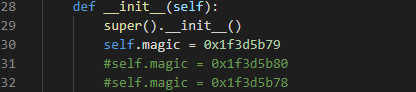
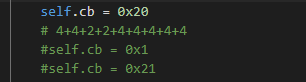
## DTXMessageHeader

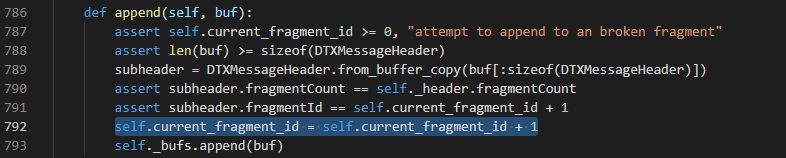
1. 修改dtxMessageHeader 的magic ， 该数值只能为**0x1f3d5b79**，当不为该数字时，**make\_channel**会失败；

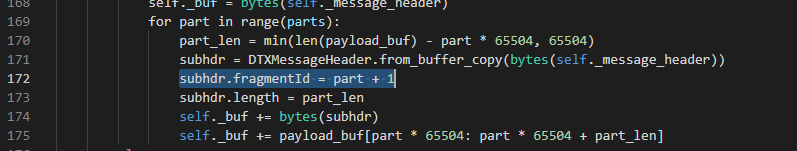


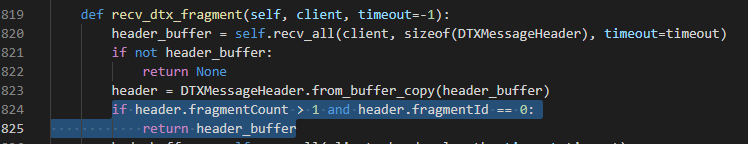
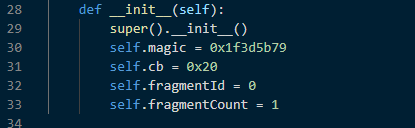
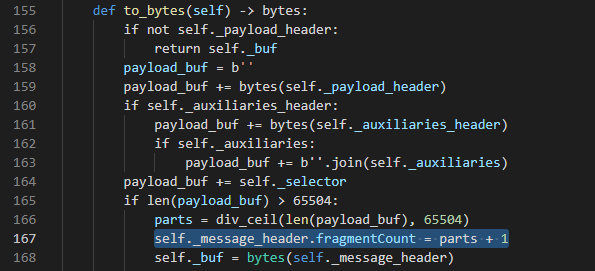
1. Cb 为dtxMessageHeader头部的长度，**0x20** 换为十进制为32 = 4+4+2+2+ 4+4+4+4, 改变头部的大小并不会产生实质影响；



