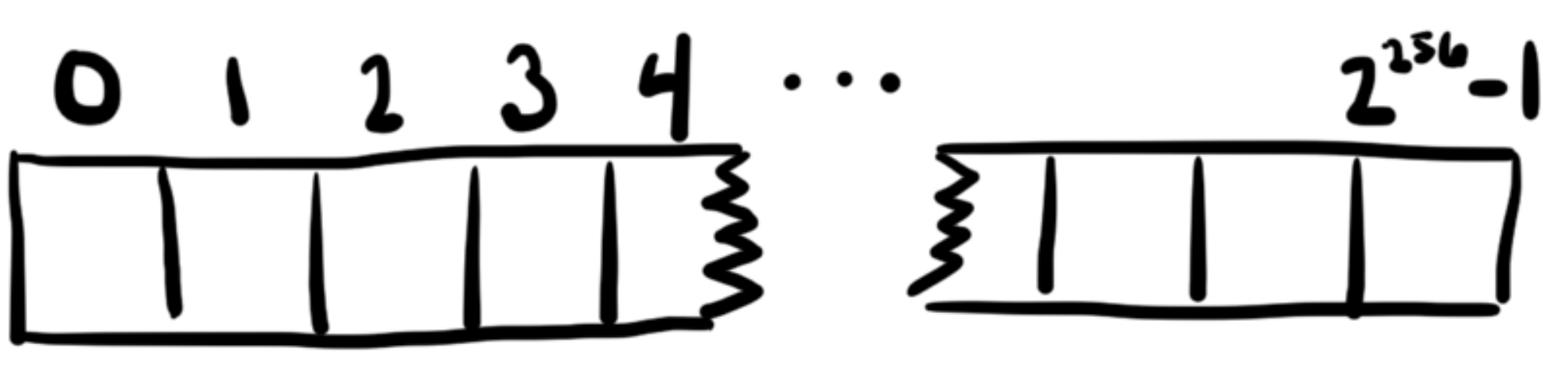
理解以太坊中智能合约中的存储

发表于 2019-04-23 | 更新于 2022-06-02 | 分类于 智能合约 | 阅读次数： 4372
本文字数： 3.8k | 阅读时长 ≈ 3 分钟

以太坊中的智能合约使用了一种不常见的存储模型，如果开发者想要对合约中的变量进行寻址，并修改相应的值，那么本文会给予你一些帮助。在接下来篇幅中，我将详细解释以太坊中合约的存储模型，并展示如何使用solidity编程语言使用它。

超大范围的存储空间

每个运行在以太坊虚拟机（EVM）中的合约会在永久存储空间中（storage）维护一个状态。这个存储空间（storage）是一个非常大的数组，数组长度为2^{256}，其中每个元素的大小为32个字节，数组中的每个元素的初始化值为0。任何智能合约都可以从这个存储空间中任何位置读取值，或写入值。



2^{256}长度的存储空间是不能在组成以太坊网络的物理计算机上实现的，实际上的存储空间是非常稀疏的，不需要存储0值，在寻找的时候通过32字节的键映射到32字节的值，这种键值对的方式完成。

