# 平台和策略交互

涉及的文件夹很多，做灾备任务主要涉及algo和strategy_public2_c,其他的接触到了再记录

algo这个文件夹主要涉及三类线程,根据流程依次为:

1. 平台消息处理线程：MsgDealThread.h
2. 方案处理线程：AlgoDealThread.h
3. 委托下单处理线程：AlgoOrder.h 还有个拆单线程，没怎么涉及，先不记录

## 平台消息处理线程

MsgDealThread.h

1. 消息结构体

```
struct TReqMsg
{
public:
    TReqMsg(void *lpData, int nLength)
    {
        // ...
        Buflen = nLength;
        pDateBuf = (void*)malloc(Buflen);
        // ...
    }

    ~TReqMsg()
    {
        // try-catch
    }
    //重载<<        =>  输出消息结构体的时候，只打印主题编号等部分内容即可
    friend std::ostream& operator<<(std::ostream &out, TReqMsg* p_obj);

    std::string sTopicName;         // 1.主题编号
    void *pDateBuf;                 // 2.包体数据，使用UnPack解包
    int Buflen;                     // 3. 包体长度
    EReqMsgType m_eMsgType;         // 4. 消息类型
    std::string sSchemeID;          //5.方案ID
    int iMsgNo;                     // 6. 消息序号用来做按序消费
    void* m_pTimeTask;              // 7.消息需要处理其他任务，定时任务对象
};
```

2. 线程类

都是加锁队列

```
class CSchemeCreateThread:public CThread
{
public:
    long Run();
    static const int WAIT_TIME = 10;
    int AddMsg(TReqMsg *msg);     // 日初重建用于方案删除的
private:
    CThreadMutex m_MsgMutex;
    std::list<TReqMsg*> m_MsgList;  //全局的消息队列
    std::map<std::string, int> m_schemeDeleteTime;  //方案删除等待时间
};

//消息处理线程，处理来自接入平台的各类消息
class CMsgDealThread:public CThread
{
    public:
        CMsgDealThread(const char *sName);   // m_schemeCreateThread = new
CSchemeCreateThread();
        virtual ~CMsgDealThread();
    public:
        long Run();
    private:
        std::list<TReqMsg*> m_MsgList;      //全局的消息队列
        CThreadMutex m_MsgListMutex;        //消息队列锁
    public:
        /***
        AlgoPublicInfo.h有如下定义
        extern CMsgDealThread *g_MsgDealThread_P2S; // 平台到 服务器的数据（订阅数
据）
        extern CMsgDealThread *g_MsgDealThread_T2S; // 策略到服务器的数据（策略返回
的数据）
         ***/
        // 将平台到服务器（ALGO）的消息加入m_MsgList（订阅数据，服务器怎么订阅平台的消
息）
        int AddMsg(TReqMsg *Msg);             // ALGO收到一个消息，将消息序号加1
        std::string m_sThreadName;
        int m_iMsgNo;                // 消息序号大于50w，重置为0  => 所以ALGO最多能处
理50w的并发消息?
        CEvent m_event_run;          //线程运行信号,CEvent看不到定义,
start(),wait(),stop()
        CSchemeCreateThread* m_schemeCreateThread;     // 初始化一个方案创建线程类
};
```