1. FragmentId 与 fragmentCount共同影响message的分段，以及recv阶段message的“组装”；







1. Length字段，该长度为 = payloadHeader长度 + auxiliary部分长度( 可分为auxiliariesHeader+auxiliariesHeaderLen ) + selector部分长度

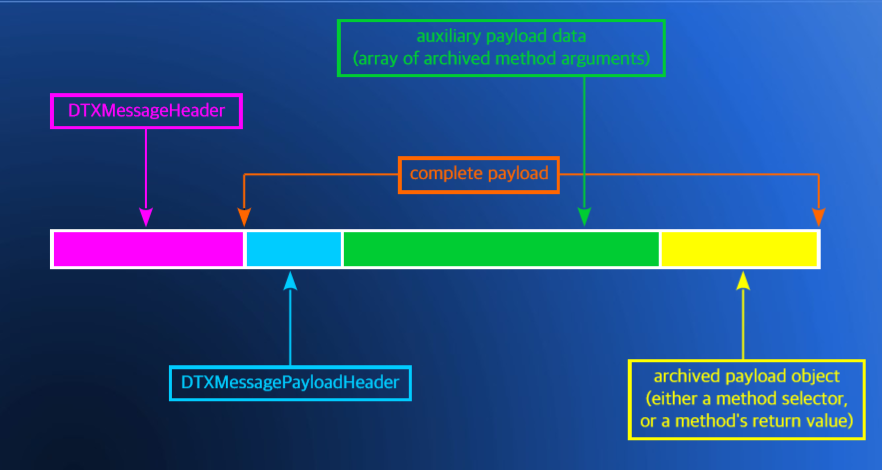




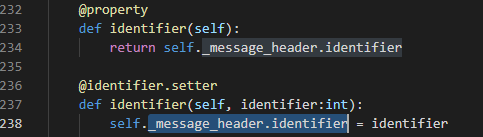






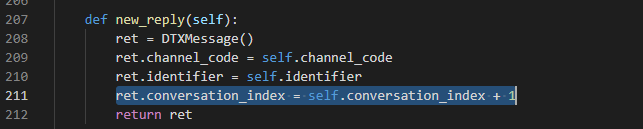


1. Identifier 字段，所在message信息的唯一标识符，用于区别其它dtxMessage信息，以避免混淆；

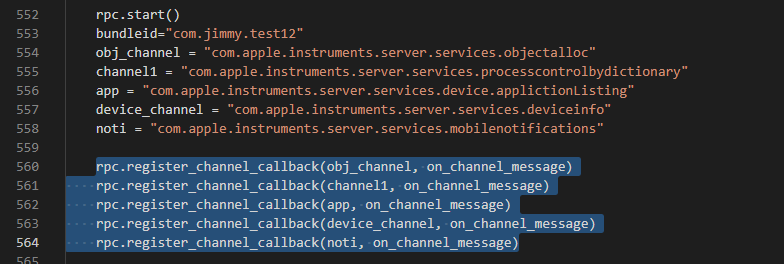




1. ConversationIndex字段，通过该段来验证当前的message是否是要求的message，



1. ChannelCode字段，表示所支持的服务，将服务与其对应的操作相绑定；

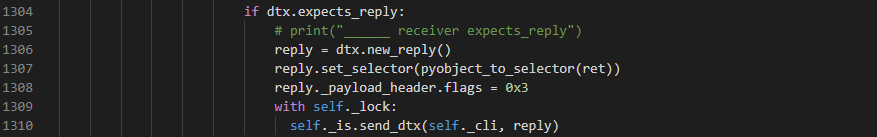








1. ExpectsReply字段，判断一个selector的请求是否要求返回信息





## DTXPayloadHeader

1. Flags 初始为0x2，其含义为有效载荷消息的类型， 修改之后则无法出现结果

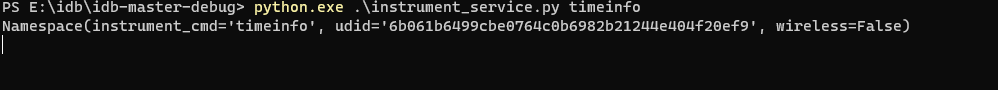
类似TCP 的flags：

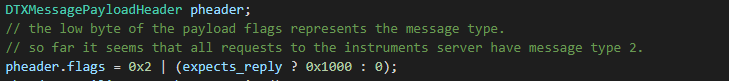
0x1 为紧急指针；

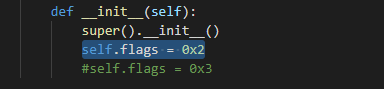
0x2 为主动发送；

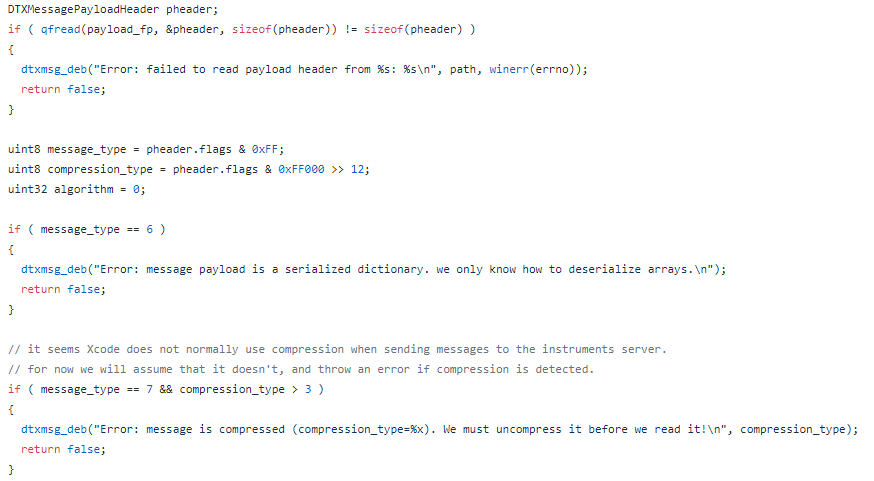
0x3 为消息确认；

……









flags & 0xFF == 6 代表payload为串行化的字典类型；

flags & 0xFF000 >> 12 代表着payload的压缩类型；

其中，message\_type == 7 && compression\_type > 3代表payload被压缩；

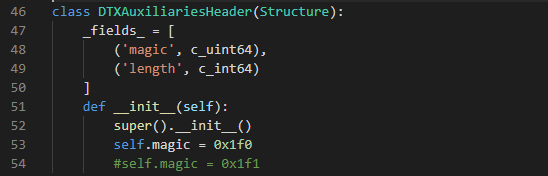
1. auxiliaryLength 字段，其含义为auxiliary 部分的长度，由于将auxiliary部分也分为头部和载荷两部分，因此其为：



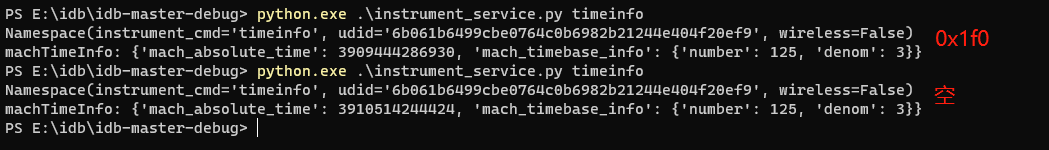
1. totalLength 字段，其含义为complete payload部分除去其payload头部之外的部分长度：

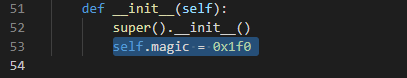


## DTXAuxiliariesHeader



1. Magic 字段，初始为0x1f0，有无该字段似乎对输出不影响；



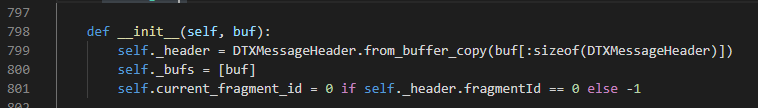


1. Length字段，其含义为auxiliary部分的长度；

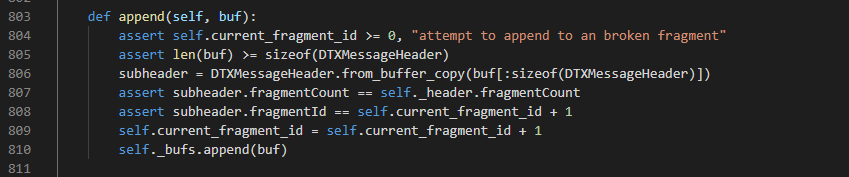
## 类作用

### DTXFragment(分段)

1. \_init\_(self, buf)，用来初始化“一组”dtxmsg信息的头部；



1. Append(self, buf)，将buf中的内容转换为字节数据添加到一组dtxmsg中，一组中由于分段而后来的dtxmsgHeader，其fragmentId为了保证顺序而随之变大；

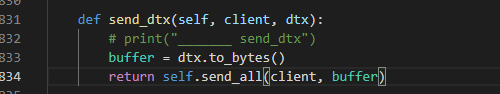


1. key是一个包含channelCode和identifier的元组；
2. completed用来标识当前信息组是否以及接收完毕；
3. message将字节流的数据转换为字符数据；

