## 

## **一、websocket的特点**

### **1.2、websocket与http**

WebSocket是[HTML5](http://lib.csdn.net/base/html5" \o "HTML5知识库" \t "https://www.cnblogs.com/duanxz/p/_blank)出的东西（协议），也就是说HTTP协议没有变化，或者说没关系，但HTTP是不支持持久连接的（长连接，循环连接的不算）

首先HTTP有 1.1 和 1.0 之说，也就是所谓的 keep-alive ，把多个HTTP请求合并为一个，但是 Websocket 其实是一个新协议，跟HTTP协议基本没有关系，只是为了兼容现有浏览器的握手规范而已，也就是说它是HTTP协议上的一种补充可以通过这样一张图理解



有交集，但是并不是全部。

**共同点是**：都是基于TCP协议进行client-server的链接，websocket是HTML5提出的一套补缺HTTP链接中不能持久链接的特点（除长连接，长轮询）。

回顾下下面几个概念：

* 轮询（AJAX）：指的是浏览器端定时发送请求到服务器端，服务器接收到请求后作出响应并关闭连接该次链接。（缺点是：在时间定时上需要考虑更多，多久进行一次轮询等）
* 长轮询：跟轮询的原理相似，但是有个重要的特点是，浏览器发送请求后，服务器此时是保持该链接（主要方法是死循环执行，定时sleep超时后断开连接），在定时断开链接之前，获取到最新数据并返回浏览器后，在由浏览器重新发起一个新的请求。    （缺点是：会并发性的发起很多的http请求。）

    轮询与长轮询共同的缺点是：需要浪费很多的http资源，请求的数量较大。

* 长连接：类似前端中跨域请求中，利用iframe的src的跨域特点，在页面中隐藏一个iframe进行定时获取后台数据，反馈到iframe中，实现后台数据的获取。（ 长链接的缺点也是：必须要定时发起请求服务器更新数据资源。）

**websocket解决的问题：**

实质的推送方式是服务器主动推送，只要有数据就推送到请求方。（变被动为主动）

websocket采用异步回调的方式接受消息，当建立通信连接，可以做到持久性的连接，并进行通信。而不像上面的几种方式一样需要定时进行发起请求到服务器获取最新更新信息，显得相当的被动）

websocket通过自己的 WS 协议（此处与HTTP协议有所区别）创建一个基于HTTP request请求并创建TCP链接之后，之后的数据交换都不需要再次去创建链接，实现真正的长连接。

websocket协议本质上是一个基于TCP的协议。建立连接需要握手，客户端（浏览器）首先向服务器（web server）发起一条特殊的http请求，web server解析后生成应答到浏览器，这样子一个websocket连接就建立了，直到某一方关闭连接。

当然基于Node.js编写的一个Socket.IO 是一个开源实现WebSocket的库，它通过node.js实现服务端的同时，也提供了客户端js库，socket.io 支持事件触发为基础进行的双向数据通信。

### **与HTTP比较**

同样作为应用层的协议，WebSocket在现代的软件开发中被越来越多的实践，和HTTP有很多相似的地方，这里将它们简单的做一个纯个人、非权威的比较：

#### **相同点**

1. 都是基于TCP的应用层协议。
2. 都使用Request/Response模型进行连接的建立。
3. 在连接的建立过程中对错误的处理方式相同，在这个阶段WS可能返回和HTTP相同的返回码。
4. 都可以在网络中传输数据。

#### **不同点**

1. WS使用HTTP来建立连接，但是定义了一系列新的header域，这些域在HTTP中并不会使用。
2. WS的连接不能通过中间人来转发，它必须是一个直接连接。
3. WS连接建立之后，通信双方都可以在任何时刻向另一方发送数据。
4. WS连接建立之后，数据的传输使用帧来传递，不再需要Request消息。
5. WS的数据帧有序。

#### **IMG_259**

#### **HTTP**

HTTP1.1的连接默认使用持续连接（persistent connection），持续连接指的是，有时是客户端会需要在短时间内向服务端请求大量的相关的资源，如果不是持续连接，那么每个资源都要建立一个新的连接，HTTP底层使用的是TCP，那么每次都要使用三次握手建立TCP连接，将造成极大的资源浪费。

