http://geek.csdn.net/news/detail/172310

前言

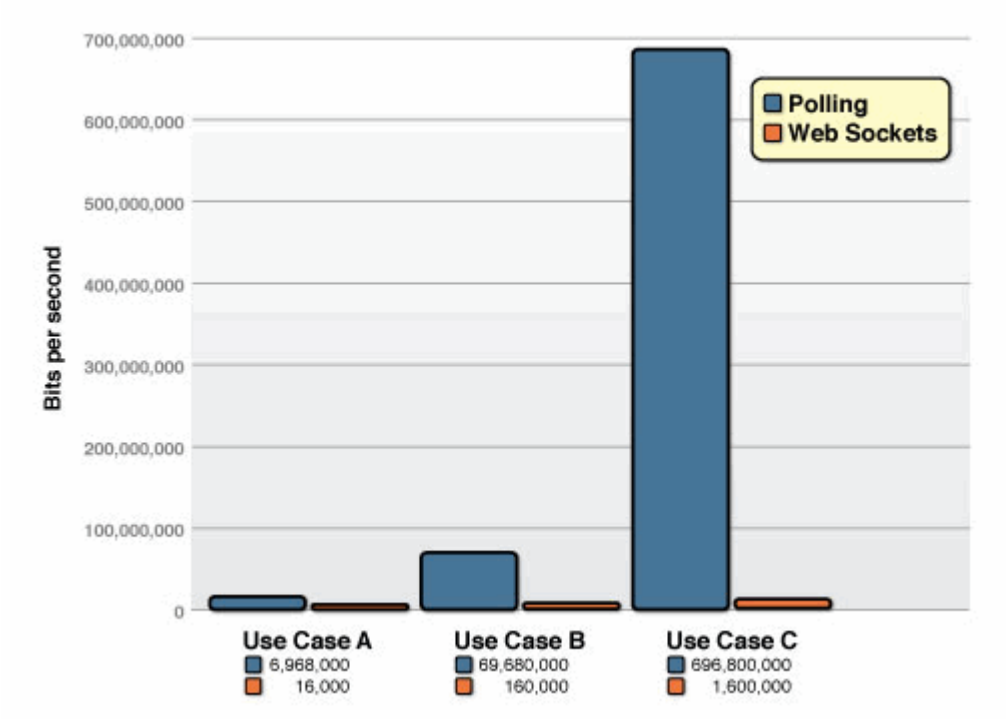
在 WebSocket API 尚未被众多浏览器实现和发布的时期，开发者在开发需要接收来自服务器的实时通知应用程序时，不得不求助于一些“hacks”来模拟实时连接以实现实时通信，最流行的一种方式是长轮询。长轮询主要是发出一个 HTTP 请求到服务器，然后保持连接打开以允许服务器在稍后的时间响应（由服务器确定）。为了这个连接有效地工作，许多技术需要被用于确保消息不错过，如需要在服务器端缓存和记录多个的连接信息（每个客户）。虽然长轮询是可以解决这一问题的，但它会耗费更多的资源，如 CPU、内存和带宽等，要想很好的解决实时通信问题就需要设计和发布一种新的协议。

WebSocket 是伴随 HTML5 发布的一种新协议。它实现了浏览器与服务器全双工通信(full-duplex)，可以传输基于消息的文本和二进制数据。WebSocket 是浏览器中最靠近套接字的 API，除最初建立连接时需要借助于现有的 HTTP 协议，其他时候直接基于 TCP 完成通信。它是浏览器中最通用、最灵活的一个传输机制，其极简的 API 可以让我们在客户端和服务器之间以数据流的形式实现各种应用数据交换（包括 JSON 及自定义的二进制消息格式），而且两端都可以随时向另一端发送数据。在这个简单的 API 之后隐藏了很多的复杂性，而且还提供了更多服务，如：

- 连接协商和同源策略；
- 与既有 HTTP 基础设施的互操作；
- 基于消息的通信和高效消息分帧；
- 子协议协商及可扩展能力。

所幸，浏览器替我们完成了上述工作，我们只需要简单的调用即可。任何事物都不是完美的，设计限制和性能权衡始终会有，利用 WebSocket 也不例外，在提供自定义数据交换协议同时，也不再享有在一些本由浏览器提供的服务和优化，如状态管理、压缩、缓存等。

随着 HTML5 的发布，越来越多的浏览器开始支持 WebSocket，如果你的应用还在使用长轮询，那就可以考虑切换了。下面的图表显示了在一种常见的使用案例下，WebSocket 和长轮询之间的带宽消耗差异：



1.WebSocket API

WebSocket 对象提供了一组 API，用于创建和管理 WebSocket 连接，以及通过连接发送和接收数据。浏览器提供的 WebSocket API 很简洁，调用示例如下：

```
var ws = new WebSocket('wss://example.com/socket'); // 创建安全 WebSocket 连接（wss）


ws.onerror = function (error) { ... } // 错误处理

ws.onclose = function () { ... } // 关闭时调用


ws.onopen = function () { // 连接建立时调用

  ws.send("Connection established. Hello server!"); // 向服务端发送消息
}
```

```
ws.onmessage = function(msg) { // 接收服务端发送的消息

  if(msg.data instanceof Blob) { // 处理二进制信息

    processBlob(msg.data);

  } else {

    processText(msg.data); // 处理文本信息

  }
}
```

