# Chain\_plugin

代码基于v1.6.0-rc2

## 主要结构

Namespace chainApi{

Class read\_only;

Class read\_write;

}

Class chain\_plugin.

## Read\_only

主要实现nodeos对外的一些可读信息接口，包括各种结构定义和接口。如：

Get\_info

Get\_account

Get\_code

Get\_abi

Abi\_json\_to\_bin

Abi\_bin\_to\_json

Get\_required\_keys

Get\_transaction\_id

Get\_block

Get\_table\_rows

Get\_producers

主要数据：

**const controller& db;**

## read\_write

写链的接口，包括结构定义和接口：

Push\_block

Push\_transaction

Push\_transactions

主要数据：

**controller& db;**

## chain\_plugin

继承自application的子类plugin

接口包括实现plugin定义的通用接口：

void plugin\_initialize(const variables\_map& options);

void plugin\_startup();

void plugin\_shutdown();

set\_program\_options

和一些自己的：

get\_read\_only\_api

get\_read\_write\_api

accept\_block

accept\_transaction

bool block\_is\_on\_preferred\_chain(const chain::block\_id\_type& block\_id);

recover\_reversible\_blocks

import\_reversible\_blocks

export\_reversible\_blocks

get\_chain\_id

该类的实体是chain\_plugin\_impl，也是个class，主要的数据结构：

bfs::path blocks\_dir;

bool readonly = false;

flat\_map<uint32\_t,block\_id\_type> loaded\_checkpoints;

fc::optional<fork\_database> fork\_db;

fc::optional<block\_log> block\_logger;

fc::optional<controller::config> chain\_config;

**fc::optional<controller> chain;**

fc::optional<chain\_id\_type> chain\_id;

//txn\_msg\_rate\_limits rate\_limits;

**fc::optional<vm\_type> wasm\_runtime;**

fc::microseconds abi\_serializer\_max\_time\_ms;

fc::optional<bfs::path> snapshot\_path;

以及一堆的channel，methods，provider\_handlers,connections。初始化时就会通过app->get\_channel和app->get\_method把channel和method初始化好。

#### plugin\_initialize

设定chain\_config，初始值是从controller读的，然后把调用plugin\_initialize时传入的参数加载进来，如黑白名单，trusted\_bp。

如果输入参数有check\_point，则读出，加载在loaded\_checkpoints

读出输入参数中的其他参数，初始化自己的本地数据。包括wasm-runtime。

如果输入参数有export-reversible-blocks，会把reversible-blocks export出来放在blocks\_dir/ reversible下。

如果输入参数有delete-all-blocks，会把state\_dir和blocks\_dir下的内容删掉

如果输入参数有hard-replay-blockchain，会执行如下操作：

清空state\_dir

调用block\_log->repair\_log，把区块一个个重新过滤一遍，看是否有错误的。这一过程可以通过在输入参数中加“truncate-at-block”指定查到哪个block为止。

如果backup\_dir里有reversible文件夹，或者输入参数有fix-reversible-blocks，则recover\_reversible\_blocks，会建立new\_reversible数据库并将数据读入。

通过option字段判断是read-mode,还是validation-mode

初始化一堆provider，connection

**my->chain.emplace( \*my->chain\_config );**

my->chain\_id.emplace( my->chain->get\_chain\_id());

my->chain->add\_indices();

其中my->chain.emplace( \*my->chain\_config );需要特别注意。

my->chain就是controller，是这里的核心数据，这里会将其初始化后加到chain\_plugin，为了不显得支离破碎，把它放在后面分析。

#### plugin\_startup

调用my->chain->startup，还是controller的处理

#### plugin\_shutdown

各种资源释放，以及my->chain.reset();

#### accept\_block

通过调用一个method完成，这个method的实现在producer\_plugin的：

void on\_incoming\_block(const signed\_block\_ptr& block)