### DTXClientMix(用来发送\接收dtx信息)

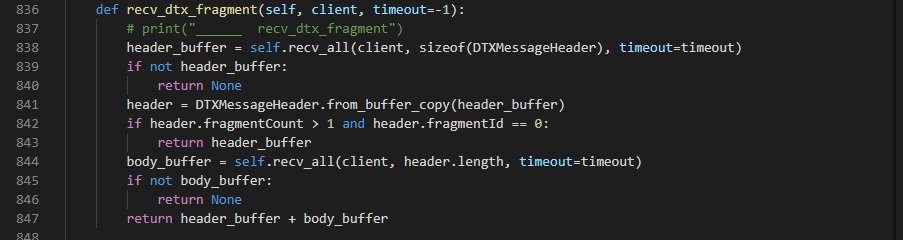
1. Send\_dtx(self, client, dtx)，用来向client发送dtx数据：

* 调用dtx.to\_bytes()，将已经生成的dtx信息转换为字节类型信息；
* 调用send\_all 向client发送字节类型信息；



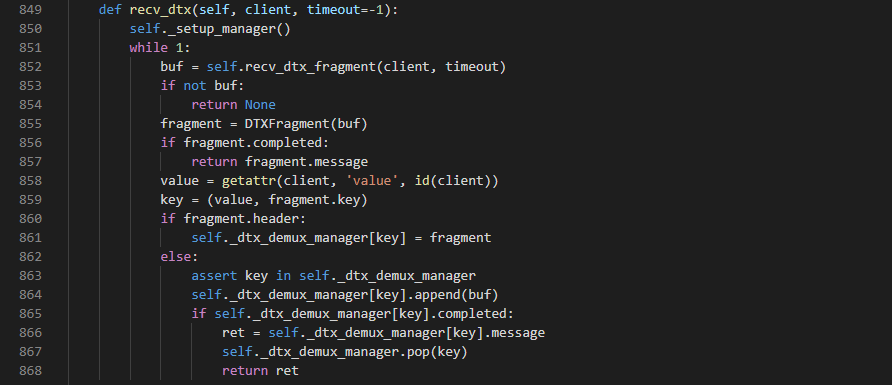
1. Recv\_dtx\_fragment：将recv\_all接收到的信息按DTXMessage的格式进行“分段”：

* 第一次调用recv\_all从client接收长度为DTXMessage头部长度的信息；
* fragmentId 为0说明是头部信息，fragmentCount > 1说明还有未接收的信息，直接返回header\_buffer是因为0th的dtxmsg步携带负荷；
* 第二次调用recv\_all 从client接收长度为header.length，也就是除去dtxmsg的负载部分长度，最后返回两部分的信息；



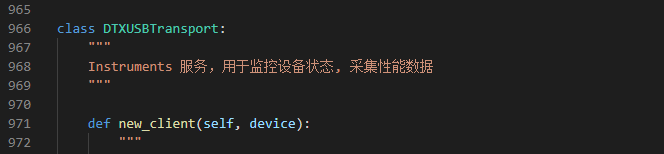
1. Recv\_dtx(self, client, timeout=-1)：

* 调用\_setup\_manager，个人理解这个函数的主要作用是：将来自同一个设备，同一channel，同一次调用的信息在一个字典\_dtx\_demux\_manager中进行分类“整合”，以防信息出现错乱；
* 通过从client接收的分段信息来“拼装”每个完整的信息片段；
* 对于每段信息，先接收存储到buf中，再判断该段是不是“结束段”，再通过\_dtx\_demux\_manager来对每段进行归类，拼接；
* 当一组的段完成拼接，将其组数据从字典中删除，并返回其数据；

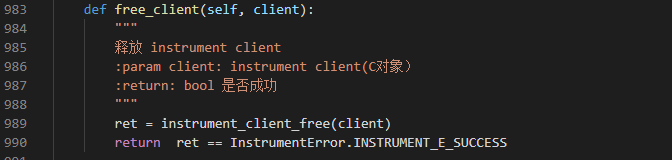


### DTXUSBTransport(用来建立socket连接，以及发送，接收原始信息)

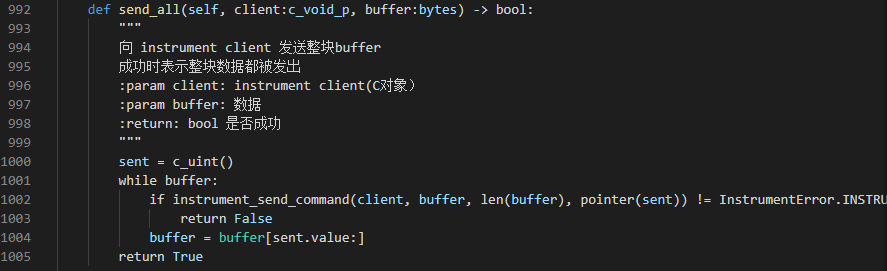
1. New\_client(self, device)， 通过创建socket，返回socket句柄；



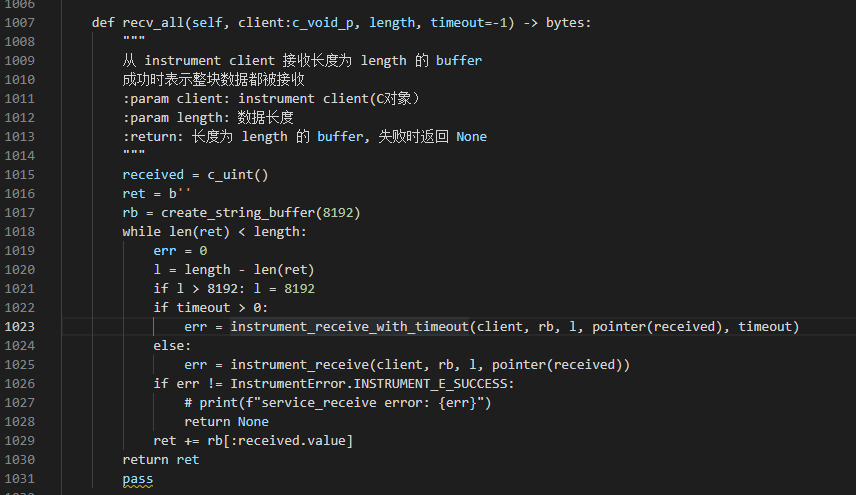
1. Free\_client(self, client)， 释放client套接字；



