

EOS 智能合约数据结构 API

EOS 编译、测试指南（四）



扫一扫关注欧链小秘书

EOS 团队于 2017 年 9 月 14 日推出了 Dawn 1.0 release 版本，熟悉 Linux 和 C++ 的开发人员可以在此基础上尝试自己的合约开发了。OracleChain 团队一直以来都在对 EOS 代码进行了编译和测试，陆续发布了三份《EOS 编译、测试指南》，详细介绍了 EOS 的入门知识、基本结构，帮助开发人员学习、研究 EOS。基于 EOS 刚刚推出的 Dawn 1.0，OracleChain 团队进行了测试，本次将以代币的发行为例，详细介绍 EOS 上 WASM 的数据库机制和智能合约的数据库 API，基于此开发人员可以在 EOS 上发行自己的代币和开发钱包应用。

该指南主要依据 EOS 的官方文档，以及欧链团队的实践经验编写，开发者有任何技术问题可以关注欧链小秘书、在欧链科技社区里向 OracleChain 团队提问或者发送邮件到 contact@oraclechain.io。

EOS 文档：<https://eosio.github.io/eos/>

EOS 代码：<https://github.com/EOSIO/eos>

备注：OracleChain 团队使用 Mac 环境进行开发。

EOS.IO 的架构目标，是成为一个高效的消息分发平台，消息被分发到 account（合约）中然后被高效地并发执行。

EOS 底层需要一个高效的沙盒运行环境。沙盒中，底层的脚本语言，执行环

境都是可以被替换的。在白皮书中，EOS 宣传支持 EVM 和 WASM 两种沙盒运行环境。目前所知，WASM 的运行效率比 EVM 高很多，这点为 EOS 提供了有力的竞争条件。高效的执行环境，提供了更快的交易反馈时间，从而在单个出块周期处理更多的业务，可以运行更大规模的运算。

一、智能合约的数据库 API

EOS 的合约执行依赖于 WASM 的沙盒运行环境。

上一份指南中，OracleChain 团队分析了 block 数据结构的存储，EOS 使用 `boost::interprocess`，将 block 信息通过文件映射存储到磁盘上。

而在 EOS 合约执行过程中，所有的 `account` 状态都存储在沙盒的运行内存中。EOS 对 `account` 的操作，通过调用 `contract/eoslib/db.hpp` 的 `table` 模版类，进行类似于传统的增删查改操作。`db.hpp` 通过一个 `c` 接口，接入 WASM 沙盒环境，实现了跨平台的储存机制。WASM 对数据存储的具体实现，依赖 `chainbase.hpp` 中，`database` 类实现的 `mmap` 数据存储接口。

`account` 的状态数据，均存储在一个 `table` 的数据结构中，我们可以在 `db.hpp` 中可以看到对其的实现：

```
template<uint64_t scope, uint64_t code, uint64_t table, typename Record,
typename PrimaryType, typename SecondaryType = void>
struct Table {
...
}
struct Table<scope,code,table,Record,PrimaryType,void>
{
...
}
```

由此可见智能合约的数据结构大致如下：

```
* - **scope** - an account where the data is stored
```

```
* - code - the account name which has write permission
* - table - a name for the table that is being stored
* - record - a row in the table
* - PrimaryType - primary key in one record
* - SecondaryType - secondary key in one record
```

开发者使用 `db.hpp` 调用中 C 或 C++ 包装的 `api`，来实现数据存取。

在每一条交易信息中会使用 `scope` 指定哪些账号需要对该消息读、写。`code` 决定了对应 `account` 可以采取的操作，这种操作权限和代码分离的设计提升了安全性。同时，如果修改了不在 `scope` 中的 `account` 的信息或者执行了非 `code` 的操作，会直接导致该交易失败。

以代币的发行实例来看 (代码部分参见 `contract/currency.cpp`)，代币的所有账户都以 `Table` 的形式存储：

```
using Accounts = Table<N(currency),N(currency),N(account),Account,uint64_t>;
```

合约文件中，提供了初始化代币和代币转账接口，当代币合约被存储时，各节点会自动执行一次合约中的 `init` 接口。当每次合约被调用，将调用合约中的 `apply` 接口：

```
extern "C" {
    void init() {
        storeAccount(N(currency),Account( CurrencyTokens(1000ll*1000ll*1000ll) ));
    }
    void apply( uint64_t code, uint64_t action ) {
        if( code == N(currency) ) {
            if( action == N(transfer) )
                currency::apply_currency_transfer(currentMessage<
TOKEN_NAME::Transfer >());
        }
    }
}
```

`table` 将不同 `scope/code/table` 的 `account` 数据，存储到了不同的内存映射中。

在 `currency.cpp` 合约中，`eos` 使用 `table` 实现了代币账户下各种业务逻辑。

当用户发行 `currency` 合约时，会用工具把 `cpp` 生成 `currency.wast.cpp` 文件，

同时需要一个 `abi` 文件作为第三方输入到合约的数据接口，接着 `wasm-jit` 会把 `wast` 代码，连同 `abi` 接口，一起存储到 `tx` 的 `message` 中，让其他节点验证并存储此合约，等待合约被调用。

以下代码中，实现了打包 `currency` 合约到 `transaction` 中的流程：

```
types::setcode handler;
handler.account = "currency";

auto wasm = assemble_wast( currency_wast );
handler.code.resize(wasm.size());
memcpy( handler.code.data(), wasm.data(), wasm.size() );

eos::chain::SignedTransaction trx;
trx.scope = {"currency"};
trx.messages.resize(1);
trx.messages[0].code = config::EosContractName;

trx.messages[0].authorization.emplace_back(types::AccountPermission{"currency", "active"});
transaction_set_message(trx, 0, "setcode", handler);
trx.expiration = chain.head_block_time() + 100;
transaction_set_reference_block(trx, chain.head_block_id());
chain.push_transaction(trx);
chain.produce_blocks(1);
```