具体实现后续再分析。

#### Accept\_transaction

也是method，实现定义在producer\_plugin的：

void on\_incoming\_transaction\_async，以后再分析

#### 其他接口

不难

## Readonly 和read\_write

基本都是从controller读来做基本加工后返回的，所以重点在controller。

## Controller

Chain\_plugin的功能基本都是通过controller实现的，因此需要详细研究它。

了解其数据结构是理解其运行原理的基础，所以下面很多地方是直接介绍其重点数据结构的，为了简明，很多只列出核心数据。

#### 数据库读取模式

先了解数据库的几个读取模式：  
官方解释：<https://developers.eos.io/eosio-nodeos/docs/read-modes>  
enum class db\_read\_mode {

SPECULATIVE,

HEAD,

READ\_ONLY,

IRREVERSIBLE

};

* SPECULATIVE  
  如cleos或rpc节点，会在数据库的head block上叠加本节点上执行，但尚未被提交到链上的trx。因此它最新，但是不可靠。例如之前发现在某节点上push\_trx返回的返回值的block\_id，跟后续在区块浏览器上用trx\_id查到的block\_id，经常差1-2个block，就是这个原因造成的。
* HEAD  
  cleos或rpc节点的head\_block，当前的head\_block并不是不可逆的，而且有微分叉的可能，所以这种模式不稳定，但是比SPECULATIVE好一些。
* READ\_ONLY  
  节点是从chain上同步来的，不包括本地的未上链交易。
* IRREVERSIBLE

顾名思义，很好理解

#### 验证模式

enum class validation\_mode {

FULL,

LIGHT

};  
含义：还不清楚

#### Config

包含很多可配置项。

各种黑白名单，各种目录，各种size，一些开关，genesis，wasm\_runtime，resource\_greylist，trusted\_producers

#### Block status

enum class block\_status {

irreversible = 0, ///已经不可逆< this block has already been applied before by this node and is considered irreversible

validated = 1, ///已经被应用在本节点< this is a complete block signed by a valid producer and has been previously applied by this node and therefore validated but it is not yet irreversible

complete = 2, ///已经完成，但是还未被应用在本节点，这种情况，这个block是否是本节点产生的呢?< this is a complete block signed by a valid producer but is not yet irreversible nor has it yet been applied by this node

incomplete = 3, ///尚未完成。肯定是本节点产生的。< this is an incomplete block (either being produced by a producer or speculatively produced by a node)

};

#### Indices

using controller\_index\_set = index\_set<

account\_index,

account\_sequence\_index,

global\_property\_multi\_index,

dynamic\_global\_property\_multi\_index,

block\_summary\_multi\_index,

transaction\_multi\_index,

generated\_transaction\_multi\_index,

table\_id\_multi\_index

>;

#### 描述block的

有数据和state两种数据

struct **block\_state** : public **block\_header\_state** {

**signed\_block\_ptr block;**

bool validated = false;

bool in\_current\_chain = false;

/// this data is redundant with the data stored in block, but facilitates

/// recapturing transactions when we pop a block

vector<transaction\_metadata\_ptr> trxs;

}

struct **signed\_block** : public signed\_block\_header{

vector<transaction\_receipt> transactions; /// new or generated transactions

extensions\_type block\_extensions;

}

/\*\*

\* Extentions are prefixed with type and are a buffer that can be

\* interpreted by code that is aware and ignored by unaware code.

我理解可以认为是个int->string的map

\*/

typedef vector<std::pair<uint16\_t,vector<char>>> extensions\_type;

#### 描述transaction的

struct **transaction\_receipt** : public transaction\_receipt\_header {

fc::static\_variant<transaction\_id\_type, packed\_transaction> trx;

}

其中transaction\_id\_type是sha256.

struct **packed\_transaction** {

vector<signature\_type> signatures;

fc::enum\_type<uint8\_t,compression\_type> compression;

bytes packed\_context\_free\_data;

bytes packed\_trx;

private:

// cache unpacked trx, for thread safety do not modify after construction

signed\_transaction unpacked\_trx;

