> 死磕以太坊源码分析之EVM固定长度数据类型表示
>
> 配合以下代码进行阅读：https://github.com/blockchainGuide/
>
> 写文不易，给个小关注，有什么问题可以指出，便于大家交流学习。
>
> 翻译自 https://medium.com/@hayeah/diving-into-the-ethereum-vm-part-2-storage-layout-bc5349cb11b7

我们先看一个简单的 `Solidity` 合约的汇编代码：

```
contract C {
    uint256 a;
    function C() {
      a = 1;
    }
}
```

该合约归结于 `sstore` 指令的调用：

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

- EVM将 `0x1` 数值存储在 `0x0` 的位置上
- 每个存储槽可以存储正好32字节(或256位)

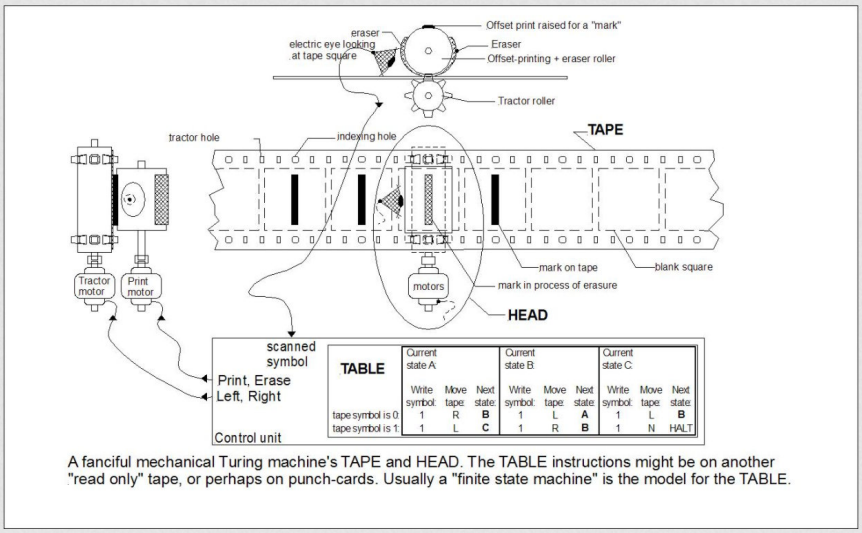
在本文中我们将会开始研究 `Solidity` 如何使用 `32` 字节的块来表示更加复杂的数据类型如结构体和数组。我们也将会看到存储是如何被优化的，以及优化是如何失败的。

在典型编程语言中理解数据类型在底层是如何表示的没有太大的作用。但是在 `Solidity` (或其他的EVM语言)中，这个知识点是非常重要的，因为存储的访问是非常昂贵的：

- `sstore` 指令成本是20000 gas，或比基本的算术指令要贵~5000x
- `sload` 指令成本是 200 gas，或比基本的算术指令要贵~100x

这里说的成本，就是真正的金钱，而不仅仅是毫秒级别的性能。运行和使用合约的成本基本上是由 `sstore` 指令和 `sload` 指令来主导的！

Parsecs磁带上的Parsecs



A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

构建一个通用计算机器需要两个基本要素：

- 一种循环的方式，无论是跳转还是递归
- 无限量的内存

`EVM` 的汇编代码有跳转， `EVM` 的存储器提供无限的内存。这对于一切就已经足够了，包括模拟一个运行以太坊的世界，这个世界本身就是一个模拟运行以太坊的世界.....

EVM的存储器对于合约来说就像一个无限的自动收报机磁带，磁带上的每个槽都能存储32个字节，就像这样：

```
[32 bytes][32 bytes][32 bytes]...
```

我们将会看到数据是如何在无限的磁带中生存的。

> 磁带的长度是2^{256}，或者每个合约~10^{77}存储槽。可观测的宇宙粒子数是10^{80}。大概1000个合约就可以容纳所有的质子、中子和电子。不要相信营销炒作，因为它比无穷大要短的多。

空磁带

存储器初始的时候是空白的，默认是0。拥有无限的磁带不需要任何的成本。

以一个简单的合约来演示一下0值的行为：

```
pragma solidity ^0.4.11;
contract C {
    uint256 a;
    uint256 b;
    uint256 c;
    uint256 d;
    uint256 e;
    uint256 f;
    function C() {
      f = 0xc0fefe;
    }
}
```

存储器中的布局很简单。

- 变量 `a` 在 `0x0` 的位置上
- 变量 `b` 在 `0x1` 的位置上
- 以此类推.....

