当我们探讨两件事物的区别和联系时，我们想探讨些什么？

前段时间写了两篇介绍 HTTP 和 WebSocket 的文章，回复中有人说希望了解下 WebSocket 和 Socket 的区别。这个问题之前也有想过，自己对此是有大概的答案，可是并不太确定，于是去搜集了些资料（其实就是各种 Google），看了很多以前的文档，觉得有些故事十分有趣，整理如下，算是一个外传。

文中图片全来自 Google 图片搜索，如侵删。

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。

Java 和 JavaScript 的关系

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 可以有很多意思，和 IT 较相关的本意大致是指**在端到端的一个连接中，这两个端叫做 Socket**。对于 IT 从业者来说，它往往指的是 TCP/IP 网络环境中的两个连接端，大多数的 API 提供者（如操作系统，JDK）往往会提供基于这种概念的接口，所以对于开发者来说也往往是在说一种编程概念。同时，操作系统中进程间通信也有 Socket 的概念，但这个 Socket 就不是基于网络传输层的协议了。

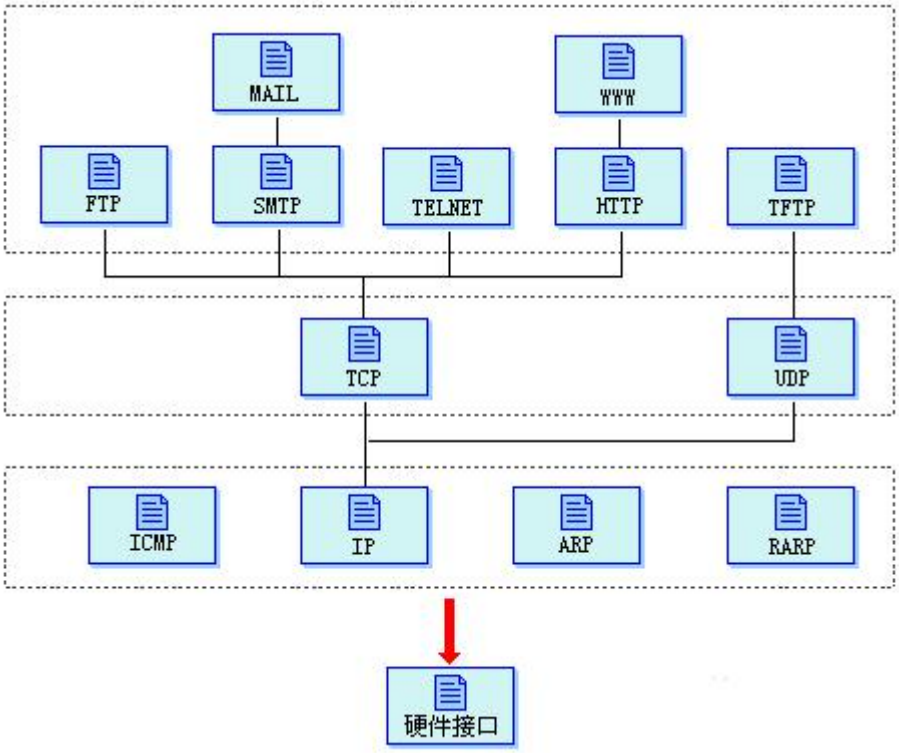
Unix 中的 Socket

操作系统中也有使用到 Socket 这个概念用来进行进程间通信，它和通常说的基于 TCP/IP 的 Socket 概念十分相似，代表了在操作系统中传输数据的两方，只是它不再基于网络协议，而是操作系统本身的文件系统。

网络中的 Socket

通常所说的 Socket API，是指操作系统中（也可能不是操作系统）提供的对于传输层（TCP/UDP）抽象的接口。现行的 Socket API 大致都是遵循了 BSD Socket 规范（包括 Windows）。这里称规范其实不太准确，规范其实是 POSIX，但 BSD Unix 中对于

Socket 的实现被广为使用，所以成为了实际的规范。如果你要使用 HTTP 来构建服务，那么就不需要关心 Socket，如果你想基于 TCP/IP 来构建服务，那么 Socket 可能就是你会接触到的 API。



在 TCP/IP 网络中 HTTP 的位置

从上图中可以看到，HTTP 是基于传输层的 TCP 协议的，而 Socket API 也是，所以只是从使用上说，可以认为 Socket 和 HTTP 类似（但一个是成文的互联网协议，一个是一直沿用的一种编程概念），是对于传输层协议的另一种直接使用，因为按照设计，网络对用户的接口都应该在应用层。

Socket 名称的由来

和很多其他 Internet 上的事物一样，Socket 这个名称来自于大名鼎鼎的 ARPANET（Advanced Research Projects Agency），早期 ARPANET 中的 Socket 指的是一个源或者目的地址——大致就是今天我们所说的 IP 地址和端口号。最早的时候一个 Socket 指的是一个 40 位的数字（RFC33 中说明了此用法，但在 RFC36 中并没有明确地说使用 40 位数字来标识一个地址），其中前 32 为指向的地址（socket number，大致相当于 IP），后 8 位为发送数据的源（link，大致相当于端口号）。对他们的叫法有很多的版本，这里列举的并不严谨。

端口号的野史

随着 ARPANET 的发展，后来（RFC433，Socket Number List）socket number 被明确地定义为一个 40 位的数字，其中后 8 位被用来制定某个特定的应用使用（比如 1 是 Telnet）。这 8 位数有很多名字 :link、socket name、AEN(another eight number，看到这个名字我也是醉了），工程师逗逼起来也是挺拼的。

