Bytecode and transactions

WHAT

So we have seen how Geth works and we are on our way to code smart contracts in Solidity. But before we abstract away the complexity, let's stay close to the metal and see what the bytecode looks like and how smart contract transactions are assembled.

Opcodes

The Ethereum Virtual Machine, EVM, is a multipurpose machine that understands and executes a machine language. This machine language is the bytecode. A complete bytecode may look like:

```
0x6160616040523460005760b6060806100166000396000f360606040526000357c0100000000000(
```

It is made of different opcode values and their arguments. You can see a handy list of Ethereum opcodes here. So to take the above example, we can disassemble it thus:

```
PC | Command   | Opcode | Operation | Argument
0  | 0x616061  | 0x61   | PUSH2     | 0x6061
3  | 0x6040    | 0x60   | PUSH1     | 0x40
5  | 0x52      | 0x52   | MSTORE    | None
6  | 0x34      | 0x34   | CALLVALUE | None
And so on...
```

Where:

- *PC* is the Program Counter, i.e. what offset the execution is, or would be, currently at.
- the *command* is a segment of bytecode composed of the *opcode* on the left followed by its arguments, if any. The opcode dictates how many arguments there should be.
- the *operation* is the human-understandable name of the opcode.

Stack

`PUSH1`? Yes, just like the JVM, the EVM is a stack-based machine, with operations, or operators, pushing operands to, or using operands from, the stack, so the opcodes seen above result in the following sequence of stack:

```
Empty -> 0: 0x6061 -> | 0: 0x40   | -> Empty -> 0: 0xde0b6b3a7640000
                      | 1: 0x6061 |
```

After the 3rd operation, `MSTORE`, the stack is empty again because `0x6061` went into memory at offset `0x40`.

`CALLVALUE` is pushing the amount of weis that were sent along with the transaction. Here 1 Ether, i.e. 10^18 weis or `0xde0b6b3a7640000` in hexadecimal. You will note that, although the bytecode can call up attributes of the transaction, it does not contain elements of the transaction in its body. In fact, the bytecode is either sent along, or called up by, a transaction.

If you have experience with using a reverse polish notation (RPN) calculator, which equates a space between operands with a `PUSH`, the above stack operations should look familiar. Here is an example of an RPN calculation: `3 5 + 2 *` means `(3+5)*2`.

Also note that:

- Each spot on the stack is 32-bytes long, hence the opcodes `PUSH1` to `PUSH32`. Why not keep a single `PUSH32` opcode? Because, why spend the gas for 32 bytes when all we need is a single byte?
- The stack's top 16 elements are directly accessible with a `DUPx`, i.e. duplicate, or `SWAPx`opcodes, with `1 <= x <= 16`. Remember this barrier if and when you hit the maximum number of local variables in a Solidity function, via a somewhat cryptic message such as "Stack too deep".
- The stack can be deeper than 16, although, to access the elements at position 17 and beyond, you have to reduce the stack size by using or `POP`ping elements from the top.

Memory

At each transaction execution, the memory starts empty and offers 2^256 slots of 1 byte each. 32 bytes are referred to as "word" in the documentation. All slots are accessible. You write to them from the stack with `MSTORE`, and copy their value to the stack with `MLOAD`. Storing and loading in memory are more gas-expensive actions than pushing to and popping from the stack. Nonetheless, it becomes handy when you handle more than 16 variables.

Also, since the values on the stack are all 32 bytes long, when you `MSTORE` such a value at offset `n`, it will overwite 32 bytes in the memory, from offset `n` to offset `n + 31`.

At the end of execution, the memory is cleared.

Storage

Storage is a space that contains 2^256 slots of 32 bytes. The difference with the memory is that its values are kept between executions and the whole space is attached to an address. `SSTORE`and `SLOAD` are also much more gas-expensive than their memory equivalents. Keep this in mind when you are tempted to keep every minute information in your contract's storage.

The storage is unique per deployed contract instance. In particular:

- it is not a global space for all contracts
- it is not per transaction sender, as a HTTP session would be.

