作者介绍：码农，热爱C++、Windows/Linux、网络编程、开源等。
博客介绍：本博客记录本人学习、工作中的所学所闻，博客中部分内容来自于网络上大牛、前辈的文章等，在此感谢。

昵称：lizhenghn
园龄：3年2个月
粉丝：65
关注：10

Websocket协议的学习、调研和实现

本文章同时发在 cpper.info。

1. websocket是什么

Websocket是html5提出的一个协议规范，参考rfc6455。

websocket约定了一个通信的规范，通过一个握手的机制，客户端（浏览器）和服务器（webserver）之间能建立一个类似tcp的连接，从而方便c－s之间的通信。在websocket出现之前，web交互一般是基于http协议的短连接或者长连接。

WebSocket是为解决客户端与服务端实时通信而产生的技术。websocket协议本质上是一个基于tcp的协议，是先通过HTTP/HTTPS协议发起一条特殊的http请求进行握手后创建一个用于交换数据的TCP连接，此后服务端与客户端通过此TCP连接进行实时通信。

注意：此时不再需要原**HTTP**协议的参与了。

2. websocket的优点

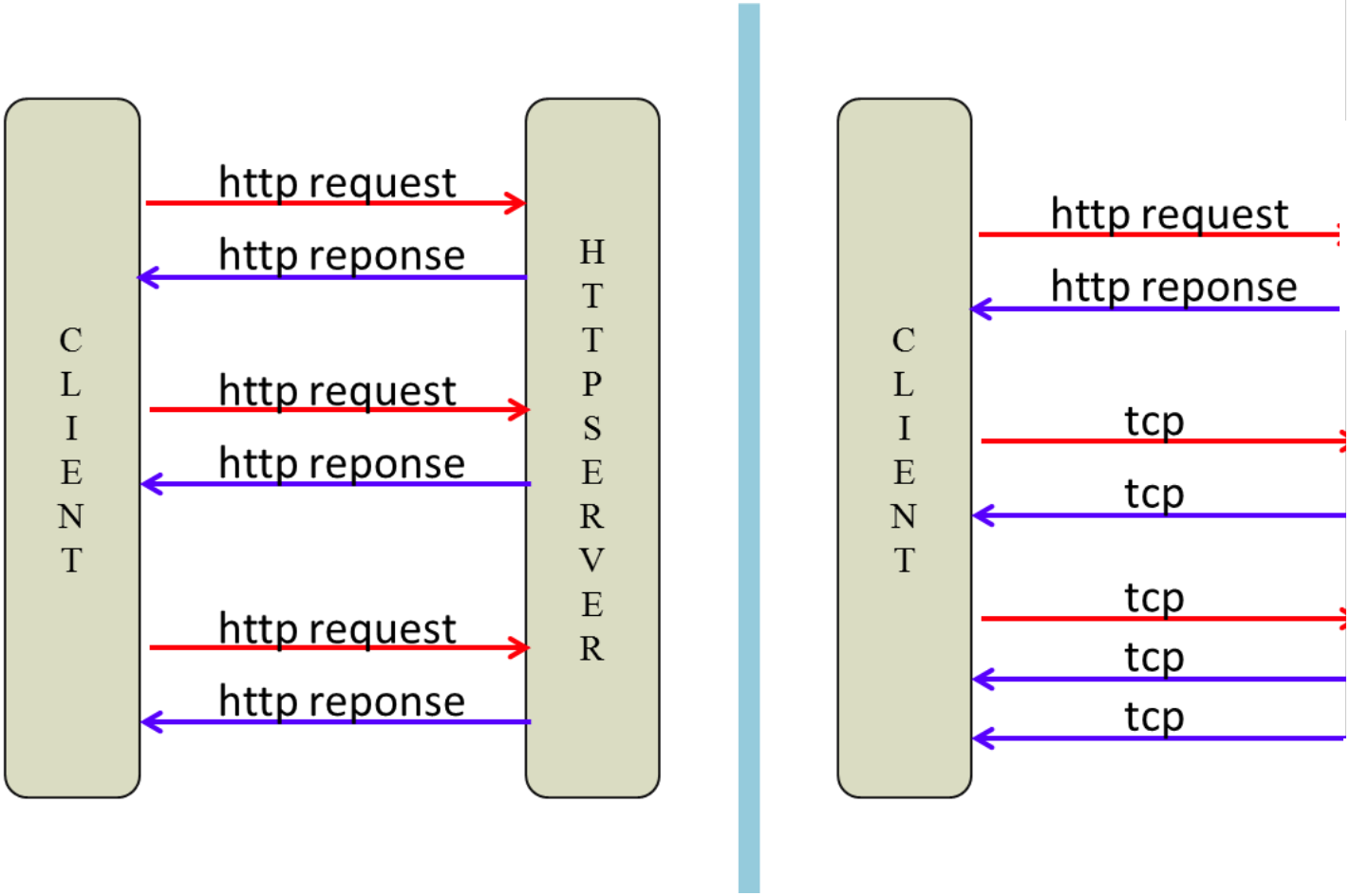
以前web server实现推送技术或者即时通讯，用的都是轮询（polling），在特点的时间间隔（比如1秒钟）由浏览器自动发出请求，将服务器的消息主动的拉回来，在这种情况下，我们需要不断的向服务器发送请求，然而HTTP request 的header是非常长的，里面包含的数据可能只是一个很小的值，这样会占用很多的带宽和服务器资源。

