Smart contract as an actor

In previous chapters, we were talking about the actor model and how is it implemented in the blockchain. Now it is time to look closer into the typical contract structure to understand how different features of the actor model are mapped to it.

This will not be a step-by-step guide on contract creation, as it is a topic for the series itself. It would be going through contract elements roughly to visualize how to handle architecture in the actor model.

An explanation how are CosmWasm execution performed can be found inSEMANTICS.md , here I will try to explain it step by step.

The state

As before we would start with the state. Previously we were working with thetg4-group contract, so let's start by looking at its code. Go tocw-plus/contracts/cw4-group/src . The folder structure should look like this:

ꟸ src ├── contract.rs ├── error.rs ├── helpers.rs ├── lib.rs ├── msg.rs └── state.rs As you may already figure out, we want to check thestate.rs first.

The most important thing here is couple of constants:ADMIN ,HOOKS ,TOTAL , andMEMBERS . Every one of such constants represents a single portion of the contract state - as tables in databases. The types of those constants represent what kind of table is this. The most basic ones areItem , which keeps zero or one element of a given type, andMapwhich is a key-value map.

You can seeItem is used to keep an admin, and some other data:HOOKS , andTOTAL .HOOKS is used by thecw4-group to allow subscription to any changes to a group - a contract can be added as a hook, so when the group changes, a message is sent to it. TheTOTAL is just a sum of all members' weights.

TheMEMBERS in the group contract is theSnapshotMap - as you can imagine it is aMap , with some steroids - this particular one, gives us access to the state of the map at some point of history, accessing it by the blockchainheight .height is the count of blocks created since the begging of blockchain, and it is the most atomic time representation in smart contracts. There is a way to access the clock time in them, but everything happening in a single block is considered happening in the same moment.

Other types of storage objects not used in group contracts are:

- IndexedMap
-
    - another map type, which allows accessing values
- by a variety of keys
- IndexedSnapshotMap
- -IndexedMap
- andSnapshotMap
- married

What is very important - every state type in the contract is accessed using some name. All of those types are not containers, just accessors to the state. Do you remember that I told you before, that blockchain is our database? And that is correct! All those types are just ORM to this database - when we will use them to get actual data from it, we would pass a specialState object to them, so they can retrieve items from it.

You may ask - why all those data for a contract are not auto-fetched by whatever is running it? That is a good question. The reason is, that we want contracts to be lazy with fetching. Copying data is a very expensive operation, and for everything happening on it, someone has to pay - it is realized by gas cost. I told you before, that as a contract developer you don't need to worry about gas at all, but it was only partially true. You don't need to know exactly how gas is calculated, but by lowering your gas cost, you would may execution of your contracts cheaper which is typically a good thing. One good practice to achieve that is to avoid fetching data you will not use in a particular call.

Messages

In a blockchain, contracts communicate with each other by some JSON messages. They are defined in most contracts in themsg.rs file. Take a look at it.

There are three types on it, let's go through them one by one. The first one is anInstantiateMsg . This is the one, which is sent on contract instantiation. It typically contains some data which are needed to properly initialize it. In most cases, it is just a simple structure.

Then there are two enums:ExecuteMsg , andQueryMsg . They are enums because every single variant of them represents a different message which can be sent. For example theExecuteMsg::UpdateAdmin corresponds to theupdate_admin message we were sending previously.

Note, that all the messages are attributed with#[derive(Serialize, Deserialize)] , and#[serde(rename_all="snake_case")] . Those attributes come from theserde create, and they help us with deserialization of them (and serialization in case of sending them to other contracts). The second one is not required, but it allows us to keep a camel-case style in our Rust code, and yet still have JSONs encoded with a snake-case style more typical to this format.

I encourage you to take a closer look at theserde documentation, like everything it is there, can be used with the messages.

One important thing to notice - empty variants of those enums, tends to use the empty brackets, likeAdmin {} instead of more RustyAdmin . It is on purpose, to make JSONs cleaner, and it is related to howserde serializes enum.

