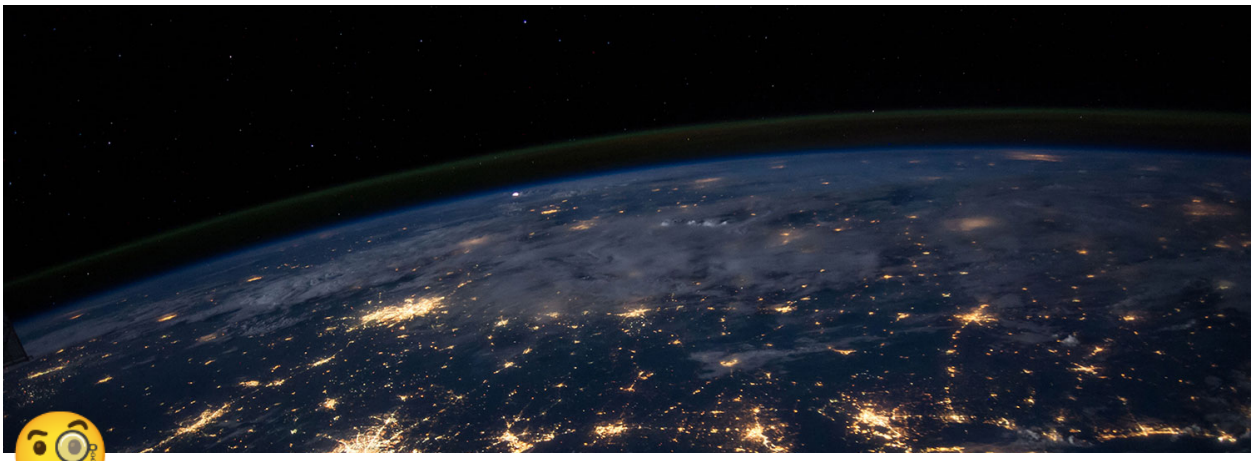


Solidity编程学习

Solidity编程基础

（一）基础语法：

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage { // contract 关键字表示一个智能合约
   uint storedData;  // 状态变量

   function set(uint x) public {
      storedData = x;
   }

   function get() public view returns (uint) {
      return storedData;
   }
}
```

Solidity ^0.6.8 以上版本要求引入 SPDX 许可证，否则会出现警告

pragma是表示源代码是为 Solidity version 0.4.0及以上版本编写的，但不包括 version 0.6.0 及以上版本。

如果^0.4.0 表示可以使用的版本为 0.4.0 ~ 0.4.9 之间的任意版本，但不能超过 0.5.0 版本。0.4.0 ~ 0.4.9 之间的小版本改动通常不会有破坏性更改，源代码应该都是兼容的。

导入文件部分，如下：

```
import "filename";
```

```
import * as symbolName from "filename";
```

保留关键字的表如下：

- abstract
- after
- alias
- apply
- auto
- case
- catch
- copyof
- default
- define
- final
- immutable
- implements
- in
- inline
- let
- macro
- match
- mutable
- null
- of
- override
- partial
- promise
- reference
- relocatable
- sealed
- sizeof
- static

- supports
- switch
- try
- typedef
- typeof
- unchecked

SPDX-License-Identifier 部分是用来声明版权的注释

（二）编译运行：

使用Remix 对相关操作进行截图

（三）代码注释：

```
//   or   /*....*/
```

（四）数据类型：

```
值类型:32位的字节型数据
地址类型:地址类型表示以太坊地址，长度为20字节。
引用类型(组合类型):有一些数据类型由值类型组合而成，这些类型通常通过名称引用，被称为引用类型。
```

4.1 值类型

类型	保留字	取值
布尔型	bool	true/false
整型	int/uint	int8/uint8~int256/uint256 uint8取值为0~2^(8)-1 int8取值为±2^(8-1)-1
定长浮点数	fixed/unfixed	fixedMxN/unfixedMxN M为按类型取的位数 取值8的倍数 N为小数点 取值0~80

上表为常见值类型

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
    uint public a = type(uint).min; // 测量类型的最小值
    uint public b = type(uint).max; // 测量类型的最大值
}
```

type(类型).min / type(类型).max

4.2 地址类型

```
address x = 0x212;
address myAddress = this;

if (x.balance < 10 && myAddress.balance >= 10)
    x.transfer(10);
```

4.3 引用类型

```
数组 (字符串与bytes是特殊的数组，所以也是引用类型)
struct (结构体)
map (映射)
```

（五）变量

- **状态变量** – 变量值永久保存在智能合约存储空间中的变量。
- **局部变量** – 变量值仅在函数执行过程中有效的变量，函数退出后，变量无效。
- **全局变量/特殊变量** – 保存在全局命名空间，用于获取区块链相关信息的特殊变量。

每个变量声明时，都有一个基于其类型的默认值。没有 undefined 或 null 的概念

常见全局变量/特殊变量

Aa 名称	返回
blockhash(uint blockNumber) returns (bytes32)	给定区块的哈希值 – 只适用于256最近区块, 不包含当前区块。
block.coinbase (address payable)	当前区块矿工的地址
block.difficulty (uint)	当前区块的难度
block.gaslimit (uint)	当前区块的gaslimit
block.number (uint)	当前区块的number
block.timestamp (uint)	当前区块的时间戳，为unix纪元以来的秒
gasleft() returns (uint256)	剩余 gas
msg.data (bytes calldata)	完成 calldata
msg.sender (address payable)	消息发送者 (当前 caller)
msg.sig (bytes4)	calldata的前四个字节 (function identifier)
msg.value (uint)	当前消息的wei值
now (uint)	当前块的时间戳
tx.gasprice (uint)	交易的gas价格
tx.origin (address payable)	交易的发送方

·**变量命名规则:**

- 不应使用 Solidity 保留关键字作为变量名。
- 不应以数字(0-9)开头，必须以字母或下划线开头。
- 变量名区分大小写。

变量默认值：

Solidity 智能合约中所有的状态变量和局部变量，都有默认值。

这些变量在没有被赋值之前，它的值已默认值的形式存在。

```
bool 类型变量默认值为 false；

int 类型变量默认值为 0；

uint 类型变量默认值为 0；

address 类型变量默认值为：0x000….000，共 40个 0；

bytes32 类型变量默认值为：0x000….000，共 64个 0。
```

变量作用域：

局部变量的作用域仅限于定义它们的函数，但是状态变量可以有四种作用域类型：

- **public** – 公共状态变量可以在内部访问，也可以从外部访问。对于公共状态变量，将自动生成一个 getter 函数。
- **private** – 私有状态变量只能从当前合约内部访问，派生合约内不能访问。
- **internal** – 内部状态变量只能从当前合约或其派生合约内访问。
- **external** - 外部状态变量只能在合约之外调用 ，不能被合约内的其他函数调用。

常量：

智能合约中，状态变量的值如果恒定不变，就可以通过 `constant` 进行修饰，定义为常量。

常量有如下规定：

- 不是所有的类型都支持常量，当前支持的仅有 `值类型` 和 `字符串`。
- `constant常量` 必须在编译期间通过一个表达式赋值
- 编译器并不会为 `constant常量` 在 `storage` 上预留空间

常量使用的 gas 值要小于状态变量。

（六）运算符

- 算术运算符：+ - * / % ++ —
- 比较运算符：

```
>  <  >=  <=  ==  !=
```

- 逻辑(或关系)运算符：&& || ！【其中 &&，|| 为短路运算符】
- 位运算符： & | ^ ~ << >>

```
序号 运算符与描述
1 & (位与)
对其整数参数的每个位执行位与操作。
例: (A & B) 为 2.
2 | (位或)
对其整数参数的每个位执行位或操作。
例: (A | B) 为 3.
3 ^ (位异或)
对其整数参数的每个位执行位异或操作。
例: (A ^ B) 为 1.
4 ~ (位非)
一元操作符，反转操作数中的所有位。
例: (~B) 为 -4.
5 << (左移位))
将一个值向左移动一个位置相当于乘以2，移动两个位置相当于乘以4，以此类推。
例: (A << 1) 为 4.
6 >> (右移位)
左操作数的值向右移动，移动位置数量由右操作数指定
例: (A >> 1) 为 1.
```

- 赋值运算符：= += -= *= /= %= 同样适用位运算符，如 :

```
<<=   >>=   &=  |=  ^=
```

- 条件(或三元)运算符：?a:b;

（七）条件语句

- if
- if...else
- if...else if

（八）循环语句

- while
- do ... while

- for
- 循环控制语句：break、continue。

（九）引用类型（复杂类型）

9.1 字符串：(string可与bytes内置转换)

字符串值使用双引号(")和单引号(')包括

转义字符表如下：

```
序号 转义字符
1 \n
开始新的一行
2 \\
反斜杠
3 \'
单引号
4 \"
双引号
5 \b
退格
6 \f
换页
7 \r
回车
8 \t
制表符
9 \v
垂直制表符
10  \xNN
表示十六进制值并插入适当的字节。
11  \uNNNN
表示Unicode值并插入UTF-8序列。
```

9.2 数组

数组是一种数据结构，它是存储同类元素的有序集合

数组大小可以是固定大小的，也可以是动态长度的。

对于 storage 数组，元素可以是任意类型(其他数组、映射或结构)。对于 memory 数组，元素类型不能是映射类型，如果它是一个 public 函数的参数，那么元素类型必须是 ABI 类型。

类型为 bytes 和字符串的变量是特殊数组。bytes 类似于 byte[]，但它在 calldata 中被紧密地打包。字符串等价于 bytes，但不允许长度或索引访问。

因此，相比于 byte[]，bytes 应该优先使用，因为更便宜。

```
1.声明数组
要声明一个固定长度的数组，需要指定元素类型和数量
type arrayName [ arraySize ];

2.初始化数组
balance[2] = 5;
```

```
3.访问数组元素
uint salary = balance[2];

4.存储数组(storage arrays)
这些数组被声明为状态变量，并且可以具有固定长度或动态长度。
动态存储数组可以调整数组的大小，它们通过访问push()和pop()方法来调节长度

5.创建内存数组(memory arrays)
可以使用 new 关键字在内存中创建动态数组。
与存储数组相反，不能通过设置 .length 成员来调整内存动态数组的长度
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.0;
contract C {
    function f(uint len) {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // a.length == 7, b.length == len
        a[6] = 8;
    }
}

6.数组成员

1）length:数组有一个 length 成员来表示元素数量。
创建后，内存数组的大小是固定的(但是是动态的，长度可以是函数参数)。
2）push:动态存储数组 和 bytes 有一个 push 成员函数，可用于在数组末尾追加一个元素，
函数返回新的长度。
```

9.3 结构体

```
1.定义结构体
struct struct_name {
   type1 type_name_1;
   type2 type_name_2;
   type3 type_name_3;
}
2.访问结构体成员
struct_name.type_name_1
```

9.4 映射

`mapping(_KeyType => _ValueType)`

- _KeyType：可以是任何内置类型，或者 bytes 和 字符串。不允许使用引用类型或复杂对象。
- _ValueType： 可以是任何类型
- 映射的数据位置(data location)只能是 storage，通常用于状态变量。
- 映射可以标记为 public，Solidity 自动为它创建 getter。

映射可以视作哈希表 ，它们在实际的初始化过程中创建每个可能的 key，并将其映射到字节形式全是零的值：一个类型的 默认值。然而下面是映射与哈希表不同的地方：在映射中，实际上并不存储 key，而是存储它的 keccak256 哈希值，从而便于查询实际的值。

正因为如此，映射是没有长度的，也没有 key 的集合或 value 的集合的概念。映射只能是存储的数据位置，因此只允许作为状态变量或作为函数内的存储引用 或 作为库函数的参数。 它们不能用合约公有函数的参数或返回值。

可以将映射声明为 public，然后来让 Solidity 创建一个 getter 函数。 _KeyType 将成为 getter 的必须参数，并且 getter 会返回 _ValueType。

9.5 枚举

solidity 的枚举类型 enum 是一种用户自定义类型，用于表示多种状态。

枚举类型 enum 内部就是一个自定义的整型，默认的类型为`uint8`，当枚举数足够多时，它会自动变成`uint16`。

枚举类型 enum可以与整数进行显式转换，但不能进行隐式转换。显示转换会在运行时检查数值范围，如果不匹配，将会引起异常。

枚举类型 enum 至少应该有一名成员。

9.6 类型转换

（十）数据位置（引用类型）

引用类型涉及到的数据量较大，复制它们可能要消耗大量Gas，非常昂贵，所以使用它们时，必须考虑存储位置。例如，是保存在内存中，还是在 EVM 存储区中。

在合约中声明和使用的变量都有一个数据位置，指明变量值应该存储在哪里。合约变量的数据位置将会影响Gas消耗量。

Solidity 提供4种类型的数据位置。

- storage
- memory
- calldata
- stack

```
1) storage
该存储位置存储永久数据，这意味着该数据可以被合约中的所有函数访问。
可以把它视为计算机的硬盘数据，所有数据都永久存储。

保存在存储区(storage)中的变量，以智能合约的状态存储，并且在函数调用之间保持持久性。
与其他数据位置相比，存储区数据位置的成本较高。
```

```
2) memory
内存位置是临时数据，比存储位置便宜。它只能在函数中访问。

通常，内存数据用于保存临时变量，以便在函数执行期间进行计算。一旦函数执行完毕，
它的内容就会被丢弃。你可以把它想象成每个单独函数的内存(RAM)。

3) calldata
calldata是不可修改的非持久性数据位置，所有传递给函数的值，都存储在这里。
此外，calldata是外部函数的参数(而不是返回参数)的默认位置。

4) stack
堆栈是由EVM (Ethereum虚拟机)维护的非持久性数据。EVM使用堆栈数据位置在执行期间加载变量。
堆栈位置最多有1024个级别的限制。

可以看到，要永久性存储，可以保存在存储区(storage)。
```

