**Async.py**

1. **class ModbusClientProtocol(protocol.Protocol, ModbusClientMixin):这代表了基本的modbus客户端协议。 所有的应用程序层代码推迟到更高级别的包装器**

**def \_\_init\_\_(self, framer=None, \*\*kwargs):初始化成帧器模块：param framer：用于协议的成帧器**

**def datagramReceived(self, data, params):获取响应，检查有效的消息，解码结果 ：param data：从服务器返回的数据：param params：主机参数发送数据报**

**def execute(self, request):开始生产者发送下一个请求consumer.write（帧（请求））**

**def \_handleResponse(self, reply):处理已处理的响应和链接以纠正延期：param reply：回复进程**

**def \_buildResponse(self, tid):帮助方法返回延迟响应 为当前请求。：param tid：此响应的事务标识符：returns：一个延迟链接到最新的请求**

1. **class ModbusClientFactory(protocol.ReconnectingClientFactory):简单客户端协议工厂**

**Common.py**

1. **class ModbusClientMixin(object):这是一个modbus客户端混合，提供额外的工厂所有当前modbus方法的方法。**

**def read\_coils(self, address, count=1, \*\*kwargs):：param address：要读取的起始地址：param count：要读取的线圈数：param unit：这个请求所针对的从属单元 ：returns：延迟响应句柄**

**def read\_discrete\_inputs(self, address, count=1, \*\*kwargs):：param address：要读取的起始地址：param count：要读取的线圈数：param unit：这个请求所针对的从属单元 ：returns：延迟响应句柄**

**def write\_coil(self, address, value, \*\*kwargs):：param address：写入的起始地址  ：param value：写入指定地址的值 ：param unit：这个请求所针对的从属单元 ：returns：延迟响应句柄**

**def write\_coils(self, address, values, \*\*kwargs)：param address：要写入的起始地址：param values：要写入指定地址的值：param unit：此请求定位的从属单元：returns：延迟响应句柄**

**def write\_register(self, address, value, \*\*kwargs):：param address：要写入的起始地址：param values：要写入指定地址的值：param unit：此请求定位的从属单元：returns：延迟响应句柄**

**def write\_registers(self, address, values, \*\*kwargs):param address：要写入的起始地址：param values：要写入指定地址的值：param unit：此请求定位的从属单元：returns：延迟响应句柄**

**def read\_holding\_registers(self, address, count=1, \*\*kwargs):param address：要写入的起始地址：param values：要写入指定地址的值：param unit：此请求定位的从属单元：returns：延迟响应句柄**

**def readwrite\_registers(self, \*args, \*\*kwargs):：param read\_address：开始读取的地址：param read\_count：从地址读取的寄存器数：param write\_address：开始写入的地址：param write\_registers：写入指定地址的寄存器：param unit：这个请求所针对的从属单元 ：returns：延迟响应句柄**

**def mask\_write\_register(self, \*args, \*\*kwargs):：param address：写入寄存器的地址 ：param and\_mask：应用于寄存器地址的位掩码 ：param or\_mask：应用于寄存器地址的位掩码：param unit：这个请求所针对的从属单元：returns：延迟响应句柄**

**Datastore/Sync.py**

1. **class BaseModbusClient(ModbusClientMixin):一个modbus同步客户端的接口。 这里定义的都是用于执行相关请求方法的方法。 派生类只需要实现运输方式并设定正确成帧器。**

**def \_\_init\_\_(self, framer, \*\*kwargs):初始化客户端实例**

**def connect(self):连接到modbus远程主机：returns：如果连接成功，则为True，否则为False**

**def close(self):关闭底层套接字连接**

**def \_send(self, request):在底层套接字上发送数据 ：param request：要发送的编码请求return：写入的字节数**

**def \_recv(self, size):从底层描述符读取数据  ：param size：要读取的字节数：return：读取的字节**

**def execute(self, request=None):：param request：请求处理 ：returns：请求执行的结果**

**def \_\_enter\_\_(self):用enter块实现客户端：returns：客户端的当前实例**

**def \_\_exit\_\_(self, klass, value, traceback):用退出块实现客户端**

**def \_\_str\_\_(self):构建连接的字符串表示形式：returns：字符串表示形式**

1. **class ModbusTcpClient(BaseModbusClient):实现一个modbus tcp客户端**

**def connect(self):实现一个modbus tcp客户端**

**def close(self):关闭底层套接字连接**

**def \_send(self, request):在底层套接字上发送数据**

**def \_recv(self, size):从底层描述符读取数据：param size：要读取的字节数：return：读取的字节**

**def \_\_str\_\_(self):构建connectio的字符串表示形式：returns：字符串表示形式**

1. **class ModbusUdpClient(BaseModbusClient):实现一个modbus udp客户端**

**def\_\_init\_\_(self,host='127.0.0.1',port=Defaults.Port,framer=ModbusSocketFramer, \*\*kwargs):初始化客户端实例**

**def \_get\_address\_family(cls, address):一个帮助方法来获取正确的地址族给定地址**

**def connect(self):连接到modbus tcp服务器**

**def close(self):关闭连接**

**def \_send(self, request):通过socket发送数据**

**def \_recv(self, size):从底层描述符读取数据**

**def \_\_str\_\_(self):构建连接的字符串表示形式**

1. **class ModbusSerialClient(BaseModbusClient):实现一个modbus串行客户端**

**def \_\_init\_\_(self, method='ascii', \*\*kwargs):初始化串行客户端实例**

**def \_\_implementation(method):返回请求的成帧器 ：方法：串行成帧器进行实例化 ：返回：所请求的串行成帧器**

**def connect(self):连接到modbus串行服务器**

**def close(self):关闭连接**

**def \_send(self, request):发送底层套接字上的数据 如果接收缓冲区仍然保存一些数据，然后进行冲洗 睡眠如果最后发送完成小于3.5个字符 以前。**

**def \_recv(self, size):从底层的描述符中读取数据 ：param size：要读取的字节数 ：return：读取的字节**

**def \_\_str\_\_(self):构建连接的字符串表示形式**

**Datastore/Context.py**

1. **class ModbusSlaveContext(IModbusSlaveContext):这将创建一个具有每个数据访问权限的模块数据模型 ,存储在自己的个人块中**

**def \_\_init\_\_(self, \*args, \*\*kwargs):初始化数据存储，默认为完全填充 如果没有在通过连续的数据块**

**def \_\_str\_\_(self):返回上下文的字符串表示形式**

**def reset(self):将所有数据存储重置为其默认值**

**def validate(self, fx, address, count=1):验证请求以确保它在范围内**

**def getValues(self, fx, address, count=1):验证请求以确保它在范围内**

**：param fx：我们正在使用的功能：参数地址：起始地址：param count：要检索的值的数量：returns：请求的值来自a：a + c**

**def setValues(self, fx, address, values):根据提供的值上传数据**

1. **class ModbusServerContext(object):这代表从属上下文的主集合。 如果将单个设置为true，它将被视为单个上下文，因此每个unit-id返回相同的上下文。 如果将单个设置为false，它将被解释为从属上下文的集合**

**def \_\_init\_\_(self, slaves=None, single=True):初始化modbus服务器上下文的新实例**

**def \_\_iter\_\_(self):迭代当前的从属上下文集合**

**def \_\_contains\_\_(self, slave):检查给定的从站是否在此列表中**

**def \_\_setitem\_\_(self, slave, context):用于设置一个新的从上下文**

**def \_\_delitem\_\_(self, slave):包装器用于访问从属上下文**

**def \_\_getitem\_\_(self, slave):用于访问从属上下文**

**Datastore/ Remote.py**

**1.class RemoteSlaveContext(IModbusSlaveContext):这将创建一个连接到的modbus数据模型 远程设备（取决于所使用的客户端）**

**def \_\_init\_\_(self, client):初始化数据存储**

**def reset(self):将所有数据存储重置为其默认值**

**def validate(self, fx, address, count=1):验证请求以确保它在范围内**

**def getValues(self, fx, address, count=1):验证请求以确保它在范围内 ：param fx：我们正在使用的功能 ：参数地址：起始地址 ：param count：要检索的值的数量 ：returns：请求的值来自a：a + c**

