

daipeng7 Lv2

2017年12月07日 阅读 9316

关注

WebSocket探秘

首先

长连接：一个连接上可以连续发送多个数据包，在连接期间，如果没有数据包发送，需要双方发链路检查包。

TCP/IP：TCP/IP属于传输层，主要解决数据在网络中的传输问题，只管传输数据。但是那样对传输的数据没有一个规范的封装、解析等处理，使得传输的数据就很难识别，所以才有了应用层协议对数据的封装、解析等，如HTTP协议。

HTTP：HTTP是应用层协议，封装解析传输的数据。从HTTP1.1开始其实就默认开启了长连接，也就是请求header中看到的Connection:Keep-alive。但是这个长连接只是说保持了（服务器可以告诉客户端保持时间Keep-Alive:timeout=200;max=20;）这个TCP通道，直接Request - Response，而不需要再创建一个连接通道，做到了一个性能优化。但是HTTP通讯本身还是Request - Response。

socket：与HTTP不一样，socket不是协议，它是在程序层面上对传输层协议（可以主要理解为TCP/IP）的接口封装。我们知道传输层的协议，是解决数据在网络中传输的，那么socket就是传输通道两端的接口。所以对于前端而言，socket也可以简单的理解为对TCP/IP的抽象协议。

WebSocket：WebSocket是包装成了一个应用层协议作为socket,从而能够让客户端和远程服务端通过web建立全双工通信。websocket提供ws和wss两种URL方案。[协议英文文档](#)和[中文翻译](#)

WebSocket API

使用WebSocket构造函数创建一个WebSocket连接，返回一个websocket实例。通过这个实例我们可以监听事件，这些事件可以知道什么时候简历连接，什么时候有消息被推过来了，什么时候发生错误了，时候连接关闭。我们可以使用node搭建一个WebSocket服务器来看看，[github](#)。同样也可以调用[websocket.org](#)网站的demo服务器[demos.kaazing.com/echo/](#)。





//创建WebSocket实例，可以使用ws和wss。第二个参数可以选填自定义协议，如果多协议，可以以数组方式

```
var socket = new WebSocket('ws://demos.kaazing.com/echo');
```

• open

服务器相应WebSocket连接请求触发

```
socket.onopen = (event) => {  
    socket.send('Hello Server!');  
};
```

• message

服务器有 响应数据 触发

```
socket.onmessage = (event) => {  
    debugger;  
    console.log(event.data);  
};
```

• error

出错时触发，并且会关闭连接。这时可以根据错误信息进行按需处理

```
socket.onerror = (event) => {  
    console.log('error');  
}
```

• close

连接关闭时触发，这在两端都可以关闭。另外如果连接失败也是会触发的。
针对关闭一般我们会做一些异常处理,关于异常参数:

1. socket.readyState

2 正在关闭 3 已经关闭

2. event.wasClean [Boolean]

true 客户端或者服务器端调用close主动关闭

false 反之

3. event.code [Number] 关闭连接的状态码。socket.close(code, reason)





```
socket.onclose = (event) => {  
  debugger;  
}
```

方法

- **send**

send(data) 发送方法 data 可以是String/Blob/ArrayBuffer/ByteBuffer等

需要注意,使用send发送数据, 必须是连接建立之后。一般会在onopen事件触发后发送:

```
socket.onopen = (event) => {  
  socket.send('Hello Server!');  
};
```

如果是需要去响应别的事件再发送消息, 也就是将WebSocket实例socket交给别的方法使用, 因为在发送时你不一定知道socket是否还连接着, 所以可以检查readyState属性的值是否等于OPEN常量, 也就是查看socket是否还连接着。

```
btn.onclick = function startSocket(){  
  //判断是否连接是否还存在  
  if(socket.readyState == WebSocket.OPEN){  
    var message = document.getElementById("message").value;  
    if(message != "") socket.send(message);  
  }  
}
```

- **close**

使用close([code[,reason]])方法可以关闭连接。code和reason均为选填

```
// 正常关闭  
socket.close(1000, "closing normally");
```

