train_group_reader_oop.py 代码逐行解析

1. 文件头部信息

```python
# -*- coding: utf-8 -*-
```

- 指定文件编码为UTF-8，确保中文注释正常显示

```python
"""
train_group_reader_oop.py
OOP 版"集群读取"脚本，保留原逻辑与节奏，新增：
1) 线程安全的全局数据仓库 SharedDataStore
2) 每次 query 后更新：
   - RxAddr, RxFuncID, RxDataLen, RxData, Mdbs_state, function_name
   - all_data_dict[function_name] = 最终可展示的数据
...
"""
```

- 文档字符串，说明脚本用途和主要改进点
- 强调了线程安全和数据更新机制

2. 导入模块

```python
import sys, time, threading
from PyQt5.QtCore import QObject, pyqtSignal, QTimer, QThread
from PyQt5.QtWidgets import QApplication
```

- `sys, time, threading`: 系统基础模块
- `PyQt5`: GUI框架，这里用于信号槽机制和多线程

```python
```

```
from modbus_send import (
    Serial_Qthread_function,
    SerialThread,
    ModbusSender,
    modbus_RTU,
    wait_for_last_frame,
)
from modbus_receive import (
    Modbus_receive_Interface,
    Frame_OK,
)
```

- 导入现有的Modbus发送和接收模块
- 复用已有的串口通信和帧处理功能

3. SharedDataStore类 - 线程安全的全局数据仓库

3.1 类定义和信号

```python
class SharedDataStore(QObject):
    snapshotUpdated = pyqtSignal(dict)        # 推送全量快照
    oneFuncUpdated = pyqtSignal(str, dict)    # 推送某个功能函数的结果
```

- 继承自`QObject`，支持PyQt信号槽机制
- 定义两个信号：全量快照更新和单个功能更新

3.2 单例模式实现

```python
    _instance = None
    _lock_inst = threading.Lock()

    @classmethod
    def instance(cls):
        with cls._lock_inst:
            if cls._instance is None:
                cls._instance = SharedDataStore()
            return cls._instance
```

- 实现线程安全的单例模式
- 确保全局只有一个数据仓库实例

3.3 数据字段初始化

```python
def __init__(self):
    super().__init__()
    # === 原全局变量：接受帧内容 ===
    self.RxAddr = 0
    self.RxFuncID = 0
    self.RxDataLen = 0
    self.RxData = []
    self.Mdbs_state = 0
    self.function_name = ""
    # === 原全局变量：汇总展示用字典 ===
    self.all_data_dict = {}

    # 线程并发保护
    self._lock = threading.Lock()
```

- 初始化原全局变量对应的实例变量
- `RxAddr`: 接收地址
- `RxFuncID`: 功能码
- `RxDataLen`: 数据长度
- `RxData`: 数据内容
- `Mdbs_state`: Modbus状态
- `function_name`: 当前处理的功能名称
- `all_data_dict`: 存储所有功能的最终结果
- `_lock`: 线程锁，保证并发安全

3.4 数据写入方法

```python
def write_last_frame(self, fn_name: str, RxAddr: int, RxFuncID: int,
                     RxDataLen: int, RxData, Mdbs_state: int):
    with self._lock:
        self.function_name = fn_name
        self.RxAddr = RxAddr
        self.RxFuncID = RxFuncID
        self.RxDataLen = RxDataLen
        self.RxData = RxData
        self.Mdbs_state = Mdbs_state
```

- 线程安全地更新最近接收帧的所有字段
- 使用`with self._lock`确保原子操作

```python
def write_func_result(self, fn_name: str, final_result: dict):
    with self._lock:
        self.all_data_dict[fn_name] = final_result

        self.oneFuncUpdated.emit(fn_name, final_result)
        snap = self._make_snapshot_locked()
        self.snapshotUpdated.emit(snap)
```

- 更新某个功能的最终结果
- 发射两个信号通知订阅者：单个功能更新和全量快照更新

3.5 快照生成

```python
def _make_snapshot_locked(self) -> dict:
    return {
        "RxAddr": self.RxAddr,
        "RxFuncID": self.RxFuncID,
        "RxDataLen": self.RxDataLen,
        "RxData": list(self.RxData) if isinstance(self.RxData, (list, tuple)) else self.RxData,
        "Mdbs_state": self.Mdbs_state,
        "function_name": self.function_name,
        "all_data_dict": dict(self.all_data_dict),
        "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
    }
```

- 在持锁状态下创建全量数据快照
- 进行浅拷贝避免外部修改影响内部数据
- 添加时间戳标记快照生成时间

4. SerialPortWorker类 - 串口工作者

```python
```

```
class SerialPortWorker(QObject):
    frameReceived = pyqtSignal(bytes)  # 完整帧接收信号

    def __init__(self, port_name: str, baudrate: int):
        super().__init__()
        self._port_name = port_name
        self._baudrate = baudrate
        self.serial_obj = Serial_Qthread_function()
        self.thread = SerialThread(self.serial_obj)
        self.serial_obj.moveToThread(self.thread)
```

