WebSocket实战

by Ji Yunpeng / 前端开发,博客 / 2012-07-06

前言

互联网发展到现在，早已超越了原始的初衷，人类从来没有像现在这样依赖过他；也正是这种依赖，促进了互联网技术的飞速发展。而终端设备的创新与发展，更加速了互联网的进化；

HTTP/1.1规范发布于1999年，同年12月24日，HTML4.01规范发布；尽管已到2012年，但HTML4.01仍是主流；虽然HTML5的草案已出现了好几个年头，但转正日期，遥遥无期，少则三五年，多则数十年；而HTML5的客户代理(对于一般用户而言，就是浏览器)，则已百家争鸣，星星向荣；再加上移动终端的飞速发展，在大多数情况下，我们都可以保证拥有一个HTML5的运行环境，所以，我们来分享一下HTML5中的WebSocket协议；

本文包含以下六个方面：
1. WebSocket的前世今生
2. WebSocket是什么
3. 为什么使用WebSocket
4. 搭建WebSocket服务器
5. WebSocket API
6. 实例解析

以上六点分为两大块，前3点侧重理论，主要让大家明白WebSocket是什么，而后3点则结合代码实战，加深对WebSocket的认知。

一、WebSocket的前世今生

Web 应用的信息交互过程通常是客户端通过浏览器发出一个请求，服务器端接收和审核完请求后进行处理并返回结果给客户端，然后客户端浏览器将信息呈现出来，这种机制对于信息变化不是特别频繁的应用尚能相安无事，但是对于那些实时要求比较高的应用来说就显得捉襟见肘了。我们需要一种高效节能的双向通信机制来保证数据的实时传输。有web TCP之称的WebSocket应运而生，给开发人员提供了一把强有力的武器来解决疑难杂症。
(PS：其实，在早期的HTML5规范中，并没有包含WebSocket的定义，一些早期的HTML5书籍中，完全没有WebSocket的介绍。直到后来，才加入到当前的草案中。)

二、WebSocket是什么?

其实，从背景介绍中，我们大致的可以猜出，WebSocket是干什么用的。前面我们提到，WebSocket有web TCP之称，既然是TCP，肯定是用来做通信的，但是它又有不同的地方，WebSocket作为HTML5中新增的一种通信协议，由通信协议和编程API组成，它能够在浏览器和服务器之间建立双向连接，以基于事件的方式，赋予浏览器原生的实时通信能力，来扩展我们的web应用，增加用户体验，提升应用的性能。何谓双向？服务器端和客户端可以同时发送并响应请求，而不再像HTTP的请求和响应。

三、为什么使用WebSocket

在WebSocket出现之前，我们有一些其它的实时通讯方案，比较常用的有轮询，长轮询，流，还有基于Flash的交换数据的方式，接下来，我们一一分析一下，各种通信方式的特点。

① 轮询
这是最早的一种实现实时web应用的方案；原理比较简单易懂，就是客户端以一定的时间间隔向服务器发送请求，以频繁请求的方式来保持客户端和服务器端的数据同步。但是问题也很明显：当客户端以固定频率向服务器端发送请求

之前的文章
- 一个页面重构工程师眼中的“用户体验”
- 发现用户言行不一的小技巧之感性篇
- iOS Web App初步
- 使用Node.JS构建Long Polling应用程序
- 各浏览器对 window.execScript 方法的支持不同

之后的文章
- 使用 node.js 开发前端打包程序
- 可用性测试
- 论“美”对用户体验的意义
- 前端开发之面向对象
- 指尖上的正则表达式 – 入门篇

友情链接
- 奇虎75team
- 阿里巴巴（中文站）UED

时，服务器端的数据可能并没有更新，这样会带来很多无谓的请求，浪费带宽，效率低下。

② 长轮询

长轮询是对定时轮询的改进和提高，目地是为了降低无效的网络传输。当服务器端没有数据更新的时候，连接会保持一段时间周期直到数据或状态改变或者时间过期，通过这种机制来减少无效的客户端和服务器间的交互。当然，如果服务端的数据变更非常频繁的话，这种机制和定时轮询比较起来没有本质上的性能的提高。