Call stack

A contract can call another contract, or, more generally, any address. We call this an *internal*transaction. You can see such a thing happening when our Ropsten faucet sends you some Ether.

Whenever an internal transaction is sent, the current stack and memory is pushed to the call stack, to be popped and restored when the other contract returns the execution.

This call stack, not to be mixed with *the* stack, is only 1,024 deep. When it reaches its limit, the EVM does not honour the calling contract's request, and does not call the other contract. This can become an attack vector if the calling contract does not check that its request has correctly executed.

On the other hand, after changes in the gas cost of some opcodes, it appears this attack vector is not possible as it would need too large a block gas limit. Unless the block gas limit increases, so beware.

EXERCISE

First bytecode

Let's build a contrived example of code, which, after a quick calculation, stores the result at slot 0:

```
PUSH1 0x03
PUSH1 0x05
ADD         // Do 3 + 5 -> 8
PUSH1 0x02
MUL         // Do 8 * 2 -> 16
```

```
PUSH1 0x00
SSTORE     // 16 goes to storage, at slot 0
```

Let's assemble this by hand into the following bytecode:

```
0x600360005501600202600055
```

Phew! Now, in your "net42" Geth, you are going to *send* this bytecode from `accounts[0]` to **none**. This is how you instruct the EVM to understand this transaction as code, or, in simpler terms, as a new contract account. In Geth:

```
> var bytecode = "0x600360005501600202600055"
undefined

> eth.sendTransaction({ from: eth.accounts[0], data: bytecode })
INFO [05-16|17:24:37] Submitted contract creation fullhash=0xbbc4356b4fb504981cc
"0xbbc4356b4fb504981cd44c3c7a2be266def9f8b14f871d1b410ba960ca61a5f3"
```

Now mine and wait for the transaction to be mined, then, take the `contract=` address, as returned next to the transaction hash, here `"0x930af5d19a0fb7563fb873da2e0a1adc9c7b87cf"`:

```
> var created = "0x930af5d19a0fb7563fb873da2e0a1adc9c7b87cf"
undefined

> eth.getStorageAt(created, 0) // Slot 0
"0x0000000000000000000000000000000000000000000000000000000000000010"
                                                          // Here ^^
```

And there you have it, our storage has changed to `0x10`, i.e. `16` in hexadecimal.

Bytecode to our contract

What would happen if we sent the following to our new contract?

```
PUSH1 0x11 // -> 17
PUSH1 0x00
SSTORE     // Send 17 to storage at offset 0, overwriting 16, presumably
```

Assemble by hand again and let's try in Geth:

```
> var bytecode2 = "0x6011600055"
undefined
```

```
// Notice how we target to: created
> eth.sendTransaction({ from: eth.accounts[0], to: created, data: bytecode2 })
INFO [08-03|12:26:46] Submitted transaction                    fullhash= 0x4b7e8
"0x4b7e822e63d1a2388430c46a09b75257fe23ba2cb5d1c85bc91a244d7f76f69a"
```

Wait for it to mine, then:

```
> eth.getStorageAt(created, 0)
"0x0000000000000000000000000000000000000000000000000000000000000010"
                                                    // Still here ^^
```

It has not changed. The reason why is that:

- when we sent `to:` none, the EVM understood the transaction as code, or as a contract creation, or deployment, and as such treated `data:` as bytecode to execute.
- when we sent `to: created`, the EVM understood the transaction as a contract invokation, and as such treated `data:` as data made available to the contract's code.

What is our contract code by the way? In Geth:

```
> eth.getCode(created)
"0x"
```

Oh! So although we executed some code in the first transaction, this code was discarded right after its execution. Now, there is no code at our supposed contract's address, it has become useless.

Make a contract

There was no contract code at our address because the EVM expects the contract code, fasten your seat belts, to be the bytecode remaining in memory, according to parameters, when the bytecode we send initially finishes execution with `RETURN`. Our in fewer words, we need to pack the final code for the EVM to unpack it.

