游戏抓包工具制作（四）—— 其他连接方式的实现（WebSocket）

原创 阿宽 游戏小测试 2024-04-08 19:07 广东

随着游戏载体的多样化，现在的游戏也不再局限于Tcp和Udp，小游戏和H5游戏的出现，尤其是可以多端输出的引擎的出现，让更多的游戏公司开始使用websocket来进行前后端的协议交互，所以抓包工具也要与时俱进，支持对websocket协议的转发与解析🐱。

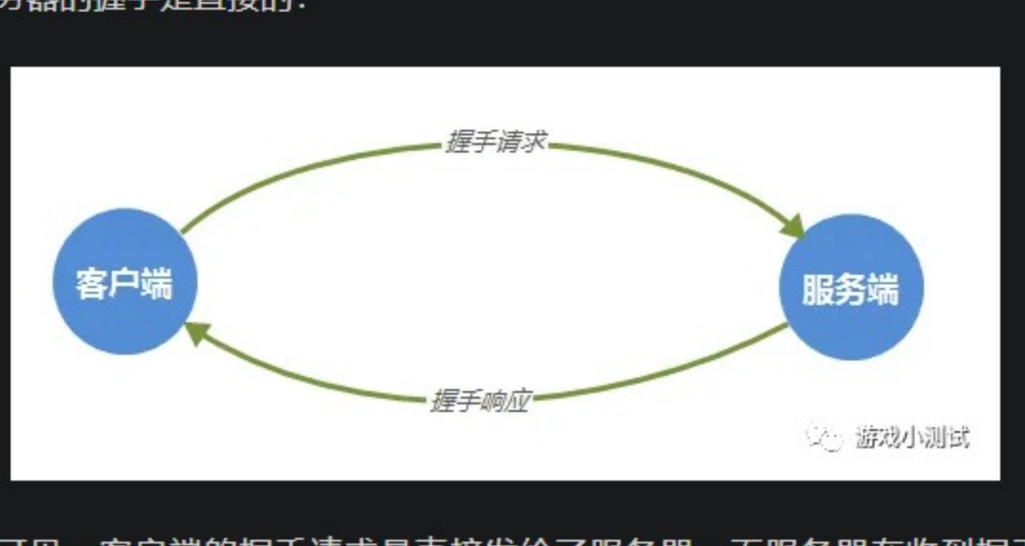
之前看了一下websocket-client的代码，发现它在底层也是使用的socket库，然后我又问了一下度娘，了解到**websocket是一种在单个Tcp连接上进行全双工通讯的协议，是建立在Tcp的基础之上的**，所以从道理上来讲，我们的代码不需要做什么大的修改，就应该能实现协议的转发（经测试也确实是如此），那么跟Tcp协议的区别在哪里呢？主要有两点：

一是**websocket协议内容的二次封装**，二是**websocket的握手**，而握手又分两种：ws连接方式的握手和wss连接方式的握手，所以今天的分享会分为两个大块来讲，内容可能有点多，还请搬好小板凳耐心的看完~

PS：websocket的解析参考了 武沛齐 老师的文章，给我带来了很大的帮助🥹。

首先来说一下websocket协议的握手。在WebSocket握手阶段，客户端会发送一个特殊的HTTP请求到服务器，请求头中包含"Upgrade: websocket"字段，表示请求将协议升级为WebSocket协议。服务器收到请求后，会返回一个响应，响应码为101，表示同意将协议进行升级。响应头中包含"Sec-WebSocket-Accept"字段，这个字段的值是从请求的"Sec-WebSocket-key"字段继承而来，包含一个特殊的响应值，必须和客户端预期精确匹配才能握手成功。一旦握手成功，WebSocket连接就建立起来了，服务器和客户端之间就可以进行双向通信。以上这些内容是我抄来的，仅供参考，如有需要，建议自行查阅相关文档🥴

