system合约

delegate_bandwidth

用户资源

```
struct user_resources {  //
      account_name  owner;
      asset         net_weight;
      asset         cpu_weight;
      int64_t       ram_bytes = 0;
      asset         governance_stake;
      time          goc_stake_freeze = 0;

      uint64_t primary_key()const { return owner; }

      // explicit serialization macro is not necessary, used here only to
improve compilation time
      EOSLIB_SERIALIZE( user_resources, (owner)(net_weight)(cpu_weight)
(ram_bytes)(governance_stake)(goc_stake_freeze) )
   };
```

用户抵押记录

```
struct delegated_bandwidth {
      account_name  from;
      account_name  to;
      asset         net_weight;
      asset         cpu_weight;

      uint64_t  primary_key()const { return to; }

      // explicit serialization macro is not necessary, used here only to
improve compilation time
      EOSLIB_SERIALIZE( delegated_bandwidth, (from)(to)(net_weight)
(cpu_weight) )

   };
```

用户赎回记录

```
struct refund_request {    //记录赎回记录
      account_name  owner;
      time          request_time;
      eosio::asset  net_amount;
      eosio::asset  cpu_amount;

      uint64_t  primary_key()const { return owner; }

      // explicit serialization macro is not necessary, used here only to
improve compilation time
      EOSLIB_SERIALIZE( refund_request, (owner)(request_time)(net_amount)
(cpu_amount) )
   };
```

函数

```
其中比较核心的几个地方是 #define CORE_SYMBOL S(4,SYS) 是
                          S(4,RAMCORE)是中间代币
                          S(0,RAM) 是ram
```

- system_contract::buyram

```
//根据当前市场的份额，将需要购买的字节数转化为指定的EOS进行购买
void system_contract::buyrambytes( account_name payer, account_name
receiver, uint32_t bytes ) {
      //在数据库中查询RAMCORE发行量，默认为100000000000000
      auto itr = _rammarket.find(S(4,RAMCORE));
      auto tmp = *itr;
      auto eosout = tmp.convert( asset(bytes,S(0,RAM)), CORE_SYMBOL );
      //通过转化后，调用buyram使用EOS购买
      buyram( payer, receiver, eosout );
   }
```

RAM的交易机制采用Bancor算法，使每字节的价格保持不变，通过中间代币(RAMCORE)来保证EOS和RAM之间的交易流通性。从上源码看首先获得RAMCORE的发行量，再通过tmp.convert方法RAM->RAMCORE，RAMCORE->EOS(CORE_SYMBOL)再调用 buyram 进行购买。这里的CORE_SYMBOL不一定是指EOS，查看core_symbol.hpp，发现源码内定义为SYS，也就是说在没有修改的前提下，需要提前发行SYS代币，才能进行RAM购买。

