# Net\_plugin

主要功能是同步区块和trx，不光从peer同步到自己，还得考虑从自己同步给peer。

## 重要数据结构

unique\_ptr<tcp::acceptor> acceptor;

tcp::endpoint listen\_endpoint; 从p2p-listen-endpoint中得到

string p2p\_address; 也来自p2p-listen-endpoint

vector<string> supplied\_peers; 来自p2p-peer-address

possible\_connections allowed\_connections;

是个enum，

None = 0,

Producers = 1 << 0,

Specified = 1 << 1,

Any = 1 << 2

来自allowed-connection，如果是specified，则必须有peer-key。并用其初始化allowed\_peers。

vector<chain::public\_key\_type> allowed\_peers; ///< peer keys allowed to connect，配合allowed\_connections，用来制定被允许连接的peer的pub key。

std::map<chain::public\_key\_type,

chain::private\_key\_type> private\_keys; 来自peer-private-key，给allowed\_connection授权？

std::set< connection\_ptr > connections; 当前的连接

unique\_ptr< sync\_manager > sync\_master; 子类，主要功能

unique\_ptr< dispatch\_manager > dispatcher; 子类，主要功能

int started\_sessions = 0;

node\_transaction\_index local\_txns;

shared\_ptr<tcp::resolver> resolver;来自app().get\_io\_service()

一堆timer和时长设置

## plugin\_initialize和plugin\_startup

plugin\_initialize中对内部数据做各种初始化，这些数据一般都来自config.ini

构造出sync\_master，dispatcher这两个主要的成员。

plugin\_startup中：

my->acceptor->bind(my->listen\_endpoint);

my->acceptor->listen();

my->start\_listen\_loop();

my->start\_monitors();

for( auto seed\_node : my->supplied\_peers ) {

connect( seed\_node );

}

### start\_listen\_loop

acceptor->async\_accept来异步监听收到的信息，对收到的信息做处理。最终会调用到connection-> process\_next\_message，这里会根据msg的内容，找出msg\_handler，调用该handler，是所有信息处理的来源。

### start\_monitors

有connector\_check和transaction\_check两个steady\_timer

connector\_check通过定时器，定时检查connection是否有错，并重连。

Transaction\_check是启动定时器，定期清理local\_txns里的trx，清理有两个目标：

1. 时间上过期的清理掉
2. 已经进lib的清理掉。

同时会在所有connection上执行同样的清理，在connection上清理时会同时清理已经进lib的block。

### 对supplied\_peers执行connect

针对每个supplied\_peers，创建connection对象，并connect

它将最终导致针对connection，调用->start\_session->send\_handshake

然后会在connection-> process\_next\_message里得到返回值，根据msg类型，调用对应的net\_plugin\_impl->handle\_message。然后开始各种后续的处理。

### Connection类

针对每一个p2p连接，都有个connection类保存其信息。

其数据结构包括：

peer\_block\_state\_index blk\_state;

transaction\_state\_index trx\_state;

optional<sync\_state> peer\_requested; // this peer is requesting info from us

socket\_ptr socket;

fc::sha256 node\_id;

handshake\_message last\_handshake\_recv;

handshake\_message last\_handshake\_sent;

int16\_t sent\_handshake\_count = 0;

bool connecting = false;

bool syncing = false;

uint16\_t protocol\_version = 0;

string peer\_addr;

unique\_ptr<boost::asio::steady\_timer> response\_expected;

optional<request\_message> pending\_fetch;

go\_away\_reason no\_retry = no\_reason;

block\_id\_type fork\_head;

uint32\_t fork\_head\_num = 0;

optional<request\_message> last\_req;

还有些发送、接收的buffer等。

其中一些multi-index的结构：

struct peer\_block\_state {

block\_id\_type id;

uint32\_t block\_num;

bool is\_known;

bool is\_noticed;

time\_point requested\_time;

};记录了peer的block情况，里面都记录哪些block？我们会定期清除这个表吗？

struct transaction\_state {

transaction\_id\_type id;

uint32\_t block\_num = 0; ///< the block number the transaction was included in

time\_point\_sec expires;

};peer的trx的情况，怎样的trx才会被记录下来呢？如何更新？

### 从零开始同步过程分析

先会发送handshake给peers。Handshake的内容如下：

struct handshake\_message {

uint16\_t network\_version = 0; ///< incremental value above a computed base

chain\_id\_type chain\_id; ///< used to identify chain

fc::sha256 node\_id; ///< used to identify peers and prevent self-connect

chain::public\_key\_type key; ///< authentication key; may be a producer or peer key, or empty

tstamp time;

fc::sha256 token; ///< digest of time to prove we own the private key of the key above

chain::signature\_type sig; ///< signature for the digest

string p2p\_address;

uint32\_t last\_irreversible\_block\_num = 0;

block\_id\_type last\_irreversible\_block\_id;

uint32\_t head\_num = 0;

block\_id\_type head\_id;

string os;

string agent;

int16\_t generation; 针对当前connection发送的handshake的次数。

};

