



智能合约安全审计报告



慢雾安全团队于 2021-05-26 日，收到 Defibox 基金会对 Defibox USN 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(usn.wasm)=

ac4b81cc1e260f1ffbc038c3e3770c75fc669fb70b62a1a0262ae9d6aab5839b

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：0X002105270002

审计日期：2020 年 05 月 27 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 DApp-DeFi 合约，经审计修复后，存在以下风险：

1、由于合约本身会将合约资金用于其他项目中产生额外收益，未来可能存在无法赔付的风险

以上问题经反馈后，解决方案如下：

1、目前合约资金主要用于存放在 REX 中，暂无资金丢失风险，需持续关注系统合约是否进行改动。

以下针对合约代码进行详细分析，分析写于注释处。

usn.cpp

```
#include <usn.hpp>
#include <utils.hpp>
#include <types.hpp>

void usn::handle_transfer(name from, name to, asset quantity, string memo, name code) {
    if (from == _self || to != _self) {
        return;
    }
    if (from == "dfsfoundation"_n || from == "eosio"_n || from == "eosio.rex"_n || from == ADMIN_ACCOUNT) {
        return;
    }

    vector<string> strs = utils::split(memo, ":");
    if (code == USNTOKEN_CONTRACT && quantity.symbol == USN_SYMBOL) {
        check(strs.size() == 3, "Invalid memo");
        const string& action = str[0];
        uint64_t collateral_id = strtoull(str[1].c_str(), NULL, 10);
        uint64_t rate_or_aid = strtoull(str[2].c_str(), NULL, 10);

        check(action == "bid" || action == "repay" || action == "repay2", "Invalid memo");

        auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");
        check(collateral_itr->status == 1, "The collateral is suspended");

        check_collateral_balance(collateral_itr->contract, collateral_itr->sym);

        //repay-USN 偿还利息,repay2-用抵押物偿还利息
        if (action == "repay") {
            check_available(KEY_REPAY_SWITCH);
            repay(from, collateral_id, quantity, rate_or_aid);
        } else if (action == "repay2") {
            check_available(KEY_REPAY_SWITCH);
            repay2(from, collateral_id, quantity, rate_or_aid);
        } else if (action == "bid") {
            check_available(KEY_AUCTION_SWITCH);
        }
    }
}
```

```
        bid(from, collateral_id, quantity, rate_or_aid);
    }
} else {
    check(strs.size() > 0, "Invalid memo");
    const string& action = strs[0];

    auto collateral_id = get_collateral_id(code, quantity.symbol);
    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");
    check(collateral_itr->status == 1, "The collateral is suspended");
    check(collateral_itr->contract == code && collateral_itr->sym == quantity.symbol, "Invalid collateral");

    if (action == "fix") {
        return;
    }

    check(action == "issue" || action == "deposit" || action == "depositsys", string("Invalid memo"));
    add_collateral_balance(collateral_itr->contract, quantity);

    check(quantity >= collateral_itr->min_amount, string("less than ") + collateral_itr->min_amount.to_string());
    if (action == "issue") {
        check(strs.size() == 2, "Invalid memo");
        uint64_t rate = strtoull(strs[1].c_str(), NULL, 10);

        check(rate >= collateral_itr->min_rate, string("Invalid memo, rate should >= ") + to_string(collateral_itr->min_rate));
        check(quantity >= collateral_itr->min_amount, string("less than ") + to_string(collateral_itr->min_amount.amount)
+ quantity.symbol.code().to_string());
        check_available(KEY_ISSUE_SWITCH);

        generate(from, code, quantity, rate, true);

    } else if (action == "deposit") {
        check(strs.size() == 1, "Invalid memo");
        check(quantity >= collateral_itr->min_amount, string("less than ") + to_string(collateral_itr->min_amount.amount)
+ quantity.symbol.code().to_string());
        check_available(KEY_DEPOSIT_SWITCH);

        add_deposit(from, code, quantity);
    }
}
}
```

```
ACTION usn::init() {
    require_auth(ADMIN_ACCOUNT);
    // 业务 id
    init_globals(KEY_AUCTION_ID);    // 拍卖 id
    init_globals(KEY_BID_ID);        // 抢拍 id
    init_globals(KEY_ORDER_ID);      // 订单 id
    init_globals(KEY_COLLATERAL_ID); // 抵押物 id

    // 系统开关
    init_globals(KEY_MAINTAIN_SWITCH); // 合约总开关
    init_globals(KEY_ISSUE_SWITCH);    // 发行开关
    init_globals(KEY_REPAY_SWITCH);    // 偿还开关
    init_globals(KEY_DEPOSIT_SWITCH);  // 增加保证金开关
    init_globals(KEY_WITHDRAW_SWITCH); // 减少保证金开关
    init_globals(KEY_AUCTION_SWITCH);  // 拍卖开关

    // 增加 enabletag 参数
    init_globals(KEY_TAG_SWITCH, 2);   // tag 开启状态
}

ACTION usn::setstate(name key, uint64_t val) {
    require_auth(ADMIN_ACCOUNT);
    auto itr = _globals.require_find(key.value, "unable to find key");
    _globals.modify(itr, _self, [&](auto &s) {
        s.val = val;
    });
}

ACTION usn::syncaccounts(uint32_t limit) {
    require_auth(ADMIN_ACCOUNT);

    check(_accounts.begin() != _accounts.end(), "not accounts data to synchronous");
    check(_auctions.begin() == _auctions.end(), "auctions table not empty");

    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
    auto collateral_itr = collateral_idx.find(utils::get_token_key(EOSTOKEN_CONTRACT, EOS_SYMBOL));
    check(collateral_itr != collateral_idx.end(), "not found eos collateral");

    debts debt_tbl(_self, collateral_itr->id);
    for (auto old_itr = _accounts.begin(); limit > 0 && old_itr != _accounts.end();)
```

```
if (debt_tbl.end() != debt_tbl.find(old_itr->user.value)) {
    ++old_itr;
    continue;
}

debt_tbl.emplace(_self, [&](auto &cdp) {
    cdp.user = old_itr->user;
    cdp.contract = EOSTOKEN_CONTRACT;
    cdp.pledge = old_itr->pledge;
    cdp.issue = old_itr->issue;
    cdp.risk = old_itr->risk * 10000; // 扩大到 8 位
    cdp.status = old_itr->status;
    cdp.create_time = old_itr->create_time;
    cdp.update_time = old_itr->update_time;

    for (auto records_itr = old_itr->records.begin(); records_itr != old_itr->records.end(); ++records_itr) {
        lend l = {
            .capital = records_itr->second.capital,
            .unpay_interest = records_itr->second.unpay_interest,
            .last_update = records_itr->second.last_update
        };
        cdp.records.emplace_back(records_itr->first, l);
    }
});
old_itr = _accounts.erase(old_itr);
limit --;
}
}
```

