Auth

主线程的逻辑

1. 获取参数

根据 main 启动的参数，得到 config file 和 wal。

【*】 wal 完全可以放到 config 中。

2. 订阅消息

订阅 mq 消息，有

Jsonrpc 发过来的 RequestNewTxBatch

Net 发过来的 Request

Consensus 发过来的 VerifyBlockReq

Chain 发过来的 BlockTxHashes

Executor 发过来的 BlackList 和 Miscellaneous

Snapshot 快照消息。

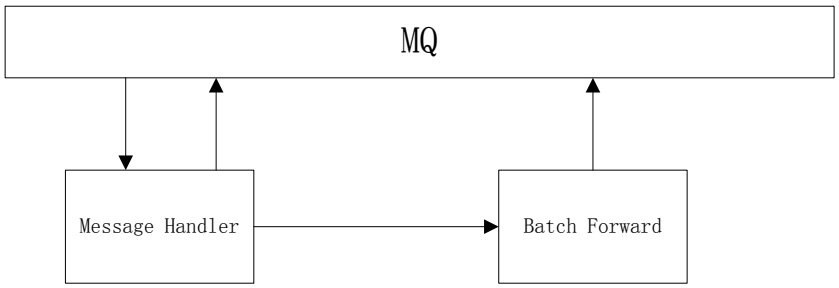
3. 搭建内部处理结构

启动一个线程 BatchForward，做批量转发

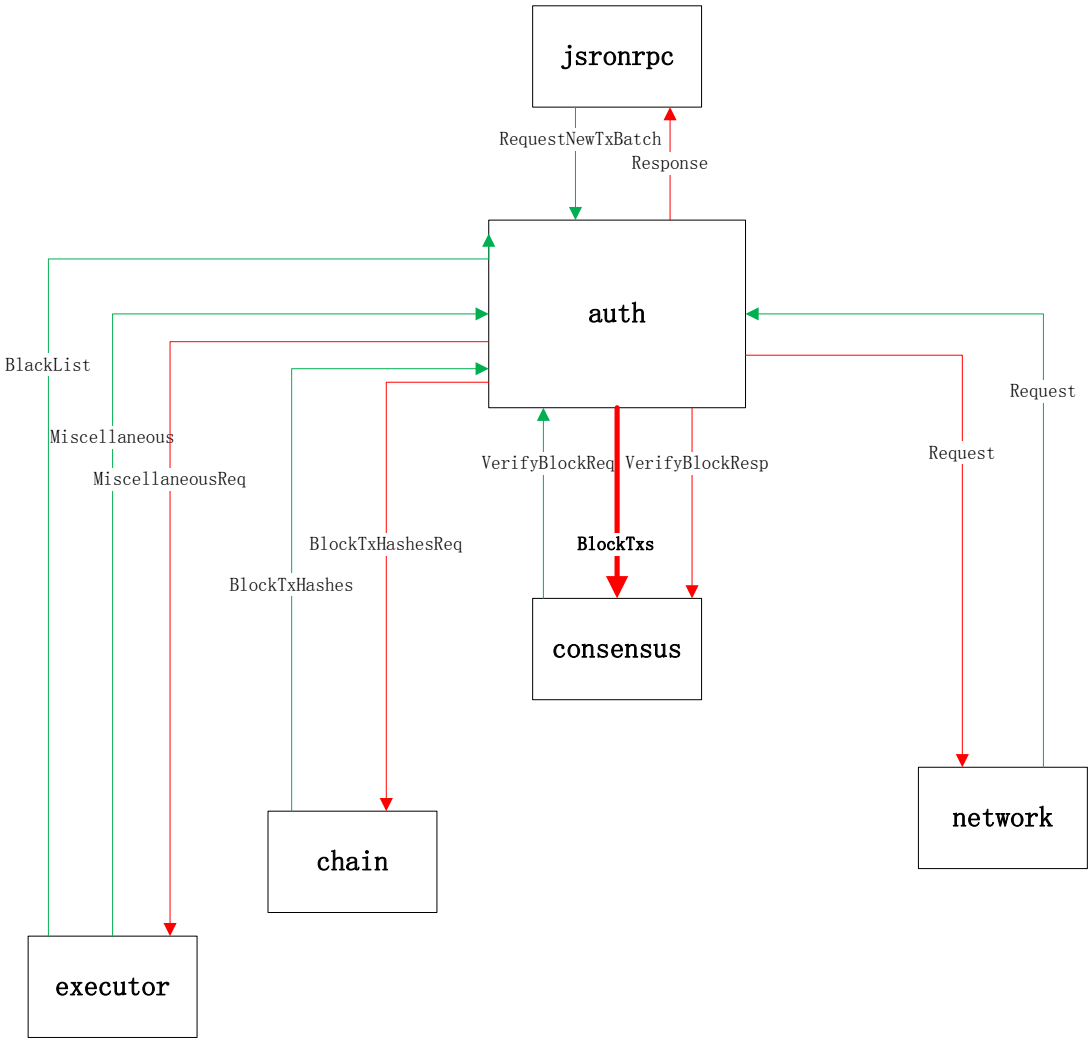
新建一个 dispatcher，用来管理交易池 tx pool

新建一个 Message Handler，用来分发消息。

下图是线程模型：



下图是 auth 模块的周边关系图：



文件结构

auth 的文件数比较少，总共有 6 个文件：

batch_forward.rs：批量转发交易

config.rs： 读取配置文件

dispatch.rs：管理 tx pool

handler.rs：消息分发处理

main.rs：主线程

txwal.rs：预写 tx

main.rs 上面已经介绍了，config.rs 是解析配置文件，下面来分析其他的文件。

1. txwal.rs

txwal 用来实现对 tx 的预写功能。收到 tx 之后写入文件，断电重启之后能够恢复出之前写进去的 tx。

具体的实现是使用数据库。

方法有：

new：新建一个数据库

regenerate：重建一个数据库

write：写入交易，key 是 tx 的 hash
delete：删除交易
delete_with_hash：使用 tx 的 hash（也就是 key）来删除交易
read：将数据库里面的所有交易都读出来，放进入参 Pool 中

