[**WebSocket协议以及ws源码分析**](https://segmentfault.com/a/1190000019293037)

[javascript](https://segmentfault.com/t/javascript)[node.js](https://segmentfault.com/t/node.js)[websocket](https://segmentfault.com/t/websocket)

发布于 2019-05-25

**¿**

本文包括如下内容:

* WebSocket协议第四章 - 连接握手
* WebSocket协议第五章 - 数据帧
* nodejs ws库源码分析 - 连接握手过程
* nodejs ws库源码分析 - 数据帧解析过程

参考

[WebSocket 协议深入探究](https://user-gold-cdn.xitu.io/2019/5/25/16aec9704c8bc80f)

[ws - github](https://github.com/websockets/ws)

本文对WebSocket的概念、定义、解释和用途等基础知识不会涉及, 稍微偏干一点, **篇幅较长, markdown大约800行, 阅读需要耐心**

**1. 连接握手过程**

关于WebSocket有一句很常见的话: **Websocket复用了HTTP的握手通道**, 它具体指的是:

客户端通过HTTP请求与WebSocket服务器协商升级协议, 协议升级完成后, 后续的数据交换则遵照WebSocket协议

**1.1 客户端: 申请协议升级**

首先由客户端换发起协议升级请求, 根据WebSocket协议规范, 请求头必须包含如下的内容

**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==

* 请求行: 请求方法必须是GET, HTTP版本至少是1.1
* 请求必须含有Host
* 如果请求来自浏览器客户端, 必须包含Origin
* 请求必须含有Connection, 其值必须含有"Upgrade"记号
* 请求必须含有Upgrade, 其值必须含有"websocket"关键字
* 请求必须含有Sec-Websocket-Version, 其值必须是13
* 请求必须含有Sec-Websocket-Key, 用于提供基本的防护, 比如无意的连接

**1.2 服务器: 响应协议升级**

服务器返回的响应头必须包含如下的内容

HTTP/1.1 101 Switching Protocols

Connection:Upgrade

Upgrade: websocket

Sec-WebSocket-Accept: Oy4NRAQ13jhfONC7bP8dTKb4PTU=

* 响应行: HTTP/1.1 101 Switching Protocols
* 响应必须含有Upgrade, 其值为"weboscket"
* 响应必须含有Connection, 其值为"Upgrade"
* 响应必须含有Sec-Websocket-Accept, 根据请求首部的Sec-Websocket-key计算出来

**1.3 Sec-WebSocket-Key/Accept的计算**

规范提到:

Sec-WebSocket-Key值由一个随机生成的16字节的随机数通过base64（见RFC4648的第四章）编码得到的

例如, 随机选择的16个字节为:

// 十六进制 数字1~16

0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f 0x10

通过base64编码后值为: AQIDBAUGBwgJCgsMDQ4PEA==

测试代码如下:

**const** list = Array.from({ length: 16 }, (v, index) => ++index)

**const** key = Buffer.from(list)

console.log(key.toString('base64'))

*// AQIDBAUGBwgJCgsMDQ4PEA==*

而Sec-WebSocket-Accept值的计算方式为:

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

此处不需要纠结神奇字符串258EAFA5-E914-47DA-95CA-C5AB0DC85B11, 它就是一个GUID, 没准儿是写RFC的时候随机生成的

测试代码如下:

**const** crypto = require('crypto')

**function** **hashWebSocketKey** (key) {

**const** GUID = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11'

**return** crypto.createHash('sha1')

.update(key + GUID)

.digest('base64')

}

console.log(hashWebSocketKey('w4v7O6xFTi36lq3RNcgctw=='))

*// Oy4NRAQ13jhfONC7bP8dTKb4PTU=*

**1.4 Sec-WebSocket-Key的作用**

前面简单提到他的作用为: **提供基础的防护, 减少恶意连接**, 进一步阐述如下:

* Key可以避免服务器收到非法的WebSocket连接, 比如http请求连接到websocket, 此时服务端可以直接拒绝
* Key可以用来初步确保服务器认识ws协议, 但也不能排除有的http服务器只处理Sec-WebSocket-Key, 并不实现ws协议
* Key可以避免反向代理缓存
* 在浏览器中发起ajax请求, Sec-Websocket-Key以及相关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