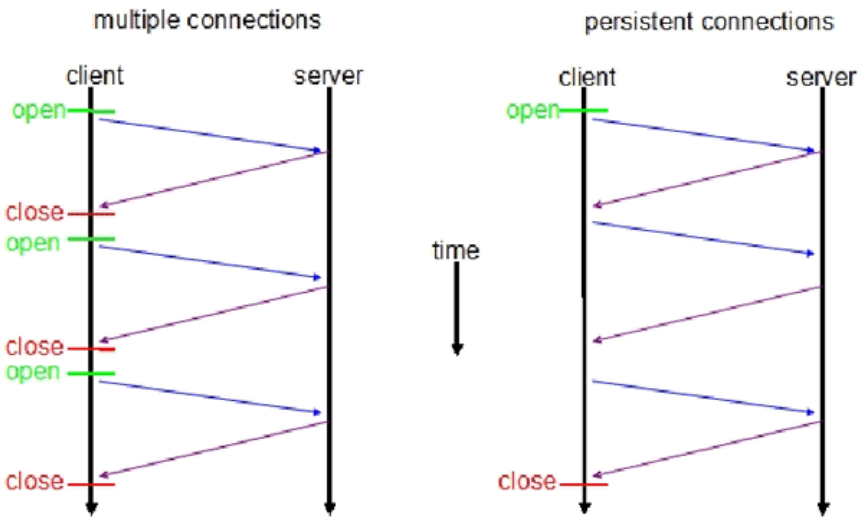
③ 流

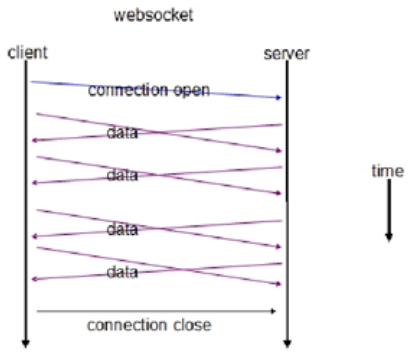
长轮询是对定时轮询的改进和提高，目地是为了降低无效的网络传输。当服务器端没有数据更新的时候，连接会保持一段时间周期直到数据或状态改变或者时间过期，通过这种机制来减少无效的客户端和服务器间的交互。当然，如果服务端的数据变更非常频繁的话，这种机制和定时轮询比较起来没有本质上的性能的提高。

④ 基于Flash的实时通讯方式

Flash有自己的socket实现，这为实时通信提供了可能。我们可以利用Flash完成数据交换，再利用Flash暴露出相应的接口，方便JavaScript调用，来达到实时传输数据的目的。这种方式比前面三种方式都要高效，而且应用场景比较广泛；因为flash本身的安装率很高；但是在当前的互联网环境下，移动终端对flash的支持并不好，以IOS为主的系统中根本没有flash的存在，而在android阵营中，虽然有flash的支持，但实际的使用效果差强人意，即使是配置较高的移动设备，也很难让人满意。就在前几天(2012年6月底)，Adobe官方宣布，不在支持android4.1以后的系统，这基本上宣告了flash在移动终端上的死亡。