常量



CONNECTING	0	连接还未开启
OPEN	1	连接开启可以通信
CLOSING	2	连接正在关闭中
CLOSED	3	连接已经关闭

属性

属性名	值类型	描述
binaryType	String	表示连接传输的二进制数据类型的字符串。默认为"blob"。
bufferedAmount	Number	只读。如果使用send()方法发送的数据过大，虽然send()方法会马上执行，但数据并不是马上传输。浏览器会缓存应用流出的数据，你可以使用bufferedAmount属性检查已经进入队列但还未被传输的数据大小。在一定程度上可以避免网络饱和。
protocol	String/Array	在构造函数中，protocol参数让服务端知道客户端使用的WebSocket协议。而在实例socket中就是连接建立前为空，连接建立后为客户端和服务端确定下来的协议名称。
readyState	String	只读。连接当前状态，这些状态是与常量相对应的。
extensions	String	服务器选择的扩展。目前，这只是一个空字符串或通过连接协商的扩展列表。

WebSocket简单实现

WebSocket 协议有两部分：握手、数据传输。

其中，握手无疑是关键，是一切的先决条件。





1. 客户端握手请求

//创建WebSocket实例，可以使用ws和wss。第二个参数可以选填自定义协议，如果多协议，可以以数组方式

```
var socket = new WebSocket('ws://localhost:8081', [protocol]);
```

出于WebSocket的产生原因是为了浏览器能实现同服务器的全双工通信和HTTP协议在浏览器端的广泛运用（当然也不全是为了浏览器，但是主要还是针对浏览器的）。所以WebSocket的握手是HTTP请求的升级。WebSocket客户端请求头示例：

```
GET /chat HTTP/1.1    //必需。
Host: server.example.com // 必需。WebSocket服务器主机名
Upgrade: websocket // 必需。并且值为" websocket"。有个空格
Connection: Upgrade // 必需。并且值为" Upgrade"。有个空格
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ== // 必需。其值采用base64编码的随机16字节长的字符串
Origin: http://example.com //浏览器必填。头域（RFC6454）用于保护WebSocket服务器不被未授权的运行
Sec-WebSocket-Protocol: chat, superchat //选填。可用选项有子协议选择器。
Sec-WebSocket-Version: 13 //必需。版本。
```

WebSocket客户端将上述请求发送到服务器。如果是调用浏览器的WebSocket API,浏览器会自动完成完成上述请求头。

2. 服务端握手响应

服务器得向客户端证明它接收到了客户端的WebSocket握手，为使服务器不接受非WebSocket连接，防止攻击者通过XMLHttpRequest发送或表单提交精心构造的包来欺骗WebSocket服务器。服务器把两块信息合并来形成响应。第一块信息来自客户端握手头域Sec-WebSocket-Key，如Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==。对于这个头域，服务器取头域的值（需要先消除空白符），以字符串的形式拼接全局唯一的（GUID，[RFC4122]）标识：258EAF5E914-47DA-95CA-C5AB0DC85B11，此值不大可能被不明白WebSocket协议的网络终端使用。然后进行SHA-1 hash（160位）编码，再进行base64编码，将结果作为服务器的握手返回。具体如下：

请求头：Sec-WebSocket-Key:dGhlIHNhbXBsZSBub25jZQ==

取值，字符串拼接后得到："dGhlIHNhbXBsZSBub25jZQ==258EAF5E914-47DA-95CA-C5AB0DC85B11";

SHA-1后得到： 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x5

Base64后得到： s3pPLMBiTxaQ9kYGzzhZRbK+x0o=





最终形成WebSocket服务器端的握手响应：

HTTP/1.1 101 Switching Protocols //必需。响应头。状态码为101。任何非101的响应都为握手未完成。
Upgrade: websocket //必需。升级类型。
Connection: Upgrade //必需。本次连接类型为升级。
Sec-WebSocket-Accept: s3pLMBiTxq9kYGzzhZRbK+x0o= //必需。表明服务器是否愿意接受连接。如果接

当然响应头还存在一些可选字段。主要的可选字段为Sec-WebSocket-Protocol，是对客户端请求中所提供的Sec-WebSocket-Protocol子协议的选择结果的响应。当然cookie什么的也是可以的。

```
//handshaking.js
const crypto = require('crypto');
const cryptoKey = '258EAF5-E914-47DA-95CA-C5AB0DC85B11';

// 计算握手响应accept-key
let challenge = (reqKey) => {
  reqKey += cryptoKey;
  // crypto.vetHashes()可以获得支持的hash算法数组，我这里得到46个
  reqKey = reqKey.replace(/\s/g, '');
  // crypto.createHash('sha1').update(reqKey).digest()得到的是一个Uint8Array的加
  return crypto.createHash('sha1').update(reqKey).digest().toString('base64')
}

exports.handshaking = (req, socket, head) => {
  let _headers = req.headers,
      _key = _headers['sec-websocket-key'],
      resHeaders = [],
      br = "\r\n";
  resHeaders.push(
    'HTTP/1.1 101 WebSocket Protocol Handshake is OK',
    'Upgrade: websocket',
    'Connection: Upgrade',
    'Sec-WebSocket-Origin: ' + _headers.origin,
    'Sec-WebSocket-Location: ws://' + _headers.host + req.url,
  );
  let resAccept = challenge(_key);
  resHeaders.push('Sec-WebSocket-Accept: ' + resAccept + br, head);
  socket.write(resHeaders.join(br), 'binary');
}
```