1. Send\_all(self, client:c\_void\_p, buffer:bytes)，将buffer中的所有内容一次性发给client；



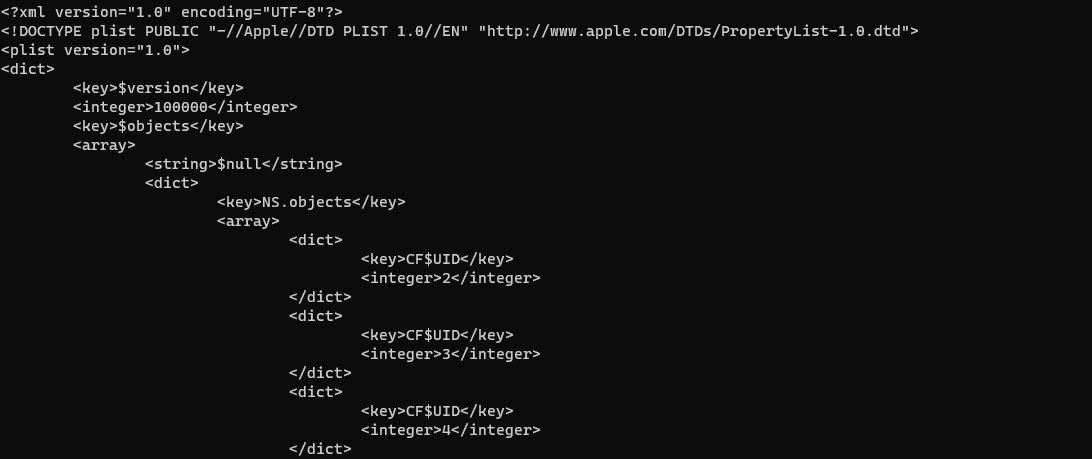
1. Recv\_all(self, client:c\_void\_p, length, timeout=-1)，从client端一次接收长度为length的数据，此处使用libimobiledevice中的函数进行接收和发送;



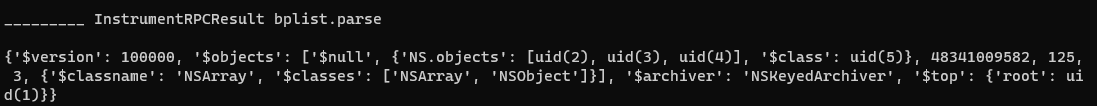
### InstrumentRPCResult(将二进制数据转换为xml\plist\parsed)