因为零不占用任何存储空间，所以我们可以通过将某个值设置为零来回回收存储空间。在智能合约中，当我们将某个变量设置为零时，系统会退还一部分gas给我们。

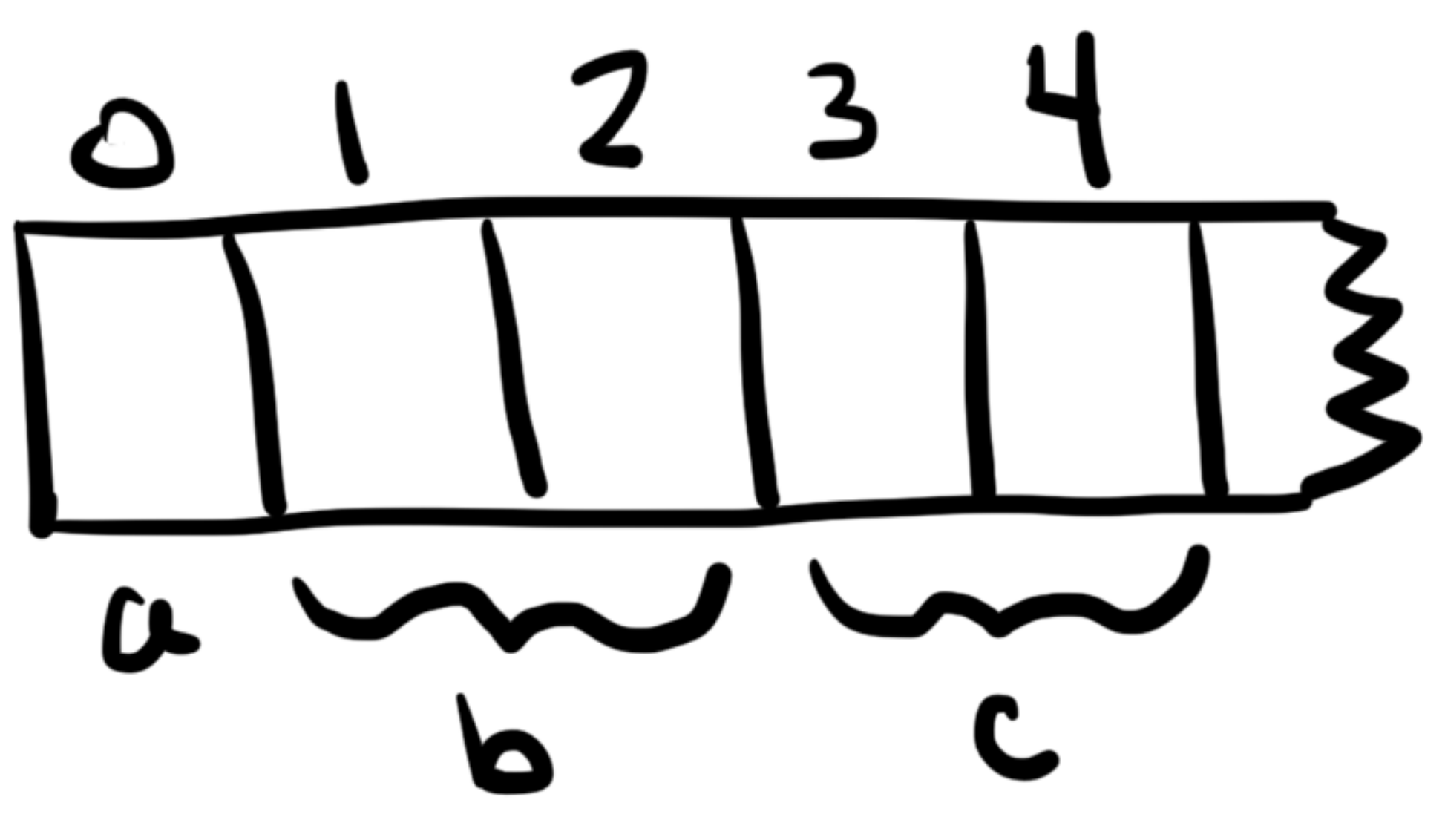
寻找固定长度的值

对于已知拥有固定长度的值，通常的方法是在存储空间给他们分配一个预留的位置存储值。

```
contract StorageTest {
    uint256 a;
    uint256[2] b;

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;
}
```

对于上面的代码：

- `a`存储在*slot* 0（在solidity属于中，存储空间中的每一个位置称为“slot”）
- `b`存储在*slot* 1和2中（因为`b`是一个数组，且数组长度为2）
- `c`存储位置从*slot* 3开始，消耗两个*slot*，因为`Entry`结构体存储了两个32字节长度的值



这些*slot*位置在合约编译的时候就确定了，并且严格按照变量在合约中的定义顺序确定的。

寻找动态长度的值

对于固定长度的值使用预分配位置的方法可以很好的解决，但对于这些动态长度的数组和`mapping`类型的值不起作用，因为无法预先知道到底需要预留多少`slot`给这些动态类型。