**def setValues(self, fx, address, values):使用提供的值设置数据存储**

**def \_\_str\_\_(self):返回上下文的字符串表示形式**

**def \_\_build\_mapping(self):一个快速帮助程序来构建函数代码映射器。**

**def \_\_extract\_result(self, fx, result):一个帮助方法来提取值**

**Datastore/ store.py**

**1.class BaseModbusDataBlock(object):modbus数据存储的基类**

**def default(self, count, value=False):用于将store初始化为一个值**

**def reset(self):将数据存储重置为初始化的默认值**

**def validate(self, address, count=1):检查请求是否在范围内**

**def getValues(self, address, count=1):从数据存储返回请求的值**

**def setValues(self, address, values):根据给定的值进行设置**

**def \_\_str\_\_(self):构建数据存储区的表示形式**

**def \_\_iter\_\_(self):迭代数据块数据**

**2.class ModbusSequentialDataBlock(BaseModbusDataBlock):创建一个连续的modbus数据存储**

**def \_\_init\_\_(self, address, values):初始化数据存储**

**def create(klass):使用工厂方法创建数据存储 完整地址空间初始化为0x00**

**def validate(self, address, count=1):检查请求是否在范围内**

**def getValues(self, address, count=1):返回数据存储的请求值**

**def setValues(self, address, values):设置数据存储的请求值**

**3.class ModbusSparseDataBlock(BaseModbusDataBlock):创建一个稀疏的modbus数据存储区**

**def \_\_init\_\_(self, values):初始化数据存储**

**def create(klass):使用工厂方法创建数据存储完整地址空间初始化为0x00**

**def validate(self, address, count=1):检查请求是否在范围内**

**def getValues(self, address, count=1):返回数据存储的请求值**

**def setValues(self, address, values):设置数据存储的请求值**

**Intrenal/ ptwisted.py**

**def InstallManagementConsole(namespace, users={'admin': 'admin'}, port=503):帮助程序启动modbus服务器的ssh管理控制台**

**def build\_protocol():创建**

**Server/async.py**

1. **class ModbusTcpProtocol(protocol.Protocol):实现了一个twisted的Modbus服务器**

**def connectionMade(self):客户端连接时的回调**

**def connectionLost(self, reason):客户端断开连接时的回调**

**def dataReceived(self, data):收到数据时的回调**

**def \_execute(self, request):执行请求并返回结果**

**def \_send(self, message):向网络发送请求（字符串）**

1. **class ModbusServerFactory(ServerFactory):一个modbus服务器的Builder类**

**def \_\_init\_\_(self, store, framer=None, identity=None, \*\*kwargs):modbus工厂的过载初始化程序**

1. **class ModbusUdpProtocol(protocol.DatagramProtocol): 实现一个modbus udp twisted服务器**

**def \_\_init\_\_(self, store, framer=None, identity=None, \*\*kwargs):modbus工厂的过载初始化程序**

**def datagramReceived(self, data, addr):接受到数据时的回调**

**def \_execute(self, request, addr):执行请求并返回结果**

**def \_send(self, message, addr):向网络发送请求（字符串）**

**def StartTcpServer(context, identity=None, address=None, console=False, \*\*kwargs):启动Modbus异步TCP服务器**

**def StartUdpServer(context, identity=None, address=None, \*\*kwargs):启动Modbus异步Udp服务器**

**defStartSerialServer(context,identity=None, framer=ModbusAsciiFramer, \*\*kwargs):启动Modbus异步串行服务器**

**Server/sync.py**

1. **class ModbusBaseRequestHandler(socketserver.BaseRequestHandler):实现modbus服务器协议**

**def setup(self):客户端连接时的回调**

**def finish(self):客户端断开连接时的回调**

**def execute(self, request):使用生成的消息调用回调**

**def handle(self):接受到数据时的回调**

**def send(self, message):向网络发送请求（字符串）**

**2.class ModbusSingleRequestHandler(ModbusBaseRequestHandler):实现modbus服务器协议**

**def handle(self):接受到数据时的回调**

**def send(self, message):向网络发送请求**

1. **class CustomSingleRequestHandler(ModbusSingleRequestHandler):实现modbus服务器协议**
2. **class ModbusConnectedRequestHandler(ModbusBaseRequestHandler):实现modbus服务器协议 这使用socketserver.BaseRequestHandler来实现 连接协议（TCP）的客户端处理程序**

**def handle(self):收到数据时的回调**

**def send(self, message):向网络发送请求（字符串）**

1. **classModbusDisconnectedRequestHandler(ModbusBaseRequestHandler）用于实现modbus服务器协议**

**def handle(self):接受到数据时的回调**

**def send(self, message):向网络发送请求（字符串）**

**6.class ModbusTcpServer(socketserver.ThreadingTCPServer): 一个modbus线程的tcp socket服务器**

**def \_\_init\_\_(self, context, framer=None, identity=None, address=None, handler=None, \*\*kwargs):socket服务器的重载初始化**

**def process\_request(self, request, client):回调用于连接新的客户端线程**

**def shutdown(self):停止serve\_forever循环**

**def server\_close(self): 关闭正在运行中的服务器的回调**

**7.class ModbusUdpServer(socketserver.ThreadingUDPServer):一个modbus线程的udp socket字服务器**

**def \_\_init\_\_(self, context, framer=None, identity=None, address=None, handler=None, \*\*kwargs):socket服务器的重载初始化**

**def process\_request(self, request, client):回调用于连接新的客户端线程**

**def server\_close(self):关闭正在运行中的服务器的回调**

1. **class ModbusSerialServer(object):一个modbus线程串口socket字服务器**

**def \_\_init\_\_(self, context, framer=None, identity=None, \*\*kwargs):socket服务器的重载初始化**

**def \_connect(self):连接到串行服务器**

**def \_build\_handler(self):一个辅助方法来创建和monkeypatch 一个串行处理程序**

**def serve\_forever(self):回调用于连接新的客户端线程**

**def server\_close(self):回调用于停止正在运行的服务器**

**def StartTcpServer(context=None, identity=None, address=None, \*\*kwargs):一个工厂启动并运行一个tcp modbus服务器**

**def StartUdpServer(context=None, identity=None, address=None, \*\*kwargs):一个工厂启动并运行一个udp modbus服务器**

**def StartSerialServer(context=None, identity=None, \*\*kwargs):启动和运行串行modbus服务器的工厂**

**Bit\_read\_message.py**

**1.class ReadBitsRequestBase(ModbusRequest):消息的基类**

**def \_\_init\_\_(self, address, count, \*\*kwargs):初始化读请求数据**

**def encode(self):编码请求pdu**

**def decode(self, data):解码请求的pdu**

**def get\_response\_pdu\_size(self):Func\_code（1字节）+字节计数（1字节）+线圈数量（n字节）/ 8，如果余数不同，则N = N + 1**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**class ReadBitsResponseBase(ModbusResponse):响应位读取值的消息的基类**

**def \_\_init\_\_(self, values, \*\*kwargs):初始化一个新的实例**

**def encode(self):编码响应的pdu**

**def decode(self, data):解码响应pdu**

**def setBit(self, address, value=1):Helper功能设置指定位**

**def resetBit(self, address):将指定的位设置为0**

**def getBit(self, address):获取指定位的值**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**2.class ReadCoilsRequest(ReadBitsRequestBase):该功能码用于从1到2000（0x7d0）连续状态读取的线圈在远程设备中。 请求PDU指定启动地址，即指定的第一个线圈的地址，以及的数量 线圈。 在PDU中，从零开始寻址线圈。 因此线圈**