二、智能合约数据存储实例

智能合约中基础的功能之一是 `token` 在某种规则下的转移。以 EOS 提供的 `token.cpp` 为例，定义了 `eos token` 的数据结构：`typedef eos::token<uint64_t,N(eos)> Tokens;`

以 `currency` 合约为例。合约中，也用类 `token` 模版类生成了代币 `currency`：`typedef eos::token<uint64_t,N(currency)> CurrencyTokens;`

有了 `eos token` 和我们发行的子代币，我们就能编写合约，让用户使用不同的代币进行交易。在 `currency.cpp` 或 `exchange.cpp` 中，`eos` 实现了发行代币、代

币流通、兑换功能。

eos 和自定义 currency 在流通时，都使用一个类似的 Transfer 结构体：

```
struct Transfer{
    AccountName  from;
    AccountName  to;
    Tokens       quantity;
};
```

这样，在转账时，调用 currency.cpp 中实现的 abi 接口，传入 Transfer 结构

表明想要转账的 token 数量：

```
Transfer MeToYou;
MeToYou.from = N(Me);
MeToYou.to = N(You);
MeToYou.quantity = Tokens(100);
```

当 eos 的合约处理接收到这样的请求时，会调用相关流程完成对对应 token 的处理。

```
void apply_transfer( const Transfer& transfer ) {
    auto from = getAccount( transfer.from );
    auto to    = getAccount( transfer.to );
    from.balance -= transfer.quantity;
    to.balance   += transfer.quantity;
    assertion    storeAccount( transfer.from, from );
    storeAccount( transfer.to, to );
}
```

最终存储结果将保存到沙盒的内存中。

三、智能合约数据库的持久化

在沙盒机制中，当我们运行一个合约、发行一个代币时，EOS 为我们提供的一些基础运行框架。其中最重要的两个：第一，实现了平台无关的 account 存储机制；第二，提供了一个 account 间结算的业务平台。同时 EOS 会将沙盒里面的数据存储接口储存在具体物理设备上来，实现数据的持久化。

在 chain/wasm_interface.cpp 中，对接了 wasm 的 context，并使用 context 获取到 db.hpp 中实现的数据存储接口，然后将这些接口实现到了 message_handling_contexts.hpp 中。

```
chain/wasm_interface.cpp:
#define DEFINE_RECORD_UPDATE_FUNCTIONS(OBJTYPE, INDEX) \

DEFINE_INTRINSIC_FUNCTION4(env,store_##OBJTYPE,store_##OBJTYPE,i32,i64,s
ope,i64,table,i32,valueptr,i32,valuelen) { \
    UPDATE_RECORD(store_record, INDEX, valuelen); \
} \

DEFINE_INTRINSIC_FUNCTION4(env,update_##OBJTYPE,update_##OBJTYPE,i32,i64,s
cope,i64,table,i32,valueptr,i32,valuelen) { \
    UPDATE_RECORD(update_record, INDEX, valuelen); \
} \

DEFINE_INTRINSIC_FUNCTION3(env,remove_##OBJTYPE,remove_##OBJTYPE,i32,i64,
scope,i64,table,i32,valueptr) { \
    UPDATE_RECORD(remove_record,      INDEX,      sizeof(typename
INDEX::value_type::key_type)*INDEX::value_type::number_of_keys); \
}

```

```
message_handling_contexts.hpp:
int32_t update_record( Name scope, Name code, Name table, typename
ObjectType::key_type *keys, char* value, uint32_t valuelen )
int32_t remove_record( Name scope, Name code, Name table, typename
ObjectType::key_type* keys, char* value, uint32_t valuelen )
int32_t load_record( Name scope, Name code, Name table, typename
IndexType::value_type::key_type* keys, char* value, uint32_t valuelen )

```

这样后面的处理流程就比较清晰了。当合约在读取数据时，将调用 message_handling_contexts.hpp 的 load_recod 接口：

```
template <typename IndexType, typename Scope>
int32_t load_record( Name scope, Name code, Name table, typename
IndexType::value_type::key_type* keys, char* value, uint32_t valuelen ) {

```

```
const auto& idx = db.get_index<IndexType, Scope>();
auto tuple = load_record_tuple<typename IndexType::value_type,
Scope>::get(scope, code, table, keys);
auto itr = idx.lower_bound(tuple);
...
}
```

上面 `load_record` 代码中，调用了 `db.get_index` 方法。此处的 `db` 也就是 `chainbase/chainbase.hpp` 中实现的 `database` 类。`database` 中使用了 `boost` 的 `managed_mapped_file`，实现了对数据的存储和读取的接口。

在 EOS 提供的插件 `plugins/chain_plugin/chain_plugin.hpp` 中提供了一种从数据库读取 `table` 的方法：

```
get_table_rows_result get_table_rows( const get_table_rows_params&
params )const;
```

利用这个方法开发者就能读取到合约目前的所有状态，开发属于自己的钱包了。

四、总结

EOS.IO 发布的 Dawn 1.0 版本已经提供了开发智能合约的基本 API，本次 OracleChain 团队从数据库结构到持久化方法介绍了 EOS 智能合约的数据库 API。基于这些 API，开发者就可以开发出自己的钱包。