而最比较新的技术去做轮询的效果是Comet – 用了AJAX。但这种技术虽然可达到全双工通信，但依然需要发出请求(reuqest)。

WebSocket API最伟大之处在于服务器和客户端可以在给定的时间范围内的任意时刻，相互推送信息。 浏览器和服务器只需要要做一个握手的动作，在建立连接之后，服务器可以主动传送数据给客户端，客户端也可以随时向服务器

发送数据。 此外，服务器与客户端之间交换的标头信息很小。

WebSocket并不限于以Ajax(或XHR)方式通信，因为Ajax技术需要客户端发起请求，而WebSocket服务器和客户端可以彼此相互推送信息；



因此从服务器角度来说，websocket有以下好处：

1. 节省每次请求的header
   http的header一般有几十字节
2. Server Push
   服务器可以主动传送数据给客户端

我的标签

C++(20)

linux(14)

开发(11)

TCP(5)

windows(5)

网络(5)

c++11(3)

glog(2)

log(2)

3. 历史沿革

3.1 http协议

1996年IETF HTTP工作组发布了HTTP协议的1.0版本，到现在普遍使用的版本1.1，HTTP协议经历了17年的发展。这种分布式、无状态、基于TCP的请求/响应式、在互联网盛行的今天得到广泛应用的协议。互联网从兴起到现在，经历了门户网站盛行的web1.0时代，而后随着ajax技术的出现，发展为web应用盛行的web2.0时代，如今又朝着web3.0的方向迈进。反观http协议，从版本1.0发展到1.1，除了默认长连接之外就是缓存处理、带宽优化和安全性等方面的不痛不痒的改进。它一直保留着无状态、请求/响应模式，似乎从来没意识到这应该有所改变。

3.2 通过脚本发送的http请求（Ajax）

传统的web应用要想与服务器交互，必须提交一个表单（form），服务器接收并处理传来的表单，然后返回全新的页面，因为前后两个页面的数据大部分都是相同的，这个过程传输了很多冗余的数据、浪费了带宽。于是Ajax技术便应运而生。

Ajax是Asynchronous JavaScript and 的简称，由Jesse James Garrett 首先提出。这种技术开创性地允许浏览器脚本（JS）发送http请求。Outlook Web Access小组于98年使用，并很快成为IE4.0的一部分，但是这个技术一直很小众，直到2005年初，google在他的goole groups、gmail等交互式应用中广泛使用此种技术，才使得Ajax迅速被大家所接受。

Ajax的出现使客户端与服务器端传输数据少了很多，也快了很多，也满足了以丰富用户体验为特点的web2.0时代 初期发展的需要，但是慢慢地也暴露了他的弊端。比如无法满足即时通信等富交互式应用的实时更新数据的要求。这种浏览器端的小技术毕竟还是基于http协议，http协议要求的请求/响应的模式也是无法改变的，除非http协议本身有所改变。

3.3 一种hack技术（Comet）

以即时通信为代表的web应用程序对数据的Low Latency要求，传统的基于轮询的方式已经无法满足，而且也会带来不好的用户体验。于是一种基于http长连接的"服务器推"技术便被hack出来。这种技术被命名为Comet，这个术语由Dojo Toolkit 的项目主管Alex Russell在博文Comet: Low Latency Data for the Browser首次提出，并沿用下来。

其实，服务器推很早就存在了，在经典的client/server模型中有广泛使用，只是浏览器太懒了，并没有对这种技术提供很好的支持。但是Ajax的出现使这种技术在浏览器上实现成为可能， google的gmail和gtalk的整合首先使用了这种技术。随着一些关键问题的解决（比如IE的加载显示问题），很快这种技术得到了认可，目前已经有很多成熟的开源Comet框架。

面向对象(2)

更多

随笔分类(84)

C++(29)

GCC(4)

Git&Github(2)

Linux(15)

Linux开发(13)

Windows(9)

软件工程(5)

网络编程(7)

随笔档案(38)

