Diving Into The Ethereum VM Part 5 — The Smart Contract Creation Process

zh

Follow

--

2

Listen

Share

In previous articles of this series, we've learned the basics of EVM assembly, as well as how ABI encoding allows the outside world to communicate with a contract. In this article, we'll see how a contract is created from nothing.

Previous articles of this series (in order).

- Introduction to the EVM assembly code.
- How fixed-length data types are represented.
- How dynamic data types are represented.
- How ABI Encodes External Method Calling.

The EVM bytecode we've seen so far is straightforward, just instructions that the EVM executes from top to bottom, no magic up its sleeve. The contract creation process is more fun, in that it blurs the barrier between code and data.

In learning how a contract is created, we'll see that sometimes data is code, and sometimes code is data.

Put on your favourite wizard hat

A Contract's Birth Certificate

Let's create a simple (and completely useless) contract:

Compile it:

And the bytecode is:

To create this contract, we'll need to create a transaction by making aneth_sendtransaction RPC call to an Ethereum node. You could use Remix or Metamask to do this.

Whatever deployment tool you use, the parameters for the RPC call would be something similar to:

There is no special RPC call or transaction type to create a contract. The same transaction mechanism is used for other purposes as well:

- Transferring Ether to an account or contract.
- Calling a contract's method with parameters.

Depending on what parameters you specified, the transaction is interpreted differently by Ethereum. To create a contract, the to

address should be null (or left out).

I've created the example contract with this transaction:

https://rinkeby.etherscan.io/tx/0x58f36e779950a23591aaad9e4c3c3ac105547f942f221471bf6ffce1d40f8401

Opening Etherscan, you should see that the input data for this transaction is the bytecode produced by the Solidity compiler:

When processing this transaction, the EVM would have executed the input data as code. And voila

, a contract was born.

What The Bytecode Is Doing

We can break the bytecode above into three separate chunks:

- Deploy code runs when the contract is being created.
- Contract code runs after the contract had been created, when its methods are called.
- (optional) Auxdata is the cryptographic fingerprint of the source code, used for verification. This is just data, and never executed by the EVM.

The deploy code has two main purposes:

1. Runs the constructor function, and sets up initial storage variables (like contract owner).
2. Calculates the contract code, and returns it to the EVM.

The deploy code generated by the Solidity Compiler loads the bytes 60606040525b600080fd00

from bytecode into memory, then returns it as the contract code. In this case, the "calculation" is merely reading a chunk of data into memory. In principle, we could programatically generate the contract code.

Exactly what the constructor does is up to the language, but any EVM language would have to return the contract code at the end.

Contract Creation

So what happens after the deploy code had run and returned the contract code. How does Ethereum create a contract from the returned contract code?

Let’s dig into the source code together to learn the details.

I’ve found that the Go-Ethereum implementation is the easiest reference to find the information I need. We get proper variable names, static type info, and symbol cross-references. Try beating that, Yellow Paper!

The relevant method is evm.Create, read it on Sourcegraph (there’s type info when you hover over a variable, pretty sweet). Let’s skim the code, omitting some error checking and fussy details. From top to bottom:

- Check whether caller has enough balance to make a transfer:
- Derive the new contract’s address from the caller’s address (passing in the creator account’s nonce):
- Create the new contract account using the derived contract address (changing the “world state” StateDB):
- Transfer the initial Ether endowment from caller to the new contract:
- Set input data as contract’s deploy code, then execute it with EVM. The ret

variable is the returned contract code:

- Check for error. Or if the contract code is too big, fail. Charge the user gas then set the contract code:

Code That Deploys Code

Let’s now dive into the nitty gritty assembly code to see how “deploy code” returns “contract code” when a contract is created. Again, we’ll analyze the example contract:

The bytecode for this contract broken into separate chunks:

The assembly for the deploy code is:

Tracing through the above assembly for returning the contract code:

dataSize(sub_0)

and dataOffset(sub_0)

are not real instructions. They are in fact PUSH instructions that put constants onto the stack. The two constants 0x1C

(28) and 0x36

(54) specifies a bytecode substring to return as contract code.