持续连接可以带来很多的好处：

1. 使用更少的TCP连接，对通信各方的压力更小。
2. 可以使用管道（pipeline）来传输信息，这样请求方不需要等待结果就可以发送下一条信息，对于单个的TCP的使用更充分。
3. 流量更小
4. 顺序请求的延时更小。
5. 不需要重新建立TCP连接就可以传送error，关闭连接等信息。

HTTP1.1的服务器使用TCP的流量控制来控制HTTP的流量，HTTP1.1的客户端在收到服务器连接中发过来的error信息，就要马上关闭此链接。关于HTTP连接还有很多细节，之后再详述。

### **1.2、websocket与socket**

**短答案**

就像Java和JavaScript，并没有什么太大的关系，但又不能说完全没关系。可以这么说：

* 命名方面，Socket是一个深入人心的概念，WebSocket借用了这一概念；
* 使用方面，完全两个东西。

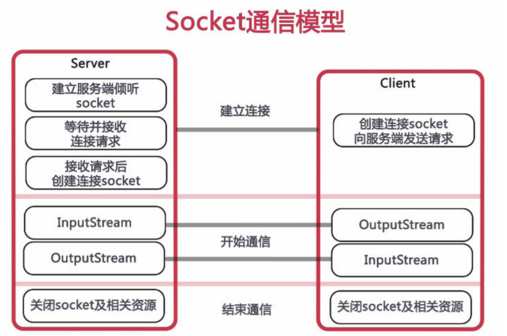
**长答案**

当我们探讨两件事物的区别和联系时，我们想探讨些什么？

对于我来说，大多数情况是想知道两件事物本身，而并不是想只想了解「区别」本身。那么对这个问题最直接的解决方法应该是去了解Socket和WebSocket的来源和用法，那么它们的区别和联系就不言自明了。

Socket

Socket又称"套接字"，应用程序通常通过"套接字"向网络发出请求或者应答网络请求。Socket的英文原义是“孔”或“插座”，作为UNIX的进程通信机制。Socket可以实现应用程序间网络通信。



Socket可以使用TCP/IP协议或UDP协议。

**TCP/IP协议**

TCP/IP协议是目前应用最为广泛的协议，是构成Internet国际互联网协议的最为基础的协议,由TCP和IP协议组成:  
TCP协议:面向连接的、可靠的、基于字节流的传输层通信协议，负责数据的可靠性传输的问题。

IP协议:用于报文交换网络的一种面向数据的协议，主要负责给每台网络设备一个网络地址，保证数据传输到正确的目的地。

**UDP协议**

UDP特点：无连接、不可靠、基于报文的传输层协议，优点是发送后不用管，速度比TCP快。

WebSocket

上边简单叙述了Socket的意义，由于年代久远，很多事情也搞不了那么清楚。但WebSocket是一个很晚近的东西，可以让我们看到它是如何成为现在我们看到的这个样子的。

WHATWG(Web Hypertext Application Technology Working Group)

关于HTML5的故事很多人都是知道的，w3c放弃了HTML，然后有一群人（也有说是这些人供职的公司，不过官方的文档上是说的个人）创立了WHATWG组织来推动HTML语言的继续发展，同时，他们还发展了很多关于Web的技术标准，这些标准不断地被官方所接受。WebSocket就属于WHATWG发布的Web Application的一部分（即HTML5）的产物。

为什么会有WebSocket

大约在08年的时候，WG的工程师在讨论网络环境中需要一种全双工的连接形式，刚开始一直叫做「TCPConnection」，并讨论了这种协议需要支持的功能，大致已经和我们今天看到的WebSocket差不多了。他们认为基于现有的HTTP之上的一些技术（如长轮询、Comet）并满足不了这种需求，有必要定义一个全新的协议。