最终需要强调的是: **Sec-WebSocket-Key/Accept并不是用来保证数据的安全性, 因为其计算/转换公式都是公开的, 而且非常简单, 最主要的作用是预防一些意外的情况**

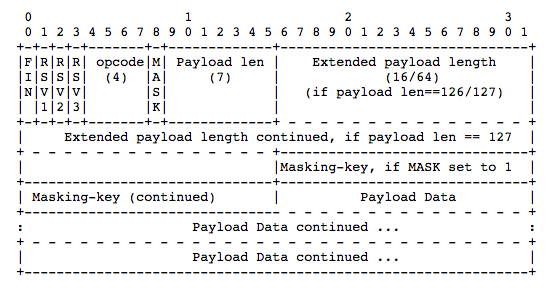
**2. 数据帧**

WebSocket通信的最小单位是帧, 由一个或多个帧组成一条完整的消息, 交换数据的过程中, 发送端和接收端需要做的事情如下:

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

**数据帧格式作为核心内容, 一眼看去似乎难以理解, 但本文作者下死命令了, 必须理解, 冲冲冲**

**2.1 数据帧格式详解**



* FIN: 占1bit
  + 0表示不是消息的最后一个分片
  + 1表示是消息的最后一个分片
* RSV1, RSV2, RSV3: 各占1bit, 一般情况下全为0, 与Websocket拓展有关, 如果出现非零的值且没有采用WebSocket拓展, 连接出错
* Opcode: 占4bit
  + %x0: 表示本次数据传输采用了数据分片, 当前数据帧为其中一个数据分片
  + %x1: 表示这是一个文本帧
  + %x2: 表示这是一个二进制帧
  + %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: 表示这是一个心跳请求(ping)
  + %xA: 表示这是一个心跳响应(pong)
  + %xB-F: 保留的操作代码, 用于后续定义的非控制帧
* Mask: 占1bit
  + 0表示不对数据载荷进行掩码异或操作
  + 1表示对数据载荷进行掩码异或操作
* Payload length: 占7或7+16或7+64bit
  + 0~125: 数据长度等于该值
  + 126: 后续的2个字节代表一个16位的无符号整数, 值为数据的长度
  + 127: 后续的8个字节代表一个64位的无符号整数, 值为数据的长度
* Masking-key: 占0或4bytes
  + 1: 携带了4字节的Masking-key
  + 0: 没有Masking-key
  + **掩码的作用并不是防止数据泄密,而是为了防止早期版本协议中存在的代理缓存污染攻击等问题**
* payload data: 载荷数据

我想如果知道byte和bit的区别, 这部分就没问题- -

**2.2 数据传递**

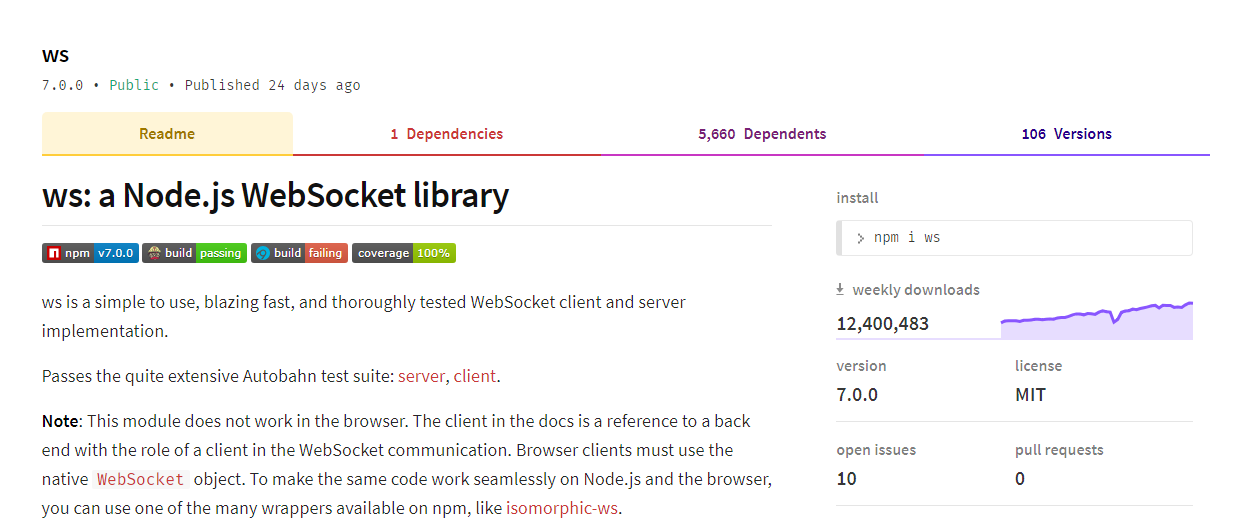
WebSocket的每条消息可能被切分成多个数据帧, 当接收到一个数据帧时,会根据FIN值来判断, 是否为最后一个数据帧

