

# 一.传统的实现即时通信的方式

## ajax轮询

ajax轮询的原理非常简单，让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息。

场景再现：

客户端：啦啦啦，有没有新信息(Request)

服务端：没有 (Response)

客户端：啦啦啦，有没有新信息(Request)

服务端：没有。。 (Response)

客户端：啦啦啦，有没有新信息(Request)

服务端：你好烦啊，没有啊。。 (Response)

客户端：啦啦啦，有没有新消息 (Request)

服务端：好啦好啦，有啦给你。 (Response)

客户端：啦啦啦，有没有新消息 (Request)

服务端：。。。。没。。。。没。。没有 (Response) — loop

## long poll

**long poll** 其实原理跟 **ajax轮询** 差不多，都是采用轮询的方式，不过采取的是阻塞模型（一直打电话，没收到就不挂电话），也就是说，客户端发起连接后，如果没消息，就一直不返回Response给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。

场景再现：

客户端：啦啦啦，有没有新信息，没有的话就等有了才返回给我吧 (Request)

服务端：额。。等待到有消息的时候。。来 给你 (Response)

客户端：啦啦啦，有没有新信息，没有的话就等有了才返回给我吧 (Request) -loop

从上面可以看出其实这两种方式，都是在不断地建立HTTP连接，然后等待服务端处理，可以体现HTTP协议的另外一个特点，被动性。

何为被动性呢，其实就是，服务端不能主动联系客户端，只能有客户端发起。

小姐：

**ajax轮询** 需要服务器有很快的处理速度和资源。（速度）

**long poll** 需要有很高的并发，也就是说同时接待客户的能力。（场地大小）

## 长连接

在页面里嵌入一个隐藏iframe，将这个隐藏iframe的src属性设为对一个长连接的请求或是采用xhr请求，服务器端就能源源不断地往客户端输入数据。

优点：消息即时到达，不发无用请求；管理起来也相对方便。

缺点：服务器维护一个长连接会增加开销，当客户端越来越多的时候，server压力大！

实例：Gmail聊天

### (1).基于http协议的长连接

在HTTP1.0和HTTP1.1协议中都有对长连接的支持。其中HTTP1.0需要在request中增加"Connection: keep-alive" header才能够支持, 而HTTP1.1默认支持.

http1.0请求与服务端的交互过程:

- a)客户端发出带有包含一个header: "Connection: keep-alive"的请求
- b)服务端接收到这个请求后,根据http1.0和"Connection: keep-alive"判断出这是一个长连接,就会在response的header中也增加"Connection: keep-alive",同是不会关闭已建立的tcp连接.
- c)客户端收到服务端的response后,发现其中包含"Connection: keep-alive", 就认为是一个长连接, 不关闭这个连接. 并用该连接再发送request.转到a)

## (2).http1.1请求与服务端的交互过程:

- a)客户端发出http1.1的请求
- b)服务端收到http1.1后就认为这是一个长连接,会在返回的response设置Connection: keep-alive,同是不会关闭已建立的连接.
- c)客户端收到服务端的response后,发现其中包含"Connection: keep-alive", 就认为是一个长连接, 不关闭这个连接. 并用该连接再发送request.转到a)

基于http协议的长连接减少了请求,减少了建立连接的时间,但是每次交互都是由客户端发起的,客户端发送消息,服务端才能返回客户端消息.因为客户端也不知道服务端什么时候会把结果准备好,所以客户端的很多请求是多余的,仅是维持一个心跳,浪费了带宽.

## Flash Socket

在页面中内嵌入一个使用了Socket类的 Flash 程序JavaScript通过调用此Flash程序提供的Socket接口与服务器端的Socket接口进行通信, JavaScript在收到服务器端传送的信息后控制页面的显示.

优点: 实现真正的即时通信, 而不是伪即时.

缺点: 客户端必须安装Flash插件, 移动端支持不好, IOS系统中没有flash的存在; 非HTTP协议, 无法自动穿越防火墙.

实例: 网络互动游戏.

## 二.websocket的方式实现服务端消息推送

1.什么是socket? 什么是websocket? 两者有什么区别? websocket是仅仅将socket的概念移植到浏览器中的实现吗?

