2023/2024 S2

Ethereum II

Dr. Hui Gong (h.gong1@westminster.ac.uk)

UNIVERSITY OF WESTMINSTER

Smart Contracts and Solidity

What is a Smart Contract?

The term smart contract has been used over the years to describe a wide variety of different things. In the 1990s, cryptographer Nick Szabo coined the term and defined it as “a set of promises, specified in digital form, including protocols within which the parties perform on the other promises.” Since then, the concept of smart contracts has evolved, especially after the introduction of decentralized blockchain platforms with the invention of Bitcoin in 2009. In the context of Ethereum, the term is actually a bit of a misnomer, given that Ethereum smart contracts are neither smart nor legal contracts, but the term has stuck. Here, we use the term “smart contracts” to refer to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol - i.e., on the decentralized Ethereum world computer.

Computer programs
Smart contracts are simply computer programs. The word “contract” has no legal meaning in this context.

Immutable
Once deployed, the code of a smart contract cannot change. Unlike with traditional software, the only way to modify a smart contract is to deploy a new instance.

Deterministic
The outcome of the execution of a smart contract is the same for everyone who runs it, given the context of the transaction that initiated its execution and the state of the Ethereum blockchain at the moment of execution.

EVM context
Smart contracts operate with a very limited execution context. They can access their own state, the context of the transaction that called them, and some information about the most recent blocks.

Decentralized world computer
The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system as a whole operates as a single “world computer.”

Smart Contracts and Solidity

Life Cycle of a Smart Contract

Smart contracts are typically written in a high-level language, such as Solidity. But in order to run, they must be compiled to the low-level bytecode that runs in the EVM. Once compiled, they are deployed on the Ethereum platform using a special contract creation transaction, which is identified as such by being sent to the special contract creation address, namely 0x0. Each contract is identified by an Ethereum address, which is derived from the contract creation transaction as a function of the originating account and nonce. The Ethereum address of a contract can be used in a transaction as the recipient, sending funds to the contract or calling one of the contract's functions. **Note that, unlike with EOAs, there are no keys associated with an account created for a new smart contract.** As the contract creator, you don't get any special privileges at the protocol level(although you can explicitly code them into the smart contract).

You certainly don't receive the private key for the contract account, which in fact does not exist-we can say that smart contract accounts own themselves.

Importantly, contracts only run if they are called by a transaction. All smart contracts in Ethereum are executed, ultimately, because of a transaction initiated from an EOA. A contract can call another contract that can call another contract, and so on, but the first contract in such a chain of execution will always have been called by a transaction from an EOA. Contracts never run "on their own" or "in the background." Contracts effectively lie dormant until a transaction triggers execution, either directly or indirectly as part of a chain of contract calls. It is also worth noting that smart contracts are not executed "in parallel" in any sense - the Ethereum world computer can be considered to be a single-threaded machine.

Programming with Solidity – Data Types

Boolean (bool)
Boolean value, true or false, with logical operators ! (not), && (and), || (or), == (equal), and != (not equal).

Integer (int, uint)
Signed (int) and unsigned (uint) integers, declared in increments of 8 bits from int8 to uint256. Without a size suffix, 256-bit quantities are used, to match the word size of the EVM.

Address
A 20-byte Ethereum address. The address object has many helpful member functions, the main ones being balance (returns the account balance) and transfer (transfers ether to the account).

Byte array (fixed)
Fixed-size arrays of bytes, declared with bytes1 up to bytes32.

Byte array (dynamic)
Variable-sized arrays of bytes, declared with `bytes` or `string`.

Enum
User-defined type for enumerating discrete values: enum NAME {LABEL1, LABEL2, ...}.

Arrays
An array of any type, either fixed or dynamic: `uint32[][5]` is a fixed-size array of five dynamic arrays of unsigned integers.

Struct
User-defined data containers for grouping variables: struct `NAME {TYPE1 VARIABLE1; TYPE2 VARIABLE2; ...}`.