10.1 **变量的数据位置规则**

1）状态变量总是存储在存储区中，不能显式地标记状态变量的位置。storage

2）函数参数包括返回参数都存储在内存中。memory

3）值类型的局部变量存储在内存中，不能显式覆盖；对于引用类型，需要显式地指定数据位置。

4）外部函数的参数(不包括返回参数)存储在 calldata中。

10.2 **赋值的数据位置规则**

数据可以通过两种方式从一个变量复制到另一个变量。一种方法是复制整个数据(按值复制)，另一种方法是引用复制。

1）**状态变量赋值给状态变量（值类型与引用类型皆是）**

将一个状态变量赋值给另一个状态变量，将创建一个新的副本(可以不随着变量而变化）

如果它不创建副本，那么 stateVar1 的值应该是30，创建副本则是20

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Locations {
  uint public stateVar1 = 10;
  uint public stateVar2 = 20;

  function do() public returns (uint) {

   stateVar1 = stateVar2;
   stateVar2 = 30;

   return stateVar1; //returns 20
  }
}
```

2）**内存局部变量复制到状态变量**

从内存局部变量复制到状态变量，总是会创建一个新的副本

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Locations {
    uint stateVar = 10; //storage

    function do() public returns(uint) {
        uint localVar = 20; //memory
        stateVar = localVar;
        localVar = 40;

        return stateVar; //returns 20
    }
}
```

3）**状态变量复制到内存局部变量**

从状态变量复制到内存局部变量，将创建一个副本。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Locations {
    uint stateVar = 10; //storage

    function do() public returns(uint) {
        uint localVar = 20; //memory

        localVar = stateVar;
        stateVar = 40;

        return localVar; //returns 10
    }
}
```

4）**内存变量复制到内存变量**

对于引用类型的局部变量，从一个内存变量复制到另一个内存变量，不会创建副本。

得到相同的结果，因为它们都指向相同的位置。

```
// SPDX-License-Identifier: MIT
pragma solidity ^ 0.8.0;

contract Locations {
```

```
    function doSomething()
        public pure returns(uint[] memory, uint[] memory) {

        uint[] memory localMemoryArray1 = new uint[](3);
        localMemoryArray1[0] = 4;
        localMemoryArray1[1] = 5;
        localMemoryArray1[2] = 6;

        uint[] memory localMemoryArray2 = localMemoryArray1;
        localMemoryArray1[0] = 10;

        return (localMemoryArray1, localMemoryArray2);
       //returns 10,4,6 | 10,4,6
    }
}
```

对于值类型的局部变量，从一个内存变量复制到另一个内存变量，仍然创建一个新副本。

```
// SPDX-License-Identifier: MIT
pragma solidity ^ 0.8.0;

contract Locations {
    function do() public pure returns(uint) {
        uint localVar1 = 10; //memory
        uint localVar2 = 20; //memory

        localVar1 = localVar2;
        localVar2 = 40;

        return localVar1; //returns 20
    }
}
```

（十一）函数

函数是一组可重用代码的包装，接受输入，返回输出

11. 1 函数定义与调用

```
function function_name(<parameter list>) <visibility> <state mutability>
[returns(<return type>)] {
    //语句
}
```

可见性（visibility）

- Private（私有）：函数只能在所定义的智能合约内部调用。
- Internal（内部）：可以在所定义智能合约内部调用该函数，也可以从继承合约中调用该函数。

- External（外部）：只能从智能合约外部调用。 如果要从智能合约中调用它，则必须使用 this。
- Public（公开）：可以从任何地方调用。

状态可变性（mutability）

- view：用view声明的函数只能读取状态，而不能修改状态。
- pure：用pure声明的函数既不能读取也不能修改状态。
- payable：用payable声明的函数可以接受发送给合约的以太币，如果未指定，该函数将自动拒绝所有发送给它的以太币

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
   function getResult() public view returns(uint){
      uint a = 1; // 局部变量
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

函数调用：要调用函数，只需使用函数名，并传入参数即可

return 语句：函数可以返回多个值

11. 2 **函数返回值**

Solidity 函数的返回值可以使用名字，也可以采用匿名方式。

Solidity 函数的返回值可以通过名字赋值，也可以使用 return 返回。

Solidity 函数支持多个返回值

```
// SPDX-License-Identifier: MIT
pragma solidity^0.8.0;

contract funreturn{

    //返回值可以有名字
    function returnTest()public view returns(uint mul){
        uint a= 10;
        return a;
    }

    //可以直接为返回值赋值
    function returnTest2()public view returns(uint mul){
        mul = 10;
    }
    //当给返回值赋值后，并且有个return，以最后的return为主
    function returnTest3()public view returns(uint mul){
```

```
        uint a= 10;
        mul = 100;
        return a;
    }

    //返回常量，自动匹配
     function returnTest4()public view returns(uint mul){
        uint a= 10;
        mul = 100;
        return 1;
    }

    //函数可以有多个返回值，多返回值赋值
     function returnTest5(uint a,uint b) public view returns(uint add,uint mul){
       add = a+b;
       mul = a*b;
    }

    //函数可以有多返回值，返回return(param list)
    function returnTest6(uint a,uint b) public view returns(uint add,uint mul){
       return(a+b,a*b);
    }

    //交换变量的值
    function returnTest7(uint a，uint b) public view returns(uint a1,uint b1){
       return (b,a);
    }
}
```

Solidity 函数多返回值的调用方法

```
// SPDX-License-Identifier: MIT
pragma solidity^0.8.0;

contract funreturn{

    // 定义多返回值函数
    function returnFunc() private pure returns(uint a, uint b){
        return (1, 2);
    }

    // 调用多返回值函数
    function callFunc()public pure returns(uint, uint){
       (uint r1, uint r2) = returnFunc();
       return (r1, r2);
    }

    // 调用多返回值函数，先定义变量
    function callFunc()public pure returns(uint, uint){
       uint r1；
       uint r2；
       (r1, r2) = returnFunc();
       return (r1, r2);
    }

   // 调用多返回值函数，但只取第一个返回值
    function callFunc()public pure returns(uint, uint){
       (uint r1, ) = returnFunc();
       return (r1, 100);
```

```
    }
}
```

11. 3 **pure函数**

solidity pure函数，也就是纯函数，是指函数不会读取或修改状态。

换言之，solidity pure函数不会操作链上数据。

如果函数中存在以下语句，则被视为读取状态，编译器将抛出警告。

- 读取状态变量。
- 访问 `address(this).balance` 或 `<address>.balance`
- 访问任何区块、交易、msg等特殊变量(msg.sig 与 msg.data 允许读取)。
- 调用任何不是纯函数的函数。
- 使用包含特定操作码的内联程序集。

如果发生错误，pure函数可以使用`revert()`和`require()`函数来还原潜在的状态更改。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
   function add(uint a, uint b) public pure returns(uint){
      return a + b;
   }
}
```

11. 4 **view函数**

solidity view函数，也就是视图函数，是指函数只会读取状态，不会修改状态。

换言之，solidity pure函数只会读取链上数据，不会修改链上数据。

如果函数中存在以下语句，则被视为修改状态，编译器将抛出警告。

- 修改状态变量。
- 触发事件。
- 创建合约。
- 使用`selfdestruct`。
- 发送以太。
- 调用任何不是视图函数或纯函数的函数

- 使用底层调用
- 使用包含某些操作码的内联程序集。

【状态变量的Getter方法默认是view函数】

11. 5 **构造函数**

Solidity构造函数是一个特殊函数，它仅能在智能合约部署的时候调用一次，之后就不能再次被调用。

Solidity构造函数常用来进行状态变量的初始化工作。

Solidity编译器中，使用关键词 constructor 作为构造函数。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
    uint a;

   //不带参数的构造函数
   constructor() {
     a = 0;
   }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
    uint a;

   // 带参数的构造函数
   constructor(uint _a) {
     a = _a;
   }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
   int public a ;
   address public owner;

   constructor(uint _a) public{
     // 将部署者地址存储到owner变量
      owner = msg.sender;
    // 将参数_a存储到a变量
      a = _a;
   }
}
```

11. 6 **modifier 函数修改器**

我们可以将一些通用的操作提取出来，包装为函数修改器，来提高代码的复用性，改善编码效率。

函数修改器 modifier 的作用与 Java Spring 中的切面功能很相似，当它作用于一个函数上，可以在函数执行前或后预先执行 modifier 中的逻辑，以增强其功能。

函数修改器 modifier 常用于在函数执行前检查某种前置条件。

函数修改器 modifier 是一种合约属性，可被继承，同时还可被派生的合约重写(override)。

```
/// Ownable 可以判断合约的调用者是否为当前合约的owner，
/// 从而避免其他人随意的调用一些合约的关键操作。
/// 同时，owner 可以指定任何其他人为此合约新的 owner，
/// 显然，只有当前owner才能指定其他人为新的owner。
contract Ownable {
    // 变量 owner 指定此合约的owner
    address public owner;
    // 发布事件 - 此合约owner已经换人（此逻辑与modifier无关，可以忽略）
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    // 构造函数 - 创建合约自动执行，初始化合约所有人为合约创建者
    function Ownable() public {
        owner = msg.sender;
    }

    // 定义一个函数修改器
    modifier onlyOwner() {
        // 判断此函数调用者是否为owner
        require(msg.sender == owner);
        _;
    }

    // owner可以用此函数将owner所有权转换给其他人，显然次函数只有owner才能调用
    // 函数末尾加上onlyOwner声明，onlyOwner正是上面定义的modifier
    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}
```

11. 7 **函数重载**

Solidity的函数重载，是指同一个作用域内，相同函数名可以定义多个函数。

这些函数的参数(参数类型或参数数量)必须不一样。仅仅是返回值不一样是不被允许。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
   function getSum(uint a, uint b) public pure returns(uint){
      return a + b;
   }
```

```
    function getSum(uint a, uint b, uint c) public pure returns(uint){
        return a + b + c;
    }
    function callSumWithTwoArguments() public pure returns(uint){
        return getSum(1,2);
    }
    function callSumWithThreeArguments() public pure returns(uint){
        return getSum(1,2,3);
    }
}
```

11. 8 数学函数

- `addmod(uint x, uint y, uint k) returns (uint)` 计算(x + y) % k，计算中，以任意精度执行加法，且不限于2^256大小。
- `mulmod(uint x, uint y, uint k) returns (uint)` 计算(x * y) % k，计算中，以任意精度执行乘法，且不限于2^256大小。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
    function callAddMod() public pure returns(uint){
        return addmod(4, 5, 3);
    }
    function callMulMod() public pure returns(uint){
        return mulmod(4, 5, 3);
    }
}
```

11. 9 加密函数

- `keccak256(bytes memory) returns (bytes32)` 计算输入的Keccak-256散列。
- `sha256(bytes memory) returns (bytes32)` 计算输入的SHA-256散列。
- `ripemd160(bytes memory) returns (bytes20)` 计算输入的RIPEMD-160散列。
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)` 从椭圆曲线签名中恢复与公钥相关的地址，或在出错时返回零。函数参数对应于签名的ECDSA值: r – 签名的前32字节; s: 签名的第二个32字节; v: 签名的最后一个字节。这个方法返回一个地址。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Test {
    function callKeccak256() public pure returns(bytes32 result){  // 此处指定了输出参数名
        return keccak256("ABC");
```

```
    }
}// 输出结果格式为 0：bytes32:result  散列值
```

（十二）进阶编程

12. 1 事件 Event

Soliddity Event 事件是以太坊虚拟机(EVM)日志基础设施提供的一个便利接口。

当被发送事件（调用）时，会触发参数存储到交易的日志中。这些日志与合约的地址关联，并记录到区块链中。

区块链是打包一系列交易的区块组成的链条，每个交易“收据”会包含0到多个日志记录，日志表明着智能合约所触发的事件。

记录区块链的日志，可以使用状态变量，也可以使用事件 Event，但 Event 使用的 gas 费比状态变量低。

```
event EventName(<parameter list>);

emit EventName(<parameter list>);

--------------------------------------
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Envent {
  // 定义 event
  event Log(string, uint);

  function operations() external{
    // 触发 event
     emit Log("Info", 123);
  }
}
----------------------------------------
事件 Event 还有一种特殊形式 event indexed，也就是索引事件:

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Envent {
  // 定义 event
  event Log(address indexed, string);

  function operations() external{
    // 触发 event
     emit Log(msg.sender, "Info");
  }
}// 一个事件最多给3个参数索引
```

12. 2 **不可变量 immutable**

Solidity immutable 是另一种常量的表达方式。与常量类似，但是不必硬编码，可以在构造函数时传值，部署后无法改变。

immutable 不可变量同样不会占用状态变量存储空间，在部署时，变量的值会被追加的运行时字节码中， 因此它**比使用状态变量便宜的多**，也同样带来了更多的安全性。

immutable 特性在很多时候非常有用， 最常见的如 ERC20 代币用来指示小数位置的 `decimals` 变量，它是一个不能修改的变量，很多时候我们需要在创建合约的时候指定它的值，这时 immutable 就大有用武之地， 类似的还有要保创建者地址、关联合约地址等。

immutable 不可变量三种赋值方式：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Immutable {
   address public immutable owner = msg.sender;
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Immutable {
   address public immutable owner;

   constructor() {
      owner = msg.sender;
   }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Immutable {
   address public immutable owner;

   constructor(address _owner) {
      owner = _owner;
   }
}
```

immutable 不可变量，只能在状态变量声明和构造函数中赋值，其它位置不允许

12. 3 **log 日志**

使用 solidity 编写的智能合约，调试时可以通过打印 Log 的方式，查看合约运行过程中的数据。

在合约中创建一个event，命名为 Log。在想要打印日志的地方调用事件 emit Log(...)，就可以查看运行过程中的数据。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SolidityTest {
    // 定义事件
    event Log(address);

    constructor()  {
        // 调用事件
        emit Log(msg.sender);
        emit Log(address(this));
    }
}
```