1.1.接收和发送数据

WebSocket 提供了极简的 API，开发者可以轻松的调用，浏览器会为我们完成缓冲、解析、重建接收到的数据等工作。应用只需监听 onmessage 事件，用回调处理返回数据即可。 WebSocket 支持文本和二进制数据传输，浏览器如果接收到文本数据，会将其转换为 DOMString 对象，如果是二进制数据或 Blob 对象，可直接将其转交给应用或将其转化为 ArrayBuffer，由应用对其进行进一步处理。从内部看，协议只关注消息的两个信息：净荷长度和类型（前者是一个可变长度字段），据以区别 UTF-8 数据和二进制数据。示例如下：

```
var wss = new WebSocket('wss://example.com/socket');
ws.binaryType = "arraybuffer";


// 接收数据

wss.onmessage = function(msg) {
  if(msg.data instanceof ArrayBuffer) {
    processArrayBuffer(msg.data);
  } else {
    processText(msg.data);
  }
}
```

```
// 发送数据

ws.onopen = function () {
  socket.send("Hello server!");
  socket.send(JSON.stringify({'msg': 'payload'}));

  var buffer = new ArrayBuffer(128);
  socket.send(buffer);

  var intview = new Uint32Array(buffer);
  socket.send(intview);

  var blob = new Blob([buffer]);
  socket.send(blob);
}
```

Blob 对象是包含有只读原始数据的类文件对象，可存储二进制数据，它会被写入磁盘；ArrayBuffer （缓冲数组）是一种用于呈现通用、固定长度的二进制数据的类型，作为内存区域可以存放多种类型的数据。

对于将要传输的二进制数据，开发者可以决定以何种方式处理，可以更好的处理数据流，Blob 对象一般用来表示一个不可变文件对象或原始数据，如果你不需要修改它或者不需要把它切分成更小的块，那这种格式是理想的；如果你还需要再处理接收到的二进制数据，那么选择 ArrayBuffer 应该更合适。

WebSocket 提供的信道是全双工的，在同一个 TCP 连接上，可以双向传输文本信息和二进制数据，通过数据帧中的一位（bit）来区分二进制或者文本。WebSocket 只提供了最基础的文本和二进制数据传输功能，如果需要传输其他类型的数据，就需要通过额外的机制进行协商。WebSocket 中的 send( ) 方法是异步的：提供的数据会在客户端排队，而函数则立即返回。在传输大文件时，不要因为回调已经执行，就错误地以为数据已经发送出去了，数据很可能还在排队。要监控在浏览器中排队的数据量，可以查询套接字的 bufferedAmount 属性：

```
var ws = new WebSocket('wss://example.com/socket');

ws.onopen = function () {
  subscribeToApplicationUpdates(function(evt) {
```

```
    if (ws.bufferedAmount == 0)

    ws.send(evt.data);

  });

};
```

前面的例子是向服务器发送应用数据，所有 WebSocket 消息都会按照它们在客户端排队的次序逐个发送。因此，大量排队的消息，甚至一个大消息，都可能导致排在它后面的消息延迟——队首阻塞！为解决这个问题，应用可以将大消息切分成小块，通过监控 bufferedAmount 的值来避免队首阻塞。甚至还可以实现自己的优先队列，而不是盲目都把它们送到套接字上排队。要实现最优化传输，应用必须关心任意时刻在套接字上排队的是什么消息！

1.2.子协议协商

在以往使用 HTTP 或 XHR 协议来传输数据时，它们可以通过每次请求和响应的 HTTP 首部来沟通元数据，以进一步确定传输的数据格式，而 WebSocket 并没有提供等价的机制。上文已经提到 WebSocket 只提供最基础的文本和二进制数据传输，对消息的具体内容格式是未知的。因此，如果 WebSocket 需要沟通关于消息的元数据，客户端和服务器必须达成沟通这一数据的子协议，进而间接地实现其他格式数据的传输。下面是一些可能策略的介绍：

- 客户端和服务器可以提前确定一种固定的消息格式，比如所有通信都通过 JSON 编码的消息或者某种自定义的二进制格式进行，而必要的元数据作为这种数据结构的一个部分；
- 如果客户端和服务器要发送不同的数据类型，那它们可以确定一个双方都知道的消息首部，利用它来沟通说明信息或有关净荷的其他解码信息；
- 混合使用文本和二进制消息可以沟通净荷和元数据，比如用文本消息实现 HTTP 首部的功能，后跟包含应用净荷的二进制消息。

上面介绍了一些可能的策略来实现其他格式数据的传输，确定了消息的串行格式化，但怎么确保客户端和服务端是按照约定发送和处理数据，这个约定客户端和服务端是如何协商的呢？这就需要 WebSocket 提供一个机制来协商，这时 WebSocket 构造器方法的第二个可选参数就派上用场了，通过这个参数客户端和服务端就可以根据约定好的方式处理发送及接收到的数据。