}

不清楚unpacked\_trx存的是什么，为什么存在这里。

Compression表示trx的压缩方式，有none和zlib两种。

struct **signed\_transaction** : public transaction{

vector<signature\_type> signatures;

vector<bytes> context\_free\_data; ///< for each context-free action, there is an entry here

}

struct **transaction** : public transaction\_header {

vector<action> context\_free\_actions;

vector<action> actions;

extensions\_type transaction\_extensions;

transaction\_id\_type id()const;

}

struct **transaction\_header** {

time\_point\_sec expiration; ///< the time at which a transaction expires

uint16\_t ref\_block\_num = 0U; ///< specifies a block num in the last 2^16 blocks.

uint32\_t ref\_block\_prefix = 0UL; ///< specifies the lower 32 bits of the blockid at get\_ref\_blocknum

fc::unsigned\_int max\_net\_usage\_words = 0UL; /// upper limit on total network bandwidth (in 8 byte words) billed for this transaction

uint8\_t max\_cpu\_usage\_ms = 0; /// upper limit on the total CPU time billed for this transaction

fc::unsigned\_int delay\_sec = 0UL; /// number of seconds to delay this transaction for during which it may be canceled.

}

transaction\_header的长度是固定的，为了方便在解析的时候不用动态分配内存。

每个trx都有个超时时间expiration，当一个block的block\_header::timestamp比expiration大，且该block变为不可逆，那就可以认为该trx永远不会上链了。

class **transaction\_metadata** {

public:

transaction\_id\_type id;

transaction\_id\_type signed\_id;

packed\_transaction\_ptr packed\_trx;

fc::microseconds sig\_cpu\_usage;

optional<pair<chain\_id\_type, flat\_set<public\_key\_type>>> signing\_keys;

std::future<std::tuple<chain\_id\_type, fc::microseconds, flat\_set<public\_key\_type>>>

signing\_keys\_future;

bool accepted = false;

bool implicit = false;

bool scheduled = false;

}

这个类用来存储一个transaction的context-free cached data，如packed/unpacked/compressed and recovered keys

struct **action** {

account\_name account;

action\_name name;

vector<permission\_level> authorization;

bytes data;

}

struct **permission\_level** {

account\_name actor;

permission\_name permission;

};

#### 描述block\_header的

struct **block\_header\_state** {

block\_id\_type id;

uint32\_t block\_num = 0;

signed\_block\_header header;

uint32\_t dpos\_proposed\_irreversible\_blocknum = 0;

uint32\_t dpos\_irreversible\_blocknum = 0;

uint32\_t bft\_irreversible\_blocknum = 0;

uint32\_t pending\_schedule\_lib\_num = 0; /// last irr block num

digest\_type pending\_schedule\_hash;

producer\_schedule\_type pending\_schedule;

producer\_schedule\_type active\_schedule;

incremental\_merkle blockroot\_merkle;

flat\_map<account\_name,uint32\_t> producer\_to\_last\_produced;

flat\_map<account\_name,uint32\_t> producer\_to\_last\_implied\_irb;

public\_key\_type block\_signing\_key;

vector<uint8\_t> confirm\_count;

vector<header\_confirmation> confirmations;

}

这里看到有dpos\_proposed\_irreversible\_blocknum，dpos\_irreversible\_blocknum，bft\_irreversible\_blocknum，原来内部有3个irreversible标准，后续还需详细了解。

struct **signed\_block\_header** : public block\_header

{

signature\_type producer\_signature;

};

struct block\_header

{

block\_timestamp\_type timestamp;

account\_name producer;

uint16\_t confirmed = 1; //当一个producer给一个block confirm的时候，这个producer其实是给所有[block\_num()-confirm, block\_num())的一系列block confirm。它不能给这个阶段的其他block confirm。以防冲突和作恶。

一个producer只能给之前的block确认，不能给当前的确认。因为它自己生产的自己默认就确认过了。

block\_id\_type previous;

checksum256\_type transaction\_mroot;

checksum256\_type action\_mroot;

uint32\_t schedule\_version = 0;

optional<producer\_schedule\_type> new\_producers;

extensions\_type header\_extensions;