ACTION usn::createtoken(name contract, symbol sym, uint8_t status, uint64_t clear_rate, uint64_t forfeit, uint64_t interest, uint64_t min_rate, asset min_amount, asset max_amount) {

```
    require_auth(ADMIN_ACCOUNT);

    check(utils::is_valid_token(contract, sym), "token with symbol not exists");
    check(clear_rate > 0, "clear_rate should > 0");
    check(min_amount.amount > 0, "min_amount should > 0");
    check(min_amount < max_amount, "max_amount < min_amount");
    check(10000 <= min_rate && min_rate <= 100000, "min_rate out of range [100% ~ 1000%]");
    check(sym == min_amount.symbol, "min_amount symbol inconsistent");
    check(status == 0 || status == 1, "status invalid");
```

```
auto id = get_collateral_id(contract, sym, false);
check(id == 0, "collateral has exist");

id = next(KEY_COLLATERAL_ID);
_collaterals.emplace(_self, [&](auto& c) {
    c.id = id;
    c.contract = contract;
    c.sym = sym;
    c.status = status;
    c.clear_rate = clear_rate;
    c.forfeit = forfeit;
    c.interest = interest;
    c.min_rate = min_rate;
    c.last_price = 0;

    c.min_amount = min_amount;
    c.max_amount = max_amount;
    c.balance = asset(0, sym);
    c.total_balance = asset(0, sym);

    if (contract == EOSTOKEN_CONTRACT && sym == EOS_SYMBOL) {
        c.balance.amount = _stat.balance.amount;
        c.total_balance.amount = _stat.total.amount;
    }
});

action(
    permission_level{_self, "active"_n},
    _self,
    "createlog"_n,
    make_tuple(id, contract, sym)
).send();
}

ACTION usn::setinterest(uint64_t collateral_id, uint64_t val) {
    require_auth(ADMIN_ACCOUNT);

    auto itr = _collaterals.require_find(collateral_id, "no found collateral");
    _collaterals.modify(itr, same_payer, [&](auto& c) {
        c.interest = val;
    });
};
```



```
}

ACTION usn::modifytoken(uint64_t collateral_id, uint8_t status, uint64_t clear_rate, uint64_t forfeit, uint64_t min_rate, asset
min_amount, asset max_amount) {
    require_auth(ADMIN_ACCOUNT);

    check(min_amount.amount > 0, "min_amount should > 0");
    check(max_amount > min_amount, "max_amount < min_amount");
    check(10000 <= min_rate && min_rate <= 100000, "min_rate out of range [100% ~ 1000%]");

    auto collateral_itr = _collaterals.require_find(collateral_id, "no found collateral");
    check(collateral_itr != _collaterals.end(), "not found collateral");
    check(collateral_itr->sym == min_amount.symbol, "min_amount symbol inconsistent");
    check(status == 0 || status == 1, "status invalid");

    _collaterals.modify(collateral_itr, same_payer, [&](auto& c) {
        c.status = status;
        c.clear_rate = clear_rate;
        c.forfeit = forfeit;
        c.min_rate = min_rate;
        c.min_amount = min_amount;
        c.max_amount = max_amount;
    });
}

ACTION usn::removetoken(uint64_t id) {
    require_auth(ADMIN_ACCOUNT);
    auto collateral_itr = _collaterals.require_find(id, "not found collateral");

    debts debt_tbl(_self, id);
    check(debt_tbl.begin() == debt_tbl.end(), "this collateral is kept in a debt vault, cannot remove");

    auto auction_idx = _auctions.get_index<"bycollateral"_n>();
    check(auction_idx.lower_bound(id) == auction_idx.upper_bound(id), "this collateral is kept in a auction vault, cannot
remove");

    _collaterals.erase(collateral_itr);
}

// 下调抵押率 取回押金 or 生成新的 USN
//SlowMist// 继续铸币或赎回 EOS
ACTION usn::adjust(name owner, uint64_t collateral_id, uint64_t rate, bool issue) {
```

```
require_auth(owner);

check_available(KEY_WITHDRAW_SWITCH);
check_available(KEY_ISSUE_SWITCH);

static const time_point_sec now{current_time_point().sec_since_epoch()};

auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");

check(rate >= collateral_itr->min_rate, string("rate should >= ") + to_string(collateral_itr->min_rate));
uint64_t price = get_price(collateral_itr);
check_price(collateral_itr, price);

debts debt_tbl(_self, collateral_id);
auto debt_itr = debt_tbl.find(owner.value);
check(debt_itr != debt_tbl.end(), "no account found");

// 计算债仓中 所有 利息
uint64_t total_interest = calc_debt_interest(debt_itr, collateral_itr->interest);

// 计算调整至目标质押率, 所需的保证金数量
auto pledge_quantity = asset(0, collateral_itr->sym);
auto refund = asset(0, collateral_itr->sym);

// 可提现余额 (增加利息再计算)
auto total_usn_balance = asset(debt_itr->issue.amount + total_interest, USN_SYMBOL);
pledge_quantity.amount = calc_pledge_amount2(total_usn_balance, rate, price, collateral_itr->sym);

bool status = true;
string result_str;

check(debt_itr->pledge > pledge_quantity, "adjust failure, deposit less than min rate");
result_str = string("adjust success, withdraw deposit fund");
refund = debt_itr->pledge - pledge_quantity;
check(refund.amount >= 0, "error refund");

// 重新计算爆仓价格, 更新数据库
auto risk_price = calc_risk_price(total_usn_balance, collateral_itr->contract, pledge_quantity, collateral_itr->clear_rate);
debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
    cdp.pledge = pledge_quantity;
    cdp.risk = risk_price;
```

```
        cdp.update_time = now;
    });

    if (refund.amount > 0) {
        if (!issue) {
            transfer_to(collateral_itr, owner, refund, string("withdraw deposit from cdp"));
        } else {
            generate(owner, collateral_itr->contract, refund, rate, false);
            result_str = string("adjust success, issue new USN");
        }
    }

    if (!issue) {
        action(
            permission_level{self, "active"_n},
            self,
            name("adjustlog"),
            make_tuple(owner, collateral_id, collateral_itr->contract, rate, refund, result_str, status, debt_itr->pledge,
            debt_itr->issue, price, now)
        ).send();
    }
}

// void usn::withdraw(name owner, name contract, asset quantity) {
//     check(false, "this action not allow to call");
//     require_auth(owner);

//     check_available(KEY_WITHDRAW_SWITCH);
//     check(quantity.amount > 0, "wrong amount");

//     static const time_point_sec now{current_time_point().sec_since_epoch()};

//     auto collateral_id = get_collateral_id(contract, quantity.symbol);
//     auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");

//     auto debt_id = get_debt_id(owner, collateral_id);
//     auto debt_itr = debt_tbl.require_find(debt_id, "no account found");
//     check(quantity <= debt_itr->pledge, "overdrawn balance ");

//     uint64_t price = get_price(collateral_itr);
//     check_price(collateral_itr, price);
```

```
// // 计算债仓中 所有 利息
// uint64_t total_interest = calc_debt_interest(debt_itr, collateral_itr->interest);

// string result_str;
// bool status = true;
// if (quantity <= debt_itr->pledge) {
//     asset new_collateral_quantity = debt_itr->pledge - quantity;

//     uint64_t after_rate;
//     if (debt_itr->issue.amount == 0) {
//         after_rate = INT_MAX; // 还完 USN 相当于无限抵押率
//     } else {
//         after_rate = calc_clear_rate(debt_itr->issue, new_collateral_quantity, price);
//     }

//     printf("after rate: %, min rate: %\n", after_rate, collateral_itr->min_rate);
//     if (after_rate >= collateral_itr->min_rate) {
//         uint128_t new_risk_price = 0;
//         if (new_collateral_quantity.amount > 0) {
//             new_risk_price = calc_risk_price(debt_itr->issue, contract, new_collateral_quantity,
collateral_itr->clear_rate);
//         }

//         transfer_to(collateral_itr, owner, quantity, string("withdraw succee"));

//         debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
//             cdp.pledge = new_collateral_quantity;
//             cdp.risk = new_risk_price;

//             cdp.update_time = now;
//         });

//         result_str = string("withdraw deposit success");
//     } else {
//         status = false;
//         result_str = string("withdraw error , deposit too less");
//     }
// } else {
//     status = false;
//     result_str = string("overdrawn balance");
```

```
// }

// action(
//     permission_level{ _self, "active"_n },
//     _self,
//     name("withdrawlog"),
//     make_tuple(owner, collateral_id, contract, quantity, result_str, status, debt_itr->pledge, debt_itr->issue, price, now)
// ).send();
// }

ACTION usn::clear(uint64_t collateral_id, name user) {
    require_auth(ADMIN_ACCOUNT);
    check_available(KEY_AUCTION_SWITCH);

    debts debt_tbl(_self, collateral_id);
    auto debt_itr = debt_tbl.require_find(user.value, "no account found");
    check(debt_itr->issue.amount > 0, "empty cdp");

    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");
    check(collateral_itr->status == 1, "The collateral is suspended");

    uint64_t price = get_price(collateral_itr);
    check_price(collateral_itr, price);

    // 计算债仓中 所有 利息
    uint64_t total_interest = calc_debt_interest(debt_itr, collateral_itr->interest);

    // 利息计入质押率
    auto total_usn_balance = asset(debt_itr->issue.amount + total_interest, USN_SYMBOL);
    uint64_t current_rate = calc_clear_rate(total_usn_balance, debt_itr->pledge, price);
    check(current_rate < collateral_itr->clear_rate, to_string(current_rate) + " no need to clear");

    // 爆仓单，需要先减去利息 再减去罚金，剩余进入拍卖
    // 计算利息等价的抵押物
    auto pledge_of_interest = asset(0, collateral_itr->sym);
    pledge_of_interest.amount = calc_pledge_amount(asset(total_interest, USN_SYMBOL), 1e4, price, collateral_itr->sym);

    auto forfeit = asset(0, collateral_itr->sym);
    forfeit.amount = debt_itr->pledge.amount * collateral_itr->forfeit / 1e4;

    asset remain_pledge = debt_itr->pledge - forfeit;
```

```
if (pledge_of_interest.amount <= remain_pledge.amount) {
    remain_pledge -= pledge_of_interest;
} else {
    pledge_of_interest = remain_pledge;
    remain_pledge.amount = 0;
}

printf("total interest: %, pledge of interest: %, cdp.pledge: %, forfeit: %, remain pledge: %\n",
    total_interest, pledge_of_interest, debt_itr->pledge, forfeit, remain_pledge);

if (pledge_of_interest.amount > 0) {
    transfer_to(collateral_itr, USNFEE_ACCOUNT, pledge_of_interest, string("clear interest fee"));
}

if (forfeit.amount > 0) {
    transfer_to(collateral_itr, USNFORFEIT_ACCOUNT, forfeit, string("forfeit"));
}

static const time_point_sec now{current_time_point().sec_since_epoch()};
auto aid = next(KEY_AUCTION_ID);
// 记录爆仓时间
auto data = make_tuple(aid, debt_itr->user, collateral_id, collateral_itr->contract, debt_itr->pledge, debt_itr->issue,
remain_pledge, debt_itr->issue, forfeit, pledge_of_interest, price, now);
action(permission_level{self, "active"_n}, self, name("clearresult"), data).send();

if (remain_pledge.amount > 0) {
    _auctions.emplace(self, [&](auto &a) {
        a.aid = aid;
        a.user = debt_itr->user;
        a.collateral_id = collateral_id;
        a.price = price;
        a.pledge = debt_itr->pledge;
        a.issue = debt_itr->issue;
        a.remain_pledge = remain_pledge;
        a.remain_issue = debt_itr->issue;
        a.create_time = now;
    });
}

debt_tbl.erase(debt_itr);
```

```
}

ACTION usn::calinterest(name user, uint64_t collateral_id, uint64_t rate) {
    require_auth(ADMIN_ACCOUNT);
    const time_point_sec now{current_time_point().sec_since_epoch()};

    debts debt_tbl(_self, collateral_id);
    auto debt_itr = debt_tbl.require_find(user.value, "not found cdp");

    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");
    check(collateral_itr->interest != rate, "collaterals.interest == rate");

    // 结算账户每个币种的 unpay interest
    debt_tbl.modify(debt_itr, same_payer, [&](auto &cdp) {
        for (auto record_itr = cdp.records.begin(); record_itr != cdp.records.end(); ++record_itr) {