**编号1-16的地址为0-15。**

**def \_\_init\_\_(self, address=None, count=None, \*\*kwargs):初始化一个新的实例**

**def execute(self, context):对数据存储区执行读取线程请求**

**3.class ReadCoilsResponse(ReadBitsResponseBase):响应消息中的线圈被打包为每位一个线圈数据字段。 状态显示为1 = ON，0 = OFF。 的LSB第一个数据字节包含查询中寻址的输出。 另一个线圈跟随该字节的高阶结尾，并从低阶到后续字节中的高位。**

**def \_\_init\_\_(self, values=None, \*\*kwargs):初始化一个新的实例**

1. **class ReadDiscreteInputsRequest(ReadBitsRequestBase):该功能码用于从1到2000（0x7d0）连续状态读取 的远程设备中的离散输入。 请求PDU指定 起始地址，即指定的第一个输入的地址，和 输入数量 在PDU中离散输入从...开始寻址 零。 因此，编号为1-16的离散输入将被寻址为0-15。**

**def \_\_init\_\_(self, address=None, count=None, \*\*kwargs):初始化一个新的实例**

**def execute(self, context):针对数据存储区运行读取离散输入请求**

**5.class ReadDiscreteInputsResponse(ReadBitsResponseBase):响应消息中的离散输入被打包为一个输入 位的数据字段。 状态显示为1 = ON; 0 = OFF。 的LSB 第一个数据字节包含查询中寻址的输入。 另一个 输入跟随该字节的高阶结尾，从低阶开始 到后续字节中的高位。**

**def \_\_init\_\_(self, values=None, \*\*kwargs):初始化一个新的实例**

**Bit\_write\_message.py**

**1.class WriteSingleCoilRequest(ModbusRequest):该功能码用于将单个输出写入ON或OFF 在远程设备中。**

**def \_\_init\_\_(self, address=None, value=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对写入线程请求进行编码**

**def decode(self, data):解码写入线程请求**

**def execute(self, context):对数据存储区执行写入线程请求**

**def get\_response\_pdu\_size(self):Func\_code（1字节）+输出地址（2字节）+输出值（2字节） ：返回**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**2.class WriteSingleCoilResponse(ModbusResponse):正常响应是请求的回应，在线圈之后返回状态已经写完了**

**def \_\_init\_\_(self, address=None, value=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):写入线圈响应**

**def decode(self, data):解码写入线圈响应**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**3.class WriteMultipleCoilsRequest(ModbusRequest):该功能代码用于强制每个线圈以一系列线圈 在远程设备中打开或关闭。**

**def encode(self):对写入线圈请求进行编码**

**def decode(self, data): 解码写入线圈的请求**

**def execute(self, context): 对数据存储区运行写入线程请求**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**4.class WriteMultipleCoilsResponse(ModbusResponse): 正常响应返回功能码，起始地址和 线圈数量被迫。**

**def \_\_init\_\_(self, address=None, count=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):写入线圈响应**

**def decode(self, data):解码写入线圈响应**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**Constants.py**

1. **class Defaults(Singleton):一个modbus默认值的集合**
2. **class ModbusStatus(Singleton):这些代表了modbus中的各种状态代码 协议**
3. **class Endian(Singleton):表示各种字节字节顺序的枚举**
4. **class ModbusPlusOperation(Singleton):表示modbus加请求的类型**
5. **class DeviceInformation(Singleton):表示要读取哪种类型的设备信息**
6. **class MoreData(Singleton):代表更多的条件**

**Device.py**

**1.class ModbusAccessControl(Singleton):这是一个简单的实现网络管理系统表。 它的目的是控制访问服务器（如果使用的话）。 我们假设，如果一个条目是在表中，则允许访问资源。 但是，如果主机没有出现在表中（全部） 未知主机），其连接将简单地关闭。**

**def \_\_iter\_\_(self):迭代网络访问表**

**def \_\_contains\_\_(self, host):检查主机是否允许访问资源**

**def add(self, host):从NMS表中添加允许的主机**

**def remove(self, host):从NMS表中删除允许的主机**

**def check(self, host):检查主机是否允许访问资源**

1. **class ModbusPlusStatistics(object):这是用来维持当前的modbus加统计数。 作为 现在这只是一个完成modbus实现的存根。 有关更多信息，请参阅modbus实现指南第87页**

**def \_\_init\_\_(self):使用默认值初始化modbus plus统计信息**

**def \_\_iter\_\_(self): 迭代统计**

**def reset(self):这将清除所有的modbus和统计信息**

**def summary(self):返回modbus加上统计信息的摘要**

**def encode(self):返回modbus加上统计信息的摘要**

1. **class ModbusDeviceIdentification(object):这用于提供设备标识 用于readDeviceIdentification函数**

**def \_\_init\_\_(self, info=None):初始化你需要的元素的数据存储。**

**def \_\_iter\_\_(self):迭代设备信息**

**def summary(self):返回主项目的概要**

**def update(self, value):更新这个身份的值使用另一个标识作为值**

**def \_\_setitem\_\_(self, key, value):包装器用于访问设备信息**

**def \_\_getitem\_\_(self, key):包装器用于访问设备信息**

**def \_\_str\_\_(self):建立该设备的一个表示**

**4.class DeviceInformationFactory(Singleton):这是一个帮助工厂，真的只是隐藏处理设备信息请求的一些复杂性（功能代码0x2b 0x0e）。**

**def get(cls, control, read\_code=DeviceInformation.Basic, object\_id=0x00):从系统获取请求的设备数据**

**def \_\_get(cls, identity, object\_id):从设备信息中读取一个object\_id**

**def \_\_gets(cls, identity, object\_ids):从设备信息中读取多个object\_id**

**5.class ModbusCountersHandler(object):这是一个帮助类，用于简化计数器的属性**

**def \_\_iter\_\_(self):迭代设备计数器**

**def update(self, values):更新这个身份的值使用另一个标识作为值**

**def reset(self):这将清除所有系统计数器**

**def summary(self):返回计数器当前状态的摘要**

**6.class ModbusControlBlock(Singleton):这是一个控制所有系统信息的全局单工**

**def \_\_str\_\_(self):构建控件块的表示**

**def \_\_iter\_\_(self):迭代设备计数器**

**def addEvent(self, event):向事件日志添加新事件**

**def getEvents(self):返回事件日志的编码集合。**

**def clearEvents(self):清除当前事件列表**

**def reset(self):这将清除所有的系统计数器和  诊断寄存器**

**def \_setListenOnly(self, value):这将触发只听状态**

**def \_setMode(self, mode):这将触发当前的序列模式**

**def \_setDelimiter(self, char):这改变了串行分隔符**

**def setDiagnostic(self, mapping):这将设置诊断寄存器中的值**

**def getDiagnostic(self, bit):这得到在诊断寄存器中的值**

**Diag\_message.py  
1.class DiagnosticStatusRequest(ModbusRequest):这是所有诊断请求功能的基类**

**def \_\_init\_\_(self, \*\*kwargs):用于诊断请求的基本初始化程序**

**def encode(self):用于诊断响应的基本编码器 我们在self.message中对数据集进行编码**

**def decode(self, data):用于诊断请求的基本解码器**

**2.class DiagnosticStatusResponse(ModbusResponse):这是所有诊断响应功能的基类**

**def \_\_init\_\_(self, \*\*kwargs):用于诊断响应的基本初始化程序**

**def encode(self):用于诊断响应的基本编码器**

**def decode(self, data):用于诊断响应的基本解码器**

**3.class DiagnosticStatusSimpleRequest(DiagnosticStatusRequest):绝大多数的诊断功能很简单 状态请求功能。 他们通过发送0x0000工作 作为数据及其功能代码，并返回 2字节的数据**