}

struct header\_confirmation {

block\_id\_type block\_id;

account\_name producer;

signature\_type producer\_signature;

};

#### pending\_state

struct pending\_state {

maybe\_session \_db\_session;

block\_state\_ptr \_pending\_block\_state;

vector<action\_receipt> \_actions;

controller::block\_status \_block\_status = controller::block\_status::incomplete;

optional<block\_id\_type> \_producer\_block\_id;

};

#### controller\_impl

一个模块，其行为可以认为主要包括两部分：（查询类不变化，不考虑）

1. 通过外界的输入，改变自身状态。也就是改变内部数据的状态。
2. 对外发出所需的通知和调用等，触发其他模块的状态变化。

其实也就是自身的状态变化和让其他模块状态变化。因此理解其自身包含变量的含义是至关重要的。下面的是controller\_impl的主要数据。

* chainbase::database **db**;  
  数据库类型，它维护很多表格。

具体的表格数据定义后面详细分析。一大坨。

* chainbase::database reversible\_blocks;  
  数据库，保存已经applied但是还没不可逆块的数据。
* block\_log blog;  
  有两个文件，一个里面存的是每个block的header及其在block文件里的位置，是append only的，只存不可逆区块的。另一个只存block头在文件里的位置。通过这个结构可以很快的随机读取block文件里的各个block。  
  其中读取实际是调用fc::raw::unpack完成的
* optional<pending\_state> pending;  
  详见下面**apply\_block**部分，是把一个block应用到本地时的临时变量，重要。
* block\_state\_ptr head;  
  当前的head of blocks
* fork\_database fork\_db;  
  新block收到后，会被加入fork\_db，它保存着所有潜在的未确认块的light-weight state . 会track最长链和最新的不可逆块，所有比最新的不可逆块老的，都会被释放。
* wasm\_interface wasmif;

虚拟机相关

* resource\_limits\_manager resource\_limits;  
  其内部也是个数据库，保存着用户可用的资源。在运行中随时需要计算调整用户资源。且其保存指向controller实体的指针
* authorization\_manager authorization;  
  鉴权相关，其内部也有个数据库。也保存着指向controller的指针
* controller::config conf;  
  配置相关
* chain\_id\_type chain\_id;
* bool replaying  
  当前是否在replay
* db\_read\_mode read\_mode  
  数据库读取模式
* map<digest\_type, transaction\_metadata\_ptr> unapplied\_transactions;  
  Transactions that were undone by pop\_block or abort\_block,当要新创建blocks时，可以从这个map里查询出可以加入的trx。

数据库表格

所有的数据在\libraries\chain\include\eosio\chain\types.h的枚举object\_type里，大概几十种，不一一列举了。这里面的不都是chain\_plugin的，应该各个plugin用到的都在里面了。主要的如下：

account\_object\_type,

permission\_object\_type,

key\_value\_object\_type,

global\_property\_object\_type,

block\_summary\_object\_type,

transaction\_object\_type,

producer\_object\_type,

account\_control\_history\_object\_type, ///< Defined by history\_plugin

public\_key\_history\_object\_type, ///< Defined by history\_plugin

table\_id\_object\_type,

resource\_limits\_object\_type,

resource\_usage\_object\_type,

account\_history\_object\_type, ///< Defined by history\_plugin

action\_history\_object\_type, ///< Defined by history\_plugin

reversible\_block\_object\_type,

**block\_summary\_object**: 用于追踪TaPOS，是做轻量级验证用的，当验证一个trx是否valid时，我们需要用一个块的块头来判断，详细方法见白皮书。结构：

id\_type id;

block\_id\_type block\_id;

**global\_property\_object**：保存global state information，(BP们, current fees)，结构：

id\_type id;

optional<block\_num\_type> proposed\_schedule\_block\_num;

shared\_producer\_schedule\_type proposed\_schedule;（里面有所有producers）

chain\_config configuration;