WebSocket 构造器方法如下所示：

```
WebSocket WebSocket(
```

```
in DOMString url, // 表示要连接的 URL。这个 URL 应该为响应 WebSocket 的地址。

in optional DOMString protocols // 可以是一个单个的协议名字字符串或者包含多个协议名字字符串的数组。默认设为一个空字符串。
);
```

通过上述 WebSocket 构造器方法的第二个参数，客户端可以在初次连接握手时，可以告知服务器自己支持哪种协议。如下所示：

```
var ws = new WebSocket('wss://example.com/socket',['appProtocol', 'appProtocol-v2']);


ws.onopen = function () {
  if (ws.protocol == 'appProtocol-v2') {
    ...
  } else {
    ...
  }
}
```

如上所示，WebSocket 构造函数接受了一个可选的子协议名字的数组，通过这个数组，客户端可以向服务器通告自己能够理解或希望服务器接受的协议。当服务器接收到该请求后，会根据自身的支持情况，返回相应信息。

- 有支持的协议，则子协议协商成功，触发客户端的 onopen 回调，应用可以查询 WebSocket 对象上的 protocol 属性，从而得知服务器选定的协议；
- 没有支持的协议，则协商失败，触发 onerror 回调，连接断开。

1.3.WS 与 WSS

WebSocket 资源 URI 采用了自定义模式：ws 表示纯文本通信（如 ws://example.com/socket），wss 表示使用加密信道通信（TCP+TLS）。为什么不使用 http 而要自定义呢？

WebSocket 的主要目的，是在浏览器中的应用与服务器之间提供优化的、双向通信机制。可是，WebSocket 的连接协议也可以用于浏览器之外的场景，可以通过非 HTTP 协商机制交换数据。考虑到这一点，HyBi Working Group 就选择采用了自定义的 URI 模式：

- **ws 协议**：普通请求，占用与 http 相同的 80 端口；
- **wss 协议**：基于 SSL 的安全传输，占用与 tls 相同的 443 端口。

各自的 URI 如下：

```
ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]

wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]
```

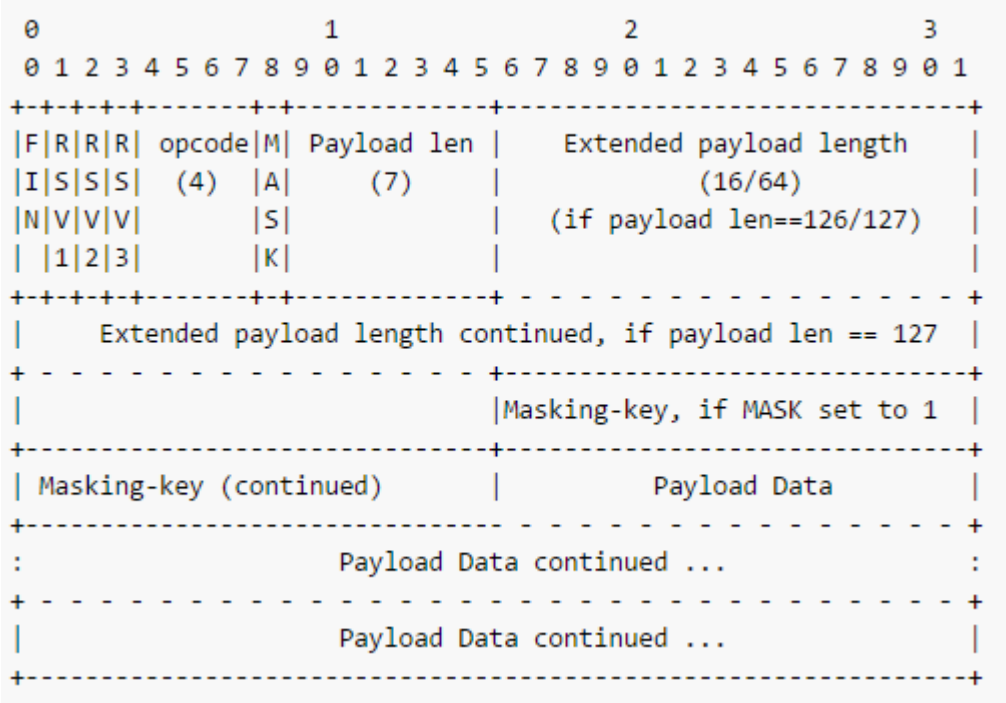
很多现有的 HTTP 中间设备可能不理解新的 WebSocket 协议，而这可能导致各种问题：盲目的连接升级、意外缓冲 WebSocket 帧、不明就里地修改内容、把 WebSocket 流量误当作不完整的 HTTP 通信，等等。这时 WSS 就提供了一种不错的解决方案，它建立一条端到端的安全通道，这个端到端的加密隧道对中间设备模糊了数据，因此中间设备就不能再感知到数据内容，也就无法再对请求做**特殊处理**。

