# Transaction运行在哪里

这是trx进入到block的入口。

Transaction运行在chain\_plugin的controller里，在调用rpc接口执行push\_transaction,或者replay时都会执行。区别在于当前的block\_status不同，replay是不可逆，push\_trx时是incomplete，当然还有其他情况。

Transaction是在controller里，pending里构建和执行的，其过程在chain\_plugin分析里有。

执行过程主要涉及两个context:

transaction\_context和apply\_context。分别对应trx和action的执行。

## transaction\_context

接口：

void init( uint64\_t initial\_net\_usage);

void init\_for\_implicit\_trx( uint64\_t initial\_net\_usage = 0 );

void init\_for\_input\_trx( uint64\_t packed\_trx\_unprunable\_size,

uint64\_t packed\_trx\_prunable\_size,

bool skip\_recording);

void init\_for\_deferred\_trx( fc::time\_point published );

void exec();

void finalize();

void squash();

void undo();

其他一些跟资源和定时相关的。

数据：比较多和杂，列些主要的：

controller& control;

const signed\_transaction& trx;

transaction\_id\_type id;

optional<chainbase::database::session> undo\_session;

transaction\_trace\_ptr trace;

vector<action\_receipt> executed;

其中：

struct **transaction\_trace** {

transaction\_id\_type id;

uint32\_t block\_num = 0;

fc::optional<block\_id\_type> producer\_block\_id;

fc::optional<transaction\_receipt\_header> receipt;//表示在block里的被接受情况，如已执行、已超时、已失败等。

vector<action\_trace> action\_traces; ///< disposable

transaction\_trace\_ptr failed\_dtrx\_trace;

fc::optional<fc::exception> except;

std::exception\_ptr except\_ptr;

};

struct **action\_trace** : public base\_action\_trace {

vector<action\_trace> inline\_traces;

};

struct **base\_action\_trace** {

action\_receipt receipt;

action act;

bool context\_free = false;

fc::microseconds elapsed;

string console;

transaction\_id\_type trx\_id; ///< the transaction that generated this action

uint32\_t block\_num = 0;

block\_timestamp\_type block\_time;

fc::optional<block\_id\_type> producer\_block\_id;

flat\_set<account\_delta> account\_ram\_deltas;

fc::optional<fc::exception> except;

};

Transaction\_context在push\_transaction里被创建，一般来说，设置一些参数后，会调用：

init\_for\_implicit\_trx或init\_for\_input\_trx。

init\_for\_input\_trx应该是平常大部分action执行的情况。（implicit是一些比如给bp发工资等系统的调用。）

init\_for\_input\_trx里会执行一些资源相关的检查和更新，验证账户等。然后调用init.init里全是计算和更新资源限制的。

初始化完成后，会在controller里调用authorization.check\_authorization检查签名是否合法。

然后调用trx\_context.exec()执行它。执行过程：

如果配置为要执行context free的action，就取出其context free actions，调用dispatch\_action执行。

如果是延时的trx，就调用schedule\_transaction，否则就调用dispatch\_action一条条执行actions。

然后trx\_context.finalize();这个主要是计算并更新本次执行耗费的资源。

然后该trx会被加入到controller的pending里面，后续会更新在block里。

## Apply\_context

Dispatch\_action是创建一个apply\_context对象，初始化为当前action，然后调用其exec。

其接口有：

运行相关的：

void exec\_one( action\_trace& trace );

void exec( action\_trace& trace );

void execute\_inline( action&& a );

void execute\_context\_free\_inline( action&& a );

void schedule\_deferred\_transaction( const uint128\_t& sender\_id, account\_name payer, transaction&& trx, bool replace\_existing );

bool cancel\_deferred\_transaction( const uint128\_t& sender\_id, account\_name sender );

bool cancel\_deferred\_transaction( const uint128\_t& sender\_id )

鉴权，Console，数据库相关的等，

主要数据：

controller& control;

chainbase::database& db; ///< database where state is stored

transaction\_context& trx\_context; ///< transaction context in which the action is running

const action& act; ///< message being applied

account\_name receiver; ///< the code that is currently running

vector<bool> used\_authorizations; ///< Parallel to act.authorization; tracks which permissions have been used while processing the message

uint32\_t recurse\_depth; ///< how deep inline actions can recurse

bool privileged = false;

bool context\_free = false;

bool used\_context\_free\_api = false;

iterator\_cache<key\_value\_object> keyval\_cache;

vector<account\_name> \_notified; ///< keeps track of new accounts to be notifed of current message

vector<action> \_inline\_actions; ///< queued inline messages

vector<action> \_cfa\_inline\_actions; ///< queued inline messages

std::ostringstream \_pending\_console\_output;

flat\_set<account\_delta> \_account\_ram\_deltas; ///< flat\_set of account\_delta so json is an array of objects

都比较明显，就不一一介绍了。

执行过程中，exec\_one是其基本执行单位，它会将一些外界信息如block\_num等写入trace，然后找到对应的action的handler并执行，再在虚拟机里执行。

trace.trx\_id = trx\_context.id;

trace.block\_num = control.pending\_block\_state()->block\_num;

trace.block\_time = control.pending\_block\_time();

trace.producer\_block\_id = control.pending\_producer\_block\_id();

trace.act = act;

trace.context\_free = context\_free;

这里trace是传入的action，即此时才会把pending里传入的block\_num等值设置给action（所以action里能取到trx,block相关的信息，但是这些信息可能在rpc节点pending里，也可能在bp上，此时block\_status可能是incomplete，也可能是ireservable，所以这个值不一定是最终值，而且我认为无法判断它是不是最终值。但是这个结论当前也没法下，因为producer\_block\_id可能是一个判断方法）。

