



智能合约安全审计报告



慢雾安全团队于 2021-04-27 日，收到 Defibox 基金会对 Defibox swap 的智能合约安全审计申请。

如下为本次智能合约安全审计细节及结果：

合约哈希：

SHA256(dbswap.wasm)=

23b1d09ae1a434e69b6ffd3677b0165bb3dab9ea9ddb1b9c3e8f9b8fe85a489a

SHA256(lptoken.wasm)=

4647d42736b415317c237016b4ffedc52976b5486bfc0f72ec386ce696502f9a

复审合约哈希：

SHA256(lptoken-2021-05-26.wasm)=

a19c26b432f0ae2296d3fd7664624725f30db9e756756fd976bf6238bbf0148e

编译器版本：

eosio-cdt-v1.7.0

本次审计项及结果：

（其他未知安全漏洞不包含在本次审计责任范围）

序号	审计大类	审计子类	审计结果
1	溢出审计	-	通过
2	权限控制审计	权限漏洞审计	通过
		权限过大审计	通过
3	安全设计审计	硬编码地址安全	通过
		显现编码安全	通过
		异常校验审计	通过
		类型安全审计	通过
4	性能优化审计	-	通过
5	设计逻辑审计	-	通过
6	拒绝服务审计	-	通过
7	回滚攻击审计	-	通过
8	重放攻击审计	-	通过
9	假通知审计	-	通过
10	假错误通知审计	-	通过

11	假币审计	-	通过
12	随机数安全审计	-	通过
13	粉尘攻击安全审计	-	通过
14	微分叉安全审计	-	通过
15	排挤攻击安全审计	-	通过

备注：审计意见及建议见代码注释 //SlowMist//.....

审计结果：**通过**

审计编号：0X002104280003

审计日期：2021 年 04 月 28 日

复审时间：2021 年 06 月 02 日

审计团队：慢雾安全团队

（声明：慢雾仅就本报告出具前已经发生或存在的事实出具本报告，并就此承担相应责任。对于出具以后发生或存在的事实，慢雾无法判断其智能合约安全状况，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于信息提供者截至本报告出具时向慢雾提供的文件和资料（简称“已提供资料”）。慢雾假设：已提供资料不存在缺失、被篡改、删减或隐瞒的情形。如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，慢雾对由此而导致的损失和不利影响不承担任何责任。慢雾仅对该项目的安全情况进行约定内的安全审计并出具了本报告，慢雾不对该项目背景及其他情况进行负责。）

总结：此为 Defibox Swap 合约，经反馈修正后，综合评估合约无已知风险。

风控建议：

1、建议增加代币黑名单功能，防止恶意欺诈；

以下针对合约代码进行详细分析，分析写于注释处。

swap/include/dbswap.cpp

```
#include <dbswap.hpp>

ACTION dbswap::createpair(name creator, token token0, token token1) {
    require_auth(creator);
    check(_config.status == 0, "Swap suspended");
    check(token0 != token1, "Same token");
    auto supply0 = get_supply(token0.contract, token0.symbol.code());
    check(supply0.amount > 0 && supply0.symbol == token0.symbol, "Invalid symbol0");
    auto supply1 = get_supply(token1.contract, token1.symbol.code());
    check(supply1.amount > 0 && supply1.symbol == token1.symbol, "Invalid symbol1");

    auto hash0 = uint128_hash(token0.to_string() + "-" + token1.to_string());
    auto hash1 = uint128_hash(token1.to_string() + "-" + token0.to_string());
    // hash_index hashes(_self, _self.value);

    //SlowMist// 不允许创建相同交易对

    auto byhash_index = _pairs.get_index<"byhash"_n>();
    auto itr0 = byhash_index.find(hash0);
    check(itr0 == byhash_index.end(), "Pair already exists");
    auto itr1 = byhash_index.find(hash1);
    check(itr1 == byhash_index.end(), "Pair already exists");

    //SlowMist// uint64_t 全局自增 ID，避免重复

    auto pair_id = get_pair_id();

    _pairs.emplace(creator, [&](auto &a) {
        a.id = pair_id;
        a.token0 = token0;
        a.token1 = token1;
        a.reserve0.symbol = token0.symbol;
        a.reserve1.symbol = token1.symbol;
        a.block_time_last = current_time_point();
    });

    auto data = make_tuple(pair_id, creator, token0, token1);
    action(permission_level[_self, "active"_n], _self, "createlog"_n, data).send();

    auto max_supply = asset(1000000000000000000, id_to_symbol(pair_id));
```

```
auto data2 = make_tuple(_self, max_supply);
// check(!max_supply.is_valid(), max_supply.to_string());
action(permission_level{ _self, "active"_n }, name(LPTOKEN_CONTRACT), "create"_n, data2).send();
}

ACTION dbswap::removepair(uint64_t pair_id) {
    check(_config.status == 0, "Swap suspended");
    auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");
    check(p_itr->liquidity_token == 0, "Unable to remove active pair.");
    // fix bug: After creation, the pair will be removed by hackers immediately.
    auto time_elapsed = (current_time_point() - p_itr->block_time_last).to_seconds();

    //SlowMist// 交易对需要在 1 分钟后移除, 防止交易对被恶意移除

    check(time_elapsed > 60, "Please remove pair after 1 minute.");

    orders_index orders(_self, pair_id);
    check(orders.begin() == orders.end(), "There are still some orders.");

    _pairs.erase(p_itr);

    auto data = make_tuple(id_to_symbol(pair_id).code());
    action(permission_level{ _self, "active"_n }, name(LPTOKEN_CONTRACT), "destroy"_n, data).send();
}

ACTION dbswap::cancel(name owner, uint64_t pair_id) {
    require_auth(owner);
    check(_config.status == 0, "Swap suspended");
    orders_index orders(_self, pair_id);
    auto itr = orders.require_find(owner.value, "You don't have any deposit.");
    auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");

    // check balance
    if (itr->quantity0.amount > 0) {
        check_currency_balance(p_itr->token0);
        inline_transfer(p_itr->token0.contract, _self, owner, itr->quantity0, std::string("Defibox: cancel refund"));
        sub_currency_balance(p_itr->token0.contract, itr->quantity0);
    }
    if (itr->quantity1.amount > 0) {
        check_currency_balance(p_itr->token1);
        inline_transfer(p_itr->token1.contract, _self, owner, itr->quantity1, std::string("Defibox: cancel refund"));
        sub_currency_balance(p_itr->token1.contract, itr->quantity1);
    }
}
```

```
}  
orders.erase(itr);  
}
```

//SlowMist// 提现逻辑

```
void dbswap::withdraw(name owner, uint64_t pair_id, asset quantity) {  
    require_auth(owner);  
  
    check(_config.status == 0, "Swap suspended");  
  
    uint64_t amount = quantity.amount;  
  
    // retire  
    auto rdata = make_tuple(quantity, string("retire LP token"));  
    action(permission_level[_self, "active"_n], name(LPTOKEN_CONTRACT), "retire"_n, rdata).send();  
  
    auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");  
    uint64_t reserve0 = p_itr->reserve0.amount;  
    uint64_t reserve1 = p_itr->reserve1.amount;  
    //double factor = (1.0 * amount / p_itr->liquidity_token);  
  
    uint64_t amount0 = (uint128_t)reserve0 * amount / p_itr->liquidity_token; //SlowMist// 结合上下文环境，此处不
```

会导致算术溢出，建议使用更友好的 `safemath` 写法

```
    uint64_t amount1 = (uint128_t)reserve1 * amount / p_itr->liquidity_token; //SlowMist// 结合上下文环境，此处不
```