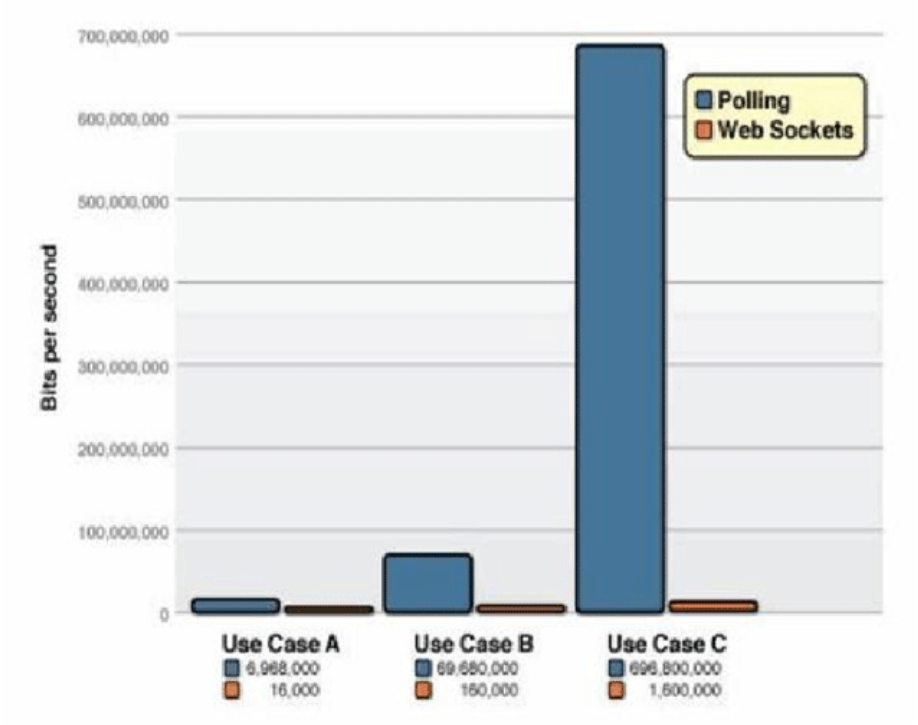
下面是轮询和长轮询的信息流转图：





对比完四种不同的实时通信方式，不难发现，除了基于flash的方案外，其它三种方式都是用AJAX方式来模拟实时的效果，每次客户端和服务器端交互时，都是一次完整的HTTP请求和应答的过程，而每一次的HTTP请求和应答都带有完整的HTTP头信息，这就增加每次的数据传输量，而且这些方案中客户端和服务端的编程实现比较复杂。

接下来，我们再来看一下WebSocket，为什么要使用它呢？高效节能，简单易用。
下图是来自websocket.org的测试结果：



在流量和负载增大的情况下，WebSocket 方案相比传统的 Ajax 轮询方案有极大的性能优势；而在开发方面，也十分简单，我们只需要实例化WebSocket，创建连接，查看是否连接成功，然后就可以发送和相应消息了。我们会在后面的实例中去详细的说明API。

四、搭建WebSocket服务器

其实，在服务器的选择上很广，基本上，主流语言都有WebSocket的服务器端实现，而我们作为前端开发工程师，当然要选择现在比较火热的NodeJS作为我们的服务器端环境了。

NodeJS本身并没有原生的WebSocket支持，但是有第三方的实现(大家要是有兴趣的话，完全可以参考WebSocket协议来做自己的实现)，我们选择了“ws”作为我们的服务器端实现。

由于本文的重点是讲解WebSocket，所以，对于NodeJS不做过多的介绍，不太熟悉的朋友可以去参考NodeJS入门指南(http://www.nodebeginner.org/index-zh-cn.html)。

安装好NodeJS之后，我们需要安装“ws”，也就是我们的WebSocket实现，安装方法很简单，在终端或者命令行中输入：

```
1 npm install ws
```

，等待安装完成就可以了。

接下来，我们需要启动我们的WebSocket服务。首先，我们需要构建自己的HTTP服务器，在NodeJS中构建一个简单的HTTP服务器很简单，so easy。代码如下：

```
1 var app = http.createServer( onRequest ).listen( 8888 );
```

onRequest()作为回调函数，它的作用是处理请求，然后做出响应，实际上就是根据接收的URL，在服务器上查找相应的资源，最终返回给浏览器。
在构建了HTTP服务器后，我们需要启动WebSocket服务，代码如下：

```
1 var WebSocketServer = require('ws').Server;
2 var wss = new WebSocketServer( { server : app } );
```

从代码中可以看出，在初始化WebSocket服务时，把我们刚才构建好的HTTP实例传递进去就好。到这里，我们的服务端代码差不多也就编写完成了。怎么样？很简单吧。

五、WebSocket API

上面我们介绍了WebSocket服务端的知识，接下来，我们需要编写客户端代码了。在前面我们说过，客户端的API也是一如既往的简单：

ready state	
CONNECTING	0
OPEN	1
CLOSING	2
CLOSED	3

readonly attribute
readyState
bufferedAmount

见上图：ready state中定义的是socket的状态，分为connection、open、closing和closed四种状态，从字面上就可以区分出它们所代表的状态。