- buy

```
void system_contract::buyram(account_name payer, account_name receiver,
asset quant)
{
      //验证权限
      require_auth(payer);
```

```
	//不能为0
	eosio_assert(quant.amount > 0, "must purchase a positive amount");

	auto fee = quant;
	fee.amount = ( fee.amount + 199 ) / 200;
	//手续费为0.5%，如果amoun为1则手续费为1，如果小于1，则手续费在amoun<free<1
quant.amount.
	//扣除手续费后的金额，也就是实际用于购买RAM的金额。如果扣除手续费后，金额为0，则
会引起下面的操作失败。
	auto quant_after_fee = quant;
	quant_after_fee.amount -= fee.amount; the next inline transfer will
fail causing the buyram action to fail.

	INLINE_ACTION_SENDER(eosio::token, transfer)
	(N(eosio.token), {payer, N(active)},
	 {payer, N(eosio.ram), quant_after_fee, std::string("buy ram")});

	//如果手续费大于0，则将手续费使用eosio.token合约中的transfer，将金额转移给
eosio.ramfee，备注为ram fee
	if (fee.amount > 0)
	{
		INLINE_ACTION_SENDER(eosio::token, transfer)
		(N(eosio.token), {payer, N(active)},
		 {payer, N(eosio.ramfee), fee, std::string("ram fee")});
	}

	int64_t bytes_out;

	//根据ram市场里的EOS和RAM实时汇率计算出能够购买的RAM总量
	const auto &market = _rammarket.get(S(4, RAMCORE), "ram market does
not exist");
	_rammarket.modify(market, 0, [&](auto &es) {
		//转化方法请参考下半部分
		bytes_out = es.convert(quant_after_fee, S(0, RAM)).amount;
	});

	//剩余总量大于0判断
	eosio_assert(bytes_out > 0, "must reserve a positive amount");

	//更新全局变量，总共可以使用的内存大小
	_gstate.total_ram_bytes_reserved += uint64_t(bytes_out);
	//更新全局变量，购买RAM冻结总金额
	_gstate.total_ram_stake += quant_after_fee.amount;

	user_resources_table userres(_self, receiver);
	auto res_itr = userres.find(receiver);
	if (res_itr == userres.end())
	{
		//在userres表中添加ram相关记录
```

```
            res_itr = userres.emplace(receiver, [&](auto &res) {
                  res.owner = receiver;
                  res.ram_bytes = bytes_out;
            });
      }
      else
      {
            //在userres表中修改ram相关记录
            userres.modify(res_itr, receiver, [&](auto &res) {
                  res.ram_bytes += bytes_out;
            });
      }
      //更新账号的RAM拥有量
      set_resource_limits(res_itr->owner, res_itr->ram_bytes, res_itr-
>net_weight.amount, res_itr->cpu_weight.amount);
}
```

- sellram

1.同样的，调用convert方法，将所出售的ram的字节数，根据市场价格换算为EOS的数量(tokens_out变量来表示),修改表格。

2.更新全局变量

- delegatebw

```
 //抵押token，换取网络与cpu资源
 //from是抵押者，receiver是token的接收者，大多数情况这俩个是同一个名字，但也可以抵押了
把币给别人，
 //transfer是true接受者可以取消。在验证完数据的合法性，以后调用changebw方法

 void system_contract::delegatebw( account_name from, account_name
receiver,
                                   asset stake_net_quantity,
                                   asset stake_cpu_quantity, bool
transfer )
   {
      eosio_assert( stake_cpu_quantity >= asset(0), "must stake a positive
amount" );
      eosio_assert( stake_net_quantity >= asset(0), "must stake a positive
amount" );
      eosio_assert( stake_net_quantity + stake_cpu_quantity > asset(0),
"must stake a positive amount" );
      eosio_assert( !transfer || from != receiver, "cannot use transfer
flag if delegating to self" );

      changebw( from, receiver, stake_net_quantity, stake_cpu_quantity,
transfer);
   } // delegatebw
```

*undelegatebw

```
// 取消抵押，释放网络和cpu。 from取消抵押取消抵押，receiver会失去投票权

void system_contract::undelegatebw( account_name from, account_name
receiver,
                                    asset unstake_net_quantity, asset
unstake_cpu_quantity )
   {
      eosio_assert( asset() <= unstake_cpu_quantity, "must unstake a
positive amount" );
      eosio_assert( asset() <= unstake_net_quantity, "must unstake a
positive amount" );
      eosio_assert( asset() < unstake_cpu_quantity + unstake_net_quantity,
"must unstake a positive amount" );
      eosio_assert( _gstate.total_activated_stake >= min_activated_stake,
                    "cannot undelegate bandwidth until the chain is
activated (at least 15% of all tokens participate in voting)" );

      changebw( from, receiver, -unstake_net_quantity, -
unstake_cpu_quantity, false);
   } // undelegatebw
```

- changebw

```
1. 要求from的授权
2.更新抵押记录表（更新del_bandwidth_table）
3.更新抵押总量表（更新user_resources_table）
4.更新退款表（更新refunds_table）
5.若有需要，发送延迟退款交易
6.更新选票权重
```