会导致算术溢出，建议使用更友好的 `safemath` 写法

```
    check(amount0 > 0 && amount1 > 0 && amount0 <= reserve0 && amount1 <= reserve1, "Insufficient liquidity");  
  
    // check balance  
    check_currency_balance(p_itr->token0, p_itr->token1);  
  
    // update pair  
    _pairs.modify(p_itr, same_payer, [&](auto &a) {  
        a.liquidity_token = safemath::sub(a.liquidity_token, amount);  
    });  
  
    inline_transfer(p_itr->token0.contract, _self, owner, asset(amount0, p_itr->token0.symbol), string("Defibox: withdraw"));  
    inline_transfer(p_itr->token1.contract, _self, owner, asset(amount1, p_itr->token1.symbol), string("Defibox: withdraw"));
```

```
sub_currency_balance(p_itr->token0.contract, asset(amount0, p_itr->token0.symbol));
sub_currency_balance(p_itr->token1.contract, asset(amount1, p_itr->token1.symbol));

uint64_t balance0 = safemath::sub(reserve0, amount0);
uint64_t balance1 = safemath::sub(reserve1, amount1);
update(pair_id, balance0, balance1, reserve0, reserve1);

auto data = make_tuple(pair_id, owner, amount, asset(amount0 * -1, p_itr->token0.symbol), asset(amount1 * -1,
p_itr->token1.symbol), p_itr->liquidity_token, p_itr->reserve0, p_itr->reserve1);
action(permission_level{self, "active"_n}, self, "liquiditylog"_n, data).send();
}

ACTION dbswap::updatestatus(uint8_t status) {
    require_auth(name(ADMIN_ACCOUNT));
    _config.status = status;
    _configs.set(_config, self);
}

ACTION dbswap::updatefees(uint8_t trade_fee, uint8_t protocol_fee, name fee_account) {
    require_auth(name(ADMIN_ACCOUNT));
    check(trade_fee > protocol_fee, "trade_fee must greater than protocol_fee");
    _config.trade_fee = trade_fee;
    _config.protocol_fee = protocol_fee;
    _config.fee_account = fee_account;
    _configs.set(_config, self);
}

//SlowMist// 转账逻辑

void dbswap::ontransfer(name from, name to, asset quantity, string memo, name code) {
    if (from == self || to != self) {
        return;
    }
    if (from == name(ASSIST_ACCOUNT)) {
        return;
    }
    if (from == name("swap.newdex"_n)) {
        return;
    }
    require_auth(from);

    check(_config.status == 0, "Swap suspended");
```

```
vector<std::string> strs = split(memo, ",");
string action = strs.size() > 0 ? strs[0] : "";
// string_to_uint64
if (action == "deposit") {
    check(strs.size() == 2, "Invalild deposit memo");
    uint64_t pid = strtoull(strs[1].c_str(), NULL, 10);
    do_deposit(pid, from, code, quantity);
} else if (action == "swap") {
    check(strs.size() == 3, "Invalild swap memo");
    uint64_t min_amount = strtoull(strs[1].c_str(), NULL, 10);
    vector<std::string> strids = split(strs[2], "-");
    vector<uint64_t> ids;
    for (uint8_t i = 0; i < strids.size(); i++) {
        uint64_t pid = strtoull(strids[i].c_str(), NULL, 10);
        ids.push_back(pid);
    }
    do_swap(ids, from, code, quantity, min_amount);
} else {
    if (code == name(LPTOKEN_CONTRACT)) {
        // lp token return
        uint64_t pid = symbol_to_id(quantity.symbol.code().to_string().substr(3));
        // withdraw
        withdraw(from, pid, quantity);
        return;
    }
    check(false, "Invalid memo");
}
}
```