**account\_object**：账号信息，所以db应该是有存储所有账号的。数据结构：

id\_type id;

account\_name name;

uint8\_t vm\_type = 0;

uint8\_t vm\_version = 0;

bool privileged = false;

time\_point last\_code\_update;

digest\_type code\_version;

block\_timestamp\_type creation\_date;

shared\_blob code;

shared\_blob abi;

**permission\_object**, permission信息，结构：

id\_type id;

permission\_usage\_object::id\_type usage\_id;

id\_type parent; ///< parent permission

account\_name owner; ///< the account this permission belongs to

permission\_name name; ///< human-readable name for the permission

time\_point last\_updated; ///< the last time this authority was updated

shared\_authority auth; ///< authority required to execute this permission

**transaction\_object**, 主要用于重复trx检测，当一个trx被block接受时，就会添加一个transaction\_object, 在block处理结束时，超时的trx会被remove。结构：

id\_type id;

time\_point\_sec expiration;

transaction\_id\_type trx\_id;

**producer\_object**, 记录producer信息。

id\_type id;

account\_name owner;

uint64\_t last\_aslot = 0;

public\_key\_type signing\_key;

int64\_t total\_missed = 0;

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

chain\_config configuration; /// The blockchain configuration values this producer recommends

**table\_id\_object**，记录(scope, code, table)的map

id\_type id;

account\_name code;

scope\_name scope;

table\_name table;

account\_name payer;

uint32\_t count = 0; /// the number of elements in the table

**reversible\_block\_object**

id\_type id;

uint32\_t blocknum = 0;

shared\_string packedblock;

构造函数

初始化db，reversible\_blocks，这两个数据库创建时需要指明是只读还是读写，还得指明size。

初始化blog,fork\_db，虚拟机，resource\_limit, authorization,等。

其他一些参数也从config里传入。

且会设置eosio的一些action的handler.如newaccount,setcode,setabi,等等

fork\_db.irreversible.connect( [&]( auto b ) {

on\_irreversible(b);

});

Startup

void controller::startup( std::function<bool()> shutdown, const snapshot\_reader\_ptr& snapshot ) {

my->head = my->fork\_db.head();

my->init(shutdown, snapshot);

}

其中snapshot\_reader\_ptr是在nodeos启动时传入的参数里读取“snapshot”参数，这是个文件夹路径，如果存在的话，就会创建对应的snapshot\_reader\_ptr传进来。

If(snapshot){

从snapshot中读出数据，初始化各个数据结构，包括fork\_db，db，多个index等等。这部分作为支线后续再专门研究。

}else{

If(!head){ //head是block\_state\_ptr类型。

initialize\_fork\_db(){

这是最初状态，链上什么都没有。需要从genesis建立起链。会从config中读出genesis文件。从中读出或生成各种genesis block所需的信息，如chain\_id，valid\_producer,pub\_key，block\_id,block\_num等，

利用如上信息生成head，即genesis\_block的head，并

**fork\_db.set( head );**

**db.set\_revision( head->block\_num );**

**initialize\_database();**

其中initialize\_database做了很多事：

db.create<多种object>

authorization和resource\_limits创建数据库，初始化。

创建eosio账户，此操作调用create\_native\_account，向db里create一个account\_object并填入其permission,resource等参数，且特殊的针对eosio账户，需要执行set\_abi。还需处理对应的resource\_limits等。

创建null,prods账户

}

auto end = blog.read\_head();

if( !end ) {//没有，说明无区块

blog.reset( conf.genesis, head->block );

} else if( end->block\_num() > head->block\_num ) {//已存的不可逆区块>当前head，说明当前的status状态未赶上不可逆的head，需要replay，同步区块时经常这样。

replay( shutdown ); //replay的逻辑见下面Replay，挺重要和典型。

report\_integrity\_hash = true;

}

}

if( shutdown() ) return;

对db和reversible\_blocks等数据库根据head等做各种检查，如果不匹配就要处理。

目标是所有的数据库和head，blog都一致。

Startup就完成了。

**Replay**：

Replay先从head到blog->head。即从当前数据库状态到所有不可逆块。

通过blog.read\_block\_by\_num( head->block\_num + 1 )循环读出每一个block，将其通过replay\_push\_block插入到当前数据库中。