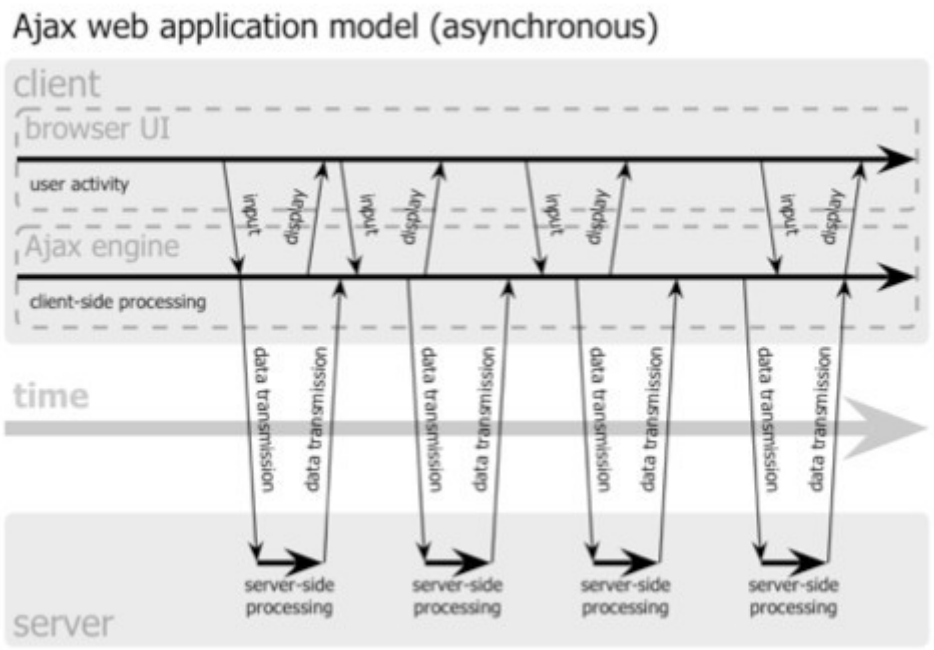
2016年3月 (2)

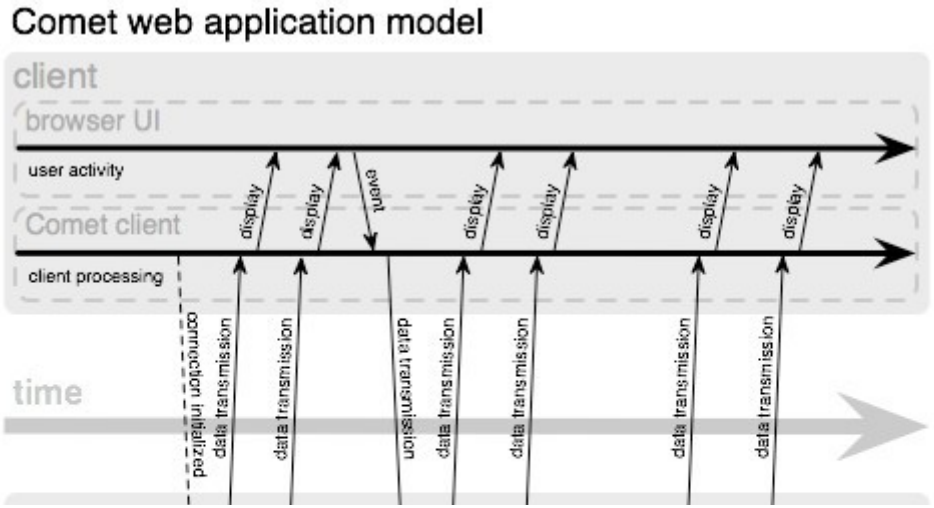
2016年1月 (3)

2014年10月 (1)

2014年9月 (1)

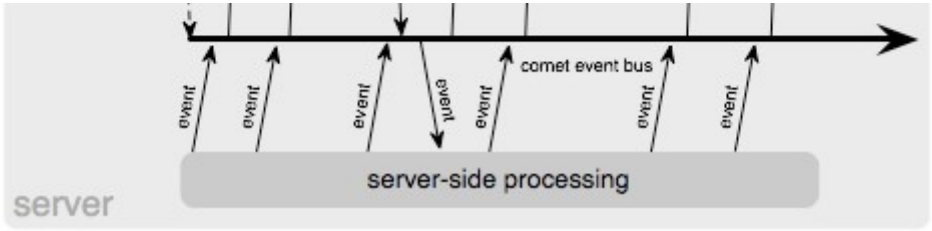
以下是典型的Ajax和Comet数据传输方式的对比，区别简单明了。典型的Ajax通信方式也是http协议的经典使用方式，要想取得数据，必须首先发送请求。在Low Latency要求比较高的web应用中，只能增加服务器请求的频率。Comet则不同，客户端与服务器端保持一个长连接，只有客户端需要的数据更新时，服务器才主动将数据推送给客户端。