**def \_\_init\_\_(self, data=0x0000, \*\*kwargs):一般初始化程序用于简单的诊断请求**

**def execute(self, \*args): 如果不执行 喚起基本函数**

1. **class DiagnosticStatusSimpleResponse(DiagnosticStatusResponse):绝大多数的诊断功能很简单 状态请求功能。 他们通过发送0x0000工作 作为数据及其功能代码，并返回 2字节的数据。**

**def \_\_init\_\_(self, data=0x0000, \*\*kwargs):一个简单的诊断响应通用的初始化**

**5.class ReturnQueryDataRequest(DiagnosticStatusRequest): 在请求数据字段中传递的数据将在响应中返回（循环回）。 整个响应消息应该与请求相同**

**def \_\_init\_\_(self, message=0x0000, \*\*kwargs):初始化请求的新实例**

**def execute(self, \*args):执行回送请求（建立响应）**

1. **class ReturnQueryDataResponse(DiagnosticStatusResponse):在请求数据字段中传递的数据将在响应中返回（循环回）。 整个响应消息应该与请求相同**

**def \_\_init\_\_(self, message=0x0000, \*\*kwargs):初始化响应的新实例**

1. **class RestartCommunicationsOptionRequest(DiagnosticStatusRequest):远程设备串行线路端口必须被初始化并重新启动，并且所有通信事件计数器都被清除。 如果端口当前处于“仅监听”模式，则不返回任何响应。 此功能是将端口设置为仅监听模式的唯一功能。 如果端口当前不处于Listen Only模式，则返回正常响应。 这在执行重新启动之前发生**

**def \_\_init\_\_(self, toggle=False, \*\*kwargs):初始化新请求**

**def execute(self, \*args):清除事件日志并重新启动**

1. **8.class RestartCommunicationsOptionResponse(DiagnosticStatusResponse):远程设备串行线路端口必须被初始化并重新启动，并且所有通信事件计数器都被清除。 如果端口当前处于“仅监听”模式，则不返回任何响应。 此功能是将端口设置为仅监听模式的唯一功能。 如果端口当前不处于Listen Only模式，则返回正常响应。 这在执行重新启动之前发生**

**def \_\_init\_\_(self, toggle=False, \*\*kwargs):初始化一个新的响应**

1. **class ReturnDiagnosticRegisterRequest(DiagnosticStatusSimpleRequest):远程设备的16位诊断寄存器的内容都在响应中返回**

**def execute(self, \*args):执行给定的设备上的诊断请求**

**10.class ReturnDiagnosticRegisterResponse(DiagnosticStatusSimpleResponse):远程设备的16位诊断寄存器的内容都在响应中返回**

**11.class ChangeAsciiInputDelimiterRequest(DiagnosticStatusSimpleRequest):在请求数据字段中传递的字符“CHAR”成为用于未来的消息的消息分隔符（替换默认LF字符）此函数是在ASCII消息的端部notrequired在换行的情况下是有用的端部**

**def execute(self, \*args):执行给定的设备上的诊断请求**

1. **class ChangeAsciiInputDelimiterResponse(DiagnosticStatusSimpleResponse):在请求数据字段中传递的字符“CHAR”成为用于未来的消息（替换默认LFcharacter）在ASCII消息的结束，则不需要此功能在一个换行的情况下是有用的端ofmessage定界符**
2. **class ForceListenOnlyModeRequest(DiagnosticStatusSimpleRequest):将寻址的远程设备强制为仅监听模式进行MODBUS通信**

**def execute(self, \*args):执行给定的设备上的诊断请求**

**14.class ForceListenOnlyModeResponse(DiagnosticStatusResponse)将寻址的远程设备强制为仅用于MODBUS的仅监听模式通信，这将与网络上的其他设备隔绝，在没有响应返回的情况下，允许他们不间断的继续沟通寻址远程设备。**

**def \_\_init\_\_(self, \*\*kwargs):初始化程序阻止返回响应**

1. **class ClearCountersRequest(DiagnosticStatusSimpleRequest):目标是清除ll计数器和诊断寄存器**

**def execute(self, \*args):执行给定的设备上的诊断请求**

1. **class ClearCountersResponse(DiagnosticStatusSimpleResponse):目标是清除ll计数器和诊断寄存器**
2. **class ReturnBusMessageCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回远程设备自上次重新启动，清除计数器操作或上电以来在通信系统上检测到的消息数量**

**def execute(self, \*args):执行给定的设备上的诊断请求**

**18.class ReturnBusMessageCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回远程设备自上次重新启动，清除计数器操作或开机后在通信系统上检测到的消息数量**

**19.class ReturnBusCommunicationErrorCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回远程设备上次重新启动，清除计数器操作或开机后遇到的CRC错误的数量**

**def execute(self, \*args):执行给定的设备上的诊断请求**

1. **class ReturnBusCommunicationErrorCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回远程设备上次重新启动，清除计数器操作或开机后遇到的CRC错误的数量**
2. **class ReturnBusExceptionErrorCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回远程设备上次重新启动，清除计数器操作或开机后返回的modbus异常响应的数量**

**def execute(self, \*args):执行诊断请求**

1. **class ReturnBusExceptionErrorCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回远程设备上次重新启动，清除计数器操作或开机后返回的modbus异常响应的数量**
2. **class ReturnSlaveMessageCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回远程设备自上次重新启动，清除计数器操作或开机以来已处理的远程设备或广播的消息数量**

**def execute(self, \*args):执行诊断请求**

1. **class ReturnSlaveMessageCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回的消息的数量寻址到所述远程装置，或广播，即自上一次重新启动，清除计数器操作，或上电时所述远程设备已经处理**
2. **class ReturnSlaveNoResponseCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回的消息的数量寻址到所述远程装置，或广播，即自上一次重新启动，清除计数器操作，或开机时所述远程设备已经处理**

**def execute(self, \*args):处理诊断请求**

1. **class ReturnSlaveNoReponseCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回的消息的数量寻址到所述远程装置，或广播，即自上一次重新启动，清除计数器操作，或开机时所述远程设备已经处理**
2. **class ReturnSlaveNAKCountRequest(DiagnosticStatusSimpleRequest):**

**响应数据字段返回自从上一次重新启动，清除计数器操作或开机以来，它返回到负值确认（NAK）异常响应的远程设备的消息数量。 异常响应在第7节中进行了描述和列出**

**def execute(self, \*args):执行给定的设备上的诊断请求**

**28.class ReturnSlaveNAKCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回自从上一次重新启动，清除计数器操作或开机以来，它返回到负值确认（NAK）异常响应的远程设备的消息数量。 异常响应在第7节中进行了描述和列出**

**29.class ReturnSlaveBusyCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回寻址到远程设备的消息数量，该数据包返回从设备忙异常响应自上次重启以来，清除计数器操作，或powerup**

**def execute(self, \*args):在给定的设备上执行诊断请求**

**30.class ReturnSlaveBusyCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回寻址到远程设备的消息数量，该数据包返回从设备忙异常响应自上次重启以来，清除计数器操作，或powerup**

1. **class ReturnSlaveBusCharacterOverrunCountRequest(DiagnosticStatusSimpleRequest):响应数据字段返回由于其上次重新启动，清除计数器操作或上电而导致由于字符溢出条件而无法处理的远程设备的消息数量。 字符溢出是由于到达端口的数据字符比可以存储的数据字符或由于硬件故障导致的字符丢失造成的。**

**def execute(self, \*args):执行诊断请求**

**32.class ReturnSlaveBusCharacterOverrunCountResponse(DiagnosticStatusSimpleResponse):响应数据字段返回由于其上次重新启动，清除计数器操作或上电而导致由于字符溢出条件而无法处理的远程设备的消息数量。 字符溢出是由于到达端口的数据字符比可以存储的数据字符或由于硬件故障导致的字符丢失造成的。**

1. **class ReturnIopOverrunCountRequest(DiagnosticStatusSimpleRequest):IOP溢出是由于到达端口的数据字符比可以存储的数据字符更快，或者由于硬件故障而导致的字符丢失。 这个功能是特定于884**