话不多说，我们现在来分析一下握手的这个过程，在代理没有当第三者的时候，客户端与服务器的握手是直接的：



由图可见，客户端的握手请求是直接发给了服务器，而服务器在收到握手请求之后，也是将响应直接返回给客户端。

而代理介入客户端和服务器当中间商的时候，他们之间的握手有了一些变化：

1.客户端的握手请求不再发给服务器，而是发到代理这里来

2.代理在收到客户端的握手请求之后，先发个假的响应给客户端，让客户端先建立与代理的连接（欺骗客户端你已经连上服务器了，尽情发包吧）

3.与此同时，代理从客户端的握手请求中获取到它的身份信息（"Sec-WebSocket-key"），然后用这个身份信息伪造一个客户端的请求发给服务器

4.由于身份信息是正常的，所以服务器会认为是正常的客户端发过来的请求，然后会对握手请求做出响应，代理在收到服务器返回的握手响应之后，与服务器的连接就也创建完成了。

至此，中间商的身份才算坐实了🤔。

接下来我们来实现这个过程，从简单的开始，我们先写一个基于ws握手方式的方法：

1.代理模拟服务器来响应客户端的握手请求

```python
def client_handshake_response(self):
    """
    模拟响应客户端的握手请求
    :return:
    """
    # 获取并解析握手请求
    data = self.client_socket.recv(1024)
    request = data.decode()
    # 解析握手请求中的关键信息：Sec-WebSocket-Key，并保存下来备用
    for line in request.split('\r\n'):
        if line.startswith('Sec-WebSocket-Key:'):
            self.key = line.split(':')[1].strip()
            break
    # 生成响应的key
    response_key = base64.b64encode(hashlib.sha1((self.key + '258EAFA5-E914-
                                          encode()).digest()).decode()
    # 生成握手响应信息
    response = 'HTTP/1.1 101 Switching Protocols\r\n'
    response += 'Upgrade: websocket\r\n'
    response += 'Connection: Upgrade\r\n'
    response += 'Sec-WebSocket-Accept: ' + response_key + '\r\n\r\n'
    # 发送握手响应给客户端
    self.client_socket.send(response.encode())
```

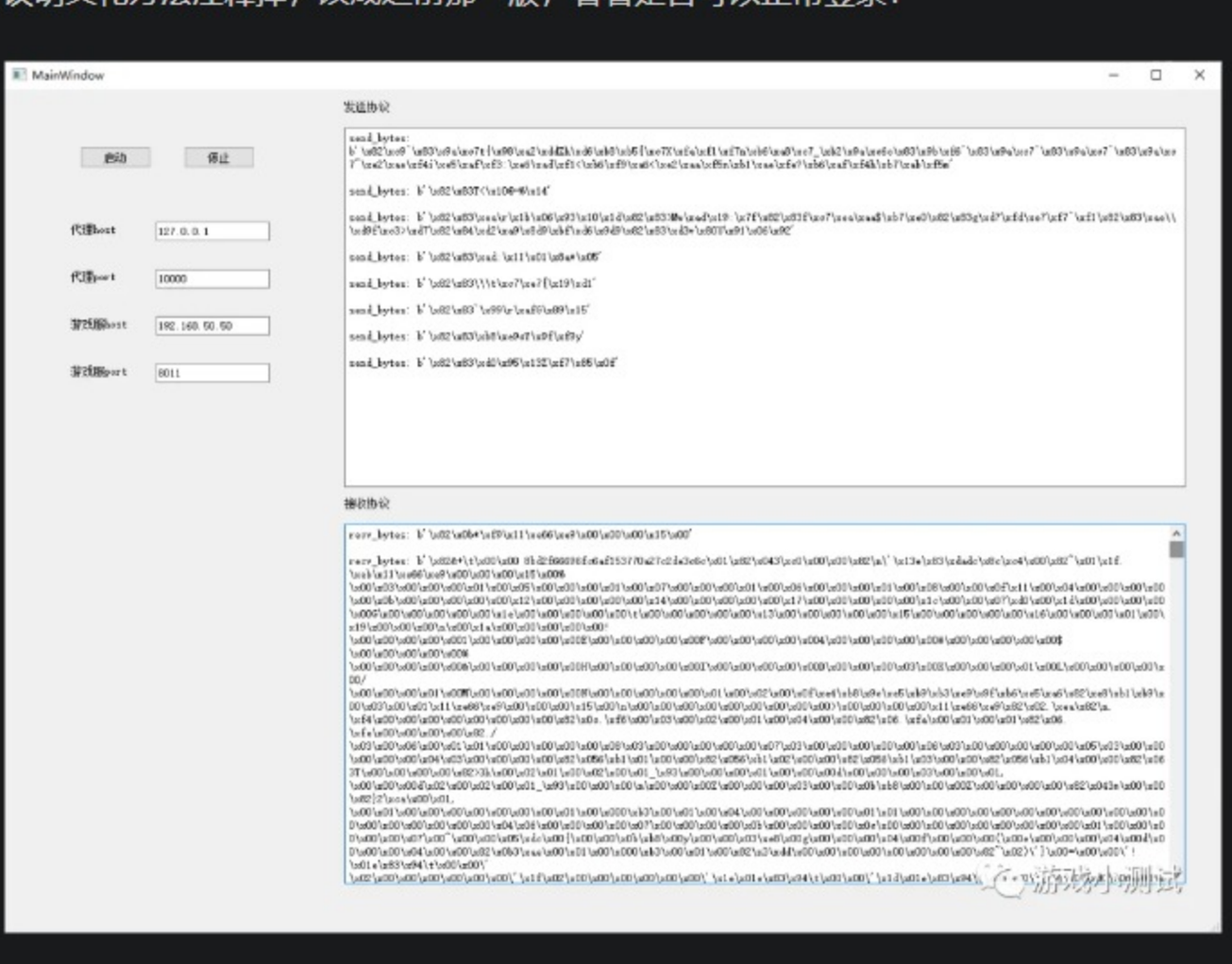
2.代理模拟客户端向服务器发起握手请求

```python
def server_handshake_request(self, host, port):
    """
    向服务器发起握手请求
    :param host:
    :param port:
    :return:
    """
    request = 'GET / HTTP/1.1\r\n'
    request += f'Host: {host}:{port}\r\n'
    request += 'Upgrade: websocket\r\n'
    request += 'Connection: Upgrade\r\n'
    request += 'Sec-Websocket-Extensions: permessage-deflate; client_max_winc
    request += 'Sec-WebSocket-Key: ' + self.key + '\r\n'
    request += 'Sec-WebSocket-Version: 13\r\n\r\n'
    # 发送握手请求给服务器
    self.server_socket.send(request.encode())
    # 接收服务器的握手响应
    self.server_socket.recv(1024)
```