Mapping
Hash lookup tables for *key* ⇒ *value* pairs: `mapping(KEY_TYPE ⇒ VALUE_TYPE) NAME`.

Predefined Global Variables

Transaction/message call context
The `msg` object is the transaction call (EOA originated) or message call (contract originated) that launched this contract execution. It contains a number of useful attributes:

`msg.sender`
We've already used this one. It represents the address that initiated this contract call, not necessarily the originating EOA that sent the transaction. If our contract was called directly by an EOA transaction, then this is the address that signed the transaction, but otherwise it will be a contract address.

`msg.value`
The value of ether sent with this call (in wei).

`msg.data`
The data payload of this call into our contract.

`msg.sig`
The first four bytes of the data payload, which is the function selector.

Block context
The `block` object contains information about the current block:

`block.blockhash(blockNumber)`
The block hash of the specified block number, up to 256 blocks in the past.

`block.coinbase`
The address of the recipient of the current block's fees and block reward.

`block.difficulty`
The difficulty (proof of work) of the current block.

`block.number`
The current block number (blockchain height).

`block.timestamp`
The timestamp placed in the current block by the miner (number of seconds since the Unix epoch).

Smart Contracts and Solidity

Functions

Within a contract, we define functions that can be called by an EOA transaction or another contract. The syntax we use to declare a function in Solidity is as follows:

```
function FunctionName([parameters])
{public|private|internal|external}
[pure|constant|view|payable] [modifiers]
[returns (return types)]
```

Let's look at each of these components:

`FunctionName`
The name of the function, which is used to call the function in a transaction (from an EOA), from another contract, or even from within the same contract.

`Parameters`
Following the name, we specify the arguments that must be passed to the function, with their names and types.

The next set of keywords (`public, private, internal, external`) specify the function's *visibility*:

`Public`
Public is the default; such functions can be called by other contracts or EOA transactions, or from within the contract. In our Faucet example, both functions are defined as public.

`External`
External functions are like public functions, except they cannot be called from within the contract unless explicitly prefixed with the keyword this.

`Internal`
Internal functions are only accessible from within the contract—they cannot be called by another contract or EOA transaction. They can be called by derived contracts (those that inherit this one).

`Private`
Private functions are like internal functions but cannot be called by derived contracts.

Contract Constructor and selfdestruct

There is a special function that is only used once. When a contract is created, it also runs the constructor function if one exists, to initialize the state of the contract. The *constructor* is run in the same transaction as the contract creation.

To address the potential problems with constructor functions being based on having an identical name as the contract, Solidity v0.4.22 introduces a constructor keyword that operates like a constructor function but does not have a name. Renaming the contract does not affect the constructor at all. Also, it is easier to identify which function is the constructor. It looks like this:

```
pragma ^0.4.22
contract MEContract {
           constructor () {
                // This is the constructor
           }
         }
```

To summarize, a contract's life cycle starts with a creation transaction from an EOA or contract account. If there is a constructor, it is executed as part of contract creation, to initialize the state of the contract as it is being created, and is then discarded.

The other end of the contract's life cycle is *contract destruction*. Contracts are destroyed by a special EVM opcode called `SELFDESTRUCT`. It used to be called `SUICIDE`, but that name was deprecated due to the negative associations of the word. In Solidity, this opcode is exposed as a high-level built-in function called selfdestruct, which takes one argument: the address to receive any ether balance remaining in the contract account. It looks like this:

```
      selfdestruct(address recipient);
```

Note that you must explicitly add this command to your contract if you want it to be deletable—this is the only way a contract can be deleted, and it is not present by default.

References to read

[1] https://github.com/ethereum/solidity

[2] https://github.com/OpenZeppelin/openzeppelin-contracts

[3] https://www.consensys.net/

[4] https://openzeppelin.com/

[5] https://soliditylang.org/