//SlowMist// 交换逻辑

```
void dbswap::do_swap(vector<uint64_t> ids, name from, name contract, asset quantity, uint64_t min_amount) {
    extended_asset current_asset(quantity, contract);
    for (uint8_t i = 0; i < ids.size(); i++) {
        auto next_asset = swap(ids[i], from, current_asset.contract, current_asset.quantity);
        // mining
        action(permission_level{self, "active"_n}, name(MINE_ACCOUNT), name("mine"), std::make_tuple(ids[i], from,
current_asset, next_asset)).send();
        current_asset = next_asset;
    }
}
```



```
//check min_amount
check(min_amount == 0 || current_asset.quantity.amount >= min_amount, "Returns less than expected");

// transfer
inline_transfer(current_asset.contract, _self, from, current_asset.quantity, string("Defibox: swap token"));

add_currency_balance(contract, quantity);
sub_currency_balance(current_asset.contract, current_asset.quantity);
}

void dbswap::do_deposit(uint64_t pair_id, name owner, name contract, asset quantity) {
    auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");

    token input = { contract, quantity.symbol };
    check(input == p_itr->token0 || input == p_itr->token1, "Invalid deposit.");

    orders_index orders(_self, pair_id);
    auto itr = orders.find(owner.value);
    if (itr == orders.end()) {
        itr = orders.emplace(_self, [&](auto &a) {
            a.owner = owner;
            a.quantity0.symbol = p_itr->token0.symbol;
            a.quantity1.symbol = p_itr->token1.symbol;
        });
    }
    orders.modify(itr, same_payer, [&](auto &a) {
        if (input == p_itr->token0) {
            a.quantity0 += quantity;
        } else if (input == p_itr->token1) {
            a.quantity1 += quantity;
        }
    });

    add_currency_balance(contract, quantity);

    // if (itr->quantity0.amount > 0 && itr->quantity1.amount > 0) {
    //     action(permission_level{_self, "active"_n, _self, name("deposit")}, std::make_tuple(owner, pair_id)).send();
    // }
}

void dbswap::deposit(name owner, uint64_t pair_id) {
    name payer = owner;
    if (has_auth(name(ADMIN_ACCOUNT))) {
```

```
payer = _self;
} else {
    require_auth(owner);
}

orders_index orders(_self, pair_id);
auto o_itr = orders.require_find(owner.value, "You don't have any deposit.");
auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");
check(o_itr->quantity0.amount > 0 && o_itr->quantity1.amount > 0, "You need transfer both tokens");

// check balance
check_currency_balance(p_itr->token0, p_itr->token1);

uint64_t amount0 = o_itr->quantity0.amount;
uint64_t amount1 = o_itr->quantity1.amount;
uint64_t amount0_refund = 0;
uint64_t amount1_refund = 0;
uint64_t reserve0 = p_itr->reserve0.amount;
uint64_t reserve1 = p_itr->reserve1.amount;

// calc amount0 and amount1
if (reserve0 > 0 || reserve1 > 0) {
    uint128_t amount_temp = (uint128_t)amount0 * reserve1 / reserve0;

    //SlowMist// 防止数值溢出，很好的检查

    check(amount_temp < asset::max_amount, "Input amount too large");
    uint64_t amount1_matched = amount_temp;
    if (amount1_matched <= amount1) {
        amount1_refund = amount1 - amount1_matched;
        amount1 = amount1_matched;
    } else {
        amount_temp = (uint128_t)amount1 * reserve0 / reserve1;
        check(amount_temp < asset::max_amount, "Input amount too large");
        uint64_t amount0_matched = amount_temp;
        amount0_refund = amount0 - amount0_matched;
        amount0 = amount0_matched;
    }
}

// calc liquidity
uint64_t liquidity = 0;
```

```
uint64_t total_liquidity = p_itr->liquidity_token;
if (total_liquidity == 0) {
    liquidity = sqrt((uint128_t)amount0 * amount1);
    check(liquidity >= MINIMUM_LIQUIDITY, "Insufficient liquidity minted");
} else {
    liquidity = std::min((uint128_t)amount0 * total_liquidity / reserve0, (uint128_t)amount1 * total_liquidity / reserve1);

    //check result
    double factor1 = amount0 < amount1 ? 1.0 * amount0 / amount1 : 1.0 * amount1 / amount0;
    double factor2 = amount0 < amount1 ? 1.0 * p_itr->reserve0.amount / p_itr->reserve1.amount : 1.0 *
p_itr->reserve1.amount / p_itr->reserve0.amount;
    check(fabs(factor1 - factor2) < 0.01, "Insufficient liquidity");
}

// mint LP
auto issue_quantity = asset(liquidity, id_to_symbol(pair_id));
// check(false, owner.to_string() + " " + issue_quantity.to_string());
auto data = make_tuple(owner, issue_quantity, string("issue LP token"));
action(permission_level[_self, "active"_n], name(LP_TOKEN_CONTRACT), "issue"_n, data).send();

// update pair liquidity
_pairs.modify(p_itr, same_payer, [&](auto &a) {
    a.liquidity_token = safemath::add(a.liquidity_token, liquidity);
});

uint64_t balance0 = safemath::add(reserve0, amount0);
uint64_t balance1 = safemath::add(reserve1, amount1);
update(pair_id, balance0, balance1, reserve0, reserve1);

// refund
if (amount0_refund > 0) {
    inline_transfer(p_itr->token0.contract, _self, owner, asset(amount0_refund, p_itr->token0.symbol), string("Defibox:
deposit refund"));
    sub_currency_balance(p_itr->token0.contract, asset(amount0_refund, p_itr->token0.symbol));
}
if (amount1_refund > 0) {
    inline_transfer(p_itr->token1.contract, _self, owner, asset(amount1_refund, p_itr->token1.symbol), string("Defibox:
deposit refund"));
    sub_currency_balance(p_itr->token1.contract, asset(amount1_refund, p_itr->token1.symbol));
}
```

```
// last
orders.erase(o_itr);

// depositlog
auto logdata = make_tuple(pair_id, owner, liquidity, asset(amount0, p_itr->token0.symbol), asset(amount1,
p_itr->token1.symbol), p_itr->liquidity_token, p_itr->reserve0, p_itr->reserve1);
action(permission_level{ _self, "active"_n }, _self, "liquiditylog"_n, logdata).send();
}

extended_asset dbswap::swap(uint64_t pair_id, name from, name contract, asset quantity) {
    auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");
    bool is_token0 = contract == p_itr->token0.contract && quantity.symbol == p_itr->token0.symbol;
    bool is_token1 = contract == p_itr->token1.contract && quantity.symbol == p_itr->token1.symbol;
    check(is_token0 || is_token1, "Contract or Symbol error");

    // check balance
    check_currency_balance(p_itr->token0, p_itr->token1);

    uint64_t amount_in = quantity.amount;
    // fees: trade_fee and protocol_fee always less than 256
    uint64_t trade_fee = (uint128_t)amount_in * _config.trade_fee / 10000;
    uint64_t protocol_fee = (uint128_t)amount_in * _config.protocol_fee / 10000;

    check(trade_fee + protocol_fee > 0, "Swap amount too small");
    if (protocol_fee > 0) {
        amount_in -= protocol_fee;
        inline_transfer(contract, _self, _config.fee_account, asset(protocol_fee, quantity.symbol), string("Defibox: swap
protocol fee"));
        sub_currency_balance(contract, asset(protocol_fee, quantity.symbol));
    }

    uint64_t reserve0 = p_itr->reserve0.amount;
    uint64_t reserve1 = p_itr->reserve1.amount;
    uint64_t amount_out = 0;
    extended_asset output;
    uint64_t balance0;
    uint64_t balance1;
    if (is_token0) {
        amount_out = get_output_amount(amount_in, reserve0, reserve1);
        check(amount_out >= 0, "Insufficient output amount");
        output.contract = p_itr->token1.contract;
```

```
output.quantity = asset(amount_out, p_itr->token1.symbol);
balance0 = safemath::add(reserve0, amount_in);
balance1 = safemath::sub(reserve1, amount_out);
} else {
    amount_out = get_output_amount(amount_in, reserve1, reserve0);
    check(amount_out >= 0, "Insufficient output amount");
    output.contract = p_itr->token0.contract;
    output.quantity = asset(amount_out, p_itr->token0.symbol);
    balance0 = safemath::sub(reserve0, amount_out);
    balance1 = safemath::add(reserve1, amount_in);
}
update(pair_id, balance0, balance1, reserve0, reserve1);

// swaplog
double amount_in_f = amount_in * 1.0 / pow(10, quantity.symbol.precision());
double amount_out_f = amount_out * 1.0 / pow(10, output.quantity.symbol.precision());
double trade_price = amount_out_f / amount_in_f;
auto data = make_tuple(pair_id, from, quantity, output.quantity, asset(trade_fee + protocol_fee, quantity.symbol),
trade_price, p_itr->reserve0, p_itr->reserve1);
action(permission_level{self, "active"_n}, self, "swaplog"_n, data).send();
return output;
}

void dbswap::update(uint64_t pair_id, uint64_t balance0, uint64_t balance1, uint64_t reserve0, uint64_t reserve1) {
    check(balance0 >= 0 && balance1 >= 0, "Update balances error");
    auto p_itr = _pairs.require_find(pair_id, "Pair does not exist.");
    uint64_t time_elapsed = current_time_point().sec_since_epoch() - p_itr->block_time_last.sec_since_epoch();
    _pairs.modify(p_itr, same_payer, [&](auto &a) {
        a.reserve0.amount = balance0;
        a.reserve1.amount = balance1;
        if (time_elapsed > 0 && reserve0 != 0 && reserve1 != 0) {
            double reserve0_f = 1.0 * balance0 / pow(10, p_itr->token0.symbol.precision());
            double reserve1_f = 1.0 * balance1 / pow(10, p_itr->token1.symbol.precision());
            auto price0 = reserve1_f / reserve0_f;
            auto price1 = reserve0_f / reserve1_f;
            a.price0_last = price0;
            a.price1_last = price1;
            a.price0_cumulative_last += price0 * time_elapsed;
            a.price1_cumulative_last += price1 * time_elapsed;
        }
        a.block_time_last = current_time_point();
    });
};
```

```
}

uint64_t dbswap::get_output_amount(uint64_t input_amount, uint64_t input_reserve, uint64_t output_reserve) {
    check(input_amount > 0, "Invalid input amount");
    check(input_reserve > 0 && output_reserve > 0, "Insufficient reserve");
    uint64_t trade_fee = (uint128_t)input_amount * _config.trade_fee / 10000;
    uint128_t input_amount_with_fee = input_amount - trade_fee;
    uint128_t numerator = input_amount_with_fee * output_reserve;
    uint128_t denominator = input_reserve + input_amount_with_fee;
    uint128_t amount_temp = numerator / denominator;
    check(amount_temp < asset::max_amount, "Swap amount too large");
    uint64_t output_amount = amount_temp;
    check(output_amount > 0, "Invalid output amount");
    return output_amount;
}

uint64_t dbswap::get_pair_id() {
    _config.pair_id++;
    _configs.set(_config, _self);
    return _config.pair_id;
}

void dbswap::createlog(uint64_t pair_id, name creator, token token0, token token1) {
    require_auth(_self);
}

void dbswap::liquiditylog(uint64_t pair_id, name owner, uint64_t liquidity, asset quantity0, asset quantity1, uint64_t
total_liquidity, asset reserve0, asset reserve1) {
    require_auth(_self);
}

void dbswap::swaplog(uint64_t pair_id, name owner, asset quantity_in, asset quantity_out, asset fee, double trade_price,
asset reserve0, asset reserve1) {
    require_auth(_self);
}

extern "C" {
    void apply( uint64_t receiver, uint64_t code, uint64_t action ) {
        if (receiver == code) {
            switch( action ) {
                EOSIO_DISPATCH_HELPER(dbswap,
(createpair)(removepair)(deposit)(cancel)(updatestatus)(updatefees)(createlog)(liquiditylog)(swaplog))
            }
        } else {
            if (action == name("transfer").value) {
                dbswap inst( name(receiver), name(code), datastream<const char*>(nullptr, 0) );
                const auto t = unpack_action_data<transfer_args>();
            }
        }
    }
}
```

```
        inst.ontransfer(t.from, t.to, t.quantity, t.memo, name(code));
    }
}
}
```

