# WebSocket : 从入门到精通

# 一、内容概览

WebSocket的出现，使得浏览器具备了实时双向通信的能力。本文由浅入深，介绍了WebSocket如何建立连接、交换数据的细节，以及数据帧的格式。此外，还简要介绍了针对WebSocket的安全攻击，以及协议是如何抵御类似攻击的。

# 二、什么是WebSocket

HTML5开始提供的一种浏览器与服务器进行全双工通讯的网络技术，属于应用层协议。它基于TCP传输协议，并复用HTTP的握手通道。

对大部分web开发者来说，上面这段描述有点枯燥，其实只要记住几点：

1. WebSocket可以在浏览器里使用
2. 支持双向通信
3. 使用很简单

## 1、有哪些优点

说到优点，这里的对比参照物是HTTP协议，概括地说就是：支持双向通信，更灵活，更高效，可扩展性更好。

1. 支持双向通信，实时性更强。
2. 更好的二进制支持。
3. 较少的控制开销。连接创建后，ws客户端、服务端进行数据交换时，协议控制的数据包头部较小。在不包含头部的情况下，服务端到客户端的包头只有2~10字节（取决于数据包长度），客户端到服务端的的话，需要加上额外的4字节的掩码。而HTTP协议每次通信都需要携带完整的头部。
4. 支持扩展。ws协议定义了扩展，用户可以扩展协议，或者实现自定义的子协议。（比如支持自定义压缩算法等）

对于后面两点，没有研究过WebSocket协议规范的同学可能理解起来不够直观，但不影响对WebSocket的学习和使用。

## 2、需要学习哪些东西

对网络应用层协议的学习来说，最重要的往往就是**连接建立过程**、**数据交换教程**。当然，数据的格式是逃不掉的，因为它直接决定了协议本身的能力。好的数据格式能让协议更高效、扩展性更好。

下文主要围绕下面几点展开：

1. 如何建立连接
2. 如何交换数据
3. 数据帧格式
4. 如何维持连接

# 三、入门例子

在正式介绍协议细节前，先来看一个简单的例子，有个直观感受。例子包括了WebSocket服务端、WebSocket客户端（网页端）。完整代码可以在 这里 找到。

这里服务端用了`ws`这个库。相比大家熟悉的`socket.io`，`ws`实现更轻量，更适合学习的目的。

## 1、服务端

代码如下，监听8080端口。当有新的连接请求到达时，打印日志，同时向客户端发送消息。当收到到来自客户端的消息时，同样打印日志。

```
var app = require('express')();
    var server = require('http').Server(app);
    var WebSocket = require('ws');

    var wss = new WebSocket.Server({ port: 8080 });

    wss.on('connection', functionconnection(ws) {
        console.log('server: receive connection.');

        ws.on('message', functionincoming(message) {
            console.log('server: received: %s', message);
        });

        ws.send('world');
    });

    app.get('/', function (req, res) {
      res.sendfile(__dirname + '/index.html');
    });

    app.listen(3000);
```

## 2、客户端

代码如下，向8080端口发起WebSocket连接。连接建立后，打印日志，同时向服务端发送消息。接收到来自服务端的消息后，同样打印日志。

```
  <script>var ws = new WebSocket('ws://localhost:8080');
      ws.onopen = function () {
        console.log('ws onopen');
        ws.send('from client: hello');
      };
      ws.onmessage = function (e) {
        console.log('ws onmessage');
        console.log('from server: ' + e.data);
      };
    </script>
```

## 3、运行结果

可分别查看服务端、客户端的日志，这里不展开。

服务端输出：

```
    server: receive connection.
    server: received hello
```

客户端输出：

```
    client: ws connection is open
    client: received world
```

# 四、如何建立连接

前面提到，WebSocket复用了HTTP的握手通道。具体指的是，客户端通过HTTP请求与WebSocket服务端协商升级协议。协议升级完成后，后续的数据交换则遵照WebSocket的协议。

## 1、客户端：申请协议升级

首先，客户端发起协议升级请求。可以看到，采用的是标准的HTTP报文格式，且只支持`GET`方法。

```
    GET/ HTTP/1.1
    Host: localhost:8080
    Origin: http://127.0.0.1:3000
    Connection: Upgrade
    Upgrade: websocket
    Sec-WebSocket-Version: 13
    Sec-WebSocket-Key: w4v7O6xFTi36lq3RNcgctw==
```