\_init\_(self, dtx)：self.raw 为dtx部分：

* Parse\_plist\_to\_xml 调用，将二进制类型的sel转换为xml格式；



* Bplist.parse 调用，将二进制类型的sel转换为plist格式；

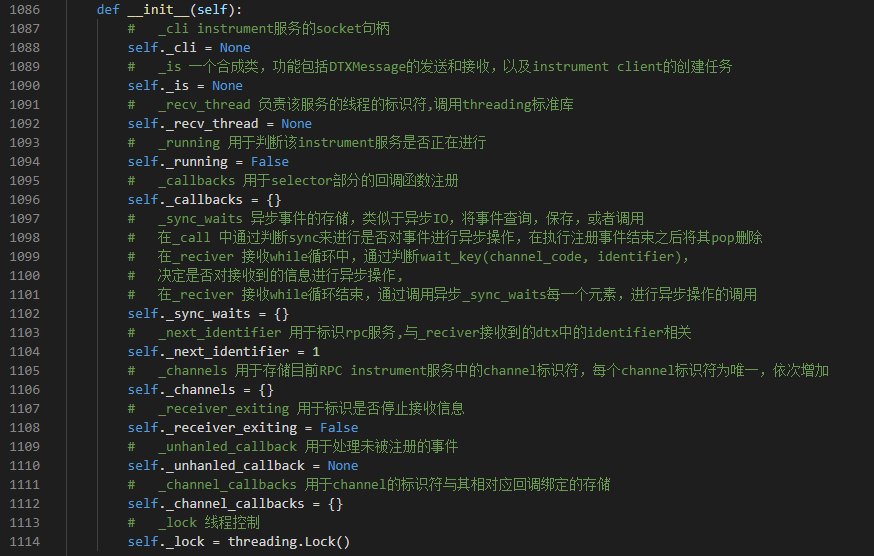


* Archiber.unarchive调用，将二进制类型的sel进行解包；

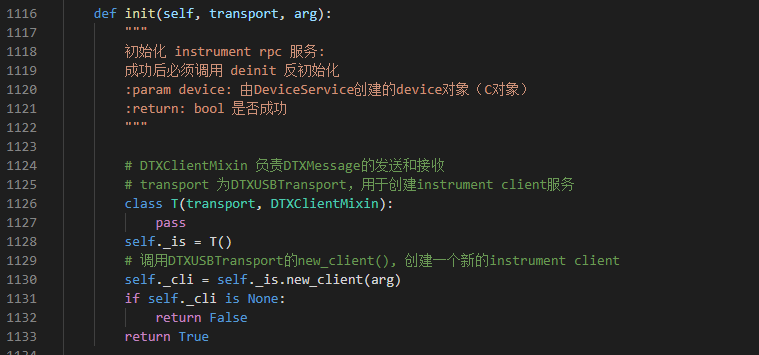


### InstrumentRPC(RPC instrument服务核心类)

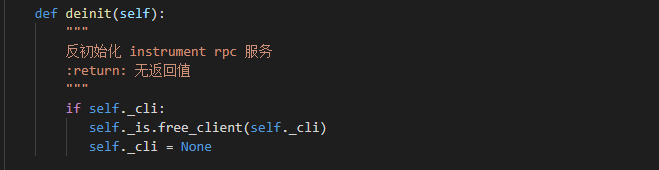
1. \_init\_ 类的初始化工作，其中类的私有数据有：



1. Init(self, transport, arg)，初始化instrument rpc服务:



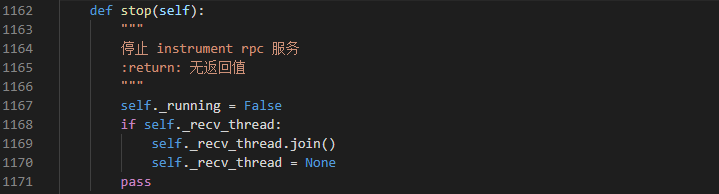
1. Deinit(self)，释放建立rpc服务的句柄，断开与rpc server的rpc连接；



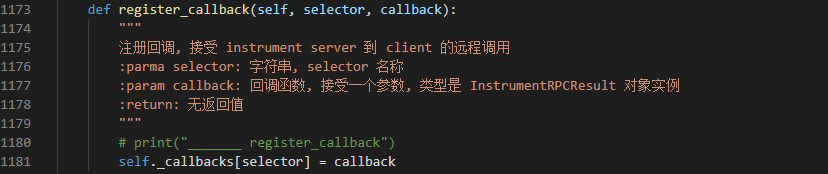
1. Start(self)，开始运行rpc服务：



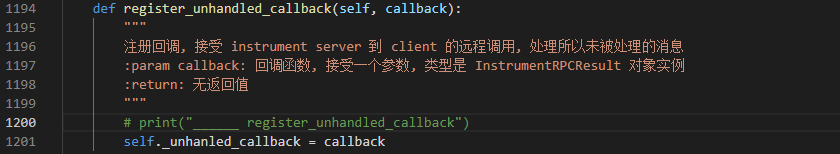
1. Stop(self)，停止执行instrument rpc服务，本质是停止recv\_thread接收线程的运行：



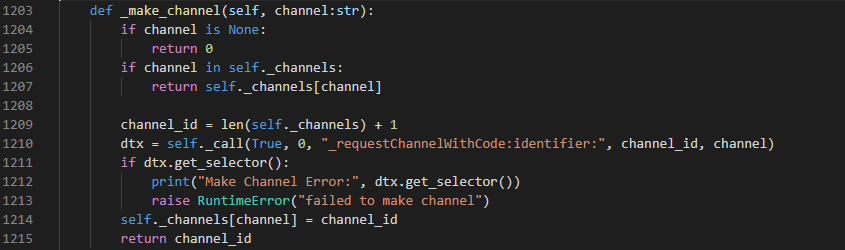
1. Register\_callback(self, selector, callback)，将selector与callback进行绑定，存储在rpc类中的\_callbacks字典中：



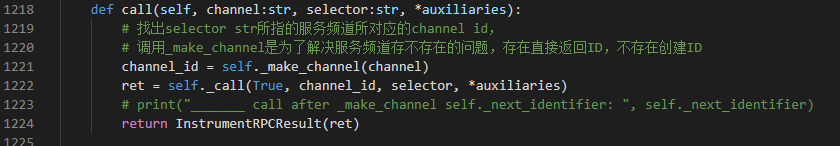
1. Register\_channel\_callback(self, channel, callback)，将channel与callbackjinx绑定，存储在rpc类中的\_channel\_callbacks字典中：



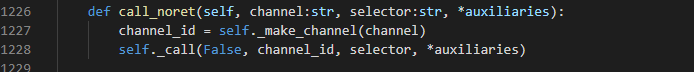
1. \_make\_channel(self, channel:str)，若channel在\_channels中存在将其返回，若channel不存在，调用\_call向instrument server申请channel将其保存在\_channels字典中；



1. Call(self, channel:str, selector:str, \*auxiliaries)，通过\_make\_channel取得channel所对应的channel\_id，再使用该id和selector调用\_call，向instrument server发送相应的请求；



1. Call\_noret(ret, channel:str, selector:str, \*auxiliaries)，作用与call一致，只是没有返回的信息；



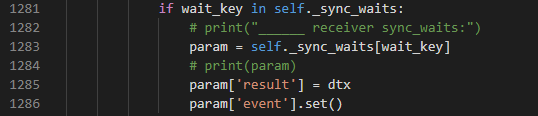
1. \_call(self, sync:bool, channel\_id:int, selector:str, \*auxiliaries)：组合dtxMessgae信息，通过self.is.send\_dtx(self.cli, dtx)向instrument server发送包装包好的dtx信息；

* 保证dtxMessage信息唯一性设置identifier；与此同时，设置channel\_id，将selector转变为二进制数据，
* 此处的sysn, wait\_key, param, dtx.expects\_reply, 以及\_sync\_waits全部都是为了等待发送之后instrument server的回复，至于\_call中返回的ret数据来源，则是该rpc线程绑定的\_receiver函数；

