

WebSocket 实现原理

 zeeyang.com/2017/07/02/websocket/


背景

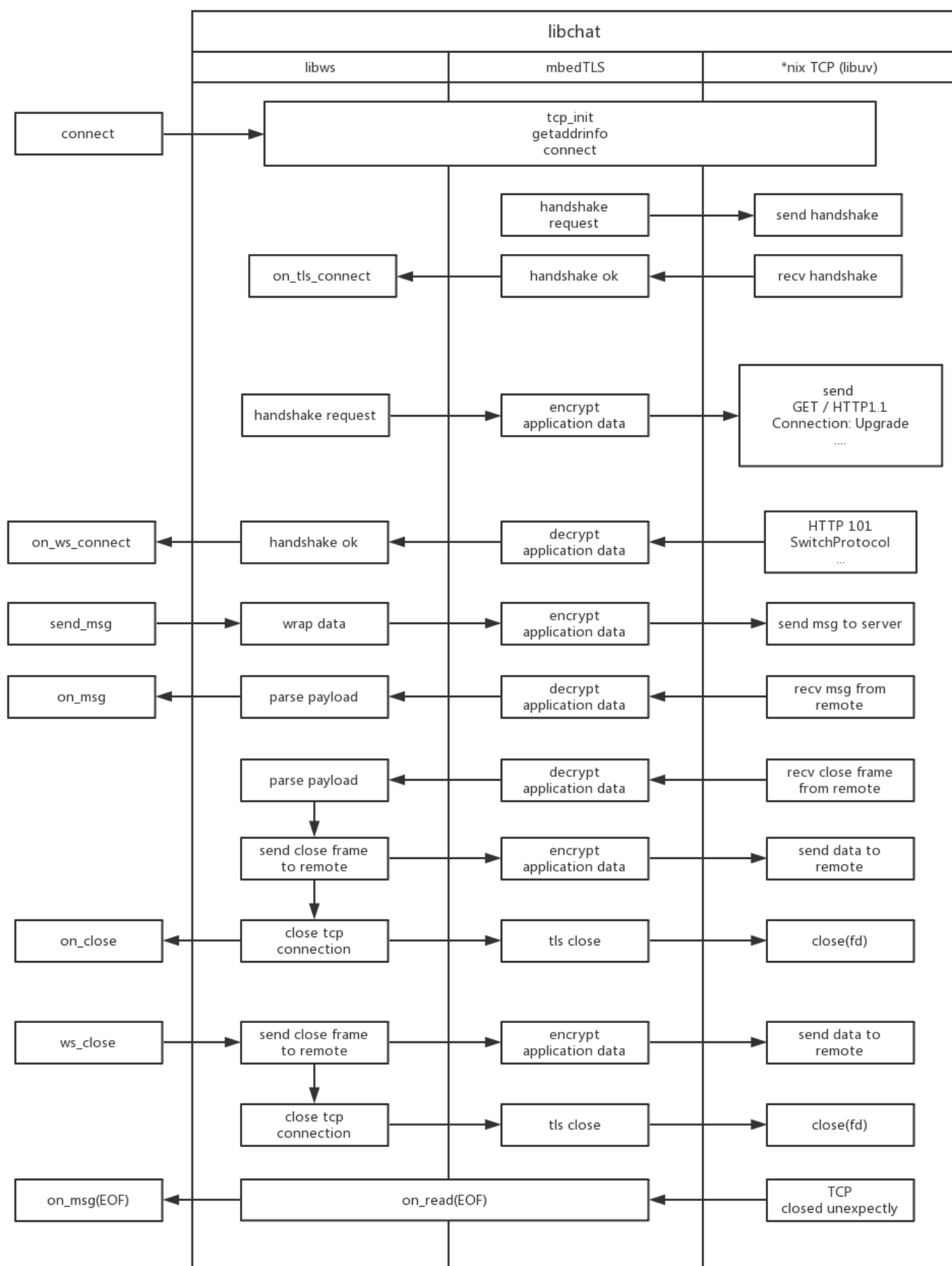
之前我们将 [CocoaAsyncSocket](#) 作为底层实现，在其上面封装了一套 Socket 通信机制以及业务接口，最近我们开始研究 WebSocket，并用来替换掉原先的 CocoaAsyncSocket，简单来说一下两者的关系，WebSocket 和 Socket 虽然名称上很像，但两者是完全不同的东西，WebSocket 是建立在 TCP/IP 协议之上，属于应用层的协议，而 Socket 是在应用层和传输层中的一个抽象层，它是将 TCP/IP 层的复杂操作抽象成几个简单的接口来提供给应用层调用。为什么要做这次替换呢？原因是我们服务端在做改造，同时网页版 IM 已经使用了 WebSocket，客户端也采用的话对于服务端来说维护一套代码会更好更方便，而且 WebSocket 在体积、实时性和扩展上都具有一定的优势。