名称的由来

在很多的关于HTML5或者WebSocket的文档中，都能看到一个名字，Hixie（Ian Hickson），他是WHATWG组织的发言人，曾供职于Netscape、Opera、Google，看工作的公司就知道这个人的背景了。



hixie

08年6月18日，一群WHATWG的工程师在讨论一些技术问题，一个工程师提到说「我们之前讨论的那个东西，不要叫TCPConnection 了，还是起个别的名字吧 」，接着几个名字被提及，DuplexConnection，TCPSocket，SocketConnection ，一个叫mcarter（Michael Carter ）的工程师说他马上要写一篇关于Comet的文章，如果可以确定这个名称，想在文章中引用这个名字。

Socket一直以来都被人用来表示网络中一个连接的两端，考虑到怎么让工程师更容易接受，后来Hixie说了一句「我看WebSocket这个名字就很适合嘛（Hixie briefly pops back online to record that "WebSocket" would probably be a good new name for the TCPConnection object）」，大家都没有异议，紧接着mcarter在Comet Daily中发表了文章[Independence Day: HTML5 WebSocket Liberates Comet From Hacks](http://cometdaily.com/2008/07/04/html5-websocket/index.html" \t "https://www.cnblogs.com/duanxz/p/_blank)，后来随着各大浏览器对WebSocket的支持，它变成了实际的标准，IETF也沿用了这个名字。

下边是在WHATWG文档中对WebSocket接口的定义

[IMG_262](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

enum BinaryType { "blob", "arraybuffer" };

[Constructor(USVString url, optional (DOMString or sequence<DOMString>) protocols = []), Exposed=(Window,Worker)]interface WebSocket : EventTarget {

readonly attribute USVString url;

// ready state

const unsigned short CONNECTING = 0;

const unsigned short OPEN = 1;

const unsigned short CLOSING = 2;

const unsigned short CLOSED = 3;

readonly attribute unsigned short readyState;

readonly attribute unsigned long long bufferedAmount;

// networking attribute EventHandler onopen;

attribute EventHandler onerror;

attribute EventHandler onclose;

readonly attribute DOMString extensions;

readonly attribute DOMString protocol;

void close([Clamp] optional unsigned short code, optional USVString reason);

// messaging attribute EventHandler onmessage;

attribute BinaryType binaryType;

void send(USVString data);

void send(Blob data);

void send(ArrayBuffer data);

void send(ArrayBufferView data);

};

[IMG_263](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

接口的内容可以分为三类：**状态变量、网络功能和消息处理等**。

1. 构造函数WebSocket(url, protocols)：构造WebSocket对象，以及建立和服务器连接; protocols可选字段，代表选择的子协议
2. 状态变量readyState: 代表当前连接的状态，短整型数据，取值为CONNECTING(值为0)， OPEN(值为1), CLOSING(值为2), CLOSED(值为3)
3. 方法变量close(code, reason)： 关闭此WebSocket连接。
4. 状态变量bufferedAmount: send函数调用后，被缓存并且未发送到网络上的数据长度
5. 方法变量send(data): 将数据data通过此WebSocket发送到对端
6. 回调函数onopen/onmessage/onerror/onclose: 当相应的事件发生时会触发此回调函数

内容的确定

大多数新技术的出现都是建立在已有技术的铺垫之上的，WebSocket内容的确定也是如此，其中就有Comet看不到的贡献，Comet是一个很有趣的技术，有兴趣可以[看看这里](http://www.ibm.com/developerworks/cn/web/wa-lo-comet/" \t "https://www.cnblogs.com/duanxz/p/_blank)

结论

可以把WebSocket想象成HTTP，HTTP和Socket什么关系，WebSocket和Socket就是什么关系。

### **1.3、WebSocket**

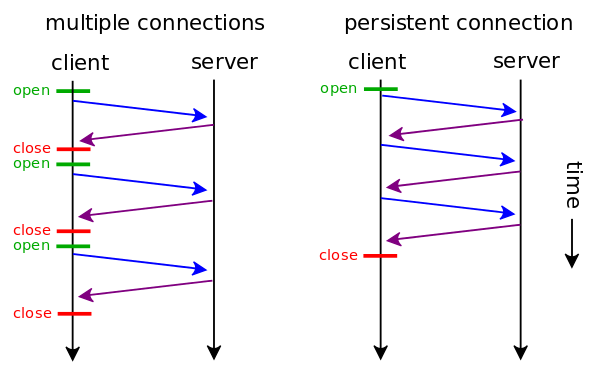
只从RFC发布的时间看来，WebSocket要晚近很多，HTTP 1.1是1999年，WebSocket则是12年之后了。WebSocket协议的开篇就说，本协议的目的是为了解决基于浏览器的程序需要拉取资源时必须发起多个HTTP请求和长时间的轮训的问题而创建的。