基于ws握手的方法就写完了，由于我们之前写的tcp启动方法是没有握手这个环节的，所以我们需要对之前的代码进行一点小小的修改，以便让客户端和服务器收发协议之前能成功握手，找到我们之前的start()方法，修改如下(由于这里的握手方法再次用到了服务器的host和port，所以我在__init__方法中把这两个值保存为agent的属性了，记得改一下哦~)：

```python
def start(self):
    """
    代理启动方法
    :return:
    """
    # 等待客户端连接
    self.client_socket, addr = self.socket_service.accept()
    # 代理模拟服务器，向客户端响应握手
    self.client_handshake_response()
    # 代理模拟客户端，向服务器发起握手请求
    # 由于这里要传入服务器的ip和端口
    # 因此需要我们在__init__方法中，将ip和端口保存成自己的属性以备后用
    self.server_handshake_request(self.server_host, self.server_port)
    # 启动转发线程，这里用的是Timer定时器，它也是继承自thread线程类，所以会单独起两个
    Timer(0, self.client_to_server).start()
    Timer(0, self.server_to_client).start()
```

由于websocket协议自己封装了一套加密方法，所以我们先把上篇文章中提到的协议明文化方法注释掉，改成之前那一版，看看是否可以正常登录：



经测试是可以登录的，说明基于ws连接方式的握手是没有问题的😁。

接下来我们来实现基于wss连接方式的握手，首先需要说明的是，wss属于安全连接协议，以我的能力无法进行破解，如果客户端是通过wss方式发起的请求，单单修改服务器列表中的连接地址肯定是没有用的，因为代理无法给客户端响应正确的wss握手协议，所以如果服务器是基于wss连接方式的话，我的做法是，**客户端给代理服发起的仍然是ws连接，代理在获取到客户端请求之后给客户端返回ws握手响应，然后向服务器发起一个模拟wss方式的握手申请，让代理与服务器创建正常的wss连接，与客户端创建不那么安全的ws连接。**

关于响应客户端的握手请求，由于仍然是ws，所以没有必要对上面的握手代码进行修改，向服务器发起握手请求的方法，也可以不进行修改，我们需要修改的是server_socket这个连接，如果要实现wss连接的话，就不能再用原来的socket了，需要对它做出一些包装。

