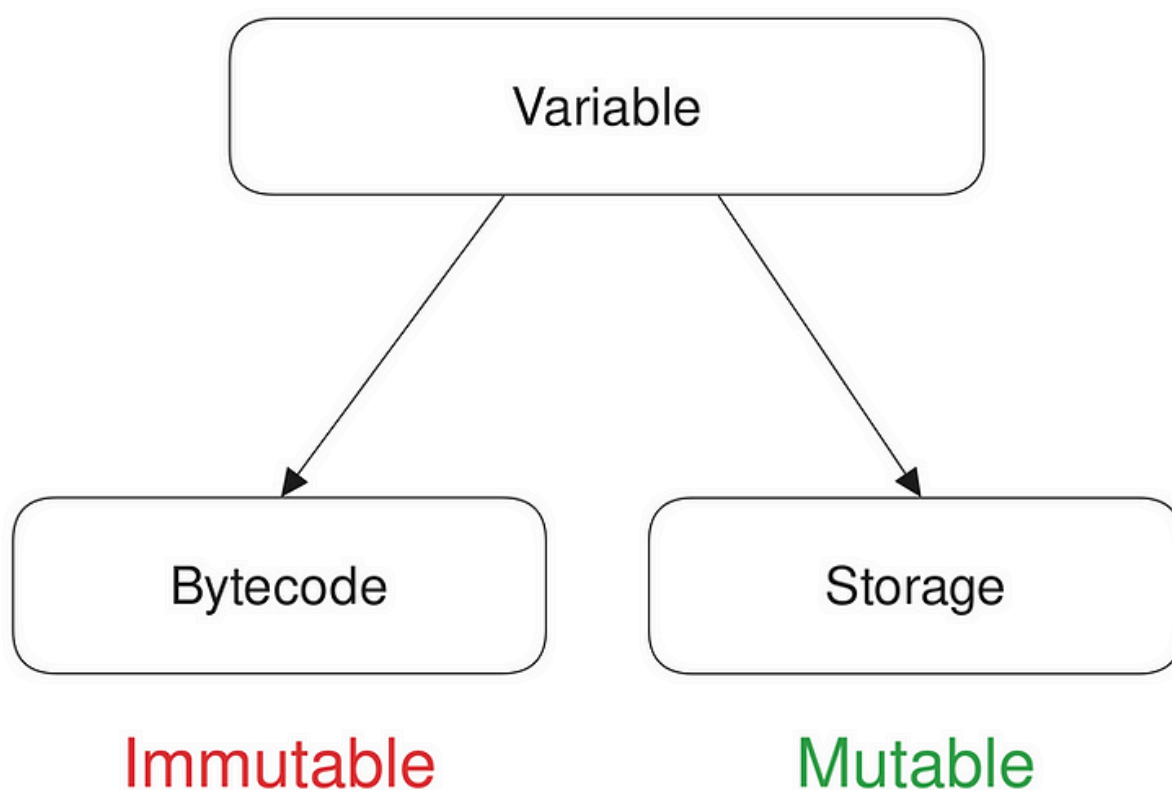


上半节

插槽机制

Solidity的插槽机制和yul语言是SOLIDITY的高级特性，理解这两节内容对理解GAS优化和一些框架代码会有好处，本节课作为相关内容的介绍，不做深度展开，如果有需要了解细节，在文末会有相关的引申资料

状态变量的存储位置



状态变量在Solidity里存放在两个位置，

对于不可更改的数据（immutable,constant）存放于bytecode当中，也就是在合约被编译后就写入了合约字节码文件

对于可更改的数据，存放在storage当中。

📖 实验1，通过讲一个变量设置为immutable/constant来观察gas费的减少

```
contract ImmutableVariables {  
    uint256 constant myConstant = 100;  
}
```

```

uint256 immutable myImmutable;

constructor(uint256 _myImmutable) {
    myImmutable = _myImmutable;
}

function access() public view returns (uint256) {
    return myConstant+1;
}
}

```

🧐 思考1， immutable 和constant都是常量，有什么区别

回答：

- (1) constant在声明时就要被设置初始值
- (2) immutable在声明时可以不初始化，但是在构造函数中需要对其进行初始化

插槽机制介绍 (Slot)