WebSocket 最新的协议是 [13 RFC 6455](#)，要理解 WebSocket 的实现，一定要去理解它的协议！~

前言

WebSocket 的实现分为握手，数据发送/读取，关闭连接。

这里首先放上一张我们组 [@省长](#)（推荐大家去读一读省长的博客，干货很多 ）整理出来的流程图，方便大家去理解，其中mbedTLS做的是数据的加解密，可以暂时不用关心：



握手

握手要从请求头去理解。

WebSocket 首先发起一个 HTTP 请求，在请求头加上 `Upgrade` 字段，该字段用于改变 HTTP 协议版本或者是换用其他协议，这里我们把 `Upgrade` 的值设为 `websocket`，将它升级为 WebSocket 协议。

同时要注意 `Sec-WebSocket-Key` 字段，它由客户端生成并发给服务端，用于证明服务端接收到的是一个可受信的连接握手，可以帮助服务端排除自身接收到的由非 WebSocket 客户端发起的连接，该值是一串随机经过 `base64` 编码的字符串。

```
GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key:
dGhlIHNBbXBsZSBub25jZQ==

Origin: http://example.com

Sec-WebSocket-Protocol: chat, superchat

Sec-WebSocket-Version: 13
```

我们可以简化请求头，将请求以字符串方式发送出去，当然别忘了最后的两个空行作为包结束：

```
const char * fmt = "GET %s HTTP/1.1\r\n"

    "Upgrade: websocket\r\n"

    "Connection: Upgrade\r\n"

    "Host: %s\r\n"

    "Sec-WebSocket-Key: %s\r\n"

    "Sec-WebSocket-Version: 13\r\n"

    "\r\n";

size = strlen(fmt) + strlen(path) + strlen(host) + strlen(ws->key);

buf = (char *)malloc(size);

sprintf(buf, fmt, path, host, ws->key);

size = strlen(buf);

nbytes = ws->io_send(ws, ws->context, buf, size);
```