networking
onopen
onerror
onclose

readonly attribute
extensions
protocol

上图描述的是WebSocket的事件，分为onopen、onerror和onclose；

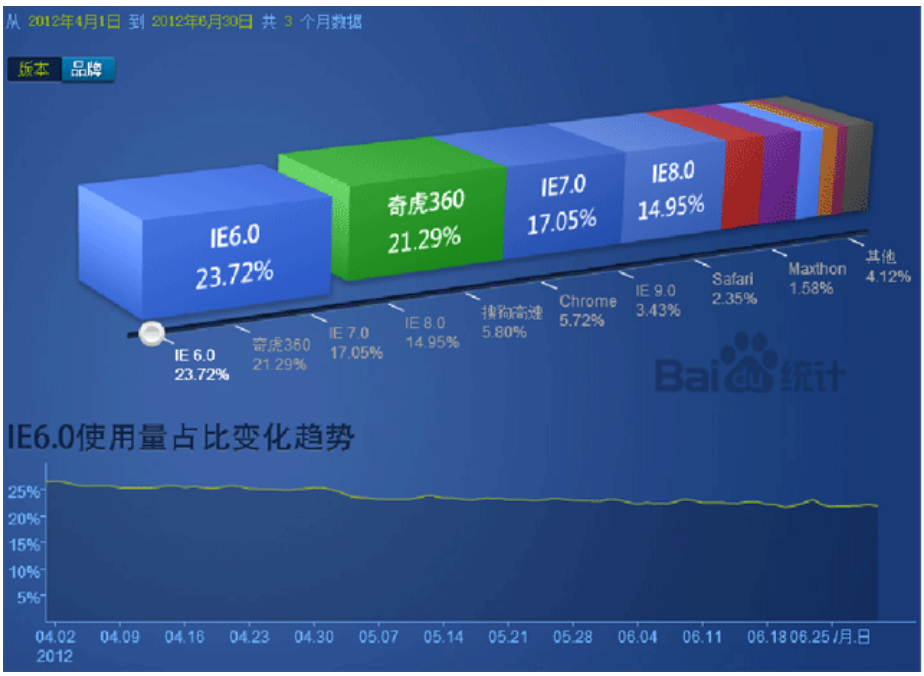
messaging
onmessge
send(DOMString data)
send(ArrayBufferView data)
send(Blob data)

上图为消息的定义，主要是接收和发送消息。注意：可以发送二进制的数据。

以上个图的具体的含义就不再一一赘述，详细描述请参考：
http://www.w3.org/TR/2012/WD-websockets-20120524/

PS：由于WebSocket API（截止到2012年7月）还是草案，API文档和上文所描述的会有所不同，请以官方文档为主，这也是我为什么不详细描述API中各个属性的原因。

另外一点需要提醒大家的是：在前端开发中，浏览器兼容是必不可少的，而WebSocket在主浏览器中的兼容还是不错的，火狐和Chrome不用说，最新版的支持非常不错，而且支持二进制数据的发送和接收。但是IE9并不支持，对于国内的大多数应用场景，WebSocket无法大规模使用。



截图来自（http://tongji.baidu.com/data/browser），之所以选择百度的统计数据，是因为更加符合国内的实际情况。图中所展示的是2012年4月1日到2012年6月30日之间的统计数据，从图中不难看出IE6.0、奇虎360、IE7.0和IE8.0加起来一共占据了77%的市场，FireFox属于其他，chrome只有5.72%的份额，再一次告诉我们，我们的主战场依然是IE系。

既然是IE系，那么对于WebSocket在实际app中的应用就基本不可能了。但我们完全可以在chrome、FireFox、以及移动版的IOS浏览器中使用它。

六、实例解析

搭建好了服务端，熟悉了API，接下来，我们要开始构建我们的应用了。鉴于WebSocket自身的特点，我们的第一个demo选择了比较常见的聊天程序，我们暂且取名为chat。

说到聊天，大家最先想到的肯定是QQ，没错，我们所实现的应用和QQ类似，而且还是基于web的。因为是demo，我们的功能比较简陋，仅实现了最简单的会话功能。就是启动WebSocket服务器后，客户端发起连接，连接成功后，任意客户端发送消息，都会被服务器广播给所有已连接的客户端，包括自己。

既然需要客户端，我们需要构建一个简单的html页面，页面中样式和元素，大家可以自由发挥，只要能够输入消息，有发送按钮，最后有一个展示消息的区域即可。具体的样子大家可以看附件中的demo。

写玩HTML页面之后，我们需要添加客户端脚本，也就是和WebSocket相关的代码；前面我们说过，WebSocket的API本身很简单，所以，我们的客户端代码也很直接，如下：

```
var wsServer = 'ws://localhost:8888/';
var websocket = new WebSocket(wsServer);
websocket.binaryType = "arraybuffer";
websocket.onopen = onOpen;
websocket.onclose = onClose;
websocket.onmessage = onMessage;
websocket.onerror = onError;
```