导入ssl库，对agent的__init__()方法中的self.server_socket进行包装：

```python
class Agent(object):
    def __init__(self, agent_host, agent_port, server_host, server_port, ui_t
        """
        代理初始化一个实例
        :param agent_host: 代理绑定的ip
        :param agent_port: 代理绑定的端口
        :param server_host: 游戏服务器的ip
        :param server_port: 游戏服务器的端口
        """
        self.ui_thread = ui_thread
        # 设置一个socket server，以便后续监听客户端请求
        self.socket_service = socket.socket(socket.AF_INET, socket.SOCK_STREA
        # 将socket server绑定指定的ip和端口号
        self.socket_service.bind((agent_host, agent_port))
        self.socket_service.listen(5)
        # 设置客户端socket属性，先把它设置成一个空对象，后续有客户端请求过来之后再更
        self.client_socket = None
        self.server_host = server_host
        self.server_port = server_port
        # 设置服务器socket属性，生成一个socket对象
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM
        # 创建SSL Context
        context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
        context.check_hostname = False
        context.verify_mode = ssl.CERT_NONE
        # 使用 SSL context 包装 server_socket，创建一个 SSL socket
        self.server_socket = context.wrap_socket(self.server_socket,
                                                 server_hostname=f'{server_ho
        # 将server_socket连接到游戏服务器
        self.server_socket.connect((self.server_host, self.server_port))
        # 给代理设置一个是否存活的状态，用于在遇到某些异常的时候终止转发线程
        self.alive = True
```

接下来就跟ws的连接方式一毛一样了，经测试也是可以成功连接上基于wss连接方式的服务器的，图我就不上了，省点流量😂。

PS：请根据服务器的连接类型选择不同的连接方式哦，如果服务器是ws的连接方式，那么使用wss方式连接的话会出错的哦，请根据服务器的实际情况选择是否要对server socket进行封装。

连接的问题解决了，接下来就是解析了。ws和wss连接的协议加密和解密方式是一样的，所以这里就不用区分是ws还是wss了，只要写一套加密解密的方法就可以了。

不过在这里要区分一下是客户端发送的包，还是服务器返回的包，服务器返回的包是没有掩码处理的，你可以认为跟tcp一样是明文的，直接解析就好了，而客户端发给服务器的包是进行了加密处理的，我们无法直接解析，在解析之前，需要先把它进行还原处理。