Log(msg.sender) 在日志中输出了合约部署者的地址。

Log(address(this)) 在日志中输出了合约地址。

查看合约在部署时的日志结果：

```
[
    {
        "from": "0xE3Ca443c9fd7AF40A2B5a95d43207E763e56005F",
        "topic": "0x2c2ecbc2212ac38c2f9ec89aa5fcef7f532a5db24dbf7cad1f48bc82843b7428",
        "event": "log",
        "args": {
            // 合约部署者的地址
            "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"
        }
    },
    {
        "from": "0xE3Ca443c9fd7AF40A2B5a95d43207E763e56005F",
        "topic": "0x2c2ecbc2212ac38c2f9ec89aa5fcef7f532a5db24dbf7cad1f48bc82843b7428",
        "event": "log",
        "args": {
            // 合约地址
            "0": "0xE3Ca443c9fd7AF40A2B5a95d43207E763e56005F"
        }
    }
]
```

12. 4 **合约继承**

Solidity 语言是一种面向对象的编程语言，提供了对合约继承的支持，继承是扩展合约功能的一种方式。

Solidity 语言的合约继承通过关键字 is 来实现。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Person{
  string public name;
  uint public age;
  function getSalary() external pure returns(uint){
    return 0;
  }
}
contract Employee is Person{
}
```

合约 Employee 继承了合约 Person，运行后，我们看到 Employee 继承了状态变量 name、age 和方法 getSalary。

solidity 引入了 virtual，override 关键字，用于重写函数。

父合约可以使用 virtual 关键字声明一个虚函数，子合约使用 override 关键字来覆盖父合约的方法，如果合约 Manager 又继承了 Employee，而且还需要覆盖 getSalary 方法，那么需要如下写法：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Person{
  string public name;
  uint public age;
  function getSalary() external pure virtual returns(uint){
    return 0;
  }
}
contract Employee is Person{
  function getSalary() external pure virtual override returns(uint){
    return 3000;
  }
}

contract getSalary is Employee{
  function getSex() external pure override returns(uint){
    return 20000;
  }
}
```

抽象合约

solidity 还允许在基类中只声明函数原型，没有实现，而在派生类中再去实现。

solidity 使用 abstract 关键字来标记基类。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

abstract contract Employee {
    function getSalary() public  pure virtual returns(int);
}
```

```
contract Manager is Employee {
    function getSalary() public  pure override returns(int){
        return 20000;
    }
}
```

抽象合约 abstract 的作用是将函数定义和具体实现分离，从而实现解耦、可拓展性，其使用规则为：

- 当合约中有未实现的函数时，则合约必须修饰为abstract；
- 当合约继承的base合约中有构造函数，但是当前合约并没有对其进行传参时，则必须修饰为abstract；
- abstract合约中未实现的函数必须在子合约中实现，即所有在abstract中定义的函数都必须有实现；
- abstract合约不能单独部署，必须被继承后才能部署；

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Animal {
    string public species;
    constructor(string memory _base) {
        species = _base;
    }
}

abstract contract Feline {
    uint public num;
    function utterance() public pure virtual returns (bytes32);

    function base(uint _num) public returns(uint, string memory) {
        num = _num;
        return (num, "hello world!");
    }
}

// 由于Animal中的构造函数没有进行初始化，所以必须修饰为abstract
abstract contract Cat1 is Feline, Animal {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}

contract Cat2 is Feline, Animal("Animal") {
    function utterance() public pure override returns (bytes32) { return "miaow"; }
}
```

子类访问父类权限修饰符包括：public、internal、private

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract A{
  uint stateVar;

  function somePublicFun() public{}
  function someInternalFun() internal{}
  function somePrivateFun() private{}
```

```
}

contract B is A{
  function call(){
    //访问父类的`public`方法
    somePublicFun();

    //访问父类的状态变量（状态变量默认是internal权限）
    stateVar = 10;

    //访问父类的`internal`方法
    someInternalFun();

    //不能访问`private`
    //somePrivateFun();
  }
}
```

子类传参数到父类有两种方式:

```
// 直接传递
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Base{
  uint a;
  constructor(uint _a){
    a = _a;
  }
}

contract Derive is Base(1){
  function getBasePara() external view returns(uint){
    return a;
  }
}
-------------------------------------

// 根据输入值传递
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Base{
  uint a;

  constructor(uint _a){
    a = _a;
  }
}

contract T is Base{
  constructor(uint _a) Base(_a) {
  }

  function getBasePara() external view returns (uint){
    return a;
  }
}
```

多重继承中的重名

多重继承中不允许出现相同的函数名、事件名、修改器名以及状态变量名等。

```
pragma solidity ^0.8.0;

contract Employee1 {
    function getSalary() public pure returns(int){
        return 1;
    }
}

contract Employee2 {
    function getSalary() public pure returns(int){
        return 1;
    }
}

contract Manager is Employee1,Employee2 {
}
```

基类 Employee1、Employee2 中同时包含函数 getSalary，报错

```
pragma solidity ^0.8.0;

contract Employee {
    function getSalary() public  pure returns(int){
        return 1;
    }
}

contract Manager is Employee {
    function getSalary() public  pure returns(int){
        return 2;
    }
}
```

基类 Employee 和 父类 Manager 中同时包含函数 getSalary，构成重名，所以以上代码会出现编译错误。

还有一种比较隐蔽的情况，默认状态变量的`getter`函数导致的重名:

```
pragma solidity ^0.8.0;

contract Employee1 {
  uint public data = 10;
}

contract Employee2 {
  function data() returns(uint){
    return 1;
  }
}
contract Manager is Employee1, Employee2{}
```

Employee1 的状态变量 data，会默认生成 getter，函数名为 data()，于是 Employee1 和 Employee2 函数重名出错。

12. 5 多重继承

Solidity 语言提供了对合约继承的支持，而且支持多重继承。

Solidity 语言的多重继承采用线性继承方式。继承顺序很重要，判断顺序的一个简单规则是按照“最类似基类”到“最多派生”的顺序指定基类。

第一种情况：基类 X，Y 没有继承关系，派生类 Z 继承了 X，Y。

```
X  Y
/  \
\  /
 Z
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract X{
   function x() external pure returns(string memory) {
      return "x";
   }
}

contract Y{
   function y() external pure returns(string memory) {
      return "y";
   }
}

contract Z is X,Y{
   function z() external pure returns(string memory) {
      return "z";
   }
}
```

派生合约 Z 继承了 X 的 x() 和 Y 的 y()，另外还定义了自己的函数 z()。

第二种情况：派生类 Y 继承了基类 X，派生类 Z 同时继承了 X，Y:

```
  X
/   \
Y   |
\   /
  Z
```

我们按照线性继承原则，理清继承顺序：X，Y，Z。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract X{
   function foo() external pure virtual returns(string memory) {
      return "x foo";
   }

   function bar() external pure virtual returns(string memory) {
      return "x bar";
   }

   function x() external pure returns(string memory) {
      return "x";
   }
}

contract Y is X{
   function foo() external pure virtual override returns(string memory) {
      return "y foo";
   }

   function bar() external pure virtual override returns(string memory) {
      return "y bar";
   }

   function y() external pure returns(string memory) {
      return "y";
   }
}

contract Z is X,Y{
   function foo() external pure override(X,Y) returns(string memory) {
      return "y foo";
   }

   function bar() external pure override(X,Y) returns(string memory) {
      return "z bar";
   }

   function z() external pure returns(string memory) {
      return "z";
   }
}
```

按照线性继承顺序，Z 继承了 Y，Y 继承了X，那么 Z 的写法是 Z is X,Y。如果写成 Z is Y,X，编译器就会报错。Z的方法 foo() ，覆盖了 X 和 Y 的方法 foo() ，所以需要写为 override(X,Y)

```
   X
 /   \
Y     A
|     |
|     B
 \   /
   Z
```

我们按照线性继承原则，理清继承顺序：X，Y，A，B，Z

12. 6 多重继承的构造函数

1) 已知基类初始化参数

可以在派生类的继承声明中，直接传递参数给基类的构造函数:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract X {
   string public name;
   constructor(string memory _name) {
      name = _name;
   }
}

contract Y {
   string public value;
   constructor(string memory _value) {
      value = _value;
   }
}

// 派生类的继承声明中，直接传递参数给基类的构造函数
contract Z is X("n"),Y("v") {
}
```

2) 部署时传递初始化参数

要在部署时或者运行时，由调用方传递基类初始化参数。在这种情况下，我们需要编写一个新的构造函数，传递参数给基类。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract X {
   string public name;
   constructor(string memory _name) {
      name = _name;
   }
}

contract Y {
   string public value;
   constructor(string memory _value) {
      value = _value;
   }
}

// 编写一个新的构造函数，传递参数给基类
contract Z is X,Y {
   constructor(string memory _name, string memory _value) X(_name) Y(_value){
```

```
   }
}
```

3) **混写方式**

同时使用上面的两种方式，定义和部署时，分别传递参数给基类的构造函数

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract X {
   string public name;
   constructor(string memory _name) {
      name = _name;
   }
}

contract Y {
   string public value;
   constructor(string memory _value) {
      value = _value;
   }
}

// 混写方式
contract Z is X("n"),Y {
   constructor(string memory _value) Y(_value){
   }
}
```

4) **继承顺序**

多重继承中，构造函数的执行会按照定义时的继承顺序进行，与构造函数中定义顺序无关

```
contract Z is X,Y {
   // 会按照继承顺序X,Y，先执行X的构造函数，再执行Y的构造函数，最后执行Z的构造函数
   constructor() Y("v"), X("n"){
       // 先运行 X("n")，再运行 Y("v")
   }
}
```

12. 7 **调用父类函数**

- 使用父级合约名称调用
- 使用super调用

使用父级合约名称调用，格式为：<parent contract>.<method>

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract X {
   function foo() internal pure returns(uint) {
      return 1;
   }
}

contract Z is X {
   function test() external pure returns(uint) {
      // X 为父级合约名称，foo为父级合约的方法名称
      uint result = X.foo();
      return result;
   }
}
```

使用super调用，格式为：super.<method> 单个继承：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract X {
   function foo() internal pure returns(uint) {
      return 1;
   }
}

contract Z is X {
   function test() external pure returns(uint) {
      // supper 表示父级合约，foo为父级合约的方法名称
      uint result = supper.foo();
      return result;
   }
}
```

在多重继承中使用super调用，将会调用所有父级合约(基类)的方法

```
  X
/   \
Y   Z
\   /
  T
-----------------------------
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract X {
   event log(string message);

   function foo() public virtual {
      emit log("X.foo");
```

```
    }
}

contract Y is X {
    function foo() public virtual override {
        emit log("Y.foo");
        super.foo();
    }
}

contract Z is X {
    function foo() public virtual override {
        emit log("Z.foo");
        super.foo();
    }
}

contract T is Y,Z {
    function foo() public override(Y,Z) {
        super.foo();
    }
}
```

12. 8 **异常处理**

Solidity 是通过回退状态的方式来处理异常错误。

Solidity 发生异常时，会撤消当前调用和所有子调用改变的状态，同时给调用者返回一个错误标识。

Solidity 提供了require 、assert 和 revert 来处理异常。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SolidityTest {
    bool public flag;
    function setFlag() external {
        // 判断调用者地址 是否等于当前合约地址，成立则继续运行。
        require(msg.sender == address(this));
        flag = true;
    }
}
```

require 函数常常用来检查输入变量或状态变量是否满足条件，以及验证调用外部合约的返回值。

require 可以有返回值，例如：require(condition, 'Something bad happened');。

require 的返回值不宜过长，因为返回信息需要消耗 gas。

1) **require、assert、revert 共同点**

assert()与require()语句都需要满足括号中的条件，才能进行后续操作，若不满足则抛出错误。

以下三个语句的功能完全相同：

```
// revert
if(msg.sender != owner) {
   revert();
 }
// require
require(msg.sender == owner);

// assert
assert(msg.sender == owner);
```

2) **require、assert 不同点**

assert(false) 编译为 0xfe，这是一个无效的操作码，所以会消耗掉所有剩余的 gas，并恢复所有的操作。

require(false) 编译为 0xfd，这是revert()的操作码，所以会退还所有剩余的 gas，同时可以返回一个自定义的报错信息。

require 的 gas 消耗要小于 assert，而且可以有返回值，使用更为灵活。

3) **require、assert 使用场景**

require() 函数用于检测输入变量或状态变量是否满足条件，以及验证调用外部合约的返回值。

require() 语句的失败报错，应该被看作一个正常的判断语句流程不能通过的事件。

assert()语句的失败报错，意味着发生了代码层面的错误事件，很大可能是合约中有一个bug需要修复。

4) **使用 require() 的场景**

- 验证用户输入，例如：require(input_var>100)
- 验证外部合约的调用结果，例如：require(external.send(amount))
- 在执行状态更改操作之前验证状态条件，例如：require(block.number > 49999) 或 require(balance[msg.sender]>=amount)