对比我们之前比较熟悉的`RAM`内存分配方式，你可能会期望有一个"allocation"方法来找到可用空间，然后有一个"free"方法将空间放回到可用的存储池中。

很遗憾这方法并不可行，由于智能合约中的存储空间是天文数字规模的，存储中有2^{256}可供选择的位置，这大约是已知可观测宇宙中原子的数目。你可以随机选择存储位置，而不用担心会发生冲突。在solidity中使用hash函数为动态长度的类型值计算存储位置。

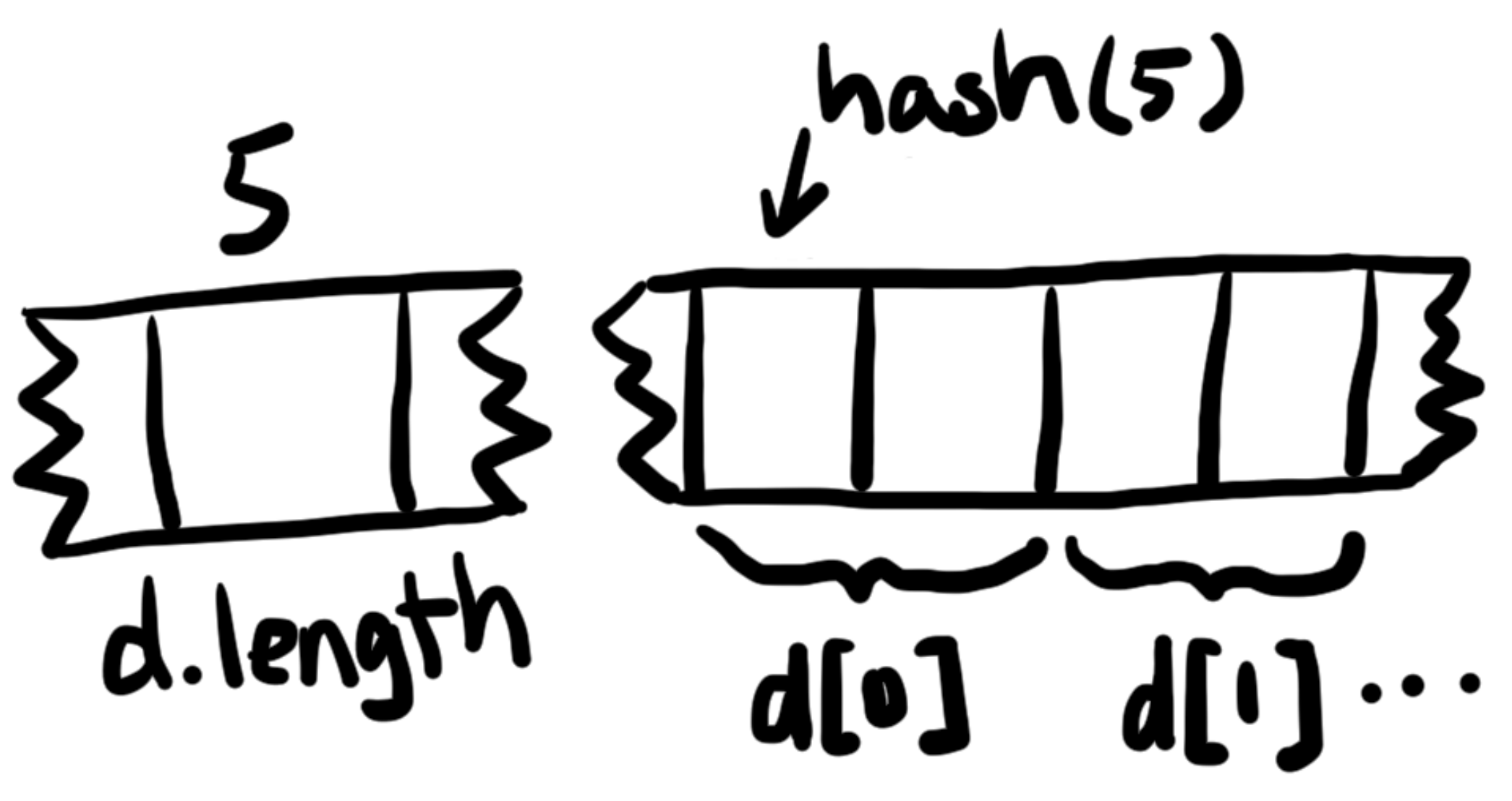
动态长度数组

动态长度的数组需要一个位置来存储数组的长度和数组中的所有元素。

```
contract StorageTest {
    uint256 a;     // slot 0
    uint256[2] b;  // slots 1-2

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;       // slots 3-4
    Entry[] d;
}
```