2. WebSocket 协议

HyBi Working Group 制定的 WebSocket 通信协议（RFC 6455）包含两个高层组件：开放性 HTTP 握手用于协商连接参数，二进制消息分帧机制用于支持低开销的基于消息的文本和二进制数据传输。WebSocket 协议尝试在既有 HTTP 基础设施中实现双向 HTTP 通信，因此也使用 HTTP 的 80 和 443 端口。不过，这个设计不限于通过 HTTP 实现 WebSocket 通信，未来的实现可以在某个专用端口上使用更简单的握手，而不必重新定义一个协议。WebSocket 协议是一个独立完善的协议，可以在浏览器之外实现。不过，它的主要应用目标还是实现浏览器应用的双向通信。

2.1.数据成帧

WebSocket 使用了自定义的二进制分帧格式，把每个应用消息切分成一或多个帧，发送到目的地之后再组装起来，等到接收到完整的消息后再通知接收端。基本的成帧协议定义了帧类型有操作码、有效载荷的长度，指定位置的 Extension data 和 Application data，统称为 Payload data，保留了一些特殊位和操作码供后期扩展。在打开握手完成后，终端发送一个关闭帧之前的任何时间里，数据帧可能由客户端或服务器的任何一方发送。具体的帧格式如下所示：



- **FIN**： 1 bit 。表示此帧是否是消息的最后帧，第一帧也可能是最后帧。
- **RSV1，RSV2，RSV3**： 各 1 bit 。必须是 0，除非协商了扩展定义了非 0 的意义。
- **opcode**：4 bit。表示被传输帧的类型：x0 表示一个后续帧；x1 表示一个文本帧；x2 表示一个二进制帧；x3-7 为以后的非控制帧保留；x8 表示一个连接关闭；x9 表示一个 ping；xA 表示一个 pong；xB-F 为以后的控制帧保留。
- **Mask**： 1 bit。表示净荷是否有掩码（只适用于客户端发送给服务器的消息）。
- **Payload length**： 7 bit, 7 + 16 bit, 7 + 64 bit。 净荷长度由可变长度字段表示： 如果是 0~125，就是净荷长度；如果是 126，则接下来 2 字节表示的 16 位无符号整数才是这一帧的长度； 如果是 127，则接下来 8 字节表示的 64 位无符号整数才是这一帧的长度。
- **Masking-key**：0 或 4 Byte。用于给净荷加掩护，客户端到服务器标记。
- **Extension data**： x Byte。默认为 0 Byte，除非协商了扩展。
- **Application data**： y Byte。 在”Extension data”之后，占据了帧的剩余部分。
- **Payload data**：(x + y) Byte。”extension data” 后接 “application data”。

• **帧**：最小的通信单位，包含可变长度的帧首部和净荷部分，净荷可能包含完整或部分应用消息。
• **消息**：一系列帧，与应用消息对等。

是否把消息分帧由客户端和服务器实现决定，应用并不需要关注 WebSocket 帧和如何分帧，因为客户端（如浏览器）和服务端为完成该工作。那么客户端和服务端是按照什么规则进行分帧的呢？RFC 6455 规定的分帧规则如下：

1. 一个未分帧的消息包含单个帧，FIN 设置为 1，opcode 非 0。

2. 一个分帧了的消息包含：开始于：单个帧，FIN 设为 0，opcode 非 0；后接 ：0 个或多个帧，FIN 设为 0，opcode 设为 0；终结于：单个帧，FIN 设为 1，opcode 设为 0。一个分帧了消息在概念上等价于一个未分帧的大消息，它的有效载荷长度等于所有帧的有效载荷长度的累加；然而，有扩展时，这可能不成立，因为扩展定义了出现的 Extension data 的解释。例如，Extension data 可能只出现在第一帧，并用于后续的所有帧，或者 Extension data 出现于所有帧，且只应用于特定的那个帧。在缺少 Extension data 时，下面的示例示范了分帧如何工作。举例：如一个文本消息作为三个帧发送，第一帧的 opcode 是 0x1，FIN 是 0，第二帧的 opcode 是 0x0，FIN 是 0，第三帧的 opcode 是 0x0，FIN 是 1。

3. 控制帧可能被插入到分帧了消息中，控制帧必须不能被分帧。如果控制帧不能插入，例如，如果是在一个大消息后面，ping 的延迟将会很长。因此要求处理消息帧中间的控制帧。

4. 消息的帧必须以发送者发送的顺序传递给接受者。

5. 一个消息的帧必须不能交叉在其他帧的消息中，除非有扩展能够解释交叉。

6. 一个终端必须能够处理消息帧中间的控制帧。

7. 一个发送者可能对任意大小的非控制消息分帧。

8. 客户端和服务器必须支持接收分帧和未分帧的消息。

9. 由于控制帧不能分帧，中间设施必须不尝试改变控制帧。

10. 中间设施必须不修改消息的帧，如果保留位的值已经被使用，且中间设施不明白这些值的含义。

在遵循了上述分帧规则之后，一个消息的所有帧属于同样的类型，由第一个帧的 opcdoe 指定。由于控制帧不能分帧，消息的所有帧的类型要么是文本、二进制数据或保留的操作码中的一个。