重点请求首部意义如下：

- `Connection: Upgrade`：表示要升级协议
- `Upgrade: websocket`：表示要升级到websocket协议。
- `Sec-WebSocket-Version: 13`：表示websocket的版本。如果服务端不支持该版本，需要返回一个`Sec-WebSocket-Version`header，里面包含服务端支持的版本号。
- `Sec-WebSocket-Key`：与后面服务端响应首部的`Sec-WebSocket-Accept`是配套的，提供基本的防护，比如恶意的连接，或者无意的连接。

> 注意，上面请求省略了部分非重点请求首部。由于是标准的HTTP请求，类似Host、Origin、Cookie等请求首部会照常发送。在握手阶段，可以通过相关请求首部进行 安全限制、权限校验等。

## 2、服务端：响应协议升级

服务端返回内容如下，状态代码`101`表示协议切换。到此完成协议升级，后续的数据交互都按照新的协议来。

```
    HTTP/1.1101 Switching Protocols
    Connection:Upgrade
    Upgrade: websocket
    Sec-WebSocket-Accept: Oy4NRAQ13jhfONC7bP8dTKb4PTU=
```

> 备注：每个header都以`\r\n`结尾，并且最后一行加上一个额外的空行`\r\n`。此外，服务端回应的HTTP状态码只能在握手阶段使用。过了握手阶段后，就只能采用特定的错误码。

## 3、Sec-WebSocket-Accept的计算

`Sec-WebSocket-Accept`根据客户端请求首部的`Sec-WebSocket-Key`计算出来。

计算公式为：

1. 将`Sec-WebSocket-Key`跟`258EAFA5-E914-47DA-95CA-C5AB0DC85B11`拼接。
2. 通过SHA1计算出摘要，并转成base64字符串。

伪代码如下：

```
    >toBase64( sha1( Sec-WebSocket-Key + 258EAFA5-E914-47DA-95CA-C5AB0DC85B11
)  )
```

验证下前面的返回结果：

```
const crypto = require('crypto');
    const magic = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11';
    const secWebSocketKey = 'w4v7O6xFTi36lq3RNcgctw==';

    let secWebSocketAccept = crypto.createHash('sha1')
    .update(secWebSocketKey + magic)
    .digest('base64');

    console.log(secWebSocketAccept);
    // Oy4NRAQ13jhfONC7bP8dTKb4PTU=
```

# 五、数据帧格式

客户端、服务端数据的交换，离不开数据帧格式的定义。因此，在实际讲解数据交换之前，我们先来看下WebSocket的数据帧格式。

WebSocket客户端、服务端通信的最小单位是帧（frame），由1个或多个帧组成一条完整的消息（message）。

1. 发送端：将消息切割成多个帧，并发送给服务端；
2. 接收端：接收消息帧，并将关联的帧重新组装成完整的消息；

本节的重点，就是讲解**数据帧**的格式。详细定义可参考 RFC6455 5.2节 。

## 1、数据帧格式概览

下面给出了WebSocket数据帧的统一格式。熟悉TCP/IP协议的同学对这样的图应该不陌生。

1. 从左到右，单位是比特。比如`FIN`、`RSV1`各占据1比特，`opcode`占据4比特。
2. 内容包括了标识、操作代码、掩码、数据、数据长度等。（下一小节会展开）

```
0123
      01234567890123456789012345678901
     +-+-+-+-+-------+-+-------------+-------------------------------+
     |F|R|R|R| opcode|M| Payload len |    Extended payload length    |
     |I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
     |N|V|V|V|       |S|             |   (if payload len==126/127)   |
     | |1|2|3|       |K|             |                               |
     +-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
     |     Extended payload length continued, if payload len == 127  |
     + - - - - - - - - - - - - - - - +-------------------------------+
     |                               |Masking-key, if MASK set to 1  |
     +-------------------------------+-------------------------------+
     | Masking-key (continued)       |          Payload Data         |
     +-------------------------------- - - - - - - - - - - - - - - - +
     :                     Payload Data continued ...                :
     + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
     |                     Payload Data continued ...                |
     +---------------------------------------------------------------+
```