- - ***refund函数***：在***undelegateb函数***调用解除代币抵押后，将抵押的代币退回账户，会有个缓冲时间。

```
void system_contract::refund( const account_name owner );
```

system.cpp与system.hpp

system.hpp

- eosio.system.hpp中定义了合约类eosiosystem::system_contract，和一些结构体，以及函数的申明。。 1. eosio_global_state 全局状态
    1. producer_info 生产者信息
        1. voter_info 投票人信息
        2. goc_proposal_info goc提案信息
        3. goc_vote_info goc投票信息
        4. goc_reward_info goc奖励信息
- eosio_global_state , 定义了全局状态

```
struct eosio_global_state : eosio::blockchain_parameters {
      uint64_t free_ram()const { return max_ram_size -
total_ram_bytes_reserved; }

      uint64_t             max_ram_size = 64ll*1024 * 1024 * 1024;
      uint64_t             total_ram_bytes_reserved = 0;
      int64_t              total_ram_stake = 0;

      block_timestamp      last_producer_schedule_update;
      uint64_t             last_pervote_bucket_fill = 0;
      int64_t              pervote_bucket = 0;
      int64_t              perblock_bucket = 0;
      uint32_t             total_unpaid_blocks = 0; /// all blocks which
have been produced but not paid
      int64_t              total_activated_stake = 0;
      uint64_t             thresh_activated_stake_time = 0;
      uint16_t             last_producer_schedule_size = 0;
      double               total_producer_vote_weight = 0; /// the sum of
all producer votes
      block_timestamp      last_name_close;

      //GOC parameters
      int64_t              goc_proposal_fee_limit=  10000000;
      int64_t              goc_stake_limit = 1000000000;
      int64_t              goc_action_fee = 10000;
      int64_t              goc_max_proposal_reward = 1000000;
      uint32_t             goc_governance_vote_period = 24 * 3600 * 7;  //
7 days
      uint32_t             goc_bp_vote_period = 24 * 3600 * 7;  // 7 days
      uint32_t             goc_vote_start_time = 24 * 3600 * 3;  // vote
start after 3 Days

      int64_t              goc_voter_bucket = 0;
```

```
        int64_t               goc_gn_bucket = 0;
        uint32_t              last_gn_bucket_empty = 0;
    };
```

- producer_info 定义了生产者的信息

```
struct producer_info {
      // 生产者节点的拥有者
      account_name          owner;
      // 获得的投票数
      double                total_votes = 0;
      /// a packed public key object
      eosio::public_key     producer_key;
      bool                  is_active = true;
      // 生产者的url
      std::string           url;
      // 生产的区块数量
      uint32_t              unpaid_blocks = 0;
      // 生产产出区块的时间
      uint64_t              last_claim_time = 0;
      // 生产者位置
      uint16_t              location = 0;

      // 主键通过owner索引
      uint64_t primary_key()const { return owner;
        }
      // 二级索引通过投票数
      double   by_votes()const    { return is_active ? -total_votes :
total_votes;  }
      bool     active()const      { return is_active;
        }
      void     deactivate()       { producer_key = public_key(); is_active
= false; }

      // explicit serialization macro is not necessary, used here only to
improve compilation time
      EOSLIB_SERIALIZE( producer_info, (owner)(total_votes)(producer_key)
(is_active)(url)
                        (unpaid_blocks)(last_claim_time)(location) )
   };```


* voter_info 定义了投票人信息
```

struct voter_info { account_name owner = 0; /// the voter account_name proxy = 0; /// the proxy set by the voter, if any std::vector<account_name> producers; /// the producers approved by this voter if no proxy set int64_t staked = 0;

```
    /**
     *  Every time a vote is cast we must first "undo" the last vote weight,
before casting the
     *  new vote weight.  Vote weight is calculated as:
     *
     *  stated.amount * 2 ^ ( weeks_since_launch/weeks_per_year)
     */
    double                      last_vote_weight = 0; /// the vote weight
cast the last time the vote was updated

    /**
     * Total vote weight delegated to this voter.
     */
    double                      proxied_vote_weight= 0; /// the total vote
weight delegated to this voter as a proxy
    bool                        is_proxy = 0; /// whether the voter is a
proxy for others
```

```
    uint32_t                    reserved1 = 0;
    time                        reserved2 = 0;
    eosio::asset                reserved3;

    uint64_t primary_key()const { return owner; }
```

};