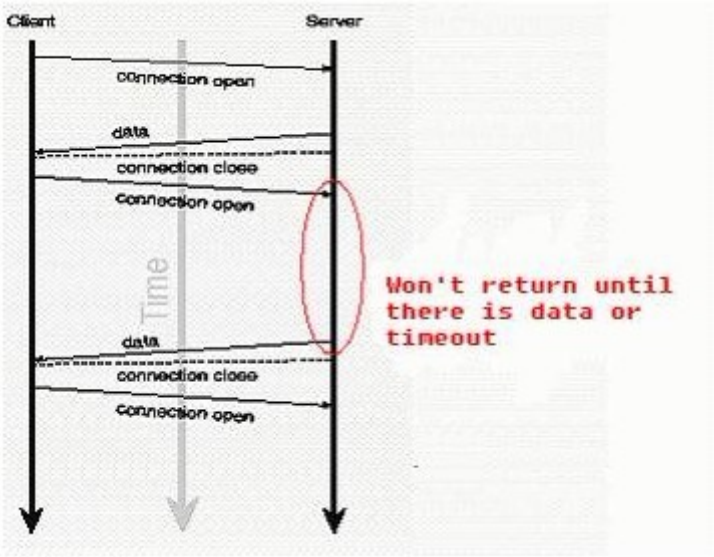
2014年8月 (1)
2014年7月 (1)
2014年6月 (1)
2014年5月 (8)
2014年4月 (11)
2014年3月 (8)
2014年2月 (1)

最新评论
1. Re:Websocket协议的学习、调研和实现
@嘻嘻小北加我qq聊：1576410833...
--承影剑



Comet的实现主要有两种方式：

- 基于Ajax的长轮询（long-polling）方式



- 基于 Iframe 及 htmlfile 的流（http streaming）方式

Iframe是html标记，这个标记的src属性会保持对指定服务器的长连接请求，服务器端则可以不停地返回数据，相对于第一种方式，这种方式跟传统的服务器推则更接近。
在第一种方式中，浏览器在收到数据后会直接调用JS回调函数，但是这种方式该如何响应数据呢？可以通过在返回数据中嵌入JS脚本的方式，如“”，服务器端将返回的数据作为回调函数的参数，浏览器在收到数据后就会执行这段JS脚本。

2. Re:Websocket协议的学习、调研和实现

@承影剑哥们,能把你c#实现的代码给我瞧瞧吗？我老是在握手时出现500或者502...

--嘻嘻小北

3. Re:Websocket协议的学习、调研和实现

good ， 内容很好，谢谢楼主

--David_Lin

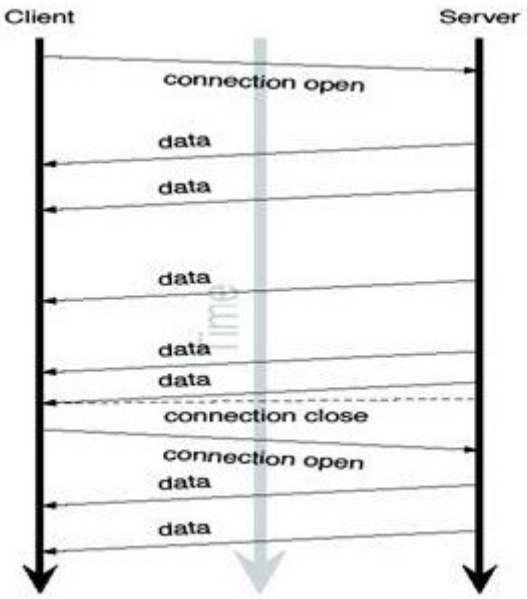
4. Re:Websocket协议的学习、调研和实现

前段时间研究了一下goeasy,代码简洁易读，服务稳定。后台推送只需要两行代码， js前端推送也只需要3，4行，而且文档齐全，还提供了后台查询信息收发情况，所以我觉得GoEasy推送服务是个不错的选择.....