MsgDealThread.cpp

消息队列读取的写法都是一样的

```
long CMsgDealThread::Run()
{
    // 1. 取消息队列while
    while(!IsTerminated())
```

```
    {
        // 2.略初始化时加载策略参数，加载完成才能处理新建方案消息，
g_InitUserParamsIsFinished=1表示加载完成
        bool bContinue = false;
        {
            CAutoMutex _autolock(g_IsDealMsgLock);
            if(g_InitUserParamsIsFinished == 0)
            {
                bContinue = true;
            }
        }
        if (bContinue)
        {
            FBASE2::SleepX(10);
            continue;
        }
        // 3. 从队列中提取消息，为空超时等待
        if(m_MsgList.size() == 0)
        {
            if (EVENT_OK != m_event_run.Wait(100)) // 等待超时时间100ms
            {
                continue;
            }
        }
        // 4. 有的消息队列是取头部消息，这里是全部取
        std::list<TReqMsg*> temp_msg_list;
        {
            CAutoMutex _autolock(&m_MsgListMutex);
            m_MsgList.swap(temp_msg_list);
            m_event_run.Reset();
        }
        // 5. 消息处理
        std::list<TReqMsg*>::iterator iter = temp_msg_list.begin();
        for(; iter != temp_msg_list.end(); ++iter)
        {
            TReqMsg *lpMsg = *iter;
            // 消息处理
            // ...
            // 5.1 根据消息类型做不同处理
            if(lpMsg->sTopicName == MSGTYPE_ALGOJR_DAILY_INIT){
                // ...
                // 5.2 特殊消息（如新建方案）取负载最小的线程进行处理
                CAlgoDealThread *lpDealThread = g_AlgoDealThreadPool-
>GetMinDealThread();
                if(lpDealThread)
                {
                    // ...
                    CAlgoOrder *lpOrder = lpDealThread->CreateScheme(lpMsg-
>sSchemeID);    // 创建方案
                }
            }
            if(...){
                // ...
            }
```

```
            // 6，消息处理后，进行下一步操作
            {
                //这里使用全局方案索引的锁来保证方案不被删除。那么在方案删除的时候，必须对该锁进行加锁。
                CAutoMutex _autolock(g_pOrderIndexLock);
                CAlgoOrder *lpOrder = FindOrderBySchemeID_NoLock(lpMsg->sSchemeID);
                if(lpOrder)
                {
                    LogDebug("全局消息成功扔给方案" << lpMsg->sSchemeID);
                    // g_AlgoDealThreadPool->AddMsg(ReqMsg, m_iThreadNo);线程池的一个处理后续逻辑
                    // 放入新的消息队列m_ReqMsgList
                    lpOrder->AddMsg(lpMsg);     // 又加到m_MsgList了
                }
                // 6.1 删除消息
                delete lpMsg;
                continue;
            }
            // 7. 也可以执行一些定时任务
            uint64 timetmp = GetMill();
            if (timetmp - m_lLastDoRutineTime >= 10)
            {
                // ...
            }
        }
    }
}
```

# 方案处理线程

AlgoDealThread.h

1. 方案处理线程类

```
class CAlgoDispatchThread;

class CAlgoDealThread:public CThread
{
public:
    CAlgoDealThread(bool SingleThreadModel);
    ~CAlgoDealThread();

public:
    long Run();
    void ClearAllAlgoOrder();
public:
    std::vector<CAlgoOrder*> m_AlgoOrderList;          //方案队列  方案明细、子单等信息与处理
    CThreadMutex *m_pOrderListLock;                    //方案队列锁
```

```
    CAlgoDispatchThread *m_AlgoDispatchThread;          //拆单调度线程

    std::list<TReqMsg*> m_ReqMsgList;                    //消息队列    平台消息
    CThreadMutex m_pReqMsgListLock;                     //消息队列锁
    CEvent m_event_run;          //线程运行信号
    // 看下面的线程池是一个map，然后用线程号来索引
    // 为什么不用数组，可能是为了动态扩展线程的数量吧
    int m_iThreadNo;             // 当前线程编号
    uint64 m_lLastDoRutineTime; // 上次轮询处理时间，用于定时轮询，如每隔10s查一次消息
队列
public:
    int m_iSchemeCnt;           //方案个数，根据该值进行负载均衡，一个方案分配给一个线
程后，该值加1
public:
    CAlgoOrder* CreateScheme(std::string m_SchemeID);       //根据方案ID，创建方案
对象
    void DeleteScheme();                    // 删除方案
    // 有很多的AddMsg，如往Algo服务器、方案处理线程、子单处理线程添加消息
    int AddMsg(TReqMsg* ReqMsg);            // 往这个线程添加消息
};
```

2. 方案处理线程池

```
//方案处理线程池
class CAlgoDealThreadPool
{
public:
    CAlgoDealThreadPool(int ThreadCnt, bool SingleThreadModel = false);
    ~CAlgoDealThreadPool();
    void Start();
    void ClearAllAlgoOrder();
    void Stop();
public:
    std::map<int, CAlgoDealThread*> m_AlgoDealThreadList;       // 方案处理线程队列
    CThreadMutex *m_AlgoDealThreadListLock;                      // 方案处理线程队
列锁
private:
    int thread_cnt;          //线程个数
public:
    CAlgoDealThread* GetMinDealThread();      // 最小负载的线程
    int AddMsg(TReqMsg* ReqMsg, int iThreadNo);     // 处理消息
m_ReqMsgList.push_back(ReqMsg);
};
```