一般来说，使用require()的频率更多，通常应用于函数的开头。

5) **使用 assert() 的场景**

- 检查溢出
- 检查不变量
- 更改后验证状态
- 预防永远不会发生的情况

一般来说，使用assert()的频率较少，通常用于函数的结尾。

基本上，require() 应该用于检查条件，而 assert() 只是为了防止发生任何非常糟糕的事情。

6) **try...catch**

要是能够捕获外部合约调用异常，然后根据情况做自己的处理不是更好吗？所以，这种场景下适应于使用 try...catch 语句。

```
pragma solidity ^0.8.0;
contract Manager {
    function count() public pure returns(int){
        require(1==2,"require error");
        return 2;
    }

    function test() public view returns(string memory) {
        try this.count()  {
            return "success";
        } catch Error(string memory reason/* 出错原因 */) {
            // 调用 count() 失败时执行，通常是不满足 require 语句条件或触发 revert 语句时所引起的调用失败
            return reason;
        } catch (bytes memory) {
            // 调用 count() 异常时执行，通常是触发 assert 语句或除 0 等比较严重错误时会执行
            return "assert error";
        }
    }
}
```

以上代码将会触发 catch Error(string memory reason) ，最终输出 require error

```
pragma solidity ^0.8.0;
contract Manager {
    function count() public pure returns(int){
        assert(1==2);
        return 2;
    }

    function test() public view returns(string memory) {
        try this.count()  {
            return "success";
        } catch Error(string memory reason/* 出错原因 */) {
            // 调用 count() 失败时执行，通常是不满足 require 语句条件或触发 revert 语句时所引起的调用失败
            return reason;
        } catch (bytes memory) {
            // 调用 count() 异常时执行，通常是触发 assert 语句或除 0 等比较严重错误时会执行
            return "assert error";
        }
    }
}
```

代码将会触发 catch (bytes memory) ，最终输出 assert error

12. 9 **编程风格**

良好统一的编程风格，有助于提高代码的可读性和可维护性

1) **代码布局:**

- **缩进:** 使用4个空格代替制表符作为缩进，避免空格与制表符混用。
- **空2行规则:** 2个合约定义之间空2行。
- **空1行规则：**2个函数之间空1行。在只有声明的情况下，不需要空行
- **行长度:** 一行不超过79个字符。
- **换行规则:** 函数声明中左括号不换行，每个参数一行并缩进，右括号换行，并对齐左括号所在行。
- **源码编码** UTF-8
- **Import** Import语句应该放在文件的顶部，pragma声明之后。
- **函数顺序** 函数应该根据它们的可见性来分组。
- **避免多余空格**避免在圆括号、方括号或大括号后有空格。
- **控制结构**大括号的左括号不换行，右括号换行，与左括号所在行对齐。
- **函数声明**使用上面的大括号规则。添加可见性标签。可见性标签应该放在自定义修饰符之前。
- **映射**在声明映射变量时避免多余空格。
- **变量声明**声明数组变量时避免多余空格。
- **字符串声明**使用双引号声明字符串，而不是单引号。

2) **代码中各部分的顺序**

代码中各部分顺序如下：

- Pragma 语句
- Import 语句
- Interface
- 库
- Contract

在Interface、Library或Contract中，各部分顺序应为：

- Type declaration / 类型声明(enum，struct)
- State variable / 状态变量
- Event / 事件
- Function / 函数

3) **命名约定**

- 合约和库应该使用驼峰式命名。例如，SmartContract, Owner等。
- 合约和库名应该匹配它们的文件名。
- 如果文件中有多个合约/库，请使用核心合约/库的名称。
- **结构体名称** 驼峰式命名，例如: SmartCoin
- **事件名称** 驼峰式命名，例如：AfterTransfer
- **函数名** 驼峰式命名，首字母小写，比如：initiateSupply
- **局部变量和状态变量** 驼峰式命名，首字母小写，比如creatorAddress、supply
- **常量** 大写字母单词用下划线分隔，例如：MAX_BLOCKS
- **修饰符的名字** 驼峰式命名，首字母小写，例如：onlyAfter
- **枚举的名字** 驼峰式命名，例如：TokenGroup

12. 10 **访问权限**

1）**访问权限**

private 函数和状态变量仅在当前合约中可以访问，在继承的合约内不可访问。

internal 函数和状态变量可以在当前合约或继承合约里调用。需要注意的是不能加前缀 this，前缀 this 是表示通过外部方式访问。

public 函数是合约接口的一部分，可以通过内部或者消息来进行调用。对于 public 类型的状态变量，会自动创建一个访问器。

external 外部函数是合约接口的一部分，所以我们可以从其它合约或通过交易来发起调用。一个外部函数 f，不能通过内部的方式来发起调用，如f()不可以调用，但可以通过this.f()。外部函数在接收大的数组数据时更加有效。

2）**默认状态**

状态变量在函数外部声明(类似于class的属性)，并永久存储在以太坊区块链中，更具体地说存储在存储 Merkle Patricia 树中，这是形成帐户状态信息的一部分。状态变量默认类型为 internal。internal 和 private 类型的变量不能被外部访问，而 public 变量能够被外部访问。

合约中的方法默认为 public 类型。

子合约可以访问 public 和 internal，无法访问 private 类型。

12. 11 **存储位置 memory，storage，calldata**

引用类型在 Solidity 中数据有一个额外的属性：存储位置，可选项为 memory 和 storage。

- memory：存储在内存中，即分配、即使用，越过作用域则不可访问，等待被回收。
- storage：永久存储在以太坊区块链中，更具体地说存储在存储 Merkle Patricia 树中，形成帐户状态信息的一部分。一旦使用这个类型，数据将永远存在。
- calldata：存储函数参数，它是只读的，不会永久存储的一个数据位置。外部函数的参数被强制指定为 calldata，效果与 memory 类似。

1） **强制的数据位置**

外部函数 (external function) 的参数强制为：calldata。

状态变量强制为: storage。

2） **默认数据位置**

函数参数，返回参数：memory。

局部变量：storage。

3） **转换问题**

storage–>storage 只是修改其指针(引用传递)

memory–>storage 分为 2 种情况：

a. 将 memory–>状态变量; 即将内存中的变量拷贝到存储中（值传递）

b.将memeory–>局部变量 报错

storage–>memory: 即将数据从storage拷贝到memory中

memory–>memory 是引用传递

12. 12 **引用类型注意点**

[引用类型进行传递时传递的是其指针，而引用类型进行传递时可以为值传递也可以为引用传递]

1）**可变字节数组**

- string:是一个动态尺寸的utf-8编码字符串，他其实是一个特殊的可变字节数组，同时其也是一个引用类型

bytes：动态字节数组

注：

1.string并没有提供方法获取其字符串长度，也没提供方法修改某个索引的字节码，但是可以把string转换成bytes进行相应的操作(例如：bytes(XXX).length ; bytes(XXX)[0]=a)

2.可变字节数组创建方式：bytes public a = new bytes(1);

3.清空字节数组的方式：（1）a.length = 0;　(2)delete a;

4.push方法:例如 a.push(b) 往字节数组添加字节

5.字节数组与字符串之间的转换：

动态大小字节数组—>string

固定大小字节数组—>动态大小字节数组—>string

a.固定字节数组转动态字节数组如下：

```
contract Test{
        bytes4 public a = 0x54657374;
        function test1() constant returns(bytes){
           bytes memory b = new bytes(a.length);//创建可变字节数组
           for(uint i = 0;i<a.length;i++){
                b[i] = a[i];
            }
           return b;
        }
   }
```

b.动态字节数组转string

```
function getString() constant returns(string){
        return string(test1());  // 返回值是动态字节数组
    }
```

2）**数组**

1. 固定长度数组:声明方式uint[5] T = [1,2,3,4,5];

2. 可变长度数组：

方式1. uint [] T = [1,2,3,4,5];

方式2.uint [] T = new uint[](5);

注：固定长度数组创建后不可对长度进行修改,但是可以对内容进行修改（这是与不可变字节数组之间不同点）

3. 二维数组：

uint [2][3] T = [[1,2],[3,4],[5,6]]; 行列顺序相反

T.length 为 3

这点与java不同，java创建则是 [[1,2,3],[4,5,6]]

注：uint[2][] T = new uint[2][] (n); 创建新数组

注：

1.uint [] memory a = new uint;

用此方式创建数组时，若数组为成员变量, 则默认为storage类型；若为局部变量默认为memory类型，memory类型的数组长度创建后不可变。

```
contract T {
    uint[] memory b = new uint[](5);//错误，状态类型只能是storage
    function test(){
        uint[] memory a = new uint[](5);
        a[5] = 8;
        //a.length = 6; 错误
    }
}
```

数组内元素类型转换

```
contract T {
   function t() public {
     s([1,2]);
   }
   function s(uint[2] _arr) public {

   }
}
```

但是注意:函数 s 中数组类型是uint256，而函数 t 中输入的数组类型是uint8, 这里需要将 uint8 转换一下 s([uint(1),2]);

3）**结构体**

```
contract T{
    struct Test{
        uint a;
        uint b;
    }
}
```

将一个 struct 赋值给一个局部变量（默认是storage类型），实际是拷贝的引用，所以修改局部变量值时，会影响到原变量。

初始化：Test t = Test(1,2)；

12. 13 **值类型注意点 [值类型传值时会将值拷贝一份，对其修改时并不会对原来值有影响]**

1）**整型补充说明**

（1）var声明：var类型表示第一次使用时所表示的类型

（2）除法截断：整数的除法会被截断（例如：1/4结果为0）,但是使用字面量的方式不会被截断

```
pragma solidity ^0.4.0;
contract IntegerTest{
    function get() returns (int){
        int a = 1;
```

```
        int b = 4;
        var c = 1 / 4 * 4;//未截断
    return c;
    }
}
```

2）**Address 地址类型**

表示一个账户的地址，在以太坊中地址的长度为20字节，一字节8位，一个 address 就是 160 位，所以 address 可以用 uint160 表示。

- 地址类型的成员：

属性：balance

函数：transfer()，send()，call()，callcode()，delegatecall()

(1) .balance:它能得到以Wei为单位的地址类型的余额。

(2) .transfer(uint256 amount)：向地址类型发送数量为amount的Wei，失败时抛出异常，不可调节。

(3) .send(uint256 amount) returns (bool):向地址类型 发送数量为 amount 的 Wei，失败时返回 false，不可调节。

补充：send 与transfer对应，但send更底层。如果执行失败，transfer不会因异常停止，而send会返回false。send() 执行有一些风险：如果调用栈的深度超过1024或gas耗光，交易都会失败。因此，为了保证安全，必须检查send的返回值，如果交易失败，会回退以太币。如果用transfer会更好。

(4) .call(…) returns (bool):发出低级函数 CALL，失败时返回 false，发送所有可用 gas，可调节。

(5) .callcode(…) returns (bool)：发出低级函数 CALLCODE，失败时返回 false，发送所有可用 gas，可调节。

(6) .delegatecall(…) returns (bool):发出低级函数 DELEGATECALL，失败时返回 false，发送所有可用 gas，可调节。

上面的这三个方法call()，callcode()，delegatecall()都是底层的消息传递调用，最好仅在万不得已才进行使用，因为他们破坏了Solidity的类型安全。

注：

(1).msg.sender:表示当前调用方法时的发起人，调用方法的人很多，如何判断合约的拥有者？在第一次部署的时候进行定义，即在构造函数中定义

```
contract Test {
    address public _owner;
    function Test() {
        _owner = msg.sender;
    }
}
```

(2).合约地址：合约部署后，会有一个合约地址，合约地址表示合约本身，可以用this表示

```
contract Test {
    address public _owner;
    function Test() {
        _owner = msg.sender;
    }
     function returnContractAddress() constant returns (address) {
        return this;
    }
}
```

3）**定长字节数组（固定大小字节数组）**

定义方式bytesN，其中N可取1~32中的任意整数，bytes1代表只能存储一个字节。一旦声明，其内部的字节长度不可修改，内部字节不可修改

运算符

比较：<=，<，==，!=，>=，>，返回值为bool类型。

位运算符：&，|，^(异或)，~非

支持序号的访问，与大多数语言一样，取值范围[0, n)，其中n表示长度。

注：

可以通过 .length返回字节个数，可以通过索引读取对应索引的字节。

4）**枚举（Enums）**

枚举类型是在Solidity中的一种用户自定义类型。

```
enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
```

ActionChoices 可以理解为一个自定义的整型，当枚举数量不够多时默认类型为uint8，此例子中可以理解成uint8,当枚举数量足够多时，他会自动变成uint16。

5）**函数**

- 可以将一个函数赋值给一个变量，一个函数类型的变量。
- 还可以将一个函数作为参数进行传递。
- 也可以在函数调用中返回一个函数。

注：函数调用方式有两种;internal和external。（这里的external是调用方式，不要访问权限中的那个混淆）

内部函数(internal) –默认是这种类型

因为不能在当前合约的上下文环境以外的地方执行，内部函数只能在当前合约内被使用。如在当前的代码块内，包括内部库函数，和继承的函数中。

外部函数（External）–调用此函数需要用this：这个this是指合约。

外部函数由地址和函数方法签名两部分组成。可作为外部函数调用的参数，或者由外部函数调用返回。

```
pragma solidity ^0.4.5;
contract FuntionTest{
    function internalFunc() internal{}
    function externalFunc() external{}
    function callFunc(){
        //直接使用内部的方式调用
        internalFunc();
        //不能在内部调用一个外部函数，会报编译错误。
        // externalFunc();
        //不能通过`external`的方式调用一个`internal`
        //this.internalFunc();
        //使用`this`以`external`的方式调用一个外部函数
        this.externalFunc();
    }
}
contract FunctionTest1{
function externalCall(FuntionTest ft){
    //调用另一个合约的外部函数
    ft.externalFunc();
    //不能调用另一个合约的内部函数
    //ft.internalFunc();
    }
}
```