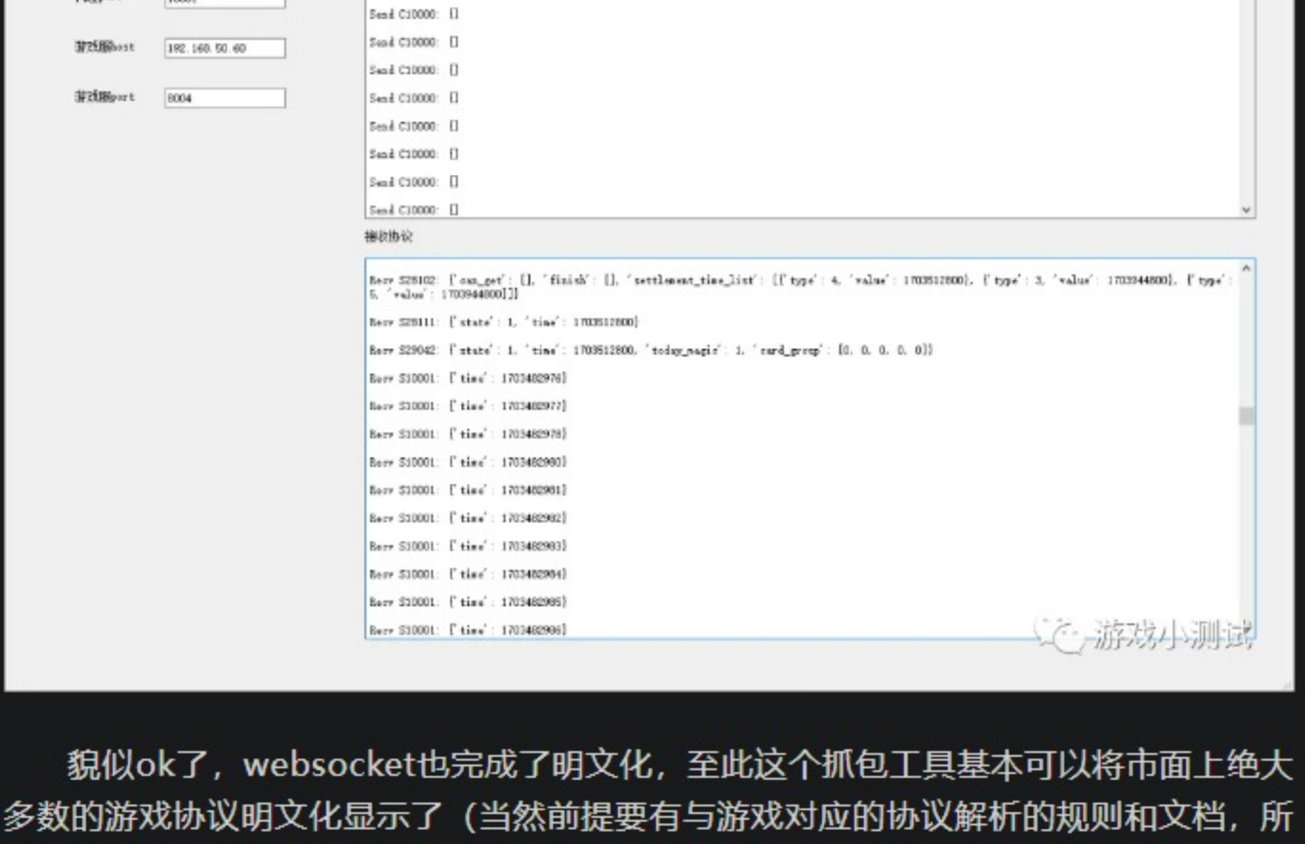
1.websocket服务端返回协议明文化显示

```python
def server_to_client(self):
    """
    处理服务器发送给客户端的包
    1.获取服务器返回的字节流
    2.获取原始的封包内容
    3.调用协议明文化方法
    4.将服务器返回的字节流通过client socket转发给客户端
    3.循环此步骤
    :return:
    """
    while self.alive:
        # 获取协议头
        head_data = self.server_socket.recv(2)
        # 解析类型和长度
        opcode = head_data[0] & 0x0f
        payload_length = head_data[1] & 0x7f
        # 如果是bytes类型，则根据payload_length判断协议长度
        # 解释一下：payload_length保存的是数据的长度，如果长度小于126
        #          则payload_length就是真正的协议长度
        #          如果长度解析出来是126，则说明长度比较大，是用2个字节来保存
        #          我们需要读取2个字节来解析出真正的协议字节长度
        #          如果长度解析出来是127，说明长度巨大，用4个字节来保存的长度信息
        #          我们需要读取4个字节来解析出真正的协议长度
        if opcode == 0x2:
            if payload_length == 126:
                length_data = self.server_socket.recv(2)
                payload_length = struct.unpack(">H", length_data)[0]
            elif payload_length == 127:
                length_data = self.server_socket.recv(8)
                payload_length = struct.unpack(">Q", length_data)[0]
            elif payload_length < 126:
                length_data = b''
            else:
                length_data = b''
                # 正常来讲不会到这里来，但还是加个打印吧，防止出现解析错误的情况
                print(f'出错了，长度信息无法解析')
            # 再根据真实的协议长度，读取出ws协议的字节流，这个字节流中包含的是原始的
            proto_data = self.server_socket.recv(payload_length)
            # 解析出协议号（需要与前后端协议封装方式一致，以4个字节保存协议号为例）
            # ps:此例协议结构为：协议号+协议参数 封装的字节流
            proto_id = struct.unpack('!I', proto_data[:4])[0]
            # 然后根据协议号proto_id生成协议对象，这个协议对象初始化的时候不传入参数
            # 这里我用了eval方法，这个方法可以将字符串当作代码运行，获取到对应的值（
            # 你也可以创造一个字典，用协议号作为key， 协议类作为value，通过协议号获
            proto_obj = eval(f'S{proto_id}()')
            # 调用decode方法，将字节进行解析，并将值赋予协议对象的对应的属性
            proto_obj.decode(proto_data[4:])
            # 然后再调用对象明文化的方法， 获取到对象的明文字典
            self.ui_thread.recv_signal.emit(f'Recv S{proto_id}: {dispose_obj
            # 将协议转发给客户端
            self.client_socket.send(head_data + length_data + proto_data)
        else:
            # 如果收到的不是bytes类型，说明已经出错了，直接关闭吧
            print(f'出错了，收到了非bytes类型的数据')
            self.alive = False
```