Storage的定义

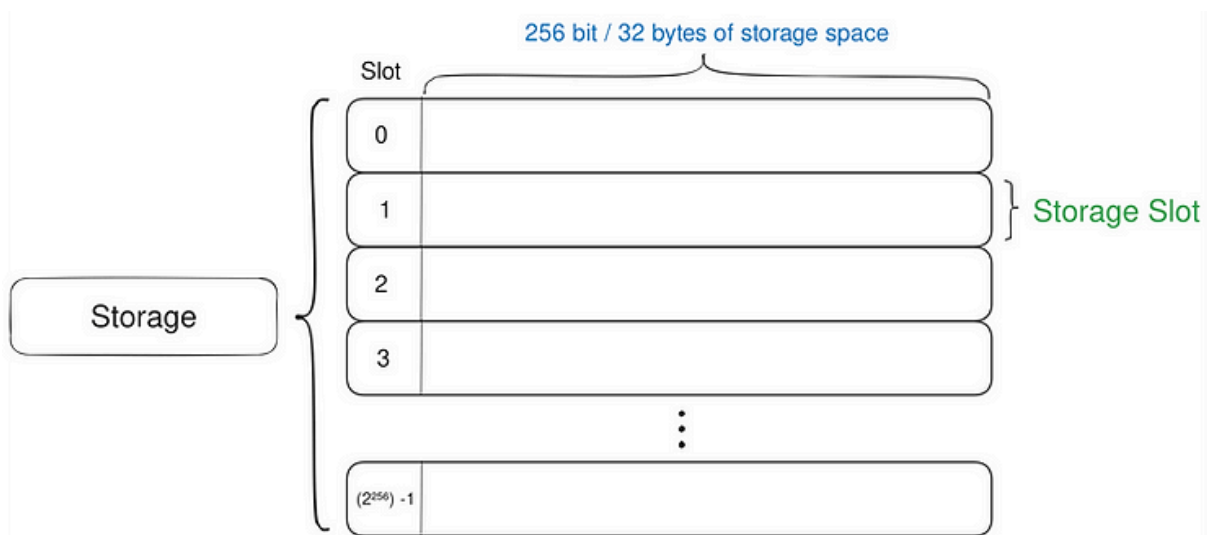


状态变量（除了不可更改的变量），均定义在storage当中，当我们和这些变量交互的时候，实际上就是读写这些storage

Storage Slots

Slots是智能合约存储的一种组织形式，每一个slot都是256bit，也就是32个字节大小。

slot的索引从0~ $2^{256}-1$



💡 分配原则：

Solidity编译器分配Storage空间给状态变量，按照一种有序的、确定性的规则，基于这些变量的定义顺序。如果变量没有被初始化，他们slot的默认值为0。所有的状态变量会被默认设置为0直到他们被显式设置了值

比如下面的x,y就分别占用了slot0， slot1两个slot

```
contract StorageVariables {
    uint256 public x; // first declared storage variable
    uint256 public y; // second declared storage variable
}
```

Slot值的内容

每一个独立的storage按照256-bit格式进行存储，

📖 实验2，演示下uint256 x在slot中的存储形式

```
contract StorageVariables {
    uint256 public x; // Uninitialized storage variable

    function return_uninitialized_X() public view returns (uint256) {
        return x; // returns zero
    }
}
```

```

function set_x(uint256 value) external {
    x = value;
}

function retrieveSlotContent(uint256 index) external view returns(byte
s32 content){
    assembly{
        content:=sload(index)
    }
}

```

基础数据类型的存储

基础数据类型是指长度固定的数据类型，例如uint, int, address, bool, bytes32等

```

contract PrimitiveTypes {
    uint256 a;
    int256 b;
    address owner;
    bool isTrue;
}

```

因为这些数据类型长度固定，不会随着程序逻辑变更长度，所以对于他们空间的分配在编译时就能完全确定，所以他们会按照数据格式大小，放在slot当中，并遵循storage打包的规范

Storage Packing

对于基础数据类型，为了Solidity为了节省空间，为将不满256bit的数据进行压缩，凑满一个slot

```

contract AddressVariable{
    address owner = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
}

```


实验4，分别演示map，array在内存中的存储

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.24;

contract MapStorage{

    uint128 a=111;
    uint128 d=111;
    uint256 c=123;

    mapping(uint⇒uint) test;
    //
    constructor(){
        test[2]=4;
    }

    //传入 2， 2
    function queryMapSlot(uint key,uint slot) external pure returns(uint256 location){
        location=uint256(keccak256(abi.encode(key,slot)));
    }

    function retrieveSlotContent(uint256 index) external view returns(bytes32 content){
        assembly{
            content:=sload(index)
        }
    }
}