关闭握手不可使用TCP且按关闭连接的方法来关闭握手。但是TCP关闭握手不是端到端可靠的，特别是出现拦截代理和其他的中间设施。也可以任何一端发送带有指定控制序号（比如说状态码1002,协议错误）的数据的帧来开始关闭握手，当另一方接收到这个关闭帧，就必须关闭连接。

数据传输

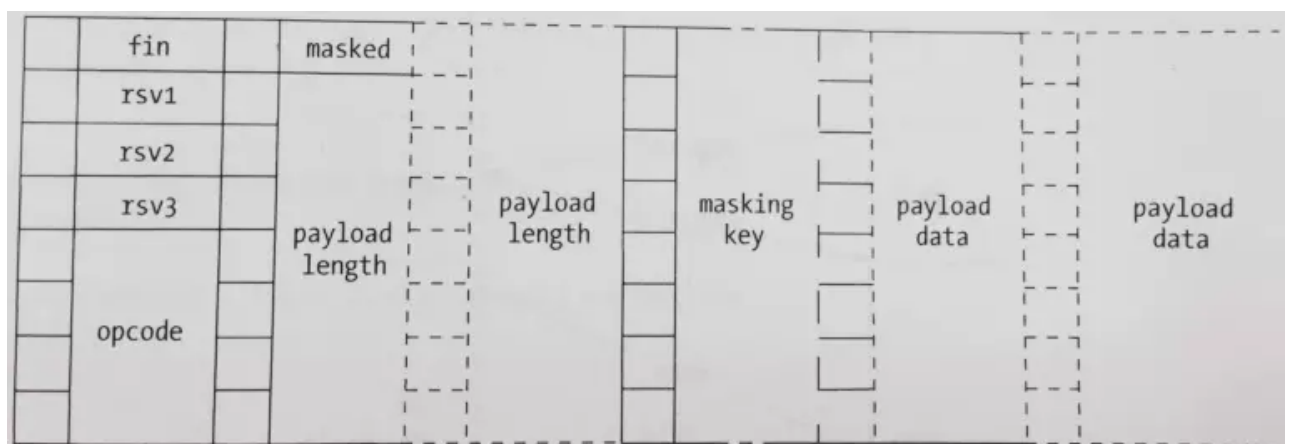
在WebSocket协议中,数据传输阶段使用frame（数据帧）进行通信，frame分不同的类型，主要有：文本数据，二进制数据。出于安全考虑和避免网络截获，客户端发送的数据帧必须进行掩码处理后才能发送到服务器，不论是否是在TLS安全协议上都要进行掩码处理。服务器如果没有收到掩码处理的数据帧时应该关闭连接，发送一个1002的状态码。服务器不能将发送到客户端的数据进行掩码处理，如果客户端收到掩码处理的数据帧必须关闭连接。

那我们服务器端接收到的数据帧是怎样的呢？

```
socket receive data :
▶ Uint8Array(19) [129, 141, 100, 7, 38, 140, 44, 98, ...]
```

1. 数据帧

WebSocket的数据传输是要遵循特定的数据格式-数据帧（frame）。



每一列代表一个字节，一个字节8位，每一位又代表一个二进制数。

fin: 标识这一帧数据是否是该分块的最后一帧。

1 为最后一帧

0 不是最后一帧。需要分为多个帧传输





%x0 表示一个继续帧
%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
%x10 表示一个pong
%x11-15 为以后的控制帧保留

masked: 占第二个字节的一位，定义了masking-key是否存在。并且使用masking-key掩码解析Payload data。

1 客户端发送数据到服务端
0 服务端发送数据到客户端

payload length: 表示Payload data的总长度。占7位，或者7+2个字节、或者7+8个字节。

0-125，则是payload的真实长度
126，则后面2个字节形成的16位无符号整型数的值是payload的真实长度，125<数据长度<65535
127，则后面8个字节形成的64位无符号整型数的值是payload的真实长度，数据长度>65535

