EVM (general)

Overview

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts, enabling compatibility with Ethereum-based dApps. Sei is an EVM compatible blockchain. Sei's parallelized EVM ensures high performance and efficiency.

Here are some key points about the EVM:

1. Turing Completeness
2. : The EVM is Turing complete, meaning it can execute any computable function. This allows developers to write complex smart contracts.
3. Gas
4. : Transactions and contract executions on the EVM compatible network consume gas. Gas is a measure of computational work, and users pay for it in usei on Sei networks . Gas ensures that malicious or inefficient code doesn't overload the network.
5. Bytecode Execution
6. : Smart contracts are compiled into bytecode (low-level machine-readable instructions) and deployed to the EVM compatible network. The EVM executes this bytecode.

Smart contract languages

The two most popular languages for developing smart contracts on the EVM areSolidity andVyper .

Solidity

- Object-oriented, high-level language for implementing smart contracts.
- Curly-bracket language that has been most profoundly influenced by C++.
- Statically typed (the type of a variable is known at compile time).
- Supports:* Inheritance (you can extend other contracts).
-
  - Libraries (you can create reusable code that you can call from different contracts – like static functions in a static class in other object oriented programming languages).
-
  - Complex user-defined types.

Example solidity contract

// SPDX-License-Identifier: GPL-3.0 pragma solidity

=

0.7 . 0 ;

contract Coin { // The keyword "public" makes variables // accessible from other contracts address

public minter; mapping (address

=>

uint) public balances;

// Events allow clients to react to specific // contract changes you declare event

Sent (address from, address to, uint amount);

// Constructor code is only run when the contract // is created constructor () { minter = msg.sender; }

// Sends an amount of newly created coins to an address // Can only be called by the contract creator function

mint (address receiver ,

uint amount) public { require (msg.sender == minter); require (amount <

1e60); balances[receiver] += amount; }

// Sends an amount of existing coins // from any caller to an address function

send (address receiver ,

uint amount) public { require (amount <= balances[msg.sender] ,

"Insufficient balance."); balances[msg.sender] -= amount; balances[receiver] += amount; emit

Sent (msg.sender , receiver , amount); } }

Vyper

- Pythonic programming language
- Strong typing
- Small and understandable compiler code
- Efficient bytecode generation
- Deliberately has less features than Solidity with the aim of making contracts more secure and easier to audit. Vyper does not support:* Modifiers
-
    - Inheritance
-
    - Inline assembly
-
    - Function overloading
-
    - Operator overloading
-
    - Recursive calling
-
    - Infinite-length loops
-
    - Binary fixed points

Example Vyper contract

Open Auction

Auction params

Beneficiary receives money from the highest bidder

beneficiary :

public (address) auctionStart :

public (uint256) auctionEnd :

public (uint256)

Current state of auction

highestBidder :

public (address) highestBid :

public (uint256)

Set to true at the end, disallows any change

ended :

public (bool)

Keep track of refunded bids so we can follow the withdraw pattern

pendingReturns :

public (HashMap[address, uint256])

Create a simple auction with _bidding_time

seconds bidding time on behalf of the

beneficiary address _beneficiary.

@external def

init (_beneficiary : address ,

_bidding_time : uint256): self . beneficiary = _beneficiary self . auctionStart = block . timestamp self . auctionEnd = self . auctionStart + _bidding_time

Bid on the auction with the value sent

together with this transaction.

The value will only be refunded if the

auction is not won.

@external @payable def

bid ():

Check if bidding period is over.

assert block . timestamp < self . auctionEnd

Check if bid is high enough

assert msg . value

self . highestBid

Track the refund for the previous high bidder

self . pendingReturns [self . highestBidder]

+= self . highestBid

Track new high bid

self . highestBidder = msg . sender self . highestBid = msg . value

Withdraw a previously refunded bid. The withdraw pattern is

used here to avoid a security issue. If refunds were

directly

sent as part of bid(), a malicious bidding contract could block

those refunds and thus block new higher bids from coming in.

@external def

withdraw (): pending_amount : uint256 = self . pendingReturns [msg . sender] self . pendingReturns [msg . sender]

=

0 send (msg.sender, pending_amount)

End the auction and send the highest bid

to the beneficiary.

@external def

endAuction ():

It is a good guideline to structure functions that interact

with other contracts (i.e. they call functions or send ether)

into three phases:

1. checking conditions

2. performing actions (potentially changing conditions)

3. interacting with other contracts

If these phases are mixed up, the other contract could call

back into the current contract and modify the state or cause

effects (ether payout) to be performed multiple times.