The deploy code assembly roughly corresponds to the following Python3 code:

The resulting memory content is:

Which corresponds to the assembly (plus auxdata):

Looking again on Etherscan, this is exactly what was deployed as the contract code:

Ethereum Account 0x2c7f561f1fc5c414c48d01e480fdaae2840b8aa2 Info

The Ethereum BlockChain Explorer, API and Analytics Platform

rinkeby.etherscan.io

CODECOPY

The deploy code uses codecopy

to copy from transaction's input data to memory.

The exact behaviour and arguments for the codecopy

instruction is less obvious than other simpler instructions. If I were to look it up in the Yellow Paper, I'd probably get more confused. Instead, let's refer to the go-ethereum source code to see what it's doing.

See CODECOPY:

No Greek letters!

The line evm.interpreter.intPool.put(memOffset, codeOffset, length)

recyles objects (big integers) for later uses. It is just an efficiency optimization.

Constructor Argument

Aside from returning the contract code, the other purpose of the deploy code is to run the constructor to set things up. If there are constructor arguments, the deploy code needs to somehow load the arguments data from somewhere.

The Solidity convention for passing constructor arguments is by appending the ABI encoded parameter values at the end of the bytecode when calling eth_sendtransaction

. The RPC call would pass in the bytecode and ABI encoded params together as input data, like this:

Let's look at an example contract with one constructor argument:

I've created this contract, passing in the value 66

. The transaction on Etherscan:

https://rinkeby.etherscan.io/tx/0x2f409d2e186883bd3319a8291a345ddbc1c0090f0d2e182a32c9e54b5e3fdbd8

The input data is:

We can see the constructor argument at the very end, which is the number 66, but ABI encoded as a 32 bytes number:

To process the arguments in the constructor, the deploy code copies the ABI parameters from the end of the calldata

into memory, then from memory onto the stack.

A Contract That Creates Contracts

The FooFactory

contract can create new instances of Foo

by calling makeNewFoo

:

The full assembly for this contract is In This Gist. The structure of the compiler output is more complicated, because there are two sets of “install time” and “run time” bytecode. It’s organized like this:

The FooFactoryContractCode

essentially copies the bytecode for Foo

in tag_8

then jump back to tag_7

to execute the create

instruction.

The create

instruction is like the eth_sendtransaction

RPC call. It provides a way to create new contracts from within the EVM.

See opCreate for the go-ethereum source code. This instruction calls evm.Create

to create a contract:

We’ve seen evm.Create

earlier, but this time the caller is an Smart Contract, not a human.

AUXDATA

If you absolutely must know all about what auxdata is, readContract Metadata. The gist of it is that auxdata

is a hash that you can use to fetch metadata about the deployed contract.

The format of auxdata is:

Deconstructing the auxdata bytesequence we’ve seen previously:

There you go, one more Ethereum trivia for the party night.

Conclusion

The way contracts are created is similar to how a self-extracting software installer works. When the installer runs, it configures the system environment, then extracts the target program onto the system by reading from its program bundle.

- There is enforced separation between “install time” and “run time”. No way to run the constructor twice.
- Smart contracts can use the same process to create other smart contracts.
- It is easy for a non-Solidity languages to implement.

At first, I found it confusing that different parts of the “smart contract installer” is packed together in the transaction as a data

byte string:

How data

should be encoded is not obvious from reading the documentation for eth_sendtransaction

. I couldn't figure out how constructor arguments are passed into a transaction until a friend told me that they are ABI-encoded then appended to the end of the bytecode.

An alternative design that would’ve made it clearer is perhaps to send these parts as separate properties in a transaction:

On more thoughts, though, I think that it’s actually very powerful that the Transaction object is so simple. To a Transaction, data

is just a byte string, and it doesn't dictate a language model for how the data should be interpreted. By keeping the Transaction object simple, language implementers have a blank canvas for design and experiments.

Indeed, data

could even be interpreted by a different virtual machine in the future.

If you like furry animals, you should follow me on Twitter

[@hayeah.

](https://twitter.com/hayeah)