数据帧传递示例:

1. FIN=0, Opcode=0x1: 发送文本类型, 消息还没有发送完成,还有后续帧
2. FIN=0, Opcode=0x0: 消息没有发送完成, 还有后续帧, 接在上一条后面
3. FIN=1, Opcode=0x0: 消息发送完成, 没有后续帧, 接在上一条后面组成完整消息

**3. ws库源码分析: 连接握手过程**

虽然之前用的都是socket.io, 偶然发现了ws, 使用量竟然还挺大, 周下载量是socket.io的六倍



在NodeJS中, 每当遇到协商升级请求时, 就会触发http模块的upgrade事件, 这便是实现WebSocketServer的切入点, 原生示例代码如下:

*// 创建 HTTP 服务器。*

**const** srv = http.createServer( (req, res) => {

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('响应内容');

});

srv.on('upgrade', (req, socket, head) => {

*// 特定的处理, 以实现Websocket服务*

});

并且, 在一般的使用中, **都是在一个已有的httpServer基础上进行拓展, 以实现WebSocket, 而不是创建一个独立的WebSocketServer**

在一个已有httpServer的基础上, ws使用的实例代码为

**const** http = require('http');

**const** WebSocket = require('ws');

**const** server = http.createServer();

**const** wss = **new** WebSocket.Server({ server });

server.listen(8080);

已有的httpServer作为参数传给了WebSocket.Server构造函数, 所以源码分析的核心切入点为:

new WebSocket.Server({ server });

通过这个切入点, 就可以**完整复现连接握手的过程**

**3.1 分析WebSocketServer类**

因为httpServer已作为参数传递进来, 因此其构造函数变得十分简单:

**class** **WebSocketServer** **extends** **EventEmitter** {

**constructor**(options, callback) {

**super**()

*// 在提供了http server的基础上, 代码可以简化为*

**if** (options.server) {

**this**.\_server = options.server

}

*// 监听事件*

**if** (**this**.\_server) {

**this**.\_removeListeners = addListeners(**this**.\_server, {

listening: **this**.emit.bind(**this**, 'listening'),

error: **this**.emit.bind(**this**, 'error'),

*// 核心*

upgrade: (req, socket, head) => {

*// 下一步切入点*

**this**.handleUpgrade(req, socket, head, (ws) => {

**this**.emit('connection', ws, req)

})

}

})

}

}

}

*// 这是一段非常带秀的代码, 在绑定多个事件监听器的同时返回一个移除多个事件监听器的函数*

**function** **addListeners**(server, map) {

**for** (**const** event **of** Object.keys(map)) server.on(event, map[event]);

**return** **function** **removeListeners**() {

**for** (**const** event **of** Object.keys(map)) {

server.removeListener(event, map[event]);

}

};

}

可以看到, 在构造函数中, 为httpServer注册了upgrade事件的监听器, 触发时, 会执行this.handleUpgrade函数, 这便是下一步的方向

**3.2 过滤非法请求: handleUpgrade函数**

这个函数主要用来过滤掉不合法的请求, 检查的内容包括:

* Sec-WebSocket-Key值
* Sec-WebSocket-Version值
* WebSocket请求的路径

关键代码如下:

**const** keyRegex = /^[+/0-9A-Za-z]{22}==$/;

handleUpgrade(req, socket, head, cb) {

socket.on('error', socketOnError)