虽然客户端和服务端都遵循同样的分帧规则，但也是有些差异的。在客户端往服务端发送数据时，为防止客户端中运行的恶意脚本对不支持 WebSocket 的中间设备进行缓存投毒攻击（cache poisoning attack），发送帧的净荷都要使用帧首部中指定的值加掩码。被标记的帧必须设置 MASK 域为 1，Masking-key 必须完整包含在帧里，它用于标记 Payload data。Masking-key 是由客户端随机选择的 32 位值，标记键应该是不可预测的，给定帧的 Masking-key 必须不能简单到服务器或代理可以预测 Masking-key 是用于一序列帧的，不可预测的 Masking-key 是阻止恶意应用的作者从 wire 上获取数据的关键。由于客户端发送到服务端的信息需要进行掩码处理，所以客户端发送数据的分帧开销要大于服

务端发送数据的开销，服务端的分帧开销是 2~10 Byte，客户端是则是 6~14 Byte。

控制帧

控制帧由操作码标识，操作码的最高位是 1。当前为控制帧定义的操作码有 0x8（关闭）、0x9（Ping）和 0xA（Pong），操作码 0xB-0xF 是保留的，未定义。控制帧用来交流 WebSocket 的状态，能够插入到消息的多个帧的中间。所有的控制帧必须有一个小于等于 125 字节的有效载荷长度，必须不能被分帧。

- **关闭**：操作码为 0x8。关闭帧可能包含一个主体（帧的应用数据部分）指明关闭的原因，如终端关闭，终端接收到的帧太大，或终端接收到的帧不符合终端的预期格式。从客户端发送到服务器的关闭帧必须标记，在发送关闭帧后，应用程序必须不再发送任何数据。如果终端接收到一个关闭帧，且先前没有发送关闭帧，终端必须发送一个关闭帧作为响应。终端可能延迟发送关闭帧，直到它的当前消息发送完成。在发送和接收到关闭消息后，终端认为 WebSocket 连接已关闭，必须关闭底层的 TCP 连接。服务器必须立即关闭底层的 TCP 连接；客户端应该等待服务器关闭连接，但并非必须等到接收关闭消息后才关闭，如果它在合理的时间间隔内没有收到反馈，也可以将 TCP 关闭。如果客户端和服务器同时发送关闭消息，两端都已发送和接收到关闭消息，应该认为 WebSocket 连接已关闭，并关闭底层 TCP 连接。

- **Ping**：操作码为 0x9。一个 Ping 帧可能包含应用程序数据。当接收到 Ping 帧，终端必须发送一个 Pong 帧响应，除非它已经接收到一个关闭帧。它应该尽快返回 Pong 帧作为响应。终端可能在连接建立后、关闭前的任意时间内发送 Ping 帧。注意：Ping 帧可作为 keepalive 或作为验证远程终端是否可响应的手段。

- **Pong**：操作码为 0xA。Pong 帧必须包含与被响应 Ping 帧的应用程序数据完全相同的数据。如果终端接收到一个 Ping 帧，且还没有对之前的 Ping 帧发送 Pong 响应，终端可能选择发送一个 Pong 帧给最近处理的 Ping 帧。一个 Pong 帧可能被主动发送，这作为单向心跳。对主动发送的 Pong 帧的响应是不希望的。

数据帧

数据帧携带需要发送的目标数据，由操作码标识，操作码的最高位是 0。当前为数据帧定义的（文本），0x2（二进制），操作码 0x3-0x7 为以后的非控制帧保留，未定义。

操作码决定了数据的解释：

- **文本**：操作码为 0x1。有效载荷数据是 UTF-8 编码的文本数据。特定的文本帧可能包含部分的 UTF-8 序列，然而，整个消息必须包含有效的 UTF-8，当终端以 UTF-8 解释字节流时发现字节流不是一个合法的 UTF-8 流，那么终端将关闭连接。

- **二进制**：操作码为 0x2。有效载荷数据是任意的二进制数据，它的解释由应用程序层唯一决定。

2.2.协议扩展

从上述的数据分帧格式可以知道，有很多扩展位预留，WebSocket 规范允许对协议进行扩展，可以使用这些预留位在基本的 WebSocket 分帧层之上实现更多的功能。

下面是负责制定 WebSocket 规范的 HyBi Working Group 进行的两项扩展：

- **多路复用扩展**（A Multiplexing Extension for WebSockets）：这个扩展可以将 WebSocket 的逻辑连接独立出来，实现共享底层的 TCP 连接。每个 WebSocket 连接都需要一个专门的 TCP 连接，这样效率很低。多路复用扩展解决了这个问题。它使用“信道 ID”扩展每个 WebSocket 帧，从而实现多个虚拟的 WebSocket 信道共享一个 TCP 连接。

- **压缩扩展**（Compression Extensions for WebSocket）：给 WebSocket 协议增加了压缩功能。基本的 WebSocket 规范没有压缩数据的机制或建议，每个帧中的净荷就是应用提供的净荷。虽然这对优化的二进制数据结构不是问题，但除非应用实现自己的压缩和解压缩逻辑，否则很多情况下都会造成传输载荷过大的问题。实际上，压缩扩展就相当于 HTTP 的传输编码协商。

