【翻译】 深入理解以太坊虚拟机 - EVM汇编代码简介

2018-06-12

区块链 Qtum 翻译

原文：https://blog.qtum.org/diving-into-the-ethereum-vm-6e8d5d2f3c30
译者：中山大学数学学院（珠海）林学渊
大二时给量子做的翻译，转载注明出处，谢谢

EVM汇编代码简介

Solidity 提供了很多高级语言抽象，但这些功能很难让我理解程序运行时到底发生了什么。阅读 Solidity 的文档仍然使我对一些基础的东西感到疑惑。
string, bytes32, byte[], bytes 有什么区别?

什么时候应该用哪个?
把 string 转为 bytes 发生了什么？转为 byte[] 呢?
这些需要多少 gas ?

mapping 在以太坊虚拟机里是怎么存的?

为什么不能把 mapping 删了?
能构建 maping 到 maping 的数据结构吗?（当然可以，但这是怎么实现的？）
为什么有存储 mapping ，但是没有内存 mapping ?

编译后的合约在以太坊虚拟机里长什么样?

合约如何创建?
构造方法是什么？真的吗?
回退函数是什么?

我想，学习一门在以太坊虚拟机(EVM)上运行的高级语言如 Solidity 会是一个好的自我投资。有以下原因。

1. Solidity 不是最后一门语言。更好的 EVM 语言正在到来。(漂亮，对不对？)
2. EVM是个数据库引擎。理解用任意一种 EVM 语言写的智能合约前，必须理解数据是如何被组织、存储和操控的。
3. 了解如何成为一个贡献者。以太坊工具链刚刚起步，深入理解EVM会帮助你给你自己或其他人造出惊艳的工具。
4. 智力挑战。EVM使得你能在密码学、数据结构和程序语言设计的交汇处获得最佳实践。

在本系列文章中，我会解构一些简单的 Solidity 智能合约，以便理解它们作为 EVM 字节码时如何工作。
我希望学习和写作的要点：

EVM 字节码的基础
不同数据类型(mapping, array)的表现形式
合约创建时发生了什么
一个方法调用时发生了什么
ABI 桥如何区别了 EVM 语言

我的终极目标是能够完全理解一个编译后的 Solidity 合约。先从一些基础的 EVM 字节码开始阅读吧！
一个有用的引用：EVM 指令集

一个简单的智能合约

我们的第一个合约有构造函数和一个常量：

```
// c1.sol
pragma solidity ^0.4.11;
contract C {
    uint256 a;
    function C() {
      a = 1;
    }
}
```

用 `solc` 命令编译：

```
$ solc --bin --asm c1.sol
======= c1.sol:C =======
EVM assembly:
    /* "c1.sol":26:94  contract C {... */
  mstore(0x40, 0x60)
    /* "c1.sol":59:92  function C() {... */
  jumpi(tag_1, iszero(callvalue))
  0x0
  dup1
  revert
tag_1:
tag_2:
    /* "c1.sol":84:85  1 */
  0x1
    /* "c1.sol":80:81  a */
  0x0
    /* "c1.sol":80:85  a = 1 */
  dup2
  swap1
  sstore
  pop
    /* "c1.sol":59:92  function C() {... */
tag_3:
    /* "c1.sol":26:94  contract C {... */
tag_4:
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x0
  codecopy
  0x0
  return
stop
sub_0: assembly {
        /* "c1.sol":26:94  contract C {... */
      mstore(0x40, 0x60)
```

```
    tag_1:
      0x0
      dup1
      revert
auxdata: 0xa165627a7a72305820af3193f6fd31031a0e0d2de1ad2c27352b1ce081b4f3c92b5650ca4dd542bb770029
}
Binary:
6060604052341560 0e57600080fd5b5b60016000819055505b5b603680602660003960 00f30060606040525b600080fd00a165627a7a7230582
```

数字 `6060604052...` 是 EVM 真正运行的字节码。

蹒跚学步

一半的汇编是模板，以至于在大多数 Solidity 程序中都一样。我们等下再来看这些。现在，我们来实验我们合约独特的一部分，存储变量的声明：

```
1
```

```
a = 1
```

这个声明的字节码表示是 `6001600081905550` 。根据指令换行：

```
1
2
3
4
5
6
```

```
60 01
60 00
81
90
55
50
```

EVM 底层循环是从上到下运行每一条指令。
我们注释一下汇编代码(以 `tag_2` 开头)以便阅读：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
tag_2:
  // 60 01
  0x1
  // 60 00
  0x0
  // 81
  dup2
  // 90
  swap1
  // 55
  sstore
  // 50
  pop
```

注意汇编中的 `0x1` 实际上是 `push(0x1)` 的缩写。这条指令表示吧数字 `1` 入栈。
如果只盯着这个看，很难捕获到发生了什么。不要担心，模仿 EVM 一行一行地走，很简单的。

模仿 EVM

EVM 是堆栈机器。指令可以使用栈中的值作为参数，也可以把某一些值入栈作为结果。举个例子， `add` 指令。
假设栈中有 2 个值：