首先，我们需要指定WebSocket的服务地址，也就是var wsServer = ‘ws://localhost:8888/’；

然后，我们实例化WebSocket，new WebSocket(wsServer)，
剩下的就是指定相应的回调函数了，分别是onOpen，onClose，onMessage和onError，对于咱们的实验应用来说，onopen、onclose、onerror甚至可以不管，咱们重点关注一下onmessage。

onmessage()这个回调函数会在客户端收到消息时触发，也就是说，只要服务器端发送了消息，我们就可以通过onmessage拿到发送的数据，既然拿到了数据，接下去该怎么玩，就随便我们了。请看下面的伪代码：

```
function onMessage(evt) {
    var json = JSON.parse(evt.data);
    commands[json.event](json.data);
}
```

因为onmessage只接收字符串和二进制类型的数据，如果需要发送json格式的数据，就需要我们转换一下格式，把字符串转换成JSON格式。只要是支持WebSocket，肯定原生支持window.JSON，所以，我们可以直接使用JSON.parse()和JSON.stringify()来进行转换。
转换完成后，我们就得到了我们想要的数据了，接下来所做的工作就是将消息显示出来。实际上就是

```
Elements.innerHTML += data + '</br>';
```

上面展现了客户端的代码，服务器端的代码相对要简单一些，因为我们的服务器端使用的是第三方实现，我们只需要做一些初始化工作，然后在接收到消息时，将消息广播出去即可，下面是具体的代码：

```
var app = http.createServer( onRequest ).listen( 8888 );
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer( { server : app } );
wss.on('connection', function( ws ) {
    console.log('connection successful!');
    ws.on('message', function( data, flags ) {
        console.log(data);
        //do something here
    });
    ws.on('close', function() {
        console.log('stopping client');
    });
});
```

我们可以通过wss.clients获得当前已连接的所有客户端，然后遍历，得到实例，调用send()方法发送数据；

```
var clients = wss.clients, len = clients.length, i = 0;
        for( ; i < len; i = i + 1 ){
            clients[i].send( msg );
        }
```

说到这里，一个双向通信的实例基本完成，当然，上面都是伪代码，完整的demo请查看附件。

除了常见的聊天程序以外，大家完全可以发挥创意，构建一些“好玩”的应用；
接下来，分享另外一个应用，“你画我猜”这个应用，很多人都接触过，大致上是：某个人在屏幕上画一些图形，这些图片会实时展示在其它人的屏幕上，然后来猜画的是什么。

利用WebSocket和canvas，我们可以很轻松的构建类似的应用。当然，我们这里只是demo，并没有达到产品级的高度，这里只是为大家提供思路；
首先，我们再次明确一下，WebSocket赋予了我们在浏览器端和服务器进行双向通信的能力，这样，我们可以实时的将数据发送给服务器，然后再广播给所有的客户端。这和聊天程序的思路是一致的。

接下来，服务器端的代码不用做任何修改，在html页面中准备一个canvas，作为我们的画布。如何在canvas上用鼠标画图形呢？我们需要监听mousedown、mousemove和mouseup三个鼠标事件。说到这里，大家应该知道怎么做了。没错，就是在按下鼠标的时候，记录当前的坐标，移动鼠标的时候，把坐标发送给服务器，再由服务器把坐标数据广播给所有的客户端，这样就可以在所有的客户端上同步绘画了；最后，mouseup的时候，做一些清理工作就ok了。下面是一些伪代码：

```
var WhiteBoard = function( socket, canvasId ){
                var lastPoint = null,
                    mouseDown = false,
                    canvas = getById(canvasId),
                    ctx = canvas.getContext('2d');

                var handleMouseDown = function(event) {
                    mouseDown = true;
                    lastPoint = resolveMousePosition.bind( canvas, event )();
                };

                var handleMouseUp = function(event) {
                    mouseDown = false;
                    lastPoint = null;
                };

                var handleMouseMove = function(event) {
                    if (!mouseDown) { return; }
                    var currentPoint = resolveMousePosition.bind( canvas, event )();
                    socket.send(JSON.stringify({
                        event: 'draw',
                        data: {
                            points: [
                                lastPoint.x,
                                lastPoint.y,
                                currentPoint.x,
                                currentPoint.y
                            ]
                        }
                    }));

                    lastPoint = currentPoint;
                };

                var init = function(){
                    addEvent( canvas, 'mousedown', handleMouseDown );
                    addEvent( canvas, 'mouseup', handleMouseUp );
```