然后再把reversible\_blocks里的block都读出来，通过replay\_push\_block插入到数据库。针对这两种，在执行replay\_push\_block时输入的state不同，一个是irreversible，一个是validated。

replay\_push\_block的关键步骤：  
{

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

auto new\_header\_state = fork\_db.add( b, skip\_validate\_signee );

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

if ( read\_mode != db\_read\_mode::IRREVERSIBLE ) {

maybe\_switch\_forks( s );

}

// on replay irreversible is not emitted by fork database, so emit it explicitly here

if( s == controller::block\_status::irreversible )

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

}

其中emit是往channel里发数据。

1. emit( self.pre\_accepted\_block, b );

函数定义在chain\_plugin.plugin\_initialize里的my->pre\_accepted\_block\_connection。

实际是把信息继续传：my->pre\_accepted\_block\_channel.publish(blk);

这个会最终调用producer\_plugin的on\_incoming\_block。后续在producer\_plugin再研究。

这个动作是把replay的block发给producer\_plugin，对其产生影响。

1. auto new\_header\_state = fork\_db.add( b, skip\_validate\_signee );

插入fork\_db的index，（如果fork\_db中有需要清除的block，就清除，这跟当前论题无关）

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

这会调用到bnet\_plugin的on\_accepted\_block\_header

1. maybe\_switch\_forks( s );

对于read\_mode不是不可逆的，都要执行这一句。

这个操作很复杂，下面详细描述。

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

会调用到bnet\_plugin的on\_irreversible\_block

Replay的总结：

操作对象是从当前head到blog的head，即当前到所有的不可逆块。操作过程是从blog中循环读出所有的block，然后更新到当前数据库中，这个过程会更新fork\_db，调用maybe\_switch\_forks,且向producer\_plugin发送on\_incoming\_block，向bnet\_plugin发送on\_accepted\_block\_header和on\_irreversible\_block。

**maybe\_switch\_forks**

1. if(new\_head.head.previous==head->id)，说明当前新block跟之前的head连接起来了，这是正常状态，此时动作：  
   apply\_block( new\_head->block, s );

fork\_db.mark\_in\_current\_chain( new\_head, true );

fork\_db.set\_validity( new\_head, true );

head = new\_head;

1. else if( new\_head->id != head->id )

此时既没接上，两者的id又不一样，说明发生了分叉，需要进行分支切换。要从head切到new\_head。

先找到new\_head和head的共同祖先块，然后把从head到这个共同祖先块之间的块都pop掉，然后把从共同祖先到new\_head之间的块一个个调用apply\_block将其加到fork\_db上。

**apply\_block:**

是个非常重要的操作，即把新块的动作执行后，更新到head。

Apply\_block通过四部分完成，pending是此过程中的核心数据，每个block都是暂存在pending中做各种操作的。

1.Startblock:

初始化pending，将其内容初始化为new block的，

创建包含onchain 这个action的trx，并且将其push进pending，这个action是跟对producer的奖励有关的。类似于btc的coinbase交易。

更新producer\_authority（没看懂做什么）

2.针对当前block里的每一个trx，调用push\_transaction执行它。push\_transaction是个很重要的调用，它会创建虚拟机实例且将trx在其中运行，并在执行完后emit一些channel数据。其结果会在pending里加入trx。后续详细研究。

3.finalize\_Block，计算并在数据库中更新最新的cpu,net资源情况。计算并设置action\_merkel,trx\_merkel,block\_id，即完成block header里的内容。

4.commit\_block,根据参数和配置决定是否把pending加进fork\_db，和reversible\_blocks这两个数据库。

emit( self.accepted\_block, pending->\_pending\_block\_state );

pending->push();

pending.reset();

start up 就这样了，之后就是application.exec，应该是application监听从io来的信息，通过其他plugin，利用method和channel调起chain\_plugin的相应功能。

