

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，尤其可以以多端输出的引擎的出现，让更多的游戏开始使用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协议传输的二次封装**，二是**websocket的握手**，而握手又分两种：ws连接方式的握手+ws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，还请做好心理准备哈~

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🙏。

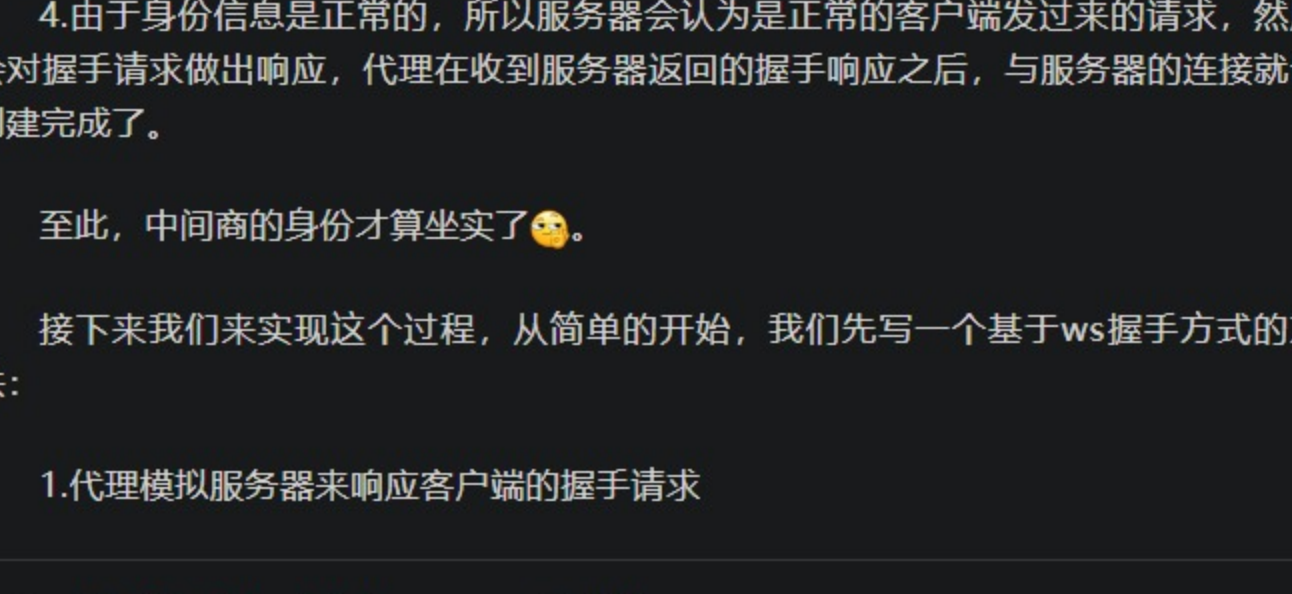
首先来说一下websocket协议的握手，在WebSocket握手阶段，客户端会发送一个特殊的HTTP请求到服务器，请求头中包含“Upgrade: websocket”字段，表示请求将协议升级为WebSocket协议。服务器收到请求后，会返回一个响应，响应码为101，表示同意将协议进行升级。响应头中包含“Sec-WebSocket-Accept”字段，这个字段的值是从请求的“Sec-WebSocket-key”字段推导而来，包含一个特殊的响应值，必须和客户端预期精确匹配才能握手成功。一旦握手成功，WebSocket连接就建立起来了，服务器和客户端之间就可以进行双向通信。以上这些内容是我抄来的，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

差不多说，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：



由此可见，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端当中间接的时候，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，先会个响的响应给客户端，让客户端先建立与代理的连接（感谢客户端你已经登上服务器了，尽情享受吧）
- 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构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，与服务器的连接就成功建立完成了。

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

```
1 def client_handshake_response(self):
2     """
3     模拟响应客户端握手请求
4     :return:
5     """
6     # 获取并解析握手请求
7     data = self.client_socket.recv(1024)
8     request = data.decode()
9     # 解析握手请求中的关键信息：Sec-WebSocket-Key，并保存下来备用
10    for line in request.split("\r\n"):
11        if line.startswith("Sec-WebSocket-Key:"):
12            self.key = line.split(":")[1].strip()
13            break
14    # 生成响应值key
15    response_key = base64.b64encode(hashlib.sha1((self.key + "258EAFA5-E914-4C86-B7DD-78A7F9849756").encode()).digest()).decode()
16    # 生成握手响应信息
17    response = "HTTP/1.1 101 Switching Protocols\r\n"
18    response += "Upgrade: websocket\r\n"
19    response += "Connection: Upgrade\r\n"
20    response += "Sec-WebSocket-Accept: " + response_key + "\r\n\r\n"
21    # 发送握手响应给客户端
22    self.client_socket.send(response.encode())
```

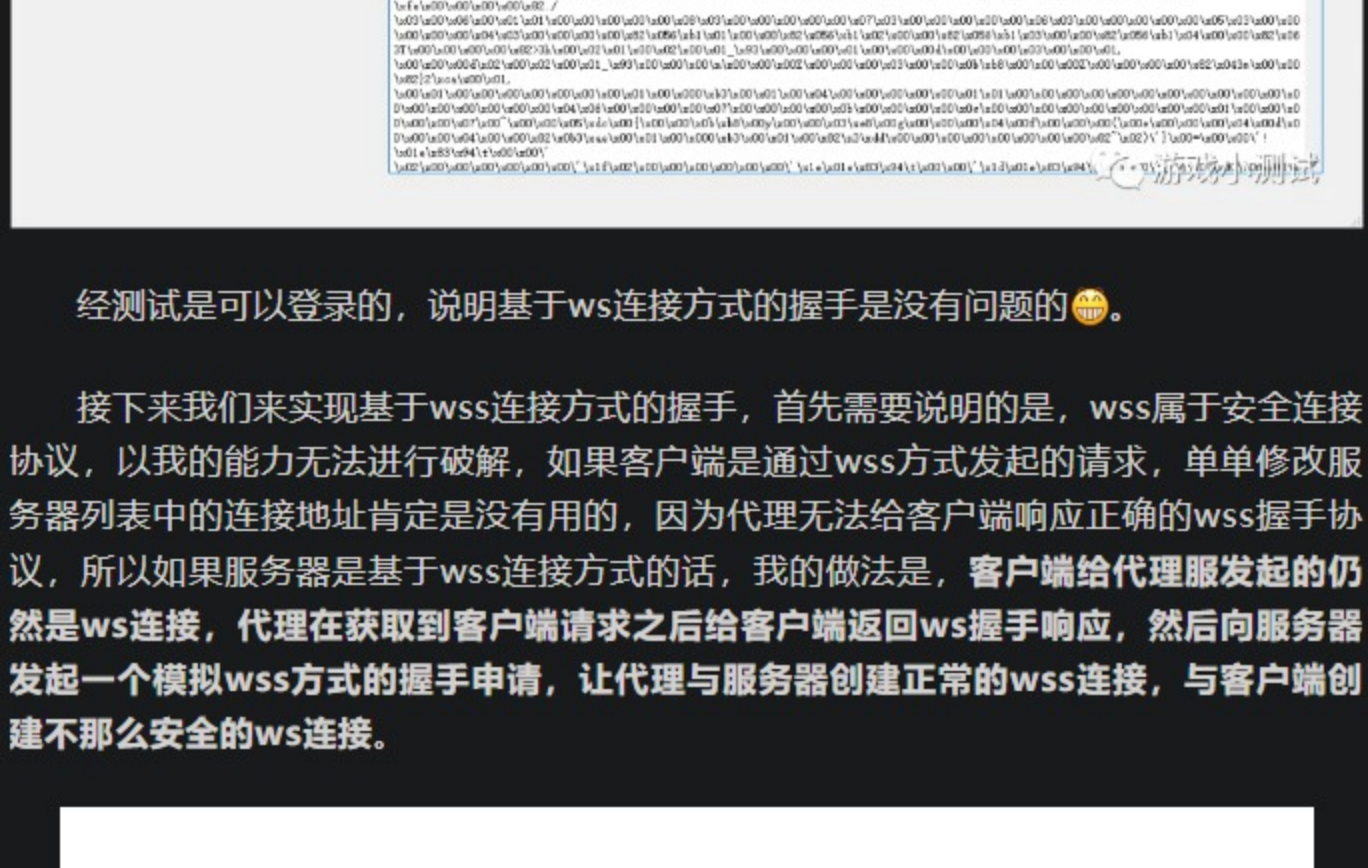
2.代理模拟客户端向服务器发起握手请求

```
1 def server_handshake_request(self, host, port):
2     """
3     向服务器发起握手请求
4     :param host:
5     :param port:
6     :return:
7     """
8     request = "GET / HTTP/1.1\r\n"
9     request += f"Host: {host}:{port}\r\n"
10    request += "Upgrade: websocket\r\n"
11    request += "Connection: Upgrade\r\n"
12    request += "Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits\r\n"
13    request += "Sec-WebSocket-Key: " + self.key + "\r\n"
14    request += "Sec-WebSocket-Version: 13\r\n\r\n"
15    # 发送握手请求给服务器
16    self.server_socket.send(request.encode())
17    # 接收服务器的握手响应
18    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端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~)：

```
1 def start(self):
2     """
3     代理启动方法
4     :return:
5     """
6     # 等待客户端连接
7     self.client_socket, addr = self.socket_service.accept()
8     # 代理模拟服务器，向客户端响应握手
9     self.client_handshake_response()
10    # 代理模拟客户端，向服务器发起握手请求
11    # 由于这里条件为服务器open和端口
12    # 因此这里我们使用__init__方法中，self和端口保存自己的属性以备调用
13    self.server_handshake_request(self.server_host, self.server_port)
14    # 启动两个线程，这里用到的是多线程，它是通过多线程调度的，所以会同时执行
15    timer(t, self.client_to_server).start()
16    timer(t, 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，如果想要实现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：

```
1 class Agent(object):
2     def __init__(self, agent_host, agent_port, server_host, server_port, ui):
3         """
4         代理初始化一个实例
5         :param agent_host: 代理绑定的ip
6         :param agent_port: 代理绑定的端口
7         :param server_host: 游戏服务器的ip
8         :param server_port: 游戏服务器的端口
9         """
10        self.ui_thread = ui_thread
11        # 设置一个socket server，以便接收客户端请求
12        self.socket_service = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13        # 设置socket server绑定的ip和端口号