我们知道, 在网络中的两个应用程序(进程)需要全双工相互通信(全双工即双方可同时向对方发送消息), 需要用到的就是socket, 它能够提供端对端通信, 对于程序员来讲, 他只需要在某个应用程序的一端(暂且称之为客户端)创建一个socket实例并且提供它所连接一端(暂且称之为服务端)的IP地址和端口, 而另外一端(服务端)创建另一个socket并绑定本地端口进行监听, 然后客户端进行连接服务端, 服务端接受连接之后双方建立了一个端对端的TCP连接, 在该连接上就可以双向通讯了, 而且一旦建立这个连接之后, 通信双方就没有客户端服务端之分了, 提供的就是端对端通信了. 我们可以采取这种方式构建一个桌面版的im程序, 让不同主机上的用户发送消息. 从本质上来说,

socket并不是一个新的协议，它只是为了便于程序员进行网络编程而对tcp/ip协议族通信机制的一种封装。

socket传送门：<http://blog.csdn.net/luokehua789789/article/details/54378264>

websocket是html5规范中的一个部分，它借鉴了socket这种思想，为web应用程序客户端和服务端之间（注意是客户端服务端）提供了一种全双工通信机制。同时，它又是一种新的应用层协议，[websocket协议](#)是为了提供web应用程序和服务端全双工通信而专门制定的一种应用层协议，通常它表示为：

ws://echo.websocket.org/?encoding=text HTTP/1.1，可以看到除了前面的协议名和http不同之外，它的表示地址就是传统的url地址。

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



**websocket具有以下几个方面的优势：**

- (1) 建立在 TCP 协议之上，服务器端的实现比较容易。**
- (2) 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。**
- (3) 数据格式比较轻量，性能开销小，通信高效。**
- (4) 可以发送文本，也可以发送二进制数据。**
- (5) 没有同源限制，客户端可以与任意服务器通信。**
- (6) 协议标识符是`ws`（如果加密，则为`wss`），服务器网址就是 URL。**

## **2.websocket的通信原理和机制**

既然是基于浏览器端的web技术，那么它的通信肯定少不了http,websocket本身虽然也是一种新的应用层协议，但是它也不能够脱离http而单独存在。具体来讲，我们在客户端构建一个websocket实例，并且为它绑定一个需要连接到的服务器地址，当客户端连接服务端的时候，会向服务端发送一个类似下面的http报文

```
1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
6 Sec-WebSocket-Protocol: chat, superchat
7 Sec-WebSocket-Version: 13
8 Origin:
  http://example.com
Origin:
```

```
1 Upgrade: websocket
2 Connection: Upgrade
```

**这个就是Websocket的核心了，告诉Apache、Nginx等服务器：发起的是websocket协议。**

```
1 Sec-WebSocket-Key: x3JJHmbDL1EzLkh9GBhXDw==
2 Sec-WebSocket-Protocol: chat, superchat
3 Sec-WebSocket-Version: 13
```

**首先，Sec-WebSocket-Key 是一个Base64 encode的值，这个是浏览器随机生成的，告诉服务器：泥煤，不要忽悠窝，我要验证尼是不是真的是Websocket助理。**

**然后，Sec\_WebSocket-Protocol 是一个用户定义的字符串，用来区分同URL下，不同的服务所需要的协议。简单理解：今晚我要服务A，别搞错啦~**

**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。。不过现在还好，已经定下来啦~大家都使用的一个东西~ 脱水：服务员，我要的是13岁的噢→\_→ 然后服务器会返回下列东西，表示已经接受到请求， 成功建立Websocket啦！**

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGwk=
5 Sec-WebSocket-Protocol: chat
```

依然是固定的，告诉客户端即将升级的是Websocket协议，而不是mozillasocket, lurnarsocket或者shitsocket。

然后，Sec-WebSocket-Accept 这个则是经过服务器确认，并且加密过后的 Sec-WebSocket-Key。

服务器：好啦好啦，知道啦，给你看我的ID CARD来证明行了吧。。

后面的，Sec-WebSocket-Protocol 则是表示最终使用的协议。

返回的状态码为101，表示同意客户端协议转换请求，并将它转换为websocket协议。以上过程都是利用http通信完成的，称之为websocket协议握手(websocket Protocol handshake)，进过这握手之后，客户端和服务端就建立了websocket连接，以后的通信走的都是websocket协议了。所以总结为websocket握手需要借助于http协议，建立连接后通信过程使用websocket协议。同时需要了解的是，该websocket连接还是基于我们刚才发起http连接的那个TCP连接。一旦建立连接之后，我们就可以进行数据传输了，websocket提供两种数据传输：文本数据和二进制数据。

至此，HTTP已经完成它所有工作了，接下来就是完全按照Websocket协议进行了。

基于以上分析，我们可以看到，websocket能够提供低延迟，高性能的客户端与服务端的双向数据通信。它颠覆了之前web开发的请求处理响应模式，并且提供了一种真正意义上的客户端请求，服务器推送数据的模式，特别适合实时数据交互应用开发。





对比前面的http的客户端服务器的交互图可以发现WebSocket方式减少了很多TCP打开和关闭连接的操作，WebSocket的资源利用率高。

### 3.websocket的创建和常用的属性方法

以下 API 用于创建 WebSocket 对象。