WebSocket协议还很年轻，RFC文档相比HTTP的发布时间也很短，它的诞生是为了创建一种「双向通信」的协议，来作为HTTP协议的一个替代者。那么首先看一下它和HTTP（或者HTTP的长连接）的区别。

#### **1.4、为什么要用WebSocket来替代HTTP**

上一篇中提到WebSocket的目的就是解决网络传输中的双向通信的问题，HTTP1.1默认使用持久连接（persistent connection），在一个TCP连接上也可以传输多个Request/Response消息对，但是HTTP的基本模型还是一个Request对应一个Response。这在双向通信（客户端要向服务器传送数据，同时服务器也需要实时的向客户端传送信息，一个聊天系统就是典型的双向通信）时一般会使用这样几种解决方案：

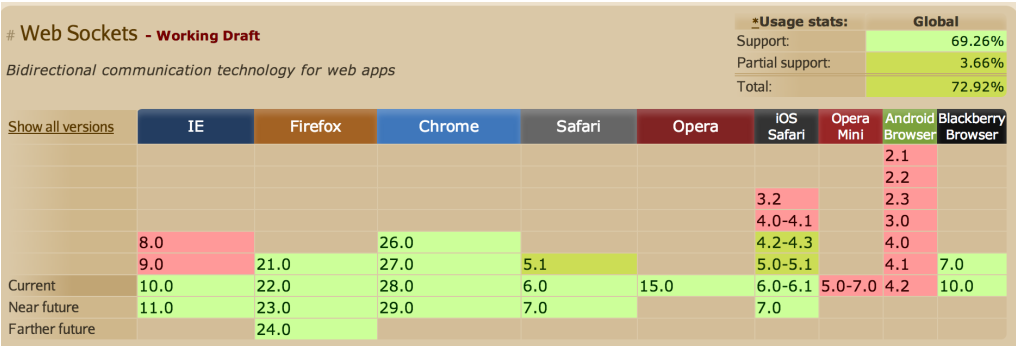
1. 轮询（polling），轮询就会造成对网络和通信双方的资源的浪费，且非实时。
2. 长轮询，客户端发送一个超时时间很长的Request，服务器hold住这个连接，在有新数据到达时返回Response，相比#1，占用的网络带宽少了，其他类似。
3. 长连接，其实有些人对长连接的概念是模糊不清的，我这里讲的其实是HTTP的长连接（1）。如果你使用Socket来建立TCP的长连接（2），那么，这个长连接（2）跟我们这里要讨论的WebSocket是一样的，实际上TCP长连接就是WebSocket的基础，但是如果是HTTP的长连接，本质上还是Request/Response消息对，仍然会造成资源的浪费、实时性不强等问题。



HTTP的长连接模型

websocket缺点：

少部分浏览器不支持，浏览器支持的程度与方式有区别。



## **二、协议基础**

WebSocket的目的是取代HTTP在双向通信场景下的使用，而且它的实现方式有些也是基于HTTP的（WS的默认端口是80和443）。现有的网络环境（客户端、服务器、网络中间人、代理等）对HTTP都有很好的支持，所以这样做可以充分利用现有的HTTP的基础设施，有点向下兼容的意味。

简单来讲，**WS协议有两部分组成：握手和数据传输**。

### **2.1、握手（handshake）**

出于兼容性的考虑，WS的握手使用HTTP来实现（此文档中提到未来有可能会使用专用的端口和方法来实现握手），客户端的握手消息就是一个「普通的，带有Upgrade头的，HTTP Request消息」。所以这一个小节到内容大部分都来自于RFC2616，这里只是它的一种应用形式，下面是RFC6455文档中给出的一个客户端握手消息示例：