14        self.socket_service.bind((agent_host, agent_port))
15        self.socket_service.listen(5)
16        # 设置客户端socket连接，先把它设置成一个空对象，后续再客户端请求过来之后再来设置
17        self.client_socket = None
18        self.server_host = server_host
19        self.server_port = server_port
20        # 设置服务器socket连接，生成一个socket对象
21        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22        # 设置SSL context
23        context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
24        context.check_hostname = False
25        context.verify_mode = ssl.CERT_NONE
26        # 设置SSL context 设置 server_socket 创建一个 SSL socket
27        self.server_socket = context.wrap_socket(self.server_socket,
28                                              server_hostname=server_host)
29        # 设置server_socket使用ssl连接
30        self.server_socket.connect((self.server_host, self.server_port))
31        # 给代理设置一个是否活跃的状态，用于检测到某处异常的时候终止异常操作
32        self.alive = True
```

接下来来搞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，跟我就不上了，盲点流整🙏。

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方式连接的还会出错的哦，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加密处理的，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服务器返回协议明文处理

```
1 def server_to_client(self):
2     """
3     处理服务器发送给客户端的包
4     1.获取服务器返回的字节流
5     2.获取原始的包内容
6     3.调用协议明文方法
7     4.将服务器返回的字节流通过client_socket转发给客户端
8     3.重复步骤
9     :return:
10    """
11    while self.alive:
12        # 接收字节流
13        head_data = self.server_socket.recv(2)
14        # 解析类型和长度
15        opcode = head_data[0] & 0x0f
16        payload_length = head_data[1] & 0x7f
17        # 如果payload_length是0，则表示payload_length为0
18        # 如果payload_length是127，则表示payload_length为127
19        # 如果payload_length是128，则表示payload_length为128
20        # 如果payload_length是129，则表示payload_length为129
21        # 如果payload_length是130，则表示payload_length为130
22        # 如果payload_length是131，则表示payload_length为131
23        # 如果payload_length是132，则表示payload_length为132
24        if opcode == 0x01:
25            length_data = self.server_socket.recv(2)
26            payload_length = struct.unpack(">H", length_data)[0]
27            elif payload_length == 127:
28                length_data = self.server_socket.recv(8)
29                payload_length = struct.unpack(">Q", length_data)[0]
30            elif payload_length < 126:
31                length_data = b''
32            else:
33                length_data = b''
34                # 正常来讲不会超过126，但这里加了个打印吧，防止出现解析错误的情况
35                print(f'出了点问题，长度信息无法解析！')
36        # 再根据payload_length，读取payload_length个字节流，这个字节流中包含的是原始的
37        proto_data = self.server_socket.recv(payload_length)
38        # 根据proto_data，生成一个字节流对象，并设置proto_data为key，proto_data为value，通过proto_data
39        proto_obj = struct.unpack('11', proto_data[4:11])[0]
40        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
41        proto_obj = struct.unpack('11', proto_data[4:11])[0]
42        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
43        proto_obj = struct.unpack('11', proto_data[4:11])[0]
44        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
45        proto_obj = struct.unpack('11', proto_data[4:11])[0]
46        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
47        proto_obj = struct.unpack('11', proto_data[4:11])[0]
48        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
49        self.ui_thread.recv_signal.emit(f'hacv {proto_id}: {dispos_obj}')
50        # 将字节流转发给客户端
51        self.client_socket.send(head_data + length_data + proto_data)
52    else:
53        # 如果使用的不是bytes类型，说明已经出错了，直接关闭吧
54        print(f'出了点问题，收到了非bytes类型的数据！')
55        self.alive = False
```

2.websocket客户端发送协议解密

```
1 def client_to_server(self):
2     """
3     转发客户端发送给服务器的包
4     1.获取发送的包头信息
5     2.获取发送包的长度信息
6     3.获取包内容
7     4.获取包内容的协议内容
8     5.协议解密