后来在 Internet 的规范制定中，才真正的用起了 port number 这个词。至于为什么端口号是 16 位的，我想可能有两个原因，一是对于当时的工程师来说，如果每个端口号来标识一个程序，65535 个端口号也差不多够用了。二可能是为了对齐吧，^_^!!。

Socket 原本的意思

在上边提到的历史中使用到的 Socket，包括 TCP 文档中使用到的 Socket，其实指的是网络传输中的一端，是一个虚拟化的概念。

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，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

```
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;
```

```
21  void close([Clamp] optional unsigned short code, optional USVString reason);
22
23  // messaging
24  attribute EventHandler onmessage;
25  attribute BinaryType binaryType;
26  void send(USVString data);
27  void send(Blob data);
28  void send(ArrayBuffer data);
29  void send(ArrayBufferView data);
    };
```

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

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。

去年光棍节的时候，我写过一篇 quick-cocos2d-x 中的 socket 技术选择：LuaSocket 和 WebSocket 。这篇文章介绍了我为何决定在项目中使用 LuaSocket 。

现在想起来，当时对 WebSocket 是很感兴趣的，但由于服务端的限制，最终依然选择了 LuaSocket。我后来对 LuaSocket 进行了封装，使其更好用。

现在，面对一个全新的项目，我自然而然地选择了 WebSocket。

因此，我需要了解下面这些问题：

1. Socket 和 WebSocket 有哪些区别和联系？
2. WebSocket 和 HTML5 是什么关系？
3. 必须在浏览器中才能使用 WebSocket 吗？
4. WebSocket 能和 Socket 一样传输 raw 数据么？
5. WebSocket 和 Socket 相比会多耗费流量么？

但是，目前网上全面介绍这两种协议的中文文章并不多，或者说不够全面。我无法找到一篇文章能解决上面的所有问题。因此，我写了本文，把找到的 Socket 和 WebSocket 的相关资料做一个梳理，以方便理解。

本文并不能直接完整回答上面提出的几个问题，但读完本文，要理解上面的那些问题，是很容易的事。

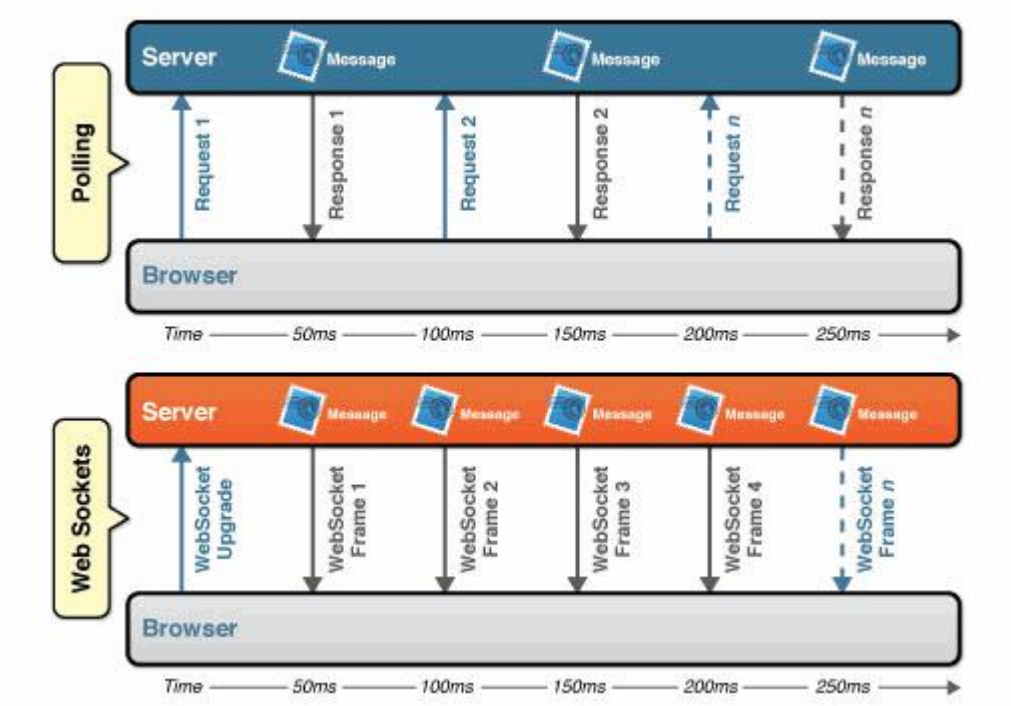
由于能力有限，本文不可能很长。而且，技术细节并非所有人都愿意仔细了解。本文包含了大量的外部链接，跟随这些链接，可以找到足够多的细节，满足你/我的求知欲。