Also worth noting is that those message types are not set in stone, they can be anything. This is just a convention, but sometimes you would see things likeExecuteCw4Msg , or similar. Just keep in mind, to keep your message name obvious in terms of their purpose - sticking toExecuteMsg /QueryMsg is generally a good idea.

Entry points

So now, when we have our contract message, we need a way to handle them. They are send to our contract via entry points. There are three entry points in thecw4-group contract:

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

instantiate (deps :

DepsMut , env :

Env , _info :

MessageInfo , msg :

InstantiateMsg ,)

->

Result < Response ,

ContractError

{ // ... }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

execute (deps :

DepsMut , env :

Env , info :

MessageInfo , msg :

ExecuteMsg ,)

->

Result < Response ,

ContractError

{ // .. }

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

query (deps :

Deps , _env :

Env , msg :

QueryMsg)

->

StdResult < Binary

{ // .. } Those functions are called by the CosmWasm virtual machine when a message is to be handled by contract. You can think about them as themain function of normal programs, except they have a signature that better describes the blockchain itself.

What is very important, is that the names of those entry points (similarly to themain function) are fixed - it is relevant, so the virtual machine knows exactly what to call.

So let's start with the first line. Every entry point is attributed with#[cfg_attr(not(feature = "library"), entry_point)] . It may look a bit scary, but it is just a conditional equivalent of#[entry_point] - the the attribute would be there if and only if the "library" feature is not set. We do this, to be able to use our contracts as dependencies for other contract - the final binary can contain only one copy of each entry point, so we make sure, that only the top-level one is compiled without this feature.

Theentry_point attribute is a macro which generates some boilerplate. As the binary is run by WASM virtual machine, it doesn't know much about Rust types - the actual entry point signatures are very inconvenient to use. To overcome this issue, there is a macro created, which generates entry points for us, and those entry points are just calling our functions.

Now take a look at functions arguments. Every single entry point takes as the last argument a message which triggered the execution of it (except forreply - I would explain it later). In addition to that, there are additional arguments provided by blockchain:

- Deps
- orDepsMut
- object is the gateway to the world outside the smart contract context. It allows
- accessing the contract state, as well as querying other contracts, and
- also delivers anApi
- object with a couple of useful utility functions.
- The difference is, thatDepsMut
- allows updating state, whileDeps
- allows only to look at it.
- Env
- object delivers information about the blockchain state in the
- moment of execution - its height, the timestamp of execution and information
- about the executing contract itself.
- MessageInfo
- object is information about the contract call - it
- contains the address which sends the message, and the funds send with the
- message.

Keep in mind, that the signatures of those functions are fixed (except the messages type), so you cannot interchangeDeps withDepsMut to update contract state in the query call.

The last portion of entry points is the return type. Every entry point returns aResult type, with any error which can be turned into a string - in case of contract failure, the returned error is just logged. In most cases, the error type is defined for a contract itself, typically using athiserror crate.Thiserror is not required here, but strongly recommended - using it makes the error definition very straightforward, and also improves the testability of the contract.

The important thing is theOk part ofResult . Let's start with thequery because this one is the simplest. The query always returns theBinary object on theOk case, which would contain just serialized response. The common way to create it is just calling ato_binary method on an object implementingserde::Serialize , and they are typically defined inmsg.rs next to messages types.

Slightly more complex is the return type returned by any other entry point - thecosmwasm_std::Response type. This one keeps everything needed to complete contract execution. There are three chunks of information in that.

The first one is anevents field. It contains all events, which would be emitted to the blockchain as a result of the execution. Events have a really simple structure: they have a type, which is just a string, and a list of attributes which are just string-string key-value pairs.

You can notice that there is anotherattributes field on theResponse This is just for convenience - most executions would return only a single event, and to make it a bit easier to operate one, there is a set of attributes directly on response. All of them would be converted to a singlewasm event which would be emitted. Because of that, I considerevents andattributes to be the same chunk of data.

