HTML5 中新的 WEB 交互方式

--msn:xiaohuiq@hotmail.com

Overview

首先设想一下这样的需求：

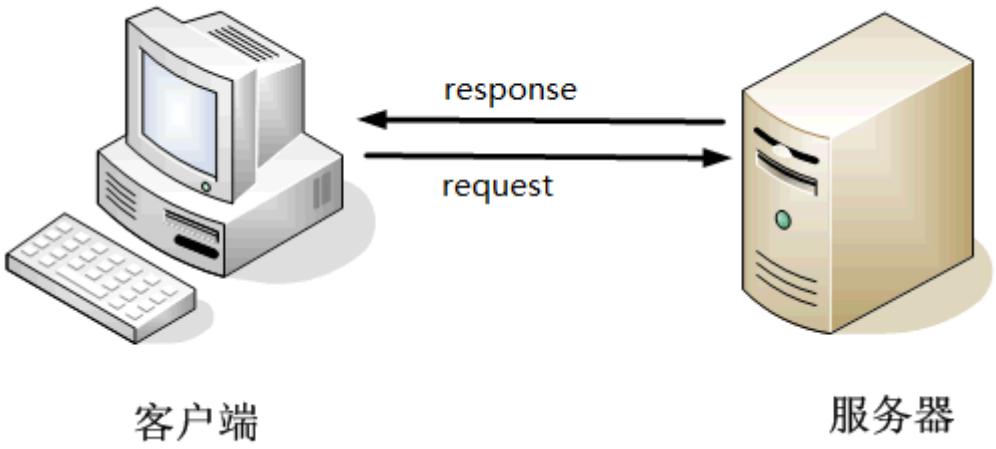
使用 WEB，来构建实时的展示页面来展示股票信息、火车票余票、医疗设备读取的信息，或是即时聊天等等

你想到了什么实现方式？轮询？长轮询？flashsocket？...好吧，不管是什么方式，在使用过程中你感觉这种方式如何？是不是有些缺点或是值得改进的地方？

WebSocket is coming!

HTML5 中的 WebSocket 主要就是为了解决这种实时交互而设计的！这是一种全双工的通信方式，实现了服务端完美的 PUSH！

当然，在介绍这个激动人心的技术之前还是要先简单说一下传统的 WEB 交互方式，对各个技术都了解了才懂得取舍



WEB 开发人员看到这个图总是会感觉很亲切，接下来我们就看看在这种请求响应模型中如何做到即时交互

Constantly Refresh

简述

HTTP 协议原本是设计用于传输简单的文档和文件，而非实时的交互。

根据 HTTP 协议，一个客户端如浏览器，向服务器打开一个连接，发出请求，等待回应，之后关闭连接。如果客户端需要更多数据，则需要打开一个新连 接，以此循环往复。如果服务器有了新的信息，它必须等待客户端发出请求而不是立即发送消息。

举个例子：我（客户端）要从你（服务器）那运货，先修条路，然后派个使者去通知你要什么货，然后你把货给我送过来，最后把修好的路作废，再想运送别的货物，对不起，请重新修路

那么要看到页面中要展示信息的最新情况，应该怎么办？不断刷新！是的，以前就是这么做的

评价

这种方式现在已经被完全淘汰，发送了很多不必要的请求，浪费大量带宽，页面不断刷新，用户体验差，而且做不到真正的实时，服务端有了新数据也不能立马推送给客户端，使得秒级的实时信息交互难以实现

Polling

简介

这可以算是第一种风格的 comet
每隔一固定时间发送一个 Ajax 请求拉取数据，根据服务端返回的数据通过 DOM 操作做一些展现

评价

比不断刷新好一些，浏览器不用一闪一闪的重新加载了，而且只传送感兴趣的那一小部分数据，占用带宽变小。但是，客户端并不知道服务端什么时候准备好了自己感兴趣的数据，无法很好的设置轮询时间，只能根据经验设置一个固定的时间，这样就会发送很多不必要的请求

Long-Polling

简介

这可以算作第二种风格的 comet

客户端发送一个 request 给服务端，服务端会在一个设定的时间段内保持这个 request 为打开状态，如果在这段时间内， 服务端程序收到一个 notification，就会把携带了最新消息的 response 回送给客户端，如果在这个设定的时间内一直没有收到新消息，没有收到 notification，服务端程序在时间到的时候同样会发送一个 response 给客户端

