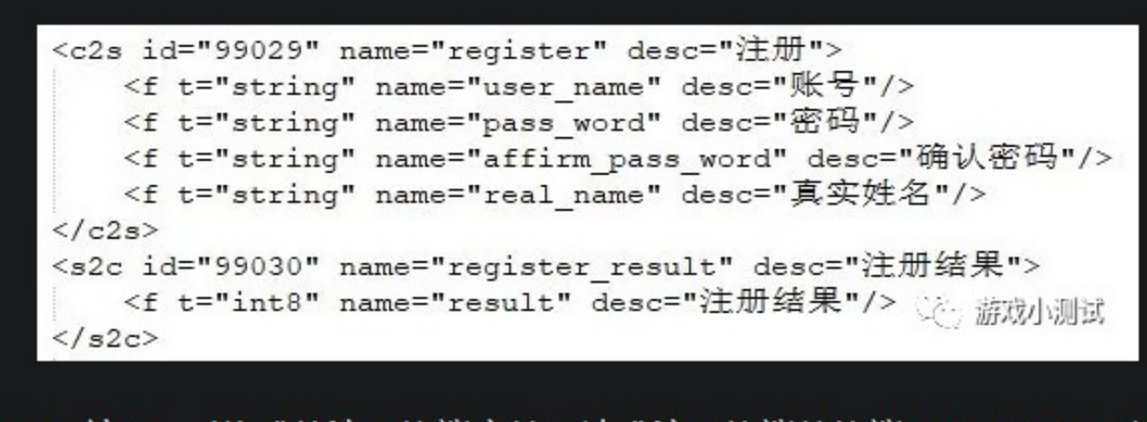
游戏抓包工具制作（三）—— 协议解析方法

原创 阿宽 游戏小测试 2024-04-08 19:07 广东

最近工作比较忙，所以近期没有做整理，今天正好空下来一点时间，写个后续~😳

之前我们讲了如何通过界面启动一个tcp的代理，并通过代理成功登录了游戏，将协议内容显示到了工具界面上。但是看到那一大堆bytes数据，大家可能都会比较懵逼，这是啥玩儿意😵？不管它认不认识我，反正我不认识它对不对？不要着急，今天我们就来尝试将这一大堆不知道是啥玩意的东西转化成我们想要的信息。

从程序角度来说，其实每一条协议都是一个对象，每个协议参数都是这个对象的属性，协议里的每个值都是这个对象参数的值，前后端交互的网络封包，只是对这些值进行了序列化和反序列化的操作，以我现在使用的自定义协议结构进行说明：



```
<c2s id="99029" name="register" desc="注册">
    <f t="string" name="user_name" desc="账号"/>
    <f t="string" name="pass_word" desc="密码"/>
    <f t="string" name="affirm_pass_word" desc="确认密码"/>
    <f t="string" name="real_name" desc="真实姓名"/>
</c2s>
<s2c id="99030" name="register_result" desc="注册结果">
    <f t="int8" name="result" desc="注册结果"/>
</s2c>
```

如图，这是xml格式的注册协议文档，请求注册协议的协议号是99029，参数一共有4个，分别是user_name， pass_word， affirm_pass_word， real_name，玩家在进行注册的时候，客户端会将玩家填入的这4个参数，连同协议号99029进行序列化，然后拼接在一起，发送给服务器，服务器在收到协议之后，首先会解析出这条协议的协议号99029，然后根据协议格式分别解析出user_name， pass_word，affirm_pass_word， real_name，调用注册方法，完成注册逻辑，然后将注册的结果返回给客户端。

在返回注册结果的时候，跟请求注册的逻辑是一致的，服务器会将99030协议id和result结果进行序列化，然后通过网络传输给客户端，客户端在收到网络封包之后，先解析出协议id 99030，然后再解析出注册结果，最终将注册结果展示给玩家。

我们来看一下在python中99030这个协议对象的实现方法：

```python
class S99030:
    # 构造函数
    def __init__(self, result=0):
        self.result = result  # (int8) 1_注册结果

    # 序列化
    def encode(self):
        result_bin = struct.pack("!b", self.result)
        buff = result_bin
        return 99030, len(buff), buff

    # 反序列化
    def decode(self, buff=None):
        i = 0
        # (int8) 1_注册结果
        self.result, = struct.unpack("!b", buff[i:i + 1])
        i += 1
        return i
```