收到请求后，服务端也会做一次响应：

```
HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

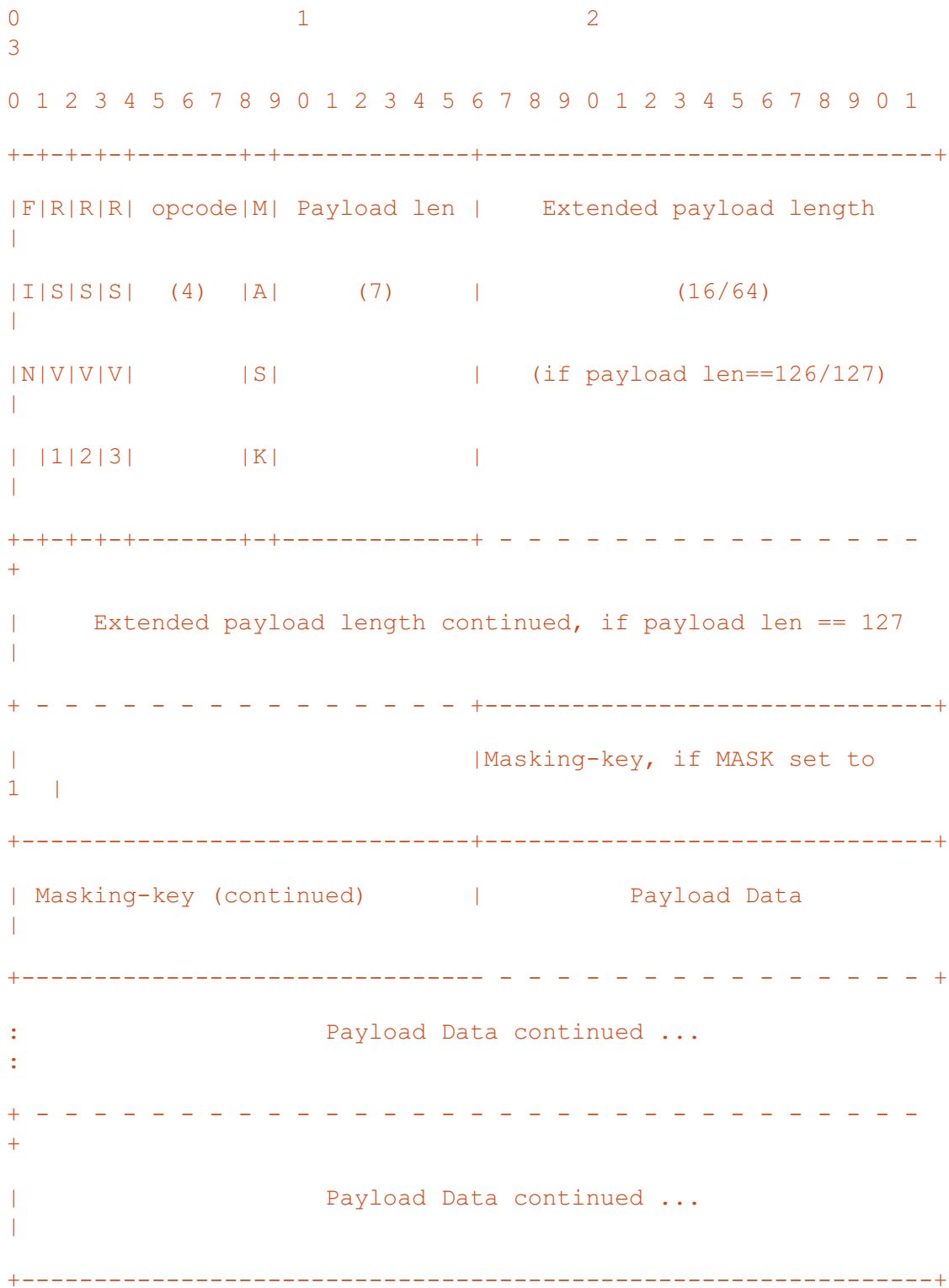
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

里面重要的是 `Sec-WebSocket-Accept`，服务端通过从客户端请求头中读取 `Sec-WebSocket-Key` 与一串全局唯一的标识字符串（俗称魔串）“258EAFa5-E914-47DA-95CA-C5AB0DC85B11”做拼接，生成长度为160位的 `SHA-1` 字符串，然后进行 `base64` 编码，作为 `Sec-WebSocket-Accept` 的值回传给客户端，客户端再去解析这个值，与自己加密编码后的字符串进行比较。

处理握手 HTTP 响应解析的时候，可以用 nodejs 的 `http-paser`，解析方式也比较简单，就是对头信息的逐字读取再处理，具体处理你可以看一下它的状态机实现。解析完成后你需要对其内容进行解析，看返回是否正确，同时去管理你的握手状态。

数据发送/读取

数据的处理就要拿这个帧协议图来说明了：



首先我们来看看数字的含义，数字表示位，0-7表示有8位，等于1个字节。

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
0 1
```