**def execute(self, \*args):执行给定的诊断请求**

**34.class ReturnIopOverrunCountResponse(DiagnosticStatusSimpleResponse):**

**响应数据字段返回的消息的数量给从机，它不能处理因884 IOP溢出条件，自上次重新启动，清除计数器操作或开机**

1. **class ClearOverrunCountRequest(DiagnosticStatusSimpleRequest):清除溢出错误计数器并复位错误标志应该清除错误标志，但在规范提及中没有其他标记，所以它被忽略**

**def execute(self, \*args):执行给定的诊断请求**

**36.class ClearOverrunCountResponse(DiagnosticStatusSimpleResponse):**

**清除溢出错误计数器并复位错误标志**

1. **class GetClearModbusPlusRequest(DiagnosticStatusSimpleRequest):**

**除查询中的功能代码（08）和子功能代码（00 15十六进制）之外，两字节操作字段用于指定“获取统计”或“清除统计信息”操作。 这两个操作是独占的 - “获取”操作无法清除统计信息，“清除”操作在清除之前不会返回统计信息。 在从设备上电时，统计信息也被清除除了查询中的功能代码（08）和子功能代码（00 15十六进制）之外，两字节操作字段用于指定“获取统计信息”或 “清除统计”操作。 这两个操作是独占的 - “获取”操作无法清除统计信息，“清除”操作在清除之前不会返回统计信息。 从设备开机时也会清除统计信息**

**def get\_response\_pdu\_size(self):在响应的数据字段中返回一系列54个16位字（108字节）（该函数与数据字段的通常的两个字节长度不同）。 数据包含从站设备中Modbus Plus对等体处理器的统计信息。 Func\_code（1byte）+子功能码（2字节）+操作（2字节）+数据（108字节）**

**def execute(self, \*args):执行给定的诊断请求**

**def encode(self):用于诊断响应的基本编码器，我们对self.message中的数据集进行编码**

**38.class GetClearModbusPlusResponse(DiagnosticStatusSimpleResponse):**

**在响应的数据字段中返回一系列54个16位字（108字节）（该函数与数据字段的通常的两个字节长度不同）。 数据包含从站设备中Modbus Plus对等体处理器的统计信息**

**Events.py**

**1.class ModbusEvent(object):modbus 事件**

**def encode(self):将状态位编码为事件消息**

**def decode(self, event):将事件消息解码为其状态位**

1. **class RemoteReceiveEvent(ModbusEvent):远程设备MODBUS接收事件**

**def \_\_init\_\_(self, \*\*kwargs): 初始化一个新的实例**

**def encode(self): 将状态位编码为事件消息**

**def decode(self, event):将事件消息解码未其状态位**

1. **class RemoteSendEvent(ModbusEvent):远程设备MODBUS发送事件**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def encode(self):将状态位编码为事件消息**

**def decode(self, event):将事件消息解码为其状态位**

1. **class EnteredListenModeEvent(ModbusEvent):远程设备进入监听模式**

**def encode(self):将状态位编码为事件消息**

**def decode(self, event):将事件消息解码为状态位**

1. **class CommunicationRestartEvent(ModbusEvent):远程设备发起的通信重新启动**

**def encode(self):将状态位编码为事件消息**

**def decode(self, event):将事件消息解码为状态位**

**Exceptions.py**

1. **class ModbusException(Exception):**

**def \_\_init\_\_(self, string):初始化**

**def \_\_str\_\_(self):返回错误**

1. **class ModbusIOException(ModbusException):数据i / o导致错误**

**def \_\_init\_\_(self, string=""):初始化**

1. **class ParameterException(ModbusException):**

**def \_\_init\_\_(self, string=""):初始化异常**

1. **class NoSuchSlaveException(ModbusException):由于向从机发出不存在的请求而导致的错误**

**def \_\_init\_\_(self, string=""):初始化**

1. **class NotImplementedException(ModbusException):由于未实现的功能导致错误**

**def \_\_init\_\_(self, string=""):初始化**

**6.class ConnectionException(ModbusException):错误连接造成的异常错误**

**def \_\_init\_\_(self, string=""):初始化异常**

**7.class InvalidMessageRecievedException(ModbusException):接收或解码无效响应而导致的错误**

**def \_\_init\_\_(self, string=""):初始化异常**

**Factory.py**

**1. class ServerDecoder(IModbusDecoder):请求消息工厂（服务器）**

**def \_\_init\_\_(self):初始化客户端查找表**

**def decode(self, message):包装器解码请求包**

**def lookupPduClass(self, function\_code):使用`function\_code`来确定PDU的类别**

**def \_helper(self, data):这个工厂被用于生成从一个有效的请求分组中的正确的请求对象。 这从当前实现的请求类型的列表中解码**

**2.class ClientDecoder(IModbusDecoder):响应消息工厂（客户端）**

**def \_\_init\_\_(self):初始化客户端查找表**

**def lookupPduClass(self, function\_code):使用`function\_code`来确定PDU的类别**

**def decode(self, message):包装器解码响应包**

**def \_helper(self, data):该工厂用于从有效的响应数据包生成正确的响应对象。 这从当前实现的请求类型的列表中解码**

**File\_message.py**

**1.class FileRecord(object):表示文件记录及其相关数据**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def \_\_eq\_\_(self, relf):将左对象与右对齐**

**def \_\_ne\_\_(self, relf):将左对象与右对齐**

**def \_\_repr\_\_(self):提供文件记录的数据**

1. **class ReadFileRecordRequest(ModbusRequest):此功能代码用于执行文件记录读取。 所有请求数据长度以字节数提供，并且所有记录长度都是根据寄存器提供的**

**def \_\_init\_\_(self, records=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):对传入的请求进行解码**

**def execute(self, context):对存储区运行一个read exeception状态请求**

1. **class ReadFileRecordResponse(ModbusResponse):正常响应是一系列“子响应”，每个“子请求”一个。 字节计数字段是所有“子响应”中的总字节数的总和。 此外，每个“子响应”都包含一个显示自己的字节数的字段**

**def \_\_init\_\_(self, records=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data): 解码响应**

1. **class WriteFileRecordRequest(ModbusRequest):此功能代码用于执行文件记录写入。 所有请求数据长度以字节数提供，并且所有记录长度以16位字的数量提供**

**def \_\_init\_\_(self, records=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):解码传入的请求**

**def execute(self, context):根据上下文运行写入文件记录请求**

1. **class WriteFileRecordResponse(ModbusResponse):正常响应是请求的回应**

**def \_\_init\_\_(self, records=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):解码传入的请求**

1. **class ReadFifoQueueRequest(ModbusRequest):该功能代码允许读取远程设备中寄存器的先入先出（FIFO）队列的内容。 该函数返回队列中的寄存器的计数，后跟排队的数据。最多可以读取32个寄存器：计数，最多31个队列数据寄存器**

**def \_\_init\_\_(self, address=0x0000, \*\*kwargs):初始化一个新的实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):解码传入的请求**

**def execute(self, context):针对运行在store读取异常状态请求**

1. **class ReadFifoQueueResponse(ModbusResponse):在正常响应中，字节计数显示要跟踪的字节数，包括队列计数字节和值寄存器字节（但不包括错误检查字段）。 队列计数是队列中数据寄存器的数量（不包括计数寄存器）。**

**def calculateRtuFrameSize(cls, buffer):计算消息的大小**

**def \_\_init\_\_(self, values=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):对响应进行解码**

**Interfaces.py**

1. **class Singleton(object):基类**

**def \_\_new\_\_(cls, \*args, \*\*kwargs):创建一个新的实例**

**2.class IModbusDecoder(object):Modbus解码器基类**

**def decode(self, message):包装器来解码给定的数据包**

**def lookupPduClass(self, function\_code):使用`function\_code`来确定PDU的类别**

1. **class IModbusFramer(object):成帧策略界面。 这个想法是我们抽出所有的东西 详细介绍如何检测当前消息帧是否存在，解码它，发送它等，以便我们可以插入一个新的Framer对象（tcp，rtu，ascii）**