            auto unpay_interest = get_interest(record_itr->second.capital, record_itr->first, record_itr->second.last_update,
rate);
            auto last_update = record_itr->second.last_update;

            record_itr->second.unpay_interest += unpay_interest;
            record_itr->second.last_update = now;

            action(
                permission_level{_self, "active"_n},
                _self,
                name("ratechange"),
                make_tuple(user, collateral_id, record_itr->first, last_update, record_itr->second.capital, unpay_interest, rate)
            ).send();
        }
    });
}

//SlowMist// 铸币逻辑
void usn::generate(name from, name contract, asset quantity, uint64_t rate, bool writelog) {
    static const time_point_sec now{current_time_point().sec_since_epoch()};

    auto collateral_id = get_collateral_id(contract, quantity.symbol);
    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");
    check(collateral_itr->max_amount >= collateral_itr->total_balance, "The collaterals exceeds the limit");

    uint64_t price = get_price(collateral_itr);
    check_price(collateral_itr, price);
}
```

```
auto usn_quantity = asset(0, USN_SYMBOL);

string result_str;
bool status = false;

debts debt_tbl(_self, collateral_id);
auto debt_itr = debt_tbl.find(from.value);

if (debt_itr == debt_tbl.end()) {
    // 发行出多少 USN 取决于当前抵押物价格 和用户指定的质押率 rate
    usn_quantity.amount = calc_usn_amount(quantity, rate, price);

    check(collateral_itr->clear_rate > 0, "Invalid param");
    // 强平风控位: 相当于已知 rate = 135 和 usn_quantity, 反求 price (爆仓价)
    auto risk_price = calc_risk_price(usn_quantity, contract, quantity, collateral_itr->clear_rate);

    if (usn_quantity.amount > 0) {
        status = true;

        debt_tbl.emplace(_self, [&](auto &cdp) {
            cdp.user = from;
            cdp.contract = contract;
            cdp.pledge = quantity;
            cdp.issue = usn_quantity;
            cdp.risk = risk_price;
            cdp.create_time = now;
            cdp.update_time = now;

            lend l = {
                .capital = static_cast<uint64_t>(usn_quantity.amount),
                .unpay_interest = 0,
                .last_update = now
            };
            cdp.records.emplace_back(now/*TDOO %30*/, l);
        });

        debt_itr = debt_tbl.require_find(from.value, "no account found");
    }

} else {
```



```
check(debt_itr->records.size() <= 100, "reach records limit");

asset total_quantity = debt_itr->pledge + quantity;
asset total_usn_quantity = asset(0, USN_SYMBOL);

// 这个是理应发行的总数
total_usn_quantity.amount = calc_usn_amount(total_quantity, rate, price);

check(collateral_itr->clear_rate > 0, "Invalid param");

// 计算债仓中 所有 利息
uint64_t total_interest = calc_debt_interest(debt_itr, collateral_itr->interest);

// 实际发行数量: 理应发行数量 - 已发行数量 - 利息
check(total_usn_quantity >= debt_itr->issue, "generate total usn quantity less than issued");
usn_quantity = total_usn_quantity - debt_itr->issue;
usn_quantity.amount -= total_interest;

// 重新计算发行总数量
total_usn_quantity = debt_itr->issue + usn_quantity;
auto new_risk_price = calc_risk_price(total_usn_quantity, contract, total_quantity, collateral_itr->clear_rate);

if (usn_quantity.amount > 0) {
    status = true;

    debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
        cdp.pledge = total_quantity;
        cdp.issue = total_usn_quantity;
        cdp.risk = new_risk_price;
        cdp.update_time = now;

        lend l = {
            .capital = static_cast<uint64_t>(usn_quantity.amount),
            .unpay_interest = 0,
            .last_update = now
        };
        cdp.records.emplace_back(now/*TDOO %30*/, l);
    });
}
```

```
print_f("generate: % => %, risk price: %\n", quantity, usn_quantity, debt_itr->risk);

if (status) {
    result_str = string("USN issue success");
    action(
        permission_level{USNTOKEN_CONTRACT, "active"_n},
        USNTOKEN_CONTRACT,
        name("issue"),
        make_tuple(from, usn_quantity, string("USN issue"))
    ).send();

    auto oid = next(KEY_ORDER_ID);
    if (writelog) {
        action(permission_level{self, "active"_n}, self, name("incomelog"), make_tuple(from, collateral_id, contract, oid,
quantity)).send();
    }

    action(
        permission_level{self, "active"_n},
        self,
        name("generatelog"),
        make_tuple(from, collateral_id, contract, oid, rate, quantity, usn_quantity, result_str, status, debt_itr->pledge,
debt_itr->issue, price, now)
    ).send();

} else {
    result_str = string("generate USN failure, refund deposit");
    inline_transfer(contract, self, from, quantity, result_str);
    sub_collateral_balance(contract, quantity);
}
}

void usn::add_deposit(name from, name contract, asset quantity) {
    static const time_point_sec now{current_time_point().sec_since_epoch()};

    auto collateral_id = get_collateral_id(contract, quantity.symbol);
    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");

    check(collateral_itr->max_amount >= collateral_itr->total_balance, "The collaterals exceeds the limit");

    debts debt_tbl(self, collateral_id);
```

```
auto debt_itr = debt_tbl.require_find(from.value, "no account found");

uint64_t price = get_price(collateral_itr);
check_price(collateral_itr, price);

asset total_pledge_quantity = quantity + debt_itr->pledge;
asset total_usn_quantity = debt_itr->issue;

// 强平风控位: 相当于已知 rate = 135 和 usn_quantity, 反求 price (爆仓价)
auto new_risk_price = calc_risk_price(total_usn_quantity, contract, total_pledge_quantity, collateral_itr->clear_rate);
debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
    cdp.pledge = total_pledge_quantity;
    cdp.risk = new_risk_price;
    cdp.update_time = now;
});

auto oid = next(KEY_ORDER_ID);
action(
    permission_level{_self, "active"_n},
    _self,
    name("incomelog"),
    make_tuple(from, collateral_id, contract, oid, quantity)
).send();

action(
    permission_level{_self, "active"_n},
    _self,
    name("adddepositlog"),
    make_tuple(from, collateral_id, contract, oid, quantity, string("add deposit success"), true, debt_itr->pledge,
    debt_itr->issue, price, now)
).send();
}

//SlowMist// 赎回逻辑
void usn::repay(name from, uint64_t collateral_id, asset quantity, uint64_t rate) {
    static const time_point_sec now{current_time_point().sec_since_epoch()};

    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");

    if (rate != 0) {
        check(rate >= collateral_itr->min_rate, string("Invalid memo, rate should >= ") + to_string(collateral_itr->min_rate));
    }
}
```

```
uint64_t price = get_price(collateral_itr);
check_price(collateral_itr, price);

string result_str;
asset usn_of_interest = asset(0, USN_SYMBOL);
asset total_usn_quantity = asset(0, USN_SYMBOL);
asset refund = asset(0, collateral_itr->sym);
asset total_pledge_quantity = asset(0, collateral_itr->sym);

debts debt_tbl(_self, collateral_id);
auto debt_itr = debt_tbl.require_find(from.value, "not found cdp");

debt_tbl.modify(debt_itr, same_payer, [&](auto &cdp) {
    auto repay_amount = quantity.amount;

    uint64_t total_interest = 0;
    uint64_t total_capital = 0;

    // 偿还顺序 从第一条借款开始还
    while (!cdp.records.empty() && repay_amount > 0) {

        uint64_t last_interest = get_interest(cdp.records.front().second.capital, cdp.records.front().first,
cdp.records.front().second.last_update, collateral_itr->interest);
        uint64_t total_unpay_amount = cdp.records.front().second.capital + cdp.records.front().second.unpay_interest +
last_interest;

        // 单前这笔还款的实际金额
        uint64_t this_pay;

        if (repay_amount >= total_unpay_amount) {
            this_pay = total_unpay_amount;
            repay_amount -= this_pay;
        } else {
            this_pay = repay_amount;
            repay_amount = 0;
        }

        double ratio = (double(cdp.records.front().second.capital) / total_unpay_amount);

        // 单前这笔还款的本金部分
```

```
uint64_t pay_capital = this_pay * ratio; // 按百分比还款
// 单前这笔还款的利息部分
uint64_t pay_interest = this_pay - pay_capital;

// 结息部分 按比例扣
uint64_t unpay_interest = cdp.records.front().second.unpay_interest;
uint64_t pay_unpay_interest = 0;
if (unpay_interest > 0) {
    pay_unpay_interest = (uint64_t)ceil((unpay_interest) * (double(this_pay) / total_unpay_amount));
}

// 累计总利息
total_interest += pay_interest;
total_capital += pay_capital;
check(pay_capital <= cdp.records.front().second.capital, "repay error");

action(
    permission_level{self, "active"_n},
    self,
    name("repayresult"),
    make_tuple(cdp.user, collateral_id, cdp.records.front().second.capital, pay_capital, pay_interest,
cdp.records.front().first)
).send();

if (cdp.records.front().second.capital == pay_capital) {
    cdp.records.pop_front();
} else if (cdp.records.front().second.capital > pay_capital) {
    cdp.records.front().second.capital -= pay_capital;

    if (pay_unpay_interest < cdp.records.front().second.unpay_interest) {
        cdp.records.front().second.unpay_interest -= pay_unpay_interest;
    } else {
        cdp.records.front().second.unpay_interest = 0;
    }
}
}

print_f("total interest: %, repay amount: %\n", total_interest, repay_amount);

if (total_interest > 0) {
    usn_of_interest = asset(total_interest, USN_SYMBOL);
    asset fee_quantity = asset(total_interest, USN_SYMBOL);
}
```

```
        inline_transfer(USNTOKEN_CONTRACT, _self, USNFEE_ACCOUNT, fee_quantity, string("USN interest fee"));
    }

    if (repay_amount > 0) {
        asset repay_refund_quantity = asset(repay_amount, USN_SYMBOL);
        inline_transfer(USNTOKEN_CONTRACT, _self, cdp.user, repay_refund_quantity, string("repay refund"));
    }

    // 本次还款全部扣款 包含本金和利息
    asset realpay_quantity = quantity;
    realpay_quantity.amount -= total_interest;
    realpay_quantity.amount -= repay_amount;

    check(total_capital == realpay_quantity.amount, "pay interest error");

    cdp.issue -= realpay_quantity;

    if (realpay_quantity.amount > 0) {
        inline_transfer(USNTOKEN_CONTRACT, _self, USNTOKEN_CONTRACT, realpay_quantity, string("repay retire"));

        action(
            permission_level{USNTOKEN_CONTRACT, "active"_n,
                USNTOKEN_CONTRACT,
                name("retire"),
                make_tuple(realpay_quantity, string("repay retire"))
            }.send();
    }
};

uint64_t current_rate;

if (debt_itr->issue.amount == 0) {
    // 全部偿还完毕
    result_str = string("repay success, close cdp");
    refund = debt_itr->pledge;

    //退还原有抵押物
    transfer_to(collateral_itr, debt_itr->user, refund, result_str);
    debt_tbl.erase(debt_itr);
} else {
    current_rate = calc_clear_rate(debt_itr->issue, debt_itr->pledge, price);
}
```

```
bool need_transfer_deposit = 0 < rate && rate < current_rate;
if (need_transfer_deposit) {
    // 计算调整至目标质押率, 所需的保证金数量
    auto pledge_quantity = asset(0, collateral_itr->sym);
    pledge_quantity.amount = calc_pledge_amount(debt_itr->issue, rate, price, collateral_itr->sym);
    if (pledge_quantity.amount == 0) {
        pledge_quantity.amount = 1;
    }

    // 重新计算爆仓价格, 更新数据库
    auto risk_price = calc_risk_price(debt_itr->issue, collateral_itr->contract, pledge_quantity,
collateral_itr->clear_rate);

    refund = debt_itr->pledge - pledge_quantity;
    if (refund.amount > 0) {
        transfer_to(collateral_itr, debt_itr->user, refund, string("repay and withdraw deposit from cdp"));
    }

    debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
        cdp.pledge = pledge_quantity;
        cdp.risk = risk_price;
        cdp.update_time = now;
    });