If functions called internally include interaction with external

contracts, they also have to be considered interaction with

external contracts.

1. Conditions

Check if auction endtime has been reached

assert block . timestamp

= self . auctionEnd

Check if this function has already been called

assert

not self . ended

2. Effects

self . ended =

True

3. Interaction

send (self.beneficiary, self.highestBid)

Deploying EVM contract on Sei

Since Sei is an EVM compatible chain, existing EVM tooling likehardhat(opens in a new tab) ,foundry forge(opens in a new tab) or other could be re-used.

In this example we will be usingfoundry tooling(opens in a new tab) .

Install thefoundry tooling(opens in a new tab) by following thisInstallation guide(opens in a new tab) .

Create a new project following theCreating New Project Guide(opens in a new tab) .

Also make sure you have a wallet on Sei network.

Once project is created, tweak the contract code to the following, by adding agetCounter function:

// SPDX-License-Identifier: UNLICENSED pragma

solidity ^0.8.13;

contract Counter { uint256

public number;

function

setNumber (uint256 newNumber) public { number = newNumber; }

function

increment () public { number ++ ; }

function

getCount () public

view

returns (uint256) { return number; } } And the test code to the following:

// SPDX-License-Identifier: UNLICENSED pragma

solidity ^0.8.13;

import { Test , console } from

"forge-std/Test.sol" ; import { Counter } from

"../src/Counter.sol" ;

contract

CounterTest

is

Test { Counter public counter;

function

setUp () public { counter =

new

Counter (); counter. setNumber (0); }

function

test_Increment () public { counter. increment (); assertEq (counter. number () ,

1); }

function

testFuzz_SetNumber (uint256 x) public { counter. setNumber (x); assertEq (counter. number () , x); }

function

test_GetCount () public { uint256 initialCount = counter. getCount (); counter. increment (); assertEq (counter. getCount () , initialCount +

1); } } Run the tests with the following command:

forge

test If tests pass, deploy the contract to the Sei chain with the following command:

forge

create

--rpc-url SEI_NODE_URI --mnemonic MNEMONIC src/Counter.sol:Counter WhereSEI_NODE_URI is the URI of the Sei node andMNEMONIC is the mnemonic of the account that will deploy the contract. If you run local Sei node, the address will behttp://localhost:8545 , otherwise you could grab aevm_rpc url from the[registry(opens in a new tab)](registry) . If deployment is successful, you will get the EVM contract address in the output.

[⠊] Compiling... No

files

changed,

compilation

skipped Deployer: 0X_DEPLOYER_ADDRESS Deployed to: 0X_CONTRACT_ADDRESS Transaction

hash: 0X_TX_HASH Let's use thecast command to query the contract:

cast

call 0X_CONTRACT_ADDRESS "getCount()(uint256)"

--rpc-url SEI_NODE_URI The command should return0 as the initial value of the counter.

Now we can use thecast command to call theincrement function:

cast

send 0X_CONTRACT_ADDRESS "increment()"

--mnemonic MNEMONIC --rpc-url SEI_NODE_URI If command is successful, you will get the transaction hash and other info back.

Now let's call thegetCount function again and this case it should return1 .

Calling contract from JS client

To call contract from frontend, you could useethers like:

import {ethers} from

"ethers" ;

const

signer

=

await

getEthSigner (); const

provider

=

await

getProvider (); if (! signer) { console .log ('No signer found'); return ; } const

abi

= [{ "type" :

"function" , "name" :

"setNumber" , "inputs" : [{ "name" :

"newNumber" , "type" :

"uint256" , "internalType" :

"uint256" }] , "outputs" : [] , "stateMutability" :

"nonpayable" } , { "type" :

"function" , "name" :

"getCount" , "inputs" : [] , "outputs" : [{ "name" :

"" , "type" :

"int256" , "internalType" :

"int256" }] , "stateMutability" :

"view" } , { "type" :

"function" , "name" :

"increment" , "inputs" : [] , "outputs" : [] , "stateMutability" :

"nonpayable" }];

// Define the address of the deployed contract const

contractAddress

= 0 X_CONTRACT_ADDRESS ;

// Create a new instance of the ethers.js Contract object const

contract

=

new

ethers .Contract (contractAddress , abi , provider);

// Call the contract's functions async

function

getCount () { const

count

=

await

contract .getCount (); console .log (count .toString ()); }

await

getCount (); Last updated onMay 24, 2024 CosmWasm (General) EVM (CLI)