评价

长 轮询相对于一般轮询的优点在于，数据一旦可用，便立即从服务器发送到客户机。请求可能等待较长的时间，期间没有任何数据返回，但是一旦有了新的数据，它将 立即被发送到客户机。因此没有延时
这种情况在服务端消息比较少的情况下表现还不错，但是服务端有大量消息要推送的时候，Long-Polling 与 Polling 相比，实际并没有什么本质的提高，它还是需要等待客户端的请求，然后才能发消息，而不是自动推送！

Streaming

简介

这是第三种风格的 comet
按照这种风格，服务器将数据推回客户机，但是不关闭连接。连接将一直保持开启，直到过期，并导致重新发出请求。XMLHttpRequest 规范表明，可以检查 readyState 的值是否为 3 或 Receiving（而不是 4 或 Loaded），并获取正从服务器“流出”的数据

评价

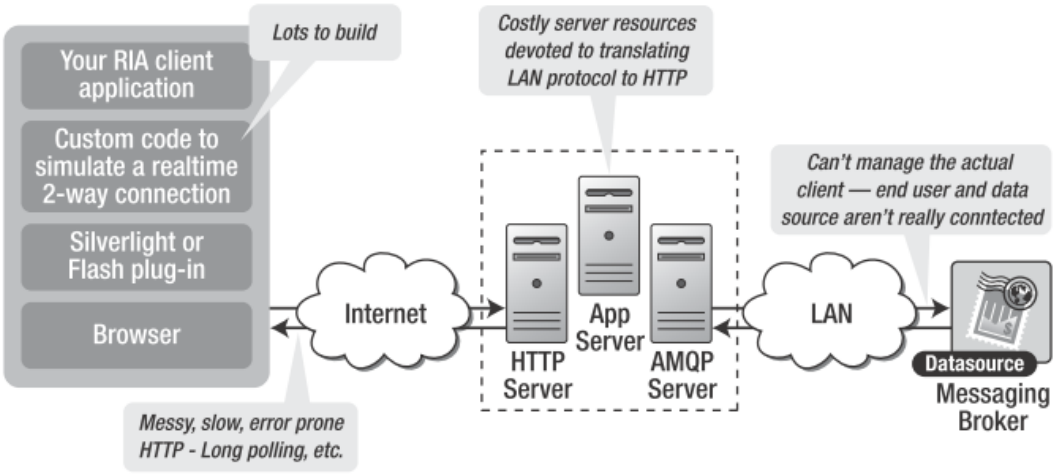
和长轮询一样，这种方式也没有延时。当服务器上的数据就绪时，该数据被发送到客户机。这种方式的另一个优点是可以大大减少发送到服务器的请求，从 而避免了与设置服务器连接相关的开销和延时。但是，Streaming 也是在 HTTP 的基础上做了包装，介入的防火墙和代理服务器会对 response 做缓存，增加了消息传输的延迟。所以，当遇到代理服务器做了缓存的情况，很多流式解决方案就会回退到 Long-Polling 方式。当然，TLS(SSL)连接可以用来逃避 response 被缓存，但是建立和销毁每个连接的代价实在太高了。另外不幸的是，XMLHttpRequest 在不同的浏览器中有很多不同的实现。这项技术只能在较新版本的 Mozilla Firefox 中可靠地使用。对于 Internet Explorer 或 Safari，仍需使用长轮询

小结

上面提到的所有的实时交互方式都会用到 HTTP 的请求头和响应头，它们包含有很多不必要的额外的信息以至于增加了延迟。

其实全双工通信方式不仅是从服务端到客户端的下行数据流就可以解决的。在这半双工的 HTTP 基础上来模仿全双工通信，现在的一些解决方案是使用了两个连接：一个是下行数据流，一个是上行数据流。但是要维持这两个连接并且使之协同工作，不但非常复杂，而且需要消耗大量资源。简单说来，HTTP 就不是为了实时和全双工通信而设计的。

下图展示了创建一个 WEB 应用的复杂性，它是在 HTTP 之上，使用发布/订阅模型，从后端往前端传输实时数据



FlashSocket

简介

Socket 套接字连接允许 Flash 播放器通过指定的端口与服务器通信，socket 连接与其他通信技术最大的不同是 socket 连接在数据传输完成后不会自动关闭。这个特点决定了它可以作为即时通信