swap/include/dbswap.hpp

```
#include <utils.hpp>
#include <eosio/singleton.hpp>
#include <safemath.hpp>
#include <math.h>

CONTRACT dbswap : public contract {
public:
    using contract::contract;

    static const uint64_t MINIMUM_LIQUIDITY = 10000;

    dbswap(name receiver, name code, datastream<const char *> ds): contract(receiver, code, ds),
        _pairs(_self, _self.value),
        _configs(_self, _self.value) {
        if (_configs.exists()) {
            _config = _configs.get();
        } else {
            config default_config = { 0, 0, 20, 0, "fees.defi"_n };
            _config = default_config;
        }
    }

    ACTION updatestatus(uint8_t status);
    ACTION updatefees(uint8_t trade_fee, uint8_t protocol_fee, name fee_account);

    ACTION createpair(name creator, token token0, token token1);
    ACTION removepair(uint64_t pair_id);

    ACTION deposit(name owner, uint64_t pair_id);
    ACTION cancel(name owner, uint64_t pair_id);

    ACTION createlog(uint64_t pair_id, name creator, token token0, token token1);
    ACTION liquiditylog(uint64_t pair_id, name owner, uint64_t liquidity, asset quantity0, asset quantity1, uint64_t
total_liquidity, asset reserve0, asset reserve1);
```

```
ACTION swaplog(uint64_t pair_id, name owner, asset quantity_in, asset quantity_out, asset fee, double trade_price,
asset reserve0, asset reserve1);
```

```
void ontransfer(name from, name to, asset quantity, string memo, name code);
```

```
void do_swap(vector<uint64_t> ids, name from, name contract, asset quantity, uint64_t min_out);
```

```
void do_deposit(uint64_t pair_id, name owner, name contract, asset quantity);
```

```
private:
```

```
TABLE pair {
```

```
    uint64_t id;
```

```
    token token0;
```

```
    token token1;
```

```
    asset reserve0;
```

```
    asset reserve1;
```

```
    uint64_t liquidity_token;
```

```
    double price0_last;
```

```
    double price1_last;
```

```
    uint64_t price0_cumulative_last;
```

```
    uint64_t price1_cumulative_last;
```

```
    time_point_sec block_time_last;
```

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

```
    uint128_t hash() const { return uint128_hash(token0.to_string() + "-" + token1.to_string()); }
```

```
    EOSLIB_SERIALIZE(pair,
```

```
(id)(token0)(token1)(reserve0)(reserve1)(liquidity_token)(price0_last)(price1_last)(price0_cumulative_last)(price1_cumulative_last)(block_time_last))
};
```

```
TABLE order {
```

```
    name owner;
```

```
    asset quantity0;
```

```
    asset quantity1;
```

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

```
    EOSLIB_SERIALIZE(order, (owner)(quantity0)(quantity1))
```

```
};
```

```
TABLE config {
```

```
    uint8_t status;
```

```
    uint64_t pair_id;
```

```
    uint8_t trade_fee;
```

```
    uint8_t protocol_fee;
```

```
    name fee_account;
```



```
EOSLIB_SERIALIZE(config, (status)(pair_id)(trade_fee)(protocol_fee)(fee_account))
};

TABLE contract_balance {
    asset    balance;
    uint64_t primary_key() const { return balance.symbol.code().raw(); }
    EOSLIB_SERIALIZE(contract_balance, (balance))
};

typedef multi_index<"pairs"_n, pair, indexed_by<"byhash"_n, const_mem_fun<pair, uint128_t, &pair::hash>>>
pairs_index;

typedef multi_index<"orders"_n, order> orders_index;
typedef multi_index<"balances"_n, contract_balance> balances_index;
typedef singleton<"config"_n, config> config_index;

extended_asset swap(uint64_t pair_id, name from, name contract, asset quantity);
void update(uint64_t pair_id, uint64_t balance0, uint64_t balance1, uint64_t reserve0, uint64_t reserve1);
void withdraw(name owner, uint64_t pair_id, asset quantity);

uint64_t get_output_amount(uint64_t amount_in, uint64_t reserve_in, uint64_t reserve_out);
uint64_t get_pair_id();

config_index _configs;
config _config;
pairs_index _pairs;

// 检查币种余额
void check_currency_balance(const token &token0) {
    asset balance = get_balance(token0.contract, _self, token0.symbol);
    balances_index balances(_self, token0.contract.value);
    auto itr = balances.require_find(token0.symbol.code().raw(), "Balance check: symbol not exists");
    check(balance >= itr->balance, "Balance check: exception");
}

void check_currency_balance(const token &token0, const token &token1) {
    check_currency_balance(token0);
    check_currency_balance(token1);
}

// 入账时增加币种余额
void add_currency_balance(const name &contract, const asset &quantity) {
```

```
balances_index balances(_self, contract.value);
auto itr = balances.find(quantity.symbol.code().raw());
if (itr == balances.end()) {
    balances.emplace(_self, [&]( auto& s) {
        s.balance = quantity;
    });
} else {
    balances.modify(itr, same_payer, [&]( auto& s) {
        s.balance += quantity;
    });
}
}

void update_currency_balance(const name &contract, const asset &quantity) {
    balances_index balances(_self, contract.value);
    auto itr = balances.find(quantity.symbol.code().raw());
    if (itr == balances.end()) {
        balances.emplace(_self, [&]( auto& s) {
            s.balance = quantity;
        });
    } else {
        balances.modify(itr, same_payer, [&]( auto& s) {
            s.balance = quantity;
        });
    }
}

// 出账时减少币种余额
void sub_currency_balance(const name &contract, const asset &quantity) {
    balances_index balances(_self, contract.value);
    auto itr = balances.find(quantity.symbol.code().raw());
    if (itr != balances.end()) {
        if (itr->balance == quantity) {
            balances.erase(itr);
        } else {
            balances.modify(itr, same_payer, [&]( auto& s) {
                s.balance -= quantity;
            });
        }
    } else {
        check(false, "Cannot find the quantity symbol");
    }
}
```

```
    }  
};
```

swap/include/defines.hpp

```
#include <eosio/eosio.hpp>  
#include <eosio/asset.hpp>  
#include <string>  
using namespace eosio;  
using namespace std;  
#define ASSIST_ACCOUNT "newdexassist"  
#define ADMIN_ACCOUNT "admin.defi"  
#define MINE_ACCOUNT "mine2.defi"  
#define LPTOKEN_CONTRACT "lptoken.defi"  
struct transfer_args {  
    name from;  
    name to;  
    asset quantity;  
    string memo;  
};  
struct account_balance {  
    asset balance;  
    uint64_t primary_key() const { return balance.symbol.code().raw(); }  
};  
struct currency_stats {  
    asset supply;  
    asset max_supply;  
    name issuer;  
    uint64_t primary_key() const { return supply.symbol.code().raw(); }  
};  
struct token {  
    name contract;  
    symbol symbol;  
    string to_string() const {  
        return contract.to_string() + "-" + symbol.code().to_string();  
    };  
    EOSLIB_SERIALIZE(token, (contract)(symbol))  
};  
bool operator==(token a, token b) { return a.contract == b.contract && a.symbol == b.symbol; } bool operator!=(token a, token  
b) { return !(a == b); }  
typedef eosio::multi_index<"accounts"_n, account_balance > accounts; typedef eosio::multi_index<"stat"_n, currency_stats>  
stats;
```