从对方收到回应后，会回调到net\_plugin\_impl.handle\_message(handshake…)的函数。如果是第一次跟对方握手，会判断如chain\_id，protocol\_version等是否一致，不一致就断开（发送go\_away\_message）。一致的话，会检查谁的lib更新，如果自己的更新，且对方的在fork上（对方的lib跟本地lib所在block\_id不一致），向对方发送go\_away\_message( forked )并退出。

如果一切正常，调用sync\_master->recv\_handshake(c,msg);

sync\_manager是sync的主要实现类，其数据结构：

class sync\_manager {

uint32\_t sync\_known\_lib\_num; 已经知道的lib\_num

uint32\_t sync\_last\_requested\_num;

uint32\_t sync\_next\_expected\_num;

uint32\_t sync\_req\_span; 一次同步几个block，在config.ini的sync-fetch-span里。主网同步的有用2000的。

connection\_ptr source; 从哪个connection同步

stages state; enum，有lib\_catchup,head\_catchup,in\_sync三个值。

}

在sync\_master->recv\_handshake(c,msg)里：

reset\_lib\_num(c);

sync检查，有5种情况：

1. 本地head\_block\_id==peer head block id,我们都已经同步到最新
2. 本地head\_block\_num<peer lib，我们没同步完，调用start\_sync发起同步。
3. 本地 lib > peer head num, 对方不同步，如果generation!=1, 对其发last\_irr\_catch\_up notice
4. 本地 head number<= peer head num,更新sync state并发起catchup请求
5. 本地 head number > peer head num，如果generation!=1,对其发catchup notice.

Start\_sync：经过一番检查和对要sync的block的设置后，调用source. request\_sync\_blocks发起请求。请求中会传入本次要sync的起点，终点（数量由config.ini设置，如2000）.如果该请求超时，会重新发起。

Peer收到该请求后的处理，在net\_plugin\_impl.handle\_message(..sync\_request\_message…)中：

调用connection. enqueue\_sync\_block。从自己的chain\_plugin中取出该signed\_block，发送给对方。只发了一个block。

本地接收到该消息后，调用net\_plugin\_impl.handle\_message(…signed\_block\_ptr…)处理：

判断本地是否已经有该block\_id，如果已有，调用sync\_master->recv\_block进行下一block的同步。注意sync\_master的同步有两个阶段，lib\_catchup和head\_catchup，当判断lib\_catchup已经完成的时候，会重新向所有已知的connection发送handshake，在新的handshake的处理中，可以得到更新的lib,或开始head\_catchup，或判断完成了同步。

如果本地没有该block\_id，执行操作：

dispatcher->recv\_block

在dispatcher. received\_blocks里insert该block，

connection->add\_peer\_block

chain\_plug->accept\_block(msg) //以下操作详见producer\_plugin和chain\_plugin

->producer\_plugin. on\_incoming\_block

auto bsf = chain.create\_block\_state\_future( block );

chain.abort\_block();//abort current pending block

chain.push\_block( bsf );

emit( self.pre\_accepted\_block, b );//发送signal

fork\_db.add( new\_header\_state, false );

emit( self.accepted\_block\_header, new\_header\_state );

maybe\_switch\_forks( s );

->apply\_block

针对该block中的每一笔trx：

1. 检查本地的local\_txns中是否有，有的话，将该block的id设进去。（说明该trx其他节点也收到了，且执行了，包含在该block中）
2. 检查该connection的trx\_state，如果能找到该trx，将block id设进去。

sync\_master->recv\_block

block的同步就是这样了。

### Trx的同步

Trx可能来自两个地方可以参考《一个trx的生命周期》：

1. 本地执行，如cleos，此时先从dispatcher-> bcast\_transaction开始。
2. 从其他node接受到此trx，此时从net\_plugin->handle\_message(packed\_transaction\_ptr)开始。

两种方式大体上是类似的，即先在本地chain\_plugin.db里集成该trx，如果成功，就发送给net\_plugin，然后由net\_plugin广播给所有peer。每个trx只广播一次。

主要的数据结构是net\_plugin-> local\_txns。它保存所有本地的尚未进lib的,已经在本地db里的trx。

收到上面两个地方来的trx时就push进该local\_trxs。

当收到block时，如果该block里包含该trx，就将该trx的block\_id更新进去

通过transaction\_check这个timer，定时清理超时的timer。有两种清理：

1. 该trx设置的超时时间超时了。
2. 该trx所在的block进入lib