要使用扩展，客户端必须在第一次的 Upgrade 握手中通知服务器，服务器必须选择并确认要在商定连接中使用的扩展。下面就是对升级协商的介绍。

2.3.升级协商

从上面的介绍可知，WebSocket 具有很大的灵活性，提供了很多强大的特性：基于消息的通信、自定义的二进制分帧层、子协议协商、可选的协议扩展等等。上面也讲到，客户端和服务端需先通过 HTTP 方式协商适当的参数后才可建立连接，完成协商之后，所有信息的发送和接收不再和 HTTP 相关，全由 WebSocket 自身的机制处理。当然，完成最初的连接参数协商并非必须使用 HTTP 协议，它只是一种实现方案，可以有其他选择。但使用 HTTP 协议完成最初的协商，有以下好处：让 WebSockets 与现有 HTTP 基础设施兼容：WebSocket 服务器可以运行在 80 和 443 端口上，这通常是对客户端唯一开放的端口；可以重用并扩展 HTTP 的 Upgrade 流，为其添加自定义的 WebSocket 首部，以完成协商。

在协商过程中，用到的一些头域如下：

- **Sec-WebSocket-Version**：客户端发送，表示它想使用的 WebSocket 协议版本（13 表示 RFC 6455）。如果服务器不支持这个版本，必须回应自己支持的版本。
- **Sec-WebSocket-Key**：客户端发送，自动生成的一个键，作为一个对服务器的“挑战”，以验证服务器支持请求的协议版本；
- **Sec-WebSocket-Accept**：服务器响应，包含 Sec-WebSocket-Key 的签名值，证明它支持请求的协议版本；
- **Sec-WebSocket-Protocol**：用于协商应用子协议：客户端发送支持的协议列表，服务器必须只回应一个协议名；
- **Sec-WebSocket-Extensions**：用于协商本次连接要使用的 WebSocket 扩展：客户端发送支持的扩展，服务器通过返回相同的首部确认自己支持一或多个扩展。

在进行 HTTP Upgrade 之前，客户端会根据给定的 URI、子协议、扩展和在浏览器情况下的 origin，先打开一个 TCP 连接，随后再发起升级协商。升级协商具体如下：

```
GET /socket HTTP/1.1 // 请求的方法必须是 GET，HTTP 版本必须至少是 1.1
Host: thirdparty.com
Origin: http://example.com
Connection: Upgrade
Upgrade: websocket // 请求升级到 WebSocket 协议
Sec-WebSocket-Version: 13 // 客户端使用的 WebSocket 协议版本
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ== // 自动生成的键，以验证服务器对协议的支持，其值必须是 nonce 组成的随机选择的 16 字节的被 base64 编码后的值
Sec-WebSocket-Protocol: appProtocol, appProtocol-v2 // 可选的应用指定的子协议列表
```

```
Sec-WebSocket-Extensions: x-webkit-deflate-message, x-custom-extension // 可选的客户端支持的协议扩展列表，指示了客户端希望使用的协议级别的扩展
```

在安全工程中，Nonce 是一个在加密通信只能使用一次的数字。在认证协议中，它往往是一个随机或伪随机数，以避免重放攻击。Nonce 也用于流密码以确保安全。如果需要使用相同的密钥加密一个以上的消息，就需要 Nonce 来确保不同的消息与该密钥加密的密钥流不同。

与浏览器中客户端发起的任何连接一样，WebSocket 请求也必须遵守同源策略：浏览器会自动在升级握手请求中追加 Origin 首部，远程服务器可能使用 CORS 判断接受或拒绝跨源请求。要完成握手，服务器必须返回一个成功的“Switching Protocols”（切换协议）响应，具体如下：

```
HTTP/1.1 101 Switching Protocols // 101 响应码确认升级到 WebSocket 协议
Upgrade: websocket
Connection: Upgrade

Access-Control-Allow-Origin: http://example.com // CORS 首部表示选择同意跨源连接

Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo= // 签名的键值验证协议支持

Sec-WebSocket-Protocol: appProtocol-v2 // 服务器选择的应用子协议

Sec-WebSocket-Extensions: x-custom-extension // 服务器选择的 WebSocket 扩展
```

所有兼容 RFC 6455 的 WebSocket 服务器都使用相同的算法计算客户端挑战的答案：将 Sec-WebSocket-Key 的内容与标准定义的唯一 GUID 字符串拼接起来，计算出 SHA1 散列值，结果是一个 base-64 编码的字符串，把这个字符串发给客户端即可。Sec-WebSocket-Accept 这个头域的 ABNF [RFC2616]定义如下：

```
   Sec-WebSocket-Accept = base64-value-non-empty

    base64-value-non-empty = (1*base64-data [ base64-padding ]) |
```

```
base64-padding

base64-data = 4base64-character

base64-padding = (2base64-character "==") |

(3base64-character "=")

base64-character = ALPHA | DIGIT | "+" | "/"
```