补：回退函数 fallback

每一个合约有且仅有一个没有名字的函数。这个函数无参数，也无返回值。当调用的函数找不到时，就会调用默认的 fallback 函数

```
pragma solidity ^0.4.0;
contract SimpleFallback{
  function(){
    //fallback function
  }
}
```

12. 14 合约结构

合约包含内容：

usingFor声明,状态变量（State Variables）,结构类型（Structs Types）,构造函数,函数修饰符（Function Modifiers）,函数（Functions）,事件（Events）,枚举类型（Enum Types）

```
pragma solidity ^0.4.0;    //版本声明
import "./A.sol";   //导入声明
contract SolidityStructure{   //合约声明

   uint balance;//状态变量

   address owner;
```

```
    struct Hello { // 结构类型
        uint helloNum;
        address hello;

    constructor() public{  //构造函数
        owner = msg.sender;
    }
    //function HelloWorld(){
    //} 这种方式也可以

    modifier onlySeller() { // 修饰器
        require(
            msg.sender != owner
        );
        _;
    }

    function test() public {  //函数
       uint step = 10;
       if (owner == msg.sender) {
          balance = balance + step;
       }
    }

    function update(uint amount) constant returns (address, uint){ //带返回值的函数
        balance += amount;
        return (msg.sender, balance);
    }

  using LibraryTest for uint; //using声明

  uint a = 1;

  function getNewA()returns (uint){
      return a.add();
  }

    function kill() {   //析构函数
        if (owner == msg.sender) {
             selfdestruct(owner);
         }
     }

    event HighestBidIncreased(address bidder, uint amount);//事件 log日志打印

    function bid() public payable {
        emit HighestBidIncreased(msg.sender, msg.value); // 触发事件打印相关日志
    }

     enum State { Created, Locked, Inactive } // 枚举
}
```

1）**状态变量**

类似java中类的属性变量,状态变量是永久的存储在合约中的值（强制是storage类型）

状态变量可以被定义为constant即常量，例如：uint constant x = 1;

2）**结构类型**

自定义的将几个变量组合在一起形成的类型,有点类似javabean

3）**构造函数**

构造函数可用constructor关键字进行声明，也可用function HelloWorld(){} 这种方式声明，当合约对象创建时会先调用构造函数对数据进行初始化操作,构造函数只允许存在一个

4）**函数修饰符（函数修改器）**

函数修饰符用于’增强语义’，可以用来轻易的改变一个函数的行为，比如用于在函数执行前检查某种前置条件。修改器是一种合约属性，可被继承，同时还可被派生的合约重写。_表示使用修改符的函数体的替换位置。当然函数修饰器可以传参数

5）**成员函数**

```
function test() public  /  function update(uint amount) constant returns (address,uint)
```

这两种都可以为合约的成员函数，成员函数类似java中基本函数，但是略有不同，不同点在于有返回值时在函数上指定返回值returns(uint),函数调用方式可以设置为内部（Internal）的和外部（External）的，在权限章节会进行介绍

注意：constant只是一个承诺，承诺该函数不会修改区块链上的状态

###using for

使用方式是using A for B

用来把A中的函数关联到到任意类型B，B类型的对象调用A里面的函数，被调用的函数，将会默认接收B类型的对象的实例作为第一个参数。

```
pragma solidity ^0.4.0;
library LibraryTest{
function use(uint a) returns(uint){
    return a+1;
    }
}

contract usingTest{
    using LibraryTest for uint;//把LibraryTest中的函数关联到uint类型
    uint test = 1;
    function testusing() returns (uint){
        return test.use();
//uint类型的对象实例test调用LibraryTest里的函数add();add()会默认接收test作为第一个参数。
    }
}
```

6）**析构函数**

```
selfdestruct()
```

所谓的析构函数是和构造函数相对应，构造函数是初始化数据，而析构函数是销毁数据

7）**事件**

事件是以太坊虚拟机(EVM)日志基础设施提供的一个便利接口。用于获取当前发生的事件。事件在合约中可被继承。

8）**枚举**

可以显式的转换与整数进行转换，但不能进行隐式转换。显示的转换会在运行时检查数值范围，如果不匹配，将会引起异常。 枚举类型应至少有一名成员。

12. 15 编写智能合约

.interface 接口

函数不允许有函数体

```
interface A{
   function testA();
}
```

.library 库

库与合约类似，但它的目的是在一个指定的地址，且仅部署一次，然后通过 EVM 的特性来复用代码

```
library Set {
    struct Data { mapping(uint => bool) flags; }
    function test(){
    }
}
```

其他合约调用库文件的内容直接通过库文件名.方法名例如：Set.test()。

注：一份源文件可以包含多个版本声明、多个导入声明和多个合约声明。

12. 16 mapping delete

1）mapping

一种键值对的映射关系存储结构。语法：mapping(_Key => _Value)，键值对类型，键是唯一的，其赋值方式为：map[a]=test; 意思是键为a,值为test；

注意：

- 1.键的类型允许除映射外的所有类型，如数组，合约，枚举，结构体。值的类型无限制。
- 2.在映射表里没有长度，键集合，值集合这样的概念，同时映射并没有做迭代的方法，可以自行实现

2）**delete**

用于将某个变量重置为初始值。对于整数，运算符的效果等同于a = 0。而对于定长数组，则是把数组中的每个元素置为初始值，变长数组则是将长度置为0。对于结构体，也是类似，是将所有的成员均重置为初始值。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.0;
contract A {
    uint data;

    function change(uint  i) internal {
        data = i;
    }

    function getData() public returns (uint) {
        delete data;
        return data;
    }
}
```

12. 17 **0.6 重大变化**

1）**新的 fallback 函数写法**

在 0.6 之前的版本，我们可以定义下面的 fallback 函数，用来通过合约接收 eth 转账或未指定明确合约函数的调用。

```
function() external payable {
    currentBalance = address(this).balance + msg.value;
}
```

从 0.6 开始，这种写法就要报编译错误了。

新的写法是下面这样的:

```
fallback() external {
}
receive() payable external {
   currentBalance = currentBalance + msg.value;
}
```

对于这种新的写法，有几点是要注意的：

- 1. fallback 和 receive 不是普通函数，而是新的函数类型，有特别的含义，所以在它们前面加 function 这个关键字。加上 function 之后，它们就变成了一般的函数，只能按一般函数来去调用。
- 2. 每个合约最多有一个不带任何参数不带 function 关键字的 fallback 和 receive 函数。

- 3. receive 函数类型必须是 payable 的，并且里面的语句只有在通过外部地址往合约里转账的时候执行。
- 4. fallback 函数类型可以是 payable，也可以不是 payable 的，如果不是 payable 的，可以往合约发送非转账交易，如果交易里带有转账信息，交易会被 revert；如果是 payable 的，自然也就可以接受转账了。
- 5. 尽管 fallback 可以是 payable 的，但并不建议这么做，声明为 payable 之后，其所消耗的 gas 最大量就会被限定在 2300。

2） **对合约继承更好的支持**

这个版本之前合约继承可以这么写的，看起来比较简单，语义上并不是很清晰

```
contract Employee {
    function getSalary() public;
}

contract Manager is Employee {
    function increaseSalary() public {

    }
    function getSalary() public {

    }
}
```

从 0.6 开始，solidity 引入了 abstract, virtual, override 几个关键字，继承关系需要用下面的写法

```
abstract contract Employee {
    function getSalary() public virtual;
}

contract Manager is Employee {
    function increaseSalary() public {

    }

    function getSalary() public override {

    }
}
```

3）**其它特性**

上面只是列了几个比较大的变化，还有一些其它变化也是值得注意的：

- 1. 动态数组的长度从 0.6 开始不可更改了。
- 2. 开始部分支持数组切片了。
- 3. 结构体和枚举类型可以在合约外声明了，之前是只能在合约内声明的。

- 4. 如果父合约声明了某个非 private 的状态变量，子合约中就不能再声明同名状态变量。
- 5. 从 address 到 address payable 的转换现在可以通过 payable(x) 进行 ，其中 x 必须是 address 类型。

12. 18 transfer 实现转账

使用 Solidity 智能合约转账可以使用 transfer 函数。智能合约里面需要有一定的以太，不然合约将无法给调用者发送以太，可以在创建合约时给合约发送一定的以太来测试。

具有转账功能的智能合约的 constructor 必须显式的指定为 payable。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.0;
contract  cs{
    constructor() payable{
    }

    function getETH() public{
        require(address(this).balance>=1 ether,"no money");
        address payable _owner = msg.sender;
        _owner.transfer(1 ether);
    }

    fallback() external{
    }

    receive() payable external{
    }
}
```

12. 19 transfer、send、call的区别和用法

transfer、send、call都可以在合约之间相互转账，但是用法有很大的不同！

1）**transfer**

- 如果异常会转账失败，抛出异常(等价于require(send()))（合约地址转账）
- 有gas限制，最大2300
- 函数原型：<address payable>.transfer(uint256 amount)

2）**send**

- 如果异常会转账失败，仅会返回false，不会终止执行（合约地址转账）
- 有gas限制，最大2300
- 函数原型：<address payable>.send(uint256 amount) returns (bool)

3）**call**

- 如果异常会转账失败，仅会返回false，不会终止执行（调用合约的方法并转账）
- 没有gas限制
- <address>.call(bytes memory) returns (bool, bytes memory)

4）共同点

- addr.transfer(1 ether)、addr.send(1 ether)、addr.call.value(1 ether)的接收方都是addr。
- 如果使用addr.transfer(1 ether)、addr.send(1 ether)，addr合约中必须增加fallback回退函数！
- 如果使用addr.call.value(1 ether)，那么被调用的方法必须添加payable修饰符，否则转账失败！

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.0;

contract  cs{
    constructor() payable{
    }

    function getETH() public returns(bool) {
        address payable _owner = msg.sender;
        return(_owner.send(1 ether));
    }
    # 如果使用transfer或send函数必须添加fallback回退函数
    fallback() external{
    }

    receive() payable external{
    }
}
}
```

地址类型的成员

`<address>.balance` (`uint256`) :

魏的地址平衡

`<address payable>.transfer(uint256 amount)` :

发送给定量的魏到地址，恢复失败，转发2300燃气补助，不可调

`<address payable>.send(uint256 amount) returns (bool)` :

发送给定量的魏到地址，返回 `false` 失败，转发2300燃气补助，不可调

`<address>.call(bytes memory) returns (bool, bytes memory)` :

`CALL` 使用给定的有效负载发出低级别，返回成功条件并返回数据，转发所有可用的气体，可调节

`<address>.delegatecall(bytes memory) returns (bool, bytes memory)` :

`DELEGATECALL` 使用给定的有效负载发出低级别，返回成功条件并返回数据，转发所有可用的气体，可调节

`<address>.staticcall(bytes memory) returns (bool, bytes memory)` :

`STATICCALL` 使用给定的有效负载发出低级别，返回成功条件并返回数据，转发所有可用的气体，可调节

有关更多信息，请参阅“ 地址 ”部分。

注意

以前版本的Solidity允许这些函数接收任意参数，并且还可以处理 `bytes4` 不同类型的第一个参数。在版本0.5.0中删除了这些边缘情况。

可以使用 `.gas()` 修改器调整供应的气体：

```
address(nameReg).call.gas(1000000)(abi.encodeWithSignature("register(string)", "MyName"));
```

同样，也可以控制提供的Ether值：