---

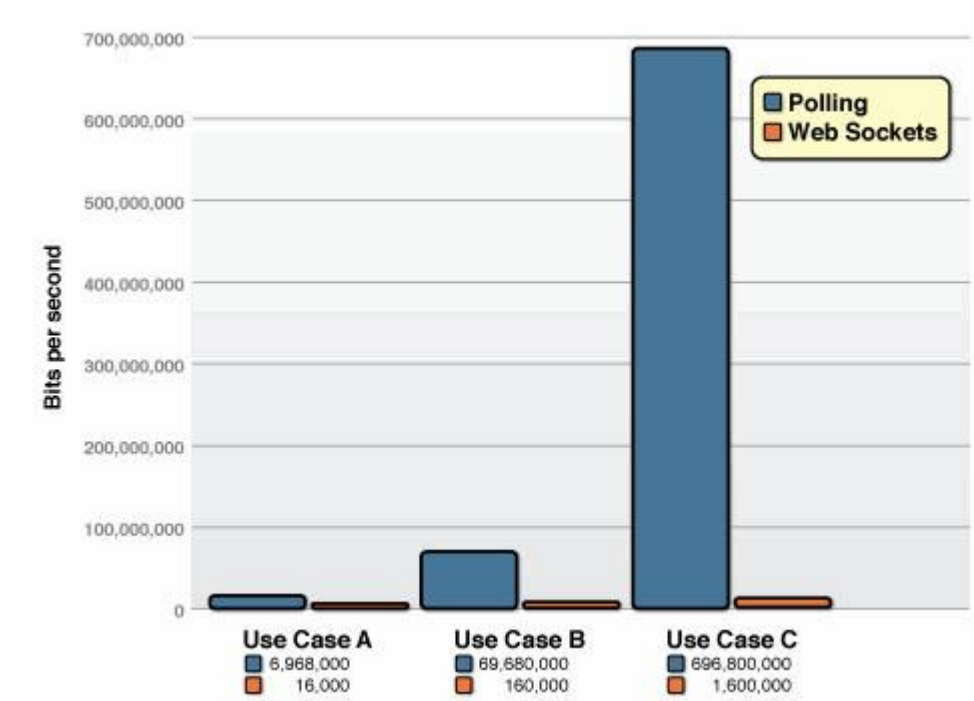
1. 概述

选择了 WebSocket 技术之后，不可避免的，我要将它和其他协议以及技术做一下比较。最常见的，就是需要比较 WebSocket 与 HTTP、Socket 技术的异同。

WebSocket 是为了满足基于 Web 的日益增长的实时通信需求而产生的。在传统的 Web 中，要实现实时通信，通用的方式是采用 HTTP 协议不断发送请求。但这种方式即浪费带宽（HTTP HEAD 是比较大的），又消耗服务器 CPU 占用（没有信息也要接受请求）。（下图来自 WebSocket.org）



而是用 WebSocket 技术，则会大幅降低上面提到的消耗：（下图来自 websocket.org）



关于更详细的描述，尹立的这篇文章讲得非常好：WebSocket（2）—为什么引入 WebSocket 协议 。

那么，WebSocket 到底与 HTTP 协议到底是一个什么样的关系呢？它和 Socket 又有什么联系？这就要讲到 OSI 模型和 TCP/IP 协议族。

2. OSI 模型与 TCP/IP

以下是 维基百科 中关于 OSI 模型的说明：

> 开放式系统互联通信参考模型（英语：Open System Interconnection Reference Model，ISO/IEC 7498-1），简称为 OSI 模型（OSI model），一种概念模型，由国际标准化组织（ISO）提出，一个试图使各种计算机在世界范围内互连为网络的标准框架。

而 TCP/IP 协议可以看做是对 OSI 模型的一种简化（以下内容来自 维基百科）：

> 它将软件通信过程抽象化为四个抽象层，采取协议堆叠的方式，分别实作出不同通信协议。协议套组下的各种协议，依其功能不同，被分别归属

到这四个阶层之中 7，常被视为是简化的七层 OSI 模型。

这里有一张图详细介绍了 TCP/IP 协议族中的各个协议在 OSI 模型 中的分布，一图胜千言（下图来自 科来）：

科来软件
Colasoft Co., Ltd.

TC

第7层　应用层

各种应用程序协议，如HTTP、FTP、SMTP、POP3。

常见使用TCP协议的应用层服务

HTTP 超文本传输协议	FTP 文件传输协议	SMTP 简单邮件传输协议	TELNET TCP/IP终端仿真协
POP3 邮局协议第3版	Finger 用户信息协议	NNTP 网络新闻传输协议	IMAP4 因特网信息访问协议第

HP网络服务

NTF HP 网络文件传输协议	RDA HP 远程数据库访问协议	VT 虚拟终端仿真协议	RFA HP 远程文件访问协议	RPC Remote Process Comm.

S-HTTP 安全超文本传输协议

GDP 网关发现协议

X-Window X-Window

CMOT 基于TCP/IP的CMIP协议

7

第6层　表示层

信息的语法语义以及它们的关联，如加密解密、转换翻译、压缩解压缩。

DECnet NSP

LPP 轻量级表示协议

NBSSN NetBIOS会话服务协议

6

第5层　会话层

不同机器上的用户之间建立及管理会话。

安全协议

SSL 安全套接字层协议	TLS 传输层安全协议

目录访问协议

DAP 目录访问协议	LDAP 轻量级目录访问协议

5

第4层　传输层

接受上一层的数据，在必要的时候把数据进行分割，并将这些数据交给网络层，且保证这些数据段有效到达对端。

DSI NetBIOS	IP NeTBIOS ISO-TP SSP	SMB MSRPC

NetBIOS

XOT 基于TCP之上的X.25协议

Van Jacobson 压缩TCP协议

ISO-DE ISO开发环境

4

TCP 传输控制协议

UDP 用户数据报协议

TCP/IP 协议和 OSI 模型的内容，在互联网上有很多。我没有必要再次介绍它们。在这里，我们只需要知道，HTTP、WebSocket 等协议都是处于 OSI 模型的最高层： **应用层** 。而 IP 协议工作在网络层（第 3 层），TCP 协议工作在传输层（第 4 层）。