*// 获取sec-websocket-key*

**const** key = req.headers['sec-websocket-key'] !== undefined

? req.headers['sec-websocket-key']

: false

*// 获取sec-websocket-version*

**const** version = +req.headers['sec-websocket-version']

*// 获取协议拓展, 本篇不涉及*

**const** extensions = {};

*// 对于不合法的请求, 中断握手*

**if** (

req.method !== 'GET' ||

req.headers.upgrade.toLowerCase() !== 'websocket' ||

!key ||

!keyRegex.test(key) ||

(version !== 8 && version !== 13) ||

*// 该函数是对Websocket请求路径的判断, 与option.path相关, 不展开*

!**this**.shouldHandle(req)

) {

**return** abortHandshake(socket, 400)

}

*// 对于合法的请求, 给它升级!*

**this**.completeUpgrade(key, extensions, req, socket, head, cb)

}

对于不合法的请求, 直接400 bad request了, abortHandshake如下:

**const** { STATUS\_CODES } = require('http');

**function** **abortHandshake**(socket, code, message, headers) {

*// net.Socket 也是双工流，因此它既可读也可写*

**if** (socket.writable) {

message = message || STATUS\_CODES[code];

headers = {

Connection: 'close',

'Content-type': 'text/html',

'Content-Length': Buffer.byteLength(message),

...headers

};

socket.write(

`HTTP/1.1 ${code} ${STATUS\_CODES[code]}\r\n` +

Object.keys(headers)

.map((h) => `${h}: ${headers[h]}`)

.join('\r\n') +

'\r\n\r\n' +

message

);

}

*// 移除handleUpgrade中添加的error监听器*

socket.removeListener('error', socketOnError);

*// 确保在该 socket 上不再有 I/O 活动*

socket.destroy();

}

如果一切顺利, 我们来到completeUpgrade函数

**3.3 完成握手: completeUpgrade函数**

这个函数主要用来, 返回正确的响应, 触发相关的事件, 记录值等, 代码比较简单

**const** { createHash } = require('crypto');

**const** { GUID } = require('./constants');

**const** WebSocket = require('./websocket');

**function** **completeUpgrade**(key, extensions, req, socket, head, cb) {

*// Destroy the socket if the client has already sent a FIN packet.*

**if** (!socket.readable || !socket.writable) **return** socket.destroy()

*// 生成sec-websocket-accept*

**const** digest = createHash('sha1')

.update(key + GUID)

.digest('base64');

*// 组装Headers*

**const** headers = [

'HTTP/1.1 101 Switching Protocols',

'Upgrade: websocket',

'Connection: Upgrade',

`Sec-WebSocket-Accept: ${digest}`

];

*// 创建一个Websocket实例*

**const** ws = **new** Websocket(null)

**this**.emit('headers', headers, req);

*// 返回响应*

socket.write(headers.concat('\r\n').join('\r\n'));

socket.removeListener('error', socketOnError);

*// 下一步切入点*

ws.setSocket(socket, head, **this**.options.maxPayload);

*// 通过Set记录处于连接状态的客户端*

**if** (**this**.clients) {

**this**.clients.add(ws);

ws.on('close', () => **this**.clients.delete(ws));

}

*// 触发connection事件*

cb(ws);

}

到这里, 就完成了整个握手阶段, 但还没涉及到对数据帧的处理

**4. ws库源码分析: 数据帧处理**

上一章末尾, 启示下文的代码为completeUpgrade中的:

**ws**.setSocket(**socket**, **head**, **this**.options.maxPayload);

进入WebSocket类中的setSocket方法, 关于数据帧处理代码主要可以简化为:

Class WebSocket **extends** EventEmitter {

...

setSocket(socket, head, maxPayload) {

*// 实例化一个可写流, 用于处理数据帧*

**const** receiver = **new** Receiver(

**this**.\_binaryType,

**this**.\_extensions,

maxPayload

);

receiver[kWebSocket] = **this**;

socket.on('data', socketOnData);

}

}

**function** **socketOnData**(chunk) {

**if** (!**this**[kWebSocket].\_receiver.write(chunk)) {

**this**.pause();

}

}

此处忽略了很多事件处理, 例如error, end, close等, 因为他们与本文目标无关, 对于一些API, 也不做介绍

所以核心切入点为Receiver类, **它就是用于处理数据帧的核心**

**4.1 Receiver类基本构造**

Receiver类继承自可写流, 还需要明确两点基本概念:

* stream所有的流都是EventEmitter的实例
* 实现可写流需要实现writable.\_write方法, 该方法供内部使用

**const** { Writable } = require('stream')

**class** **Recevier** **extends** **Writable** {

**constructor**(binaryType, extensions, maxPayload) {

**super**()

**this**.\_binaryType = binaryType || BINARY\_TYPES[0]; *// nodebuffer*

**this**[kWebSocket] = undefined; *// WebSocket实例的引用*

**this**.\_extensions = extensions || {}; *// WebSocket协议拓展*

**this**.\_maxPayload = maxPayload | 0; *// 100 \* 1024 \* 1024*

**this**.\_bufferedBytes = 0; *// 记录buffer长度*

**this**.\_buffers = []; *// 记录buffer数据*

**this**.\_compressed = false; *// 是否压缩*

**this**.\_payloadLength = 0; *// 数据帧 PayloadLength*

**this**.\_mask = undefined; *// 数据帧Mask Key*

**this**.\_fragmented = 0; *// 数据帧是否分片*

**this**.\_masked = false; *// 数据帧 Mask*

**this**.\_fin = false; *// 数据帧 FIN*

**this**.\_opcode = 0; *// 数据帧 Opcode*

**this**.\_totalPayloadLength = 0; *// 载荷总长度*

**this**.\_messageLength = 0; *// 载荷总长度, 与this.\_compressed有关*

**this**.\_fragments = []; *// 载荷分片记录数组*

**this**.\_state = GET\_INFO; *// 标志位, 用于startLoop函数*

**this**.\_loop = false; *// 标志位, 用于startLoop函数*

}

\_write(chunk, encoding, cb) {

**if** (**this**.\_opcode === 0x08 && **this**.\_state == GET\_INFO) **return** cb();

**this**.\_bufferedBytes += chunk.length;

**this**.\_buffers.push(chunk);

**this**.startLoop(cb);

}

}

可以看到, 每当收到新的数据帧, 就会将其记录在\_buffers数组中, 并立即开始解析流程startLoop

**4.2 数据帧解析流程: startLoop函数**

startLoop(cb) {

let err;

**this**.\_loop = true;

**do** {

switch (**this**.\_state) {

case GET\_INFO:

err = **this**.getInfo();

**break**;

case GET\_PAYLOAD\_LENGTH\_16:

err = **this**.getPayloadLength16();

**break**;

case GET\_PAYLOAD\_LENGTH\_64:

err = **this**.getPayloadLength64();

**break**;

case GET\_MASK:

**this**.getMask();

**break**;

case GET\_DATA:

err = **this**.getData(cb);

**break**;

**default**:

*// `INFLATING`*

**this**.\_loop = false;

**return**;

}

} **while** (**this**.\_loop);

cb(err);

}

解析流程很简单:

* getInfo首先解析FIN, RSV, OPCODE, MASK, PAYLOAD LENGTH等数据
* 因为payload length分为三种情况(具体后面叙述, 此处只列出分支):
  + 0~125: 调用haveLength方法
  + 126: 先触发getPayloadLength16方法, 再调用haveLength方法
  + 127: 先出法getPayloadLength64方法, 再调用haveLength方法
* haveLength方法中, 如果存在掩码(mask), 先调用getMask方法, 再调用getData方法

整体流程和状态通过this.\_loop和this.\_state控制, 比较直观

**4.3 消费Buffer的方式: consume方法**

按理说第一步应该分析getInfo方法, 不过里面涉及到了consume方法, 这个函数**提供了一种简洁的方式消费已获取的Buffer**, 这个函数接受一个参数n, 代表需要消费的字节数, 最后返回消费的字节

假如需要获得数据帧的第一个字节的数据(包含了 FIN + RSV + OPCODE), 只需要通过this.consume(1)即可

记录值this.\_buffers是一个buffer数组, 最开始, 里面存放完整的数据帧, **随着消费的进行, 数据则会逐渐变小**, 那么每次消费存在三种可能:

1. 消费的字节数恰好等于一个chunk的字节数
2. 消费的字节数小于一个chunk的字节数
3. 消费的字节数大于一个chunk的字节数

对于第一种情况, 只需要**移出 + 返回**即可

**if** (n === **this**.\_buffers[0].length) **return** **this**.\_buffers.shift()

对于第二种情况, 只需要**裁剪 + 返回**即可

**if** (n < **this**.\_buffers[0].length) {

**const** buf = **this**.\_buffers[0]

**this**.\_buffers[0] = buf.slice(n)

**return** buf.slice(0, n)

}

对于第三种情况, 会稍微复杂一点, 首先我们要申请一个大小为需要消费字节数的buffer空间, 用于存储返回的buffer

*// buffer空间是否初始化并不重要, 因为最终他都会被全部覆盖*

**const** dst = Buffer.allocUnsafe(n)

在这种情况中, **可以保证他的长度大于第一个chunk, 但不能确定在消费一个chunk之后, 是否还大于第一个chunk(消费之后索引前移), 因此需要循环**

*// do...while可以避免一次无意义判断, 首先执行一次循环体, 再判断条件*

**do** {

**const** buf = **this**.\_buffers[0]

*// 如果长度大于第一个chunk, 移除 + 复制即可*

**if** (n >= buf.length) {

**this**.\_buffers.shift().copy(dst, dst.length - n);

}

*// 如果长度小于一个chunk, 裁剪 + 复制即可*

**else** {

*// buf.copy这个api就自己复习一下嗷*

buf.copy(dst, dst.length - n, 0, n);

**this**.\_buffers[0] = buf.slice(n);

}

n -= buf.length;

} **while** (n > 0)

**4.4 分析数据帧: getInfo方法**

一个最小的数据帧必须包含如下的数据:

**FIN** (1 bit) + **RSV** (3 bit) + **OPCODE** (4 bit) + **MASK** (1 bit) + **PAYLOADLENGTH** (7 bit)

最少2个字节, 因此少于两个字节的数据帧是错误的, 简化的getInfo如下

getInfo() {

**if** (**this**.\_bufferedBytes < 2) {

**this**.\_loop = false

**return**

}

**const** buf = **this**.consume(2)

*// 只保留了数据帧中的几个关键数据*

**this**.\_fin = (buf[0] & 0x80) === 0x80

**this**.\_opcode = buf[0] & 0x0f

**this**.\_payloadLength = buf[1] & 0x7f

**this**.\_masked = (buf[1] & 0x80) === 0x80

*// 对应Payload Length的三种情况*

**if** (**this**.\_payloadLength === 126) **this**.\_state = GET\_PAYLOAD\_LENGTH\_16

**else** **if** (**this**.\_payloadLength === 127) **this**.\_state = GET\_PAYLOAD\_LENGTH\_64

**else** **return** **this**.haveLength()

}

此处的核心就是按位于运算符&的含义, 先以FIN为例, FIN在数据帧中处于第一个bit

*// FIN的值用[]指代, X代表第一个字节中的后续bit*

[]xxxxxxx

*// 十六进制数0x80代表二进制*

10000000

*// 两者按位与, 结果与后面7个bit无关*

[]0000000

*// 因此, 只需要比较[]0000000 和 10000000是否相等即可, 简化即得到*

**this**.\_fin = (buf[0] & 0x80) === 0x80

OPCODE和PAYLOAD LENGTH同理

// OPCODE处于第一个字节的后四位, 与0000 1111按位与即可

xxxx[][][][] & 0000 1111 (也就是0x0f)

// PAYLOAD LENGTH处于第二个字节的后七为, 与0111 1111按位于即可

x[][][][][][][][] & 0111 1111 (也就是0x7f)

**4.5 Payload Length三种情况与大小端**

三种情况如下:

* 0-125: 载荷实际长度就是0-125之间的某个数
* 126: 载荷实际长度为**随后2个字节代表的一个16位的无符号整数的数值**
* 127: 载荷实际长度为**随后8个字节代表的一个64位的无符号整数的数值**