```
1
```

```
[1, 2]
```

当 EVM 看到 add 时，它把栈顶的 2 项出栈相加，然后把结果入栈回去，操作后：

```
1
```

```
[3]
```

以后我们仍然用 [ ] 这个符号来表示栈：

```
// 空栈
stack: []
// 有3个元素的栈. 栈顶元素是 3. 栈底元素是 1.
stack: [3 2 1]
```

用 { } 来表示合约存储:

```
// 空存储
store: {}
// 值 0x1 存储在地址 0x0.
store: { 0x0 => 0x1 }
```

现在我们来看一些实际的字节码。我们将模仿EVM运行字节序列 `6001600081905550` ，同时写出每一条指令运行后的机器状态：

```
// 60 01: 将1入栈
0x1
  stack: [0x1]
// 60 00: 将0入栈
0x0
  stack: [0x0 0x1]
// 81: 复制栈中的第二个元素，入栈
dup2
  stack: [0x1 0x0 0x1]
// 90: 交换栈顶2个元素
swap1
  stack: [0x0 0x1 0x1]
// 55: 把值 0x1 存储到地址 0x0
// 这条指令使用了栈顶的两个元素
sstore
  stack: [0x1]
  store: { 0x0 => 0x1 }
// 50: 出栈，即丢掉栈顶一个元素
pop
  stack: []
  store: { 0x0 => 0x1 }
```

运行完了。栈空了，同时有一个元素存储到了存储器里。
值得注意的是 Solidity 决定把状态变量 `uint256 a` 存储到地址 `0x0` 。很可能其他语言会把状态变量存到其他地方。
写出伪代码， EVM 运行 `6001600081905550` 就像是这样：

```
// a = 1
sstore(0x0, 0x1)
```

看仔细一点，会发现 dup2, swap1, pop 是多余的。汇编代码可以更简单：

```
0x1
0x0
sstore
```

你可以试着模拟运行上面的3条指令，肯定会惊喜地发现它们结束时的机器状态是一样的：

```
stack: []
store: { 0x0 => 0x1 }
```

2 个存储变量

添加另一个相同数据类型的存储变量：

```
// c2.sol
pragma solidity ^0.4.11;
contract C {
    uint256 a;
    uint256 b;
    function C() {
      a = 1;
      b = 2;
    }
}
```

编译，注意 `tag_2`：

```
$ solc --bin --asm c2.sol
// ... more stuff omitted
tag_2:
    /* "c2.sol":99:100  1 */
  0x1
    /* "c2.sol":95:96  a */
  0x0
    /* "c2.sol":95:100  a = 1 */
  dup2
  swap1
  sstore
  pop
    /* "c2.sol":112:113  2 */
  0x2
    /* "c2.sol":108:109  b */
  0x1
    /* "c2.sol":108:113  b = 2 */
  dup2
  swap1
  sstore
  pop
```

汇编伪代码：

```
// a = 1
sstore(0x0, 0x1)
// b = 2
sstore(0x1, 0x2)
```

现在我们知道这两个存储变量是依次存储的，`a` 存储在地址 `0x0`，`b` 存储在地址 `0x1`。

打包存储

每个存储单元能存 32 字节。如果全部使用 32 字节的话，如果一个变量只要 16 字节，那就很浪费了。 Solidity 通过把 2 个短的数据类型打包成 1 个来提高存储效率。
把 `a` 和 `b` 改成每个 16 字节：

```
pragma solidity ^0.4.11;
contract C {
    uint128 a;
    uint128 b;
    function C() {
      a = 1;
      b = 2;
    }
}
```

编译合约：

```
$ solc --bin --asm c3.sol
```

生成的汇编代码更复杂了：

```
tag_2:
  // a = 1
  0x1
  0x0
  dup1
  0x100
  exp
  dup2
  sload
  dup2
  0xffffffffffffffffffffffffffffffff
  mul
  not
  and
  swap1
  dup4
  0xffffffffffffffffffffffffffffffff
  and
  mul
  or
  swap1
  sstore
  pop
  // b = 2
  0x2
  0x0
  0x10
  0x100
  exp
  dup2
  sload
  dup2
  0xffffffffffffffffffffffffffffffff
  mul
  not
  and
  swap1
  dup4
  0xffffffffffffffffffffffffffffffff
  and
  mul
  or
  swap1
  sstore
  pop
```

上面的汇编代码把 2 个变量打包到1个存储地址(`0x0`)，像这样：

```
[         b         ][         a         ]
[16 bytes / 128 bits][16 bytes / 128 bits]
```

打包的原因是目前最贵的操作就是存储空间的使用：

`sstore` 花费 20,000 gas 来第一次写入一个新地址
`sstore` 花费 5,000 gas 来随后写入一个已存在的地址
`sload` 花费 500 gas
大多数指令只花费 3~10 gas

通过使用相同的地址， Solidity 为第二个变量存储只支付 5,000 而不是 20,000，省了 15,000 gas。

更多优化

不分别同 2 个 `sstore` 指令来保存 `a` 和 `b`， 而把2个128比特的数字打包到内存里再使用1个 `sstore`， 从而节省 5,000 gas。
你可以通过 `optimize` 标志来让Solidity做这个操作：