Then we have the messages field, ofSubMsg type. This one is the clue of cross-contact communication. Those messages would be sent to the contracts after processing. What is important - the whole execution is not finished, unless the processing of all sub-messages scheduled by the contract finishes. So if the group contract sends some messages as a result ofupdate_members execution, the execution would be considered done only if all the messages sent by it would also be handled (even if they failed).

So when all the sub-messages sent by contract are processed, then all the attributes generated by all sub calls and top-level calls are collected and reported to the blockchain. But there is one additional information - thedata . So this is anotherBinary field, just like the result of a query call, and just like it, it typically contains serialized JSON. Every contract call can return some additional information in any format, You may ask - in this case, why do we even bother returning attributes? It is because of a completely different way of emitting events and data. Any attributes emitted by the contract would be visible on blockchain eventually (unless the whole message handling fails). So if your contract emitted some event as a result of being sub call of some bigger use case, the event would always be there visible to everyone. This is not true for data. Every contract call would return only singledata chunk, and it has to decide if it would just forward thedata field of one of the sub-calls, or maybe it would construct something by itself. I would explain it in a bit more detail in a while.

Sending submessages

I don't want to go into details of theResponse API, as it can be read directly from documentation, but I want to take a bit closer look to the part about sending messages.

The first function to use here isadd_message , which takes as an argument theCosmosMsg (or rather anything convertible to it). Message added to response this way, would be sent and processed, and its execution would not affect the result of the contract at all.

The other function to use isadd_submessage , taking aSubMsg argument. It doesn't differ much fromadd_message - SubMsg just wraps theCosmosMsg , adding some info to it: theid field, andreply_on . There is also agas_limit thing, but it is not so important - it just causes sub-message processing to fail early if the gas threshold is reached.

The simple thing isreply_on - it describes if thereply message should be sent on processing success, on failure, or both.

Theid field is an equivalent of order id in our KFC example from the very beginning. If you send multiple different sub-messages, it would be impossible to distinguish them without that field. It would not be even possible to figure out, what type of original message is reply handling! This is why theid field is there - sending a sub-message you can set it to any value, and then on the reply, you can figure out what is happening based on this field.

An important note here - you don't need to worry about some sophisticated way of generating ids. Remember, that the whole processing is atomic, and only one execution can be in progress at once. In most cases, your contract sends a fixed number of sub-messages on very concrete executions. Because of that, you can hardcode most of those ids while sending (preferably using some constant).

To easily create sub-messages, instead of setting all the fields separately, you would typically use helper constructors:SubMsg::reply_on_success ,SubMsg::reply_on_error andSubMsg::reply_always .

CosmosMsg

If you took a look at theCosmosMsg type, you could be very surprised - there are so many variants of them, and it is not obvious how do they relate to communication with other contracts.

The message you are looking for this is theWasmMsg (CosmosMsg::Wasm variant). This one is very much similar to what we already know - it has a couple of variants of operation to be performed by contracts:Execute , but alsoInstantiate (so we can create new contracts in contract executions), and alsoMigrate ,UpdateAdmin , andClearAdmin - those are used to manage migrations (will tell a bit about them at the end of this chapter).

Another interesting message is theBankMsg (CosmosMsg::Bank). This one allows a contract to transfer native tokens to other contracts (or burn them - equivalent to transferring them to some black whole contract). I like to think about it as sending a message to very special contract responsible for handling native tokens - this is not a true contract, as it is handled by the blockchain itself, but at least to me it simplifies things.

Other variants ofCosmosMsg are not very interesting for now. TheCustom one is there to allow other CosmWasm-based blockchains to add some blockchain-handled variant of the message. This is a reason why most message related types in CosmWasm are generic over someT - this is just a blockchain-specific type of message. We will never use it in thewasmd . All other messages are related to advanced CosmWasm features, and I will not describe them here.