可能听起来比较绕, 看代码, 以126分支为例:

getPayloadLength16() {

**if** (**this**.\_bufferedBytes < 2) {

**this**.\_loop = false;

**return**;

}

**this**.\_payloadLength = **this**.consume(2).readUInt16BE(0);

**return** **this**.haveLength();

}

可以看到, 处理长度的核心为readUInt16BE(0), 这便涉及到大小端了:

* 大端(Big endian)**认为第一个字节是最高位字节, 和我们对十进制数字大小的认知相似**
* 小端(Little endian)**认为第一个字节是最低位字节**

那么, 规范中提到的**随后2个字节代表的一个16位的无符号整数的数值**, 自然指的是大端了

大端 vs 小端对比:

// 假设后面两个字节二进制值为

1111 1111 0000 0001

// 转为十六进制为

0xff 0x01

// 大端输出 65281

console.log(Buffer.from([0xff, 0x01]).readUInt16BE(0).toString(10))

// 小端输出 511

console.log(Buffer.from([0xff, 0x01]).readUInt16LE(0).toString(10))

除此之外, 7 + 64的模式还有一点额外的处理, 代码如下:

getPayloadLength64() {

**if** (**this**.\_bufferedBytes < 8) {

**this**.\_loop = false;

**return**;

}

**const** buf = **this**.consume(8);

**const** num = buf.readUInt32BE(0);

*//*

*// The maximum safe integer in JavaScript is 2^53 - 1. An error is returned*

*// if payload length is greater than this number.*

*//*

**if** (num > Math.pow(2, 53 - 32) - 1) {

**this**.\_loop = false;

**return** error(

RangeError,

'Unsupported WebSocket frame: payload length > 2^53 - 1',

false,

1009

);

}

**this**.\_payloadLength = num \* Math.pow(2, 32) + buf.readUInt32BE(4);

**return** **this**.haveLength();

}

**4.6 获得载荷数据: getData**

在获得载荷之前, 如果getInfo中mask为1, 需要进行getMask操作, 获取Mask Key(一共四个字节)

getMask() {

**if** (**this**.\_bufferedBytes < 4) {

**this**.\_loop = false;

**return**;

}

**this**.\_mask = **this**.consume(4);

**this**.\_state = GET\_DATA;

}

getData源码简化为如下

getData(cb) {

*// data为 Buffer.alloc(0)*

let **data** = EMPTY\_BUFFER;

*// 消费payload*

**data** = **this**.consume(**this**.\_payloadLength)

*// 如果有mask, 根据mask key进行解码, 此处不展开*

**if** (**this**.\_masked) unmask(**data**, **this**.\_mask)

*// 将其记录进分片数组*

**this**.\_fragments.push(**data**)

*// 如果该数据帧表示: 连接断开, 心跳请求, 心跳响应*

**if** (**this**.\_opcode > 0x07) **return** **this**.controlMessage(**data**)

*// 如果该数据帧表示: 数据分片、文本帧、二进制帧*

**return** **this**.dataMessage()

}

**4.7 组装载荷数据: dataMessage**

接着分析dataMessage()函数, 它用于将多个帧的数据合并, 简化之后也比较简单

dataMessage() {

**if** (**this**.\_fin) {

**const** messageLength = **this**.\_messageLength

**const** fragments = **this**.\_fragments

**const** buf = concat(fragments, messageLength)

**this**.emit('message', buf.toString())

}

}

*// 简明易懂哦, 不解释啦*

function concat(list, totalLength) {

**if** (list.length === 0) **return** EMPTY\_BUFFER;

**if** (list.length === 1) **return** list[0];

**const** target = Buffer.allocUnsafe(totalLength);

let offset = 0;

**for** (let i = 0; i < list.length; i++) {

**const** buf = list[i];

buf.copy(target, offset);

offset += buf.length;

}

**return** target;

}

**5. 总结**

**本文篇幅较长且并不是面试题那种小块的知识点, 阅读急需耐心, 已尽量避免贴大段代码, 能看到这里我都想给你打钱了**

通过本篇分析, 完整的介绍以及复现了WebSocket中的两个关键阶段:

* 连接握手阶段
* 数据交换极端

个人认为最关键便是: **涉及到了对Node.js的buffer模块以及stream模块的使用**, 这也是收获最大的一部分