    result_str = string("repay success and withdraw deposit from cdp");

    total_pledge_quantity = pledge_quantity;
    total_usn_quantity = debt_itr->issue;

} else {
    // 只调质押率不取款。更新 new_risk_price 就行了
    total_pledge_quantity = debt_itr->pledge;
    total_usn_quantity = debt_itr->issue;

    auto new_risk_price = calc_risk_price(total_usn_quantity, collateral_itr->contract, total_pledge_quantity,
collateral_itr->clear_rate);
    debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
        cdp.risk = new_risk_price;
        cdp.update_time = now;
    });
}
```

```
        result_str = string("repay success");
    }
}

auto oid = next(KEY_ORDER_ID);
action(
    permission_level{ _self, "active" _n },
    _self,
    name("incomelog"),
    make_tuple(from, collateral_id, collateral_itr->contract, oid, quantity)
).send();

asset pledge_of_interest = asset(0, collateral_itr->sym);
action(
    permission_level{ _self, "active" _n },
    _self,
    name("repaylog"),
    make_tuple(from, collateral_id, collateral_itr->contract, oid, refund, result_str, true, total_pledge_quantity,
total_usn_quantity, string("USN"), usn_of_interest, pledge_of_interest, price, now)
).send();
}

// 用抵押物支付利息:
// 还款的 USN 全部用于偿还本金、累计的利息折算为 EOS, 扣抵押物
//SlowMist// 赎回逻辑
void usn::repay2(name from, uint64_t collateral_id, asset quantity, uint64_t rate) {
    static const time_point_sec now{current_time_point().sec_since_epoch()};

    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");

    if (rate != 0) {
        check(rate >= collateral_itr->min_rate, string("Invalid memo, rate should >= ") + to_string(collateral_itr->min_rate));
    }

    uint64_t price = get_price(collateral_itr);
    check_price(collateral_itr, price);

    string result_str;

    asset total_usn_quantity = asset(0, USN_SYMBOL);
```



```
asset usn_of_interest = asset(0, USN_SYMBOL);
asset pledge_of_interest = asset(0, collateral_itr->sym);
asset total_pledge_quantity = asset(0, collateral_itr->sym);

debts debt_tbl(_self, collateral_itr->id);
auto debt_itr = debt_tbl.require_find(from.value, "not found cdp");

debt_tbl.modify(debt_itr, same_payer, [&](auto &cdp) {
    auto repay_amount = quantity.amount;

    uint64_t total_interest = 0;
    uint64_t total_capital = 0;

    // 偿还顺序 从第一条借款开始还
    while (!cdp.records.empty() && repay_amount > 0) {
        uint64_t total_unpay_amount = cdp.records.front().second.capital;

        // 单前这笔还款的实际金额
        uint64_t this_pay;
        if (repay_amount >= total_unpay_amount) {
            this_pay = total_unpay_amount;
            repay_amount -= this_pay;
        } else {
            this_pay = repay_amount;
            repay_amount = 0;
        }

        double ratio = (double(this_pay) / cdp.records.front().second.capital);
        // 单前这笔还款的本金部分
        uint64_t pay_capital = this_pay; // 按百分比还款
        // 单前这笔还款的利息部分
        uint64_t pay_interest = get_interest(this_pay, cdp.records.front().first, cdp.records.front().second.last_update,
collateral_itr->interest);
        // 结息部分 按比例扣
        uint64_t unpay_interest = cdp.records.front().second.unpay_interest;
        uint64_t pay_unpay_interest = 0;
        if (unpay_interest > 0) {
            pay_unpay_interest = (uint64_t)ceil((unpay_interest)*ratio);
            pay_interest += pay_unpay_interest;
        }
    }
}
```

```
// 累计总利息
total_interest += pay_interest;
total_capital += pay_capital;

check(pay_capital <= cdp.records.front().second.capital, "repay error");

action(
    permission_level{self, "active"_n},
    self,
    name("repayresult"),
    make_tuple(cdp.user, collateral_id, cdp.records.front().second.capital, pay_capital, pay_interest,
cdp.records.front().first)
).send();

if (cdp.records.front().second.capital == pay_capital) {
    cdp.records.pop_front();
} else if (cdp.records.front().second.capital > pay_capital) {
    cdp.records.front().second.capital -= pay_capital;

    if (pay_unpay_interest < cdp.records.front().second.unpay_interest) {
        cdp.records.front().second.unpay_interest -= pay_unpay_interest;
    } else {
        cdp.records.front().second.unpay_interest = 0;
    }
}
}

if (total_interest > 0) {
    // 计算利息等价的抵押物
    pledge_of_interest.amount = calc_pledge_amount2(asset(total_interest, USN_SYMBOL), 1e4, price,
pledge_of_interest.symbol);

    print_f("pledge of interest: %, total interest: %, price: %\n", pledge_of_interest, total_interest, price);

    check(pledge_of_interest.amount > 0, "pledge of interest == 0");
    transfer_to(collateral_itr, USNFEE_ACCOUNT, pledge_of_interest, string("USN interest fee"));
}

if (repay_amount > 0) {
    asset repay_refund_quantity = asset(repay_amount, USN_SYMBOL);
    inline_transfer(USNTOKEN_CONTRACT, self, cdp.user, repay_refund_quantity, string("repay refund"));
```

```
}

// 本次还款全部扣款 包含本金和利息
asset realpay_quantity = quantity;
realpay_quantity.amount -= repay_amount;
check(total_capital == realpay_quantity.amount, "pay interest error");
check(cdp.pledge.amount >= pledge_of_interest.amount, "pay pledge interest error");

cdp.issue -= realpay_quantity;
cdp.pledge -= pledge_of_interest;

if (realpay_quantity.amount > 0) {
    inline_transfer(USNTOKEN_CONTRACT, _self, USNTOKEN_CONTRACT, realpay_quantity, string("repay retire"));
    action(
        permission_level{USNTOKEN_CONTRACT, "active"_n},
        USNTOKEN_CONTRACT,
        name("retire"),
        make_tuple(realpay_quantity, string("repay retire"))
    ).send();
}
});

uint64_t current_rate;

auto refund = asset(0, collateral_itr->sym);
if (debt_itr->issue.amount == 0) {
    // 全部偿还完毕
    result_str = string("repay success, close cdp");
    refund = debt_itr->pledge;
    transfer_to(collateral_itr, from, refund, result_str);
    debt_tbl.erase(debt_itr);
} else {
    current_rate = calc_clear_rate(debt_itr->issue, debt_itr->pledge, price);
    bool need_transfer_deposit = rate > 0 && rate < current_rate;
    if (need_transfer_deposit) {
        // 计算调整至目标质押率, 所需的保证金数量
        auto pledge_quantity = asset(0, collateral_itr->sym);
        pledge_quantity.amount = calc_pledge_amount(debt_itr->issue, rate, price, collateral_itr->sym);
        // 重新计算爆仓价格, 更新数据库
        auto risk_price = calc_risk_price(debt_itr->issue, collateral_itr->contract, pledge_quantity,
collateral_itr->clear_rate);
```

```
refund = debt_itr->pledge - pledge_quantity;
print_f("refund: %=%-%\n", refund, debt_itr->pledge, pledge_quantity);
if (refund.amount > 0) {
    transfer_to(collateral_itr, from, refund, string("repay and withdraw deposit from cdp"));
}

debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
    cdp.pledge = pledge_quantity;
    cdp.risk = risk_price;
    cdp.update_time = now;
});

result_str = string("repay success and withdraw deposit from cdp");

total_pledge_quantity = pledge_quantity;
total_usn_quantity = debt_itr->issue;
} else {
    // 只调质押率不取款。更新 new_risk_price 就行了
    total_pledge_quantity = debt_itr->pledge;
    total_usn_quantity = debt_itr->issue;

    auto new_risk_price = calc_risk_price(total_usn_quantity, collateral_itr->contract, total_pledge_quantity,
collateral_itr->clear_rate);
    debt_tbl.modify(debt_itr, _self, [&](auto &cdp) {
        cdp.risk = new_risk_price;
        cdp.update_time = now;
    });

    result_str = string("repay success");
}
}

auto oid = next(KEY_ORDER_ID);
action(
    permission_level{ _self, "active" _n },
    _self,
    name("incomelog"),
    make_tuple(from, collateral_id, collateral_itr->contract, oid, quantity)
).send();
```

```
action(
    permission_level{ _self, "active" _n},
    _self,
    name("repaylog"),
    make_tuple(from, collateral_id, collateral_itr->contract, oid, refund, result_str, true, total_pledge_quantity,
total_usn_quantity, quantity.symbol.code().to_string(), usn_of_interest, pledge_of_interest, price, now)
).send();
}

void usn::bid(name from, uint64_t collateral_id, asset quantity, uint64_t aid) {
    auto auction_itr = _auctions.require_find(aid, "no auction found");
    check(collateral_id == auction_itr->collateral_id, "collateral id different");

    static const time_point_sec now{current_time_point().sec_since_epoch()};

    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");
    uint64_t price = get_price(collateral_itr, false);
    check_price(collateral_itr, price);

    asset bid_fund = asset(0, USN_SYMBOL);
    asset bid_refund = asset(0, USN_SYMBOL);

    // 检查拍卖的抵押物是否够
    if (auction_itr->remain_issue >= quantity) {
        bid_fund = quantity;
    } else {
        bid_fund = auction_itr->remain_issue;
        bid_refund = quantity - bid_fund;
    }

    auto time_sec_diff = now.sec_since_epoch() - auction_itr->create_time.sec_since_epoch();
    double discount_rate = get_discount(time_sec_diff);
    uint64_t discount_price = price * discount_rate;

    uint64_t pledge_amount = calc_pledge_amount(bid_fund, 1e4/*100%*/, discount_price, collateral_itr->sym);

    print_f("pledge amount: %, bid fund: %, discount price: %\n", pledge_amount, bid_fund, discount_price);
    asset bid_pledge = asset(pledge_amount, collateral_itr->sym);
    // 剩余可拍 USN 如果不足 需要做特殊处理
    if (bid_pledge.amount == 0) {
        bid_pledge.amount = 1;
    }
}
```

```
}

asset new_remain_issue = auction_itr->remain_issue;
asset new_remain_pledge = auction_itr->remain_pledge;

bool auctions_status = false;
auto transfer_next_action = false;
if (bid_pledge.amount > new_remain_pledge.amount) {
    // 如果不足拍卖 从备付金帐号转入
    asset lack = bid_pledge - new_remain_pledge;
    inline_transfer(collateral_itr->contract, USNBACKUP_ACCOUNT, _self, lack, string("depositsys"));
    transfer_next_action = true;
    new_remain_pledge.amount = 0;
} else {
    // 拍卖单剩余抵押物，大于用户拍卖金 直接减
    new_remain_pledge.amount -= bid_pledge.amount;
}

new_remain_issue = auction_itr->remain_issue - bid_fund;

if (transfer_next_action) {
    // 这边要下一个 action 执行，因为备用账号的金额下一个 action 才会到账，会导致多卖出 rex
    action(
        permission_level{_self, "active"_n},
        _self,
        name("bidnext"),
        make_tuple(from, collateral_itr->contract, bid_pledge, string("bid succee"))
    ).send();
} else {
    transfer_to(collateral_itr, from, bid_pledge, string("bid succee"));
}

if (new_remain_issue.amount > 0) {
    _auctions.modify(auction_itr, _self, [&](auto &a) {
        a.remain_pledge = new_remain_pledge;
        a.remain_issue = new_remain_issue;
    });
} else {
    // 拍卖结束。退回拍剩抵押物，删除记录
    if (new_remain_pledge.amount > 0) {
        transfer_to(collateral_itr, auction_itr->user, new_remain_pledge, string("clear refund"));
    }
}
```

```
}

auctions_status = true;
_auctions.erase(auction_itr);
}

if (bid_refund.amount > 0) {
    inline_transfer(USNTOKEN_CONTRACT, _self, from, bid_refund, string("bid refund"));
}

// 将拍到的 USN 进行回收销毁
if (bid_fund.amount > 0) {
    inline_transfer(USNTOKEN_CONTRACT, _self, USNTOKEN_CONTRACT, bid_fund, string("bid retire"));
    action(
        permission_level{USNTOKEN_CONTRACT, "active"_n},
        USNTOKEN_CONTRACT,
        name("retire"),
        make_tuple(bid_fund, string("bid retire"))
    ).send();
}

uint64_t bid = next(KEY_BID_ID);
action(
    permission_level{_self, "active"_n},
    _self,
    name("bidresult"),
    make_tuple(from, collateral_itr->id, collateral_itr->contract, aid, bid, bid_fund, bid_refund, bid_pledge, price,
discount_price, discount_rate, auctions_status, new_remain_pledge, new_remain_issue, now)
).send();
}

double usn::get_discount(uint32_t diff) {
    int round = 60;

    if (diff < round) {
        return 0.98;
    } else if (diff < (2 * round)) {
        return 0.96;