* goc_proposal _ info 定义了抵押过GOC token的用户提交propsal的信息

```c
struct goc_proposal_info {
      uint64_t              id;
      account_name          owner;
      asset                 fee;
      std::string           proposal_name;
      std::string           proposal_content;
      std::string           url;

      time                  create_time;
      time                  vote_starttime;
      time                  bp_vote_starttime;
      time                  bp_vote_endtime;

      time                  settle_time = 0;
      asset                 reward;

      double                total_yeas;
      double                total_nays;
      uint64_t              total_voter = 0;
```

```
        double                  bp_nays;
        uint16_t                total_bp = 0;

        uint64_t  primary_key()const     { return id; }
        uint64_t  by_endtime()const      { return bp_vote_endtime; }
        bool      vote_pass()const       { return total_yeas > total_nays;  }
        //need change to bp count
        bool      bp_pass()const         { return bp_nays < -7.0;  }
    };
```

- goc_vote _info 定义了proposal投票的信息

```
struct goc_vote_info {
      account_name            owner;
      bool                    vote;
      time                    vote_time;
      time                    vote_update_time;
      time                    settle_time = 0;

      uint64_t primary_key()const { return owner; }
    };
```

- goc_reward _info 对于BP出块的奖励信息

```
struct goc_reward_info {
       time           reward_time;
       uint64_t       proposal_id;
       eosio::asset   rewards = asset(0);

       uint64_t  primary_key()const { return proposal_id; }
      };
```

system.cpp

- void system_contract::setram(uint64_t max_ram_size)

```
    设置最大内存大小
```

- void system_contract::setparams(const eosio::blockchain_parameters& params)

  设置区块链参数
- void system_contract::setpriv(account_name account, uint8_t ispriv)

  设置账户权限

- void system_contract::rmvproducer(account_name producer)

  移除生产节点

- void system_contract::bidname(account_name bidder, account_name newname, asset bid)

  账号竞拍，对指定名字的账号进行竞拍，但有特殊要求，长度为0-11字符或者12字符且包含点新的竞价要比当前最高的竞价多出百分之10

- void native::newaccount(account_name creator，account_name newact /* no need to parse authorities const authority& owner，const authority& active*/)

  > 在创建新帐户后调用此Action，执行新账户的资源限制规则和命名规则

system合约

exchange_state.cpp

以eos为基础币，发行的2种类型的代币。一个账户的usd代币，给自己的btc代币转换100个usd调用这个cpp 例如一个dan 0 eos。 supply 1e+11 eos dan 100 usd quote 1e+8 usd dan 0 btc base 1e+8 btc

```
// 给出当前的state，计算一个新的state回来
  asset exchange_state::convert( asset from, symbol_type to ) {
     auto sell_symbol  = from.symbol;    // usd
     auto ex_symbol    = supply.symbol; // eos
     auto base_symbol  = base.balance.symbol; // btc
     auto quote_symbol = quote.balance.symbol;  // ust

     //print( "From: ", from, " TO ", asset( 0,to), "\n" );
     //print( "base: ", base_symbol, "\n" );
     //print( "quote: ", quote_symbol, "\n" );
     //print( "ex: ", supply.symbol, "\n" );

     if( sell_symbol != ex_symbol ) {     // 币的名字不相同
        if( sell_symbol == base_symbol ) {     // 如果卖出者的币名与base的币名相同
           from = convert_to_exchange( base, from );
        } else if( sell_symbol == quote_symbol ) {
           from = convert_to_exchange( quote, from );
        } else {
           eosio_assert( false, "invalid sell" );
        }
     } else {
        if( to == base_symbol ) {
```

```
            from = convert_from_exchange( base, from );
         } else if( to == quote_symbol ) {
            from = convert_from_exchange( quote, from );
         } else {
            eosio_assert( false, "invalid conversion" );
         }
      }

      if( to != from.symbol )
         return convert( from, to );

      return from;
   }
```

