



链滴

nodejs- 从 0 写一个 websocket

作者: [bylx](#)

原文链接: <https://ld246.com/article/1682523662663>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

搬运了我b站的文，原创指数100(0)

bilibili专栏

我写的源码只有200行，已经足够完成websocket服务器的基础功能了。无论是自己添加功能，排查题的便利度还是性能方面，都一定是比主流框架强的。学习也好，自己的小项目用也好，我的代码是二之选。可以直接使用npm i iiws来安装。注意如果要修改源码的话记得在package.json中把main ws.min.js改成ws.js。

源码放这里了: <https://github.com/Bylx666/iiws>

简单说一下这个怎么用

```
const http = require("http");
const WSS = require('iiws');
```

```
const httpServer = http.createServer();
httpServer.listen(500);
```

```
const ws = new WSS(httpServer); // 直接把创建的服务器作为参数
ws.on("connect", (cli)=> { // 有任何客户端连接都会触发，cli就是连接的客户端的对象
```

```
  console.log("一个客户端连接了")
  ws.broadcast("Hello! every client!"); // 广播，给所有在线的客户端都说一声
  cli.send("Welcome, a user"); // 给这个客户端发消息
  cli.ping(); // ping一下
  cli.connectTime = Date.now(); // 给客户端的对象直接赋一个新属性
  console.log(ws.clients); // [cli1, cli2, cli3...] 所有客户端的列表，要联动其他客户端时可以调用(提
  : 别忘了可以直接用ws.clients[n].connectTime取得别的客户端对象赋的属性哦)
```

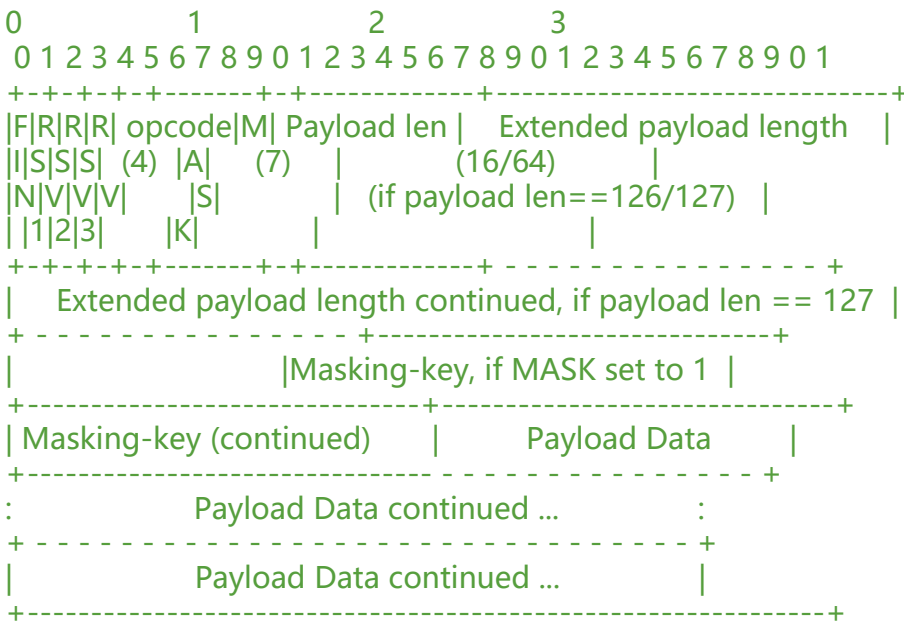
```
  cli.on("message", (data)=> {
    console.log(data); // data是个buffer，可以直接toString作为文本处理
  });
  cli.on("close", ()=> {
    console.log("a client closed"); // 客户端断开时会触发
  });
  cli.on("error", (err)=> {
    console.log(err.message); // 出错？小项目谁处理错误？(bushi)
  });
});
```