- 封装串口操作为独立的工作者对象
- 使用多线程处理串口通信，避免阻塞主线程

```
python

        # 基本信号连接
        self.serial_obj.signal_pushButton_Open.connect(self.serial_obj.slot_pushButton_Open)
        self.serial_obj.signal_SendData.connect(self.serial_obj.slot_SendData)
```

- 连接串口对象的内部信号和槽

```
python

    def start(self):
        self.thread.start()
        open_param = {'PortName_master': self._port_name, 'BaudRate_master': self._baudrate}
        self.serial_obj.signal_pushButton_Open.emit(open_param)
```

- 启动串口线程并发送打开串口的信号

5. ModbusClient类 - Modbus客户端

```
python

class ModbusClient(QObject):
    def __init__(self, serial_worker: SerialPortWorker):
        super().__init__()
        self.serial_worker = serial_worker
        self._sender = ModbusSender()
```

- 组合串口工作者和Modbus发送器

```
python
```

```python
    def modbus_query(self, slave_addr: int, func_id: int, start_addr: int, quantity: int):
        # 构造无 CRC 的 PDU
        if func_id == 1:
            hex_no_crc = self._sender.read_coils(slave_addr, start_addr, quantity)
        elif func_id == 2:
            hex_no_crc = self._sender.read_discrete_inputs(slave_addr, start_addr, quantity)
        # ... 其他功能码处理
```

- 根据不同功能码构造相应的Modbus请求
- 功能码1-4为读操作，5-6为单个写操作，15-16为批量写操作

```python
        # 加 CRC -> 发送
        payload = modbus_RTU(hex_no_crc)
        self.serial_worker.send_bytes(payload)

        # 等待一帧（带超时）
        hex_str = wait_for_last_frame(self.serial_worker.serial_obj, timeout_ms=2000)
        return hex_str
```

- 添加CRC校验后发送数据
- 等待响应帧，设置2秒超时

6. FrameParser类 - 帧解析器

```python
class FrameParser(QObject):
    def parse(self, hex_str: str):
        # 返回：RxAddr, RxFuncID, RxDataLen, RxData, Mdbs_state, MdbsCNT
        return Modbus_receive_Interface(hex_str)
```

- 简单封装现有的帧解析接口
- 将十六进制字符串解析为结构化数据

7. Analyzer类 - 结果分析器

```python
class Analyzer(QObject):
    def analyze(self, func_name: str, parsed: dict) -> dict:
        final_dict = dict(parsed)  # 浅拷贝
        return final_dict
```

- 默认透传解析结果
- 预留接口供后续扩展业务逻辑处理

8. GroupQueryScheduler类 - 查询调度器

8.1 初始化和信号定义

```python
class GroupQueryScheduler(QObject):
    oneQueryFinished = pyqtSignal(str, dict)  # (func_name, result_dict)
    oneRoundFinished = pyqtSignal()

    def __init__(self, modbus_client: ModbusClient, parser: FrameParser,
                 functions: list, cycle_interval_ms: int = 5000, analyzer: Analyzer = None):
```

- 定义查询完成和轮次完成信号
- 接受Modbus客户端、解析器、功能列表和循环间隔参数

8.2 定时器设置

```python
        self._step_timer = QTimer(self)
        self._step_timer.setSingleShot(True)
        self._step_timer.timeout.connect(self._do_one)

        self._between_round_timer = QTimer(self)
        self._between_round_timer.setSingleShot(True)
        self._between_round_timer.timeout.connect(self.start)
```

- 使用两个单次定时器控制调度节奏
- `_step_timer`: 控制单个查询间隔（1秒）
- `_between_round_timer`: 控制轮次间隔（默认5秒）

8.3 核心调度逻辑

```python
    def _do_one(self):
        if self._idx >= len(self.functions):
            print(f"所有查询执行完成，本轮结束。等待 {self._cycle_interval_ms/1000} 秒进入下一轮...")
            self.oneRoundFinished.emit()
            self._between_round_timer.start(self._cycle_interval_ms)
            return
```

- 检查是否完成一轮查询，如是则等待下一轮

```python
    func = self.functions[self._idx]
    func_name = func.__name__
    print(f"[GroupQuery] 执行第 {self._idx+1}/{len(self.functions)} 个：{func_name}")

    slave, fid, start, qty = func()
```

- 获取当前要执行的功能函数
- 调用函数获取Modbus查询参数

```python
    # —— 发送并等待帧 —— #
    hex_str = self.client.modbus_query(slave, fid, start, qty)

    result = {'hex': None, 'parsed': None, 'ok': False}
```