[IMG_266](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

GET /chat HTTP/1.1 //1

Host: server.example.com //2

Upgrade: websocket //3

Connection: Upgrade //4

Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ== //5

Origin: http://example.com //6

Sec-WebSocket-Protocol: chat, superchat //7

Sec-WebSocket-Version: 13 //8

[IMG_267](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

可以看到，前两行跟HTTP的Request的起始行一模一样，而真正在WS的握手过程中起到作用的是下面几个header域。

Upgrade：upgrade是HTTP1.1中用于定义转换协议的header域。它表示，如果服务器支持的话，客户端希望使用现有的「网络层」已经建立好的这个「连接（此处是TCP连接）」，切换到另外一个「应用层」（此处是WebSocket）协议。

Connection：HTTP1.1中规定Upgrade只能应用在「直接连接」中，所以带有Upgrade头的HTTP1.1消息必须含有Connection头，因为Connection头的意义就是，任何接收到此消息的人（往往是代理服务器）都要在转发此消息之前处理掉Connection中指定的域（不转发Upgrade域）。  
如果客户端和服务器之间是通过代理连接的，那么在发送这个握手消息之前首先要发送CONNECT消息来建立直接连接。

首先， Sec-WebSocket-Key 是一个 Base64 encode 的值，这个是浏览器随机生成的，发送给服务器使用（服务器会使用此字段组装成另一个key值放在握手返回信息里发送客户端）。用于验证服务端是不是真的是Websocket代理。

然后， Sec\_WebSocket-Protocol 是一个用户定义的字符串，标识了客户端支持的子协议的列表。用来区分同URL下，不同的服务所需要的协议。

最后， Sec-WebSocket-Version 是告诉服务器所使用的 Websocket Draft（协议版本），在最初的时候，Websocket协议还在 Draft 阶段，各种奇奇怪怪的协议都有，而且还有很多期奇奇怪怪不同的东西，什么Firefox和Chrome用的不是一个版本之类的，当初Websocket协议太多可是一个大难题。

Origin：作安全使用，防止跨站攻击，浏览器一般会使用这个来标识原始域。

如果服务器接受了这个请求，可能会发送如下这样的返回信息，这是一个标准的HTTP的Response消息。101表示服务器收到了客户端切换协议的请求，并且同意切换到此协议。RFC2616规定只有切换到的协议「比HTTP1.1更好」的时候才能同意切换。

HTTP/1.1 101 Switching Protocols //1

Upgrade: websocket. //2

Connection: Upgrade. //3

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo= //4

Sec-WebSocket-Protocol: chat. //5

1.第1行，101表示服务器收到了客户端切换协议的请求，并且同意切换到此协议。

2.第2,3行，依然是固定的，告诉客户端即将升级的是 Websocket 协议。

3.第4行，这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key。

4.第5行，则是表示最终使用的协议。

##### **WebSocket协议Uri**

ws协议默认使用80端口，wss协议默认使用443端口。

[IMG_268](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]

wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]

host = <host, defined in [RFC3986], Section 3.2.2>

port = <port, defined in [RFC3986], Section 3.2.3>

path = <path-abempty, defined in [RFC3986], Section 3.3>

query = <query, defined in [RFC3986], Section 3.4>

[IMG_269](https://www.cnblogs.com/duanxz/p/javascript:void(0);)

##### **在客户端发送握手之前要做的一些小事**

在握手之前，客户端首先要先建立连接，一个客户端对于一个相同的目标地址（通常是域名或者IP地址，不是资源地址）同一时刻只能有一个处于CONNECTING状态（就是正在建立连接）的连接。从建立连接到发送握手消息这个过程大致是这样的：