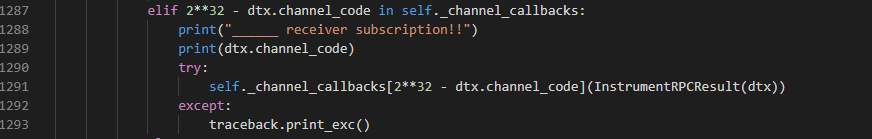


1. \_receiver(self)，从instrument server接收信息，并将接收到的信息设置到当前的rpc服务上：

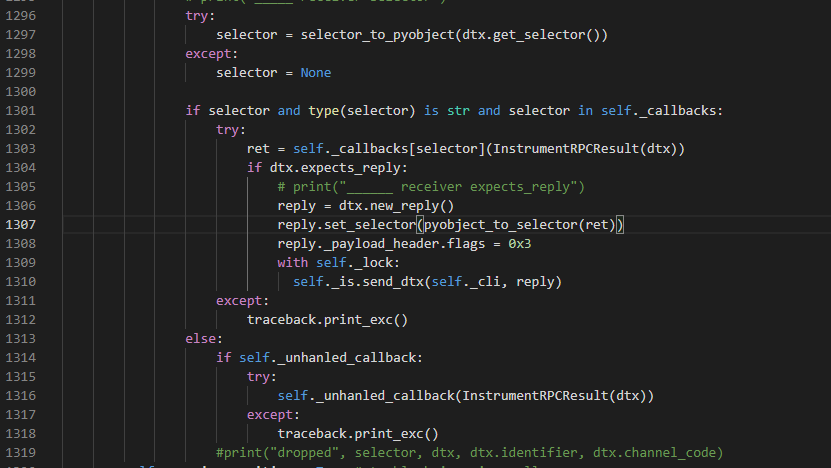
* 从instrument server接收dtx信息，根据接收到的每一个dtx的identifier重新设置当前的\_next\_identifier；
* 类型一：**异步回复，**若接收到的信息属于目前正在等待的wait\_key，则更新对正在等待的wait\_key的数据，并将其等待的接收方使用set()移出等待区;



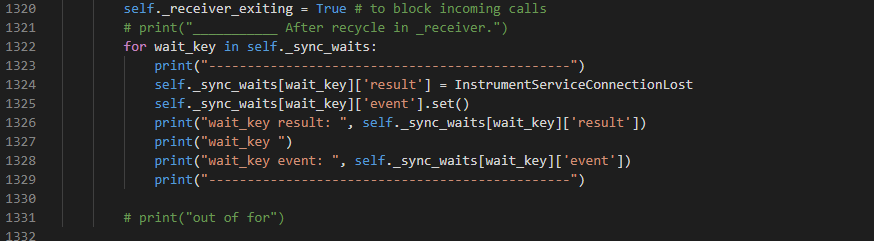
* 类型二：**channel的回调，**判断返回的channel\_code是否在\_channel\_callbacks中，\_call函数中发送的channel\_id与接收到的dtx.channel\_code其和为232，也就是说发送和接收到的channel\_code其和保持不变，通过这个特性查看接收到的channel\_code是否在\_channel\_callbacks中；



* 类型三：**selector的回调**，从接收到的dtx中取得selector部分，判断当前selector调用是否期望收到client的回复，若期望则建立一个新的dtxmessage，将selector回调函数的结果“打包”后发送给server； 若selector无效，则调用\_unhanlde\_callback事件的调用；

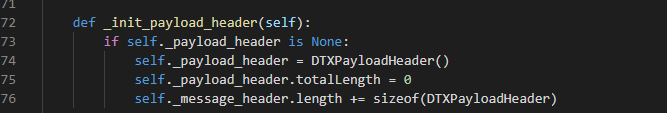


* 在所有信息接收完毕，若还有异步事件未接收到消息，则调用InstrumentServiceConnectionLost，进行丢失事件的垃圾处理；

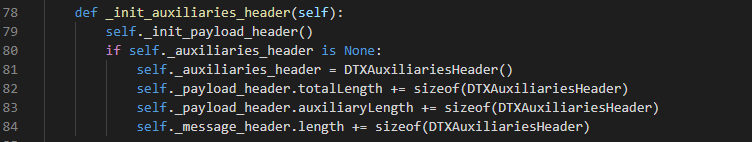


### DTXMessage(dtxmsg各部分的“组装”，“分拆”，以及“转换”)

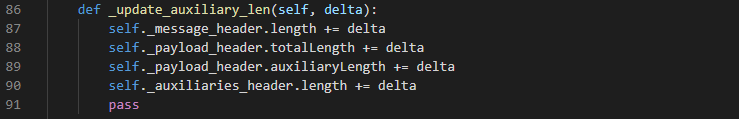
1. \_init\_ 定义所需的数据；
2. \_init\_payload\_header， 初始化payload部分的头部信息：



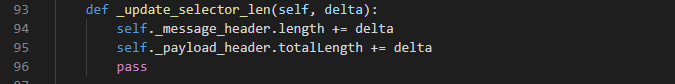
1. \_init\_auxiliaries\_header，初始化auxiliaries部分的头部信息：



1. \_update\_auxiliary\_len(self, delta)，更新auxiliary部分的长度信息，但实际上会更新整个DTXMessage各部分的长度信息：

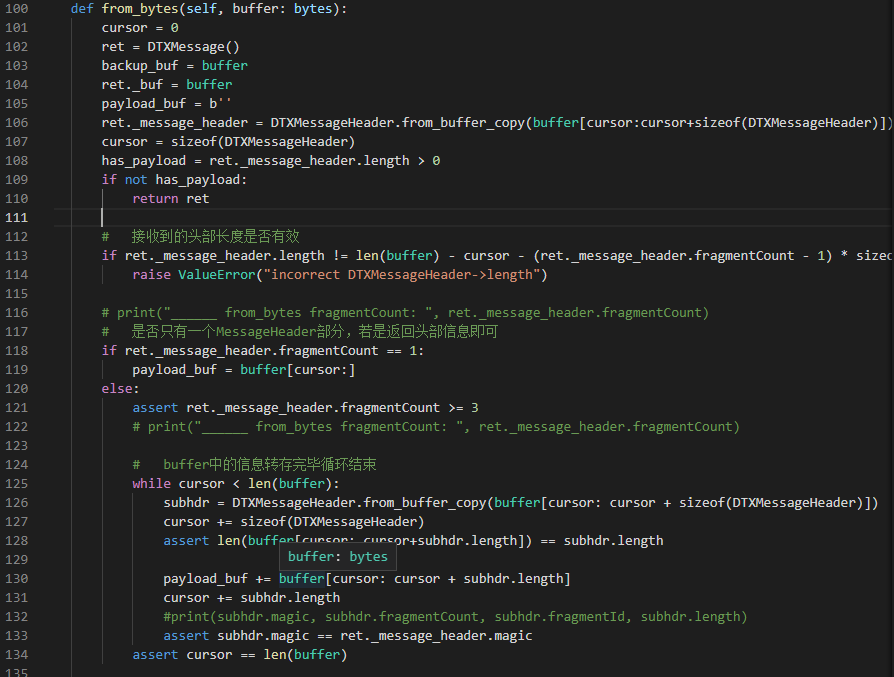


1. \_update\_selector\_len(self, delta)，更新selector部分的长度信息，但实际上会更新DTXMessage总长度以及payload部分的长度：