```
address(nameReg).call.value(1 ether)(abi.encodeWithSignature("register(string)", "MyName"));
```

最后，可以组合这些修饰符。他们的订单无关紧要：

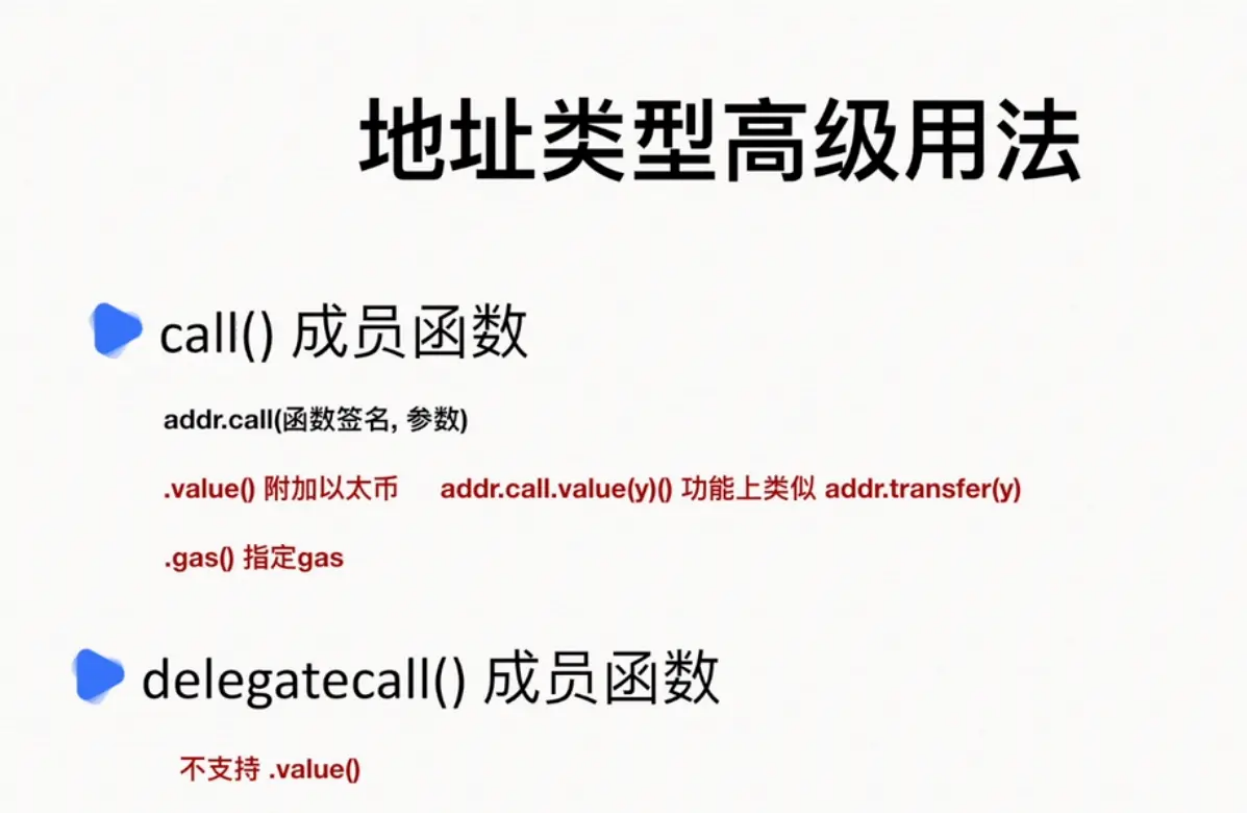
```
address(nameReg).call.gas(1000000).value(1 ether)(abi.encodeWithSignature("register(string)", "MyName"))
```

以类似的方式， `delegatecall` 可以使用该函数：不同之处在于仅使用给定地址的代码，所有其他方面（存储，平衡.....）取自当前合同。目的 `delegatecall` 是使用存储在另一个合同中的库代码。用户必须确保两个合同中的存储布局都适合使用委托调用。

注意

在宅基地之前，只有一种有限的变体 `callcode` 可用，不能提供对原始 `msg.sender` 和 `msg.value` 价值的访问。此功能已在0.5.0版中删除。

因为拜占庭 `staticcall` 也可以使用。这基本上是相同的 `call` ，但如果被调用的函数以任何方式修改状态，它将恢复。



12. 20 **单位**

1）Ether

单位关键字有wei, gwei, finney, szabo, ether，换算格式如下：

- 1 ether = 1 * 10^18 wei
- 1 ether = 1 * 10^9 gwei
- 1 ether = 1 * 10^6 szabo
- 1 ether = 1* 10^3 finney

2）Time

单位关键字有seconds, minutes, hours, days, weeks, years，换算格式如下：

- 1 == 1 seconds
- 1 minutes == 60 seconds
- 1 hours == 60 minutes
- 1 days == 24 hours

- 1 weeks == 7 days
- 1 years == 365 days

如果你需要进行使用这些单位进行日期计算，需要特别小心，因为不是每年都是365天，且并不是每天都有24小时，因为还有闰秒。由于无法预测闰秒，必须由外部的oracle来更新从而得到一个精确的日历库（内部实现一个日期库也是消耗gas的）。

```
pragma solidity 0.4.20;
/**
 * 对 Time 单位进行测试
 */
contract testTime {

    // 定义全局变量
    uint time;

    function testTime() public{
      time = 100000000;
    }

    function fSeconds() public view returns(uint){
      return time + 1 seconds; //100000001
    }

    function fMinutes() public view returns(uint){
      return time + 1 minutes; //100000060
    }

    function fHours() public view returns(uint){
      return time + 1 hours; //100003600
    }

    function fWeeks() public view returns(uint){
      return time + 1 weeks; //100604800
    }

    function fYears() public view returns(uint){
      return time + 1 years; //131536000
    }
}
```

12. 21 **constant**

constant、view 和 pure 三个修饰词有什么区别和联系？简单来说，在Solidity v4.17之前，只有constant，后来有人嫌constant这个词本身代表变量中的常量，不适合用来修饰函数，所以将constant拆成了view和pure。view的作用和constant一模一样，可以读取状态变量但是不能改；pure则更为严格，pure修饰的函数不能改也不能读状态变量，否则编译通不过。

constant、view、pure 三个函数修饰词的作用是告诉编译器，函数不改变/不读取状态变量，这样函数执行就可以不消耗gas了，因为不需要矿工来验证。所以用好这几个关键词很重要，不言而喻，省gas就是省钱！

1）**为什么使用 constant**

也就是说，当执行函数时不会去修改区块中的数据状态时，那么这个函数就可以被声明成 constant 的，比如说 getter 类的方法。

同时，当函数被 constant 修饰时也是提示 web3js（或其他json-rpc客户端）调用此方法时要使用 eth_call 函数而不是 eth_sendTransaction。

constant 需要编程时明确指定，即使状态不会改变，编译器也不会自动添加。一般情况下调用 constant 声明的方法不需要花费 gas，如果未使用 constant 修饰的函数在调用的过程中可能会生成一笔交易并且产生交易费用。

2）**constant与view的区别**

constant 是 view的别名，不过 constant 在 0.5.0 以上版本中已经被去掉。这也是我们在写智能合约时需要注意的事项。目前网络上的示例基本上还都采用constant来进行修饰。

那么，文档中已经描述这两者是相同的，那么为什么要用view来替代constant呢？基本上原因是这样的，使用constant有一定的误导性，比如用constant修饰的方法返回的结果并不是常量，而是根据一定的情况有所变化。而且，用constant来修饰并不是那么细致入微。因此，引入了更有意义和更有用的view和pure来代替constant。

3） **替换前后的变化**

替换当前：

- constant修饰的函数不应该修改状态；
- constant 修饰的变量（类中的变量而不是方法）每次调用时都会被重新计算；

替换之后：

- 关键词view用来修饰函数，替换掉 constant。调用view修饰的函数不能改变未来与任何合约交互的行为。这意味着被修饰的函数不能使用SSTORE，不能发送或接收以太币，只能调用其他view或pure修饰的函数。
- 关键字pure用来修饰函数，是在view修饰函数上附加了一些限制，pure修饰的函数不能改也不能读状态变量，否则编译通不过。这意味着它不能使用SSTORE，SLOAD，不能发送或接收以太币，不能使用msg或block而只能调用其他pure函数。
- 关键字 constant 针对函数无效。
- 任何用 constant 修饰的变量将不能被修改。

12. 22 **this 和 msg.sender 的用法**

Solidity 中 this 代表合约对象本身，可以通过 `address(this)` 获取合约地址。合约地址与合约创建者地址、合约调用者地址并不相同。

Solidity 中 `msg.sender` 代表合约调用者地址。一个智能合约既可以被合约创建者调用，也可以被其它人调用。

合约创建者，即合约拥有者，也就是指合约部署者，它的地址可以在合约的 `constructor()` 中，通过 `msg.sender` 获得，因为合约在部署的时候会首先调用 `constructor()`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SolidityTest {
    address public owner;

    event log(address);

    constructor()  {
        owner = msg.sender;
        emit log(msg.sender);
        emit log(address(this));
    }
}
```

owner 被赋值为合约部署者的地址。

log(msg.sender) 在日志中输出了合约部署者的地址。

log(address(this)) 在日志中输出了合约地址。

12. 23 address 地址

以太坊中的地址 address 的长度为20字节，一字节等于8位，一共160位，所以 address 也可以用 uint160 来声明。

以太坊钱包地址是以 16 进制的形式呈现，我们知道一个十六进制的数字等于4 bit，所以以太坊钱包地址的十六进制表示法的长度为 160 ／ 4 = 40，例如：钱包地址 DF12793CA392ff748adF013D146f8dA73df6E304的长度为40。

```
0xDF12793CA392ff748adF013D146f8dA73df6E304  //以太坊钱包地址
```

进制转换

```
pragma solidity ^0.4.4;

contract  test {
  address _owner;
  uint160 _ownerUint;

  function test() {
```

```
    _owner = 0xDF12793CA392ff748adF013D146f8dA73df6E304;
    _ownerUint = 1273516916528256943268872459582090959717186069252;
  }

  function owner() constant returns (address) {
    return _owner;
  }

  function ownerUint160() constant returns(uint160){
      //转换10进制 1273516916528256943268872459582090959717186069252
     return uint160(_owner);
  }

  function ownerUintToAddress() constant returns (address) {
    return address(_ownerUint);
  }
}
```

- 合约拥有者

msg.sender就是当前调用方法时的发起人，一个合约部署后，通过钱包地址操作合约的人很多，但是如何正确判断谁是合约的拥有者，判断方式很简单，就是第一次部署合约时，谁出的gas，谁就对合约具有拥有权。

```
pragma solidity ^0.4.4;

contract Test {

    address public _owner;

    uint public _number;

    function Test() {
        _owner = msg.sender;
        _number = 100;
    }

    function msgSenderAddress() constant returns (address) {
        return msg.sender;
    }

    function setNumberAdd1() {
        _number = _number + 5;
    }

    function setNumberAdd2() {
        if (_owner == msg.sender) {
            _number = _number + 10;
        }
    }
}
```

• 合约地址

```
pragma solidity ^0.4.4;

contract Test {

    address public _owner;

    uint public _number;

    function Test() {
        _owner = msg.sender;
        _number = 100;
    }

    function msgSenderAddress() constant returns (address) {
        return msg.sender;
    }

    function setNumberAdd1() {
        _number = _number + 5;
    }

    function setNumberAdd2() {
        if (_owner == msg.sender) {
            _number = _number + 10;
        }
    }

    function returnContractAddress() constant returns (address) {
        return this;
    }

}
```

一个合约部署后，会有一个合约地址，这个合约地址就代表合约自己。

this在合约中到底是msg.sender还是合约地址，由上图不难看出，this即是当前合约地址。

支持的运算符

```
pragma solidity ^0.4.4;

contract Test {

    address address1;
    address address2;

    // <=, <, ==, !=, >=和>

    function Test() {
        address1 = 0xF055775eBD516e7419ae486C1d50C682d4170645;
        address2 = 0xEAEC9B481c60e8cDc3cdF2D342082C349E5D6318;
    }


    // <=
    function test1() constant returns (bool) {
        return address1 <= address2;
    }
```

```
    // <
    function test2() constant returns (bool) {
        return address1 < address2;
    }

    //  !=
    function test3() constant returns (bool) {
        return address1 != address2;
    }

    // >=
    function test4() constant returns (bool) {
        return address1 >= address2;
    }

    // >
    function test5() constant returns (bool) {
        return address1 > address2;
    }
}
```

成员变量和函数

1）balance

如果我们需要查看一个地址的余额，我们可以使用balance属性进行查看。

```
pragma solidity ^0.4.4;

contract addressBalance{

    function getBalance(address addr) constant returns (uint){
        return addr.balance;
    }

}
```

2）this 查看当前合约地址余额

```
pragma solidity ^0.4.4;

contract addressBalance{

    function getBalance() constant returns (uint){
        return this.balance;
    }

    function getContractAddrees() constant returns (address){
        return this;
    }

    function getBalance(address addr) constant returns (uint){
        return addr.balance;
    }

}
```

3） transfer

transfer：从合约发起方向某个地址转入以太币(单位是wei)，地址无效或者合约发起方余额不足时，代码将抛出异常并停止转账。

```
pragma solidity ^0.4.4;

contract PayableKeyword{


    // 从合约发起方向 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c 地址转入 msg.value 个以太币，单位是 wei
    function deposit() payable{

        address Account2 = 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c;
        Account2.transfer(msg.value);
    }


    // 读取 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c 地址的余额
    function getAccount2Balance() constant returns (uint) {

        address Account2 = 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c;

        return Account2.balance;
    }

    // 读取合约发起方的余额
    function getOwnerBalance() constant returns (uint) {

        address Owner = msg.sender;
        return Owner.balance;
    }

}
```

4） send

send相对transfer方法较底层，不过使用方法和transfer相同，都是从合约发起方向某个地址转入以太币(单位是wei)，地址无效或者合约发起方余额不足时，send不会抛出异常，而是直接返回false

```
pragma solidity ^0.4.4;

contract PayableKeyword{


    function deposit() payable returns (bool){

        address Account2 = 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c;
        return Account2.send(msg.value);
    }


    function getAccount2Balance() constant returns (uint) {

        address Account2 = 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c;
```

```
        return Account2.balance;
    }


    function getOwnerBalance() constant returns (uint) {

        address Owner = msg.sender;
        return Owner.balance;
    }
```

send()方法执行时有一些风险

- 调用递归深度不能超1024。
- 如果gas不够，执行会失败。
- 所以使用这个方法要检查成功与否。
- transfer相对send较安全

12. 24 **众筹智能合约**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.0;

// 众筹智能合约
contract Crowdfunding{
    // 出资人
    struct Donor {
        address addr; //出资人地址
        uint amount; //出资人金额
    }

    // 募资人
    struct Donee {
        address addr; //募资人地址
        uint goal; //募资目标金额
        uint amount; //已筹集金额
        uint donorCount; //捐赠者数量
        bool status; //项目有效性：true 有效 false 无效
        mapping(uint => Donor) donorMap; //出资人字典
    }

    uint doneeCount;// 募资人数量
    mapping(uint => Donee) doneeMap; //募资人字典

    address payable owner; //合约拥有者

    // 构造函数
    constructor(){
        // 设置合约拥有者
        owner = msg.sender;
    }

    // 销毁合约
    function destroy() public onlyOwner{
```

```
        selfdestruct(owner);
    }

    // 校验合约拥有者
    modifier onlyOwner() {
        // 判断函数调用者是否为owner
        require(msg.sender == owner);
        _;
    }