## 2、数据帧格式详解

针对前面的格式概览图，这里逐个字段进行讲解，如有不清楚之处，可参考协议规范，或留言交流。

**FIN**：1个比特。

如果是1，表示这是消息（message）的最后一个分片（fragment），如果是0，表示不是是消息（message）的最后一个分片（fragment）。

**RSV1, RSV2, RSV3**：各占1个比特。

一般情况下全为0。当客户端、服务端协商采用WebSocket扩展时，这三个标志位可以非0，且值的含义由扩展进行定义。如果出现非零的值，且并没有采用WebSocket扩展，连接出错。

**Opcode**：4个比特。

操作代码，Opcode的值决定了应该如何解析后续的数据载荷（data payload）。如果操作代码是不认识的，那么接收端应该断开连接（fail the connection）。可选的操作代码如下：

- %x0：表示一个延续帧。当Opcode为0时，表示本次数据传输采用了数据分片，当前收到的数据帧为其中一个数据分片。
- %x1：表示这是一个文本帧（frame）
- %x2：表示这是一个二进制帧（frame）
- %x3-7：保留的操作代码，用于后续定义的非控制帧。
- %x8：表示连接断开。
- %x9：表示这是一个ping操作。
- %xA：表示这是一个pong操作。
- %xB-F：保留的操作代码，用于后续定义的控制帧。

**Mask**：1个比特。

表示是否要对数据载荷进行掩码操作。从客户端向服务端发送数据时，需要对数据进行掩码操作；从服务端向客户端发送数据时，不需要对数据进行掩码操作。

如果服务端接收到的数据没有进行过掩码操作，服务端需要断开连接。

如果Mask是1，那么在Masking-key中会定义一个掩码键（masking key），并用这个掩码键来对数据载荷进行反掩码。所有客户端发送到服务端的数据帧，Mask都是1。

掩码的算法、用途在下一小节讲解。

**Payload length**：数据载荷的长度，单位是字节。为7位，或7+16位，或1+64位。

假设数Payload length === x，如果

- x为0~126：数据的长度为x字节。
- x为126：后续2个字节代表一个16位的无符号整数，该无符号整数的值为数据的长度。
- x为127：后续8个字节代表一个64位的无符号整数（最高位为0），该无符号整数的值为数据的长度。

此外，如果payload length占用了多个字节的话，payload length的二进制表达采用网络序（big endian，重要的位在前）。

**Masking-key**：0或4字节（32位）

所有从客户端传送到服务端的数据帧，数据载荷都进行了掩码操作，Mask为1，且携带了4字节的Masking-key。如果Mask为0，则没有Masking-key。

备注：载荷数据的长度，不包括mask key的长度。

**Payload data**：(x+y) 字节

载荷数据：包括了扩展数据、应用数据。其中，扩展数据x字节，应用数据y字节。

扩展数据：如果没有协商使用扩展的话，扩展数据数据为0字节。所有的扩展都必须声明扩展数据的长度，或者可以如何计算出扩展数据的长度。此外，扩展如何使用必须在握手阶段就协商好。如果扩展数据存在，那么载荷数据长度必须将扩展数据的长度包含在内。

应用数据：任意的应用数据，在扩展数据之后（如果存在扩展数据），占据了数据帧剩余的位置。载荷数据长度 减去 扩展数据长度，就得到应用数据的长度。

## 3、掩码算法

掩码键（Masking-key）是由客户端挑选出来的32位的随机数。掩码操作不会影响数据载荷的长度。掩码、反掩码操作都采用如下算法：

首先，假设：

- original-octet-i：为原始数据的第i字节。
- transformed-octet-i：为转换后的数据的第i字节。
- j：为`i mod 4`的结果。
- masking-key-octet-j：为mask key第j字节。

算法描述为：original-octet-i 与 masking-key-octet-j 异或后，得到 transformed-octet-i。

> j = i MOD 4transformed-octet-i = original-octet-i XOR masking-key-octet-j

# 六、数据传递

一旦WebSocket客户端、服务端建立连接后，后续的操作都是基于数据帧的传递。

WebSocket根据`opcode`来区分操作的类型。比如`0x8`表示断开连接，`0x0`-`0x2`表示数据交互。

## 1、数据分片

WebSocket的每条消息可能被切分成多个数据帧。当WebSocket的接收方收到一个数据帧时，会根据`FIN`的值来判断，是否已经收到消息的最后一个数据帧。

FIN=1表示当前数据帧为消息的最后一个数据帧，此时接收方已经收到完整的消息，可以对消息进行处理。FIN=0，则接收方还需要继续监听接收其余的数据帧。

此外，`opcode`在数据交换的场景下，表示的是数据的类型。`0x01`表示文本，`0x02`表示二进制。而`0x00`比较特殊，表示延续帧（continuation frame），顾名思义，就是完整消息对应的数据帧还没接收完。