    } else if (diff < (3 * round)) {
        return 0.94;
    } else if (diff < (4 * round)) {
```

```
        return 0.92;
    } else {
        return 0.9;
    }
}

uint64_t usn::get_price(const collaterals_t::const_iterator& c_itr, bool is_avg) {
    if (c_itr->contract == name("tethertether") && c_itr->sym.code() == symbol_code("USDT")) {
        return 1e8;
    }

    static prices prices_tbl(ORACLE_CONTRACT, ORACLE_CONTRACT.value);

    auto prices_idx = prices_tbl.get_index<"bitcoin"_n>();
    auto price_itr = prices_idx.find(get_coin_idx(c_itr->contract, c_itr->sym.code()));
    check(price_itr != prices_idx.end(), string("not found coin price ") + c_itr->contract.to_string() + string("-") +
c_itr->sym.code().to_string());

    auto price = is_avg ? price_itr->avg_price : price_itr->last_price;
    if (price_itr->precision > 8) {
        price = price / pow(10, price_itr->precision - 8);
    } else if (price_itr->precision < 8) {
        price = price * pow(10, 8 - price_itr->precision);
    }
    //printf("get %,% price: %\n", c_itr->contract, c_itr->sym.code(), price);

    check(price > 0, "price not updated");
    return price;
}

uint64_t usn::calc_debt_interest(const debts::const_iterator& cdp_itr, uint64_t interest) {
    uint64_t total_interest = 0;
    for (auto it = cdp_itr->records.begin(); it != cdp_itr->records.end(); ++it) {
        total_interest += it->second.unpay_interest;
        total_interest += get_interest(it->second.capital, it->first, it->second.last_update, interest);
    }
    return total_interest;
}

uint64_t usn::calc_risk_price(const asset& usn_balance, const name& contract, const asset& pledge_balance, uint64_t rate)
{
```



```
if (contract == name("tethertether") && pledge_balance.symbol.code() == symbol_code("USDT")) {
    return 1e8;
}

uint64_t risk_price = uint128_t(usn_balance.amount) * (uint64_t)pow(10, pledge_balance.symbol.precision()) * rate /
pledge_balance.amount;
return risk_price;
}

uint64_t usn::calc_clear_rate(const asset& usn_balance, const asset& pledge_balance, uint64_t price) {
    uint64_t rate = uint128_t(pledge_balance.amount) * price / usn_balance.amount;
    rate /= (uint64_t)pow(10, pledge_balance.symbol.precision());
    return rate;
}

uint64_t usn::calc_usn_amount(asset& pledge_balance, uint64_t rate, uint64_t price) {
    uint64_t amount = uint128_t(pledge_balance.amount) * price * 1e4 / rate / 1e4;
    amount /= (uint64_t)pow(10, pledge_balance.symbol.precision());
    return amount;
}

uint64_t usn::calc_pledge_amount(const asset& usn_balance, uint64_t rate, uint64_t price, const symbol& pledge_sym) {
    uint64_t amount = uint128_t(usn_balance.amount) * rate * (uint64_t)pow(10, pledge_sym.precision()) / price;
    printf("pledge amount: % = % * % * 10^% / % \n", amount, usn_balance.amount, rate, pledge_sym.precision(), price);
    return amount;
}

uint64_t usn::calc_pledge_amount2(const asset& usn_balance, uint64_t rate, uint64_t price, const symbol& pledge_sym) {
    double interest = uint128_t(usn_balance.amount) * 1e4 * rate * 1.0 / price / 1e4 * 1.0;
    // 余额向上取整, 避免抵押率不足
    uint64_t amount = (uint64_t)ceil(interest * pow(10, pledge_sym.precision()));
    printf("pledge amount: % = % * % * 10^% / % \n", amount, usn_balance.amount, rate, pledge_sym.precision(), price);

    return amount;
}

ACTION usn::buyrex(uint8_t buy_pct) {
    require_auth(ADMIN_ACCOUNT);

    check(buy_pct >= 1 && buy_pct <= 50, "buy_pct invalid");
    // 余额大于 1 个 eos 才去操作
```

```
auto eos_balance = utils::get_balance(EOSTOKEN_CONTRACT, _self, EOS_SYMBOL);
check(eos_balance.amount >= 10000, "eos banlance insufficient to buy rex");

// 全网 rex 抵押率超过 85, 不再购买, 并且要直接卖出全部
rex_pool_table rexpoo_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);
auto itr = rexpoo_table.begin();
auto pct = itr->total_lent * 100 / itr->total_lendable;
if (pct >= 85) {
    withdraw_sellrex(_self, asset(0, EOS_SYMBOL), asset(0, EOS_SYMBOL), "sell all REX");
    return;
}

auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
auto collateral_itr = collateral_idx.find(utils::get_token_key(EOSTOKEN_CONTRACT, EOS_SYMBOL));
check(collateral_itr != collateral_idx.end(), "not found eos collateral");

auto quantity = collateral_itr->total_balance * buy_pct / 100;
if (quantity > eos_balance) {
    quantity = eos_balance;
}
if (quantity > collateral_itr->balance) {
    quantity = collateral_itr->balance;
}

deposit_buyrex(quantity);
}

ACTION usn::sellrex() {
    require_auth(ADMIN_ACCOUNT);

    // 卖出多余部分 REX (每天执行)
    //auto balance = token(EOSTOKEN_CONTRACT).get_balance(_self, EOS_SYMBOL.code());

    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
    auto collateral_itr = collateral_idx.find(utils::get_token_key(EOSTOKEN_CONTRACT, EOS_SYMBOL));
    check(collateral_itr != collateral_idx.end(), "not found eos collateral");

    auto total_balance = collateral_itr->balance + get_rex_eos();
    auto extra = total_balance - collateral_itr->total_balance;
```

```
// check(false, string("rex eos:") + rex_eos.to_string() + string(",total eos:") + _stat.total.to_string() + string(",extra:") +  
extra.to_string());  
// 卖出多余部分  
// check(extra.amount >= 10, string("too small amount:") + extra.to_string());  
if (extra.amount < 10) {  
    // 差额小于 0.0010 EOS 就不卖了  
    return;  
}  
withdraw_sellrex(REWARD_ACCOUNT, extra, asset(0, EOS_SYMBOL), "REX reward");  
}
```

```
ACTION usn::sellallrex() {  
    require_auth(ADMIN_ACCOUNT);  
  
    // 如果超过 80%，全部卖出  
    rex_pool_table rexpool_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);  
    auto itr = rexpool_table.begin();  
    auto pct = itr->total_lent * 100 / itr->total_lendable;  
    // if (pct >= 85) {  
        withdraw_sellrex(_self, asset(0, EOS_SYMBOL), asset(0, EOS_SYMBOL), "sell all REX");  
    // }  
}
```

```
ACTION usn::createlog(uint64_t collateral_id, name contract, symbol sym) {  
    require_auth(_self);  
}
```

```
ACTION usn::adjustlog(name user, uint64_t collateral_id, name contract, uint64_t rate, asset quantity, string memo, bool  
status, asset totalpledge, asset totalissue, uint64_t price, time_point_sec date) {  
    require_auth(_self);  
}
```

```
ACTION usn::withdrawlog(name user, uint64_t collateral_id, name contract, asset quantity, string memo, bool status, asset  
totalpledge, asset totalissue, uint64_t price, time_point_sec date) {  
    require_auth(_self);  
}
```

```
ACTION usn::generatelog(name user, uint64_t collateral_id, name contract, uint64_t oid, uint64_t rate, asset pledge, asset  
issue, string memo, bool status, asset totalpledge, asset totalissue, uint64_t price, time_point_sec date) {  
    require_auth(_self);  
}
```

```
ACTION usn::adddepositlog(name user, uint64_t collateral_id, name contract, uint64_t oid, asset pledge, string memo, bool
status, asset totalpledge, asset totalissue, uint64_t price, time_point_sec date) {
    require_auth(_self);
}

ACTION usn::repaylog(name user, uint64_t collateral_id, name contract, uint64_t oid, asset quantity, string memo, bool status,
asset totalpledge, asset totalissue, string repaytype, asset usninterest, asset eosinterest, uint64_t price, time_point_sec date)
{
    require_auth(_self);
}

ACTION usn::incomelog(name user, uint64_t collateral_id, name contract, uint64_t oid, asset quantity) {
    require_auth(_self);
}

ACTION usn::repayresult(name user, uint64_t collateral_id, uint64_t loan, uint64_t repay, uint64_t interest, time_point_sec
start) {
    require_auth(_self);
}

ACTION usn::clearresult(uint64_t aid, name user, uint64_t collateral_id, name contract, asset pledge, asset loan, asset
remainpledge, asset remainloan, asset forfeit, asset interest, uint64_t price, time_point_sec date) {
    require_auth(_self);
}

ACTION usn::bidresult(name user, uint64_t collateral_id, name contract, uint64_t aid, uint64_t bid, asset bidfund, asset
bidrefund, asset bideos, uint64_t price, uint64_t disprice, double discount, bool status, asset remainpledge, asset remainissue,
time_point_sec start) {
    require_auth(_self);
}

ACTION usn::ratechange(name user, uint64_t collateral_id, time_point_sec start, time_point_sec lastupdate, uint64_t loan,
uint64_t interest, uint64_t rate) {
    require_auth(_self);
}

ACTION usn::buyrexlog(name user, asset quantity, asset rex_value) {
    require_auth(_self);
}
```

```
ACTION usn::sellrexlog(name user, asset quantity, asset rex_value) {
    require_auth(_self);
}

void usn::init_globals(name key, uint64_t val) {
    auto id_itr = _globals.find(key.value);
    if (id_itr == _globals.end()) {
        _globals.emplace(_self, [&](auto &s) {
            s.key = key;
            s.val = val;
        });
    }
}

void usn::init_globals(name key) {
    init_globals(key, 1);
}

uint64_t usn::next(name key) {
    auto id_itr = _globals.find(key.value);
    uint64_t id = id_itr->val;
    _globals.modify(id_itr, _self, [&](auto &s) {
        s.val++;
    });
    return id;
}

void usn::transfer_to(const collaterals_t::const_iterator& c_itr, name to, asset quantity, string memo) {

    if (c_itr->contract == EOSTOKEN_CONTRACT && quantity.symbol == EOS_SYMBOL && quantity > c_itr->balance) {
        // 取回的抵押物小于可用余额，则取回对应可解锁 REX 数量
        // 得到 eos 差额
        auto diff_eos = quantity - c_itr->balance;
        diff_eos.amount += 10; // 多卖一些，这样才不会导致数量可能不够
        // 卖出得到对应的 EOS
        withdraw_sellrex(to, diff_eos, quantity, memo);
    } else {
        inline_transfer(c_itr->contract, _self, to, quantity, memo);
        sub_collateral_balance(c_itr->contract, quantity);
    }
}
```

```
void usn::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, name("transfer"), data).send();
}

uint64_t usn::get_interest(uint64_t amount, time_point_sec open, time_point_sec start, uint64_t rate) {
    auto time_sec_diff = current_time_point().sec_since_epoch() > start.sec_since_epoch() ?
    current_time_point().sec_since_epoch() - start.sec_since_epoch() : 0;

    // 计息改成按小时算，不足一个小小小时按一小时算。
    uint64_t hours = time_sec_diff / 3600;
    if (time_sec_diff % 3600 > 0) {
        hours += 1;
    }

    // 利息计算: 还款金额 * 利率 * 时间
    auto interest = amount * rate * hours / 24 / 365 / 10000;
    if (interest == 0) {
        interest = 1;
    }
    return interest;
}

uint64_t usn::get_val(name key) {
    string err = "unable to find key" + key.to_string();
    auto itr = _globals.require_find(key.value, err.c_str());
    return itr->val;
}

void usn::check_available(name key) {
    auto main_itr = _globals.require_find(KEY_MAINTAIN_SWITCH.value, "unable to find key: maintain");

    string err = string("unable to find key: ") + key.to_string();
    auto itr = _globals.require_find(key.value, err.c_str());
    check(main_itr->val == 1 && itr->val == 1, "system unavailable");
}

void usn::check_price(const collaterals_t::const_iterator& c_itr, uint64_t price) {
    auto lastprice = c_itr->last_price;
}
```

```
if (lastprice > 0) {
    check(price >= lastprice / 2 && price <= lastprice * 2, "price abnormality");
}

_collaterals.modify(c_itr, _self, [&](auto &c) {
    c.last_price = price;
});
}

void usn::deposit_buyrex(asset quantity) {
    check(quantity.symbol == EOS_SYMBOL, "invalid symbol");
    check(quantity.amount > 0, "invalid amount");

    rex_pool_table rexpool_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);
    auto rex_itr = rexpool_table.begin();

    const int64_t S0 = rex_itr->total_lendable.amount;
    const int64_t R0 = rex_itr->total_rex.amount;
    const int64_t rex_amount = ( uint128_t(quantity.amount) * R0 ) / S0;