现在再回到上面讲的内容，通过这个python对象重新进行说明：

首先看服务器的部分，伪代码如下：

```python
def return_register_result(server, result):
    """
    服务器响应注册结果
    :param server: 服务端
    :param result: 注册结果
    :return:
    """
    # 首先构造协议对象
    result_obj = S99030(result=result)
    # 对协议进行序列化，获得协议号，协议长度，和协议实际的字节流
    proto_id, proto_length, proto_buff = result_obj.encode()
    # 将协议长度，协议号进行序列化
    head_buff = struct.pack('!HI', proto_length, proto_id)
    # 将协议头和协议体进行拼接，组成一个完整的协议封包
    data = head_buff + proto_buff
    # 将这个协议封包通过与客户端创建的socket发送给客户端
    server.socket.send(data)
```

然后再看客户端接收到协议之后的处理逻辑，同样也是伪代码：

```python
def client_unpack(client):
    """
    客户端进行字节流解析
    :param client:
    :return:
    """
    # 首先读取2个字节，这两个字节代表的是协议长度
    length_buff = client.socket.recv(2)
    # 将长度的数值解析出来
    proto_length = struct.unpack('!H', length_buff)[0]
    # 再读取4个字节，这4个字节代表的是协议号
    id_buff = client.socket.recv(4)
    # 将协议号解析出来
    proto_id = struct.unpack('!I', id_buff)[0]
    # 然后再根据proto_length读取相应长度的字节
    proto_buff = client.socket.recv(proto_length)
    # 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，使用默
    obj = S99030()
    # 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
    obj.decode(proto_buff)
    # 然后再调用客户端的结果处理方法，对obj.result进行处理
    # 此处调用client的处理方法，省略
```

由于我们需要对协议进行抓取和明文显示，因此我们需要有一套跟客户端与服务器一样的处理逻辑，现在每个公司的游戏协议的实现方式不尽相同，所以还需要大家各自针对自己项目的协议方式写一套转化脚本，将协议文档转化成类似的协议类（有些协议如proto_buf有python的三方库可以直接调用和解析），然后在agent代理收到网络封包之后，调用这些协议类的decode()方法，就能获得含有正确参数值的协议对象了。

完成上面这一步，离明文化也就不远了，我们现在已经在脚本中获取到了协议对象，接下来只要将它转成类似json格式的字符串就可以了。

在实际游戏中，协议对象可能并不只有int值这种参数类型，还可能会有字符串，列表，甚至是对象类型的属性，我们写两个方法来对协议对象进行解析，由于python中的object类自带__dict__这个属性（对象属性的字典格式），所以我们可以走个捷径：

```python
def dispose_obj(obj):
    """
    解析对象，返回解析之后的属性和对应的值字典
    :param obj: 待解析的对象
    :return: 解析后的对象的属性和值的键值对
    """
    result = {}
    for key, value in obj.__dict__.items():
        if type(value) == int or type(value) == str:
            result.update({key: value})
        elif type(value) == list:
            result.update({key: dispose_list(value)})
        elif type(value) == bytes:
            result.update({key: value.decode()})
        else:
            result.update({key: dispose_obj(value)})
    return result
```

这是一个解析对象的方法，我们根据对象的字典属性，依此获取key（属性名）和value（属性值），如果是int类型和字符串类型，则直接放到一个字典中，如果是对象类型，则重复调用这个解析对象的方法，如果是list类型，则调用我们下面的解析列表的方法，方法如下：

```python
def dispose_list(pose_list):
    """
    解析列表类型的协议，返回解析后的结果字典
    :param pose_list: 待解析的列表，如果是int或者字符串类型，
                      则直接添加到result列表中，如果是list类型，则回调此方法，
                      如果是obj类型，则调用dispose_obj方法进行解析
    :return: 解析后的列表
    """
    result = []
    for i in pose_list:
        if type(i) == int or type(i) == str:
            result.append(i)
        elif type(i) == list:
            result.append(dispose_list(i))
        elif type(i) == bytes:
            result.append(i.decode())
        else:
            result.append(dispose_obj(i))
    return result
```