9     6.调用协议明文方法
10    7.将协议转发给服务器
11    8.重复步骤
12    :return:
13    """
14    while self.alive:
15        # 接收字节流
16        head_data = self.client_socket.recv(2)
17        # 解析类型和长度
18        opcode = head_data[0] & 0x0f
19        payload_length = head_data[1] & 0x7f
20        # 如果payload_length是0，则表示payload_length为0
21        # 如果payload_length是127，则表示payload_length为127
22        # 如果payload_length是128，则表示payload_length为128
23        # 如果payload_length是129，则表示payload_length为129
24        # 如果payload_length是130，则表示payload_length为130
25        # 如果payload_length是131，则表示payload_length为131
26        # 如果payload_length是132，则表示payload_length为132
27        if opcode == 0x01:
28            length_data = self.client_socket.recv(2)
29            payload_length = struct.unpack(">H", length_data)[0]
30            elif payload_length == 127:
31                length_data = self.client_socket.recv(8)
32                payload_length = struct.unpack(">Q", length_data)[0]
33            elif payload_length < 126:
34                length_data = b''
35            else:
36                length_data = b''
37                # 正常来讲不会超过126，但这里加了个打印吧，防止出现解析错误的情况
38                print(f'出了点问题，长度信息无法解析！')
39        # 再根据payload_length，读取payload_length个字节流，这个字节流中包含的是原始的
40        proto_data = self.client_socket.recv(payload_length)
41        # 根据proto_data，生成一个字节流对象，并设置proto_data为key，proto_data为value，通过proto_data
42        proto_obj = struct.unpack('11', proto_data[4:11])[0]
43        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
44        proto_obj = struct.unpack('11', proto_data[4:11])[0]
45        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
46        proto_obj = struct.unpack('11', proto_data[4:11])[0]
47        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
48        proto_obj = struct.unpack('11', proto_data[4:11])[0]
49        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
50        proto_obj = struct.unpack('11', proto_data[4:11])[0]
51        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
52        proto_obj = struct.unpack('11', proto_data[4:11])[0]
53        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
54        proto_obj = struct.unpack('11', proto_data[4:11])[0]
55        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
56        proto_obj = struct.unpack('11', proto_data[4:11])[0]
57        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
58        proto_obj = struct.unpack('11', proto_data[4:11])[0]
59        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
60        proto_obj = struct.unpack('11', proto_data[4:11])[0]
61        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
62        proto_obj = struct.unpack('11', proto_data[4:11])[0]
63        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
64        proto_obj = struct.unpack('11', proto_data[4:11])[0]
65        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
66        proto_obj = struct.unpack('11', proto_data[4:11])[0]
67        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
68        proto_obj = struct.unpack('11', proto_data[4:11])[0]
69        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
70        proto_obj = struct.unpack('11', proto_data[4:11])[0]
71        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
72        proto_obj = struct.unpack('11', proto_data[4:11])[0]
73        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
74        proto_obj = struct.unpack('11', proto_data[4:11])[0]
75        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
76        proto_obj = struct.unpack('11', proto_data[4:11])[0]
77        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
78        proto_obj = struct.unpack('11', proto_data[4:11])[0]
79        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