```
$ solc --bin --asm --optimize c3.sol
```

这个方式生成的汇编代码只使用1个 `sload` 和1个 `sstore`：

```
tag_2:
    /* "c3.sol":95:96  a */
  0x0
    /* "c3.sol":95:100  a = 1 */
  dup1
  sload
    /* "c3.sol":108:113  b = 2 */
  0x20000000000000000000000000000000
  not(sub(exp(0x2, 0x80), 0x1))
    /* "c3.sol":95:100  a = 1 */
  swap1
  swap2
  and
    /* "c3.sol":99:100  1 */
  0x1
    /* "c3.sol":95:100  a = 1 */
  or
  sub(exp(0x2, 0x80), 0x1)
    /* "c3.sol":108:113  b = 2 */
  and
  or
  swap1
  sstore
```

字节码是

```
600080547002000000000000000000000000000000000060016080600020a0319909116600117600160806000 20a0316179055
```

格式化字节码成一行一条指令的形式：

```
// push 0x0
60 00
// dup1
80
// sload
54
// push17 作为 32 字节的数字，把接下来的 17 字节入栈
70 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
/* not(sub(exp(0x2, 0x80), 0x1)) */
// push 0x1
60 01
```

```
// push 0x80 (32)
60 80
// push 0x80 (2)
60 02
// exp
0a
// sub
03
// not
19
// swap1
90
// swap2
91
// and
16
// push 0x1
60 01
// or
17
/* sub(exp(0x2, 0x80), 0x1) */
// push 0x1
60 01
// push 0x80
60 80
// push 0x02
60 02
// exp
0a
// sub
03
// and
16
// or
17
// swap1
90
// sstore
55
```

在汇编代码里有4个魔法变量：

0x1 (16 字节), 使用低 16 位字节

```
// 字节码表示 0x01
16:32 0x00000000000000000000000000000000
00:16 0x00000000000000000000000000000001
```

0x2 (16 字节), 使用高 16 位字节

```
//字节码表示 0x20000000000000000000000000000000
16:32 0x00000000000000000000000000000002
00:16 0x00000000000000000000000000000000
```

`not(sub(exp(0x2, 0x80), 0x1))`

```
// 高 16 字节的二进制掩码
16:32 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
00:16 0x00000000000000000000000000000000
```

`sub(exp(0x2, 0x80), 0x1)`

```
// 低 16 字节的二进制掩码
16:32 0x00000000000000000000000000000000
00:16 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

代码对这些值做了位交换以获得需要的结果：

```
16:32 0x00000000000000000000000000000002
00:16 0x00000000000000000000000000000001
```

最后，这个 32 字节的值存储在地址 0x0。

gas 的使用

> 6000805470020000000000000000000000000000000000600160806002 0a03199091166001176001608060020a0316179055

注意 `0x2000000000000000000000000000000000` 嵌在字节码里了。但编译器也可能选择用指令 `exp(0x2, 0x81)` 计算值，这将生成更短的字节码序列。
结果好像是 `0x2000000000000000000000000000000000` 比 `exp(0x2, 0x81)` 更便宜。我们看一下分别需要花费的 gas：

4 gas 花在一笔交易中的每一个为 0 的数据或代码
68 gas 花在一笔交易中的每一个非 0 的数据或代码

比较一下总的gas花费：

字节码 `0x2000000000000000000000000000000000`. 它有很多 0 ，更便宜
(1 *68) + (16* 4) = 196.
字节码 `608160020a`. 更短，但没有0.
5 * 68 = 340.

更长但有更多 0 的序列实际上更便宜！

总结

EVM 编译器实际上没有优化字节码大小或速度抑或内存效率。取而代之的是，它优化了 gas 的使用，这是一个间接的层面，可以激励以太坊区块链进行高效计算。
我们已经看到了 EVM 一些诡异的方面：

EVM 是 256 比特机器。以 32 字节为块来操作数据最自然。
持久化存储很贵。
Solidity 编译器为了最小化 gas 的使用采取了有趣的做法。

gas 成本的设定是任意的，以后可能会变。随着成本的变化，编译器会做出不同的选择。

在本系列文章中，关于 EVM 我会写：

EVM 汇编代码的介绍
定长数据类型如何表示
动态数据类型如何表示
ABI编码的外部函数如何调用
一个新合约创建时发生了什么

Prev: 【翻译】 深入理解以太坊虚拟机 - 如何表示固定长度的数据类型
Next: 【翻译】 QtumJS 文档

Leave a comment

Styling with Markdown is supported

Comment

Powered by Gitment

Social

| |

Links

raytaylorism主题作者 Github地址

Blog powered by Hexo | Theme raytaylorism