## 2、数据分片例子

直接看例子更形象些。下面例子来自MDN，可以很好地演示数据的分片。客户端向服务端两次发送消息，服务端收到消息后回应客户端，这里主要看客户端往服务端发送的消息。

**第一条消息**

FIN=1, 表示是当前消息的最后一个数据帧。服务端收到当前数据帧后，可以处理消息。opcode=0x1，表示客户端发送的是文本类型。

**第二条消息**

1. FIN=0，opcode=0x1，表示发送的是文本类型，且消息还没发送完成，还有后续的数据帧。
2. FIN=0，opcode=0x0，表示消息还没发送完成，还有后续的数据帧，当前的数据帧需要接在上一条数据帧之后。

3. FIN=1，opcode=0x0，表示消息已经发送完成，没有后续的数据帧，当前的数据帧需要接在上一条数据帧之后。服务端可以将关联的数据帧组装成完整的消息。

```
    Client: FIN=1, opcode=0x1, msg="hello"
    Server: (process complete message immediately) Hi.
    Client: FIN=0, opcode=0x1, msg="and a"
    Server: (listening, new message containing text started)
    Client: FIN=0, opcode=0x0, msg="happy new"
    Server: (listening, payload concatenated to previous message)
    Client: FIN=1, opcode=0x0, msg="year!"
    Server: (process complete message) Happy new year to you too!
```

# 七、连接保持+心跳

WebSocket为了保持客户端、服务端的实时双向通信，需要确保客户端、服务端之间的TCP通道保持连接没有断开。然而，对于长时间没有数据往来的连接，如果依旧长时间保持着，可能会浪费包括的连接资源。

但不排除有些场景，客户端、服务端虽然长时间没有数据往来，但仍需要保持连接。这个时候，可以采用心跳来实现。

- 发送方->接收方：ping
- 接收方->发送方：pong

ping、pong的操作，对应的是WebSocket的两个控制帧，`opcode`分别是`0x9`、`0xA`。

举例，WebSocket服务端向客户端发送ping，只需要如下代码（采用`ws`模块）

```
    ws.ping('', false, true);
```

# 八、Sec-WebSocket-Key/Accept的作用

前面提到了，`Sec-WebSocket-Key/Sec-WebSocket-Accept`在主要作用在于提供基础的防护，减少恶意连接、意外连接。

作用大致归纳如下：

1. 避免服务端收到非法的websocket连接（比如http客户端不小心请求连接websocket服务，此时服务端可以直接拒绝连接）
2. 确保服务端理解websocket连接。因为ws握手阶段采用的是http协议，因此可能ws连接是被一个http服务器处理并返回的，此时客户端可以通过Sec-WebSocket-Key来确保服务端认识ws协议。（并非百分百保险，比如总是存在那么些无聊的http服务器，光处理Sec-WebSocket-Key，但并没有实现ws协议。。。）
3. 用浏览器里发起ajax请求，设置header时，Sec-WebSocket-Key以及其他相关的header是被禁止的。这样可以避免客户端发送ajax请求时，意外请求协议升级（websocket upgrade）
4. 可以防止反向代理（不理解ws协议）返回错误的数据。比如反向代理前后收到两次ws连接的升级请求，反向代理把第一次请求的返回给cache住，然后第二次请求到来时直接把cache住的请求给返回（无意义的返回）。
5. Sec-WebSocket-Key主要目的并不是确保数据的安全性，因为Sec-WebSocket-Key、Sec-WebSocket-Accept的转换计算公式是公开的，而且非常简单，最主要的作用是预防一些常见的意外情况（非故意的）。

> 强调：Sec-WebSocket-Key/Sec-WebSocket-Accept 的换算，只能带来基本的保障，但连接是否安全、数据是否安全、客户端/服务端是否合法的 ws客户端、ws服务端，其实并没有实际性的保证。

# 九、数据掩码的作用

WebSocket协议中，数据掩码的作用是增强协议的安全性。但数据掩码并不是为了保护数据本身，因为算法本身是公开的，运算也不复杂。除了加密通道本身，似乎没有太多有效的保护通信安全的办法。