80        proto_obj = struct.unpack('11', proto_data[4:11])[0]
81        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
82        proto_obj = struct.unpack('11', proto_data[4:11])[0]
83        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
84        proto_obj = struct.unpack('11', proto_data[4:11])[0]
85        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
86        proto_obj = struct.unpack('11', proto_data[4:11])[0]
87        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
88        proto_obj = struct.unpack('11', proto_data[4:11])[0]
89        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
90        proto_obj = struct.unpack('11', proto_data[4:11])[0]
91        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
92        proto_obj = struct.unpack('11', proto_data[4:11])[0]
93        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
94        proto_obj = struct.unpack('11', proto_data[4:11])[0]
95        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
96        proto_obj = struct.unpack('11', proto_data[4:11])[0]
97        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
98        proto_obj = struct.unpack('11', proto_data[4:11])[0]
99        # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
100       proto_obj = struct.unpack('11', proto_data[4:11])[0]
101       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
102       proto_obj = struct.unpack('11', proto_data[4:11])[0]
103       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
104       proto_obj = struct.unpack('11', proto_data[4:11])[0]
105       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
106       proto_obj = struct.unpack('11', proto_data[4:11])[0]
107       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
108       proto_obj = struct.unpack('11', proto_data[4:11])[0]
109       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
110       proto_obj = struct.unpack('11', proto_data[4:11])[0]
111       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
112       proto_obj = struct.unpack('11', proto_data[4:11])[0]
113       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
114       proto_obj = struct.unpack('11', proto_data[4:11])[0]
115       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
116       proto_obj = struct.unpack('11', proto_data[4:11])[0]
117       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
118       proto_obj = struct.unpack('11', proto_data[4:11])[0]
119       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
120       proto_obj = struct.unpack('11', proto_data[4:11])[0]
121       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
122       proto_obj = struct.unpack('11', proto_data[4:11])[0]
123       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
124       proto_obj = struct.unpack('11', proto_data[4:11])[0]
125       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
126       proto_obj = struct.unpack('11', proto_data[4:11])[0]
127       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
128       proto_obj = struct.unpack('11', proto_data[4:11])[0]
129       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
130       proto_obj = struct.unpack('11', proto_data[4:11])[0]
131       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
132       proto_obj = struct.unpack('11', proto_data[4:11])[0]
133       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
134       proto_obj = struct.unpack('11', proto_data[4:11])[0]
135       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
136       proto_obj = struct.unpack('11', proto_data[4:11])[0]
137       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
138       proto_obj = struct.unpack('11', proto_data[4:11])[0]
139       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
140       proto_obj = struct.unpack('11', proto_data[4:11])[0]