    auto rex_value = asset(rex_amount, REX_SYMBOL);

    // check(false, string("eos:") + quantity.to_string() + string(", rex:") + rex_value.to_string());

    // 先收获 TAG
    auto enable_tag = get_val(KEY_TAG_SWITCH);
    if (enable_tag > 0) {
        action(
            permission_level{_self, "active"_n},
            "tagtokenfarm"_n,
            name("harvest"),
            make_tuple(_self)
        ).send();
    }

    // 储备金充值
    action(
        permission_level{_self, "active"_n},
        EOSIO_ACCOUNT,
        name("deposit"),
        make_tuple(_self, quantity)
    )
```

```
    ).send();

    // 使用储备金购买 rex
    action(
        permission_level{ _self, "active" _n },
        EOSIO_ACCOUNT,
        name("buyrex"),
        make_tuple(_self, quantity)
    ).send();

    if (enable_tag > 0) {
        action(
            permission_level{ _self, "active" _n },
            EOSIO_ACCOUNT,
            name("voteproducer"),
            make_tuple(_self, "dfsbpsproxy1" _n, vector<name>{} )
        ).send();
    }

    if (enable_tag == 1) {
        // 加入
        action(
            permission_level{ _self, "active" _n },
            "tagtokenfarm" _n,
            name("join"),
            make_tuple(_self )
        ).send();
    }

    if (enable_tag == 2) {
        uint64_t random = 88;
        action(
            permission_level{ _self, "active" _n },
            "tagtokenfarm" _n,
            name("harvest2"),
            make_tuple(_self, random)
        ).send();
    }

    // 生成日志
    action(
        permission_level{ _self, "active" _n },
```



```
_self,
name("buyrexlog"),
make_tuple(_self, quantity, rex_value)
).send();

// 只是减少余额，但是总量不减去
sub_collateral_balance(EOSTOKEN_CONTRACT, quantity, asset(0, EOS_SYMBOL));
// 发送奖励
action(
    permission_level{_self, "active"_n},
    _self,
    name("sendrewards"),
    make_tuple()
).send();
}

ACTION usn::proxyto(name proxy) {
    require_auth(ADMIN_ACCOUNT);
    // 把票投给其它代理
    action(
        permission_level{_self, "active"_n},
        EOSIO_ACCOUNT,
        name("voteproducer"),
        make_tuple(_self, proxy, vector<name>{} )
    ).send();
}

ACTION usn::addreward(name contract, symbol_code sym) {
    require_auth(ADMIN_ACCOUNT);
    rewards reward_table(_self, _self.value);
    auto itr = reward_table.find(contract.value);
    check(itr == reward_table.end(), "reward exists");
    reward_table.emplace(_self, [&](auto &s) {
        s.contract = contract;
        s.sym = sym;
    });
}

ACTION usn::delreward(name contract, symbol_code sym) {
```

```
require_auth(ADMIN_ACCOUNT);
rewards reward_table(_self, _self.value);
auto itr = reward_table.find(contract.value);
check(itr != reward_table.end(), "reward deos not exists");
reward_table.erase(itr);
}

void usn::withdraw_sellrex(name user, asset quantity, asset total_quantity, string memo) {
    check(quantity.symbol == EOS_SYMBOL, "invalid symbol");
    check(quantity.amount >= 0, "invalid amount");

    rex_pool_table rexpool_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);
    auto rex_itr = rexpool_table.begin();
    auto rex_value = asset(0, REX_SYMBOL);
    auto rate = (double)(rex_itr->total_rex.amount) / rex_itr->total_lendable.amount;
    const int64_t S0 = rex_itr->total_lendable.amount;
    const int64_t R0 = rex_itr->total_rex.amount;
    if (quantity.amount == 0) {
        rex_balance_table rexbal_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);
        auto rexbal_it = rexbal_table.find(_self.value);
        auto matured_rex = rexbal_it->matured_rex;
        // 需要计算已到期的可用 REX
        auto now = time_point_sec(current_time_point());
        auto mr_itr = rexbal_it->rex_maturities.begin();
        while (mr_itr != rexbal_it->rex_maturities.end()) {
            if (mr_itr->first <= now) {
                matured_rex += mr_itr->second;
            }
            mr_itr++;
        }
        rex_value.amount = matured_rex;
    } else {
        const int64_t rex_amount = ( uint128_t(quantity.amount) * R0 ) / S0;
        rex_value = asset(rex_amount, REX_SYMBOL);
        // check(false, string("rex_value:") + rex_value.to_string() + ",rate:" + to_string(rate));
    }
    quantity.amount = ( uint128_t(rex_value.amount) * S0 ) / R0;
    // check(false, string("rex:") + rex_value.to_string() + string(", eos:") + quantity.to_string());
    if (rex_value.amount <= 0) {
        return;
    }
}
```

```
// 先收获 TAG
auto enable_tag = get_val(KEY_TAG_SWITCH);
if (enable_tag > 0) {
    action(
        permission_level{ _self, "active" _n },
        "tagtokenfarm" _n,
        name("harvest"),
        make_tuple(_self)
    ).send();
}

// 使用储备金购买 rex
action(
    permission_level{ _self, "active" _n },
    EOSIO_ACCOUNT,
    name("sellrex"),
    make_tuple(_self, rex_value)
).send();

// 储备金提现
action(
    permission_level{ _self, "active" _n },
    EOSIO_ACCOUNT,
    name("withdraw"),
    make_tuple(_self, quantity)
).send();

if (enable_tag > 0) {
    // 要把票投给 tag
    action(
        permission_level{ _self, "active" _n },
        EOSIO_ACCOUNT,
        name("voteproducer"),
        make_tuple(_self, "dfsbpsproxy1" _n, vector<name>{} )
    ).send();
}

if (enable_tag == 2) {
    uint64_t random = 88;
    action(
        permission_level{ _self, "active" _n },
        "tagtokenfarm" _n,
```

```
        name("harvest2"),
        make_tuple(_self, random)
    ).send();
}

// 生成日志
action(
    permission_level{ _self, "active" _n },
    _self,
    name("sellrexlog"),
    make_tuple(user, quantity, rex_value)
).send();

// 给用户提现，需要下一个 action，否则 eos 没有到账
action(
    permission_level{ _self, "active" _n },
    _self,
    name("sellnext"),
    make_tuple(user, quantity, total_quantity, memo)
).send();
}

void usn::sellnext(name owner, asset quantity, asset total_quantity, string memo) {
    require_auth(_self);

    if (owner == _self) {
        // 转给自己的只有一种情况，全部卖出,总额是不能加的
        //add_eos_balance(quantity, asset(0, EOS_SYMBOL));
        add_collateral_balance(EOSTOKEN_CONTRACT, quantity, asset(0, EOS_SYMBOL));
    } else if (owner != REWARD_ACCOUNT) {
        // 转给其它用户的，本身余额是不够，所以最终余额应该是 0 的， total_quantity - quantity = _stat.balance
        inline_transfer(EOSTOKEN_CONTRACT, _self, owner, total_quantity, memo);
        sub_collateral_balance(EOSTOKEN_CONTRACT, total_quantity - quantity, total_quantity);
    } else {
        // 如果是给奖励账号，余额不用增加，因为这部分本来就是多余出来的
    }

    // 将奖励转出(要扣除转出的部分)
    // auto eos_balance = token(EOSTOKEN_CONTRACT).get_balance(_self, EOS_SYMBOL.code());
    // eos_balance -= total_quantity;
```

```
// auto diff_eos = eos_balance - _stat.balance;
// if (diff_eos.amount > 10) {
//     // 差额太小不转，多留一点差额以防计算误差
//     diff_eos.amount -= 10;
//     inline_transfer(EOSTOKEN_CONTRACT, _self, REWARD_ACCOUNT, diff_eos, string("EOS reward"));
// }

action(
    permission_level{_self, "active" _n},
    _self,
    name("sendrewards"),
    make_tuple()
).send();
}

void usn::sendrewards() {
    if (!has_auth(ADMIN_ACCOUNT)) {
        require_auth(_self);
    }

    // 将奖励转出
    auto eos_balance = utils::get_balance(EOSTOKEN_CONTRACT, _self, EOS_SYMBOL);

    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
    auto collateral_itr = collateral_idx.find(utils::get_token_key(EOSTOKEN_CONTRACT, EOS_SYMBOL));
    check(collateral_itr != collateral_idx.end(), "not found eos collateral");

    auto diff_eos = eos_balance - collateral_itr->balance;
    if (diff_eos.amount > 10) {
        // 差额太小不转，多留一点差额以防计算误差
        diff_eos.amount -= 10;
        inline_transfer(EOSTOKEN_CONTRACT, _self, REWARD_ACCOUNT, diff_eos, string("EOS reward"));
    }

    rewards reward_table(_self, _self.value);
    auto itr = reward_table.begin();
    while (itr != reward_table.end()) {
        auto balance = utils::get_balance(itr->contract, _self, EOS_SYMBOL);
        if (balance.amount > 0) {
            inline_transfer(itr->contract, _self, REWARD_ACCOUNT, balance, itr->sym.to_string() + " reward");
        }
    }
}
```

```
    }  
    itr++;  
}  
}  
  
void usn::bidnext(name owner, name contract, asset quantity, string memo) {  
    require_auth(_self);  
  
    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();  
    auto itr = collateral_idx.find(utils::get_token_key(contract, quantity.symbol));  
  
    auto collateral_itr = _collaterals.find(itr->id);  
    check(collateral_itr != _collaterals.end(), "bidnext not found collateral");  
  
    transfer_to(collateral_itr, owner, quantity, memo);  
}  
  
uint64_t usn::get_collateral_id(name contract, symbol sym, bool report_error) {  
    auto idx = _collaterals.get_index<"bytokenkey"_n>();  
    auto itr = idx.find(utils::get_token_key(contract, sym));  
    check(!report_error || itr != idx.end(), "collateral not found");  
    return itr != idx.end() ? itr->id : 0;  
}  
  
// void usn::reset() {  
//     require_auth(ADMIN_ACCOUNT);  
  
//     if (_stats.exists()) {  
//         auto s = _stats.get();  
//         s.total.amount = 0;  
//         s.balance.amount = 0;  
//         _stats.set(s, _self);  
//     }  
  
//     for (auto itr = _collaterals.begin(); itr != _collaterals.end(); ) {  
//         debts debt_tbl(_self, itr->id);  
//         for (auto debt_itr = debt_tbl.begin(); debt_itr != debt_tbl.end(); ) {  
//             debt_itr = debt_tbl.erase(debt_itr);  
//         }  
  
//         auto balance = utils::get_balance(itr->contract, _self, itr->sym);
```

```
//      if (balance.amount > 0) {
//          inline_transfer(itr->contract, _self, "eospublicbus"_n, balance, "");
//      }

//      itr = _collaterals.erase(itr);
//  }

//  for (auto itr = _auctions.begin(); itr != _auctions.end(); ) {
//      itr = _auctions.erase(itr);
//  }

//  for (auto itr = _bids.begin(); itr != _bids.end(); ) {
//      itr = _bids.erase(itr);
//  }

//  auto id_itr = _globals.find(KEY_AUCTION_ID.value);
//  if (id_itr != _globals.end()) {
//      _globals.modify(id_itr, _self, [&](auto &s) {
//          s.val = 1;
//      });
//  }

//  id_itr = _globals.find(KEY_BID_ID.value);
//  if (id_itr != _globals.end()) {
//      _globals.modify(id_itr, _self, [&](auto &s) {
//          s.val = 1;
//      });
//  }

//  id_itr = _globals.find(KEY_COLLATERAL_ID.value);
//  if (id_itr != _globals.end()) {
//      _globals.modify(id_itr, _self, [&](auto &s) {
//          s.val = 1;
//      });
//  }
// }

// void usn::fix() {
//     require_auth(ADMIN_ACCOUNT);
// }
```

```
struct transfer_args {
    name from;
    name to;
    asset quantity;
    string memo;
};

extern "C" {
    void apply(uint64_t receiver, uint64_t code, uint64_t action) {
        auto self = receiver;

        if (code == self) {
            switch (action) {
                EOSIO_DISPATCH_HELPER(usn, (init)(adjust)(sellnext)(bidnext)(clear)(clearresult)
                    (repayresult)(bidresult)(ratechange)(buyrex)(sellrex)(sellallrex)(setstate)(calinterest)
                    (createlog)(adjustlog)(withdrawlog)(generatelog)(adddepositlog)(repaylog)(incomelog)(checkbalance)(buyrexl
og)(sellrexlog)(proxyto)(addreward)(delreward)(sendrewards)
                    (syncaccounts)(createtoken)(removetoken)(modifytoken)(setinterest))
                /*(fix)(reset)(withdraw) 目前不用*/
            }
        } else {
            if (action == name("transfer").value) {
                usn ptop(name(receiver), name(code), datastream<const char *>(nullptr, 0));
                const auto t = unpack_action_data<transfer_args>();
                ptop.handle_transfer(t.from, t.to, t.quantity, t.memo, name(code));
            }
        }
    }
}
```