理论知识

websocket不同于http，只有头和内容。**websocket的通讯要作为“帧”来发送**，而帧都是二进制

, 相对来说服务器端处理会相对麻烦。

一个“帧”的基本格式如下



无论是向服务器发送还是服务器向客户端发送都是需要发送帧形式的。第一次接触二进制处理的话会懵，我简单说几个概念：1 byte = 8 bits, bit只有1和0。buffer[n]取得的都是1byte, 也就是8个bit上表的数字代表的都是bit, 也就是一行是32 bits, 4 bytes。

上表中fin为1代表消息结束, 0代表这是个片段帧。 0的话需要先把这一帧的内容加到上一个fin为0的内容后面, 直到出现fin为1的帧, 代表了这个消息的结束。原生api中只提供了"data"接口, 这个api只给你散落的数据信息, 并不能把客户端发送的消息完完整整的直接给你, 就需要我们自己去处理。

opcode是operation code, 就是操作码。

- |0x0 |是个片段帧? 实际好像没啥用, 就算是片段帧也和opcode没关系, 只用fin判断
- |0x1 |是个文本帧
- |0x2 |是个二进制帧
- |0x8 |控制帧, 告诉另一方我要断开连接了
- |0x9 |控制帧, 发送ping, 另一方应该回复pong的控制帧告诉你我们俩的连接还在线
- |0xA |控制帧pong

二进制帧用的少, 源码中未涉及。一般使用json+base64传输二进制内容。之所以opcode占4位是因0xF(16进制的f)=1111(2进制), 所以说一个byte可以容纳2个16进制数。

payload length意思是内容的大小, 在编程中一般叫做长度。** **这个设计就很有意思了。首先看个7位的payload length, 如果塞满的话它就是1111111, 也就是127。减一呢? 1111110, 也就是26。塞满的话意思就是就直接告诉你我要64位大小来塞这个消息的大小数值, 126就是要16位。小于26的话就直接拿这7位来装这个数字。

mask就是是否有掩码。 1有0无。注意一般客户端是浏览器的话不要设置这个为1, 规范中明确说了。

masking-key就是4 bytes的掩码键, 没那么复杂, 一行就能搞定 (等会看代码)。 如果mask是0就没有掩码键后面直接是内容。

payload就是内容, 不多说。

代码和思路

写代码第一步，造小轮子（明显要用的框架）

```
function Event() {  
  
  var listeners = {};  
  this.on = (event, callback)=> {  
  
    if(listeners[event]) listeners[event].push(callback);  
    else listeners[event] = [callback];  
  
  };  
  this.off = (event, callback)=> {  
  
    var e = listeners[event];  
    var i = e.indexOf(callback);  
    if(e&&i!==-1) e.splice(i, 1);  
  
  };  
  this.emit = (event, param)=> {  
  
    if(listeners[event]) listeners[event].forEach((callback)=> callback(param));  
  
  };  
}
```