评价

FlashSocket 透明地提供了 WebSocket 的功能，即使是在不支持 HTML5 WebSocket 的浏览器上也是如此

FlashSocket 有着下面的这些缺点：

1). 其需要安装 Flash 插件（通常情况下，所有浏览器都会有该插件）。但是某些客

户端，比如 iPhone/iPad，不支持 flash
2). 其要求防火墙的 843 端口是打开的，这样 Flash 组件才能发出 HTTP 请求来检索包含了域授权的策略文件。如果 843 端口是不可到达的话，则库应该有回退动作或是给出一个错误，所有的这些处理都需要一些时间（最多 3 秒，这取决于库），而这会降低网站的速度。
3). 如果客户端处在某个代理服务器的后面的话，到端口 843 的连接可能会被拒绝

WebSocket

简介

这是 WEB 通信方式的革新，基于浏览器原生 socket，实现了全双工通信，使 WEB 上的真正的实时通信成为可能。与 Ajax 相比，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；XHR 受到域的限制，而 WebSocket 允许跨域通信。WebSocket 标准正由 W3C 制定，目前正处于草稿阶段，但是相信在不久的将来，它将会改变 WEB 实时通信方式

客户端支持

Chrome5+
Safari5+
FireFox6+(由于 WebSocket 目前只是草案阶段，故 FireFox 也仅仅是提供了一个临时方案，标准的 interface 是 WebSocket，FireFox 是命名为 MozWebSocket)
IE9 都还不支持

客户端 API

```
[Constructor(in DOMString url, in optional DOMString
protocols)]
[Constructor(in DOMString url, in optional DOMString[]
protocols)]
interface WebSocket {
  readonly attribute DOMString 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 bufferedAmount;
```

```
  // networking
          attribute Function onopen;
          attribute Function onmessage;
          attribute Function onerror;
          attribute Function onclose;
  readonly attribute DOMString protocol;
  void send(in DOMString data);
  void close();
};
WebSocket implements EventTarget;
```

上面就是 WebSocket 客户端编程所需的全部了，下面给出一个小例子：

```
<html>
<head>
    <title>WebSocket Test</title>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <script type="text/javascript" src="js/jquery-1.4.4.min.js"></script>
    <script type='text/javascript'>
        if (!window.WebSocket) {
            //连 WebSocket 这个对象都没有，那说明客户端根本就不支持 WebSocket
            alert("WebSocket is not supported!");
        } else {
            var socket = {
                start : function() {
                    //注意协议前缀不再是 http，而是 ws，https 相对应的是 wss
                    var location = "ws://localhost/ws/echo";
                    this._ws = new WebSocket(location);
                    //握手成功建立连接之后触发
                    this._ws.onopen = this.whenOpen;
                    //当接收到服务端推过来的消息时触发
                    this._ws.onmessage = this.whenMessage;
                    //当连接断开时触发
                    this._ws.onclose = this.whenClose;
                },
                whenOpen : function(m) {
                    alert("the connection is open;Handshake is ok");
                },
                sendMsg : function(message) {
                    if (this._ws) {
                        //除了几个事件驱动回调函数之外，WebSocket 还内置一个 send 方法
                        this._ws.send(message);
                    }
```

```
                    },
                    whenMessage : function(m) {
                        $("#msgSpan").html(m.data);
                    },
                    whenClose : function(m) {
                        this._ws = null;
                    }
                };
                socket.start();
                $(function() {
                    $("#sendBtn").click(function() {
                        alert("点击事件触发");
                        socket.sendMsg($("#sendTxt").val());
                    });
                });
            }
        </script>
</head>
<body>
<h3>WebSocket 实例</h3>
<input type="text" id="sendTxt">
<input type="button" value="发送" id="sendBtn"><br><br>
服务端回送的消息：<span id="msgSpan"></span><br>
</body>
</html>
```

是的，WebSocket 客户端编程就是这么简单，不管你信不信，反正我是信了。通过 onmessage 方法随时准备接收服务端发过来的消息，通过 send 方法直接发送消息