接下来我们写个对象测试一下：

```python
class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
        self.friends = []


p = Person('John', 37, '123 Main St')
p1 = Person('Kung', 38, '124 Main St')
p2 = Person('Jack', 36, '125 Main St')
p.friends.append(p1)
p.friends.append(p2)

print(dispose_obj(p))
```

运行结果：

```
{'name': 'John', 'age': 37, 'address': '123 Main St', 'friends': [{'name': 'Kung', 'age': 38, 'address': '124 Main St', 'friends': []}, {'name': 'Jack', 'age': 36, 'address': '125 Main St', 'friends': []}]}
```

貌似没啥问题，之后我们只要将协议对象放入这个方法，就可以把它明文化了。比如上面提到的S99030对象，解析后是这个样子：

```
{'result': 1}
```

接下来把思路拉回到我们上一篇文章，上一篇文章中，我们已经可以显示原始字节流的协议了，那么我们今天要做的，主要是根据上面的伪代码部分，对原始的字节流进行拆分，解析出协议号，生成协议对象，并调用decode()方法，将协议字节流解析成对应的数值赋给对象的属性，最后再把对象明文化显示。

PS：重复声明一下，每个游戏的协议结构可能不一样，我的例子不一定适合所有游戏，只是提供一个思路，具体的协议组成方式需要咨询一下前后端程序。

PS：重复声明一下，每个游戏的协议结构可能不一样，我的例子不一定适合所有游戏，只是提供一个思路，具体的协议组成方式需要咨询一下前后端程序。

PS：重复声明一下，每个游戏的协议结构可能不一样，我的例子不一定适合所有游戏，只是提供一个思路，具体的协议组成方式需要咨询一下前后端程序。

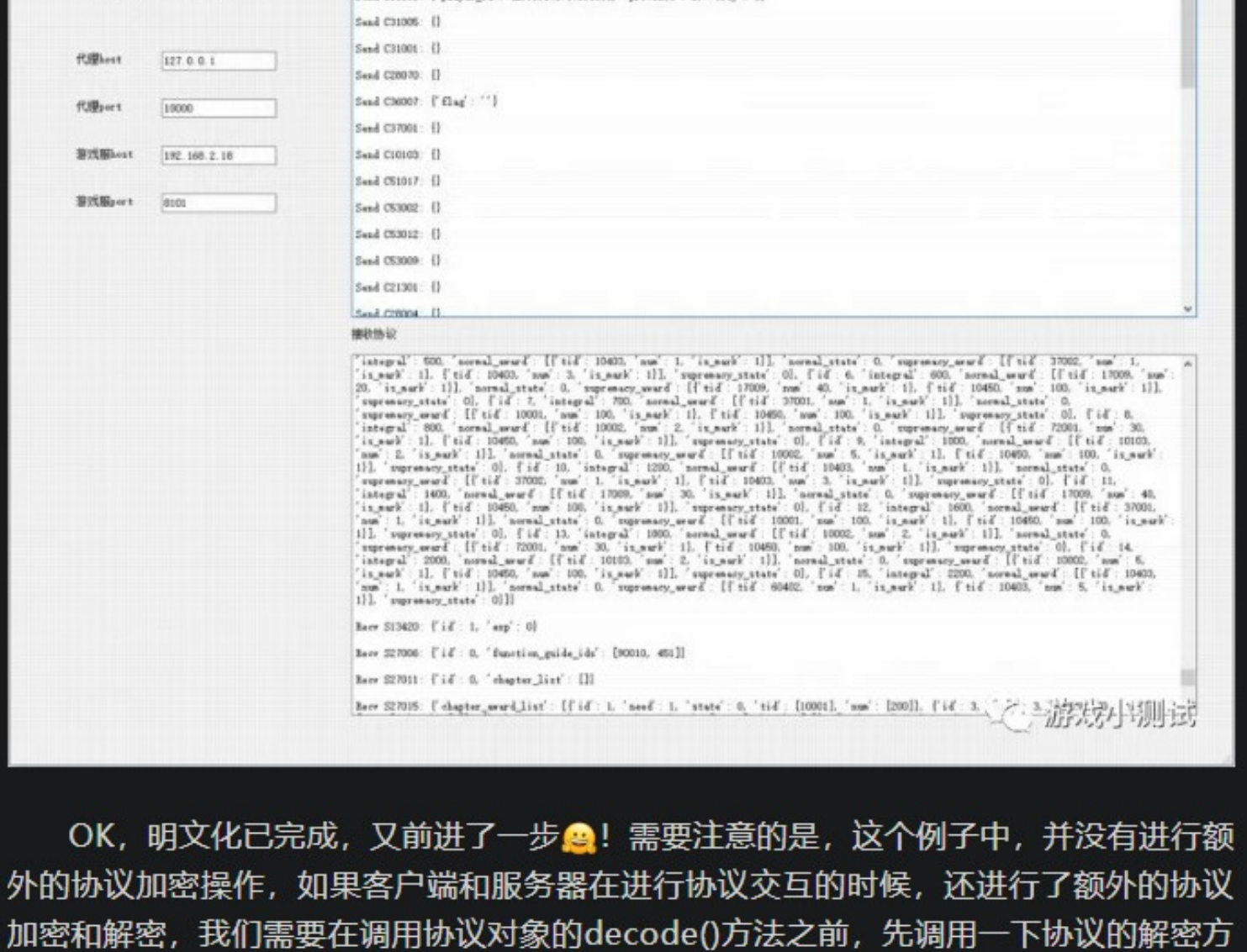
修改一下agent类中的client_to_server方法：

```python
def client_to_server(self):
    """
    转发客户端发送给服务器的包
    1.获取客户端发起的字节流
    2.通过server socket进行转发
    3.循环此步骤
    :return:
    """
    while self.alive:
        # 首先读取2个字节，这两个字节代表的是协议长度
        length_buff = self.client_socket.recv(2)
        # 将长度的数值解析出来
        proto_length = struct.unpack('!H', length_buff)[0]
        # 再读取4个字节，这4个字节代表的是协议号
        id_buff = self.client_socket.recv(4)
        # 将协议号解析出来
        proto_id = struct.unpack('!I', id_buff)[0]
        # 然后再根据proto_length读取相应长度的字节
        proto_buff = self.client_socket.recv(proto_length)
        # 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，使
        # 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（需要
        # 你也可以创造一个字典，用协议号作为key， 协议类作为value，通过协议号获取到
        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(proto_buff)
        # 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(prot
        # 将协议转发给服务器
        self.server_socket.send(length_buff + id_buff + proto_buff)
```