对于上面的代码：

- 动态数组`d`存储在*slot* 5位置中，但*slot* 5中存储的值是数组`d`的长度，数组中的元素连续存储在以`hash(5)`开始的位置中。即通过对动态数组`d`的`slot`进行hash运算，求出数组中的元素存储位置。



下面一段`solidity`代码用来计算动态数组中元素的位置：

```
function arrLocation(uint256 slot, uint256 index, uint256 elementSize)
    public
    pure
    returns (uint256)
{
    return uint256(keccak256(slot)) + (index * elementSize);
}
```

Mappings

映射需要找到与给定键对应的位置的有效方法。散列键是一个好的开始，但是必须注意确保不同的映射生成不同的位置。

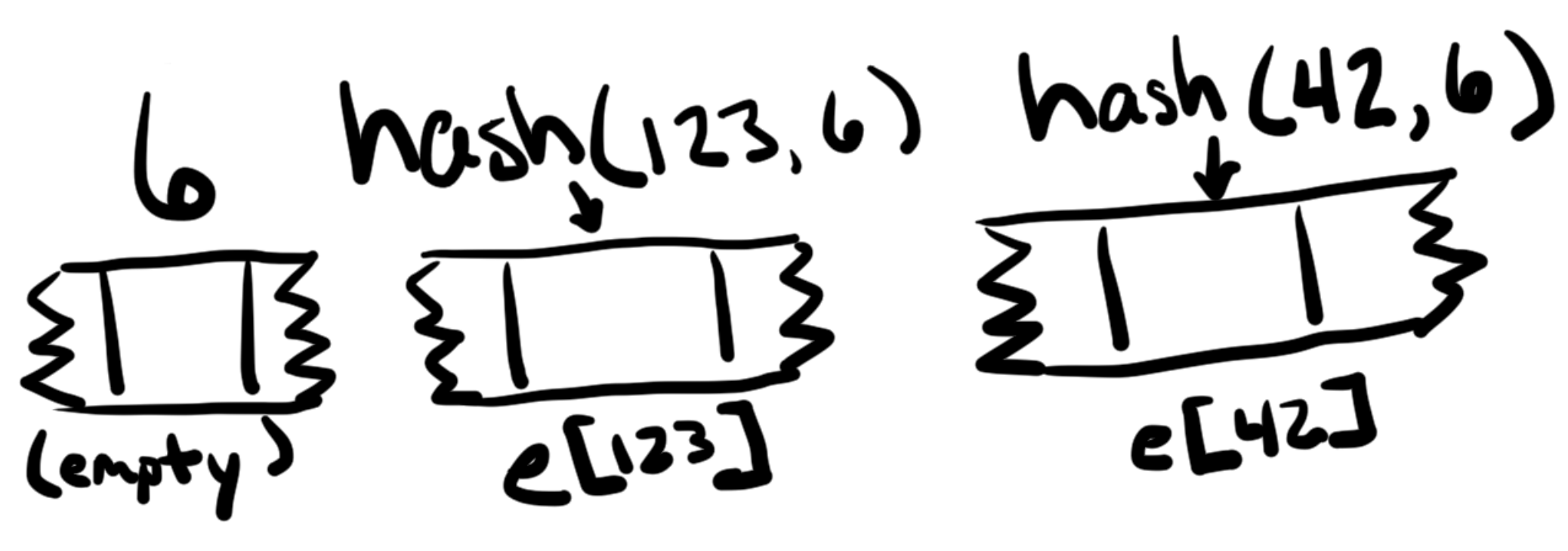
`mapping`需要一个有效的方法通过给定的key找到相应的存储位置，通过对`mapping`的key进行hash运算是一个不错的方法，但需要确保针对不同的`mapping`对象的相同的`key`生成不同的存储位置。

```
contract StorageTest {
    uint256 a;     // slot 0
    uint256[2] b;  // slots 1-2

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;       // slots 3-4
    Entry[] d;     // slot 5 for length, keccak256(5)+ for data

    mapping(uint256 => uint256) e;
    mapping(uint256 => uint256) f;
}
```

