Errors

In Rust Stylus contracts, error handling is a crucial aspect of writing robust and reliable smart contracts. Rust differentiates between recoverable and unrecoverable errors. Recoverable errors are represented using theResult type, which can either beOk , indicating success, orErr , indicating failure. This allows developers to manage errors gracefully and maintain control over the flow of execution. Unrecoverable errors are handled with thepanic! macro, which stops execution, unwinds the stack, and returns a dataless error.

In Stylus contracts, error types are often explicitly defined, providing clear and structured ways to handle different failure scenarios. This structured approach promotes better error management, ensuring that contracts are secure, maintainable, and behave predictably under various conditions. Similar to Solidity and EVM, errors in Stylus will undo all changes made to the state during a transaction by reverting the transaction. Thus, there are two main types of errors in Rust Stylus contracts:

- Recoverable Errors
- : The Stylus SDK provides features that make using recoverable errors in Rust Stylus contracts convenient. This type of error handling is strongly recommended for Stylus contracts.
- Unrecoverable Errors
- : These can be defined similarly to Rust code but are not recommended for smart contracts if recoverable errors can be used instead.

Learn More

- Solidity docs: Expressions and Control Structures
- #[derive(SolidityError)]
- alloy_sol_types::SolError
- Error handling: Rust book

Recoverable Errors

Recoverable errors are represented using theResult type, which can either beOk , indicating success, orErr , indicating failure. The Stylus SDK provides tools to define custom error types and manage recoverable errors effectively.

Example: Recoverable Errors

Here's a simplified Rust Stylus contract demonstrating how to define and handle recoverable errors:

note This code has yet to be audited. Please use at your own risk.

![cfg_attr(not(feature =

"export-abi"), no_main)] extern

crate

alloc ;

use

alloy_sol_types :: sol ; use

stylus_sdk :: { abi :: Bytes ,

alloy_primitives :: { Address ,

U256 } ,

call :: RawCall ,

prelude :: * } ;

[storage]

[entrypoint]

pub

struct

MultiCall ;

// Declare events and Solidity error types sol!

{ error ArraySizeNotMatch ( ) ; error CallFailed (uint256 call_index) ; }

[derive(SolidityError)]

pub

enum

MultiCallErrors

{ ArraySizeNotMatch (ArraySizeNotMatch) , CallFailed (CallFailed) , }

[public]

impl

MultiCall

{ pub

fn

multicall (& self , addresses :

Vec < Address

, data :

Vec < Bytes

,)

->

Result < Vec < Bytes

,

MultiCallErrors

{ let addr_len = addresses . len ( ) ; let data_len = data . len ( ) ; let

mut results :

Vec < Bytes

=

Vec :: new ( ) ; if addr_len != data_len { return

Err (MultiCallErrors :: ArraySizeNotMatch (ArraySizeNotMatch

{ })) ; } for i in

0 .. addr_len { let result :

Result < Vec < u8

,

Vec < u8

= RawCall :: new ( ) . call (addresses [i] , data [i] . to_vec ( ) . as_slice ( )) ; let data =

match result { Ok (data)

=> data , Err (_data)

=>

return

Err (MultiCallErrors :: CallFailed (CallFailed

{ call_index :

U256 :: from (i)

})) , } ; results . push (data . into ( )) } Ok (results) } } * UsingSolidityError * Derive Macro * : The#[derive(SolidityError)] * attribute is used for theMultiCallErrors * enum, automatically implementing the necessary traits for error handling. * Defining Errors * : Custom errorsArraySizeNotMatch * andCallFailed * is declared inMultiCallErrors * enum.CallFailed * error includes acall_index * parameter to indicate which call failed. * ArraySizeNotMatch Error Handling * : Themulticall * function returnsArraySizeNotMatch * if the size of addresses and data vectors are not equal. * CallFailed Error Handling * : Themulticall * function returns aCallFailed * error with the index of the failed call if any call fails. Note that we're using match to check if the result of the call is an error or a return data. We'll describe match pattern in the further sections.

Unrecoverable Errors

Here are various ways to handle such errors in themulticall function, which calls multiple addresses and panics in different scenarios:

Usingpanic!

Directly panics if the call fails, including the index of the failed call.

for i in

0 .. addr_len { let result =

RawCall :: new ( ) . call (addresses [i] , data [i] . to_vec ( ) . as_slice ( )) ; let data =

match result { Ok (data)

=> data , Err (_data)

=>

panic! ("Call to address {:?} failed at index {}" , addresses [i] , i) , } ; results . push (data . into ( )) ; } Handling Call Failure withpanic! : The function panics if any call fails and the transaction will be reverted without any data.

Usingunwrap

Usesunwrap to handle the result, panicking if the call fails.

for i in

0 .. addr_len { let result =

RawCall :: new ( ) . call (addresses [i] , data [i] . to_vec ( ) . as_slice ( )) . unwrap ( ) ; results . push (result . into ( )) ; } Handling Call Failure withunwrap : The function usesunwrap to panic if any call fails, including the index of the failed call.

Usingmatch

Uses amatch statement to handle the result ofcall , panicking if the call fails.

for i in

0 .. addr_len { let result =

RawCall :: new ( ) . call (addresses [i] , data [i] . to_vec ( ) . as_slice ( )) ; let data =

match result { Ok (data)

=> data , Err (_data)

=>

return

Err (MultiCallErrors :: CallFailed (CallFailed

{ call_index :

U256 :: from (i)

})) , } ; results . push (data . into ( )) ; } Handling Call Failure withmatch : The function uses amatch statement to handle the result ofcall , returning error if any call fails.

Using the?

Operator

Uses the? operator to propagate the error if the call fails, including the index of the failed call.

for i in

0 .. addr_len { let result =

RawCall :: new ( ) . call (addresses [i] , data [i] . to_vec ( ) . as_slice ( )) . map_err (| _ |

MultiCallErrors :: CallFailed (CallFailed

{ call_index :

U256 :: from (i)

})) ? ; results . push (result . into ( )) ; } Handling Call Failure with? Operator: The function uses the? operator to propagate the error if any call fails, including the index of the failed call.

Each method demonstrates a different way to handle unrecoverable errors in themulticall function of a Rust Stylus contract, providing a comprehensive approach to error management.

Note that as mentioned above, it is strongly recommended to use custom error handling instead of unrecoverable error handling.

Boilerplate

src/lib.rs

The lib.rs code can be found at the top of the page in the recoverable error example section.

Cargo.toml

[package] name

=

"stylus-multicall-contract" version

=

"0.1.7" edition

=

"2021"

[dependencies] alloy-primitives

=

"=0.7.6" alloy-sol-types

=

"=0.7.6" stylus-sdk

=

"0.6.0" hex

=

"0.4.3"

[dev-dependencies] tokio

=

{

version

=

"1.12.0" ,

features

=

["full"]

} ethers

=

"2.0" eyre

=

"0.6.8"

[features] export-abi

=

["stylus-sdk/export-abi"]

[[bin]] name

=

"stylus-multicall-contract" path

=

"src/main.rs"

[lib] crate-type

=

["lib" ,

"cdylib"] Edit this page Previous Function Next Events