所以如果要组装一个帧数据可以这样子：

```
char *rev = (rev
*)malloc(4);

rev[0] = (char)(0x81 &
0xff);

rev[1] = 126 & 0x7f;

rev[2] = 1;

rev[3] = 0;
```

ok，了解了帧数据的样子，我们反过来去理解值对应的帧字段。

首先0x81是什么，这个是十六进制数据，转换成二进制就是1000 0001，是一个字节的长度，也就是这一段里面每一位的值：

```
0 1 2 3 4 5 6 7
8

+-+--+--+-----+

|F|R|R|R| opcode|

|I|S|S|S| (4)
|

|N|V|V|V|
|

| |1|2|3|
|

+-+--+--+-----+
```

- FIN 表示该帧是不是消息的最后一帧，1表示结束，0表示还有下一帧。

RSV1, RSV2,

- RSV3 必须为0，除非扩展协商定义了一个非0的值，如果没有定义非0值，且收到了非0的 RSV，那么 WebSocket 的连接会失效，建议是断开连接。

- `opcode` 用来描述 `Payload data` 的定义，如果收到了一个未知的 `opcode`，同样会使 WebSocket 连接失效，协议定义了以下值：
 - `%x0` 表示连续的帧
 - `%x1` 表示 text 帧
 - `%x2` 表示二进制帧
 - `%x3-7` 预留给非控制帧
 - `%x8` 表示关闭连接帧
 - `%x9` 表示 ping
 - `%xA` 表示 pong
 - `%xB-F` 预留给控制帧

连续帧是和 FIN 值相关联的，它表明可能由于消息分片的原因，将原本一个帧的数据分为多个帧，这时候前一帧的 `opcode` 就是 0，FIN 也是 0，最后一帧的 `opcode` 就不再是 0，FIN 就是 1 了。