--gongmaolan123

5. Re:C++中指针常量和常量指针的区别

描述的很清晰，受益了^-^



3.4 Websocket---未来的解决方案

如果说Ajax的出现是互联网发展的必然，那么Comet技术的出现则更多透露出一种无奈，仅仅作为一种hack技术，因为没有更好的解决方案。Comet解决的问题应该由谁来解决才是合理的呢？浏览器，html标准，还是http标准？主角应该是谁呢？本质上讲，这涉及到数据传输方式，http协议应首当其冲，是时候改变一下这个懒惰的协议的请求/响应模式了。

W3C给出了答案，在新一代html标准html5中提供了一种浏览器和服务器间进行全双工通讯的网络技术Websocket。从Websocket草案得知，Websocket是一个全新的、独立的协议，基于TCP协议，与http协议兼容、却不会融入http协议，仅仅作为html5的一部分。于是乎脚本又被赋予了另一种能力：发起websocket请求。这种方式我们应该很熟悉，因为Ajax就是这么做的，所不同的是，Ajax发起的是http请求而已。

4. websocket逻辑

与http协议不同的请求/响应模式不同，Websocket在建立连接之前有一个Handshake（Opening Handshake）过程，在关闭连接前也有一个Handshake（Closing Handshake）过程，建立连接之后，双方即可双向通信。
在websocket协议发展过程中前前后后就出现了多个版本的握手协议，这里分情况说明一下：

- 基于flash的握手协议
  使用场景是IE的多数版本，因为IE的多数版本不都不支持WebSocket协议，以及FF、CHROME等浏览器的低版

本，还没有原生的支持WebSocket。此处，server唯一要做的，就是准备一个WebSocket-Location域给client，没有加密，可靠性很差。

客户端请求：

```
GET /ls HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: www.qixing318.com
Origin: http://www.qixing318.com
```

服务器返回：

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://www.qixing318.com
WebSocket-Location: ws://www.qixing318.com/ls
```

- 基于md5加密方式的握手协议
  客户端请求：

  GET /demo HTTP/1.1
  Host: example.com
  Connection: Upgrade
  Sec-WebSocket-Key2:
  Upgrade: WebSocket
  Sec-WebSocket-Key1:
  Origin: http://www.qixing318.com
  [8-byte security key]

服务端返回：

```
HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://www.qixing318.com
```

--FindADK

阅读排行榜

评论排行榜

5. do{...}while(0)的妙用(9)

推荐排行榜

1. Linux下常用的shell命令记录(6)

2. Websocket协议的学习、调研和实现(5)

3. Linux下Vim工具常用命令(5)

4. do{...}while(0)的妙用(5)

5. C++中的静态多态和动态多态(4)

```
WebSocket-Location: ws://example.com/demo
[16-byte hash response]
```

其中 Sec-WebSocket-Key1，Sec-WebSocket-Key2 和 [8-byte security key] 这几个头信息是web server用来生成应答信息的来源，依据 draft-hixie-thewebsocketprotocol-76 草案的定义。
web server基于以下的算法来产生正确的应答信息：

1. 逐个字符读取 Sec-WebSocket-Key1 头信息中的值，将数值型字符连接到一起放到一个临时字符串里，同时统计所有空格的数量；
2. 将在第（1）步里生成的数字字符串转换成一个整型数字，然后除以第（1）步里统计出来的空格数量，将得到的浮点数转换成整数型；
3. 将第（2）步里生成的整型值转换为符合网络传输的网络字节数组；
4. 对 Sec-WebSocket-Key2 头信息同样进行第（1）到第（3）步的操作，得到另外一个网络字节数组；
5. 将 [8-byte security key] 和在第（3）、（4）步里生成的网络字节数组合并成一个16字节的数组；
6. 对第（5）步生成的字节数组使用MD5算法生成一个哈希值，这个哈希值就作为安全密钥返回给客户端，以表明服务器端获取了客户端的请求，同意创建websocket连接

- 基于sha加密方式的握手协议
  也是目前见的最多的一种方式，这里的版本号目前是需要13以上的版本。
  客户端请求：

  GET /ls HTTP/1.1
  Upgrade: websocket
  Connection: Upgrade
  Host: www.qixing318.com
  Sec-WebSocket-Origin: http://www.qixing318.com
  Sec-WebSocket-Key: 2SCVXUeP9cTjV+0mWB8J6A==
  Sec-WebSocket-Version: 13

服务器返回：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: mLDKNeBNWz6T9SxU+o0Fy/HgeSw=
```

其中 server就是把客户端上报的key拼上一段GUID（“258EAFA5-E914-47DA-95CA-C5AB0DC85B11”），拿这个字符串做SHA-1 hash计算，然后再把得到的结果通过base64加密，最后再返回给客户端。

4.1 Opening Handshake：

客户端发起连接Handshake请求

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

服务器端响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

- Upgrade：WebSocket
  表示这是一个特殊的 HTTP 请求，请求的目的就是要将客户端和服务器端的通讯协议从 HTTP 协议升级到 WebSocket 协议。