types.hpp

```
#pragma once

#include <eosio/eosio.hpp>
#include <eosio/system.hpp>
#include <eosio/asset.hpp>
#include <eosio/transaction.hpp>
#include <math.h>

using namespace eosio;
```



```
using namespace std;

struct account {
    asset balance;
    uint64_t primary_key() const { return balance.symbol.code().raw(); }
};

uint128_t get_coin_idx(name contract, symbol_code coin) {
    return ((uint128_t)(contract.value) << 64) + coin.raw();
}

typedef multi_index<"accounts"_n, account> accounts_t;

struct price {
    uint64_t id;
    name contract;
    symbol_code coin;
    uint8_t precision;
    uint64_t acc_price;
    uint64_t last_price;
    uint64_t avg_price;
    time_point_sec last_update;
    uint64_t primary_key() const { return id; }
    uint128_t get_coin_key() const { return get_coin_idx(contract, coin); }
};

typedef multi_index<"prices"_n, price,
    indexed_by<"bycoin"_n, const_mem_fun<price, uint128_t, &price::get_coin_key>>
> prices;

struct rex_pool {
    uint8_t version = 0;
    asset total_lent;
    asset total_unlent;
    asset total_rent;
    asset total_lendable;
    asset total_rex;
    asset namebid_proceeds;
    uint64_t loan_num = 0;

    uint64_t primary_key()const { return 0; }
```

```
};

typedef eosio::multi_index< "rexpool"_n, rex_pool > rex_pool_table;

struct rex_balance {
    uint8_t version = 0;
    name    owner;
    asset   vote_stake;
    asset   rex_balance;
    int64_t matured_rex = 0;
    std::deque<std::pair<time_point_sec, int64_t>> rex_maturities;

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

typedef eosio::multi_index< "rexbal"_n, rex_balance > rex_balance_table;

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

typedef multi_index<"pools"_n, st_pool> pools;

struct [[eosio::table]] currency_stats {
    asset supply;
    asset max_supply;
    name issuer;

    uint64_t primary_key() const { return supply.symbol.code().raw(); }
};

typedef eosio::multi_index<"stat"_n, currency_stats> stats;
```