sell_symbol 与 基础币不相同，调用conver_to_exchange 转化为eos代币。 然后to 与from 不相同，在继续调用conver_from_exchange 转为btc代币

```
asset exchange_state::convert_to_exchange( connector& c, asset in ) {

      real_type R(supply.amount);  // 1e + 11;

      real_type C(c.balance.amount+in.amount);  // state资产新发行的代币。

      real_type F(c.weight/1000.0); // 代币所占比重,初始化设置好的

      real_type T(in.amount);// 新发行数量 100
      real_type ONE(1.0);
        // 根据这个算法得到对应的state资产的增发量。
       real_type E = -R * (ONE - std::pow( ONE + T / C, F) );

      //print( "E: ", E, "\n");

      int64_t issued = int64_t(E); // 大概是48999 ，增发100个usd，实际上要增发state这么多


      supply.amount += issued;  // 更新总发行量，然后某稳定数字资产下的token余额增加了
      c.balance.amount += in.amount;  //

       return asset( issued, supply.symbol );  // 以eos资产实际上增加的形式返回.
```

```
asset exchange_state::convert_from_exchange( connector& c, asset in ) {
      eosio_assert( in.symbol== supply.symbol, "unexpected asset symbol input" );
```

```
       real_type R(supply.amount - in.amount);  // 先找回原值，1e+11;
       real_type C(c.balance.amount);            // btc 余额不变，仍为1e+8
       real_type F(1000.0/c.weight);
       real_type E(in.amount);           // 489999
       real_type ONE(1.0);


      // potentially more accurate:
      // The functions std::expm1 and std::log1p are useful for financial
calculations, for example,
      // when calculating small daily interest rates: (1+x)n
      // -1 can be expressed as std::expm1(n * std::log1p(x)).
      // real_type T = C * std::expm1( F * std::log1p(E/R) );

       real_type T = C * (std::pow( ONE + E/R, F) - ONE); //大概是96
       //print( "T: ", T, "\n");
       int64_t out = int64_t(T);

       supply.amount -= in.amount; // 将eos增发的部分减掉，维持原来的1e+11
       c.balance.amount -= out;  // btc的总量减少了96.

       return asset( out, c.balance.symbol );
    }
```

最终的结果： dan 0 eos supply 1e+11 dan 0 usd base 1e+8usd dan 96 btc quote 9.999e+7btc

system.governance.cpp

对函数进行介绍

gocstake()

功能：进行抵押，在goc中可以通过抵押goc成为gn结点，从而提出提案

参数：

```
gocstake(account_name payer)
```

过程：

首先进行验证，获取当前时间，以及可以抵押的最大限度-10e。

然后在用户资源表中查找当前用户的用户资源，如果没有找到，则为该payer创建用户资源表；如果找到了，则判断是否有足够的资源可以抵押。

如果可以抵押，把goc抵押到gstake账户。

gocunstake()

** 功能：**赎回抵押的goc

参数：

```
gocunstake(account_name receiver)
```

过程：

验证权限，获取当前时间，判断是否非法输入，以及是否能够赎回

在可以的情况下，首先修改用户资源表中的对应数据项，然后调用内部的action把goc从系统账户(gstake)中取回。

gocnewprop()

功能：提出一个提案

参数：

```
gocnewprop(const account_name owner,
           asset fee,
           const std::string &pname,
           const std::string &pcontent,
           const std::string &url,
           const std::string &hash,
           uint16_t start_type)
```

过程：

验证权限，获取当前时间，判断是否为非法输入，然后判断缴纳费用的token是否为支持的类型，以及数量是否足够

调用内部action,把提案的费用从提出提案的账户转到系统的储蓄账户(saving),并赋予该提案一个唯一的id

调用emplace函数，创建该提案的详细内容，初始化提案的生成日期，票数。

返回proposal的ID

关于proposal：

Proposal是goc中特有的一种治理模式，用户可以支付一定的费用，提出一个proposal，维持三天，然后交由gn节点进行投票(耗时七天)，gn节点投票通过后，交由bp节点进行最终投票(耗时七天)，决定是否执行。 还有另一种目前仅为测试的的proposal类型，bp节点和gn节点共同投票。

gocupprop()

功能：更新提案的内容

参数：

```
gocupprop(const account_name owner,
          uint64_t id,
          const std::string &pname,
          const std::string &pcontent,
          const std::string &url,
          const std::string &hash)
```