- Sec-WebSocket-Key
  是一段浏览器base64加密的密钥，server端收到后需要提取Sec-WebSocket-Key 信息，然后加密。

- Sec-WebSocket-Accept
  服务器端在接收到的Sec-WebSocket-Key密钥后追加一段神奇字符串“258EAFA5-E914-47DA-95CA-C5AB0DC85B11”，并将结果进行sha-1哈希，然后再进行base64加密返回给客户端（就是Sec-WebSocket-Key）。 比如：

```
function encry($req)
{
    $key = $this->getKey($req);
    $mask = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
    # 将 SHA-1 加密后的字符串再进行一次 base64 加密
```

```
    return base64_encode(sha1($key . '258EAFA5-E914-47DA-95CA-C5AB0DC85B11', true));
}
```

如果加密算法错误，客户端在进行校检的时候会直接报错。如果握手成功，则客户端侧会出发onopen事件。

- Sec-WebSocket-Protocol
  表示客户端请求提供的可供选择的子协议，及服务器端选中的支持的子协议，“Origin”服务器端用于区分未授权的websocket浏览器

- Sec-WebSocket-Version: 13
  客户端在握手时的请求中携带，这样的版本标识，表示这个是一个升级版本，现在的浏览器都是使用的这个版本。

- HTTP/1.1 101 Switching Protocols
  101为服务器返回的状态码，所有非101的状态码都表示handshake并未完成。

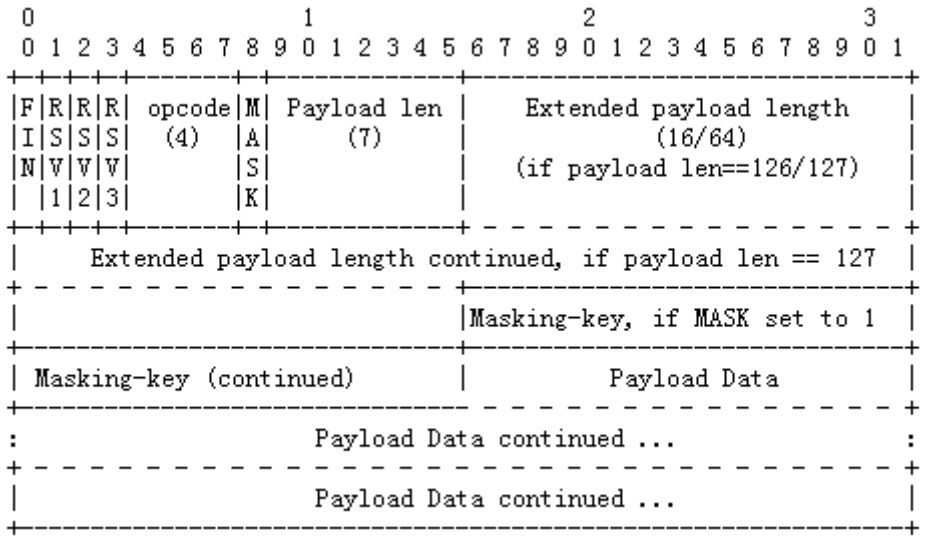
4.2 Data Framing

Websocket协议通过序列化的数据帧传输数据。数据封包协议中定义了opcode、payload length、Payload data等字段。其中要求：

1. 客户端向服务器传输的数据帧必须进行掩码处理：服务器若接收到未经过掩码处理的数据帧，则必须主动关闭连接。

2. 服务器向客户端传输的数据帧一定不能进行掩码处理。客户端若接收到经过掩码处理的数据帧，则必须主动关闭连接。

针对上情况，发现错误的一方可向对方发送close帧（状态码是1002，表示协议错误），以关闭连接。
具体数据帧格式如下图所示：



- FIN
  标识是否为此消息的最后一个数据包，占 1 bit
- RSV1, RSV2, RSV3: 用于扩展协议，一般为0，各占1bit
- Opcode
  数据包类型（frame type），占4bits
  0x0：标识一个中间数据包
  0x1：标识一个text类型数据包
  0x2：标识一个binary类型数据包
  0x3-7：保留
  0x8：标识一个断开连接类型数据包
  0x9：标识一个ping类型数据包
  0xA：表示一个pong类型数据包
  0xB-F：保留
- MASK：占1bits
  用于标识PayloadData是否经过掩码处理。如果是1，Masking-key域的数据即是掩码密钥，用于解码PayloadData。客户端发出的数据帧需要进行掩码处理，所以此位是1。

- Payload length
  Payload data的长度，占7bits，7+16bits，7+64bits：
  - 如果其值在0-125，则是payload的真实长度。
  - 如果值是126，则后面2个字节形成的16bits无符号整型数的值是payload的真实长度。注意，网络字节序，需要转换。
  - 如果值是127，则后面8个字节形成的64bits无符号整型数的值是payload的真实长度。注意，网络字节序，需要转换。