至于 OSI 模型的各个层次都有什么系统和它们对应，这里有篇很好的文章可以满足大家的求知欲：OSI 七层模型详解 。

3. WebSocket、HTTP 与 TCP

从上面的图中可以看出，HTTP、WebSocket 等应用层协议，都是基于 TCP 协议来传输数据的。我们可以把这些高级协议理解成对 TCP 的封装。

既然大家都使用 TCP 协议，那么大家的连接和断开，都要遵循 TCP 协议中的三次握手和四次握手 ，只是在连接之后发送的内容不同，或者是断开的时间不同。

更详细内容可阅读：wireshark 抓包图解 TCP 三次握手/四次挥手详解

对于 WebSocket 来说，它必须依赖 HTTP 协议进行一次握手 ，握手成功后，数据就直接从 TCP 通道传输，与 HTTP 无关了。

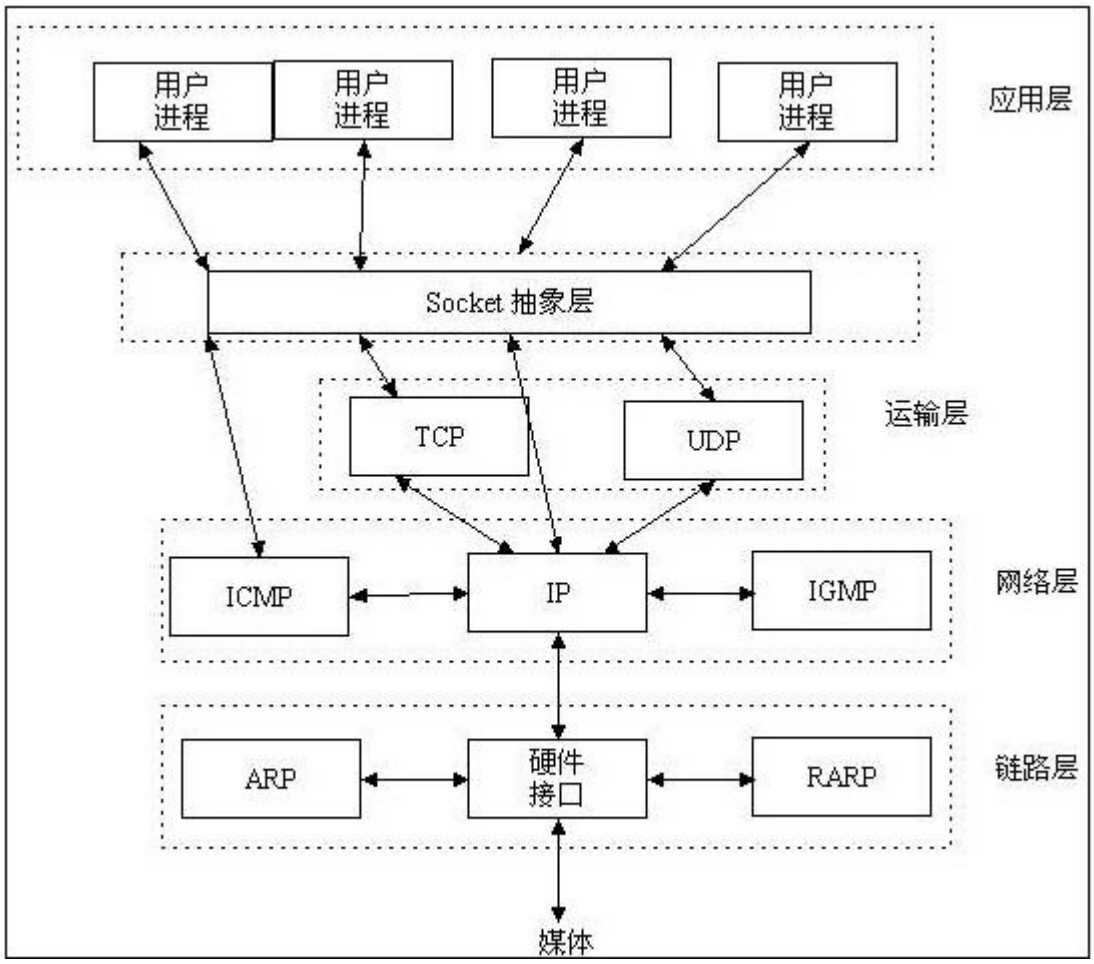
4. Socket 与 WebScoket

Socket 其实并不是一个协议。它工作在 OSI 模型会话层(第 5 层)，是为了方便大家直接使用更底层协议（一般是 TCP 或 UDP ）而存在的一个抽象层。

最早的一套 Socket API 是 Berkeley sockets ，采用 C 语言实现。它是 Socket 的事实标准，POSIX sockets 是基于它构建的，多种编程语言都遵循这套 API，在 Java、Python 中都能看到这套 API 的影子。

下面摘录一段更容易理解的文字（来自 http 和 socket 之长连接和短连接区别）：

> *Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，以符合指定的协议。*



主机 A 的应用程序要能和主机 B 的应用程序通信，必须通过 Socket 建立连接，而建立 Socket 连接必须需要底层 TCP/IP 协议来建立 TCP 连接。建立 TCP 连接需要底层 IP 协

> *议来寻址网络中的主机。我们知道网络层使用的 IP 协议可以帮助我们根据 IP 地址来找到目标主机，但是一台主机上可能运行着多个应用程序，如何才能与指定的应用程序通信就要通过 TCP 或 UPD 的地址也就是端口号来指定。这样就可以通过一个 Socket 实例唯一代表一个主机上的一个应用程序的通信链路了。*