关键问题是：如果我们只使用 `f` ，我们需要为 `a` ， `b` ， `c` ， `d` ， `e` 支付多少成本?

编译一下再看：

```
$ solc --bin --asm --optimize c-many-variables.sol
```

汇编代码：

```
// sstore(0x5, 0xc0fefe)
tag_2:
  0xc0fefe
  0x5
  sstore
```

所以一个存储变量的声明不需要任何成本，因为没有初始化的必要。Solidity为存储变量保留了位置，但是只有当你存储数据进去的时候才需要进行付费。

这样的话，我们只需要为存储 `0x5` 进行付费。

如果我们手动编写汇编代码的话，我们可以选择任意的存储位置，而用不着"扩展"存储器：

```
// 编写一个任意的存储位置
sstore(0xc0fefe, 0x42)
```

读取零

你不仅可以写在存储器的任意位置，你还可以立刻读取任意的位置。从一个未初始化的位置读取只会返回 `0x0` 。

让我们看看一个合约从一个未初始化的位置 `a` 读取数据：

```
pragma solidity ^0.4.11;
contract C {
    uint256 a;
    function C() {
      a = a + 1;
    }
}
```

编译：

```
$ solc --bin --asm --optimize c-zero-value.sol
```

汇编代码：

```
tag_2:
  // sload(0x0) returning 0x0
  0x0
  dup1
  sload
  // a + 1; where a == 0
  0x1
  add
  // sstore(0x0, a + 1)
  swap1
  sstore
```

注意生成从一个未初始化的位置 `sload` 的代码是无效的。

然而，我们可以比Solidity编译器聪明。既然我们知道 `tag_2` 是构造器，而且 `a` 从未被写入过数据，那么我们可以用 `0x0` 替换掉 `sload` ，以此节省5000 gas。

结构体的表示

来看一下我们的第一个复杂数据类型，一个拥有 6 个域的结构体：

```
pragma solidity ^0.4.11;
contract C {
    struct Tuple {
      uint256 a;
      uint256 b;
      uint256 c;
      uint256 d;
      uint256 e;
      uint256 f;
    }
    Tuple t;
    function C() {
      t.f = 0xC0FEFE;
    }
}
```

存储器中的布局和状态变量是一样的：

- `t.a` 域在 `0x0` 的位置上
- `t.b` 域在 `0x1` 的位置上
- 以此类推.....

就像之前一样，我们可以直接写入 `t.f` 而不用为初始化付费。

编译一下：

```
$ solc --bin --asm --optimize c-struct-fields.sol
```

然后我们看见一模一样的汇编代码：

```
tag_2:
  0xc0fefe
  0x5
  sstore
```

固定长度数组

让我们来声明一个定长数组：

```
pragma solidity ^0.4.11;
contract C {
    uint256[6] numbers;
    function C() {
      numbers[5] = 0xC0FEFE;
    }
}
```

因为编译器知道这里到底有几个 `uint256` (32字节)类型的数值，所以它可以很容易让数组里面的元素依次存储起来，就像它存储变量和结构体一样。

在这个合约中，我们再次存储到 `0x5` 的位置上。

编译：

```
$ solc --bin --asm --optimize c-static-array.sol
```

汇编代码：

```
tag_2:
  0xc0fefe
  0x0
  0x5
tag_4:
  add
  0x0
tag_5:
  pop
  sstore
```

这个稍微长一点，但是如果你仔细一点，你会看见它们其实是一样的。我们手动的来优化一下：

```
tag_2:
  0xc0fefe
  // 0+5. 替换为0x5
  0x0
  0x5
  add
  // 压入栈中然后立刻出栈。没有作用，只是移除
  0x0
  pop
  sstore
```