**def checkFrame(self):检查并解码下一帧**

**def advanceFrame(self):跳过当前的框架消息这允许我们在处理它或确定它包含错误之后跳过当前消息。 它还必须重置当前帧头句柄**

**def addToFrame(self, message):将下一条消息添加到帧缓冲区在解码之前应该使用循环，将接收到的数据添加到缓冲区句柄**

**def isFrameReady(self):检查我们是否应该继续解码逻辑这是为了在解码阶段的while循环中使用，让解码器知道缓冲区中仍然有数据。**

**def getFrame(self):从缓冲区获取下一帧**

**def populateResult(self, result):填充具有当前帧报头中的MODBUS结果**

**def processIncomingPacket(self, data, callback): 新的数据包处理模式**

**def buildPacket(self, message):创建一个准备发送modbus数据包**

1. **class IModbusSlaveContext(object):MODBUS从站数据环境的接口**

**def decode(self, fx):将功能代码转换为数据存储**

**def reset(self):将所有数据存储重置为其默认值**

**def validate(self, fx, address, count=1):验证请求以确保它在范围内**

**def getValues(self, fx, address, count=1):验证请求以确保它在范围内**

**def setValues(self, fx, address, values):使用提供的值设置数据存储**

1. **class IPayloadBuilder(object):这是一个可以为modbus寄存器写命令构建有效负载的类的接口。 它应该将编码数据的编解码器抽象为所需的格式（bcd，binary，char等）**

**def build(self):将有效负载缓冲区作为列表返回**

**Mei\_message.py**

**1.class ReadDeviceInformationRequest(ModbusRequest):该功能代码仅允许读取相对于远程设备的物理和功能描述的标识和附加信息**

**def \_\_init\_\_(self, read\_code=None, object\_id=0x00, \*\*kwargs):**

**初始化一个新的实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):解码消息的数据部分**

**def execute(self, context):对store运行读取异常状态请求**

**def \_\_str\_\_(self):构建请求的表示**

**2.class ReadDeviceInformationResponse(ModbusResponse): 获取响应信息**

**def calculateRtuFrameSize(cls, buffer):计算消息的大小**

**def \_\_init\_\_(self, read\_code=None, information=None, \*\*kwargs):**

**初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):解码响应**

**def \_\_str\_\_(self):构建响应的表示**

**Other\_message.py**

1. **class ReadExceptionStatusRequest(ModbusRequest):此功能代码用于读取远程设备中八个异常状态输出的内容。 该函数提供了访问此信息的简单方法，因为异常输出引用已知（在函数中不需要输出引用）**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def encode(self):对消息进行编码**

**def decode(self, data):解码消息的数据部分。**

**def execute(self, context=None):对store运行读取异常状态请求**

**def \_\_str\_\_(self):构建请求的表示**

1. **class ReadExceptionStatusResponse(ModbusResponse):正常响应包含八个异常状态输出的状态。 输出被打包成一个数据字节，每个输出一个位。 最低输出参考值的状态包含在该字节的最低有效位中。 八个异常状态输出的内容是器件特定的**

**def \_\_init\_\_(self, status=0x00, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):解码响应**

**def \_\_str\_\_(self):构建响应的表示**

**3.class GetCommEventCounterRequest(ModbusRequest):该功能代码用于从远程设备的通信事件计数器获取状态字和事件计数**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def encode(self):对消息进行编码**

**def decode(self, data):解码消息的数据部分。**

**def execute(self, context=None):针对运行在store读取异常状态请求**

**def \_\_str\_\_(self):构建请求的表示**

1. **class GetCommEventCounterResponse(ModbusResponse):正常响应包含一个两字节的状态字和两个字节的事件计数。 如果先前发出的程序命令仍由远程设备处理（状态为忙），状态字将为全部（FF FF十六进制）。 否则，状态字将全为零**

**def \_\_init\_\_(self, count=0x0000, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):解码响应**

**def \_\_str\_\_(self):建立响应的表示**

**5.class GetCommEventLogRequest(ModbusRequest):此功能代码用于从远程设备获取状态字，事件计数，消息计数和事件字节字段**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def encode(self):对消息进行编码**

**def decode(self, data):解码消息的数据部分**

**def execute(self, context=None):针对运行在store读取异常状态请求**

**def \_\_str\_\_(self):构建请求的表示**

**6.class GetCommEventLogResponse(ModbusResponse): 正常响应包含两个字节的状态字字段，两个字节的事件计数字段，一个两字节的消息计数字段以及一个包含0-64字节事件的字段。 字节计数字段定义了这四个字段中数据的总长度**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):解码响应**

**def \_\_str\_\_(self):建立响应的表示**

**7.class ReportSlaveIdRequest(ModbusRequest):此功能代码用于读取特定于远程设备的类型，当前状态和其他信息的描述**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行解码**

**def decode(self, data):解码消息的数据部分。**

**def execute(self, context=None):针对运行在store读取异常状态请求**

**def \_\_str\_\_(self):构建请求的表示**

1. **class ReportSlaveIdResponse(ModbusResponse):正常响应的格式如下例所示。 数据内容特定于每种类型的设备**

**def \_\_init\_\_(self, identifier=b'\x00', status=True, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):对响应进行解码**

**def \_\_str\_\_(self):建立响应的表示**

**Payload.py**

1. **class BinaryPayloadBuilder(IPayloadBuilder):一个实用程序，可帮助构建要用各种modbus消息写入的有效负载消息。 它实际上只是一个简单的包装框架结构模块，但它可以节省时间查找格式字符串。**

**def \_\_init\_\_(self, payload=None, endian=Endian.Little):初始化有效载荷构建器的新实例**

**def to\_string(self):返回有效负载缓冲区作为字符串**

**def \_\_str\_\_(self):返回有效负载缓冲区作为字符串**

**def reset(self):重置有效载荷缓冲区**

**def to\_registers(self):将有效负载缓冲区转换为可用作上下文块的寄存器布局**

**def build(self):将有效负载缓冲区作为列表返回**

**def add\_bits(self, values):添加要编码的位的集合**

**def add\_8bit\_uint(self, value):在缓冲区中添加一个8位unsigned int**

**def add\_16bit\_uint(self, value):向缓冲区添加一个16位unsigned int**

**def add\_32bit\_uint(self, value):向缓冲区添加一个32位unsigned int**

**def add\_64bit\_uint(self, value):向缓冲区添加64位无符号int**

**def add\_8bit\_int(self, value):在缓冲区中添加一个8位有符号的int**

**def add\_16bit\_int(self, value):在缓冲区中添加一个16位有符号的int**

**def add\_32bit\_int(self, value):在缓冲区中添加一个32位有符号的int**

**def add\_64bit\_int(self, value):在缓冲区中添加一个64位有符号的int**

**def add\_32bit\_float(self, value):向缓冲区添加一个32位浮点数**

**def add\_64bit\_float(self, value):向缓冲区添加64位浮点数（double）**

**def add\_string(self, value):向缓冲区添加一个字符串**

**2.class BinaryPayloadDecoder(object):一个有助于解码来自modbus响应消息的有效负载消息的实用程序。 它实际上只是一个简单的包装框架结构模块，但它可以节省时间查找格式字符串。 以下是一个简单的例子**