Reply handling

So now that we know how to send sub-message, it is time to talk about handling the reply. When sub-message processing is finished, and it is requested to reply, the contract is called with an entry point:

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

reply (deps :

DepsMut , env :

Env , msg :

Reply)

->

Result < Response ,

ContractError

{ // ... } TheDepsMut , andEnv arguments are already familiar, but there is the new one, substituting the typical message argument: thecosmwasm_std::Reply .

This is a type representing the execution status of the sub-message. It is slightly processedcosmwasm_std::Response . First important thing it contains, is anid - the same, which you set sending sub-message, so now you can identify your response. The other one is theContractResult , which is very similar to the RustResult type, except it is there for serialization purposes. You can easily convert it into aResult with aninto_result function.

In the error case ofContracResult , there is a string - as I mentioned before, errors are converted to string right after execution. TheOk case containsSubMsgExecutionResponse with two fields:events emitted by sub-call, and thedata field embedded on response.

As said before, you never need to worry about forwarding events - CosmWasm would do it anyway. Thedata however is another story. As mentioned before, every call would return only a single data object. In the case of sending sub-messages and not capturing a reply, it would always be whatever is returned by the top-level message. But it is not a case whenreply is called. If a reply is called, then it is a function deciding about the finaldata . It can decide to either forward the data from the sub-message (by returningNone) or to overwrite it. It cannot choose, to return data from the original execution processing - if the contract sends sub-messages waiting for replies, it is supposed to not return any data, unless replies are called.

But what happens, if multiple sub-messages are sent? What would the finaldata contain? The rule is - the last non-None. All sub-messages are always called in the order of adding them to theResponse . As the order is deterministic and well defined, it is always easy to predict which reply would be used.

Migrations

I mentioned migrations earlier when describing theWasmMsg . So migration is another action possible to be performed by contracts, which is kind of similar to instantiate. In software engineering it is a common thing, to release an updated version of applications. It is also a case in the blockchain - SmartContract can be updated with some new features. In such cases, new code is uploaded, and the contract is migrated - so it knows, that from this point, its messages are handled by another, updated contract code.

However, it may be, that the contract state used by the older version of the contract differs from the new one. It is not a problem if some info was added (for example some additional map - it would be just empty right after migration). But the problem is, when the state changes, for example, the field is renamed. In such a case, every contract execution would fail because of (de)serialization problems. Or even more subtle cases, like adding a map, but one which should be synchronized with the whole contract state, not empty.

This is the purpose of themigration entry point. It looks like this:

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

migrate (deps :

DepsMut , env :

Env , msg :

MigrateMsg)

->

Result < Response < T

,

ContracError

{ // .. } MigrateMsg is the type defined by the contract inmsg.rs . Themigrate entry point would be called at the moment of performing the migration, and it is responsible for making sure, the state is correct after the migration. It is very similar to schema migrations in traditional database applications. And it is also kind of difficult, because of version management involved - you can never assume, that you are migrating a contract from the previous version - it can be migrated from any version, released anytime - even later than that version we are migrating to!

It is worth binging back one issue from the past - the contract admin. Do you remember the--no-admin flag we set previously on every contract instantiation? It made our contract unmigrateable. Migrations can be performed only by contract admin. To be able to use it, you should pass--admin address flag instead, withaddress being the address which would be able to perform migrations.

Sudo

Sudo is the last possible entry point inCosmWasm , and it is the one we would never use inwasmd . It is equivalent toCosmosMsg::Custom , but instead of being a special blockchain-specific message to be sent and handled by a blockchain itself, it is now a special blockchain-specific message sent by blockchain to contract in some conditions. There are many uses for those, but I will not cover of them, because it is not related toCosmWasm itself. The signature ofsudo looks like:

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

sudo (deps :

DepsMut , env :

Env , msg :

SudoMsg)

->

Result < Response ,

ContractError

{ // .. } The important difference is, that becausesudo messages are blockchain specific, theSudoMsg type is typically defined by some blockchain helper crate, not the contract itself. Previous Actors in blockchain Next CosmWasm IDE Tutorial * The state * Messages * Entry points * Sending submessages * CosmosMsg * Reply handling * Migrations * Sudo