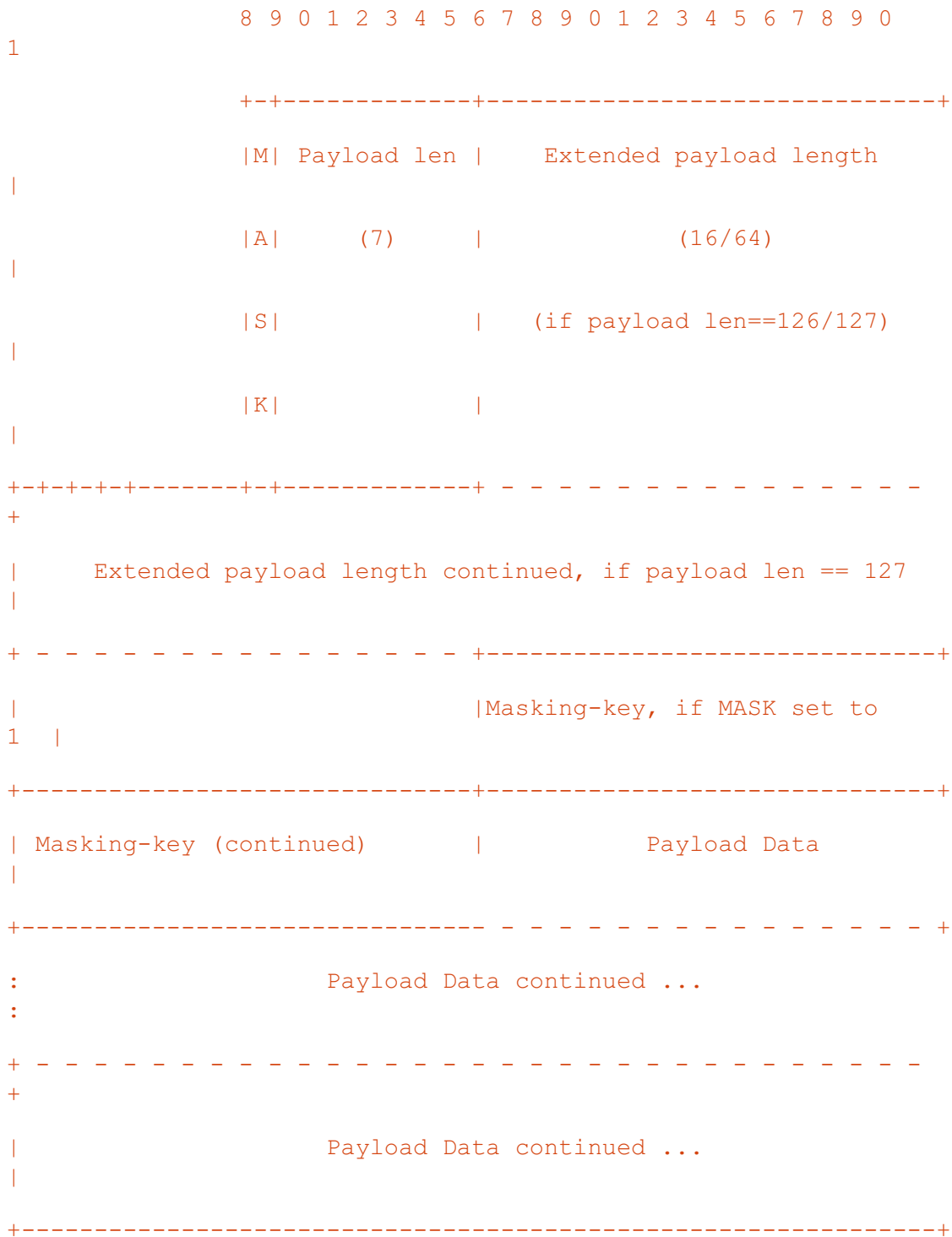
再可以看到 `opcode` 预留了非控制帧和控制帧，这两个又是什么？

控制帧表示 WebSocket 的状态信息，像是定义的分片，关闭连接，ping和pong。

非控制帧就是数据帧，像是 text 帧，二进制帧。

`0xff` 作用就是取出需要的二进制值。

下面再来看 `126`，`126`则表示的是 `len` ^{`Payload`}，也就是 `Payload` 的长度：



- **MASK** 表示 **data** 是否要加掩码，如果设成1，则需要赋值 **Masking-key**。所有从客户端发到服务端的帧都要加掩码
- **len** 表示 **Payload** 的长度，这里分为三种情况
 - 长度小于126，则只需要7位

- 长度是126，则需要额外2个字节的大小，也就是 `length` `Extended payload`
- 长度是127，则需要额外8个字节的大小，也就是 `length` `Extended payload` +
`Extended payload length` `Extended payload`
`continued` , `length` 是2个字节，
`Extended payload length`
`continued` 是6个字节

- `len` `Payload` 则表示 `Extension data` 与 `Application data` 的和
- `Masking-key` 是在 `MASK` 设置成1之后，随机生成的4字节长度的数据，然后和 `Payload Data` 做异或运算
- `Payload Data` 就是我们发送的数据

而数据的发送和读取就是对帧的封装和解析。

数据发送：

```
int ws__wrap_packet(_WS_IN websocket_t *ws,
                   _WS_IN const char *payload,
                   _WS_IN unsigned long long payload_size,
                   _WS_IN int flags,
                   _WS_OUT char** out,
                   _WS_OUT uint64_t *out_size) {
    struct timeval tv;
    char mask[4];
    unsigned int mask_int;
    unsigned int payload_len_bits;
    unsigned int payload_bit_offset = 6;
    unsigned int extend_payload_len_bits, i;
    unsigned long long frame_size;
    const int MASK_BIT_LEN = 4;
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec * tv.tv_sec);
    mask_int = rand();
    memcpy(mask, &mask_int, 4);

    *payload_len_bits
```

```

    payload_len_bits

* ref to https://tools.ietf.org/html/rfc6455#section-5.2

* If 0-125, that is the payload length

*

* If payload length is equals 126, the following 2 bytes interpreted
as a

* 16-bit unsigned integer are the payload length

*

* If 127, the following 8 bytes interpreted as a 64-bit unsigned
integer (the

* most significant bit MUST be 0) are the payload length.

*/

if (payload_size <= 125) {

    extend_payload_len_bits = 0;

    frame_size = 1 + 1 + MASK_BIT_LEN + payload_size;

    payload_len_bits = payload_size;

} else if (payload_size > 125 && payload_size <= 0xffff) {

    extend_payload_len_bits = 2;

    frame_size = 1 + 1 + extend_payload_len_bits + MASK_BIT_LEN +
payload_size;

    payload_len_bits = 126;

    payload_bit_offset += extend_payload_len_bits;

} else if (payload_size > 0xffff && payload_size <=
0xffffffffffffffffLL) {

    extend_payload_len_bits = 8;

    frame_size = 1 + 1 + extend_payload_len_bits + MASK_BIT_LEN +
payload_size;

    payload_len_bits = 127;

    payload_bit_offset += extend_payload_len_bits;

} else {

    if (ws->error cb) {