So, let's prepare our contract code. The final (unpacked) contract code, not the (packed) code we send along the initial transaction. For instance, let's say we want it to take data from every invokation transaction and save it at offset `0`:

```
PUSH1 0x00
CALLDATALOAD // Push 32 bytes of the transaction data from 0x00 onwards on the s
PUSH1 0x00
SSTORE       // Save the transaction data at offset 0
```

Which assembles into:

```
0x600035600055
```

Short and sweet, 6 bytes. But that's the final (unpacked) contract code; the code you want to get with `eth.getCode(address)`. We need to (pack and) write some more code to pass in the deployment transaction and have the EVM pick up the final code:

```
00: PUSH1 0x06 // The number of bytes of the final contract code
02: PUSH1 0x0c // The offset at which the contract bytecode is sent in the deplo
// see the end of this assembly for the 0x0c value
04: PUSH1 0x00 // The memory slot at which we send the code
06: CODECOPY   // Copy a subset of this very chunk of code using top 3 stack va
// so, into memory at slot 0 but only 0x06 bytes starting at offset 0x0c
07: PUSH1 0x06 // The length of the code to pick
09: PUSH1 0x00 // The memory slot at which the code is now found
0b: RETURN     // Tell EVM to pick the code from the memory.
// Execution stops here, whatever comes after is not executed. That's why we...
0c: 600035600055   // Just pad with our contract code at offset 0x0c
```

Which assembles into:

```
0x6006600c60003960066000f3600035600055
//                       ^ The final contract code starts here
```

Let's try that in Geth:

```
> var bytecode3 = "0x6006600c60003960066000f3600035600055"
undefined

> eth.sendTransaction({ from: eth.accounts[0], data: bytecode3 })
INFO [08-03|12:33:40] Submitted contract creation              fullhash= 0xfbbf4
"0xfbbf4d4d2a07834f5bb54cf074466095f6c87517adf057bc593e0268dfadb56d"

// Again, pick *your* contract value above
> var created3 = "0xe905c16309a422740e439a48479049a7c45de8ff"
undefined
```

Mine, then check:

```
> eth.getCode(created3)
"0x600035600055"
```

Yes! We have our contract with the desired bytecode.

Invoke our contract

Now that we have our contract, that supposedly stores whatever we send it, let's try and invoke it:

```
> eth.getStorageAt(created3, 0)
"0x0000000000000000000000000000000000000000000000000000000000000000"

> eth.sendTransaction({ from: eth.accounts[0], to: created3, data: "0x67" })
INFO [08-03|12:37:42] Submitted transaction                    fullhash= 0xe98f1
"0xe98f11c877c7d3c7ceaf5ff32f99818d5c35bc5f2297cb53d5f64053cc83cd1e"

// Mine, then
> eth.getStorageAt(created3, 0)
"0x6700000000000000000000000000000000000000000000000000000000000000"
// ^^ Here
```

Here we have it, our `0x67` in the transaction data found its way into storage, although in the high order bits. That's because, when the EVM picked 32 bytes from the call data, it found `0x67`, and kept taking 31 more bytes out of thin air, hence the `0x00`s. You will recall that, if there is no data where you ask, there always are zeroes.

Let's try again with 33 bytes of data:

```
// Note the deliberately 33-bytes long data
> eth.sendTransaction({ from: eth.accounts[0], to: created3, data: "0x000102030
INFO [08-03|12:40:28] Submitted transaction                    fullhash= 0x9308
"0x93085112ae6a3a6673bd1a29a9e87c58b6b2c0c3e9a08a1eee6701893ce4306c"

// Mine, then
> eth.getStorageAt(created3, 0)
"0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f"
// Only the first 32 bytes were saved, 0x20 were chopped off

> eth.getStorageAt(created3, 1)
"0x0000000000000000000000000000000000000000000000000000000000000000"
// The lost 0x20 did not overflow to the next slot.
```

To conclude

We have seen a tedious way to:

1. send code to be executed once and right away, and then be discarded
2. send code to be stored as a smart contract
3. invoke a smart contract

Solidity will abstract a lot of this tediousness, respectively to:

1. offer a constructor function, executed once and right away, and then be discarded
2. compile and package our smart contract to be picked by the EVM
3. package function calls, or invokations, appropriately into `data:`