1. 客户端检查输入的Uri是否合法。
2. 客户端判断，如果当前已有指向此目标地址（IP地址）的连接（A）仍处于CONNECTING状态，需要等待这个连接（A）建立成功，或者建立失败之后才能继续建立新的连接。  
   **PS**：如果当前连接是处于代理的网络环境中，无法判断IP地址是否相同，则认为每一个Host地址为一个单独的目标地址，同时客户端应当限制同时处于CONNECTING状态的连接数。  
   **PPS**：这样可以防止一部分的DDOS攻击。  
   **PPPS**：客户端并不限制同时处于「已成功」状态的连接数，但是如果一个客户端「持有大量已成功状态的连接的」，服务器或许会拒绝此客户端请求的新连接。

如果客户端处于一个代理环境中，它首先要请求它的代理来建立一个到达目标地址的TCP连接。  
例如，如果客户端处于代理环境中，它想要连接某目标地址的80端口，它可能要收现发送以下消息：

CONNECT example.com:80 HTTP/1.1

Host: example.com

如果客户端没有处于代理环境中，它就要首先建立一个到达目标地址的直接的TCP连接。

1. 如果上一步中的TCP连接建立失败，则此WebSocket连接失败。
2. 如果协议是wss，则在上一步建立的TCP连接之上，使用TSL发送握手信息。如果失败，则此WebSocket连接失败；如果成功，则以后的所有数据都要通过此TSL通道进行发送。

##### **对于客户端握手信息的一些小要求**

1. 握手必须是RFC2616中定义的Request消息
2. 此Request消息的方法必须是GET，HTTP版本必须大于1.1 。  
   以下是某WS的Uri对应的Request消息：

ws://example.com/chat

GET /chat HTTP/1.1

1. 此Request消息中Request-URI部分（RFC2616中的概念）所定义的资型必须和WS协议的Uri中定义的资源相同。
2. 此Request消息中必须含有Host头域，其内容必须和WS的Uri中定义的相同。
3. 此Request消息必须包含Upgrade头域，其内容必须包含websocket关键字。
4. 此Request消息必须包含Connection头域，其内容必须包含Upgrade指令。
5. 此Request消息必须包含Sec-WebSocket-Key头域，其内容是一个Base64编码的16位随机字符。
6. 如果客户端是浏览器，此Request消息必须包含Origin头域，其内容是参考[RFC6454](https://tools.ietf.org/html/rfc6454" \t "https://www.cnblogs.com/duanxz/p/_blank)。
7. 此Request消息必须包含Sec-WebSocket-Version头域，在此协议中定义的版本号是13。
8. 此Request消息可能包含Sec-WebSocket-Protocol头域，其意义如上文中所述。
9. 此Request消息可能包含Sec-WebSocket-Extensions头域，客户端和服务器可以使用此header来进行一些功能的扩展。
10. 此Request消息可能包含任何合法的头域。如RFC2616中定义的那些。

##### **在客户端接收到Response握手消息之后要做的一些事情**

1. 如果返回的返回码不是101，则按照RFC2616进行处理。如果是101，进行下一步，开始解析header域，所有header域的值不区分大小写。
2. 判断是否含有Upgrade头，且内容包含websocket。
3. 判断是否含有Connection头，且内容包含Upgrade
4. 判断是否含有Sec-WebSocket-Accept头，其内容在下面介绍。
5. 如果含有Sec-WebSocket-Extensions头，要判断是否之前的Request握手带有此内容，如果没有，则连接失败。
6. 如果含有Sec-WebSocket-Protocol头，要判断是否之前的Request握手带有此协议，如果没有，则连接失败。

##### **服务端的概念**

服务端指的是所有参与处理WebSocket消息的基础设施，比如如果某服务器使用Nginx（A）来处理WebSocket，然后把处理后的消息传给响应的服务器（B），那么A和B都是这里要讨论的服务端的范畴。