服务端支持

由于 WebSocket 目前仅仅是草稿阶段，所以服务端的支持还不是很完善，也没有统一的规范接口，各自做各自的，希望不久的将来这点能有所改观

php - http://code.google.com/p/phpwebsocket/

jetty - http://jetty.codehaus.org/jetty/ (版本 7 才开始支持 websocket)

netty - http://www.jboss.org/netty

ruby - http://github.com/gimite/web-socket-ruby

Kaazing - http://www.kaazing.org/confluence/display/KAAZING/Home

Node.js - http://nodejs.org/

服务端 DEMO

这里我采用的是 jetty-hightide-8.0.0.M2 做为服务器，代码如下

```
package org.qxh.ws.web;

import javax.servlet.http.HttpServletRequest;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.jetty.websocket.WebSocket;
import org.eclipse.jetty.websocket.WebSocketServlet;

import java.io.IOException;

/**
 * @author qinxiaohui01
 */
public class EchoWebSocketServlet extends WebSocketServlet {

	private final static Log log = LogFactory.getLog(EchoWebSocketServlet.class);

	protected WebSocket doWebSocketConnect(HttpServletRequest request, String protocol) {
		//用一个 WebSocket 来处理浏览器的请求
		return new MyWebSocket();
	}

	public class MyWebSocket implements WebSocket {
		Outbound outbound;
		public MyWebSocket() {
			super();
		}
		public void onConnect(Outbound outbound) {
			//建立连接时的处理方法
			this.outbound = outbound;
			System.out.println("onConnect");
		}

		public void onDisconnect() {
			//断开连接时的处理方法
			System.out.println("onDisconnect");
		}
		public void onMessage(final byte frame, String data) {
			//接到客户端消息时的处理方法
			System.out.println("onMessage");
			try {
				this.outbound.sendMessage(frame,"Your message is:"+data);
			} catch (IOException e) {
```

```
                log.error(e);
            }
        }

        public void onMessage(byte frame, byte[] arg1, int arg2, int arg3) {
            onMessage(frame, new String(arg1, arg2, arg3));
        }

        public void onFragment(boolean arg0, byte arg1, byte[] arg2, int arg3, int arg4) {
            System.out.println("onFragment");
        }
    }
}
```

编译这个 servlet 的 jar 在 jetty 根目录下的 lib 目录下都可以找到，要不给大家看一下我的部分依赖吧：

```
        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-servlet</artifactId>
            <version>8.0.0.M2</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-util</artifactId>
            <version>8.0.0.M2</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-websocket</artifactId>
            <version>8.0.0.M2</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-xml</artifactId>
            <version>8.0.0.M2</version>
            <scope>provided</scope>
        </dependency>

        <dependency>
```

```
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.4</version>
            <scope>compile</scope>
        </dependency>
```

像普通 servlet 一样，在 web.xml 中配置一下，我的 url-pattern 姑且为：/echo

针对没有用过 jetty 的人我再啰嗦两句，使用 jetty 其实很简单，下载 zip 文件（http://dist.codehaus.org/jetty/）解压缩，在根目录下运行 java -jar start.jar 即可启动。

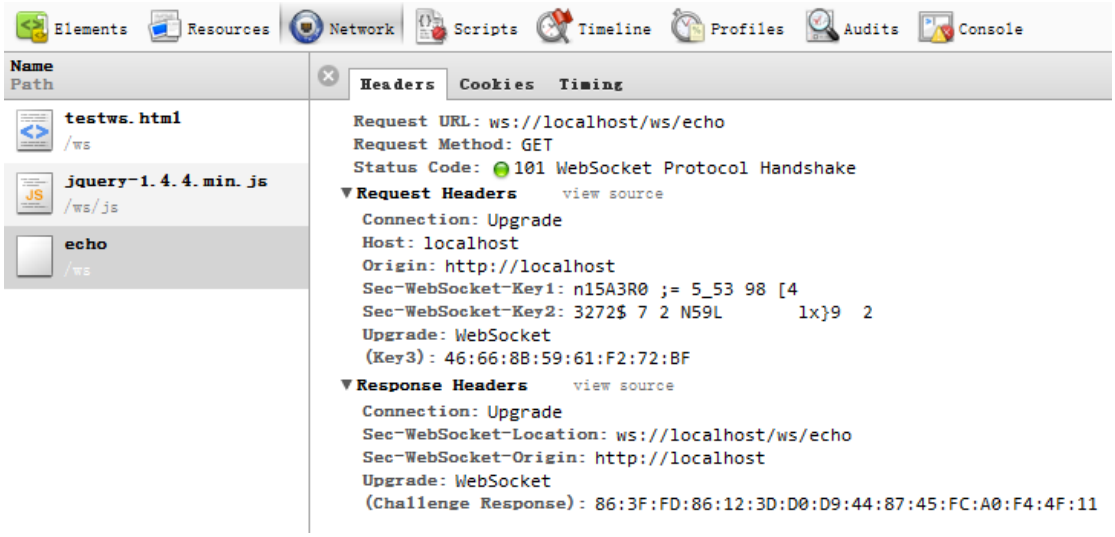
要想把刚才写的项目配置到 jetty 中，可以写一个 mydemo.xml 文件，内容如下：