这里的长度表示遵循一个原则，用最少的字节表示长度（尽量减少不必要的传输）。举例说，payload真实长度是124，在0-125之间，必须用前7位表示；不允许长度1是126或127，然后长度2是124，这样违反原则。

- Payload data
  应用层数据

  server解析**client**端的数据

  接收到客户端数据后的解析规则如下：

- 1byte
  - 1bit: frame-fin，x0表示该message后续还有frame；x1表示是message的最后一个frame
  - 3bit: 分别是frame-rsv1、frame-rsv2和frame-rsv3，通常都是x0
  - 4bit: frame-opcode，x0表示是延续frame；x1表示文本frame；x2表示二进制frame；x3-7保留给非控制frame；x8表示关 闭连接；x9表示ping；xA表示pong；xB-F保留给控制frame
- 2byte
  - 1bit: Mask，1表示该frame包含掩码；0表示无掩码
  - 7bit、7bit+2byte、7bit+8byte: 7bit取整数值，若在0-125之间，则是负载数据长度；若是126表示，后两个byte取无符号16位整数值，是负载长度；127表示后8个 byte，取64位无符号整数值，是负载长度
  - 3-6byte: 这里假定负载长度在0-125之间，并且Mask为1，则这4个byte是掩码
  - 7-end byte: 长度是上面取出的负载长度，包括扩展数据和应用数据两部分，通常没有扩展数据；若Mask为1，则此数据需要解码，解码规则为- 1-4byte掩码循环和数据byte做异或操作。

示例代码：

```
/// 解析客户端数据包
/// <param name="recBytes">服务器接收的数据包</param>
/// <param name="recByteLength">有效数据长度</param>
private static string AnalyticData(byte[] recBytes, int recByteLength)
{
    if(recByteLength < 2)
    {
        return string.Empty;
    }

    bool fin = (recBytes[0] & 0x80) == 0x80; // 1bit，1表示最后一帧
    if(!fin)
    {
        return string.Empty;// 超过一帧暂不处理
    }

    bool mask_flag = (recBytes[1] & 0x80) == 0x80; // 是否包含掩码
    if(!mask_flag)
    {
        return string.Empty;// 不包含掩码的暂不处理
    }

    int payload_len = recBytes[1] & 0x7F; // 数据长度

    byte[] masks = new byte[4];
    byte[] payload_data;

    if(payload_len == 126)
    {
        Array.Copy(recBytes, 4, masks, 0, 4);
        payload_len = (UInt16)(recBytes[2] << 8 | recBytes[3]);
        payload_data = new byte[payload_len];
        Array.Copy(recBytes, 8, payload_data, 0, payload_len);

    }
    else if(payload_len == 127)
    {
```

```
            Array.Copy(recBytes, 10, masks, 0, 4);
            byte[] uInt64Bytes = new byte[8];
            for(int i = 0; i < 8; i++)
            {
                uInt64Bytes[i] = recBytes[9 - i];
            }
            UInt64 len = BitConverter.ToUInt64(uInt64Bytes, 0);

            payload_data = new byte[len];
            for(UInt64 i = 0; i < len; i++)
            {
                payload_data[i] = recBytes[i + 14];
            }
        }
        else
        {
            Array.Copy(recBytes, 2, masks, 0, 4);
            payload_data = new byte[payload_len];
            Array.Copy(recBytes, 6, payload_data, 0, payload_len);

        }

        for(var i = 0; i < payload_len; i++)
        {
            payload_data[i] = (byte)(payload_data[i] ^ masks[i % 4]);
        }
        return Encoding.UTF8.GetString(payload_data);
    }
```

server发送数据至client

服务器发送的数据以0x81开头，紧接发送内容的长度（若长度在0-125，则1个byte表示长度；若长度不超过0xFFFF，则后2个byte 作为无符号16位整数表示长度；若超过0xFFFF，则后8个byte作为无符号64位整数表示长度），最后是内容的byte数组。
示例代码：

```
/// 打包服务器数据
/// <param name="message">数据</param>
/// <returns>数据包</returns>
```

```
private static byte[] PackData(string message)
{
    byte[] contentBytes = null;
    byte[] temp = Encoding.UTF8.GetBytes(message);

    if(temp.Length < 126)
    {
        contentBytes = new byte[temp.Length + 2];
        contentBytes[0] = 0x81;
        contentBytes[1] = (byte)temp.Length;
        Array.Copy(temp, 0, contentBytes, 2, temp.Length);
    }
    else if(temp.Length < 0xFFFF)
    {
        contentBytes = new byte[temp.Length + 4];
        contentBytes[0] = 0x81;
        contentBytes[1] = 126;
        contentBytes[2] = (byte)(temp.Length & 0xFF);
        contentBytes[3] = (byte)(temp.Length >> 8 & 0xFF);
        Array.Copy(temp, 0, contentBytes, 4, temp.Length);
    }
    else
    {
        // 暂不处理超长内容
    }

    return contentBytes;
}
```