    // 校验募资项目ID合法性
    modifier validDonee(uint doneeID) {
        require(doneeID>0 && doneeID<=doneeCount);
        _;
    }

    // 设置募资人和募资金额
    function setDonee(address addr, uint goal) public onlyOwner{
        for(uint i=0;i<doneeCount;i++){
            Donee storage d = doneeMap[i+1];
            if(d.addr == addr){
                d.goal = goal;
                return;
            }
        }

        doneeCount++;
        Donee storage donee = doneeMap[doneeCount];
        donee.addr = addr;
        donee.goal = goal;
        donee.status = true;
    }

    // 出资人捐赠
    function donate(uint doneeID) public payable validDonee(doneeID){
        Donee storage donee = doneeMap[doneeID];
        require(donee.status);

        donee.donorCount++;
        donee.amount += msg.value;//出资人金额

        Donor storage donor = donee.donorMap[donee.donorCount];
        donor.addr = msg.sender;
        donor.amount = msg.value;
    }

    // 完成目标给募资人转账
    function transfer(uint doneeID) public payable onlyOwner validDonee(doneeID) {
        Donee storage donee = doneeMap[doneeID];
        if(donee.amount >= donee.goal){
            // 给募资人转账
            payable(donee.addr).transfer(donee.goal);
        } else {
            // 金额不足
            revert();
        }
    }

    // 合约转账到拥有者
    function withdraw() public payable onlyOwner{
        msg.sender.transfer(address(this).balance);
    }
```

```
    // 查询募资人数量
    function getDoneeCount() public view returns(uint) {
        return doneeCount;
    }

    // 获取募资人信息
    function getDonee(uint doneeID) public view returns(address doneeAddr,uint doneeGoal,uint doneeAmount){
        return (doneeMap[doneeID].addr,doneeMap[doneeID].goal,doneeMap[doneeID].amount);
    }

    // 获取合约余额
    function getBalance() public view returns(uint) {
        return address(this).balance;
    }

    // 设定项目状态是否有效
   function setStatus(uint doneeID, bool status) public onlyOwner {
        Donee storage donee = doneeMap[doneeID];
        donee.status = status;
    }

    // 获取项目状态
    function getStatus(uint doneeID) public view validDonee(doneeID) returns(bool) {
        Donee storage donee = doneeMap[doneeID];
        return donee.status;
    }

    fallback() external{
    }

    receive() payable external{
    }
}
```

12. 25 **interface 接口**

接口本意是物体之间连接的部位。例如：电脑的 usb 接口可以用来连接鼠标也可以连接U盘和硬盘。因此，使用标准的接口可以极大的拓展程序的功能。在 solidity 语言中，接口可以用来接受相同规则的合约，实现可更新的智能合约。

interface 类似于抽象合约，但它们不能实现任何功能。还有其他限制：

- 无法继承其他合约或接口。
- 所有声明的函数必须是 external 的。
- 无法定义构造函数。
- 无法定义变量。
- 无法定义结构。

1）**接口定义**

接口需要有interface关键字，并且内部只需要有函数的声明，不用实现。

只要某合约中有和词接口相同的函数声明，就可以被此合约所接受。

```
interface 接口名{
    函数声明;
}
```

2）**接口使用**

在下面的例子中，定义了cat合约以及dog合约。他们都有eat方法.以此他们都可以被上面的animalEat接口所接收。

```
contract cat{
    string name;
    function eat() public returns(string){
        return "cat eat fish";
    }

    function sleep() public returns(string){
         return "sleep";
    }
}

contract dog{
    string name;
    function eat() public returns(string){
        return "dog miss you";
    }

    function swim() public returns(string){
         return "sleep";
    }
}

interface animalEat{
      function eat() public returns(string);
}

contract animal{
    function test(address _addr) returns(string){
        animalEat generalEat = animalEat(_addr);
        return generalEat.eat();
    }
}
```

在合约 animal 中，调用函数 test，如果传递的是部署的 cat 的合约地址，那么我们在调用接口的 eat 方法时，实则调用了cat 合约的 eat 方法。 同理，如果传递的是部署的 dog 的合约地址，那么我们在调用接口的 eat 方法时，实则调用了 dog 合约的 eat 方法。

3）**Uniswap使用示例**

```
interface UniswapV2Factory {
    function getPair(address tokenA, address tokenB)
        external
```

```
        view
        returns (address pair);
}

interface UniswapV2Pair {
    function getReserves()
        external
        view
        returns (
            uint112 reserve0,
            uint112 reserve1,
            uint32 blockTimestampLast
        );
}

contract UniswapExample {
    address private factory = 0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f;
    address private dai = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
    address private weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

    function getTokenReserves() external view returns (uint, uint) {
        address pair = UniswapV2Factory(factory).getPair(dai, weth);
        (uint reserve0, uint reserve1, ) = UniswapV2Pair(pair).getReserves();
        return (reserve0, reserve1);
    }
}
```

12. 26 优化智能合约gas使用的模式

在以太坊区块链上，Gas被用来补偿矿工为智能合约的存储与执行所提供的算力。目前以太坊的利用在逐渐增长，而交易手续费成本也水涨船高 —— 现在每天的gas成本已经高达数百万美元。随着以太坊生态系统的扩大，Solidity智能合约开发者也需要关注gas利用的优化问题了。本文将介绍在使用Solidity开发以太坊智能合约时常用的一些Gas优化模式。

1）使用短路模式排序Solidity操作

短路（short-circuiting）是一种使用或/与逻辑来排序不同成本操作的solidity合约开发模式，它将低gas成本的操作放在前面，高gas成本的操作放在后面，这样如果前面的低成本操作可行，就可以跳过（短路）后面的高成本以太坊虚拟机操作了

```
// f(x) 是低gas成本的操作
// g(y) 是高gas成本的操作

// 按如下排序不同gas成本的操作
f(x) || g(y)
f(x) && g(y)
```

2）删减不必要的Solidity库

在开发Solidity智能合约时，我们引入的库通常只需要用到其中的部分功能，这意味着其中可能会包含大量对于你的智能合约而言其实是冗余的solidity代码。如果可以在你自己的合约里安全有效地实现所依赖的库功能，那么就能够达到优化solidity合约的gas利用的目的。

原本如下：

```
import './SafeMath.sol' as SafeMath;

contract SafeAddition {
  function safeAdd(uint a, uint b) public pure returns(uint) {
    return SafeMath.add(a, b);
  }
}
```

优化如下：

```
contract SafeAddition {
  function safeAdd(uint a, uint b) public pure returns(uint) {
    uint c = a + b;
    require(c >= a, "Addition overflow");
    return c;
  }
}
```

3）**显式声明Solidity合约函数的可见性**

在Solidity合约开发种，显式声明函数的可见性不仅可以提高智能合约的安全性，同时也有利于优化合约执行的gas成本。例如，通过显式地标记函数为外部函数（External），可以强制将函数参数的存储位置设置为calldata，这会节约每次函数执行时所需的以太坊gas成本。

4）**使用正确的Solidity数据类型**

在Solidity中，有些数据类型要比另外一些数据类型的gas成本高。有必要了解可用数据类型的gas利用情况，以便根据你的需求选择效率最高的那种。下面是关于solidity数据类型gas消耗情况的一些规则：

- 在任何可以使用uint类型的情况下，不要使用string类型
- 存储uint256要比存储uint8的gas成本低，为什么？点击这里查看原文
- 当可以使用bytes类型时，不要在solidity合约种使用byte[]类型
- 如果bytes的长度有可以预计的上限，那么尽可能改用bytes1~bytes32这些具有固定长度的solidity类型
- bytes32所需的gas成本要低于string类型

5）**避免Solidity智能合约中的死代码**

死代码（Dead code）是指那些永远也不会执行的Solidity代码，例如那些执行条件永远也不可能满足的代码，就像下面的两个自相矛盾的条件判断里的Solidity代码块，消耗了以太坊gas资源但没有任何作用。

```
function deadCode(uint x) public pure {
  if(x < 1) {
    if(x > 2) {
```

```
      return x;
    }
  }
}
```

6）**避免使用不必要的条件判断**

有些条件断言的结果不需要Solidity代码的执行就可以了解，那么这样的条件判断就可以精简掉。例如下面的Solidity合约代码中的两级判断条件，最外层的判断是在浪费宝贵的以太坊gas资源

```
function opaquePredicate(uint x) public pure {
  if(x < 1) {
    if(x < 0) {
      return x;
    }
  }
}
```

7）**避免在循环中执行gas成本高的操作**

由于SLOAD和SSTORE操作码的成本高昂，因此管理storage变量的gas成本要远远高于内存变量，所以要避免在循环中操作storage变量。例如下面的solidity代码中，num变量是一个storage变量，那么未知循环次数的若干次操作，很可能会造成solidity开发者意料之外的以太坊gas消耗黑洞：

```
uint num = 0;

function expensiveLoop(uint x) public {
  for(uint i = 0; i < x; i++) {
    num += 1;
  }
}
```

解决上述反模式以太坊合约代码的方法，是创建一个solidity临时变量来代替上述全局变量参与循环，然后在循环结束后重新将临时变量的值赋给全局变量:

```
uint num = 0;

function lessExpensiveLoop(uint x) public {
  uint temp = num;
  for(uint i = 0; i < x; i++) {
    temp += 1;
  }
  num = temp;
}
```

8）**避免为可预测的结果使用Solidity循环**

如果一个循环计算的结果是无需编译执行Solidity代码就可以预测的，那么就不要使用循环，这可以可观地节省gas。例如下面的以太坊合约代码就可以直接设置num变量的值：

```
function constantOutcome() public pure returns(uint) {
  uint num = 0;
  for(uint i = 0; i < 100; i++) {  //其实就是num=100
    num += 1;
  }
  return num;
}
```

9）**循环合并模式**

有时候在Solidity智能合约中，你会发现两个循环的判断条件一致，那么在这种情况下就没有理由不合并它们。例如下面的以太坊合约代码：

```
function loopFusion(uint x, uint y) public pure returns(uint) {
  for(uint i = 0; i < 100; i++) {
    x += 1;
  }
  for(uint i = 0; i < 100; i++) {
    y += 1;
  }
  return x + y;
}
```

10）**避免循环中的重复计算**

如果循环中的某个Solidity表达式在每次迭代都产生同样的结果，那么就可以将其移出循环先行计算，从而节省掉循环中额外的gas成本。如果表达式中使用的变量是storage变量，这就更重要了。例如下面的智能合约代码中表达式a*b的值，并不需要每次迭代重新计算：

```
uint a = 4;
uint b = 5;
function repeatedComputations(uint x) public returns(uint) {
  uint sum = 0;
  for(uint i = 0; i <= x; i++) {
    sum = sum + a * b;
  }
}
```

12. 27 **常见编译错误**

1、报错：Expected token Semicolon got 'eth_compileSolidity' funtion setFunder(uint _u,uint _amount){

解决：funtion关键字错了，需要用function；

2、报错：Variable is declared as a storage pointer. Use an explicit "storage" keyword to silence this warning. Funder f = funders[_u]; ^------^

解决：Funder f，定义指针需要加关键字storage ；修改为Funder storage f = funders[_u];

3、报错：Invoking events without "emit" prefix is deprecated. e("newFunder",_add,_amount); ^-------------------------^

解决：调用事件需要在前面加上emit关键字，修改为emit e("newFunder",_add,_amount);

4、报错：No visibility specified. Defaulting to "public". function newFunder(address _add,uint _amount) returns (uint){ ^ (Relevant source part starts here and spans across multiple lines).

解决：定义函数必须加上public关键字，修改为function newFunder(address _add,uint _amount) public returns (uint){

5、报错："msg.gas" has been deprecated in favor of "gasleft()" uint public _gas = msg.gas; ^-----^

解决：msg.gas已经被gasleft()替换了。修改为uint public _gas = gasleft();

6、报错："throw" is deprecated in favour of "revert()", "require()" and "assert()". throw ;

解决：solidity已经不支持thorw了，需要使用require，用法require()

throw 写法：

if(msg.sender !=chairperson ||voters[_voter].voted){

throw ;

}

require写法：

require(msg.sender !=chairperson ||voters[_voter].voted);

7、报错：This declaration shadows an existing declaration. Voter delegate = voters[to]; ^----------^

解决：变量重复定义，变量名和函数名不能相同。

8、报错：error: Function state mutability can be restricted to pure

解决：以前版本是可以不指定类型internal pure(外部不可调用)，public pure(外部可调用)(如不指定表示函数为可变行，需要限制)

9、报错："sha3" has been deprecated in favour of "keccak256"

解决：sha3已经替换为keccak256

12. 28 调用合约

Solidity 支持一个合约调用另一个合约。两个合约既可以位于同一sol文件，也可以位于不同的两个sol文件。

Solidity 还能调用已经上链的其它合约。

1）调用内部合约

内部合约是指位于同一sol文件中的合约，它们不需要额外的声明就可以直接调用。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Hello {
    function echo() external pure returns(string memory){
       return "Hello World!";
   }
}

contract SolidityTest {
   function callHello(address addr) external pure returns(string memory){
      // 调用外部合约 Hello 的方法 echo
      return Hello(addr).echo();
   }

   // 另外一种写法
   function callHelloOr(Hello hello) external pure returns(string memory){
       // 调用外部合约 Hello 的方法 echo
      return hello.echo();
   }
}
```