**def \_\_init\_\_(self, payload, endian=Endian.Little):初始化新的有效载荷解码器**

**def fromRegisters(klass, registers, endian=Endian.Little):初始化有效载荷解码器，结果是从modbus设备读取寄存器的集合**

**def fromCoils(klass, coils, endian=Endian.Little):初始化有效载荷解码器，结果是从modbus设备读取线圈集合**

**def reset(self):将解码器指针重新设置为开始**

**def decode\_8bit\_uint(self):从缓冲区中解码一个8位无符号int**

**def decode\_bits(self):从缓冲区解码一个字节值的位**

**def decode\_16bit\_uint(self):从缓冲区解码16位无符号int**

**def decode\_32bit\_uint(self):从缓冲区解码32位无符号int**

**def decode\_64bit\_uint(self):从缓冲区解码一个64位的unsigned int**

**def decode\_8bit\_int(self):从缓冲区中解码一个8位有符号的int**

**def decode\_16bit\_int(self):从缓冲区解码16位有符号的int**

**def decode\_32bit\_int(self):从缓冲区解码一个32位有符号的int**

**def decode\_64bit\_int(self):从缓冲区解码一个64位有符号的int**

**def decode\_32bit\_float(self):从缓冲区解码一个32位浮点数**

**def decode\_64bit\_float（self）：从缓冲区解码64位浮点数（双）**

**def decode\_string(self, size=1):从缓冲区解码一个字符串**

**def skip\_bytes(self, nbytes):在缓冲区中跳过n个字节**

**Pdu.py**

1. **class ModbusPDU(object):所有Modbus消息的基类**

**def \_\_init\_\_(self, \*\*kwargs):初始化modbus请求的基本数据**

**def encode(self):对消息进行编码**

**def decode(self, data):解码消息的数据部分。**

**def calculateRtuFrameSize(cls, buffer):计算PDU的大小**

1. **class ModbusRequest(ModbusPDU):用于modbus请求PDU的基类**

**def \_\_init\_\_(self, \*\*kwargs):代理到较低级别的初始化程序**

**def doException(self, exception): 基于该函数构建错误响应**

**3.class ModbusResponse(ModbusPDU): 用于modbus响应PDU的基类**

**def \_\_init\_\_(self, \*\*kwargs):代理到较低级别的初始化程序**

**4.class ModbusExceptions(Singleton):枚举有效的modbus异常**

**def decode(cls, code):给出错误代码，将其转换为字符串错误名称**

**5.class ExceptionResponse(ModbusResponse):modbus异常PDU的基类**

**def \_\_init\_\_(self, function\_code, exception\_code=None, \*\*kwargs):初始化modbus异常响应**

**def encode(self):对modbus异常响应进行编码**

**def decode(self, data):解码modbus异常响应**

**def \_\_str\_\_(self):构建异常响应的表示**

**6.class IllegalFunctionRequest(ModbusRequest):定义Modbus从站异常类型“非法功能”如果从站返回此异常代码**

**def \_\_init\_\_(self, function\_code, \*\*kwargs):初始化一个IllegalFunctionRequest**

**def decode(self, data):这是这样的，所以这个失败会正常运行**

**def execute(self, context):构建非法的函数请求错误响应**

**Register\_read\_message.py**

**1class ReadRegistersRequestBase(ModbusRequest):用于读取modbus寄存器的基础类**

**def \_\_init\_\_(self, address, count, \*\*kwargs):初始化一个新的实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):解码一个注册请求包**

**def get\_response\_pdu\_size(self):Func\_code（1字节）+字节计数（1字节）+ 2 \*线圈数量（n字节）**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**2.class ReadRegistersResponseBase(ModbusResponse):用于响应modbus寄存器的基类读取**

**def \_\_init\_\_(self, values, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应数据包进行编码**

**def decode(self, data):解码一个注册响应包**

**def getRegister(self, index):获取请求的注册**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**3.class ReadHoldingRegistersRequest(ReadRegistersRequestBase):此功能码用于读取远程设备中保持寄存器的连续块的内容。 请求PDU指定起始寄存器地址和寄存器数量。 在PDU寄存器从零开始寻址。 因此，编号为1-16的寄存器的地址为0-15**

**def \_\_init\_\_(self, address=None, count=None, \*\*kwargs):初始化请求的新实例**

**def execute(self, context):对数据存储区执行读取保留请求**

**4.class ReadHoldingRegistersResponse(ReadRegistersResponseBase):此功能码用于读取远程设备中保持寄存器的连续块的内容。 请求PDU指定起始寄存器地址和寄存器数量。 在PDU寄存器从零开始寻址。 因此，编号为1-16的寄存器的地址为0-15**

**def \_\_init\_\_(self, values=None, \*\*kwargs):初始化一个新的响应实例**

**5.class ReadInputRegistersRequest(ReadRegistersRequestBase):此功能码用于从1到大约读取。 远程设备中有125个连续的输入寄存器。 请求PDU指定起始寄存器地址和寄存器数量。 在PDU寄存器从零开始寻址。 因此，编号为1-16的输入寄存器的地址为0-15**

**def \_\_init\_\_(self, address=None, count=None, \*\*kwargs):初始化一个新的响应实例**

**def execute(self, context):针对数据存储区运行读取输入请求**

**6.class ReadInputRegistersResponse(ReadRegistersResponseBase):此功能码用于从1到大约读取。 远程设备中有125个连续的输入寄存器。 请求PDU指定起始寄存器地址和寄存器数量。 在PDU寄存器从零开始寻址。 因此，编号为1-16的输入寄存器的地址为0-15**

**def \_\_init\_\_(self, values=None, \*\*kwargs):初始化一个新的响应实例**

**7.class ReadWriteMultipleRegistersRequest(ModbusRequest):该功能代码在单个MODBUS事务中执行一次读取操作和一个写入操作的组合。 在读取之前执行写入操作**

**def \_\_init\_\_(self, \*\*kwargs):初始化一个新的响应实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):解码注册请求包**

**def execute(self, context):对数据存储区执行写入单个注册请求**

**def get\_response\_pdu\_size(self):Func\_code（1字节）+字节计数（1字节）+ 2 \*线圈数量（n字节）**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**8.class ReadWriteMultipleRegistersResponse(ModbusResponse):正常响应包含读取的寄存器组中的数据。 字节计数字段指定读取数据字段中要跟随的字节数**

**def \_\_init\_\_(self, values=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):对响应数据包进行编码**

**def decode(self, data):解码寄存器响应数据包**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**Register\_write\_message.py**

1. **class WriteSingleRegisterRequest(ModbusRequest):该功能码用于在远程设备中写入单个保持寄存器**

**def \_\_init\_\_(self, address=None, value=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):编写单个注册包数据包请求**

**def decode(self, data):解码一个写入单个寄存器包数据包请求**

**def execute(self, context):对数据存储区执行写入单个注册请求**

**def get\_response\_pdu\_size(self):Func\_code（1字节）+寄存器地址（2字节）+寄存器值（2字节）**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**2.class WriteSingleRegisterResponse(ModbusResponse):正常响应是在写入寄存器内容之后返回的请求的回应**

**def \_\_init\_\_(self, address=None, value=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):编写单个注册包数据包请求**

**def decode(self, data):解码一个写入单个寄存器包数据包请求**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**3.class WriteMultipleRegistersRequest(ModbusRequest):该功能码用于在远程设备中写入一连串的寄存器（1到约120个寄存器）**

**def \_\_init\_\_(self, address=None, values=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):编写单个注册包数据包请求**

**def decode(self, data):解码一个写入单个寄存器包数据包请求**

**def execute(self, context):对数据存储区执行写入单个注册请求**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**4.class WriteMultipleRegistersResponse(ModbusResponse):正常响应返回写入的寄存器的功能代码，起始地址和数量**