移除掉标记和伪指令之后，我们再次得到相同的字节码序列：

```
tag_2:
  0xc0fefe
  0x5
  sstore
```

数组边界检查

我们看到了定长数组、结构体和状态变量在存储器中的布局是一样的，但是产生的汇编代码是不同的。这是因为 `Solidity` 为数组的访问产生了边界检查代码。

让我们再次编译数组合约，这次去掉优化的选项：

```
$ solc --bin --asm c-static-array.sol
```

汇编代码在下面已经注释了，并且打印出每条指令的机器状态：

```
tag_2:
  0xc0fefe
    [0xc0fefe]
  0x5
    [0x5 0xc0fefe]
  dup1
  /* 数组边界检查代码 */
  // 5 < 6
  0x6
    [0x6 0x5 0xc0fefe]
  dup2
    [0x5 0x6 0x5 0xc0fefe]
  lt
    [0x1 0x5 0xc0fefe]
  // bound_check_ok = 1 (TRUE)
  // if(bound_check_ok) { goto tag5 } else { invalid }
  tag_5
    [tag_5 0x1 0x5 0xc0fefe]
  jumpi
    // 测试条件为真，跳转到 tag_5.
    //  `jumpi` 从栈中消耗两项数据
    [0x5 0xc0fefe]
  invalid
// 数据访问有效，继续执行
// stack: [0x5 0xc0fefe]
tag_5:
  sstore
    []
    storage: { 0x5 => 0xc0fefe }
```

我们现在已经看见了边界检查代码。我们也看见了编译器可以对这类东西进行一些优化，但是不是非常完美。

559 lines (423 sloc) | 13.4 KB

打包行为

存储是非常昂贵的。一个关键的优化就是尽可能的将数据打包成一个32字节数值。

考虑一个有 4 个存储变量的合约，每个变量都是 64 位，全部加起来就是 256 位（32字节）：

```
pragma solidity ^0.4.11;
contract C {
    uint64 a;
    uint64 b;
    uint64 c;
    uint64 d;
    function C() {
      a = 0xaaaa;
      b = 0xbbbb;
      c = 0xcccc;
      d = 0xdddd;
    }
}
```

我们期望（希望）编译器使用一个 `sstore` 指令将这些数据存放到同一个存储槽中。

编译：

```
$ solc --bin --asm --optimize c-many-variables--packing.sol
```

汇编代码：

```
tag_2:
    /* "c-many-variables--packing.sol":121:122  a */
  0x0
    /* "c-many-variables--packing.sol":121:131  a = 0xaaaa */
  dup1
  sload
    /* "c-many-variables--packing.sol":125:131  0xaaaa */
  0xaaaa
  not(0xffffffffffffffff)
    /* "c-many-variables--packing.sol":121:131  a = 0xaaaa */
  swap1
  swap2
```

```
    and
    or
    not(sub(exp(0x2, 0x80), exp(0x2, 0x40)))
      /* "c-many-variables--packing.sol":139:149  b = 0xbbbb */
    and
    0xbbbb0000000000000000
    or
    not(sub(exp(0x2, 0xc0), exp(0x2, 0x80)))
      /* "c-many-variables--packing.sol":157:167  c = 0xcccc */
    and
    0xcccc00000000000000000000000000000000
    or
    sub(exp(0x2, 0xc0), 0x1)
      /* "c-many-variables--packing.sol":175:185  d = 0xdddd */
    and
    0xdddd0000000000000000000000000000000000000000000000
    or
    swap1
    sstore
```

这里还是有很多的位转移我没能弄明白，但是无所谓。最关键事情是这里只有一个 `sstore` 指令。

这样优化就成功！

干扰优化器

优化器并不能一直工作的这么好。让我们来干扰一下优化器。唯一的改变就是使用协助函数来设置存储变量：

```
pragma solidity ^0.4.11;
contract C {
    uint64 a;
    uint64 b;
    uint64 c;
    uint64 d;
    function C() {
      setAB();
      setCD();
    }
    function setAB() internal {
      a = 0xaaaa;
      b = 0xbbbb;
    }
    function setCD() internal {
      c = 0xcccc;
      d = 0xdddd;
    }
}
```