Swap/include/safemath.hpp

```
namespace safemath {  
    using std::string;  
    uint64_t add(const uint64_t a, const uint64_t b) {  
        uint64_t c = a + b;  
        check(c >= a, "add-overflow"); return c;  
    }  
  
    uint64_t sub(const uint64_t a, const uint64_t b) {  
        uint64_t c = a - b;  
        check(c <= a, "sub-overflow"); return c;  
    }  
  
    uint64_t mul(const uint64_t a, const uint64_t b) {  
        uint64_t c = a * b;  
        check(b == 0 || c / b == a, "mul-overflow"); return c;  
    }  
  
    uint64_t div(const uint64_t a, const uint64_t b) {  
        check(b > 0, "divide by zero");  
        return a / b;  
    }  
} // namespace safemath
```

swap/include/utils.hpp

```
#include <defines.hpp>  
  
asset get_supply(const name &token_contract, const symbol_code &sym_code) {  
    stats statstable(token_contract, sym_code.raw());  
    std::string err_msg = "invalid token contract: ";  
    err_msg.append(token_contract.to_string());  
    const auto &st = statstable.require_find(sym_code.raw(), err_msg.c_str());  
    return st->supply;  
}  
  
asset get_balance(const name &token_contract, const name &owner, const symbol &sym) {  
    asset ret = asset( 0, sym );  
    accounts accounts_table( token_contract, owner.value );  
    auto accounts_it = accounts_table.find( sym.code().raw() );
```

```
if ( accounts_it != accounts_table.end() ) {
    ret = accounts_it->balance;
}
return ret;
}

void inline_transfer(name contract, name from, name to, asset quantity, string memo) {
    auto data = make_tuple(from, to, quantity, memo);
    action(permission_level{from, "active"_n}, contract, "transfer"_n, data).send();
}

uint64_t string_to_uint64(std::string const& value) {
    uint64_t result = 0;
    char const* p = value.c_str();
    char const* q = p + value.size();
    while (p < q) {
        result = (result << 1) + (result << 3) + *(p++) - '0';
    }
    return result;
}

uint128_t uint128_hash(const string& hash) {
    return std::hash<string>{}(hash);
}

vector<string> split(const string& str, const string& delim) {
    vector<string> strs;
    size_t prev = 0, pos = 0;
    do {
        pos = str.find(delim, prev);
        if (pos == string::npos) pos = str.length();
        string token = str.substr(prev, pos-prev);
        if (!token.empty()) strs.push_back(token);
        prev = pos + delim.length();
    } while (pos < str.length() && prev < str.length());
    return strs;
}

symbol id_to_symbol(uint64_t number) {
    string str = "";
    while (number > 0) {
        int m = number % 26;
        if(m == 0) m = 26;
        str = (char)(m + 64) + str;
    }
}
```

```
        number = (number - m) / 26;
    }
    str = "BOX" + str;
    return symbol(str.c_str(), 0);
}
uint64_t symbol_to_id(string s) {
    uint64_t ans = 0;
    for (int i = 0, j = 1; i < s.size(); i++, j *= 26) {
        ans += (int)(s[s.size() - i - 1] - 64) * j;
    }
    return ans;
}
```