```

```

        -
        ws_error_t *err = ws_new_error(WSEND_DATA_TOO_LARGE_ERR);

        ws->error_cb(ws, err);

        free(err);

    }

    return WSEND_SEND_ERR;
}

*out_size = frame_size;

char *data = (*out) = (char *)malloc(frame_size);

if (data == NULL) {

    if (ws->error_cb) {

        ws_error_t *err = ws_new_error(WSEND_SEND_ERR);

        ws->error_cb(ws, err);

        free(err);

    }

    return -ENOMEM;

}

char *buf_offset = data;

bzero(data, frame_size);

*data = flags & 0xff;

buf_offset = data + 1;


*(buf_offset) = payload_len_bits | 0x80;

buf_offset = data + 2;

if (payload_len_bits == 126) {

    payload_size &= 0xffff;

} else if (payload_len_bits == 127) {

    payload_size &= 0xffffffffffffffffLL;

}

for (i = 0; i < extend_payload_len_bits; i++) {

    *(buf_offset + i) = *((char *)&payload_size +
(extend_payload_len_bits - i - 1));

```

```

    }

    * according to https://tools.ietf.org/html/rfc6455#section-5.3
    *
    * buf_offset is set to mask bit
    */
    buf_offset = data + payload_bit_offset - 4;
    for (i = 0; i < 4; i++) {
        *(buf_offset + i) = mask[i] & 0xff;
    }

    * mask the payload data
    */
    buf_offset = data + payload_bit_offset;
    memcpy(buf_offset, payload, payload_size);
    mask_payload(mask, buf_offset, payload_size);
    return OK;
}

void mask_payload(char mask[4], char *payload, unsigned long long
payload_size) {
    unsigned long long i;
    for(i = 0; i < payload_size; i++) {
        *(payload + i) ^= mask[i % 4] & 0xff;
    }
}

```

数据解析：

```

int ws_recv(websocket_t *ws) {
    if (ws->state < WS_STATE_HANDSHAKE_COMPLETED) {

```

```

        return ws_do_handshake(ws);
    }

    int ret;

    while(true) {

        ret = ws__recv(ws);

        if (ret != OK) {

            break;

        }

    }

    return ret;
}

int ws__recv(websocket_t *ws) {

    if (ws->state < WS_STATE_HANDSHAKE_COMPLETED) {

        return ws_do_handshake(ws);

    }

    int ret = OK, i;

    int state = ws->rd_state;

    char *rd_buf;

    switch(state) {

        case WS_READ_IDLE: {

            ret = ws__make_up(ws, 2);

            if (ret != OK) {

                return ret;

            }

            ws_frame_t * frame;

            if (ws->c_frame == NULL) {

                ws__append_frame(ws);

            }

            frame = ws->c_frame;

            rd_buf = ws->buf;