141       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
142       proto_obj = struct.unpack('11', proto_data[4:11])[0]
143       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
144       proto_obj = struct.unpack('11', proto_data[4:11])[0]
145       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
146       proto_obj = struct.unpack('11', proto_data[4:11])[0]
147       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
148       proto_obj = struct.unpack('11', proto_data[4:11])[0]
149       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
150       proto_obj = struct.unpack('11', proto_data[4:11])[0]
151       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
152       proto_obj = struct.unpack('11', proto_data[4:11])[0]
153       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
154       proto_obj = struct.unpack('11', proto_data[4:11])[0]
155       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
156       proto_obj = struct.unpack('11', proto_data[4:11])[0]
157       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
158       proto_obj = struct.unpack('11', proto_data[4:11])[0]
159       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
160       proto_obj = struct.unpack('11', proto_data[4:11])[0]
161       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
162       proto_obj = struct.unpack('11', proto_data[4:11])[0]
163       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
164       proto_obj = struct.unpack('11', proto_data[4:11])[0]
165       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
166       proto_obj = struct.unpack('11', proto_data[4:11])[0]
167       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
168       proto_obj = struct.unpack('11', proto_data[4:11])[0]
169       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
170       proto_obj = struct.unpack('11', proto_data[4:11])[0]
171       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
172       proto_obj = struct.unpack('11', proto_data[4:11])[0]
173       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
174       proto_obj = struct.unpack('11', proto_data[4:11])[0]
175       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
176       proto_obj = struct.unpack('11', proto_data[4:11])[0]
177       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
178       proto_obj = struct.unpack('11', proto_data[4:11])[0]
179       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
180       proto_obj = struct.unpack('11', proto_data[4:11])[0]
181       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
182       proto_obj = struct.unpack('11', proto_data[4:11])[0]
183       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
184       proto_obj = struct.unpack('11', proto_data[4:11])[0]
185       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
186       proto_obj = struct.unpack('11', proto_data[4:11])[0]
187       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
188       proto_obj = struct.unpack('11', proto_data[4:11])[0]
189       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
190       proto_obj = struct.unpack('11', proto_data[4:11])[0]
191       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
192       proto_obj = struct.unpack('11', proto_data[4:11])[0]
193       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
194       proto_obj = struct.unpack('11', proto_data[4:11])[0]
195       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
196       proto_obj = struct.unpack('11', proto_data[4:11])[0]
197       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
198       proto_obj = struct.unpack('11', proto_data[4:11])[0]
199       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
200       proto_obj = struct.unpack('11', proto_data[4:11])[0]
201       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