3. 拆单调度线程

```
class CAlgoDealThread;
class CAlgoDispatchThread:public CThread
{
```

```
public:
    CAlgoDispatchThread(CAlgoDealThread* DealThread);
    ~CAlgoDispatchThread();
public:
    long Run();
public:
    CAlgoDealThread *m_DealThread;          //绑定处理线程，使用它的方案队列和方案队
列锁
};
```

# 委托下单处理线程

AlgoOrder.h

方案处理线程在处理消息时，分支会调用到`CAlgoOrder::DealReqMsg()`进入子单处理逻辑

```
class CAlgoOrder
{
public:
    CAlgoOrder(std::string SchemeID);
    ~CAlgoOrder();
    //重载<<
    friend std::ostream& operator<<(std::ostream &out, CAlgoOrder* p_obj);
public:
    std::map<std::string, CAlgoOrderStock*> m_AlgoOrderStockList;       //方案明细
队列
    std::map<std::string, CAlgoOrderStock*> m_NewAlgoOrderStockList;        //新增
方案明细队列
    std::vector<std::string> m_DelSchemeinsCodeList;  //删除的方案明细号
    std::map<std::string, CAlgoSubOrder* > m_SubOrderList;          //子单队列


public:
    LPSCHEMEOBJ m_pBaseOrder;          //方案指针，由策略的创建方案接口返回。调用策略消
息接口时，需要传入该指针。
    // ...
};
```

# 方案灾备重建的流程

1. **平台:**
    - 发送消息`asset.algo.scheme_create`
2. **Alog**
    - `\src\s_ls_ls_algoserverflow\s_ls_ls_algo_serverflow.cpp`
    - `(CMsgDealThread)g_MsgDealThread_P2S->AddMsg(ReqMsg);` // 平台到服务器的数据（订阅数据）
    - 平台消息处理线程类：`CMsgDealThread::Run()` => `MsgDealThread.cpp`
        1. 从消息队列`m_MsgList`中取所有消息
        2. 获取线程中负载最小的线程

        - 当一个方案交给一个线程后，方案个数`m_iSchemeCnt`加1，以进行负载均衡
    3. 将方案交给最小负载线程，也就是加入这个线程的方案队列`m_AlgoOrderList`
    4. 将消息加入最小负载线程的消息队列`m_ReqMsgList`以进行处理
        - 如果消息队列为空，发送线程挂起信号 => `(CEvent)m_event_run.Wait()` => `CEvent`未声明?
    - tip: 该线程初始化了一个方案创建线程类成员变量：`CSchemeCreateThread()`
        - 用于方案等待删除
- 线程池中的方案处理线程： => `AlgoDealThread.cpp`
    1. 启动拆单线程`CAlgoDispatchThread::Run()`
    2. 处理`m_ReqMasgList`
        - => `lpOrder->DealReqMsg(lpMsg)` => `ALGOOrder.cpp`
        - => OrderInit(IF2UnPacker* lpUnPack, void *DateBuf, int Buflen) // 首先调用策略接口，把方案数据传给策略
        - => g_CallMainFun(m_pBaseOrder, FUN_ORDER_NEW, DateBuf, Buflen, sErrMsg) => CallMainFunc
        - => **策略**
            - pImpl->FrameworkCallMainFun(sFunType, pDataBuf, iBufLen, sOutParame);
                - MessageProc(const std::string & sFunType, IF2UnPacker * pUnPacker) // 普通消息
                    - OnSchemeInit
                    - GatherSchemeStocksInfo(IF2UnPacker* pUnPacker, std::string& sErrMsg)根据入参包获取成交量进行灾备恢复
                - MCMessageProc(sFunType, NULL) // MC消息，stp.api开头的
        - => 如果是方案重建：
            - CAlgoOrder::DoOrderRecover(); => `AlgoOrder.cpp`
            - `SynCallServer`同步调用平台接口 grpc
            - 获取子单信息后，`g_CallMainFun(m_pBaseOrder, FUN_INSERT_SUBORDER, DateBuf, Buflen, sErrMsg)`给策略发送消息`FUN_INSERT_SUBORDER`
                - 这个函数什么意思？ `g_CallMainFun = (pCallMainFun)g_TacticsLib->LoadFunc("CallMainFun");`
                - strategy_public2_c是所有策略的父类吗?
            - 策略 => MessageProc => OnEntrustInsert => RecoverEntrust => 在基类添加获取询价编号的函数进行字段恢复
    3. 定时扫描方案队列，隔10ms判断是否需要删除方案