2. dispatcher.rs

```
struct Dispatcher {
    txs_pool: RefCell<tx_pool:: Poll>,
    wal: TxWal,
    wal_enable: bool,
}
```

方法有：
new：新建一个 pool
add_tx_to_pool：
    将 SignedTransaction 加入 Pool 中，如果使能了 wal，就写入 wal 数据库
get_txs_from_pool：
    从 Pool 中获取 tx，入参是 height、block_gas_limit、account_gas_limit、check_quota
del_txs_from_pool_with_hash：
    将 tx 从 Pool 中删除，如果使能了 wal，起线程在 wal 数据库中删除
clear_txs_pool：
    清理 pool
txs_pool_len：
    pool 里面的交易数
proposal_tx_list：
    根据高度、block_gas_limit、account_gas_limit 从 Pool 中取出 tx，生成 BlockTxs 发送给共识
read_tx_from_wal：
    从 wal 数据库中取出 txs 并放入 txs_pool 中

3. batch_forward.rs

```
struct BatchForward {
    batch_size: usize,
    timeout: u64,
    check_duration: u32,
    last_timestamp: u64,
    request_buffer: Vec<Request>,
    rx_request: Receive<Request>,
    tx_pub: Sender<(String, Vec<u8>>,
}
```

这个结构体，request_buffer 是缓冲区，是核心，比较重要；batch_size 是缓存的大小；timeout、check_duration、last_timestamp 是超时处理机制，rx、tx 是通道相关。
有 3 个方法：
new：
    新建一个 BatchForward，把池子 request_buffer 建好

run：

循环处理，如果收到 Request，就放到 buffer 中，如果超出了 batch_size，就调用 batch_forward 进行转发；如果长期没有收到 Request，经过超时处理之后也调用 batch_forward 进行转发。

batch_forward：

从 buffer 中取出所有的 Request，发给 Net，然后清理 buffer，并更新 last_timestamp。

4. handler.rs

```
struct MsgHandler {
    rx_sub: Receiver<(String, Vec<u8>),
    tx_pub: Sender<(String, Vec<u8>),
    tx_request: Sender<Request>),

    cache_block_req: Option<VerifyBlockReq>,

    history_heights: HistoryHeights,
    history_hashes: HashMap<u64, HashSet<H256>>,

    chain_id: Option<u32>,

    cache: LruCache<H256, Option<Vec<u8>>,

    dispatcher: Dispatcher,
    tx_pool_limit: usize,

    check_quota: bool,
    block_gas_limit: u64,
    account_gas_limit: AccountGasLimit,

    is_snapshot: bool,

    black_list_cache: HashMap<Address, i8>,

    is_need_proposal_new_block: bool,
}
```

这个结构体包含了各个子功能的数据，非常繁杂，
有多个缓冲区：cache_block_req、history_xxx、cache、black_list_cache，
有多项配置：chain_id、tx_pool_limit、check_quota、block_gas_limit、account_gas_limit，
有多个标记：is_snapshot、is_need_proposal_new_block，
每个缓冲区分别对应哪个子功能，需要一一来拆悉。

下面看具体的处理流程：
auth 刚开始的时候是不 ready 的，需要由其他的消息经过 handle_remote_msg 来开始推动

auth 的运转。

刚开始会进入一个循环检测，这个是最开始的处理流程，也是超时时候的处理流程：
1). ready 的判断条件是看 chain_id 是否存在、history 数组是否初始化，否则分别向 Executor 发送 MiscellaneousReq 和向 chain 发送请求。其实系统刚启动时，history 数据是需要 chain 主动发出的，因为 auth 里面的 history 最大值和最小值都是 0，无法发出请求。

2). 会检测 cache_block_req 里是否有未处理的数据，这条消息是 consensus 发送过来的。

3). 会检测是否需要出新块，如果需要就从交易池 TxPool 中出一个新块。

因为 consensus 发过来的交易验证请求和 chain 发过来 tx hashes 后准备出块这 2 项任务比较重要，所以在超时处理中添加了这 2 条。

然后是消息处理：
1) Chain >> BlockTxHashes
* 更新 check_quota、block_gas_limit、account_gas_limit
* 更新 history_height 和 history_hashes
* 开始准备出块
* 从 Pool 中将对于 hash 的 tx 删除

2) Executor >> BlackList
更新 black_list_cache。
黑名单里有 black_list 和 clear_list，如果是 clear_list，在 cache 中将其删除；如果是 black_list，如果之前没有，cache 中的值设置为 3，如果有，就减 1.
black_list_cache 在 verify_black_list 中检查，verify_black_list 在 deal_request 中处理，是从 jsonrpc 或者 net 过来的 request。

3) Consensus >> VerifyBlockReq
处理共识模块发过来的交易验证，没有交易的时候不会要求验证
这些细节目前我还看不明白，等以后再补。

4) Jsonrpc >> RequestNexTxBatch 或者 Net >> Request
处理新的交易
这里面区分是从 Jsonrpc 发过来的，还是从 Net 发过来的。如果是从 Jsonrpc 发过来的，需要进行响应，不管成功失败。
还区分了是批量交易还是单项交易。
将交易经过验证，最终转化成 SignedTransaction，存入 Pool 中。
后续细化

5) Executor >> Miscellaneous
从 Executor 获取 chain_id。
在超时处理中也会检查是否有 chain_id，如果没有，就向 Executor 发送 MiscellaneousReq