202       proto_obj = struct.unpack('11', proto_data[4:11])[0]
203       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
204       proto_obj = struct.unpack('11', proto_data[4:11])[0]
205       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
206       proto_obj = struct.unpack('11', proto_data[4:11])[0]
207       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
208       proto_obj = struct.unpack('11', proto_data[4:11])[0]
209       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
210       proto_obj = struct.unpack('11', proto_data[4:11])[0]
211       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
212       proto_obj = struct.unpack('11', proto_data[4:11])[0]
213       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
214       proto_obj = struct.unpack('11', proto_data[4:11])[0]
215       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
216       proto_obj = struct.unpack('11', proto_data[4:11])[0]
217       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
218       proto_obj = struct.unpack('11', proto_data[4:11])[0]
219       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
220       proto_obj = struct.unpack('11', proto_data[4:11])[0]
221       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
222       proto_obj = struct.unpack('11', proto_data[4:11])[0]
223       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
224       proto_obj = struct.unpack('11', proto_data[4:11])[0]
225       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
226       proto_obj = struct.unpack('11', proto_data[4:11])[0]
227       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
228       proto_obj = struct.unpack('11', proto_data[4:11])[0]
229       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
230       proto_obj = struct.unpack('11', proto_data[4:11])[0]
231       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
232       proto_obj = struct.unpack('11', proto_data[4:11])[0]
233       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
234       proto_obj = struct.unpack('11', proto_data[4:11])[0]
235       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
236       proto_obj = struct.unpack('11', proto_data[4:11])[0]
237       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
238       proto_obj = struct.unpack('11', proto_data[4:11])[0]
239       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
240       proto_obj = struct.unpack('11', proto_data[4:11])[0]
241       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
242       proto_obj = struct.unpack('11', proto_data[4:11])[0]
243       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
244       proto_obj = struct.unpack('11', proto_data[4:11])[0]
245       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
246       proto_obj = struct.unpack('11', proto_data[4:11])[0]
247       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
248       proto_obj = struct.unpack('11', proto_data[4:11])[0]
249       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
250       proto_obj = struct.unpack('11', proto_data[4:11])[0]
251       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
252       proto_obj = struct.unpack('11', proto_data[4:11])[0]
253       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
254       proto_obj = struct.unpack('11', proto_data[4:11])[0]
255       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
256       proto_obj = struct.unpack('11', proto_data[4:11])[0]
257       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
258       proto_obj = struct.unpack('11', proto_data[4:11])[0]
259       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
260       proto_obj = struct.unpack('11', proto_data[4:11])[0]
261       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
262       proto_obj = struct.unpack('11', proto_data[4:11])[0]