编译：

```
$ solc --bin --asm --optimize c-many-variables--packing-helpers.sol
```

输出的汇编代码太多了，我们忽略了大多数的细节，只关注结构体：

```
// 构造器函数
tag_2:
  // ...
  // 通过跳到tag_5来调用setAB()
  jump
tag_4:
  // ...
  //通过跳到tag_7来调用setCD()
  jump
// setAB()函数
tag_5:
  // 进行位转移和设置a，b
  // ...
  sstore
tag_9:
  jump  // 返回到调用setAB()的地方
//setCD()函数
tag_7:
  // 进行位转移和设置c，d
  // ...
  sstore
tag_10:
  jump  // 返回到调用setCD()的地方
```

现在这里有两个 `sstore` 指令而不是一个。Solidity编译器可以优化一个标签内的东西，但是无法优化跨标签的。

调用函数会让你消耗更多的成本，不是因为函数调用昂贵（他们只是一个跳转指令），而是因为 `sstore` 指令的优化可能会失败。

为了解决这个问题，Solidity编译器应该学会如何内联函数，本质上就是不用调用函数也能得到相同的代码：

```
a = 0xaaaa;
b = 0xbbbb;
c = 0xcccc;
d = 0xdddd;
```

> 如果我们仔细阅读输出的完整汇编代码，我们会看见 `setAB()` 和 `setCD()` 函数的汇编代码被包含了两次，不仅使代码变得臃肿了，并且还需要花费额外的gas来部署合约。在学习合约的生命周期时我们再来谈谈这个问题。

为什么优化器会被干扰?

因为优化器不会跨标签进行优化。思考一下"1+1"，在同一个标签下，它会被优化成 `0x2`：

```
// 优化成功!
tag_0:
  0x1
  0x1
  add
  ...
```

但是如果指令被标签分开的话就不会被优化了：

```
// 优化失败!
tag_0:
  0x1
  0x1
tag_1:
  add
  ...
```

在0.4.13版本中上面的行为都是真实的。也许未来会改变。

再次干扰优化器

让我们看看优化器失败的另一种方式，打包适用于定长数组吗？思考一下：

```
pragma solidity ^0.4.11;
contract C {
    uint64[4] numbers;
    function C() {
      numbers[0] = 0x0;
      numbers[1] = 0x1111;
      numbers[2] = 0x2222;
      numbers[3] = 0x3333;
    }
}
```

再一次，这里有4个64位的数值我们希望能打包成一个32位的数值，只使用一个 `sstore` 指令。

编译的汇编代码太长了，我们就数数 `sstore` 和 `sload` 指令的条数：

```
$ solc --bin --asm --optimize c-static-array--packing.sol | grep -E '(ssto
  sload
  sstore
  sload
  sstore
  sload
  sstore
  sload
  sstore
```

哦，不！即使定长数组与等效的结构体和存储变量的存储布局是一样的，优化也失败了。现在需要4对 `sload` 和 `sstore` 指令。

快速的看一下汇编代码，可以发现每个数组的访问都有一个边界检查代码，它们在不同的标签下被组织起来。优化无法跨标签，所以优化失败。

不过有个小安慰。其他额外的3个 `sstore` 指令比第一个要便宜：

- `sstore` 指令第一次写入一个新位置需要花费 20000 gas
- `sstore` 指令后续写入一个已存在的位置需要花费 5000 gas

所以这个特殊的优化失败会花费我们35000 gas而不是20000 gas，多了额外的75%。

总结

如果Solidity编译器能弄清楚存储变量的大小，它就会将这些变量依次的放入存储器中。如果可能的话，编译器会将数据紧密的打包成32字节的块。

总结一下目前我们见到的打包行为：

- 存储变量：打包
- 结构体：打包
- 定长数组：不打包。在理论上应该是打包的

因为存储器访问的成本较高，所以你应该将存储变量作为自己的数据库模式。当写一个合约时，做一个小实验是比较有用的，检测汇编代码看看编译器是否进行了正确的优化。

我们可以肯定Solidity编译器在未来肯定会改良。对于现在而言，很不幸，我们不能盲目的相信它的优化器。

它需要你真正的理解存储变量。