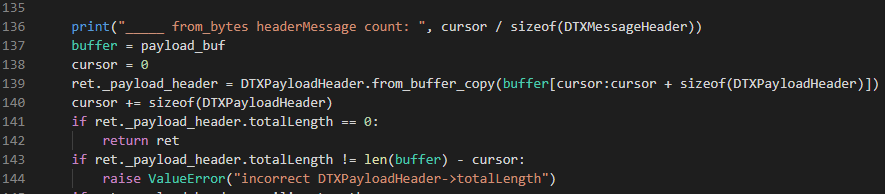


1. from\_bytes(self, buffer: bytes), 将二进制数据转变为字符数据：

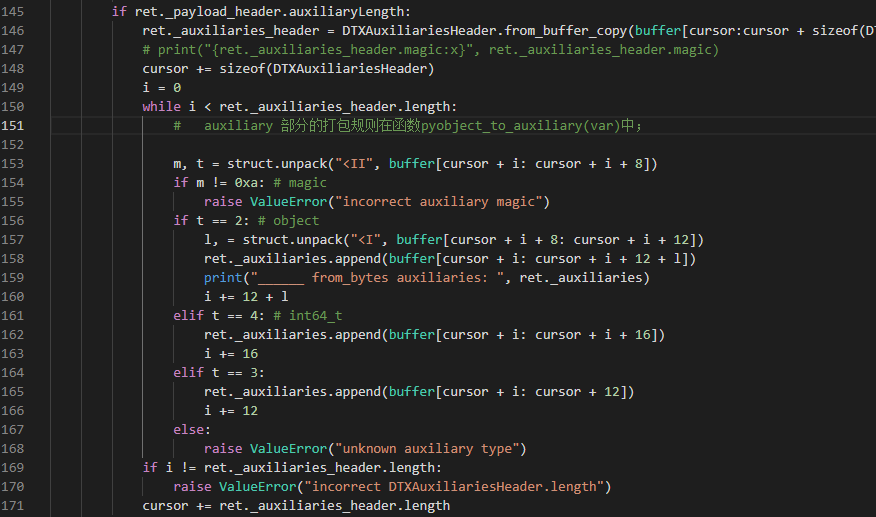
* 按DTXMessageHeader长度来从buffer中取头部信息，用头部信息来获取payload部分长度，cursor增加；



* 以DTXPayloadHeader长度来从payloadBuffer中取payload部分的头部长度，cursor增加；

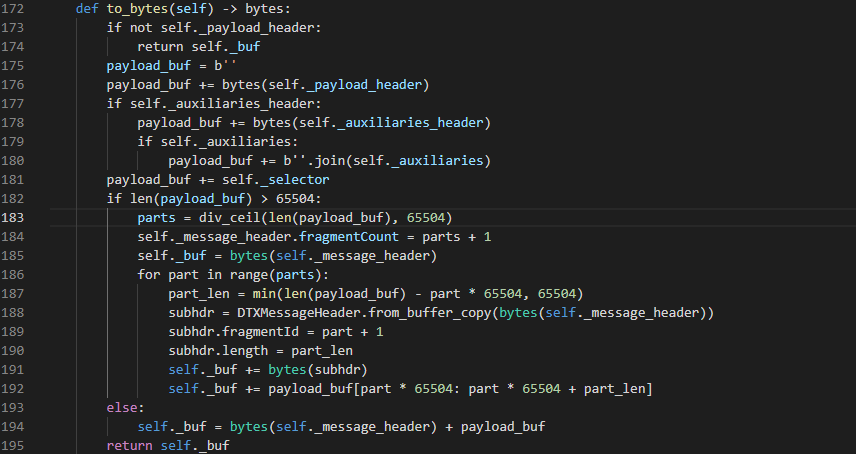


* 在payloadBuffer中对auxiliaries部分进行解包，调用struct.unpack；

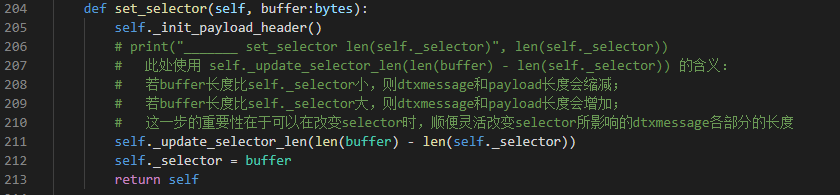


1. to\_bytes(self)，将dtxmessage存储的数据转变为二进制数据返回：

* 按照payloadHeader, auxiliariesHeader, auxiliaries, selector的顺序进行“组装”；
* 长度大于65504时，对装载的信息进行“切割”，通过div\_ceil判断切割成几段，根据段数增加detxmessage的fragmentCount；
* 通过min保证切割的每段长度都小于等于65504，对切割的每段dtxmessage进行初始化(**设置fragmentId, length，\_buf**)；
* 返回二进制数据；



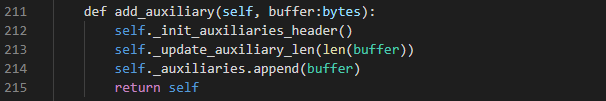
1. set\_selector(self, buffer:bytes)，更新\_selector的长度，信息：