masking key: 0或4字节，当masked为1的时候才存在，为4个字节，否则为0，用于对我们需要的数据进行解密

payload data: 我们需要的数据，如果masked为1，该数据会被加密，要通过masking key进行异或运算解密才能获取到真实数据。

2. 关于数据帧

因为WebSocket服务端接收到的数据有可能是连续的数据帧，一个message可能分为多个帧发送。但如果使用fin来做消息边界是有问题的。

我发送了一个27378个字节的字符串，服务器端共接收到2帧，两帧的fin都为1,而且根据规范计算出来的两帧的payload data的长度为27372少了6个字节。这缺少的6个字节其实刚好等于2个固有字节加上maskingKey的4个字节，也就是说第二帧就是一个纯粹的数据帧。这又是怎么回事呢？





分片

分片的主要目的是允许当消息开始但不必缓冲该消息时发送一个未知大小的消息。如果消息不能被分片，那么端点将不得不缓冲整个消息以便在首字节发生之前统计出它的长度。对于分片，服务器或中间件可以选择一个合适大小的缓冲，当缓冲满时，写一个片段到网络。

我们27378个字节的消息明显是知道message长度，那么就算这个message很大，根据规范1帧的数据长度理论上是 $0 < \text{数据长度} < 65535$ 的，这种情况下应该1帧搞定，他也只是当做一帧来发送，但是由于传输限制，所以这一个帧（我们收到的像是好几帧一样）会被拆分成几块发送，除了第一块是带有fin、opcode、masked等标识符，之后收到的块都是纯粹的数据（也就是第一块的payload data 的后续部分），这个就是socket的将WebSocket分好的一帧数据进行了分包发送。那么这种一帧被socket分包发送，导致像是分帧（分片）发送的情况（服务器端本应该只就收一帧），在服务器端我暂时还没有想到怎样获取状态来处理。

总结，客户端发送数据，在实现时还是需要手动进行分帧（分片），不然就按照一帧发送，小数据量无所谓；如果是大数据量，就会被socket自动分包发送。这个与WebSocket协议规范所标榜的自动分帧（分片），存在的差异应该是各个浏览器在对WebSocket协议规范的实现上偷工减料所造成的。所以我们看见socket.io等插件会有一个客户端接口，应该就是为了重新是实现WebSocket协议规范。从原理出发，我们接下来还是以小数据量（单帧）数据传输为例了。

3. 解析数据帧

```
//dataHandler.js
// 收集本次message的所有数据
getData(data, callback) {
  this.getState(data);
  // 如果状态码为8说明要关闭连接
  if(this.state.opcode == 8) {
    this.OPEN = false;
    this.closeSocket();
    return;
  }
  // 如果是心跳pong,回一个ping
  if(this.state.opcode == 10) {
    this.OPEN = true;
    this.pingTimes = 0;// 回了pong就将次数清零
    return;
  }
  // 收集本次数据流数据
```





```
        if(this.state.remains == 0){
            let buf = Buffer.concat(this.dataList, this.state.payloadLength);
            //使用掩码maskingKey解析所有数据
            let result = this.parseData(buf);
            // 数据接收完成后回调回业务函数
            callback(this.socket, result);
            //重置状态，表示当前message已经解析完成了
            this.resetState();
        }else{
            this.state.index++;
        }
    }

    // 收集本次message的所有数据
    getData(data, callback) {
        this.getState(data);

        // 收集本次数据流数据
        this.dataList.push(this.state.payloadData);

        // 长度为0，说明当前帧位最后一帧。
        if(this.state.remains == 0){
            let buf = Buffer.concat(this.dataList, this.state.payloadLength);
            //使用掩码maskingKey解析所有数据
            let result = this.parseData(buf);
            // 数据接收完成后回调回业务函数
            callback(this.socket, result);
            //重置状态，表示当前message已经解析完成了
            this.resetState();
        }else{
            this.state.index++;
        }
    }

    // 解析本次message所有数据
    parseData(allData, callback){
        let len = allData.length,
            i = 0;
        for(; i < len; i++){
            allData[i] = allData[i] ^ this.state.maskingKey[ i % 4 ]; // 异或运算，使用maskin
        }
        // 判断数据类型，如果为文本类型
        if(this.state.opcode == 1) allData = allData.toString();

        return allData;
    }
}
```





```
// 组装数据帧, 发送是不需要掩码加密
createData(data){
    let dataType = Buffer.isBuffer(data); // 数据类型
    let dataBuf, // 需要发送的二进制数据
        dataLength, // 数据真实长度
        dataIndex = 2; // 数据的起始长度
    let frame; // 数据帧

    if(dataType) dataBuf = data;
    else dataBuf = Buffer.from(data); // 也可以不做类型判断, 直接Buffer.from(data)
    dataLength = dataBuf.byteLength;

    // 计算payload data在frame中的起始位置
    dataIndex = dataIndex + (dataLength > 65535 ? 8 : (dataLength > 125 ? 2 : 0));

    frame = new Buffer.alloc(dataIndex + dataLength);

    // 第一个字节, fin = 1, opcode = 1
    frame[0] = parseInt(10000001, 2);

    // 长度超过65535的则由8个字节表示, 因为4个字节能表达的长度为4294967295, 已经完全够用, 因此直接将
    if(dataLength > 65535){
        frame[1] = 127; // 第二个字节
        frame.writeUInt32BE(0, 2);
        frame.writeUInt32BE(dataLength, 6);
    } else if(dataLength > 125){
        frame[1] = 126;
        frame.writeUInt16BE(dataLength, 2);
    } else{
        frame[1] = dataLength;
    }

    // 服务端发送到客户端的数据
    frame.write(dataBuf.toString(), dataIndex);

    return frame;
}
```