如果客户端发送的 key 值为：”dGhlIHNhbXBsZSBub25jZQ==”，服务端将把”258EAFA5-E914-47DA-95CA-C5AB0DC85B11” 这个唯一的 GUID 与它拼接起来，就是”dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CAC5AB0DC85B11”，然后对其进行 SHA-1 哈希，结果为”0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea”，再进行 base64-encoded 即可得”s3pPLMBiTxaQ9kYGzzhZRbK+xOo=”。

成功的 WebSocket 握手必须是客户端发送协议版本和自动生成的挑战值，服务器返回 101 HTTP 响应码（Switching Protocols）和散列形式的挑战答案，确认选择的协议版本。

一旦客户端打开握手发送出去，在发送任何数据之前，客户端必须等待服务器的响应。客户端必须按如下步骤验证响应：

1. 如果从服务器接收到的状态码不是 101，按 HTTP【RFC2616】程序处理响应。在特殊情况下，如果客户端接收到 401 状态码，可能执行认证；服务器可能用 3xx 状态码重定向客户端（但不要求客户端遵循他们）。否则按下面处理。

2. 如果响应缺失 Upgrade 头域或 Upgrade 头域的值没有包含大小写不敏感的 ASCII 值”websocket”，客户端必须使 WebSocket 连接失败。

3. 如果响应缺失 Connection 头域或其值不包含大小写不敏感的 ASCII 值”Upgrade”，客户端必须使 WebSocket 连接失败。

4. 如果响应缺失 Sec-WebSocket-Accept 头域或其值不包含 [Sec-WebSocket-Key]（作为字符串，非 base64 解码的）+ “258EAFA5-E914-47DA-95CA-C5AB0DC85B11” 的 base64 编码 SHA-1 值，客户端必须使 WebSocket 连接失败。

5. 如果响应包含 Sec-WebSocket-Extensions 头域，且其值指示使用的扩展不出现在客户端发送的握手（服务器指示的扩展不是客户端要求的），客户端必须使 WebSocket 连接失败。

6. 如果响应包含 Sec-WebSocket-Protocol 头域，且这个头域指示使用的子协议不包含在客户端的握手(服务器指示的子协议不是客户端要求的)，客户端必须使 WebSocket 连接失败。

如果客户端完成了对服务端响应的升级协商验证，该连接就可以用作双向通信信道交换 WebSocket 消息。从此以后，客户端与服务器之间不会再发生 HTTP 通信，一切由 WebSocket 协议接管。

3.使用场景及性能

Websocket 协议具有极简的 API，开发者可以很简便的调用，而且提供了二进制分帧、可扩展性以及子协议协商等强大特性，使得 WebSocket 成为在浏览器中采用自定义应用协议的最佳选择。但，在计算机世界里，任何技术和理论一般都是为解决特定问题而生的，并不是普世化的解决方案，WebSocket 亦是如此。WebSocket 不能取代 XHR 或 SSE，何时以及如何使用，毋庸置疑会对性能产生巨大影响，要获得最佳性能，我们必须善于利用它的长处！下面将对现有的一些协议与 WebSocket 对比进行一个大致介绍。

	XMLHttpRequest	Server-Sent Event	WebSocket
请求流	否	否	是
响应流	受限	是	是
分帧机制	HTTP	事件流	二进制分帧
二进制数据传输	是	否（base64）	是
压缩	是	是	受限
应用传输协议	HTTP	HTTP	WebSocket
网络传输协议	TCP	TCP	TCP

请求和响应流

XHR 是专门为“事务型”请求/ 响应通信而优化的：客户端向服务器发送完整的、格式良好的 HTTP 请求，服务器返回完整的响应。这里不支持请求流，在 Streams API 可用之前，没有可靠的跨浏览器响应流 API。 SSE 可以实现服务器到客户端的高效、低延迟的文本数据流：客户端发起 SSE 连接，服务器使用事件源协议将更新流式发送给客户端。客户端在初次握手后，不能向服务器发送任何数据。 WebSocket 是唯一一个能通过同一个 TCP 连接实现双向通信的机制，客户端和服务器随时可以交换数据。因此，WebSocket 在两个方向上都能保证文本和二进制应用数据的低延迟交付。

客户端到服务端传递消息的总时延由以下四个部分构成：

- **传播延迟**：消息从发送端到接收端需要的时间，是信号传播距离和速度的函数，传播时间取决于距离和信号通过的媒介，播速度通常不超过光速；
- **传输延迟**：把消息中的所有比特转移到链路中需要的时间，是消息长度和链路速率的函数，由传输链路的速率决
  定，与客户端到服务器的距离无关；

- **处理延迟**：处理分组首部、检查位错误及确定分组目标所需的时间，常由硬件完成，因此相应的延迟一般非常短；
- **排队延迟**：如果分组到达的速度超过了路由器的处理能力，那么分组就要在入站缓冲区排队，到来的分组排队等待处理的时间就是排队延迟。

无论是什么样的传输机制，都不会减少客户端与服务器间的往返次数，数据包的传播延迟都一样。但，采用不同的传输机制可以有不同的排队延迟。对 XHR 轮询而言，排队延迟就是客户端轮询间隔：服务器上的消息可用之后，必须等到下一次客户端 XHR 请求才能发送。相对来说，SSE 和 WebSocket 使用持久连接，这样服务器（和客户端——如果是 WebSocket）就可以在消息可用时立即发送它，消除了消息的排队延迟，也就使得总的传输延迟更小。