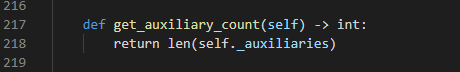
1. get\_selector(self)，返回\_selector信息：



1. add\_auxiliary(self, buffer:bytes)，将buffer中的部分添加到dtxmessage的auxiliary部分，更新数据，更新长度：



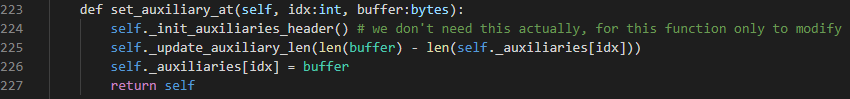
1. get\_auxiliary\_count(self)，返回auxiliary部分的长度：



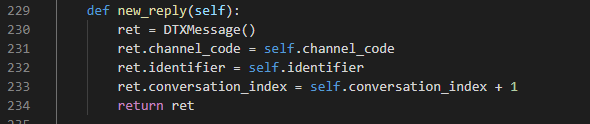
1. get\_auxiliary\_at(self, idx:int, buffer:bytes)，得到auxiliary的特定部分：



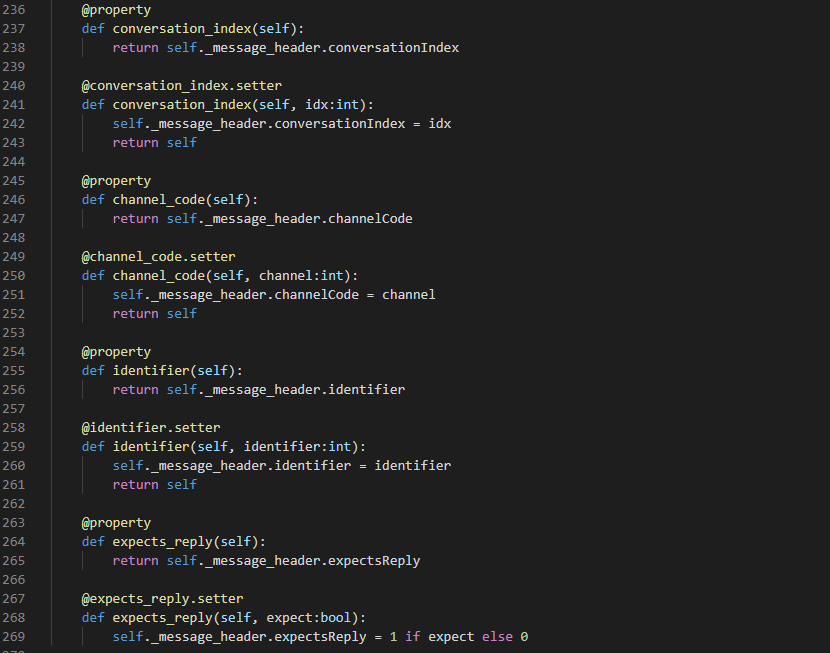
1. set\_auxiliary\_at(self, idx:int, buffer:bytes)，设置auxiliary的特定部分：



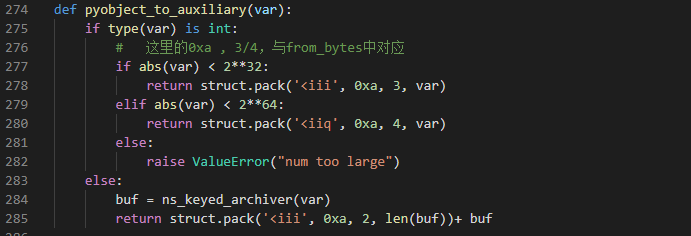
1. new\_reply(self)，一个新的dtxmessage，通过conversationIndex该段来验证当前的message是否是要求的message；



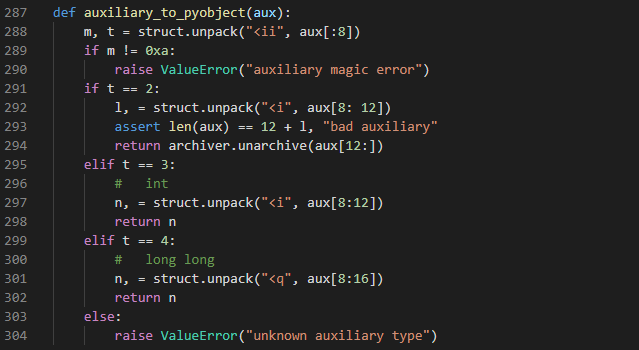
1. 方法：



1. pyobject\_to\_auxiliary(var)，将var中的数据进行打包：



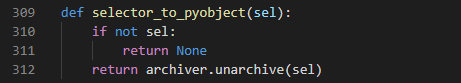
1. Auxiliary\_to\_pyobject(aux)，对aux进行解包：



1. pyobject\_to\_selector(s)，对s的selector进行打包：



1. Selector\_to\_pyobject(sel)，对sel的selector数据进行解包：



## 调用库

### ctypes(c语言中的类型)

### Struct(以及打包解包等)

### bpylist(Apple Binary Plist和NSKeyedArchiver格式的实现)

## instrument中各调用的流程

### argparse 命令行参数解析

### usb设备连接

1. get\_device(self)， 获取已经连接的设备列表

### Instrument RPC服务建立

1. get\_usb\_rpc(device)，在已连接的设备device上建立instrument rpc连接；
2. Instrument rpc建立成功，尝试调用各类方法；

### 方法使用

1. channels，cmd\_channels(rpc):

* 定义回调函数\_notifyOfPublishedCapabilities，让回调函数与方法\_notifyOfPublishnedCapabilities:，运行调用线程；
* 运行结束，停止线程，退出服务；

1. timeinfo，cmd\_timeinfo(rpc)：

* 运行线程，使用rpc.call向instrument server端发送时间请求；
* 输出结果；
* 运行结束，停止线程，退出服务；