##### **接受了客户端的连接请求，服务端要做的一些事情**

如果请求是HTTPS，则首先要使用TLS进行握手，如果失败，则关闭连接，如果成功，则之后的数据都通过此通道进行发送。

之后服务端可以进行一些客户端验证步骤（包括对客户端header域的验证），如果需要，则按照RFC2616来进行错误码的返回。

如果一切都成功，则返回成功的Response握手消息。

##### **服务端发送的成功的Response握手**

此握手消息是一个标准的HTTP Response消息，同时它包含了以下几个部分：

1. 状态行（如上一篇RFC2616中所述）
2. Upgrade头域，内容为websocket
3. Connection头域，内容为Upgrade
4. Sec-WebSocket-Accept头域，其内容的生成步骤：
   1. 首先将Sec-WebSocket-Key的内容加上字符串258EAFA5-E914-47DA-95CA-C5AB0DC85B11（一个UUID）。
   2. 将#1中生成的字符串进行SHA1编码。
   3. 将#2中生成的字符串进行Base64编码。
5. Sec-WebSocket-Protocol头域（可选）
6. Sec-WebSocket-Extensions头域（可选）

一旦这个握手发出去，服务端就认为此WebSocket连接已经建立成功，处于OPEN状态。它就可以开始发送数据了。

##### **WebSocket的一些扩展**

Sec-WebSocket-Version可以被通信双方用来支持更多的协议的扩展，RFC6455中定义的值为13，WebSocket的客户端和服务端可能回自定义更多的版本号来支持更多的功能。其使用方法如上文所述。

#### **2.2、发送数据**

WebSocket中所有发送的数据使用帧的形式发送。客户端发送的数据帧都要经过掩码处理，服务端发送的所有数据帧都不能经过掩码处理。否则对方需要发送关闭帧。

一个帧包含一个帧类型的标识码，一个负载长度，和负载。负载包括扩展内容和应用内容。

##### **帧类型**

帧类型是由一个4位长的叫Opcode的值表示，任何WebSocket的通信方收到一个位置的帧类型，都要以连接失败的方式断开此连接。  
RFC6455中定义的帧类型如下所示：

Opcode == 0 继续

表示此帧是一个继续帧，需要拼接在上一个收到的帧之后，来组成一个完整的消息。由于这种解析特性，非控制帧的发送和接收必须是相同的顺序。

1. Opcode == 1 文本帧
2. Opcode == 2 二进制帧
3. Opcode == 3 - 7 未来使用（非控制帧）
4. Opcode == 8 关闭连接（控制帧）  
   此帧可能会包含内容，以表示关闭连接的原因。  
   通信的某一方发送此帧来关闭WebSocket连接，收到此帧的一方如果之前没有发送此帧，则需要发送一个同样的关闭帧以确认关闭。如果双方同时发送此帧，则双方都需要发送回应的关闭帧。  
   理想情况服务端在确认WebSocket连接关闭后，关闭相应的TCP连接，而客户端需要等待服务端关闭此TCP连接，但客户端在某些情况下也可以关闭TCP连接。
5. Opcode == 9 Ping  
   类似于心跳，一方收到Ping，应当立即发送Pong作为响应。
6. Opcode == 10 Pong  
   如果通信一方并没有发送Ping，但是收到了Pong，并不要求它返回任何信息。Pong帧的内容应当和收到的Ping相同。可能会出现一方收到很多的Ping，但是只需要响应最近的那一次就可以了。
7. Opcode == 11 - 15 未来使用（控制帧）

目前有6种帧是可用的。**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。**

##### **帧的格式**

具体的每一项代表什么意思在这里就不做详细的阐述了。

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 ... |

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

其中最重要的字段为opcode(4bit)和MASK(1bit)：

* MASK值，从客户端进行发送的帧必须置此位为1，从服务器发送的帧必须置为0。如果任何一方收到的帧不符合此要求，则发送关闭帧(Close frame)关闭连接。
* opcode的值： 0x1代表此帧为文本数据帧, 0x2代表此帧为二进制数据帧, 0x8为控制帧中的连接关闭帧(close frame), 0x9为控制帧中的Ping帧, 0xA(十进制的10)为控制帧中的Pong帧。
* Ping/Pong帧： Ping帧和Pong帧用于连接的保活(keepalive)或者诊断对端是否在线。这两种帧的发送和接收不对WEB应用公开接口，由实现WebSocket协议的底层应用(例如浏览器)来实现它。