263       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
264       proto_obj = struct.unpack('11', proto_data[4:11])[0]
265       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
266       proto_obj = struct.unpack('11', proto_data[4:11])[0]
267       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
268       proto_obj = struct.unpack('11', proto_data[4:11])[0]
269       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
270       proto_obj = struct.unpack('11', proto_data[4:11])[0]
271       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
272       proto_obj = struct.unpack('11', proto_data[4:11])[0]
273       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
274       proto_obj = struct.unpack('11', proto_data[4:11])[0]
275       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
276       proto_obj = struct.unpack('11', proto_data[4:11])[0]
277       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
278       proto_obj = struct.unpack('11', proto_data[4:11])[0]
279       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
280       proto_obj = struct.unpack('11', proto_data[4:11])[0]
281       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
282       proto_obj = struct.unpack('11', proto_data[4:11])[0]
283       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
284       proto_obj = struct.unpack('11', proto_data[4:11])[0]
285       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
286       proto_obj = struct.unpack('11', proto_data[4:11])[0]
287       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
288       proto_obj = struct.unpack('11', proto_data[4:11])[0]
289       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
290       proto_obj = struct.unpack('11', proto_data[4:11])[0]
291       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
292       proto_obj = struct.unpack('11', proto_data[4:11])[0]
293       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
294       proto_obj = struct.unpack('11', proto_data[4:11])[0]
295       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
296       proto_obj = struct.unpack('11', proto_data[4:11])[0]
297       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
298       proto_obj = struct.unpack('11', proto_data[4:11])[0]
299       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
300       proto_obj = struct.unpack('11', proto_data[4:11])[0]
301       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
302       proto_obj = struct.unpack('11', proto_data[4:11])[0]
303       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
304       proto_obj = struct.unpack('11', proto_data[4:11])[0]
305       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
306       proto_obj = struct.unpack('11', proto_data[4:11])[0]
307       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
308       proto_obj = struct.unpack('11', proto_data[4:11])[0]
309       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
310       proto_obj = struct.unpack('11', proto_data[4:11])[0]
311       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
312       proto_obj = struct.unpack('11', proto_data[4:11])[0]
313       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
314       proto_obj = struct.unpack('11', proto_data[4:11])[0]
315       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
316       proto_obj = struct.unpack('11', proto_data[4:11])[0]
317       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
318       proto_obj = struct.unpack('11', proto_data[4:11])[0]
319       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
320       proto_obj = struct.unpack('11', proto_data[4:11])[0]
321       # 根据proto_obj生成一个字典，字典的key为proto_obj，字典的value为proto_data
322       proto_obj = struct.unpack('11', proto_data[4:11])[0]
323      
```