EOSIO Smart Contracts

cc32d9 / Europechain

Execution workflow

- nodeos executes each action sequentially.
- An action can spawn new inline actions. They are executed after the action finishes.
- If any of actions throws an exception, the whole transaction fails.
- Each action is executed in its own VM instance

apply(), code entry point

- Smart contract always exports one function: apply().
- CDT provides a convenience wrapper.

```
extern "C" void apply(uint64_t receiver, uint64_t code, uint64_t action) {
...
}
```

`apply()` arguments

- **`receiver`**: account where the action was executed.
- **`code`**: our “self”, the account that is executing the code in this VM.
- **`action`**: name of the action, such as “transfer”, encoded as uint64.

`apply()` body

- The standard CDT wrapper instantiates the contract class object, deserializes action data, and calls a corresponding method in the object.
- Writing your own `apply()` is only needed if C++ API is not used (for example, while porting EOSIO to Rust or Typescript)

Inline actions

- Two types of inline actions:
  - **`require_recipient(x)`** leaves a trace in history of *x*. If *x* is a contract, the same action that we're executing is executed on *x*.
  - **`action(auth, account, aname, data)`** executes action aname on specified contract account.

Inline actions execution

- Inline actions are executed **after** the current action finishes, each in a separate VM.
- They are executed in nested order. But it’s not recommended to rely on the order.

escrowescrow contract

- https://github.com/eos-geneva/escrowescrow

Contract class

```
// CONTRACT is a macro equal to class [[eosio::contract]]

CONTRACT escrowescrow : public eosio::contract {
 public:
  escrowescrow( name self, name code, datastream<const char*> ds ):
    contract(self, code, ds),
    _deals(self, self.value)
      {}

// here the class has multi-index instances as members, and they are all
// instantiated within the constructor. You may choose to instantiate the multi-index
// within an action. If the scope depends on action arguments, the multi-index
// has to be instantiated within the action method.
```

Action handler

```
// ACTION is a macro equal to [[eosio::action]]
// it generates a dispatcher code that instantiates our class and calls this method
// when the action “newdeal” is called on contract account

// ABI compiler translates method arguments into ABI definition for the action

  ACTION newdeal(name creator, string description, name tkcontract, asset& quantity,
                 name buyer, name seller, name arbiter, uint32_t days)
  {
    require_auth(creator);

// require_auth(x) throws exception if the actor does not have “active” permission
// for account x

// also has_auth(x) returns true or false depending on actor permission

// other than “active” permission can be requested specifically:
// require_auth(permission_level(owner, name("watchdog")));
```

Notification handler

```
  // Accept funds for a deal
  [[eosio::on_notify("*::transfer")]]
    void transfer_handler (name from, name to, asset quantity, string memo) {
    if(to == _self) {
      check(memo.length() > 0, "Memo must contain a valid deal ID");

// on_notify can specify a specific contract name, such as “eosio.token”,
// or star to match all “transfer” action notifications

// Notifications can be a result of require_recipient(ourname) on transfer receiver’s
// contract, so it is important to check if the transfer receiver is us (to == _self)
```

Multi-index

- Binary tree with uint64 primary index
- Rows are serialized, row structure defined by contract
- Functional indexes: index value is calculated for each row
- Secondary non-unique indexes (`uint64_t`, `uint128_t`, `eosio::checksum256`, `double`, `long double`)

Scope

- Scope is an additional uint64 dimension: each scope value defines a new binary tree multi-index, with its own primary key space.
- Typically scope is a **`name`** that is equal to contract account name.
- Token contracts use scope as balance owner account.

Multi-index example

```
  struct [[eosio::table("deals")]] deal {
    uint64_t       id;
    name           created_by;
    string         description;
    extended_asset price;
    name           buyer;
    name           seller;
    name           arbiter;
    uint32_t       days;
    time_point_sec funded;
    time_point_sec expires;
    uint16_t       flags;
    string         delivery_memo;
    auto primary_key()const { return id; }
    uint64_t get_expires()const { return expires.utc_seconds; }
    uint64_t get_arbiter()const { return arbiter.value; }
  };

  typedef eosio::multi_index<
    name("deals"), deal,
    indexed_by<name("expires"), const_mem_fun<deal, uint64_t, &deal::get_expires>>,
    indexed_by<name("arbiters"), const_mem_fun<deal, uint64_t, &deal::get_arbiter>>>
deals;
```

Instantiation

```
// Multi-index constructor: multi_index( name code, uint64_t scope )

// “self” is the account name running the contract
// value is the uint64 member of struct name

deals _deals(self, self.value);
```

Index operations

- **`emplace()`** and **`modify()`** take lambda functions that modify struct fields.
- **`find()`** returns an iterator pointing to the element or to the end of index
- **`get()`** returns a struct reference or throws exception if not found
- **`erase()`** takes an iterator pointing to a row

Index operations (cont.)

- `lower_bound(x)` returns an iterator pointing to the first element with key not less than x, or end() if such an element does not exist.
- `upper_bound(x)` returns an iterator pointing to the first element with key greater than x, or end() if such an element does not exist.
- `begin(x)` returns an iterator pointing to the first element.
- `end(x)` returns an iterator pointing to the element past the last one.

Inserting a row

```
// emplace returns an iterator pointing to the newly created element,
// so it can be immediately used

    auto idx = _deals.emplace(creator, [&]( auto& d ) {
        d.id = id;
        d.created_by = creator;
        d.description = description;
        d.price.contract = tkcontract;
        d.price.quantity = quantity;
        d.buyer = buyer;
        d.seller = seller;
        d.arbiter = arbiter;
        d.days = days;
        d.expires = time_point_sec(current_time_point()) + NEW_DEAL_EXPIRES;
        d.flags = 0;
        if( creator == buyer ) {
          d.flags |= BUYER_ACCEPTED_FLAG;
        } else if ( creator == seller ) {
          d.flags |= SELLER_ACCEPTED_FLAG;
        }
      });
    _notify(name("new"), "New deal created", *idx);
```

Finding and modifying a row

```
// find a row by primary index value and throw exception if not found
auto dealitr = _deals.find(deal_id);
check(dealitr != _deals.end(), "Cannot find deal_id");

  // modifier lambda function sets the fields in the struct
  _deals.modify( *dealitr, party, [&]( auto& item ) {
      item.flags = flags;
      item.expires = time_point_sec(current_time_point()) + ACCEPTED_DEAL_EXPIRES;
    });

// this is the only way to modify a row in multi-index because it's stored
// in serialized form. The modify() function deserializes it, passes to the
// modifier, and serializes back
```

Finding by lower bound

```
ACTION wipeexpired(uint16_t count)
{
  bool done_something = false;
  auto _now = time_point_sec(current_time_point());
  auto dealidx = _deals.get_index<name("expires")>();
  auto dealitr = dealidx.lower_bound(1); // 0 is for deals locked for arbitration
  while( count-- > 0 && dealitr != dealidx.end() && dealitr->expires <= _now ) {
    _deal_expired(*dealitr);
    dealitr = dealidx.lower_bound(1);
    done_something = true;
  }
```

Secondary index example

```
// example from https://github.com/cc32d9/dappscatalog

    auto codeidx = _prices.get_index<name("contract")>();
    auto itr = codeidx.lower_bound(contract.value);
    while(itr != codeidx.end() &&
          itr->contract == contract &&
          itr->pnewentry.symbol != price.symbol ) {
      itr++;
    }

// “contract” is a secondary non-unique index, and [contract, price.symbol] are
// unique combinations. The loop starts from the first entry matching the contract
// and finds an entry matching the price symbol
```

Questions?