过程：

验证权限，获取当前时间，判断是否为非法输入，然后判断该id是否有对应的proposal，是否已经过了修改时间(proposal提出的三天内才可以进行修改，三天后要进行投票)，判断修改人是否为proposal的提出者(只有他可以修改提案的内容)

验证通过后，对proposal的内容进行修改

gocsetpstage()

功能：设置提案的状态

参数：

```
gocsetpstage(const account_name owner,
             uint64_t id,
             uint16_t stage,
             time start_time)
```

分别为拥有者的名称，提案的id，期望的阶段，该阶段的开始时间

一个提案的状态共有(0-4)五个阶段，分别为创建阶段，普通投票阶段，bp投票阶段，结束阶段，确定阶段。

过程：

验证权限，判断是否为非法输入，然后判断是否存在对应的proposal，是否为所有者进行操作，提案是否已经生效(已经生效的提案无法修改其状态)。

根据输入的阶段，修改提案对应的时间参数，对于当前阶段之前的阶段，时间间隔统一修改为一小时。

gocvote()

功能：所用节点(包括bp节点)对提案进行投票

参数：

```
gocvote(account_name voter,
        uint64_t pid,
        bool yea)
```

过程：

1.验证投票者，验证提案是否存在，然后判断提案的状态，是否可以投票

2.读取该投票者的用户资源表，查看其抵押的goc是否达到投票所需的数量

3.读取该提案的票数表，然后在表下查找，看该投票者是否已经投过票，如果投过票，则检查是否与之前的结果一致，若不同对旧的投票状态进行更新，然后修改该提案的总票数并返回提示，若相同则跳过并返回提示；如果投票者之前没有投过票就创建一个新的票，并修改该提案的总票数。

4.最后检查一下，stake的冻结时间是否早于bp节点投票结束时间，如果是则修改到保持一致。

gocbpvote()

功能：bp节点针对提案进行投票(记反对票)

参数：

```
gocbpvote(account_name voter,
          uint64_t pid,
          bool yea)
```

过程：

1.验证投票者，获取当前时间，检查提案是否存在，然后检查该投票者是否为bp节点，该阶段只有bp节点可以投票，且不需要抵押goc

2.查看提案的bp阶段投票的起止时间，判断当前时间是否在其区间内；

3.之后与普通投票类似，记录bp节点的投票结果

producer_pay.cpp

```
const int64_t  min_pervote_daily_pay = 100'0000;
const int64_t  min_activated_stake   = 150'000'000'0000;  //15%激活
const double   continuous_rate       = 0.04879;          // 5% annual rate
const double   perblock_rate         = 0.0025;           // 0.25%
const double   standby_rate          = 0.0075;           // 0.75%
const uint32_t blocks_per_year       = 52*7*24*2*3600;   // the number of
half seconds per year  --half seconds a block
const uint32_t seconds_per_year      = 52*7*24*3600;
const uint32_t blocks_per_day        = 2 * 24 * 3600;
const uint32_t blocks_per_hour       = 2 * 3600;
const uint64_t useconds_per_day      = 24 * 3600 * uint64_t(1000000);
const uint64_t useconds_per_year     = seconds_per_year*1000000ll;
```

Producer_pay　主要是负责支付各种节点的goc奖励，goc每年增值5%，其中分配比例如下

```
    const double   goc_bp_rate = 0.2;   // 1% for BP

    const double   goc_vote_rate = 0.1; // 0.5% for BP votes

    const double   goc_gn_rate = 0.15;  // 0.75% for GN

    const double   goc_wps_rate = 0.55; // 2.75% for WPS
```

cpp文件中的以上常量规定了各种节点的收益比例。

下面对该文件中的一些函数进行介绍

onblock()

功能： 启动区块

参数：

```
onblock( block_timestamp timestamp,
    account_name producer )
```

过程：

1.进行验证后，判断当前的抵押数额是否小于最小数目，若小于则返回。（EOS有15%以上的投票活跃度时，主网才会激活）

2.如果还没有开始产生收益，则初始化。

3.检查该生产者是否创建了对应的对象，如果没有则创建一个

4.每分钟更新一次生产者，每0.5秒更新一次区块的时间戳

claimreward()