```
1 var Socket = new WebSocket(url, [protocol] );
```

以上代码中的第一个参数 url, 指定连接的 URL。第二个参数 protocol 是可选的，指定了可接受的子协议。

### WebSocket 属性

以下是 WebSocket 对象的属性。假定我们使用了以上代码创建了 Socket 对象：

属性	描述
Socket.readyState	只读属性 <b>readyState</b> 表示连接状态，可以是以下值： <ul style="list-style-type: none"><li>• 0 - 表示连接尚未建立。</li><li>• 1 - 表示连接已建立，可以进行通信。</li><li>• 2 - 表示连接正在进行关闭。</li><li>• 3 - 表示连接已经关闭或者连接不能打开。</li></ul>
Socket.bufferedAmount	只读属性 <b>bufferedAmount</b> 已被 send() 放入正在队列中等待传输，但是还没有发出的 UTF-8 文本字节数。

- CONNECTING：值为0，表示正在连接。
- OPEN：值为1，表示连接成功，可以通信了。
- CLOSING：值为2，表示连接正在关闭。
- CLOSED：值为3，表示连接已经关闭，或者打开连接失败。

```
1 var websocket = new WebSocket(url);
2 if(websocket.readyState == websocket.CONNECTING){
3     console.log('连接正在打开');
4 }
5
```

```
6  websocket.onopen = function () {
7      websocket.send(consumerId);
8      //可以看到 "连接正在打开"并没有被打印，说明open对应的就是OPEN状态；
9      if(websocket.readyState == websocket.CONNECTING){
10         console.log('连接正在打开1');
11     }
12     if(websocket.readyState == websocket.OPEN){
13         console.log('连接已打开');
14     }
15     sendMsg();
16     window.weui.alert('已经建立连接');
17 };
18
19
20 //连接关闭时触发
21 websocket.onclose = function () {
22     if(websocket.readyState == websocket.CLOSED){
23         console.log('连接已关闭')
24     }
25     window.weui.alert('连接已断开');
26 };
27
28 //连接
29 websocket.onerror = function () {
30     window.weui.alert('连接错误，请稍后再试');
31 };
```



可以看到，当onopen触发时，对应的就是readyState的OPEN状态，不包含OPENING；onclose触发时，对应的就是CLOSED状态，不包含CLOSING状态。

## WebSocket 事件

以下是 WebSocket 对象的相关事件。假定我们使用了以上代码创建了 Socket 对象：

事件	事件处理程序	描述
open	Socket.onopen	连接建立时触发
message	Socket.onmessage	客户端接收服务端数据时触发
error	Socket.onerror	通信发生错误时触发
close	Socket.onclose	连接关闭时触发

## WebSocket 方法

以下是 WebSocket 对象的相关方法。假定我们使用了以上代码创建了 Socket 对象：

方法	描述
Socket.send()	使用连接发送数据
Socket.close()	关闭连接

```
1  <!DOCTYPE HTML>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <title>菜鸟教程(runoob.com)</title>
6
7      <script type="text/javascript">
8          function WebSocketTest()
9          {
10              if ("WebSocket" in window)
11              {
12                  alert("您的浏览器支持 WebSocket!");
13              }
14          }
15      </script>
16  </head>
17  <body>
18      <p>WebSocket 教程: 使用 WebSocket 实现实时通信。</p>
19  </body>
20 </html>
```

```
14         // 打开一个 web socket
15         var ws = new WebSocket("ws://localhost:9998/echo");
16
17         ws.onopen = function()
18         {
19             // Web Socket 已连接上，使用 send() 方法发送数据
20             ws.send("发送数据");
21             alert("数据发送中...");
22         };
23
24         ws.onmessage = function (evt)
25         {
26             var received_msg = evt.data;
27             alert("数据已接收...");
28         };
29
30         ws.onclose = function()
31         {
32             // 关闭 websocket
33             alert("连接已关闭...");
34         };
35     }
36
37     else
38     {
39         // 浏览器不支持 WebSocket
40         alert("您的浏览器不支持 WebSocket!");
41     }
42 }
43 </script>
44
45 </head>
46 <body>
47
48     <div id="sse">
49         <a href="javascript:WebSocketTest()">运行 WebSocket</a>
50     </div>
51
52 </body>
53 </html>
```

## 用websocket发送接受二进制数据

WebSocket可以通过ArrayBuffer，发送或接收二进制数据。

```
1 var socket = new WebSocket('ws://127.0.0.1:8081');
2 socket.binaryType = 'arraybuffer';
3
4 // Wait until socket is open
5 socket.addEventListener('open', function (event) {
6     // Send binary data
7     var typedArray = new Uint8Array(4);
8     socket.send(typedArray.buffer);
9 });
10
11 // Receive binary data
12 socket.addEventListener('message', function (event) {
13     var arrayBuffer = event.data;
14     // ...
15 });
```