lptoken/src/lptoken.cpp

```
#include <lptoken.hpp>
#include <safemath.hpp>

void lptoken::create(const name &issuer, const asset &maximum_supply) {
    require_auth(SWAP_CONTRACT);

    check(_pstat.status == 0, "LP deos not enable now");
    check(issuer == SWAP_CONTRACT, "wrong issuer");

    auto sym = maximum_supply.symbol;
    check(sym.is_valid(), "invalid symbol name");
    check(maximum_supply.is_valid(), "invalid supply");
    check(maximum_supply.amount > 0, "max-supply must be positive");

    stats statstable(get_self(), sym.code().raw());
    auto existing = statstable.find(sym.code().raw());
    check(existing == statstable.end(), "token with symbol already exists");

    statstable.emplace(get_self(), [&](auto &s) {
        s.supply.symbol = maximum_supply.symbol;
        s.max_supply = maximum_supply;
        s.issuer = issuer;
    });
}

void lptoken::issue(const name &to, const asset &quantity, const string &memo) {
```

```
check(_pstat.status == 0, "LP deos not enable now");
auto sym = quantity.symbol;
check(sym.is_valid(), "invalid symbol name");
check(memo.size() <= 256, "memo has more than 256 bytes");

stats statstable(_self, sym.code().raw());
auto existing = statstable.find(sym.code().raw());
check(existing != statstable.end(), "token with symbol does not exist, create token before issue");
const auto &st = *existing;

require_auth(st.issuer);
check(quantity.is_valid(), "invalid quantity");
check(quantity.amount > 0, "must issue positive quantity");

check(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
check(quantity.amount <= st.max_supply.amount - st.supply.amount, "quantity exceeds available supply");

statstable.modify(st, same_payer, [&](auto &s) {
    s.supply += quantity;
});

add_balance(st.issuer, quantity, st.issuer);

if (to != st.issuer) {
    SEND_INLINE_ACTION(*this, transfer, {{st.issuer, "active"_n}}, {st.issuer, to, quantity, memo});
}
}

void lptoken::retire(const asset &quantity, const string &memo) {
    auto sym = quantity.symbol;
    check(sym.is_valid(), "invalid symbol name");
    check(memo.size() <= 256, "memo has more than 256 bytes");

    stats statstable(get_self(), sym.code().raw());
    auto existing = statstable.find(sym.code().raw());
    check(existing != statstable.end(), "token with symbol does not exist");
    const auto &st = *existing;

    require_auth(st.issuer);
    check(quantity.is_valid(), "invalid quantity");
    check(quantity.amount > 0, "must retire positive quantity");
```

```
check(quantity.symbol == st.supply.symbol, "symbol precision mismatch");

statstable.modify(st, same_payer, [&](auto &s) {
    s.supply -= quantity;
});

sub_balance(st.issuer, quantity);
}

void lptoken::transfer(const name &from, const name &to, const asset &quantity, const string &memo) {
    check(_pstat.status == 0, "LP deos not enable now");
    check(from != to, "cannot transfer to self");
    require_auth(from);
    check(is_account(to), "to account does not exist");
    auto sym = quantity.symbol.code();
    stats statstable(get_self(), sym.raw());
    const auto &st = statstable.get(sym.raw());

    require_recipient(from);
    require_recipient(to);

    check(quantity.is_valid(), "invalid quantity");
    check(quantity.amount > 0, "must transfer positive quantity");
    check(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
    check(memo.size() <= 256, "memo has more than 256 bytes");

    auto payer = has_auth(to) ? to : from;

    sub_balance(from, quantity);
    add_balance(to, quantity, payer);
}

void lptoken::sub_balance(const name &owner, const asset &value) {
    accounts from_acnts(get_self(), owner.value);

    const auto &from = from_acnts.get(value.symbol.code().raw(), "no balance object found");

    auto balance = from.balance.amount;
    check(balance >= value.amount, "overdrawn balance");
}
```



```
if (balance - value.amount == 0) {
    from_acnts.erase(from);
} else {
    from_acnts.modify(from, owner, [&](auto &a) {
        a.balance -= value;
    });
}

if (owner != _self && owner != SWAP_CONTRACT) {
    auto data = std::make_tuple(value.symbol.code(), owner, balance, balance - value.amount);
    action(permission_level{_self, "active"_n}, _self, "tokenchange"_n, data).send();
}
}

void lptoken::add_balance(const name &owner, const asset &value, const name &ram_payer) {
    accounts to_acnts(get_self(), owner.value);
    auto to = to_acnts.find(value.symbol.code().raw());
    uint64_t pre_amount = 0;
    if (to == to_acnts.end()) {
        to_acnts.emplace(ram_payer, [&](auto &a) {
            a.balance = value;
        });
    } else {
        pre_amount = to->balance.amount;
        to_acnts.modify(to, same_payer, [&](auto &a) {
            a.balance += value;
        });
    }

    if (owner != _self && owner != SWAP_CONTRACT) {
        auto data = std::make_tuple(value.symbol.code(), owner, pre_amount, pre_amount + value.amount);
        action(permission_level{_self, "active"_n}, _self, "tokenchange"_n, data).send();
    }
}

void lptoken::open(const name &owner, const symbol &symbol, const name &ram_payer) {
    check(_pstat.status == 0, "LP deos not enable now");
    require_auth(ram_payer);

    check(is_account(owner), "owner account does not exist");
}
```

```
auto sym_code_raw = symbol.code().raw();
stats statstable(get_self(), sym_code_raw);
const auto &st = statstable.get(sym_code_raw, "symbol does not exist");
check(st.supply.symbol == symbol, "symbol precision mismatch");

accounts acnts(get_self(), owner.value);
auto it = acnts.find(sym_code_raw);
if (it == acnts.end()) {
    acnts.emplace(ram_payer, [&](auto &a) {
        a.balance = asset{0, symbol};
    });
}
}

void lptoken::close(const name &owner, const symbol &symbol) {
    check(_pstat.status == 0, "LP deos not enable now");
    require_auth(owner);
    accounts acnts(get_self(), owner.value);
    auto it = acnts.find(symbol.code().raw());
    check(it != acnts.end(), "Balance row already deleted or never existed. Action won't have any effect.");
    check(it->balance.amount == 0, "Cannot close because the balance is not zero.");
    acnts.erase(it);
}

void lptoken::destroy(symbol_code code) {
    check(_pstat.status == 0, "LP deos not enable now");
    require_auth(SWAP_CONTRACT);
    stats statstable(get_self(), code.raw());
    auto itr = statstable.require_find(code.raw(), "token does not exists");
    check(itr->supply.amount == 0, "Can not destroy non-empty token");
    statstable.erase(itr);
}

void lptoken::createpool(symbol_code code, uint32_t weight) {
    require_auth(ADMIN_ACCOUNT);
    check(_pstat.status == 0, "LP deos not enable now");

    check(weight > 0, "Weight need greater than 0");

    stats stattabl(_self, code.raw());
    stattabl.require_find(code.raw(), "Code does not exists");
}
```

```
auto p_itr = _pool_table.find(code.raw());
check(p_itr == _pool_table.end(), "Pool exists");

update_pool(code);

// create pool
auto current_time = current_time_point().sec_since_epoch();
_pool_table.emplace(_self, [&](auto &a) {
    a.code = code;
    a.weight = weight;
    a.acc_box_per_share = 0;
    a.last_reward_time = current_time;
});

_pstat.total_weight += weight;
_poolstat_table.set(_pstat, _self);
}

void lptoken::modifypool(symbol_code code, uint32_t weight) {
    require_auth(ADMIN_ACCOUNT);
    check(_pstat.status == 0, "LP deos not enable now");

    update_pool(code);

    auto p_itr = _pool_table.require_find(code.raw(), "Pool does not exists");

    auto prev_weight = p_itr->weight;
    _pool_table.modify(p_itr, same_payer, [&](auto &a) {
        a.weight = weight;
    });

    int64_t weight_update = weight - prev_weight;
    _pstat.total_weight += weight_update;

    check(_pstat.total_weight <= BOX_PER_SECOND_SEP_CNT * BOX_PER_SECOND_SEP_CNT_UNIT * 2, "The total weight
exceeds the maximum value");

    _poolstat_table.set(_pstat, _self);
}
```

```
void lptoken::removepool(symbol_code code, const name &offset, const uint16_t limit) {
    require_auth(ADMIN_ACCOUNT);
    check(_pstat.status == 0, "LP deos not enable now");

    auto p_itr = _pool_table.require_find(code.raw(), "Pool is not exists");
    check(p_itr->weight == 0, "Only pools with a weight of 0 can be deleted");

    // delete userinfo
    userinfo userinfo_table(_self, code.raw());
    auto ur_itr = offset == name("") ? userinfo_table.begin() : userinfo_table.find(offset.value);
    int i = 0;
    while (ur_itr != userinfo_table.end()) {
        userinfo_table.modify(ur_itr, same_payer, [&](auto &a) {
            a.debt = 0;
        });
        ur_itr++;
        if (++i == limit) {
            break;
        }
    }
    if (ur_itr == userinfo_table.end()) {
        _pool_table.erase(p_itr);
    }
}

void lptoken::addnotify(symbol_code code, const name &partner, uint32_t start_time, uint32_t end_time) {
    require_auth(ADMIN_ACCOUNT);
    notifylist notifylist_table(_self, code.raw());
    auto itr = notifylist_table.find(partner.value);
    check(itr == notifylist_table.end(), "partner in notifylist");
    notifylist_table.emplace(_self, [&](auto &a) {
        a.partner = partner;
        a.start_time = start_time;
        a.end_time = end_time;
    });
}

void lptoken::removenotify(symbol_code code, const name &partner) {
    require_auth(ADMIN_ACCOUNT);
    notifylist notifylist_table(_self, code.raw());
```

```
    auto itr = notifylist_table.find(partner.value);
    check(itr != notifylist_table.end(), "partner does not in notifylist");
    notifylist_table.erase(itr);
}

void lptoken::claim(const name &owner) {
    check(_pstat.status == 0, "LP deos not enable now");
    if (!has_auth(_self)) {
        require_auth(owner);
    }

    rewards reward_table(_self, _self.value);
    auto itr = reward_table.require_find(owner.value, "Reward not found");

    auto reward_amount = itr->unclaimed;
    check(reward_amount > 0, "No reward");

    // update to 0
    reward_table.modify(itr, same_payer, [&](auto &a) {
        a.unclaimed = 0;
    });

    // transfer to user
    inline_transfer(BOX_TOKEN_CONTRACT, _self, owner, asset(reward_amount, BOX_SYMBOL), "BOX reward");
}

void lptoken::update(symbol_code code, const name &owner) {
    check(_pstat.status == 0, "LP deos not enable now");
    require_auth(owner);

    accounts from_acnts(get_self(), owner.value);
    const auto &from = from_acnts.get(code.raw(), "no lp token found");

    auto balance = from.balance.amount;
    check(balance >= 0, "no lp token");

    _tokenchange(code, owner, balance, balance);
}

void lptoken::claimall(const name &owner, const symbol_code &offset, const uint16_t limit) {
```

```
check(_pstat.status == 0, "LP deos not enable now");
require_auth(owner);

accounts acnts(get_self(), owner.value);
auto itr = offset == symbol_code("") ? acnts.begin() : acnts.find(offset.raw());
uint16_t cnt = 0;
while (itr != acnts.end()) {
    auto code = itr->balance.symbol.code();
    auto p_itr = _pool_table.find(code.raw());
    if (p_itr != _pool_table.end() && p_itr->weight > 0) {
        _tokenchange(code, owner, itr->balance.amount, itr->balance.amount);
    }
    itr++;
    if (++cnt == limit) {
        // max limit = 10
        break;
    }
}
auto data2 = std::make_tuple(owner);
action(permission_level{self, "active"_n}, self, "claim"_n, data2).send();
}

void lptoken::rewardlog(symbol_code code, const name &owner, const asset &reward) {
    require_auth(self);
}

void lptoken::tokenchange(symbol_code code, const name &owner, uint64_t pre_amount, uint64_t now_amount) {
    require_auth(self);

    _tokenchange(code, owner, pre_amount, now_amount);

    // notify partners
    notifylist notifylist_table(self, code.raw());
    auto itr = notifylist_table.begin();
    while (itr != notifylist_table.end()) {
        auto now_time = current_time_point().sec_since_epoch();
        if (now_time >= itr->start_time && now_time <= itr->end_time) {
            require_recipient(itr->partner);
        }
        itr++;
    }
}
```

```
}

void lptoken::_tokenchange(symbol_code code, const name &owner, uint64_t pre_amount, uint64_t now_amount) {
    userinfo userinfo_table(_self, code.raw());
    auto ur_itr = userinfo_table.find(owner.value);
    if (ur_itr == userinfo_table.end()) {
        ur_itr = userinfo_table.emplace(_self, [&](auto &a) {
            a.owner = owner;
            a.liquidity = 0;
            a.debt = 0;
        });
    }

    uint128_t acc_box_per_share = 0;
    auto p_itr = _pool_table.find(code.raw());
    if (p_itr != _pool_table.end()) {

        update_pool(code);

        acc_box_per_share = p_itr->acc_box_per_share;

        uint128_t reward = safemath128::mul(pre_amount, acc_box_per_share) / BASE_NUMBER;
        check(reward <= asset::max_amount, "reward too large");
        uint64_t pending = (uint64_t)reward - ur_itr->debt;
        if (pending > 0) {
            // add reward to user
            add_reward(code, owner, pending);
        }
    }

    if (now_amount == 0) {
        userinfo_table.erase(ur_itr);
    } else {
        userinfo_table.modify(ur_itr, same_payer, [&](auto &a) {
            a.liquidity = now_amount;
            a.debt = now_amount * acc_box_per_share / BASE_NUMBER;
        });
    }
}

void lptoken::update_pools() {
```

```
auto itr = _pool_table.begin();
while (itr != _pool_table.end()) {
    update_pool(itr->code);
    itr++;
}

void lptoken::update_pool(symbol_code code) {
    auto p_itr = _pool_table.find(code.raw());
    if (p_itr == _pool_table.end()) {
        return;
    }
    auto current_time = current_time_point().sec_since_epoch();
    if (current_time <= p_itr->last_reward_time) {
        return;
    }
    if (p_itr->weight == 0) {
        return;
    }

    stats statstable(get_self(), p_itr->code.raw());
    auto s_itr = statstable.require_find(p_itr->code.raw(), "Symbol not found");
    auto lp_supply = s_itr->supply;
    if (lp_supply.amount == 0) {
        return;
    }

    uint64_t box_supply_per_second = BOX_SUPPLY_PER_SECOND_OLD;
    // 检查发行量
    stats boxstatstable(BOX_TOKEN_CONTRACT, BOX_SYMBOL.code().raw());
    auto box_itr = boxstatstable.require_find(BOX_SYMBOL.code().raw(), "Symbol not found");
    if (box_itr->supply.amount / 1000000 >= 1000000) {
        box_supply_per_second = BOX_SUPPLY_PER_SECOND_SEP;
    }

    uint128_t seconds = current_time - p_itr->last_reward_time;
    // if (p_itr->weight > 0 && _pstat.total_weight > 0) {
    //     box_reward = seconds * box_supply_per_second * 7 / 10 * p_itr->weight / _pstat.total_weight;
    // }
    uint64_t box_reward = seconds * box_supply_per_second * 7 / 10 * p_itr->weight / BOX_PER_SECOND_SEP_CNT_UNIT;
```



```
// issue boxes
if (current_time > p_itr->last_reward_time) {
    auto issue_amount = seconds * box_supply_per_second * p_itr->weight / BOX_PER_SECOND_SEP_CNT_UNIT;;
    auto issue_quantity = asset(issue_amount, BOX_SYMBOL);
    _pstat.last_issue_time = current_time;
    _poolstat_table.set(_pstat, _self);

    auto data = std::make_tuple(_self, issue_quantity, string("liquidity mine"));
    action(permission_level{_self, "active"_n}, BOX_TOKEN_CONTRACT, "issue"_n, data).send();
}

// update
auto acc_box_per_share = p_itr->acc_box_per_share;
_pool_table.modify(p_itr, same_payer, [&](auto &a) {
    a.acc_box_per_share = safemath128::add(acc_box_per_share, safemath128::mul(box_reward, BASE_NUMBER) /
lp_supply.amount);
    a.last_reward_time = current_time;
});
}

void lptoken::add_reward(symbol_code code, const name &owner, const uint64_t value) {
    if (value == 0) {
        return;
    }
    rewards reward_table(_self, _self.value);
    auto itr = reward_table.find(owner.value);
    if (itr == reward_table.end()) {
        reward_table.emplace(_self, [&](auto &a) {
            a.owner = owner;
            a.cumulative = value;
            a.unclaimed = value;
        });
    } else {
        reward_table.modify(itr, same_payer, [&](auto &a) {
            a.cumulative = safemath::add(a.cumulative, value);
            a.unclaimed = safemath::add(a.unclaimed, value);
        });
    }
    // printf("reward: %s %s", owner.to_string().c_str(), asset(value, BOX_SYMBOL).to_string().c_str());
    auto data = std::make_tuple(code, owner, asset(value, BOX_SYMBOL));
    action(permission_level{_self, "active"_n}, _self, "rewardlog"_n, data).send();
}
```

```
}

void lptoken::updatestatus(uint8_t status) {
    require_auth(ADMIN_ACCOUNT);

    _pstat.status = status;
    _poolstat_table.set(_pstat, _self);
}
```