通过control.find\_apply\_handler( receiver, act.account, act.name );查找出该action的handler，有的话，检查有没在黑名单里，没有就(\*native)( \*this )执行。然后control.get\_wasm\_interface().apply( a.code\_version, a.code, \*this );

奇怪，为什么两个执行？第二个是在wasm虚拟机里执行的，第一个后面研究。但对于为什么有两个，我没有理解。

Exec：

执行当前的action，然后把\_notified[]里面的每个要通知的用户找出来，将其设置为receiver，重新执行一遍action。

\_notified.push\_back(receiver);//将receiver加入\_notified，

exec\_one( trace );//执行action

/\*把当前的receiver改成\_notified里的，并执行 \*/

for( uint32\_t i = 1; i < \_notified.size(); ++i ) {

receiver = \_notified[i];

trace.inline\_traces.emplace\_back( );

exec\_one( trace.inline\_traces.back() );

}

然后在trx\_context里重新dispatch\_action，把context free action都执行了，然后同样的方法执行所有的inline actions。

for( const auto& inline\_action : \_cfa\_inline\_actions ) {

trace.inline\_traces.emplace\_back();

trx\_context.dispatch\_action( trace.inline\_traces.back(), inline\_action, inline\_action.account, true, recurse\_depth + 1 );

}

for( const auto& inline\_action : \_inline\_actions ) {

trace.inline\_traces.emplace\_back();

trx\_context.dispatch\_action( trace.inline\_traces.back(), inline\_action, inline\_action.account, false, recurse\_depth + 1 );

}

这里注意：

1. Action和其通知接收者是在同一个apply\_context里执行的。而cfa\_inline\_action，inline\_actions是在其各自的apply\_context里执行的。
2. Inline\_actions是递归调用的，但是注意不会无限制的递归下去，在exec里有对递归的层级做限制，超过就assert了，这个限制在：control.get\_global\_properties().configuration.max\_inline\_action\_depth

**总结**：

在trx的执行中，会先执行当前action，然后执行require\_receipt里通知action，他们是在同一个context里执行的，其区别只是receiver不同。

然后新建context,在其中顺序执行所有的context free actions，

然后新建context,在其中顺序执行所有的inline actions，

Context free和inline都可以递归调用，但是递归层数有限制。主网设置的是4

**问题**：

1. Exec\_one里的(\*native)( \*this )在做什么？
2. \_inline\_actions是什么时候初始化的？是在apply\_context::execute\_inline接口里把inline加进去的，但是这个什么时候执行的？

## (\*native)( \*this )分析

看代码容易知道它保存在controller的：

map< account\_name, map<handler\_key, apply\_handler> > apply\_handlers;

key是account\_name，即一个账户一个，

value也是个map：

key是typedef pair<scope\_name,action\_name> handler\_key;可以认为是针对一个action一个key。

Value是std::function<void(apply\_context&)>，即我们当前要分析的函数指针。

主map是通过set\_apply\_handler设置的。

有个宏：

#define SET\_APP\_HANDLER( receiver, contract, action) \

set\_apply\_handler( #receiver, #contract, #action, &BOOST\_PP\_CAT(apply\_, BOOST\_PP\_CAT(contract, BOOST\_PP\_CAT(\_,action) ) ) )

即可以直接把handler设置为：apply\_#contract\_#action。

在controller\_impl的构造函数中可以知道它设置了很多，例如：

SET\_APP\_HANDLER( eosio, eosio, newaccount );

也就是设置eosio的newaccount的handler是：apply\_eosio\_newaccount，可以找到该函数定义在eosio\_contract.cpp里。其内容是修改controller.db数据库，添加一个新的account，并修改对应的资源。针对eosio的其他action也是类似的。

注意到：

1. 在apply\_context里函数名被叫做native
2. 除了controller里针对eosio的几个action，其他地方找不到设置该handler的调用。

我们可以认为这个只是针对eosio的这几个action做操作而已，即这几个handler只在这几个action上才存在，即系统需要在这几个action上做特殊处理。对于普通账号，是不存在这个handler的，普通账号只会执行虚拟机的动作。

即请注意：

针对这几个action，其被调用时，除了执行合约里的代码外，还会执行额外的动作：

SET\_APP\_HANDLER( eosio, eosio, newaccount );

SET\_APP\_HANDLER( eosio, eosio, setcode );

SET\_APP\_HANDLER( eosio, eosio, setabi );

SET\_APP\_HANDLER( eosio, eosio, updateauth );

SET\_APP\_HANDLER( eosio, eosio, deleteauth );

SET\_APP\_HANDLER( eosio, eosio, linkauth );

SET\_APP\_HANDLER( eosio, eosio, unlinkauth );

SET\_APP\_HANDLER( eosio, eosio, canceldelay );

其额外动作定义在eosio\_contract.cpp里。

## execute\_inline什么时候执行的

它是在wasm\_interface->transaction\_api->send\_inline里被调用的。

Send\_inline出现在eosiolib/action.h里，说明可以在写合约时直接调用。

## 总结

Action的调用，在简单的层面上就这样了。而从cleos上调用push action开始的流程，涉及到太多内容，目前还没法整理出来。可以参考下面的参考资料，里面有分析。

1. 其执行时的block\_num等，是在pending里执行push\_transaction时填入的，是rpc上本地的，不能认为是最终的。
2. 执行过程为：  
   先执行本action，然后执行被通知的，然后在新的context里执行所有的cfa，然后再新建个context执行所有的inline, inline是可以嵌套的，主网最多4层
3. Eosio的一些action，除了执行合约里的代码外，还要额外执行eosio\_contract.cpp里的handler代码。

## 更详细的参考

<https://blog.csdn.net/SunnyWed/article/details/81078078>  
这篇文章分析的很深入，到了虚拟机层面，推荐阅读。