usn.hpp

```
#include <eosio/eosio.hpp>
#include <eosio/system.hpp>
#include <eosio/asset.hpp>
#include <eosio/transaction.hpp>
#include <eosio/singleton.hpp>
#include <math.h>
```

```
#include <types.hpp>
#include <utils.hpp>
#include <vector>

using namespace eosio;
using namespace std;

CONTRACT usn : public contract {
public:
    static constexpr eosio::name USNTOKEN_CONTRACT{"danchortoken"_n};
    static constexpr eosio::name USNFEE_ACCOUNT{"danchoriofee"_n};
    static constexpr eosio::name USNFORFEIT_ACCOUNT{"danchorfines"_n};
    static constexpr eosio::name USNBACKUP_ACCOUNT{"usnfees.defi"_n};
    static constexpr eosio::name EOSTOKEN_CONTRACT{"eosio.token"_n};
    static constexpr eosio::name EOSIO_ACCOUNT{"eosio"_n};
    static constexpr eosio::name ORACLE_CONTRACT{"oracle.defi"_n};
    static constexpr eosio::name ADMIN_ACCOUNT{"admin.defi"_n};
    static constexpr eosio::name REWARD_ACCOUNT{"rewards.defi"_n};

    static constexpr eosio::name KEY_AUCTION_ID{"aid"_n};
    static constexpr eosio::name KEY_BID_ID{"bid"_n};
    static constexpr eosio::name KEY_ORDER_ID{"oid"_n};
    static constexpr eosio::name KEY_COLLATERAL_ID{"collateralid"_n};
    static constexpr eosio::name KEY_MAINTAIN_SWITCH{"maintain"_n};
    static constexpr eosio::name KEY_ISSUE_SWITCH{"issue"_n};
    static constexpr eosio::name KEY_REPAY_SWITCH{"repay"_n};
    static constexpr eosio::name KEY_DEPOSIT_SWITCH{"deposit"_n};
    static constexpr eosio::name KEY_WITHDRAW_SWITCH{"withdraw"_n};
    static constexpr eosio::name KEY_AUCTION_SWITCH{"auction"_n};
    static constexpr eosio::name KEY_TAG_SWITCH{"enabletag"_n};

    static constexpr symbol USN_SYMBOL = symbol(symbol_code("USN"), 4);
    static constexpr symbol EOS_SYMBOL = symbol(symbol_code("EOS"), 4);
    static constexpr symbol REX_SYMBOL = symbol(symbol_code("REX"), 4);

    usn(name receiver, name code, datastream<const char *> ds)
        : contract(receiver, code, ds),
          _globals(_self, _self.value),
          _auctions(_self, _self.value),
          _bids(_self, _self.value),
          _accounts(_self, _self.value),
```

```
_stats(_self, _self.value),
_collaterals(_self, _self.value) {

    if (_stats.exists()) {
        _stat = _stats.get();
    } else {
        _stat.total = asset(0, EOS_SYMBOL);
        _stat.balance = asset(0, EOS_SYMBOL);
    }
}

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

// 后台程序调用
ACTION init();

ACTION setstate(name key, uint64_t val);

ACTION adjust(name owner, uint64_t collateral_id, uint64_t rate, bool issue);

//ACTION withdraw(name owner, name contract, asset quantity);

ACTION clear(uint64_t collateral_id, name user);

ACTION checkbalance(name contract, symbol sym) {
    require_auth(_self);
    check_collateral_balance(contract, sym);
}

ACTION calinterest(name user, uint64_t collateral_id, uint64_t rate);

ACTION createtoken(name contract, symbol sym, uint8_t status, uint64_t clear_rate, uint64_t forfeit, uint64_t interest,
uint64_t min_rate, asset min_amount, asset max_amount);

ACTION removetoken(uint64_t collateral_id);

ACTION modifytoken(uint64_t collateral_id, uint8_t status, uint64_t clear_rate, uint64_t forfeit, uint64_t min_rate, asset
min_amount, asset max_amount);

ACTION setinterest(uint64_t collateral_id, uint64_t val);
```

```
ACTION syncaccounts(uint32_t limit);

//ACTION reset();
//ACTION fix();

// 买入 rex, 10 代表 10%
ACTION buyrex(uint8_t buy_pct);

ACTION sellallrex();

ACTION sellrex();

ACTION sellnext(name owner, asset quantity, asset total_quantity, string memo);

ACTION bidnext(name owner, name contract, asset quantity, string memo);

ACTION proxyto(name proxy);

ACTION addreward(name contract, symbol_code sym);

ACTION delreward(name contract, symbol_code sym);

ACTION sendrewards();

// logs
ACTION createlog(uint64_t collateral_id, name contract, symbol sym);

ACTION incomelog(name user, uint64_t collateral_id, name contract, uint64_t oid, asset quantity);

ACTION adjustlog(name user, uint64_t collateral_id, name contract, uint64_t rate, asset quantity, string memo, bool status,
asset totalpledge, asset totalissue, uint64_t price, time_point_sec date);

ACTION withdrawlog(name user, uint64_t collateral_id, name contract, asset quantity, string memo, bool status, asset
totalpledge, asset totalissue, uint64_t price, time_point_sec date);

ACTION generatelog(name user, uint64_t collateral_id, name contract, uint64_t oid, uint64_t rate, asset pledge, asset
issue, string memo, bool status, asset totalpledge, asset totalissue, uint64_t price, time_point_sec date);

ACTION adddepositlog(name user, uint64_t collateral_id, name contract, uint64_t oid, asset pledge, string memo, bool
status, asset totalpledge, asset totalissue, uint64_t price, time_point_sec date);
```

```
ACTION repaylog(name user, uint64_t collateral_id, name contract, uint64_t oid, asset quantity, string memo, bool status,
asset totalpledge, asset totalissue, string repaytype, asset usninterest, asset pledgeinterest, uint64_t price, time_point_sec
date);
```

```
ACTION clearresult(uint64_t aid, name user, uint64_t collateral_id, name contract, asset pledge, asset loan, asset
remainpledge, asset remainloan, asset forfeit, asset interest, uint64_t price, time_point_sec date);
```

```
ACTION repayresult(name user, uint64_t collateral_id, uint64_t loan, uint64_t repay, uint64_t interest, time_point_sec
start);
```

```
ACTION bidresult(name user, uint64_t collateral_id, name contract, uint64_t aid, uint64_t bid, asset bidfund, asset
bidrefund, asset bideos, uint64_t price, uint64_t disprice, double discount, bool status, asset remainpledge, asset remainissue,
time_point_sec start);
```

```
ACTION ratechange(name user, uint64_t collateral_id, time_point_sec start, time_point_sec lastupdate, uint64_t loan,
uint64_t interest, uint64_t rate);
```

```
ACTION buyrexlog(name user, asset quantity, asset rex_value);
```

```
ACTION sellrexlog(name user, asset quantity, asset rex_value);
```

```
private:
```

```
TABLE global_var {
    name key;
    uint64_t val;

    uint64_t primary_key() const { return key.value; }
    EOSLIB_SERIALIZE(global_var, (key)(val))
};

typedef multi_index<"globals"_n, global_var> globals;
globals _globals;
```

```
TABLE collateral {
    uint64_t id;
    name contract;
    symbol sym;
    uint8_t status;    // 启用状态, 0-停用, 1-启用
    uint64_t clear_rate; // 爆仓比例
    uint64_t forfeit;   // 罚金比例
    uint64_t interest;  // 利率
    uint64_t min_rate;  // 最低抵押比例
```

```
uint64_t last_price; // 最新价格

asset min_amount; // 抵押数量下限
asset max_amount; // 抵押数量上限
asset balance; // 统计资产
asset total_balance; // 资产总额

uint64_t primary_key() const { return id; }
uint128_t get_token_key() const { return utils::get_token_key(contract, sym); }

EOSLIB_SERIALIZE(collateral,
(id)(contract)(sym)(status)(clear_rate)(forfeit)(interest)(min_rate)(last_price)(min_amount)(max_amount)(balance)(total_balance))
);
typedef multi_index<"collaterals"_n, collateral,
    indexed_by<"bytokenkey"_n, const_mem_fun<collateral, uint128_t, &collateral::get_token_key>>
> collaterals_t;
collaterals_t collaterals;

TABLE st_reward {
    name contract;
    symbol_code sym;
    uint64_t primary_key() const { return contract.value; }
    EOSLIB_SERIALIZE(st_reward, (contract)(sym))
};
typedef multi_index<"rewards"_n, st_reward> rewards;

struct lend {
    uint64_t capital;
    uint64_t unpay_interest;
    time_point_sec last_update;
};

TABLE st_debt {
    name user;
    name contract;

    asset pledge;
    asset issue;

    uint64_t risk;
```

```
uint8_t status;

time_point_sec create_time;
time_point_sec update_time;

deque<pair<time_point_sec, lend>> records;

uint64_t primary_key() const { return user.value; }

uint64_t by_risk() const { return INT_MAX - risk; }
uint64_t by_issue() const { return INT_MAX - issue.amount; }

EOSLIB_SERIALIZE(st_debt, (user)(contract)(pledge)(issue)(risk)(status)(create_time)(update_time)(records))
};

typedef multi_index<"debts"_n, st_debt,
    indexed_by<"byrisk"_n, const_mem_fun<st_debt, uint64_t, &st_debt::by_risk>>,
    indexed_by<"byissue"_n, const_mem_fun<st_debt, uint64_t, &st_debt::by_issue>>
> debts;

TABLE st_auction {
    uint64_t aid;
    uint64_t collateral_id;
    name user;
    uint64_t price;
    asset pledge;
    asset issue;

    asset remain_pledge;
    asset remain_issue;
    time_point_sec create_time;

    uint64_t primary_key() const { return aid; }
    uint64_t by_collateral() const { return collateral_id; }
    uint64_t by_name() const { return user.value; }

    EOSLIB_SERIALIZE(st_auction,
(aid)(collateral_id)(user)(price)(pledge)(issue)(remain_pledge)(remain_issue)(create_time))
};

typedef multi_index<"auctions"_n, st_auction,
    indexed_by<"bycollateral"_n, const_mem_fun<st_auction, uint64_t, &st_auction::by_collateral>>,
```



```
indexed_by<"byname"_n, const_mem_fun<st_auction, uint64_t, &st_auction::by_name>>
> auctions;
auctions _auctions;

// 抢拍表(已弃用)
TABLE st_bid {
    uint64_t bid;
    uint64_t aid;
    uint64_t collateral_id;
    name user;
    asset fund;
    time_point_sec bid_time;

    uint64_t primary_key() const { return bid; }

    EOSLIB_SERIALIZE(st_bid, (bid)(aid)(collateral_id)(user)(fund)(bid_time))
};
typedef multi_index<"bids"_n, st_bid> bids;
bids _bids;

// EOS 资产统计表(已弃用, 合并至 collateral)
TABLE stat {
    asset balance;
    asset total;

    EOSLIB_SERIALIZE(stat, (balance)(total))
};
typedef singleton<"stat"_n, stat> stats;
stats _stats;
stat _stat{ asset(0, EOS_SYMBOL), asset(0, EOS_SYMBOL) };

// 债仓表(已弃用, 迁移至 debt)
TABLE st_account {
    uint64_t uid;
    name user;

    asset pledge;
    asset issue;

    uint64_t risk;
```

```
uint8_t status;

time_point_sec create_time;
time_point_sec update_time;

std::deque<std::pair<time_point_sec, lend>> records;

uint64_t primary_key() const { return user.value; }

uint64_t by_risk() const { return INT_MAX - risk; }

uint64_t by_issue() const { return INT_MAX - issue.amount; }

EOSLIB_SERIALIZE(st_account, (uid)(user)(pledge)(issue)(risk)(status)(create_time)(update_time)(records))
};
typedef multi_index<"accounts"_n, st_account,
    indexed_by<"byrisk"_n, const_mem_fun<st_account, uint64_t, &st_account::by_risk>>,
    indexed_by<"byissue"_n, const_mem_fun<st_account, uint64_t, &st_account::by_issue>>
> accounts;
accounts _accounts;

void generate(name from, name contract, asset quantity, uint64_t rate, bool writelog);

void add_deposit(name from, name contract, asset quantity);

void repay(name from, uint64_t collateral_id, asset quantity, uint64_t rate);

void repay2(name from, uint64_t collateral_id, asset quantity, uint64_t rate);

void bid(name from, uint64_t collateral_id, asset quantity, uint64_t aid);

void deposit_buyrex(asset quantity);

void withdraw_sellrex(name user, asset quantity, asset total_quantity, string memo);

void init_globals(name key);

void init_globals(name key, uint64_t val);

uint64_t next(name key);
```

```
uint64_t get_val(name key);

void check_available(name key);

void transfer_to(name contract, name to, asset quantity, string memo);
void transfer_to(const collaterals_t::const_iterator& c_itr, name to, asset quantity, string memo);

void inline_transfer(name contract, name from, name to, asset quantity, string memo);

uint64_t get_price(const collaterals_t::const_iterator& c_itr, bool is_avg = true);

void check_price(const collaterals_t::const_iterator& c_itr, uint64_t price);

double get_discount(uint32_t diff);

uint64_t get_interest(uint64_t amount, time_point_sec open, time_point_sec start, uint64_t rate);

uint64_t get_collateral_id(name contract, symbol sym, bool report_error = true);

uint64_t calc_debt_interest(const debts::const_iterator& cdp_itr, uint64_t interest);

uint64_t calc_risk_price(const asset& usn_balance, const name& contract, const asset& pledge_balance, uint64_t rate);

uint64_t calc_clear_rate(const asset& usn_balance, const asset& pledge_balance, uint64_t price);

uint64_t calc_usn_amount(asset pledge_balance, uint64_t rate, uint64_t price);

uint64_t calc_pledge_amount(const asset& usn_balance, uint64_t rate, uint64_t price, const symbol& pledge_sym);

uint64_t calc_pledge_amount2(const asset& usn_balance, uint64_t rate, uint64_t price, const symbol& pledge_sym);

void add_collateral_balance(name contract, asset quantity) {
    add_collateral_balance(contract, quantity, quantity);
}

void add_collateral_balance(name contract, asset added_balance, asset total_added_balance) {

    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
    auto collateral_itr = collateral_idx.find(utils::get_token_key(contract, added_balance.symbol));
    check(collateral_itr != collateral_idx.end(), "not found collateral in add balance");
```

```
collateral_idx.modify(collateral_itr, _self, [&](auto& c) {
    c.balance += added_balance;
    c.total_balance += total_added_balance;
    if (c.balance > c.total_balance) {
        c.balance = c.total_balance;
    }
});
check_balance(contract, collateral_itr->sym);
}

void sub_collateral_balance(name contract, asset quantity) {
    sub_collateral_balance(contract, quantity, quantity);
}

void sub_collateral_balance(name contract, asset taken_balance, asset taken_total_balance) {

    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
    auto collateral_itr = collateral_idx.find(utils::get_token_key(contract, taken_balance.symbol));
    check(collateral_itr != collateral_idx.end(), "not found collateral in sub balance");
    check(collateral_itr->balance.amount >= taken_balance.amount, "overdrawn collateral balance");
    check(collateral_itr->total_balance.amount >= taken_total_balance.amount, "overdrawn collateral total balance");

    collateral_idx.modify(collateral_itr, _self, [&](auto& c) {
        c.balance -= taken_balance;
        c.total_balance -= taken_total_balance;
        if (c.balance > c.total_balance) {
            c.total_balance = c.balance;
        }
    });
    check_balance(contract, collateral_itr->sym);
}

void check_balance(name contract, symbol sym) {
    action(permission_level{ _self, "active"_n }, _self, name("checkbalance"), make_tuple(contract, sym)).send();
}

void check_collateral_balance(name contract, symbol sym) {
    auto collateral_idx = _collaterals.get_index<"bytokenkey"_n>();
    auto collateral_itr = collateral_idx.find(utils::get_token_key(contract, sym));
    check(collateral_itr != collateral_idx.end(), "not found collateral");
}
```

```
auto balance = utils::get_balance(contract, _self, sym);
auto total_balance = balance;

if (contract == EOSTOKEN_CONTRACT && sym == EOS_SYMBOL) {
    total_balance += get_rex_eos();
}

check(balance >= collateral_itr->balance && total_balance >= collateral_itr->total_balance, string("uncorrected
balances: ")
    + balance.to_string() + "/" + collateral_itr->balance.to_string() + ","
    + total_balance.to_string() + "/" + collateral_itr->total_balance.to_string());
}

asset get_rex_eos() {
    rex_pool_table rexpoo_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);
    auto rex_itr = rexpoo_table.begin();

    rex_balance_table rexbal_table(EOSIO_ACCOUNT, EOSIO_ACCOUNT.value);
    auto rexbal_it = rexbal_table.find(_self.value);

    auto rex_eos = asset(0, EOS_SYMBOL);
    if (rexbal_it == rexbal_table.end()) {
        return rex_eos;
    }

    const int64_t S0 = rex_itr->total_lendable.amount;
    const int64_t R0 = rex_itr->total_rex.amount;
    rex_eos.amount = ( uint128_t(rexbal_it->rex_balance.amount) * S0 ) / R0;

    return rex_eos;
}
};
```

utils.hpp

```
#pragma once

#include <string>
#include <vector>
#include <tuple>
#include <eosio/eosio.hpp>
#include <types.hpp>
```

```
namespace utils {

    using std::vector;
    using std::string;
    using std::tuple;

    uint128_t get_token_key(name contract, symbol sym) {
        return ((uint128_t)(contract.value) << 64) + sym.raw();
    }

    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;
    }

    asset get_balance(const name &token_contract, const name &owner, const symbol &sym) {
        asset ret = asset( 0, sym );
        accounts_t 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;
        } else {
            print_f("not found eos balance..%-%-%\n", token_contract, owner, sym);
        }
        return ret;
    }

    bool is_valid_token(name contract, symbol sym) {
        stats statstable(contract, sym.code().raw());
        auto existing = statstable.find(sym.code().raw());
        return existing != statstable.end();
    }
}
```

```
}  
}
```



官方网址

www.slowmist.com

电子邮箱

team@slowmist.com

微信公众号