而 WebSocket 则不同，它是一个完整的 应用层协议，包含一套标准的 API 。

所以，从使用上来说，WebSocket 更易用，而 Socket 更灵活。

5. HTML5 与 WebSocket

WebSocket API 是 HTML5 标准的一部分， 但这并不代表 WebSocket 一定要用在 HTML 中，或者只能在基于浏览器的应用程序中使用。

实际上，许多语言、框架和服务器都提供了 WebSocket 支持，例如：

- 基于 C 的 libwebsocket.org
- 基于 Node.js 的 Socket.io
- 基于 Python 的 ws4py

- 基于 C++ 的 WebSocket++
- Apache 对 WebSocket 的支持： Apache Module mod_proxy_wstunnel
- Nginx 对 WebSockets 的支持： NGINX as a WebSockets Proxy 、 NGINX Announces Support for WebSocket Protocol 、WebSocket proxying
- lighttpd 对 WebSocket 的支持：mod_websocket

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

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。

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。

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。

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/1.1Upgrade: WebSocketConnection: UpgradeHost:
www.qixing318.comOrigin: http://www.qixing318.com
```

服务器返回：

```
HTTP/1.1 101 Web Socket Protocol HandshakeUpgrade: WebSocketConnection: 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 HandshakeUpgrade: WebSocketConnection: Upgrade

WebSocket-Origin: http://www.qixing318.com

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 ProtocolsUpgrade: websocketConnection:
UpgradeSec-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/1.1Host: server.example.comUpgrade: websocketConnection:
UpgradeSec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==Origin:
http://example.comSec-WebSocket-Protocol: chat,
superchatSec-WebSocket-Version: 13
```

服务器端响应：

```
HTTP/1.1 101 Switching ProtocolsUpgrade: websocketConnection:
UpgradeSec-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/1.1Host: 192.168.14.215:9000Connection: UpgradePragma:
no-cache

Cache-Control: no-cacheUpgrade: websocketOrigin: 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

WebSocket 是 html5 新增加的一种通信协议，目前流行的浏览器都支持这个协议，例如 Chrome，Safrie，Firefox，Opera，IE 等等，对该协议支持最早的应该是 chrome，从 chrome12 就已经开始支持，随着协议草案的不断变化，各个浏览器对协议的实现也在不停的更新。该协议还是草案，没有成为标准，不过成为标准应该只是时间问题了，从 WebSocket 草案的提出到现在已经有十几个版本了，目前最新的是版本 17，所对应的协议版本号为 13，目前对该协议支持最完善的浏览器应该是 chrome，毕竟 WebSocket 协议草案也是 Google 发布的。

1. WebSocket API 简介

首先看一段简单的 javascript 代码，该代码调用了 WebSockets 的 API。

```
var ws = new WebSocket("ws://echo.websocket.org");

ws.onopen = function(){ws.send("Test!"); };

ws.onmessage = function(evt){console.log(evt.data);ws.close();};

ws.onclose = function(evt){console.log("WebSocketClosed!");};

ws.onerror = function(evt){console.log("WebSocketError!");};
```

这份代码总共只有 5 行，现在简单概述一下这 5 行代码的意义。

第一行代码是在申请一个 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。

第二行到第五行为 WebSocket 对象注册消息的处理函数，WebSocket 对象一共支持四个消息 onopen, onmessage, onclose 和 onerror，当 Browser 和 WebSocketServer 连接成功后，会触发 onopen 消息；如果连接失败，发送、接收数据失败或者处理数据出现错误，browser 会触发 onerror 消息；当 Browser 接收到 WebSocketServer 发送过来的数据时，就会触发 onmessage 消息，参数 evt 中包含 server 传输过来的数据；当 Browser 接收到 WebSocketServer 端发送的关闭连接请求时，就会触发 onclose 消息。我们可以看出所有的操作都是采用消息的方式触发的，这样就不会阻塞 UI，使得 UI 有更快的响应时间，得到更好的用户体验。

二为什么引入 WebSocket 协议？？

Browser 已经支持 http 协议，为什么还要开发一种新的 WebSocket 协议呢？我们知道 http 协议是一种单向的网络协议，在建立连接后，它只允许 Browser/UA（UserAgent）向 WebServer 发出请求资源后，WebServer 才能返回相应的数据。而 WebServer 不能主动的推送数据给 Browser/UA，当初这么设计 http 协议也是有原因的，假设 WebServer 能主动的推送数据给 Browser/UA，那 Browser/UA 就太容易受到攻击，一些广告商也会主动的把一些广告信息在不经意间强行的传输给客户端，这不能不说是一个灾难。那么单向的 http 协议给现在的网站或 Web 应用程序开发带来了哪些问题呢？

让我们来看一个案例，现在假设我们想开发一个基于 Web 的应用程序去获取当前 Web 服务器的实时数据，例如股票的实时行情，火车票的剩余票数等等，这就需要 Browser/UA 与 WebServer 端之间反复的进行 http 通信，Browser 不断的发送 Get 请求，去获取当前的实时数据。下面介绍几种常见的方式：

1. Polling

这种方式就是通过 Browser/UA 定时的向 Web 服务器发送 http 的 Get 请求，服务器收到请求后，就把最新的数据发回给客户端（Browser/UA），Browser/UA 得到数据后，就将其显示出来，然后再定期的重复这一过程。虽然这样可以满足需求，但是也仍然存在一些问题，例如在某段时间内 Web 服务器端没有更新的数据，但是 Browser/UA 仍然需要定时的发送 Get 请求过来询问，那么 Web 服务器就把以前的老数据再传送过来，Browser/UA 把这些没有变化的数据再显示出来，这样显然既浪费了网络带宽，又浪费了 CPU 的利用率。如果说把 Browser 发送 Get 请求的周期调大一些，就