//SPDX-License-Identifier:MIT
```

```
pragma solidity ^0.8.24;

contract ArrayStorage{

    uint128 a=111;
    uint128 d=111;
    uint256 c=123;

    mapping(uint⇒uint) test;
    uint[] intArray;
    //
    constructor(){
        test[2]=4;
        intArray.push(2);
        intArray.push(3);

    }

    function retrieveSlotContent(uint256 index) external view returns(bytes3
2 content){
        assembly{
            content:=sload(index)
        }

    }

}
```

YUL语法

yul语法是一种类似汇编的语言，可以被编译成指令。

使用YUL语法的好处

- (1) 能够直接使用更底层的语法和以太坊交互，gas效率更高

(2) 能够屏蔽掉上层solidity版本的变迁

这篇课程里主要讲解的是在我们学习过程中见到的yul语法

Storage的访问

solidity提供两个操作码来读写slot，sload(),sstore()

读slot

sload()指令接受一个slot号，并返回256bit的数据，

slot号可以是一个uint256的整数，也可以通过变量.slot获得

```
contract SlotReadStorage {

    uint256 public x = 11;
    uint256 public y = 22;
    uint256 public z = 33;

    function readSlotX() external view returns (uint256 value) {
        assembly {
            value := sload(x.slot)
        }
    }

    function sloadOpcode(uint256 slotNumber)
        external
        view
        returns (uint256 value)
    {
        assembly {
            value := sload(slotNumber)
        }
    }
}
```

写slot

yul提供sstore(slot,value)用来存储slot,参数的含义如下：

slot:目标slot的值，可以是一个uint256，也可以是x.slot

value:32-byte的数据

```
contract WriteStorage {
    uint256 public x = 11;
    uint256 public y = 22;
    address public owner;

    //直接复制C4地址0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
    constructor(address _owner) {
        owner = _owner;
    }

    // sstore() function

    function sstore_x(uint256 newval) public {
        assembly {
            sstore(x.slot, newval)
        }
    }

    // normal function
    function set_x(uint256 newval) public {
        x = newval;
    }
}
```

课程中出现的yul合约走读

- **write-to-any-slot**

<https://solidity-by-example.org/app/write-to-any-slot/>

💡 这里用到了访问storage的一种用法，通过定义结构体来快速访问storage，可以避免yul语法和数据转换,openzeppelin中有相关的工具包

<https://docs.openzeppelin.com/contracts/5.x/api/utils#StorageSlot>

- **verify signature**

<https://solidity-by-example.org/signature/>

💡 为什么要从32个字节开始，因为前面32个字节为数据的长度，这个是动态数据类型在memory中的表达。

add(sig, 32) 表示指针从sig位置向右移动32个字节

mload(p),从p位置开始读取32个字节

byte(n, x) ,取x的第n个节

- **deploy-any-contract**

<https://solidity-by-example.org/app/deploy-any-contract/>

(1)无参构造函数的creationcode

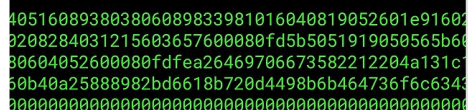
(2)有参构造函数的creationcode

引申阅读

Ethereum smart contract creation code - RareSkills

This article explains what happens at the bytecode level when an Ethereum smart contract is constructed and how the constructor arguments are interpreted.

 <https://www.rareskills.io/post/ethereum-contract-creation-code>



- **Simple Bytecode Contract**

<https://solidity-by-example.org/app/simple-bytecode-contract/>

💡 create(v, p, n)

v:value

p:creationcode的起始位置

n:creationcode的结束位置

```
addr := create(0, add(bytecode, 0x20), 0x13)
```

对汇编的理解

```
// Store run time code to memory  
PUSH10 0X60ff60005260206000f3  
PUSH1 0  
MSTORE
```

这句话是MSTORE的汇编表达形式，第一步压栈存储的runtimecode，第二步压栈0，mstore接受两个参数，第一个是存储的偏移量，第二个是存储的内容，指令入栈后，调用MSTORE时会弹出前两个参数，所以就是mstore(0,0X60ff60005260206000f3)，执行完后栈也就被清空了，数据从指令栈移到了memory中，供返回使用。连上最后的返回语句构成了creationcode，被create语句使用。因为bytes是引用类型，所以在create时前面32位需要跳过。汇编指令要按照堆栈的方式读