那么为什么还要引入掩码计算呢，除了增加计算机器的运算量外似乎并没有太多的收益（这也是不少同学疑惑的点）。

答案还是两个字：**安全**。但并不是为了防止数据泄密，而是为了防止早期版本的协议中存在的代理缓存污染攻击（proxy cache poisoning attacks）等问题。

## 1、代理缓存污染攻击

下面摘自2010年关于安全的一段讲话。其中提到了代理服务器在协议实现上的缺陷可能导致的安全问题。猛击出处。

> “We show, empirically, that the current version of the WebSocket consent mechanism is vulnerable to proxy cache poisoning attacks. Even though the WebSocket handshake is based on HTTP, which should be understood by most network intermediaries, the handshake uses the esoteric “Upgrade” mechanism of HTTP [5]. In our experiment, we find that many proxies do not implement the Upgrade mechanism properly, which causes the handshake to succeed even though subsequent traffic over the socket will be misinterpreted by the proxy.”
>
> [TALKING] Huang, L-S., Chen, E., Barth, A., Rescorla, E., and C.Jackson, "Talking to Yourself for Fun and Profit", 2010,

在正式描述攻击步骤之前，我们假设有如下参与者：

- 攻击者、攻击者自己控制的服务器（简称“邪恶服务器”）、攻击者伪造的资源（简称“邪恶资源”）
- 受害者、受害者想要访问的资源（简称“正义资源”）
- 受害者实际想要访问的服务器（简称“正义服务器”）
- 中间代理服务器

攻击步骤一：

1. **攻击者**浏览器 向 **邪恶服务器** 发起WebSocket连接。根据前文，首先是一个协议升级请求。
2. 协议升级请求 实际到达 **代理服务器。**
3. **代理服务器** 将协议升级请求转发到 **邪恶服务器。**
4. **邪恶服务器** 同意连接，**代理服务器** 将响应转发给 **攻击者。**

由于 upgrade 的实现上有缺陷，**代理服务器** 以为之前转发的是普通的HTTP消息。因此，当**协议服务器** 同意连接，**代理服务器** 以为本次会话已经结束。

攻击步骤二：

1. **攻击者** 在之前建立的连接上，通过WebSocket的接口向 **邪恶服务器** 发送数据，且数据是精心构造的HTTP格式的文本。其中包含了 **正义资源** 的地址，以及一个伪造的host（指向**正义服务器**）。（见后面报文）
2. 请求到达 **代理服务器** 。虽然复用了之前的TCP连接，但 **代理服务器** 以为是新的HTTP请求。
3. **代理服务器** 向 **邪恶服务器** 请求 **邪恶资源。**
4. **邪恶服务器** 返回 **邪恶资源。代理服务器** 缓存住 **邪恶资源**（url是对的，但host是 **正义服务器** 的地址）。

到这里，受害者可以登场了：

1. **受害者** 通过 **代理服务器** 访问 **正义服务器** 的 **正义资源。**
2. **代理服务器** 检查该资源的url、host，发现本地有一份缓存（伪造的）。

3. **代理服务器** 将 **邪恶资源** 返回给 **受害者。**
4. **受害者** 卒。

附：前面提到的精心构造的“HTTP请求报文”。

```
    Client → Server:
    POST /path/of/attackers/choice HTTP/1.1 Host: host-of-attackers-
choice.com Sec-WebSocket-Key: <connection-key>
    Server → Client:
    HTTP/1.1 200 OK
    Sec-WebSocket-Accept: <connection-key>
```

## 2、当前解决方案

最初的提案是对数据进行加密处理。基于安全、效率的考虑，最终采用了折中的方案：对数据载荷进行掩码处理。

需要注意的是，这里只是限制了浏览器对数据载荷进行掩码处理，但是坏人完全可以实现自己的WebSocket客户端、服务端，不按规则来，攻击可以照常进行。

但是对浏览器加上这个限制后，可以大大增加攻击的难度，以及攻击的影响范围。如果没有这个限制，只需要在网上放个钓鱼网站骗人去访问，一下子就可以在短时间内展开大范围的攻击。

# 十、写在后面

WebSocket可写的东西还挺多，比如WebSocket扩展。客户端、服务端之间是如何协商、使用扩展的。WebSocket扩展可以给协议本身增加很多能力和想象空间，比如数据的压缩、加密，以及多路复用等。