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/jetty/configure.dtd">
<Configure class="org.eclipse.jetty.webapp.WebAppContext">
  <Set name="contextPath">/ws</Set>
  <Set name="war">D:/ideawork/wsdemo/target/wsdemo</Set>
  <Set name="extractWAR">false</Set>
  <Set name="copyWebDir">false</Set>
  <Set name="defaultsDescriptor"><SystemProperty name="jetty.home" default="."/>/etc/webdefault.xml</Set>
</Configure>
```

我项目的名字是 wsdemo，是个 maven 项目，编译之后的路径是 D:/ideawork/wsdemo/target/wsdemo 相当与 eclipse 项目的 WebRoot 目录，按照上面的方式配置一下，然后把 mydemo.xml 文件放到 jetty 根目录下的 contexts 下即可。OK，接下来可以运行 jetty 了，cd 到 jetty 根目录 java –jar start.jar,走之，jetty 启动成功，输入 http://localhost/ws/testws.html 即可访问

WebSocket 请求响应头信息

在地址栏里敲入上面给出的 url，敲回车之前，请打开 chrome 的开发人员工具（ctrl+shift+i），访问这个地址过程中我们可以看到请求响应头信息：



大家感兴趣可以分析一下这个请求响应头信息

我用的是 chrome13，支持的是 76 这个 WebSocket 版本，其它版本可能会跟这个头信息不一样，目前在请求头中可以看到两个 key 值：

Sec-WebSocket-Key1:2901] 949315

Sec-WebSocket-Key2:289 00u07285

在 WebSocket version7 中这两个就是合二为一的

在 WebSocket 中，这个请求头信息和响应头信息会在浏览器和服务器之间建立一个握手，接下来双方就可以依托于这个信道相互推送消息了。WebSocket 规定，推送的消息均为 UTF-8 格式，以 0x00 这个 byte 作为数据帧开头，以 0xFF 这个 byte 作为数据帧结尾。中间即是你要真正发送的信息。到了这里你想到了什么？

消息大小！

是的，就是消息大小！不管是服务端推送消息给客户端还是客户端推送消息给服务端，除去你真正要发送的那部分数据，这个大小只有 2byte！而普通的 HTTP 请求头或响应头一般都是几百个 byte，有的甚至几千个 byte，粗略的估算一下，假设每个请求头平均大小为 1000byte，那么就是 1000:2!也就是说，原来承载 1 个请求头的带宽，现在可以承载 500 个！不管你激动不激动，反正我是很激动！

WebSocket 在实际项目中应用

到此为止，你已经懂得了如何利用 WebSocket 写一个 Hello World 级别的程序了，学一个技术会写 Hello World 了，其实就是学会了 50%（个人认为），那么 WebSocket 如何在真正的企业项目中应用？我这里就不给出源码了，仅仅给出一些意见----

1、 由于 WebSocket 目前还处于草案阶段，浏览器支持的不是很好，所以要把它用在实时交互的场景中并且做到兼容各个浏览器，那么一个 if else 判断是不可少的：

```
If(!window.WebSocket){
        //该浏览器不支持 WebSocket，使用 Polling 或 Long-Polling
} else {
        var socket = new WebSocket("ws://xxxx.com/xxx");
}
```

2、 所有业务逻辑都交给位于 WEB 服务器的 WebSocketServlet 来处理这显然是不合适的，

有的时候我们需要在 Service 层的某个操作结束之后就推送一个消息给浏览器，这时候我们就需要一个存放各个 WebSocket 的容器，根据容器中每个 WebSocket 的唯一标识找到相对应的 WebSocket，然后把消息交给它，让它去发送。

3、 上面提到的这个 Service 层如果也是在 Jetty（我姑且使用 Jetty 作为 WEB 服务器）中运行的，这个比较容易。但是如果我 WEB 服务器部署在一台机器上，应用服务器部署在另一台机器上，位于不同 JVM 的 Service 和 WebSocket 如何才能相互调用呢？这个时候或许你就会用到 JMS 或是 Web Service 了

最后真切希望我整理的这篇文档对大家有所帮助

参考文档：

《Pro.HTML5.Programming》
http://dev.w3.org/html5/websockets/
http://www.websocket.org/
http://en.wikipedia.org/wiki/WebSocket