但凡学过一点点js的都非常熟悉了，一笔带过。

第二步，接入原生api。 WebSocket的本质是http的upgrade。我们直接创建一个WSS类，更详细的路在代码的注释当中。

```
function WSS(server) {  
  
  Event.call(this); // this转移术  
  
  var clients = []; // 客户端列表  
  
  this.server = server; // 你的参数  
  this.clients = clients;  
  
  this.broadcast = (data)=> { // 广播，给每个在线客户端发送一条消息。  
  
    clients.forEach((v)=> {v.send(data)});  
  
  };  
  
  server.on("upgrade", (req, socket)=> { // 核心api, upgrade  
  
    socket.write([  
      "HTTP/1.1 101 Switching Protocols",  
      "Upgrade: websocket",  
      "Connection: Upgrade",  
    ]  
  )  
}
```

```
"Sec-WebSocket-Accept: "+require("crypto").createHash("sha1").update(req.headers['sec-websocket-key']+"258EAF5-E914-47DA-95CA-C5AB0DC85B11").digest("base64")
].join("\n")+"\n\n"); // 这里是固定的，握手内容。
```

```
var cli = { // 在你connect的时候得到的cli对象就是这玩意
  send(data, options) { // 发送

    socket.write(createFrame(data, options));

  },
  ping() { // 发送ping的控制帧

    socket.write(createFrame("", {opcode: 9}));

  },
  pong() { // pong控制帧

    socket.write(createFrame("", {opcode: 10}));

  },
  close() { // 关闭服务器与这个客户端的连接

    var cliI = clients.indexOf(cli);
    if(cliI === -1) return false;
    clients.splice(clients.indexOf(cli), 1); // 从clients数组删除它
    socket.write(createFrame("", {opcode: 8})); // 向客户端发送关闭信号
    cli.emit("close"); // 触发cli的close事件
    socket.destroy();

  },
  socket: socket // `cli.socket`就是on("upgrade", (req,socket)=>{})得到的socket
};
Event.call(cli); // 和上边Event同理，给cli附加事件相关的方法。

this.emit("connect", cli); // 触发connect的事件，把这个cli发到你的参数里面
clients.push(cli); // 把cli push到clients数组里面

var buf = Buffer.allocUnsafe(0); // 未处理过的数据buffer堆在一起
var messageData = Buffer.allocUnsafe(0); // 代表一条完整信息的内容
var dataList = []; // 里面放了{l: 要消耗的长度, f: 消耗的内容（或消耗结束后）做什么}的数组。消
是什么意思？消耗就是把`buf`从前往后的删掉要消耗的长度，f函数可以得到消耗掉的内容
var frameEnd = true; // 一帧消耗结束了没，只有是true的时候，下一次data事件才会执行下一
的消耗
function nextFrame() { // 消耗一帧的长度，把从`buf`中消耗得到的一帧的内容给加进`messageD
ta`，如果`fin`是1的话就直接触发message事件，把完整消息传给你

var meta = parseFrameMeta(buf); // 解析元数据，不难理解，就是得到这一帧的fin, opcode
类的东西
dataList.push({l: meta.lenMeta, f: ()=> { // 消耗这个元数据长度

frameEnd = false; // 这一帧的buffer没消耗完呢！buf只消耗了元数据的长度，就算下次data
了也不要再去消耗下一帧
dataList.push({l: meta.len, f: (d)=> { // 开始消耗这一帧的内容的长度
```

```

    messageData = Buffer.concat([ messageData, imask(d, meta.maskKey) ]); // messageDa
a += 反掩码后的 data
    frameEnd = true; // 消耗完了, 気持ちいい, 下次data来了就可以下一帧了
    if(meta.fin) { // 有时候会发现帧的fin不为1, 就不能直接触发message, 否则数据就发不完整

        if(meta.opcode===8) return cli.close();
        if(meta.opcode===9) return cli.pong();
        cli.emit("message", messageData); // fin是1就可以发喽~
        messageData = Buffer.allocUnsafe(0); // 把messageData重置掉

    }

    });

});

}

```

socket.on("data", (chunk)=> { // 这就是核心api之socket.ondata, 会给你发未处理的数据, 不有时候会给你发fin为0的片段帧, 甚至一帧都不发完整, 所以必须把发过来的东西囤到一块, 然后每data来的时候处理一次。

```

    buf = Buffer.concat([buf, chunk]); // buf += chunk
    if(frameEnd) nextFrame();
    while(dataList[0]&&buf.byteLength >= dataList[0].l) { // dataList中只要有东西就从前往后处
, 除非元数据里面说这一帧内容挺大的, 这个data来了buf整个现在也不够, 才会停下来。

        const l = dataList[0].l; // 要消耗的长度
        dataList[0].f(buf.subarray(0, l)); // 把消耗的东西传给回调函数
        buf = buf.subarray(l); // 把buf消耗的东西删掉
        dataList.splice(0, 1); // 这条处理完了, 从dataList删掉

    }

});

socket.on("end", ()=> {

    cli.close(); // 原生socket的end事件相当于cli的close事件, 可以用cli.on("close", ()=>{})捕获

});

socket.on("error", (err)=> {

    cli.emit("error", err); // 韩信加净化, 谁处理报错啊(bushi)
    cli.close();

});

});

}