lptoken/include/lptoken.hpp

```
#pragma once
#include <eosio/eosio.hpp>
#include <eosio/asset.hpp>
#include <eosio/system.hpp>
#include <eosio/singleton.hpp>

#include <string>
#include <cmath>

#include <utils.hpp>

using std::string;
using namespace eosio;

#define ADMIN_ACCOUNT name("admin.defi")
#define SWAP_CONTRACT name("swap.defi")
#define BOX_TOKEN_CONTRACT name("token.defi")
#define BOX_SUPPLY_PER_SECOND_OLD 128000 // mine1.defi
#define BOX_SUPPLY_PER_SECOND_NEW 24000 // mine1.defi
#define BOX_SYMBOL symbol("BOX", 6)
#define BASE_NUMBER 100000000 // 10^8

#define LP_ENABLE 0

class [[eosio::contract("lptoken")]] lptoken : public contract {
public:
    using contract::contract;

    lptoken(name receiver, name code, datastream<const char*> ds) : contract(receiver, code, ds),
        _pool_table(_self, _self.value),
        _poolstat_table(_self, _self.value) {
        if (_poolstat_table.exists()) {
```

```
    _pstat = _poolstat_table.get();
} else {
    _pstat.last_issue_time = 1600664400;
    _pstat.total_weight = 3330;
    _pstat.status = 1;
    _poolstat_table.set(_pstat, _self);
}
}

/**
 * Create action.
 *
 * @details Allows `issuer` account to create a token in supply of `maximum_supply`.
 * @param issuer - the account that creates the token,
 * @param maximum_supply - the maximum supply set for the token created.
 *
 * @pre Token symbol has to be valid,
 * @pre Token symbol must not be already created,
 * @pre maximum_supply has to be smaller than the maximum supply allowed by the system:  $1^{62} - 1$ .
 * @pre Maximum supply must be positive;
 *
 * If validation is successful a new entry in statstable for token symbol scope gets created.
 */
[[eosio::action]] void create(const name &issuer, const asset &maximum_supply);

/**
 * Issue action.
 *
 * @details This action issues to `to` account a `quantity` of tokens.
 *
 * @param to - the account to issue tokens to, it must be the same as the issuer,
 * @param quantity - the amount of tokens to be issued,
 * @param memo - the memo string that accompanies the token issue transaction.
 */
[[eosio::action]] void issue(const name &to, const asset &quantity, const string &memo);

/**
 * Retire action.
 *
 * @details The opposite for create action, if all validations succeed,
 * it debits the statstable.supply amount.
 *
```

```
* @param quantity - the quantity of tokens to retire,
* @param memo - the memo string to accompany the transaction.
*/

[[eosio::action]] void retire(const asset &quantity, const string &memo);

/**
 * Transfer action.
 *
 * @details Allows `from` account to transfer to `to` account the `quantity` tokens.
 * One account is debited and the other is credited with quantity tokens.
 *
 * @param from - the account to transfer from,
 * @param to - the account to be transferred to,
 * @param quantity - the quantity of tokens to be transferred,
 * @param memo - the memo string to accompany the transaction.
 */

[[eosio::action]] void transfer(const name &from, const name &to, const asset &quantity, const string &memo);

/**
 * Open action.
 *
 * @details Allows `ram_payer` to create an account `owner` with zero balance for
 * token `symbol` at the expense of `ram_payer`.
 *
 * @param owner - the account to be created,
 * @param symbol - the token to be payed with by `ram_payer`,
 * @param ram_payer - the account that supports the cost of this action.
 *
 * More information can be read [here](https://github.com/EOSIO/eosio.contracts/issues/62)
 * and [here](https://github.com/EOSIO/eosio.contracts/issues/61).
 */

[[eosio::action]] void open(const name &owner, const symbol &symbol, const name &ram_payer);

/**
 * Close action.
 *
 * @details This action is the opposite for open, it closes the account `owner`
 * for token `symbol`.
 *
 * @param owner - the owner account to execute the close action for,
 * @param symbol - the symbol of the token to execute the close action for.
 *
 * @pre The pair of owner plus symbol has to exist otherwise no action is executed,
```

```
* @pre If the pair of owner plus symbol exists, the balance has to be zero.
*/

[[eosio::action]] void close(const name &owner, const symbol &symbol);

[[eosio::action]] void destroy(symbol_code code);

[[eosio::action]] void createpool(symbol_code code, uint32_t weight);
[[eosio::action]] void modifypool(symbol_code code, uint32_t weight);
[[eosio::action]] void removepool(symbol_code code, const name &offset, const uint16_t limit);

[[eosio::action]] void updatestatus(uint8_t status);

[[eosio::action]] void update(symbol_code code, const name &owner);
[[eosio::action]] void claim(const name &owner);
[[eosio::action]] void claimall(const name &owner, const symbol_code &offset, const uint16_t limit);

[[eosio::action]] void tokenchange(symbol_code code, const name &owner, uint64_t pre_amount, uint64_t
now_amount);
[[eosio::action]] void rewardlog(symbol_code code, const name &owner, const asset &reward);
[[eosio::action]] void addnotify(symbol_code code, const name &partner, uint32_t start_time, uint32_t end_time);
[[eosio::action]] void removenotify(symbol_code code, const name &partner);

/**
 * Get supply method.
 *
 * @details Gets the supply for token `sym_code`, created by `token_contract_account` account.
 *
 * @param token_contract_account - the account to get the supply for,
 * @param sym_code - the symbol to get the supply for.
 */
static asset get_supply(const name &token_contract_account, const symbol_code &sym_code) {
    stats statstable(token_contract_account, sym_code.raw());
    const auto &st = statstable.get(sym_code.raw());
    return st.supply;
}

/**
 * Get balance method.
 *
 * @details Get the balance for a token `sym_code` created by `token_contract_account` account,
 * for account `owner`.
 */
```

```
* @param token_contract_account - the token creator account,
* @param owner - the account for which the token balance is returned,
* @param sym_code - the token for which the balance is returned.
*/
static asset get_balance(const name &token_contract_account, const name &owner, const symbol_code &sym_code) {
    accounts accountstable(token_contract_account, owner.value);
    const auto &ac = accountstable.get(sym_code.raw());
    return ac.balance;
}
```