Push\_transaction的实现

需要进行一堆资源相关计算。这部分等后续研究资源时再细看。

建立transaction\_context trx\_context，并对其进行一些初始化。

注意transaction\_context的初始化是：trx\_context(self, trn, trx->id, start);  
会把controller传给transaction\_context，用当前的trx及trx->id来初始化对象，这里需要注意pending->\_block\_status ，block\_status是个enum，有irreversible，validated，complete，incomplete几个值。其中incomplete是自己组装块，不是从别的节点同步来的，当我们向rpc节点push\_transaction时应该在节点上发生这种情况，注意此时建立apply\_contex，即action执行的context的时候，是根据pending传入的trx\_id，block\_id的，而如果此时这个pending是在rpc节点，而不是bp，那么这个block\_ num应该不是最终的。后续在bp打包时是可能会变化的。所以不要用此时的block\_ num来作为最终的block\_ num来考虑执行结果。即：**不要用向rpc节点push\_transaction时收到的返回结果里的block\_num作为最终block\_num。这是已经验证过的。**

而trx\_id貌似不会变，还需进一步了解为什么。Block\_id我想应该会变的，未验证过。

另外注意可以通过pending->\_block\_status判断当前是在replay还是在正常出块，如果是replay，这个值不会是incomplete，只有在出块的才是incomplete。

trx在执行后，会执行：

if(read\_mode != db\_read\_mode::SPECULATIVE && pending->\_block\_status == controller::block\_status::incomplete){

trx\_context.undo();

}

即如果读模式不是SPECULATIVE且当前block\_status是incomplete，这个trx是要revert的。（这种情况理论上不会出现吧）

调用authorization.check\_authorization检查签名。

调用trx\_context.exec执行.执行过程有点复杂，也比较独立，在新文档”Transaction执行过程分析”中分析吧。

调用trx\_context.finalize,这个函数主要是计算该执行耗费的资源，cpu,net,ram，并更新。

将这个trx放入pending->\_pending\_block\_state

Emit出accepted\_transaction和applied\_transaction信号。

unapplied\_transactions.erase( trx->signed\_id );

#### on\_irreversible

fork\_db里有一个：

signal<void(block\_state\_ptr)> irreversible;

这是个信号槽，也就是观察者模式的东西。当一个block state变为不可逆时，就会emit这个signal，此时要把这个block从fork\_db里移除，除非它是head。

在controller\_impl的构造函数它被连接到on\_irreversible：

fork\_db.irreversible.connect( [&]( auto b ) {

on\_irreversible(b);

});

在on\_irreversible里，会：

db.commit( s->block\_num );

blog.append(s->block);

reversible\_blocks里小于等于这个不可逆块的块全都remove

如果readmode是irreverisible的话，需要调用apply\_block将该block执行。

emit(self.irreversible\_block, s);

#### push\_scheduled\_transaction

在apply\_block时调用，当遍历block里的trx，trx没有packed\_transaction，而只有transaction\_id时，说明是scheduled\_transaction，此时调用该接口。

整体过程与push\_transaction类似，区别在于：

1. 需要通过trx\_id在db的generated\_transaction\_multi\_index里取出该trx，然后在db里remove它，(如果因为非trx自身逻辑原因造成失败，可以通过临时储存的undo\_session来恢复的)
2. 如果已超时，把超时信息加进block，直接返回返回
3. 执行时的出错处理不同。如需要调用onerror，需要判断是否是subjective错误及作出对应处理。

Scheduled\_transaction保存在db的generated\_transaction\_object中。它通过apply\_context. schedule\_deferred\_transaction添加进去。对合约来说，在transaction.h的send\_deferred里调用的。

## Read\_only的实现

基本都是从controller里取出数据，转换成chain\_plugin中定义好的结构，返回出去。

## Read\_write的实现

Push\_transaction已经分析过。

Push\_block是调用incoming::methods::block\_sync，这个method定义在producer\_plugin里的on\_incoming\_block。后续分析producer\_plugin时再看。

## Chain\_plugin与其他plugin之间的接口研究

即研究method和channel，了解它们