消息开销

在完成最初的升级协商之后，客户端和服务器即可通过 WebSocket 协议双向交换数据，消息分帧之后每帧会添加 2~14 字节的开销；SSE 会给每个 消息添加 5 字节，但仅限于 UTF-8 内容(SSE 不是为传输二进制载荷而设计的！如果有必要，可以把二进制对象编码为 base64 形式，然后再使用 SSE)；HTTP 1.x 请求(XHR 及其他常规请求)会携带 500~800 字节的 HTTP 元数据，加上 cookie； HTTP 2.0 压缩 HTTP 元数据，可以显著减少开销，如果请求都不修改首部，那么开销可以低至 8 字节。WebSocket 专门为双向通信而设计，开销很小，在实时通知应用开发中是不错的选择。

> 上述开销不包括 IP、TCP 和 TLS 分帧的开销，后者一共会给每个消息增加 60~100 字节，无论使用的是什么应用协议。

效率及压缩

在使用 HTTP 协议传输数据时，每个请求都可以协商最优的传输编码格式（如对文本数据采用 gzip 压缩）；SSE 只能传输 UTF-8 格式数据，事件流数据可以在整个会话期间使用 gzip 压缩；WebSocket 可以传输文本和二进制数据，压缩整个会话行不通，二进制的净荷也可能已经压缩过了！

鉴于 WebSocket 的特殊性，它需要实现自己的压缩机制，并针对每个消息选择应用。HyBi 工作组正在为 WebSocket 协议制定以消息为单位的压缩扩展，但这个扩展尚未得到任何浏览器支持。目前来说，除非应用通过细致优化自己的二进制净荷实现自己的压缩逻辑，同时也针对文本消息实现自己的压缩逻辑，否则传输数据过程中一定会产生很大的字节开销！

自定义应用协议

HTTP 已经诞生了数十年，具有广泛的应用，各种优化专门的优化机制也已经被浏览器及服务器等设备实施，XHR 请求自然而然就继承了所有这些功能。然

而，对于只使用 HTTP 协议完成升级协商的 WebSocket 来说，流式数据处理可以让我们在客户端和服务器间自定义协议，但也会错过浏览器提供的很多服务，应用可能必须实现自己的逻辑来填充某些功能空白，比如缓存、状态管理、元数据交付等等。

部署 WebSocket

HTTP 是专为短时突发性传输设计的，很多服务器、代理和其他中间设备的 HTTP 连接空闲超时设置都很激进。这就与 WebSocket 的长时连接、实时双向通信相悖，部署时需要关注下面的三个方面：

- 位于各自网络中的路由器、负载均衡器和代理；
- 外部网络中透明、确定的代理服务器（如 ISP 和运营商的代理）；
- 客户网络中的路由器、防火墙和代理。

鉴于用户所处的网络环境是各不相同的，不受开发者所控制。某些网络甚至会完全屏蔽 WebSocket 通信，有些设备也不支持 WebSocket 协议，这时就需要采用备用机制，使用其他技术来实现类似与 WebSocket 的通信(如 socket.io 等)。虽然，我们无法处理网络中的中间设备，但对于处在我们自己掌控下的基础设施还是可以做一些工作的，可以对通信路径上的每一台负载均衡器、路由器和 Web 服务器针对长时连接进行调优。然而，长时连接和空闲会话会占用所有中间设备及服务器的内存和套接字资源，开销很大，部署 WebSocket、SSE 及 HTTP 2.0 等赖于长时会话的协议都会对运维提出新的挑战。在使用 WebSocket 的过程中，也需要做到优化二进制净荷和压缩 UTF-8 内容以最小化传输数据、监控客户端缓冲数据的量、切分应用消息避免队首阻塞、合用的情况下利用其他传输机制等。

总结

WebSocket 协议为实时双向通信而设计，提供高效、灵活的文本和二进制数据传输，同时也错过了浏览器为 HTTP 提供的一些服务，在使用时需要应用自己实现。在进行应用数据传输时，需要根据不同的场景选择恰当的协议，WebSocket 并不能取代 HTTP、XHR 或 SSE，关键还是要利用这些机制的长处以求得最佳性能。

Socket.IO

鉴于现在不同的平台及浏览器版本对 WebSocket 支持的不同，有开发者做了一个叫做 socket.io 的为实时应用提供跨平台实时通信的库，我们可以使用它完成向 WebSocket 的切换。socket.io 旨在使实时应用在每个浏览器和移动设备上成为可能，模糊不同的传输机制之间的差异。socket.io 的名字源于它使用了浏览器支持并采用的 HTML5 WebSocket 标准，因为并不是所有的浏览器都支持 WebSocket ，所以该库支持一系列降级功能：

- Websocket
- Adobe:registered: Flash:registered: Socket
- AJAX long polling
- AJAX multipart streaming
- Forever Iframe
- JSONP Polling

在大部分情境下，你都能通过这些功能选择与浏览器保持类似长连接的功能。具体细节请看 Socket.io。