private:

```
struct [[eosio::table]] account {
    asset balance;
    uint64_t primary_key() const { return balance.symbol.code().raw(); }
};
```

```
struct [[eosio::table]] currency_stats {
    asset supply;
    asset max_supply;
    name issuer;
    uint64_t primary_key() const { return supply.symbol.code().raw(); }
};
```

```
struct [[eosio::table]] user_info {
    name owner;
    uint64_t debt;
    uint64_t liquidity;
    uint64_t primary_key() const { return owner.value; }
};
```

```
struct [[eosio::table]] pool {
    symbol_code code;
    uint32_t weight;
    uint32_t last_reward_time;
    uint128_t acc_box_per_share;
    uint64_t primary_key() const { return code.raw(); }
};
```

```
struct [[eosio::table]] poolstat {
    uint32_t total_weight;
    uint32_t last_issue_time;
```

```
uint8_t status;

};

struct [[eosio::table]] reward {
    name owner;
    uint64_t cumulative;
    uint64_t unclaimed;
    uint64_t primary_key() const { return owner.value; }
};

struct [[eosio::table]] notifyitem {
    name partner;
    uint32_t start_time;
    uint32_t end_time;
    uint64_t primary_key() const { return partner.value; }
};

// struct [[eosio::table]] whitelist {
//     name partner;
//     uint64_t cumulative;
//     uint64_t unclaimed;
//     uint64_t primary_key() const { return owner.value; }
// };

typedef eosio::multi_index<"accounts"_n, account> accounts;
typedef eosio::multi_index<"stat"_n, currency_stats> stats;
typedef eosio::multi_index<"userinfo"_n, user_info> userinfo;
typedef eosio::multi_index<"pools"_n, pool> pools;
typedef eosio::singleton<"poolstat"_n, poolstat> poolstats;
typedef eosio::multi_index<"rewards"_n, reward> rewards;
typedef eosio::multi_index<"notifylist"_n, notifyitem> notifylist;

pools _pool_table;
poolstats _poolstat_table;
poolstat _pstat;

void sub_balance(const name &owner, const asset &value);
void add_balance(const name &owner, const asset &value, const name &ram_payer);

void update_pool(symbol_code code);
void update_pools();
```

```
void add_reward(symbol_code code, const name &owner, const uint64_t value);

void _tokenchange(symbol_code code, const name &owner, uint64_t pre_amount, uint64_t now_amount);

};
```

lptoken/include/utlis.hpp

```
#pragma once
#include <eosio/eosio.hpp>
#include <eosio/asset.hpp>
using std::string;
using namespace eosio;

void inline_transfer(name contract, name from, name to, asset quantity, string memo) {
    auto data = std::make_tuple(from, to, quantity, memo);
    action(permission_level{from, "active"_n}, contract, "transfer"_n, data).send();
}

symbol id_to_symbol(uint64_t number) {
    string str = "";
    while (number > 0) {
        int m = number % 26;
        if(m == 0) m = 26;
        str = (char)(m + 64) + str;
        number = (number - m) / 26;
    }
    str = "BOX" + str;
    return symbol(str.c_str(), 0);
}
```




官方网址

www.slowmist.com

电子邮箱

team@slowmist.com

微信公众号