再修改一下server_to_client方法：

```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.调用client socket，将服务器返回的字节流直接转发给客户端
    3.循环此步骤
    :return:
    """
    while self.alive:
        # 首先读取2个字节，这两个字节代表的是协议长度
        length_buff = self.server_socket.recv(2)
        # 将长度的数值解析出来
        proto_length = struct.unpack('!H', length_buff)[0]
        # 再读取4个字节，这4个字节代表的是协议号
        id_buff = self.server_socket.recv(4)
        # 将协议号解析出来
        proto_id = struct.unpack('!I', id_buff)[0]
        # 然后再根据proto_length读取相应长度的字节
        proto_buff = self.server_socket.recv(proto_length)
        # 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，使
        # 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（需要
        # 你也可以创造一个字典，用协议号作为key， 协议类作为value，通过协议号获取到
        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_buff)
        # 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(prot
        # 将协议转发给客户端
        self.client_socket.send(length_buff + id_buff + proto_buff)
```

然后我们再运行一下之前的脚本，然后打开游戏通过代理进入服务器，观察工具上显示的内容：



OK，明文化已完成，又前进了一步🤗！需要注意的是，这个例子中，并没有进行额外的协议加密操作，如果客户端和服务器在进行协议交互的时候，还进行了额外的协议加密和解密，我们需要在调用协议对象的decode()方法之前，先调用一下协议的解密方法（这个方法需要找前后端程序请教，跟他们的方法一样就可以了），协议解密之后再调用协议对象的decode()方法。

下篇文章会插一个websocket协议的代理实现，再之后就开始实现伪造包的方法，还请小伙伴们持续关注😣，有问题的同学可以后台留言给我哦（因为平时要工作，我并不是很常看公众号的后台，如果没有回复，可能是我没有看到留言，也可以加我微信wxid_ns56wyaqeg7s22，我会在看到留言后第一时间回复）。

#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（二）—— 设计一个简单界面

下一篇 ›
游戏抓包工具制作（四）—— 其他连接方式的实现（WebSocket）

个人观点，仅供参考