可以缓解这一问题，但是如果在 Web 服务器端的数据更新很快时，这样又不能保证 Web 应用程序获取数据的实时性。

2. Long Polling

上面介绍了 Polling 遇到的问题，现在介绍一下 LongPolling，它是对 Polling 的一种改进。

Browser/UA 发送 Get 请求到 Web 服务器，这时 Web 服务器可以做两件事情，第一，如果服务器端有新的数据需要传送，就立即把数据发回给 Browser/UA，Browser/UA 收到数据后，立即再发送 Get 请求给 Web Server；第二，如果服务器端没有新的数据需要发送，这里与 Polling 方法不同的是，服务器不是立即发送回应给 Browser/UA，而是把这个请求保持住，等待有新的数据到来时，再来响应这个请求；当然了，如果服务器的数据长期没有更新，一段时间后，这个 Get 请求就会超时，Browser/UA 收到超时消息后，再立即发送一个新的 Get 请求给服务器。然后依次循环这个过程。

这种方式虽然在某种程度上减小了网络带宽和 CPU 利用率等问题，但是仍然存在缺陷，例如假设服务器端的数据更新速率较快，服务器在传送一个数据包给 Browser 后必须等待 Browser 的下一个 Get 请求到来，才能传递第二个更新的数据包给 Browser，那么这样的话，Browser 显示实时数据最快的时间为 2×RTT（往返时间），另外在网络拥塞的情况下，这个应该是不能让用户接受的。另外，由于 http 数据包的头部数据量往往很大（通常有 400 多个字节），但是真正被服务器需要的数据却很少（有时只有 10 个字节左右），这样的数据包在网络上周期性的传输，难免对网络带宽是一种浪费。

通过上面的分析可知，要是在 Browser 能有一种新的网络协议，能支持客户端和服务器端的双向通信，而且协议的头部又不那么庞大就好了。WebSocket 就是肩负这样一个使命登上舞台的。

三 websocket 协议简介

WebSocket 协议是一种双向通信协议，它建立在 TCP 之上，同 http 一样通过 TCP 来传输数据，但是它和 http 最大的不同有两点：1.WebSocket 是一种双向通信协议，在建立连接后，WebSocket 服务器和 Browser/UA 都能主动的向对方发送或接收数据，就像 Socket 一样，不同的是 WebSocket 是一种建立在 Web 基础上的一种简单模拟 Socket 的协议；2.WebSocket 需要通过握手连接，类似于 TCP 它也需要客户端和服务器端进行握手连接，连接成功后才能相互通信。

下面是一个简单的建立握手的时序图：

这里简单说明一下 WebSocket 握手的过程。

当 Web 应用程序调用 new WebSocket(url)接口时，Browser 就开始了与地址为 url 的 WebServer 建立握手连接的过程。

1. Browser 与 WebSocket 服务器通过 TCP 三次握手建立连接，如果这个建立连接失败，那么后面的过程就不会执行，Web 应用程序将收到错误消息通知。

2. 在 TCP 建立连接成功后，Browser/UA 通过 http 协议传送 WebSocket 支持的版本号，协议的字版本号，原始地址，主机地址等等一些列字段给服务器端。

例如：

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

3. WebSocket 服务器收到 Browser/UA 发送来的握手请求后，如果数据包数据和格式正确，客户端和服务器端的协议版本号匹配等等，就接受本次握手连接，并给出相应的数据回复，同样回复的数据包也是采用 http 协议传输。

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept:s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

Sec-WebSocket-Protocol: chat

4. Browser 收到服务器回复的数据包后，如果数据包内容、格式都没有问题的话，就表示本次连接成功，触发 onopen 消息，此时 Web 开发者就可以在此时通过 send 接口想服务器发送数据。否则，握手连接失败，Web 应用程序会收到 onerror 消息，并且能知道连接失败的原因。

四 websocket 与 TCP,HTTP 的关系。

WebSocket 与 http 协议一样都是基于 TCP 的，所以他们都是可靠的协议，Web 开发者调用的 WebSocket 的 send 函数在 browser 的实现中最终都是通过 TCP 的系统接口进行传输的。
WebSocket 和 Http 协议一样都属于应用层的协议，那么他们之间有没有什么关系呢？答案是肯定的，WebSocket 在建立握手连接时，数据是通过 http 协议传输的，正如我们上一节所看到的“GET/chat HTTP/1.1”，这里面用到的只是 http 协议一些简单的字段。但是在建立连接之后，真正的数据传输阶段是不需要 http 协议参与的。

具体关系可以参考下图：

五 websocket server

果要搭建一个 Web 服务器，我们会有很多选择，市场上也有很多成熟的产品供我们应用，比如开源的 Apache，安装后只需简单的配置（或者默认配置）就可以工作了。但是如果想搭建一个 WebSocket 服务器就没有那么轻松了，因为 WebSocket 是一种新的通信协议，目前还是草案，没有成为标准，市场上也没有成熟的 WebSocket 服务器或者 Library 实现 WebSocket 协议，我们就必须自己动手写代码去解析和组装 WebSocket 的数据包。要这样完成一个 WebSocket 服务器，估计所有的人都想放弃，幸好的是市场上有几款比较好的开源库供我们使用，比如 PyWebSocket,WebSocket-Node, LibWebSockets 等等，这些库文件已经实现了 WebSocket 数据包的封装和解析，我们可以调用这些接口，这在很大程度上减少了我们的工作量。

