Result and Option

In Rust, usingResult andOption types is very common. If you've usedfp-ts in Typescript, functional features in a language like Scala, or a strongly-typed functional language like Haskell, these should be familiar to you.

Both of these types are containers, or enum types. That is, they contain other values within variants.

If you're familiar with Algebraic Data types, it might be enough to say that these are effectively:

enum

Option < T

{ Some (T) , // existence None ,

// non-existence }

enum

Result < T ,

E

{ Ok (T) , // success Err (E) ,

// failure }

Result

Result is an enum type,Result , where bothT andE are generics, representing success and failure. These are called like so:

- Ok(T)
-
    - aResult
- container which has succeeded, containingT
- Err(E)
-
    - aResult
- container which has failed, containingE

A result type is conceptually similar to anEither in other functional languages. Many contract entry points are typedResult In this case,Response is theRight orSuccess branch, whileContractError is theLeft or failure case.

Result types are not just used in entry points and handlers, however.

They are often used in functions used to match enums, for example inexecute entry points.

We can see in CW20-base thatexecute is typedResult . Let's look at the function call in the first branch, which matchesExecuteMsg::Transfer .

execute_transfer (deps , env , info , recipient , amount) We might expect the match branches to call functions that are typed the same as the entry point. Andthey are .

pub

fn

execute_transfer (deps :

DepsMut , _env :

Env , info :

MessageInfo , recipient :

String , amount :

Uint128 ,)

->

Result < Response ,

ContractError

{ if amount ==

Uint128 :: zero ( )

{ return

Err (ContractError :: InvalidZeroAmount

{ }) ; }

let rcpt_addr = deps . api . addr_validate (& recipient) ? ;

BALANCES . update (deps . storage , & info . sender , | balance :

Option < Uint128

|

->

StdResult < _

{ Ok (balance . unwrap_or_default ( ) . checked_sub (amount) ?) } ,) ? ; BALANCES . update (deps . storage , & rcpt_addr , | balance :

Option < Uint128

|

->

StdResult < _

{

Ok (balance . unwrap_or_default ( )

+ amount)

} ,) ? ;

let res =

Response :: new ( ) . add_attribute ("action" ,

"transfer") . add_attribute ("from" , info . sender) . add_attribute ("to" , recipient) . add_attribute ("amount" , amount) ; Ok (res) }

StdResult

It's also worth being aware ofStdResult . This is used often inquery handlers and functions that are called from them.

For example, in thenameservice contract you can see theStdResult , which is likeResult , except without a defined error branch:

[cfg_attr(not(feature =

"library"), entry_point)] pub

fn

query (deps :

Deps , env :

Env , msg :

QueryMsg)

->

StdResult < Binary

{ match msg { QueryMsg :: ResolveRecord

{ name }

=>

query_resolver (deps , env , name) , QueryMsg :: Config

{ }

=>

to_binary (& config_read (deps . storage) . load ( ) ?) , } } Let's see the implementation ofquery_resolver .

fn

query_resolver (deps :

Deps , _env :

Env , name :

String)

->

StdResult < Binary

{ let key = name . as_bytes ( ) ;

let address =

match

resolver_read (deps . storage) . may_load (key) ?

{ Some (record)

=>

Some (String :: from (& record . owner)) , None

=>

None , } ; let resp =

ResolveRecordResponse

{ address } ;

to_binary (& resp) } The key takeaway here is that generally you can ignore container types, so long as they all line up. Once all your types are correct, your code will compile. Then you simply need to match or unwrap your container types correctly to work with the values contained within.

Option

In Rust, there is no concept ofnil ornull , unlike most other mainstream programming languages. Instead, you have theOption type, which encodes the idea of existence or non-existence into a container type.

Option is an enum type, with two variants:

- Some()
- -Some
- wraps an inner value, which can be accessed via.unwrap()
- . You will see this, as
- well as matching, all over rust code.
- None
- -None

It's useful for doing things like expressing that a value might not exist for a key in a struct:

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

pub

struct

Config

{ pub purchase_price :

Option < Coin

, pub transfer_price :

Option < Coin

, } The source ishere . We can see why this might be - these values come frominstantiation , where the values are alsoOption :

[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

pub

struct

InstantiateMsg

{ pub purchase_price :

Option < Coin

, pub transfer_price :

Option < Coin

, } In theResult examples above, we saw an example ofOption usage. When we try and read the storage, there will either be a result, or nothing. To handle situations like this, it's common to use thematch operator to pattern match the two cases:

let address =

match

resolver_read (deps . storage) . may_load (key) ?

{ Some (record)

=>

Some (String :: from (

& record . owner)) , None

=>

None , } ; In cases whereNone being returned would be an error state, convention dictates you should consider throwing an error instead of handling theNone . Previous Submessages Next cw-storage-plus * Result * * StdResult * Option