功能：按之前规定的比例，新生区块后分配奖励，生成奖励每天一次，提案的奖励每周一次

参数：

```
claimrewards( const account_name& owner )
```

过程：

1.验证身份后，检查生产者是否为激活状态，然后检查是否激活；

2.首先进行出块奖励的分配，检查距离上次分配是否已经过去一天(规定的间隔时间)；

3.然后开始声明，声明完成后更新gstate的参数

4.之后进行goc独有的投票奖励的分配，检查距离上次分配是否间隔一周

5.将所有投票结束的提案以及还没有执行的提案添加到容器中

6.如果有已经结束的提案，则对这些提案进行奖励的声明（给投票者）

6.将奖励进行声明，之后更新gstate的参数

7.然后GOC对已经声明但还未支付的奖励进行支付

voting.cpp

下面介绍该cpp文件中的函数

regproducer()

功能： 注册生产者帐号

参数：

```
const account_name producer,
const eosio::public_key& producer_key,
const std::string& url,
uint16_t location
```

过程：

1.首先验证输入格式的正确性，并进行验证

2.检查该生产者是否已经注册，如果已经注册就更新信息，如果没有就注册并复制。

upproducer()

功能： 注销生产者

参数：

```
const account_name producer,
```

过程：

1.首先验证输入格式的正确性，并进行验证

2.如果找到该生产者，就把它的状态变为未激活

update_elected_producers()

功能： 更新被选中的生产者结点，即BP结点

参数：

```
block_timestamp block_time
```

过程：

1.首先更新gstate中的bp结点列表为输入的时间

2.然后根据规则更新BP结点列表

stake2vote()

功能： 投票权重的更新公式

参数：

```
int64_t staked
```

过程：

```
double weight = int64_t( (now() - (block_timestamp::block_timestamp_epoch
/1000)) / (seconds_per_day * 7) )  / double( 52 );
 return double(staked) * std::pow( 2, weight );
```

voteproducer()

功能： 更新被选中的生产者结点，即BP结点

参数：

```
const account_name voter_name,
const account_name proxy,
const std::vector<account_name>& producers
```

过程：

1.进行验证

2.更新投票信息

update_votes()

功能： voteproducer函数中更新票数的具体过程

参数：

```
const account_name voter_name,
const account_name proxy,
const std::vector<account_name>& producers
```

过程：

1.检查是否为代理账户投票，之后分别检测相关信息是否符合要求：投票总数小于30，代理账户不能再使用代理，自己不能交给自己代理，必须先抵押然后才能投票等等）

2.如果投票者第一次投票，我们要把它的抵押额更新/添加到EOS总的抵押额度上

3.根据该投票者抵押的数额，计算他的投票权重

4.计算之前投票的权重是否丢失，如果丢失则在总的投票权重中减去这一部分

5.将增加的投票更新到系统上

regproxy()

功能： 注册一个账户为代理账户

参数：

```
const account_name proxy,
bool isproxy
```

过程：

使用了代理功能的账号无法注册为代理

propagate_weight_change()

功能： 更改投票的权重

参数：

```
const voter_info& voter
```

过程：

1.注册为代理的账号不能再使用代理

2.如果是代理账号，先更新代理账号的相关数据

3.判断权重的变化是否大于1

4.代理账号和普通账号分别修改

bios分析

bios.cpp主要功能在于控制其他账户的资源分配和权限

Head

- bios(action_name self):contract(self){}

初始化合约名字

- setpriv(account_name account,uint8_t ispriv)

检测自己的权限，然后设置账号的权限。uint8_t 是unsigned char类型的，ispriv代表着级别

- setalimits 限制指定账户的所用的资源，内存大小，网络权重，cpu权重
- setglimits

设置区块链的资源，无任何操作，暂时无效

- setprods(std::vector'eosio::producer_keyschedule) 设置区块链生产节点， schedule 参数是必要的检查

```
read_action_data(buffer,size) 读取调用action所调用的size

set_proposed_producers(bufffer,size) 设置生产区块的节点
```

- reqauth(action_from)

检测权限