**def \_\_init\_\_(self, address=None, count=None, \*\*kwargs):初始化一个新的实例**

**def encode(self):编写单个注册包数据包请求**

**def decode(self, data):解码一个写入单个寄存器包数据包请求**

**def \_\_str\_\_(self):返回实例的字符串表示形式**

**5.class MaskWriteRegisterRequest(ModbusRequest):该功能码用于使用AND掩码，OR掩码和寄存器当前内容的组合来修改指定保持寄存器的内容。 该功能可用于设置或清除寄存器中的各个位**

**def \_\_init\_\_(self, address=0x0000, and\_mask=0xffff, or\_mask=0x0000, \*\*kwargs):初始化一个新的实例**

**def encode(self):对请求数据包进行编码**

**def decode(self, data):解码传入的请求**

**def execute(self, context):对store执行掩码写入注册请求**

**6.class MaskWriteRegisterResponse(ModbusResponse):正常响应是请求的回应。 寄存器写入后返回响应**

**def \_\_init\_\_(self, address=0x0000, and\_mask=0xffff, or\_mask=0x0000,初始化一个新的实例**

**def encode(self):对响应进行编码**

**def decode(self, data):解码响应**

**Transaction.py**

**1.class ModbusTransactionManager(object):Impelements事务的管理者**

**def \_\_init\_\_(self, client, \*\*kwargs):初始化ModbusTransactionManager的实例**

**def \_set\_adu\_size(self):基于ADU大小的modbus帧以字节为单位**

**def \_calculate\_response\_length(self, expected\_pdu\_size):获取长度**

**def \_calculate\_exception\_length(self):根据Framer的类型返回Modbus异常响应的长度**

**def \_check\_response(self, response):检查响应是否为Modbus异常。**

**def execute(self, request):启动生产者将下一个请求发送到consumer.write（Frame（request））、**

**def \_send(self, packet):发送**

**def addTransaction(self, request, tid=None):向处理程序添加一个事务**

**def getTransaction(self, tid):返回与引用的tid匹配的事务**

**def delTransaction(self, tid):删除与引用的tid匹配的事务**

**def getNextTID(self):检索下一个唯一的事务标识符**

**def reset(self):重置事务标识符**

1. **class DictTransactionManager(ModbusTransactionManager):根据提供的交易ID，对结果进行键入的Impelements 进行交易**

**def \_\_init\_\_(self, client, \*\*kwargs):初始化ModbusTransactionManager的一个实例**

**def \_\_iter\_\_(self):迭代当前的管理事务**

**def addTransaction(self, request, tid=None):向处理程序添加一个事务**

**def getTransaction(self, tid):返回与引用的tid匹配的事务**

**def delTransaction(self, tid):删除与引用的tid匹配的事务**

**def getNextTID(self):检索下一个唯一的事务标识符**

**def reset(self):重置事务标识符**

**3.class FifoTransactionManager(ModbusTransactionManager):Impelements事务的管理者，其中结果以FIFO方式返回**

**def \_\_init\_\_(self, client, \*\*kwargs):初始化ModbusTransactionManager的实例**

**def \_\_iter\_\_(self):迭代当前的管理事务**

**def addTransaction(self, request, tid=None):向处理程序添加一个事务**

**def getTransaction(self, tid):返回与引用的tid匹配的事务**

**def delTransaction(self, tid):删除与引用的tid匹配的事务**

**4.class ModbusSocketFramer(IModbusFramer):Modbus Socket Frame控制器**

**def \_\_init\_\_(self, decoder):初始化成帧器的新实例**

**def checkFrame(self)检查并解码下一个框架如果我们成功，返回true**

**def advanceFrame(self):跳过当前的框架消息这允许我们在处理它或确定它包含错误之后跳过当前消息。 它还必须重置当前帧头句柄**

**def isFrameReady(self):检查我们是否应该继续解码逻辑这是为了在解码阶段的while循环中使用，让解码器工厂知道缓冲区中仍然有数据**

**def addToFrame(self, message):将新的数据包数据添加到当前帧缓冲区**

**def getFrame(self):从缓冲数据返回下一帧**

**def populateResult(self, result):使用传输特定头信息（pid，tid，uid，checksum等）填充modbus结果**

**def processIncomingPacket(self, data, callback):新的数据包处理模式**

**def \_process(self, callback, error=False):处理进入的数据包而不管错误情况**

**def resetFrame(self):重置整个消息帧**

**def getRawFrame(self):返回完整的缓冲区**

**def buildPacket(self, message):创建一个准备发送modbus数据包**

**5.class ModbusRtuFramer(IModbusFramer):Modbus RTU帧控制器**

**def \_\_init\_\_(self, decoder):初始化成帧器的新实例**

**def checkFrame(self): 检查下一帧是否可用。 如果我们成功，返回True。**

**def advanceFrame(self):跳过当前的框架消息这允许我们在处理它或确定它包含错误之后跳过当前消息。 它还必须重置当前帧头句柄**

**def resetFrame(self):重置整个消息帧**

**def isFrameReady(self):检查我们是否应该继续解码逻辑这是为了在解码阶段的while循环中使用，让解码器知道缓冲区中仍然有数据**

**def populateHeader(self):尝试设置头文件“uid”，“len”和“crc”**

**def addToFrame(self, message):在解码之前应该使用这个循环，将接收到的数据添加到缓冲区句柄**

**def getFrame(self):从缓冲区获取下一帧**

**def populateResult(self, result):填充modbus结果标题**

**def processIncomingPacket(self, data, callback):新的数据包处理模式**

**def buildPacket(self, message):创建一个准备发送modbus数据包**

**def \_process(self, callback, error=False):处理进入的数据包而不管错误情况**

**def getRawFrame(self):返回完整的缓冲区**

1. **class ModbusAsciiFramer(IModbusFramer):Modbus ASCII帧控制器**

**def \_\_init\_\_(self, decoder):初始化成帧器的新实例**

**def checkFrame(self):检查并解码下一帧**

**def advanceFrame(self):跳过当前框架的消息**

**def isFrameReady(self):检查我们是否应该继续解码逻辑这是为了在解码阶段的while循环中使用，让解码器知道缓冲区中仍然有数据**

**def addToFrame(self, message):将下一条消息添加到帧缓冲区**

**def getFrame(self):从缓冲区获取下一帧**

**def resetFrame(self):重置整个消息帧**

**def populateResult(self, result):填充modbus结果标题**

**def processIncomingPacket(self, data, callback):新的数据包处理模式**

**def buildPacket(self, message):创建一个准备发送modbus数据包建立在modbus请求/响应之上**

1. **class ModbusBinaryFramer(IModbusFramer):Modbus二进制帧控制器**

**def \_\_init\_\_(self, decoder):初始化成帧器的新实例**

**def checkFrame(self):检查并解码下一帧**

**def advanceFrame(self):跳过当前框架的消息**

**def isFrameReady(self):检查我们是否应该继续解码逻辑**

**def addToFrame(self, message):将下一条消息添加到帧缓冲区**

**def getFrame(self):从缓冲区获取下一帧**

**def populateResult(self, result):填充modbus结果标题**

**def processIncomingPacket(self, data, callback):新的数据包处理模式**

**def buildPacket(self, message):创建一个准备发送modbus数据包**

**def \_preflight(self, data):预检缓冲区测试**

**def resetFrame(self):重置整个消息帧。**

**Utilities.py**

**def default(value):给定一个python对象，返回该对象的默认值**

**def dict\_property(store, index):帮助从字典创建类属性。 基本上这可以让您删除很多可能的样板代码**

**def pack\_bitstring(bits):从位数组中创建一个字符串**

**def unpack\_bitstring(string):从字符串中创建位数组**

**def make\_byte\_string(s):从给定的字符串返回字节串**

**def \_\_generate\_crc16\_table():生成一个crc16查询表**

**def computeCRC(data):在传入的字符串中计算一个crc16。 对于modbus，这仅用于二进制串行协议（在本例中为RTU）**

**def checkCRC(data, check):检查数据是否与CRC中传递的数据匹配**

**def computeLRC(data):用于计算针对字符串的纵向冗余校验。 这仅用于串行ASCII modbus协议。 有关此实现的完整描述，请参见串行线路模块描述的附录B。**

**def checkLRC(data, check):检查传入的数据是否与LRC匹配**

**def rtuFrameSize(data, byte\_count\_pos):基于字节数计算帧的大小**