对于以上代码：

- `e`的*slot*是6，`f`的*slot*是7，但在这两个位置中并没有存储任何值，因为`mapping`没有长度值需要存储

要寻找`mapping`中的值的位置，需要将`key`与`mapping`的*slot*一起进行hash运算。



下面的`solidity`函数用于计算`mapping`中值的存储位置：

```
function mapLocation(uint256 slot, uint256 key) public pure returns (uint256) {
    return uint256(keccak256(key, slot));
}
```

注意：当传入多个参数到`keccak256`方法时，首先会将这些参数进行连接，然后在进行hash运算。因为是将`mapping`的slot值与`key`值同时进行的hash运算，所以不同`mapping`之间是不会存在冲突的。

复杂类型的组合

动态大小的数组与`mapping`可以相互嵌套在一起，当这种情况发生时，可以通过递归的方式找到值的存储位置。

```
contract StorageTest {
    uint256 a;     // slot 0
    uint256[2] b;  // slots 1-2

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;       // slots 3-4
    Entry[] d;     // slot 5 for length, keccak256(5)+ for data

    mapping(uint256 => uint256) e;    // slot 6, data at h(k . 6)
    mapping(uint256 => uint256) f;    // slot 7, data at h(k . 7)

    mapping(uint256 => uint256[]) g;  // slot 8
    mapping(uint256 => uint256)[] h;  // slot 9
}
```

要寻找这些复杂类型中的值的存储位置可以使用上面定义的函数：

- arrLocation
- mapLocation

例：寻找`g[123][0]`的存储位置：

```
// 首先找到g[123]的位置，g是mapping，g的slot是8，key是123，用mapLocation计算存储位置
arrLoc = mapLocation(8, 123);  // g is at slot 8

// 然后 查找arr[0]
itemLoc = arrLocation(arrLoc, 0, 1);
```

例：寻找 `h[2][456]` 的存储位置：

```
// 首先查找h[2]位置，h是动态数组，h的slot是9
mapLoc = arrLocation(9, 2, 1);  // h is at slot 9

// 然后查找 map[456]位置
itemLoc = mapLocation(mapLoc, 456);
```

总结

- 每个智能合约中的`storage`都是以2^{256}长度的数组形式存在的，并且数组中的所有元素的初始值为0
- 0值是不会被显示存储的，所以当给一个对象赋值0时，就相当于生命回收相应的存储空间
- 对与固定长度的值，solidity是通过预分配的方式分配存储位置的
- 对于动态长度类型的值，Solidity通过hash运算的方法动态确定存储位置

下表展示了如何计算不同类型的存储位置。`slot`表示在合约中定义的变量的位置。

Kind	Declaration	Value	Location
一般类型	`T v`	`v`	v's slot
定长数组	`T[10] v`	`v[n]`	(v's slot) + n * (size of T)
不定长数组	`T[] v`	`v[n]`	keccak256(v's slot) + n * (size of T)
		`v.length`	v's slot
Mapping	`mapping(T1 => T2) v`	`v[key]`	keccak256(key . (v's slot))

相关文章推荐

- 暂停交易？ERC20合约整数溢出安全漏洞案例技术分析（一）
- Roulette Dapps
- 科普：什么是以太坊
- zkRollup介绍 原理篇
- 零知识证明介绍

原作者： 吴寿鹤
本文链接： https://www.whatsblockchain.com/posts/c052872a.html

智能合约 # 以太坊

‹ IPOS共识算法设计
从软件分层开发角度理解tendermint ›