2.websocket客户端发送协议解密

```python
def client_to_server(self):
    """
    转发客户端发送给服务器的包
    1.获取发送的封包头信息
    2.获取发送封包的长度信息
    3.获取掩码内容
    4.获取加密后的协议内容
    5.协议解密
    6.调用协议明文化方法
    7.将协议转发给服务器
    8.重复此步骤
    :return:
    """
    while self.alive:
        # 获取协议头
        head_data = self.client_socket.recv(2)
        # 解析类型和长度
        opcode = head_data[0] & 0x0f
        payload_length = head_data[1] & 0x7f
        # 解析是否掩码加密，正常来讲客户端发给服务器的协议是掩码加密的，如果不是可能
        masked = head_data[1] & 0x80
        if not masked:
            print(f"处理字节流失败，masked类型为非加密类型{masked}，可能不是客户端
            self.alive = False
            return
        # 如果是bytes类型，则根据payload_length判断协议长度
        if opcode == 0x2:
            # 如果长度是126，则需要再读取2个字节解析真正的长度
            if payload_length == 126:
                length_data = self.client_socket.recv(2)
                payload_length = struct.unpack(">H", length_data)[0]
            elif payload_length == 127:
                length_data = self.client_socket.recv(8)
                payload_length = struct.unpack(">Q", length_data)[0]
            elif payload_length < 126:
                length_data = b''
            else:
                length_data = b''
                # 正常来讲不会到这里来，但还是加个打印吧，防止出现解析错误的情况
                print(f'出错了，长度信息无法解析')
            # 读取4位掩码
            masks_data = self.client_socket.recv(4)
            # 读取真正的协议封包内容
            proto_data = self.client_socket.recv(payload_length)
            # 创建一个字节流对象，用于保存还原后的真实的协议封包内容
            true_proto_data = bytearray()
            # 对掩码后的协议进行还原操作
            for proto_byte in proto_data:
                proto_byte ^= masks_data[len(true_proto_data) % 4]
                true_proto_data.append(proto_byte)
            # 解析出协议号（需要与前后端协议封装方式一致，以4个字节保存协议号为例）
            # ps:此例协议结构为：协议号+协议参数 封装的字节流
            proto_id = struct.unpack('!I', true_proto_data[:4])[0]
            # 然后根据协议号proto_id生成协议对象
            proto_obj = eval(f'C{proto_id}()')
            # 调用decode方法，将字节进行解析，并将值赋予协议对象的对应的属性
            proto_obj.decode(true_proto_data[4:])
            # 然后再调用对象明文化的方法，获取到对象的明文字典
            self.ui_thread.send_signal.emit(f'Send C{proto_id}: {dispose_obj
            # 将协议转发给服务器
            self.server_socket.send(head_data + length_data + masks_data + pr
        else:
            # 如果收到的不是bytes类型，说明已经出错了，直接关闭吧
            print(f'出错了，收到了非bytes类型的数据')
            self.alive = False
```

然后我们再测试一下：



貌似ok了，websocket也完成了明文化，至此这个抓包工具基本可以将市面上绝大多数的游戏协议明文化显示了（当然前提要有与游戏对应的协议解析的规则和文档，所以这个工具更多的是来**辅助测试**的，而**不是用来做协议外挂的**😲）。

从下篇文章开始，会介绍如何伪造一个客户端的协议，包括Tcp的协议伪造，以及websocket的协议伪造，协议伪造和发包是实现协议测试的核心内容，有了它，我们就可以绕开客户端的限制，对服务器直接发送伪造协议了，如果我的文章对你有帮助，还希望多多支持和转发，谢谢各位客官🤔

#抓包工具 15 #测试工具 10

测试工具 · 目录

〈上一篇 · 游戏抓包工具制作（三）—— 协议解析方法

个人观点，仅供参考