5. 心跳检测

```
// 心跳检查
sendCheckPing(){
```



[首页](#) ▼[搜索掘金](#)[登录](#) · [注册](#)

```
        if (_this.pingTimes >= 3) {
            _this.closeSocket();
            return;
        }
        //记录心跳次数
        _this.pingTimes++;
        if(_this.pingTimes == 100000) _this.pingTimes = 0;
        _this.sendCheckPing();
    }, 5000);
}
// 发送心跳ping
sendPing() {
    let ping = Buffer.alloc(2);
    ping[0] = parseInt(10001001, 2);
    ping[1] = 0;
    this.writeData(ping);
}
```

关闭连接

客户端直接调用close方法，服务器端可以使用socket.end方法。

最后

WebSocket在一定程度上让前端更加的有所作为，这个无疑是令人欣喜的，但是其规范中的很多不确定也是令人很惋惜的。因为浏览器对WebSocket规范的不完全实现，还有很多需要做的优化，这篇文章只是实现以一下WebSocket，关于期间很多的安全、稳定等方面的需要在应用中进行充实。当然用socket.io这种相对成熟的插件也是不错的选择。

关注下面的标签，发现更多相似文章

[Node.js](#)[JavaScript](#)[前端](#)[WebSocket](#)

daipeng7 Lv2 前端工程师
获得点赞 721 · 获得阅读 13,877

[关注](#)

[首页](#) ▼[登录](#) · [注册](#)

评论

xiexin Lv1 前端研发 @ 西瓜创客

学习了，最近刚好在实现一个原生的 socket

1年前



回复

yls web前端

你得把ws和wss判断下，什么情况下用ws，什么情况下用wss，其实这个就是和http|https对应的，你要判断下http的时候用ws，https的时候用wss，那么问题来了，http和https的差异是啥？=。=！！

1年前

2

回复

宽广宁静 前端开发

最近正把websocket用在直播评论上，挺好的

1年前



回复

坤少 前端开发 @ 上海帆讯

学习了 刚好正在写一个ws的功能

1年前



回复

卷家老大

最近刚用socket.io做了一个推送服务

1年前



回复

相关推荐

专栏 · 小兀666 · 4小时前 · Node.js

nodejs是如何和libuv以及v8一起合作的？(文末有彩蛋哦)

17

1

专栏 · william_li · 5小时前 · 前端

小程序篇(3)：瀑布流

11





专栏 · 故事胶片 · 3天前 · JavaScript / 前端

前端Vue中常用rules校验规则

 491

 38

专栏 · 火狼1 · 4天前 · React.js / 前端

React 开发必须知道的 34 个技巧【近1W字】

 575

 42

专栏 · 徐小夕_Lab实验室 · 1天前 · JavaScript / 前端


《前端实战总结》之使用pace.js为你的网站添加加载进度条


 90

 2

专栏 · Jiahonzheng · 4小时前 · Node.js


Node 绑定全局 TraceID


 4



专栏 · 幻灵尔依 · 2天前 · CSS / 前端

CSS简单的继承

 102

 41

专栏 · 政采云前端团队 · 3天前 · Vue.js / 前端

让你的组件千变万化，Vue slot 剖玄析微

 273

 16

专栏 · 知识小集 · 2天前 · 前端

2020 年 7 个软件开发趋势

 31

 5