下面就简单介绍一下这些开源的库文件。

1. PyWebSocket

PyWebSocket 采用 Python 语言编写，可以很好的跨平台，扩展起来也比较简单，目前 WebKit 采用它搭建 WebSocket 服务器来做 LayoutTest。

我们可以获取源码通过下面的命令

svn checkouthttp://pywebsocket.googlecode.com/svn/trunk/ pywebsocket-read-only

更多的详细信息可以从 http://code.google.com/p/pywebsocket/获取。

2. WebSocket-Node

WebSocket-Node 采用 JavaScript 语言编写，这个库是建立在 nodejs 之上的，对于熟悉 JavaScript 的朋友可参考一下，另外 Html5 和 Web 应用程序受欢迎的程度越来越高，nodejs 也正受到广泛的关注。

我们可以从下面的连接中获取源码

https://github.com/Worlize/Websocket-Node

3. LibWebSockets

LibWebSockets 采用 C/C++语言编写，可定制化的力度更大，从 TCP 监听开始到封包的完成我们都可以参与编程。

我们可以从下面的命令获取源代码

git clone git://git.warmcat.com/libwebsockets

Websocket 协议详解

关于 websocket 的协议是用来干嘛的，请参考其他文章。

WebSocket 关键词

HTML5 协议，实时，全双工通信，长连接

WebSocket 比传统 Http 的好处

- 客户端与服务端只建立一个 TCP 连接，可以使用更少的连接
- WebSocket 的服务端可以将数据推送到客户端，如实时将证券信息反馈到客户端（这个很关键），实时天气数据，比 http 请求响应模式更灵活
- 更轻量的协议头，减少数据传送量

数据帧格式

下图为手工打造的数据帧格式

```
/**

 * fin    |masked         |               |
```

```
* srv1 | length |
* srv2 | (7bit |mask 数据 |payload
* srv3 | 7+2 字节 | 4 字节 |真实数据
opcode | 7+64 字节 |
*（4bit）
*/
```

```
1
2
3
4
5
6
7
8
9
10
11
```

```
1
2
```

- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

作以下说明：

1. 前 8 个 bit（一个字节）

   —fin： 是否数据发送完成，为 1 发送完成为 0 发送未完。

   —srv1，srv2，srv3：留作后用

   —opcode:数据类型操作码，4bit 表示，其中

   TEXT: 1, text 类型的字符串

   BINARY: 2,二进制数据，通常用来保存图片

   CLOSE: 8,关闭连接的数据帧。

   PING: 9, 心跳检测。ping

   PONG: 10,心跳检测。pong

```
var events = require('events');var http = require('http');var crypto = require
('crypto');var util = require('util');
```

```
/**

 *  数据类型操作码 TEXT 字符串

 *  BINARY 二进制数据 常用来保存照片

 *  PING,PONG 用作心跳检测

 *  CLOSE 关闭连接的数据帧 (有很多关闭连接的代码 1001，1009,1007,1002)

 */var opcodes = {

    TEXT: 1,

    BINARY: 2,

    CLOSE: 8,

    PING: 9,

    PONG: 10

};var WebSocketConnection = function (req, socket, upgradeHead) {    "use strict";

    var self = this;



    var key = hashWebSocketKey(req.headers['sec-websocket-key']);

    /**
```

```
 * 写头

 */

socket.write('HTTP/1.1 101 Web Socket Protocol Handshake \r\n' +

    "Upgrade:WebSocket\r\n" +

    "Connection : Upgrade\r\n" +

    "sec-websocket-accept: " + key + '\r\n\r\n');



/**

 * 接收数据

 */

socket.on('data', function (buf) {

    self.buffer = Buffer.concat([self.buffer, buf]);

    while (self._processBuffer()) {



    }

});
```

```
    socket.on('close', function (had_error) {
        if (!self.closed) {
            self.emit("close", 1006);
            self.closed = true;
        }
    });
    this.socket = socket;
    this.buffer = new Buffer(0);
    this.closed = false;

};
//websocket 连接继承事件
util.inherits(WebSocketConnection, events.EventEmitter);
/*
 发送数据函数
 * */
WebSocketConnection.prototype.send = function (obj) {    "use strict";
```

```
    var opcode;

    var payload;

    if (Buffer.isBuffer(obj)) {

        opcode = opcodes.BINARY;

        payload = obj;

    } else if (typeof obj) {

        opcode = opcodes.TEXT;

        //创造一个 utf8 的编码  可以被编码为字符串

        payload = new Buffer(obj, 'utf8');

    } else {

        throw new Error('cannot send object.Must be string of Buffer');

    }



    this._doSend(opcode, payload);

};/*

 关闭连接函数

 * */
```

```
WebSocketConnection.prototype.close = function (code, reason) {    "use stri
ct";

    var opcode = opcodes.CLOSE;

    var buffer;

    if (code) {

        buffer = new Buffer(Buffer.byteLength(reason) + 2);

        buffer.writeUInt16BE(code, 0);

        buffer.write(reason, 2);

    } else {

        buffer = new Buffer(0);

    }

    this._doSend(opcode, buffer);

    this.closed = true;

};



WebSocketConnection.prototype._processBuffer = function () {    "use strict";

    var buf = this.buffer;
```

```
if (buf.length < 2) {
    return;
}
var idx = 2;
var b1 = buf.readUInt8(0);      //读取数据帧的前 8bit
var fin = b1 & 0x80; //如果为 0x80，则标志传输结束
var opcode = b1 & 0x0f;//截取第一个字节的后四位
var b2 = buf.readUInt8(1);//读取数据帧第二个字节
var mask = b2 & 0x80;//判断是否有掩码，客户端必须要有
var length = b2 | 0x7f;//获取 length 属性 也是小于 126 数据长度的数据真实值
if (length > 125) {
    if (buf.length < 8) {
        return;//如果大于 125，而字节数小于 8，则显然不合规范要求
    }
}
if (length === 126) {//获取的值为 126 ，表示后两个字节用于表示数据长度
```

```
        length = buf.readUInt16BE(2);//读取 16bit 的值

        idx += 2;//+2

    } else if (length === 127) {//获取的值为 126 ，表示后 8 个字节用于表示数据长度

        var highBits = buf.readUInt32BE(2);//(1/0)1111111

        if (highBits != 0) {

            this.close(1009, "");//1009 关闭代码，说明数据太大

        }

        length = buf.readUInt32BE(6);//从第六到第十个字节为真实存放的数据长度

        idx += 8;

    }



    if (buf.length < idx + 4 + length) {//不够长 4 为掩码字节数

        return;

    }



    var maskBytes = buf.slice(idx, idx + 4);//获取掩码数据
```

```
        idx += 4;//指针前移到真实数据段

        var payload = buf.slice(idx, idx + length);

        payload = unmask(maskBytes, payload);//解码真实数据

        this._handleFrame(opcode, payload);//处理操作码

        this.buffer = buf.slice(idx + length);//缓存 buffer

        return true;

    };

    /**

     * 针对不同操作码进行不同处理

     * @param 操作码

     * @param 数据

     */

    WebSocketConnection.prototype._handleFrame = function (opcode, buffer) {

        "use strict";

        var payload;

        switch (opcode) {

            case opcodes.TEXT:
```

```
                payload = buffer.toString('utf8');//如果是文本需要转化为 utf8 的编码
                this.emit('data', opcode, payload);//Buffer.toString()默认 utf8 这里是故意指示的
                break;
            case opcodes.BINARY: //二进制文件直接交付
                payload = buffer;
                this.emit('data', opcode, payload);
                break;
            case opcodes.PING://发送 ping 做响应
                this._doSend(opcodes.PING, buffer);
                break;
            case  opcodes.PONG: //不做处理
                break;
            case opcodes.CLOSE://close 有很多关闭码
                let code, reason;//用于获取关闭码和关闭原因
                if (buffer.length >= 2) {
                    code = buffer.readUInt16BE(0);
```

```
                reason = buffer.toString('utf8', 2);

            }

            this.close(code, reason);

            this.emit('close', code, reason);

            break;

        default:

            this.close(1002, 'unknown opcode');

    }

};

/**

 * 实际发送数据的函数

 * @param opcode 操作码

 * @param payload 数据

 * @private

 */

WebSocketConnection.prototype._doSend = function (opcode, payload) {    "

use strict";
```

```
    this.socket.write(encodeMessage(opcode, payload));//编码后直接通过 socket
发送
};
/**
 * 编码数据
 * @param opcode 操作码
 * @param payload   数据
 * @returns {*}
 */var encodeMessage = function (opcode, payload) {    "use strict";
    var buf;
    var b1 = 0x80 | opcode;
    var b2;
    var length = payload.length;
    if (length < 126) {
        buf = new Buffer(payload.length + 2 + 0);
        b2 |= length;
        //buffer ,offset
```

```
            buf.writeUInt8(b1, 0);//读前 8bit

            buf.writeUInt8(b2, 1);//读 8-15bit

            //Buffer.prototype.copy = function(targetBuffer, targetStart, source
Start, sourceEnd) {

            payload.copy(buf, 2)//复制数据,从 2(第三)字节开始



        } else if (length < (1 << 16)) {

            buf = new Buffer(payload.length + 2 + 2);

            b2 |= 126;

            buf.writeUInt8(b1, 0);

            buf.writeUInt8(b2, 1);

            buf.writeUInt16BE(length, 2)

            payload.copy(buf, 4);

        } else {

            buf = new Buffer(payload.length + 2 + 8);

            b2 |= 127;

            buf.writeUInt8(b1, 0);
```

```
            buf.writeUInt8(b2, 1);

            buf.writeUInt32BE(0, 2)

            buf.writeUInt32BE(length, 6)

            payload.copy(buf, 10);

        }



        return buf;

};

/**

 * 解掩码

 * @param maskBytes 掩码数据

 * @param data payload

 * @returns {Buffer}

 */var unmask = function (maskBytes, data) {

    var payload = new Buffer(data.length);

    for (var i = 0; i < data.length; i++) {
```

```
        payload[i] = maskBytes[i % 4] ^ data[i];
    }
    return payload;
};var KEY_SUFFIX = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11';
/*equals to crypto.createHash('sha1').update(key+'KEY_SUFFIX').digest('base6
4')
 * */var hashWebSocketKey = function (key) {    "use strict";
    var sha1 = crypto.createHash('sha1');
    sha1.update(key + KEY_SUFFIX, 'ascii');
    return sha1.digest('base64');
};


exports.listen = function (port, host, connectionHandler) {    "use strict";
    var srv = http.createServer(function (req, res) {
    });
```

```
    srv.on('upgrade', function (req, socket, upgradeHead) {         "use strict
";

        var ws = new WebSocketConnection(req, socket, upgradeHead);

        connectionHandler(ws);

    });

    srv.listen(port, host);

};
```