```

```

frame->fin = (*(rd_buf) & 0x80) == 0x80 ? 1 : 0;

frame->op_code = *(rd_buf) & 0x0fu;

frame->payload_len = *(rd_buf + 1) & 0x7fu;

if (frame->payload_len < 126) {

    frame->payload_bit_offset = 2;

    ws->rd_state = WS_READ_PAYLOAD;

} else if (frame->payload_len == 126) {

    frame->payload_bit_offset = 4;

    ws->rd_state = WS_READ_EXTEND_PAYLOAD_2_WORDS;

} else {

    frame->payload_bit_offset = 8;

    ws->rd_state = WS_READ_EXTEND_PAYLOAD_8_WORDS;

}

ws__reset_buf(ws, 2);

break;

}

case WS_READ_EXTEND_PAYLOAD_2_WORDS: {

#define PAYLOAD_LEN_BITS 2

    ret = ws__make_up(ws, PAYLOAD_LEN_BITS);

    if (ret != OK) {

        return ret;

    }

    rd_buf = ws->buf;

    ws_frame_t * frame = ws->c_frame;

    char *payload_len_bytes = (char *)&frame->payload_len;

    for (i = 0; i < PAYLOAD_LEN_BITS; i++) {

        *(payload_len_bytes + i) = rd_buf[PAYLOAD_LEN_BITS - 1 -

i];

    }

    ws__reset_buf(ws, PAYLOAD_LEN_BITS);

    ws->rd_state = WS_READ_PAYLOAD;

```

```

#undef PAYLOAD_LEN_BITS

        break;

    }

    case WS_READ_EXTEND_PAYLOAD_8_WORDS: {

#define PAYLOAD_LEN_BITS 8

        ret = ws__make_up(ws, PAYLOAD_LEN_BITS);

        if (ret != OK) {

            return ret;

        }

        rd_buf = ws->buf;

        ws_frame_t * frame = ws->c_frame;

        char *payload_len_bytes = (char *)&frame->payload_len;

        for (i = 0; i < PAYLOAD_LEN_BITS; i++) {

            *(payload_len_bytes + i) = rd_buf[PAYLOAD_LEN_BITS - 1 -

i];

        }

        ws__reset_buf(ws, PAYLOAD_LEN_BITS);

        ws->rd_state = WS_READ_PAYLOAD;

#undef PAYLOAD_LEN_BITS

        break;

    }

    case WS_READ_PAYLOAD: {

        ws_frame_t * frame = ws->c_frame;

        uint64_t payload_len = frame->payload_len;

        ret = ws__make_up(ws, payload_len);

        if (ret != OK) {

            return ret;

        }

        rd_buf = ws->buf;

        frame->payload = malloc(payload_len);

        memcpy(frame->payload, rd_buf, payload_len);

        ws__reset_buf(ws, payload_len);

```

```

        ws__reset_buf(ws, payload_len);

        if (frame->fin == 1) {

            ws__dispatch_msg(ws, frame);

            ws__clean_frame(ws);

        } else {

            ws__append_frame(ws);

        }

        ws->rd_state = WS_READ_IDLE;

        break;

    }

}

return ret;

}

```

关闭连接

关闭连接分为两种：服务端发起关闭和客户端主动关闭。

服务端跟客户端的处理基本一致，以服务端为例：

服务端发起关闭的时候，会客户端发送一个关闭帧，客户端在接收到帧的时候通过解析出帧的opcode来判断是否是关闭帧，然后同样向服务端再发送一个关闭帧作为回应。


```
if (op_code == OP_CLOSE) {
    int status_code;

    char *reason;

    char *status_code_buf = (char
*) &status_code;

    status_code_buf[0] = payload[1];
    status_code_buf[1] = payload[0];

    reason = payload + 2;

    if (ws->state != WS_STATE_CLOSED) {

        /* should send response to remote server
        */

        ws_send(ws, NULL, 0, OP_CLOSE |
FLAG_FIN);

        ws->state = WS_STATE_CLOSED;
    }

    if (ws->close_cb) {
        ws->close_cb(ws, status_code, reason);
    }
}
```

总结

对WebSocket的学习主要是对协议的理解，理解了协议，上面复杂的代码自然而然就会明白~

后记

对于I/O操作的原理，推荐大家可以看看这个：[epoll 或者 kqueue 的原理是什么？](#)