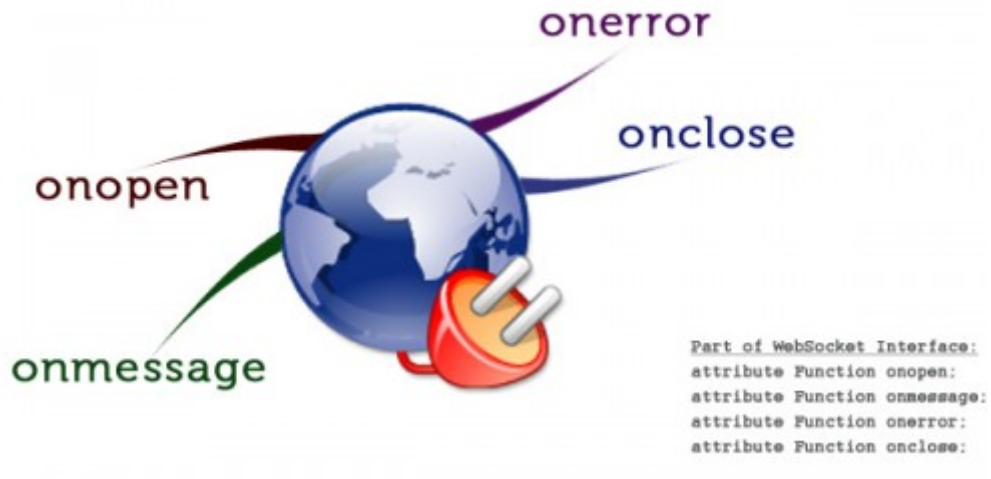
4.3 Closing Handshake

相对于Opening Handshake，Closing Handshake则简单得多，主动关闭的一方向另一方发送一个关闭类型的数据包，对方收到此数据包之后，再回复一个相同类型的数据包，关闭完成。

关闭类型数据包遵守封包协议，Opcode为0x8，Payload data可以用于携带关闭原因或消息。

4.4 websocket的事件响应

以上的Opening Handshake、Data Framing、Closing Handshake三个步骤其实分别对应了websocket的三个事件：



- onopen 当接口打开时响应
- onmessage 当收到信息时响应
- onclose 当接口关闭时响应

任何程序语言的websocket api都至少要提供上面三个事件的api接口， 有的可能还提供的有onerror事件的处理机制。

websocket 在任何时候都会处于下面4种状态中的其中一种：

- CONNECTING (0)：表示还没建立连接；
- OPEN (1)： 已经建立连接，可以进行通讯；
- CLOSING (2)：通过关闭握手，正在关闭连接；
- CLOSED (3)：连接已经关闭或无法打开；

5. 如何使用websocket

客户端
在支持WebSocket的浏览器中，在创建socket之后。可以通过onopen，onmessage，onclose即onerror四个事

件实现对socket进行响应
一个简单是示例：

```
var ws = new WebSocket("ws://localhost:8080");
ws.onopen = function()
{
  console.log("open");
  ws.send("hello");
};
ws.onmessage = function(evt)  {  console.log(evt.data); };
ws.onclose   = function(evt)  {  console.log("WebSocketClosed!"); };
ws.onerror   = function(evt)  {  console.log("WebSocketError!"); };
```

首先申请一个WebSocket对象，参数是需要连接的服务器端的地址，同http协议使用http://开头一样，WebSocket协议的URL使用ws://开头，另外安全的WebSocket协议使用wss://开头。

client先发起握手请求：

```
GET /echobot HTTP/1.1
Host: 192.168.14.215:9000
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://192.168.14.215
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/45.0.2454.101 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8
Sec-WebSocket-Key: mh3xLXeRuIWNPwq7ATG9jA==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

服务端响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: SIEylb7zRYJAEgiqJXaOW3V+ZWQ=
```

交互数据：

```
ws.send("hello");    # 用于将消息发送到服务端
ws.recv($buffer);    # 用于接收服务端的消息
```

6. 自己如何实现websocket server和client

我分别用C++、PHP、Python语言实现了websocket server和client， 只支持基本功能，也是为了加深理解websocket协议内容。

所有源代码放在github上，点此查看：websocket server & client 分别用C++/PHP/Python实现， 如何使用、测试及集成自己的逻辑也在文档中进行了说明，这里不再列出了。

7. reference

Ajax、Comet与Websocket

WebSocket使用教程

分析HTML5中WebSocket的原理

WebScoket 规范 + WebSocket 协议

websocket规范 RFC6455 中文版