```

第三步, 补全函数。 上面代码中很明显用了非原生函数, 我们来跟着刚开始说的理论写一下这些函数。

创建一帧

```
function createFrame(content, options) {  
  
  var len = Buffer.byteLength(content); // 这一帧内容的长度  
  
  var buf = null;  
  if(len > 65535) { // 65535 就是16位最大值  
  
    buf = Buffer.alloc(10+len);  
    buf[1] = 127; // 127 = 01111111 说明要用64位存内容长度。不用掩码。`len7`就是buf[1]的后  
    位写的数字  
    buf.writeUInt32BE(len, 6); // buffer api最高只有写32位整数的，所以跳32位也就是4 bytes再写  
    个数字。没错，使用这个api代表最大只能传32位最大值大小的内容。  
    buf.write(content, 10); // 32位是4 bytes, 10 = 6 + 4  
  
  } else if(len > 125) {  
  
    buf = Buffer.alloc(4+len);  
    buf[1] = 126; // 126 = 01111110 要用16位存内容长度  
    buf.writeUInt16BE(len, 2);  
    buf.write(content, 4); // 16 bits = 2 bytes, 4 = 2 + 2  
  
  } else {  
  
    buf = Buffer.alloc(2+len);  
    buf[1] = len; // 如果 len7 !== 126 或 len7 !== 127, len7就直接赋值为长度。由于mask也就是  
    个byte第一位通常取0，直接使用等号。  
    buf.write(content, 2);  
  
  }  
  
  if(options) {  
  
    const opcode = options.opcode;  
    if(opcode && opcode < 15 && opcode >= 0) buf[0] = 128 | opcode; // 128 = 10000000, opcod  
    是后四位  
  
  }  
  
  else buf[0] = 129; // 129 = 10000001, opcode = 0x1  
  
  return buf;  
  
}
```

解析一帧的元数据

```
function parseFrameMeta(source) {  
  
  var src = Buffer.from(source);  
  
  var len7 = src[1] & 127; // 127 = 01111111, len7是原buffer第二个byte的后七位  
  var len = 0; // 内容长度
```

```

var lenMeta = 0; // 元数据长度
var masked = src[1] >= 128; // 128 = 10000000, 因为第一位是`masked`, 是true的话这整个byt
>= 128
if(len7===127) {

  len = src.readUInt32BE(6);
  lenMeta = 10;

}else if(len7===126) {

  len = src.readUInt16BE(2);
  lenMeta = 4;

}else {

  len = len7;
  lenMeta = 2;

}

return {
  fin: src[0] >= 128, // 第一个byte前四位只有第一位有可能是1, 是1的话整个byte就大于1000000
, 不是1的话整个byte就会小于等于00001111
  opcode: src[0] & 15, // 15 = 01111111, &是js二进制"与"运算
  mask: masked,
  maskKey: masked?src.subarray(lenMeta, lenMeta + 4):null,
  len7: len7,
  len: len,
  lenMeta: lenMeta+(masked?4:0)
};
}

```

反掩码 (确实很简单吧)

```

function imask(data, key) {

  if(!key) return data;

  var d = Buffer.from(data);
  for(let i = 0; i < d.length; ++i) d[i] = d[i] ^ key[i % 4]; // 把元数据分4各一组进行XOR运算
  return d;

}

```

第四步, 导出

```

module.exports = WSS;

```

结语

自己学这东西的时候盯着别人的框架几千行, 费劲的理解, 花了我很长时间。最后只用200行浓缩了本质的内容时我也是很有成就感。这个教程可以说是我目前对websocket的所有理解, 即使你并不是nodejs编程, 你也可以学到websocket通信的原理以及数据处理方式的思路。希望能对看到这的人有

, 有哪里疑惑没讲明白的请在评论区指出。