- 执行Modbus查询并初始化结果字典

8.4 结果处理

```python
    if hex_str:
        result['hex'] = hex_str
        RxAddr, RxFuncID, RxDataLen, RxData, Mdbs_state, MdbsCNT = self.parser.parse(hex_str)
        parsed = {
            'RxAddr': RxAddr, 'RxFuncID': RxFuncID,
            'RxDataLen': RxDataLen, 'RxData': RxData,
            'Mdbs_state': Mdbs_state, 'MdbsCNT': MdbsCNT
        }
        result['parsed'] = parsed
        ok = bool(Mdbs_state & Frame_OK)
        result['ok'] = ok
```

- 解析接收到的响应帧
- 检查帧的有效性

```python
```

```
        # —— 【关键】写入"最近一帧"的全局字段 —— #
        DATAS.write_last_frame(func_name, RxAddr, RxFuncID, RxDataLen, RxData, Mdbs_state)

        if ok:
            # —— 分析成 UI 更友好的结果（如无规则则透传 parsed） —— #
            final_result = self._analyzer.analyze(func_name, parsed)

            # —— 【关键】写入 all_data_dict[function_name] —— #
            DATAS.write_func_result(func_name, final_result)
```

- 更新全局数据仓库
- 对有效帧进行进一步分析处理

9. PassiveListener类 - 被动监听器

python

```
class PassiveListener(QObject):
    def __init__(self, serial_worker: SerialPortWorker, parser: FrameParser):
        super().__init__()
        self.serial_worker = serial_worker
        self.parser = parser
        self.serial_worker.frameReceived.connect(self._on_frame)
```

- 监听第二个串口的被动数据
- 连接帧接收信号到处理槽

python

```
    def _on_frame(self, frame: bytes):
        hex_str = frame.hex()
        print("\n=== [被动监听-收到外部帧] ===")
        # ...解析和显示逻辑
        # 被动帧也更新"最近一帧"，但不写入 all_data_dict
        DATAS.write_last_frame("passive_rx", RxAddr, RxFuncID, RxDataLen, RxData, Mdbs_state)
```

- 处理被动接收的数据帧
- 只更新最近帧信息，不影响主动查询结果

10. 查询函数集合

python

```
def read_coils_0_13():          return (1, 1, 0, 14)
def read_coils_16_24():          return (1, 1, 16, 10)
def read_coils_27_31():          return (1, 1, 27, 5)
def read_holding_registers_0_13():  return (1, 3, 0, 13)
# ...
```

- 定义各种查询功能函数
- 返回元组：(从机地址, 功能码, 起始地址, 数量)
- 保持原有的分段读取逻辑

11. TrainGroupReaderApp类 - 应用装配

python

```
class TrainGroupReaderApp(QObject):
    def __init__(self, active_port: str, passive_port: str, baudrate: int = 9600):
        super().__init__()
        self.active_serial = SerialPortWorker(active_port, baudrate)
        self.passive_serial = SerialPortWorker(passive_port, baudrate)
```

- 应用主类，组装所有组件
- 创建主动和被动两个串口工作者

python

```
        functions = [
            read_coils_0_13,
            read_coils_16_24,
            # ... 其他查询函数
        ]
        self.scheduler = GroupQueryScheduler(
            self.client, self.parser, functions, cycle_interval_ms=5000, analyzer=self.analyzer
        )
```

- 组装查询功能列表
- 创建调度器，设置5秒循环间隔

python

```
        # 示例：订阅数据仓库的变化（UI 可以直接连这两个信号）
        DATAS.oneFuncUpdated.connect(self.on_func_data_updated)
        DATAS.snapshotUpdated.connect(self.on_snapshot_updated)
```

- 订阅数据仓库的变化信号

- 提供UI集成的钩子

12. 程序入口

```python
if __name__ == "__main__":
    app = QApplication(sys.argv)

    active_port = "COM5"  # 主动轮询口
    passive_port = "COM6"  # 被动监听口
    baudrate = 9600

    app_obj = TrainGroupReaderApp(active_port, passive_port, baudrate)
    app_obj.start()

    ret = app.exec_()
    app_obj.stop()
    sys.exit(ret)
```

- 创建PyQt应用实例
- 配置串口参数
- 启动应用并进入事件循环
- 程序退出时清理资源

总结

这个脚本实现了一个完整的Modbus集群读取系统，具有以下特点：

1. **线程安全**: 使用`SharedDataStore`保证多线程环境下的数据一致性
2. **模块化设计**: 各个组件职责明确，便于维护和扩展
3. **信号槽机制**: 利用PyQt的事件驱动模式，便于UI集成
4. **双串口支持**: 同时支持主动查询和被动监听
5. **定时调度**: 按固定间隔循环执行查询任务
6. **错误处理**: 包含超时和帧验证机制
7. **可扩展性**: 预留分析器接口供业务逻辑扩展