2）**调用外部合约**

外部合约是指位于不同文件的外部合约，以及上链的合约。

调用外部合约有两种方法：通过接口方式调用 和 通过签名方式调用。

a. **通过接口方式调用**

通过接口方式调用合约，需要在**调用者**所在的文件中声明**被调用者**的接口。

被调用者合约 hello.sol：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract IHello {
    function echo() external pure returns(string memory){
       return "Hello World!";
   }
}
```

调用者合约 test.sol：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// 被调用者接口
interface IHello {
    // 被调用的方法
    function echo() external pure returns(string memory);
}

// 调用者合约
```

```
contract SoldityTest {
   function callHello(address addr) external pure returns(string memory){
      // 调用外部合约Hello的方法：echo
      return IHello(addr).echo();
   }
}
```

我们首先要部署 hello.sol 文件中的合约 Hello，得到它的地址，例如：0x78FD83768c7492aE537924c9658BE3D29D8ffFc1。

然后再部署合约 SoldityTest，调用 SoldityTest 的方法 callHello，传入参数 0x78FD83768c7492aE537924c9658BE3D29D8ffFc1 ，就会输出调用结果："Hello World!"。

b. **通过签名方式调用**

通过签名方式调用合约，只需要传入**被调用者**的地址和调用方法声明

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// 调用者合约
contract SoldityTest {
   function callHello(address addr) external returns(string memory){
       // 调用合约
       (bool success,bytes memory data) = addr.call(abi.encodeWithSignature("echo()"));
       if(success){
           return abi.decode(data,(string));
        } else {
            return "error";
        }
   }
}
```

另一种方法：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// 调用者合约
contract SoldityTest {
   function callHello(address addr) external view returns(string memory){
       // 编码被调用者的方法签名
       bytes4 methodId = bytes4(keccak256("echo()"));

       // 调用合约
       (bool success,bytes memory data) = addr.staticcall(abi.encodeWithSelector(methodId));
       if(success){
           return abi.decode(data,(string));
        } else {
            return "error";
        }
   }
}
```

签名方式调用，发送Eth

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// 调用者合约
contract SoldityTest {
   function callHello(address addr) external returns(string memory){
       // 调用合约
       (bool success,bytes memory data) = addr.call{value:1000}(abi.encodeWithSignature("echo()"));
       if(success){
           return abi.decode(data,(string));
        } else {
            return "error";
        }
   }
}
```

12. 29 支付Eth payable

使用 payable 标记的 Solidity 函数可以用于发送和接收 Eth。payable 意味着在调用这个函数的消息中可以附带 Eth。

使用 payable 标记的 Solidity 地址变量，允许发送和接收 Eth。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Payable {
    // owner 可用于收费 eth
    address payable public owner;

    constructor() {
       // msg.sender 默认不能收发 eth，需转换
        owner = payable(msg.sender);
    }

    function deposit() external payable{
    }
}
```

payable 地址变量可以通过 balance 属性，来查看余额。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Payable {
   function deposit() external payable{
   }

   function getBalance() external view returns(uint) {
      return address(this).balance;
```

```
    }
}
```

12. 30 回退函数 fallback

solidity 回退函数 fallback 没有参数、没有返回值。

solidity 回退函数在两种情况被调用：

- 向合约转账，发送 Eth，就会执行Fallback函数
- 如果请求的合约方法不存在，就会执行Fallback函数

向合约转账:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Fallback {
   event eventFallback(string);

   fallback() external payable {
      emit eventFallback("fallbak");
   }

   //  查看合约账户余额
   function getBalance() external view returns(uint) {
      return address(this).balance;
   }
}
```

请求的合约方法不存在:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Fallback {
   event eventFallback(string);

   fallback() external payable {
      emit eventFallback("fallbak");
   }
}

contract SolditTest {
   // 外部合约
   address private fb;

   constructor(address addr) {
      fb = addr;
   }

   function callFallback() external view returns(string memory) {
     // 调用合约 Fallback 不存在的方法 echo()
```

```
      bytes4 methodId = bytes4(keccak256("echo()"));

      // 调用合约
      (bool success,bytes memory data) = fb.staticcall(abi.encodeWithSelector(methodId));
      if(success){
         return abi.decode(data,(string));
      } else {
         return "error";
      }
   }
}
```

我们先部署合约 Fallback，再使用 Fallback 的地址来部署 SolidityTest，调用 Fallback 方法 echo 方法，就会触发 Fallback 的 fallback 方法。

12. 31 接收函数 receive

solidity 接收函数 receive 没有参数、没有返回值。

solidity 向合约转账，发送 Eth，就会执行 receive 函数。

如果没有定义接收函数 receive，就会执行 fallback 函数。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Fallback {
   event eventFallback(string);

   fallback() external payable {
      emit eventFallback("fallbak");
   }

   receive() external payable {
      emit eventFallback("receive");
   }
   //  查看合约账户余额
   function getBalance() external view returns(uint) {
      return address(this).balance;
   }
}
```

receive 和 fallback 调用流程

向一个合约发送 Eth，何时调用 receive 或者 fallback 呢？下面是两者的调用流程。

```
            发送 Eth
               |
           msg.data 是否为空
             /    \
            是      否
           /         \
 是否定义了receive   fallback
       /  \
```

```
       是   否
      /      \
 receive     fallback
```

12. 32 钱包合约

我们可以使用 Solidity 编写智能合约做一个钱包。

钱包合约的功能包括：存币、取币和查看余额，而且只能由合约发布者才拥有权限。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Wallet {
    address payable public owner;

    modifier onlyOwner() {
        require(msg.sender == owner, "sender is not owner");
        _;
    }

    constructor() {
        owner = payable(msg.sender);
    }

    // 允许存币
    receive() external payable {}

    // 取币
    function withdraw(uint amount) external payable onlyOwner{
        // mgs.sender 与 owner 相等，不使用 owner 可以节省 gas
        payable(msg.sender).transfer(amount);
    }

    // 获取余额
    function getBalance() external view onlyOwner returns(uint) {
        return payable(this).balance;
    }
}
```

12. 33 发送Eth

Solidity 在智能合约中有三种方式发送 Eth。

transfer：使用 transfer 发送 Eth，会带有 2300 个gas，如果失败，就会 revert。

send：使用 send 发送 Eth，会带有 2300 个gas，并且返回一个 bool 值表示是否成功。

call：使用 call 发送 Eth，会发送所有剩余的 gas，并且返回表示是否成功 bool 值和 data 数据。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
contract SendEther {
    constructor() payable{}

    // 允许接收 Eth
    receive() external payable {}

    function transferEth(address payable _to) external payable {
        _to.transfer(100);
    }

    function sendEth(address payable _to) external payable {
        bool success = _to.send(100);
        require(success, "send failed");
    }

    function callEth(address payable _to) external payable {
        (bool success, ) = _to.call{value:100}("");
        require(success, "call failed");
    }
}

contract ReceiveEther {
    event log(uint amount, uint gas);

    // 允许接收 Eth
    receive() external payable {
        emit log(msg.value, gasleft());
    }
}
```

12. 34 自毁合约 selfdestruct

Solidity 自毁函数 selfdestruct 由以太坊智能合约提供，用于销毁区块链上的合约系统。

当合约执行自毁操作时，合约账户上剩余的以太币会强制发送给指定的目标，然后其存储和代码从状态中被移除。

所以，Solidity selfdestruct 做两件事。

- 它使合约变为无效，有效地删除该地址的字节码。
- 它把合约的所有资金强制发送到目标地址。

销毁合约示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Kill {
    function kill() external {
        selfdestruct(payable(msg.sender));
    }

    function test() external pure returns(uint) {
        return 100;
```

```
    }
}
```

部署后，先调用 test 函数，将会输出 100。然后调用 kill 函数，再次调用 test 函数，结果输出为 0，表明合约被销毁。

强制发送资金示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Kill {
    function kill() external {
        selfdestruct(payable(msg.sender));
    }

    function test() external pure returns(uint) {
        return 100;
    }
}
```

部署后，先调用 test 函数，将会输出 100。然后调用 kill 函数，再次调用 test 函数，结果输出为 0，表明合约被销毁。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Kill {
    constructor() payable {}

    function kill(address payable to) external {
        selfdestruct(to);
    }
}

contract Receive {
    function getBalance() external view returns(uint) {
        return address(this).balance;
    }
}
```

首先部署 Receive 合约，用于接收资金。再部署 Kill 合约，初始转入 Eth 123 wei，然后调用 kill 方法，并将 Receive 的地址作为参数。

我们通过 Receive 合约的 getBalance 方法查看余额，资金为 123 wei。

Receive 合约没有定义 fallback 和 receive 函数，正常情况下无法接收资金，但依然被 Receive 合约的 selfdestruct 方法强制转入了资金。

12. 35 哈希算法 keccak256

Solidity 的哈希算法使用一个内置函数 keccak256。

keccak256函数原型：

```
keccak256(bytes) returns (bytes32)
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Hash {
    function hash(string memory _text, uint _num, address _addr)
    public pure returns (bytes32) {
        return keccak256(abi.encodePacked(_text, _num, _addr));
    }
}
```

我们通常使用 abi.encodePacked 打包所有数据，然后再进行 keccak256 哈希。

但是，我们使用 abi.encodePacked 要非常小心，当将多个动态数据类型传递给 abi.encodePacked 时，可能会发生哈希冲突。

abi 编码函数除了 abi.encodePacked 外，还有函数 abi.encode。 abi.encodePacked 只是将参数转为 16 进制，再直接进行拼接，而 abi.encode 需要先进行补零 ，再进行转码拼接。

我们可以看一个例子：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Hash {
    function encode1() external pure returns(bytes memory){
       return abi.encodePacked("aa","bb");
    }

    function encode2() external pure returns(bytes memory){
       return abi.encodePacked("aab","b");
    }
}
```

两个方法返回的内容都是0x61616262，但两者的输入参数并不同。在这种情况下，您应该使用 abi.encode 代替。

或者使用 encodePacked，但是在两个参数之间再添加一个固定数字参数即可。

例如：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Hash {
    function encode1() external pure returns(bytes memory){
```

```
        return abi.encodePacked("aa",uint(1),"bb");
    }

    function encode2() external pure returns(bytes memory){
        return abi.encodePacked("aab",uint(1),"b");
    }
}
```

12. 36 工厂合约

Solidity 工厂合约是一种批量部署合约的方式。

通过一个工厂合约创建部署合约，并记录下所有部署合约的地址。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Account {
    address public bank;
    address public owner;

    constructor(address _owner) payable{
        bank = msg.sender;
        owner = _owner;
    }
}

contract Factory {
    Account[] public accounts;

    function createAccount(address owner) external payable{
        accounts.push(new Account{value:123}(owner));
        accounts.push(new Account{value:456}(owner));
    }
}
```

我们只需要部署 Factory 合约，运行 createAccount 方法，就会自动创建其它合约

12. 37 库合约 library

Solidity 智能合约中通用的代码可以提取到库 library，以提高代码的复用性和可维护性。

库 library 是智能合约的精简版，就像智能合约一样，位于区块链上，包含可以被其他合约使用的代码。

库 library 对比普通合约来说，有如下限制：

- 无状态变量
- 不能继承或被继承
- 不能接收 eth

使用库 library 的合约，可以将库合约视为隐式的父合约，当然它们不会显式的出现在继承关系中。也就是不用写 is 来继承，直接可以在合约中使用。

直接调用方法

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

library Math {
   function add(uint x, uint y) internal pure returns(uint){
      return x+y;
   }
}

contract MathTest {
   function test(uint x, uint y) external pure returns(uint){
      return Math.add(x, y);
   }
}
```

调用库合约函数的方式非常简单。如范例所示，library Math 有函数 add()，使用 Math.add 即可访问。

通常，库合约函数的可视范围为 internal，也就是对所有使用它的合约可见。

定义成 external 毫无意义，因为库合约函数只在内部使用，不独立运行。同样，定义成 private 也不行，因为其它合约无法使用。

using for 调用方法

使用库合约还有更方便的方法，那就是using for 指令。

例如：using A for B 用来将 A 库里定义的函数附着到类型 B。这些函数将会默认接收调用函数对象的实例作为第一个参数。这个语法类似 python中的 self 变量。

using A for *的效果是，库A中的函数被附着在做任意的类型上。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

library Math {
   function find(uint[] storage arr, uint val) internal view returns(uint){
      for (uint i=0; i<arr.length; i++) {
         if (arr[i] == val) {
            return i;
         }
      }
      revert("not found");
   }
}

contract MathTest {
   // 将 libray Math 附着到类型 uint[]
   using Math for uint[];
```

```
    uint[] arr = [1,2,3];

    function test() external view returns(uint){
        return arr.find(2);
    }
}
```

使用 using for 语法附着的数据类型，在使用的时候，可以直接用 <variable>.<method> 的形式调用，而且省略代表自己的第一个参数。

如范例所示，使用 using Math for uint[] 将 libray Math 附着到类型 uint[]，原来的写法 Math.find(arr, 2) 简写为 arr.find(2)。

另外，还可以使用 using Math for *，通配所有类型。

库 library 存在形式

库 library 有两种存在形式：

- 内嵌（embedded）：当库中所有的方法都是internal时，此时会将库代码内嵌在调用合约中，不会单独部署库合约；
- 链接（linked）：当库中含有external或public方法时，此时会单独将库合约部署，并在调用合约部署时链接link到库合约。

12. 38 权限控制合约

Solidity 合约中一般会有多种针对不同数据的操作，例如对于存证内容的增加、更新及查询，所以需要制定一套符合要求的权限控制。

如何对合约的权限进行划分？我们针对Solidity语言设计了一套通过地址标记的解决方案。

合约中划分了角色和账户两级权限，如下所示：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Access {
    mapping(bytes32 =>mapping(address =>bool)) public roles;

    // 0xf23ec0bb4210edd5cba85afd05127efcd2fc6a781bfed49188da1081670b22d8
    bytes32 constant private ADMIN = keccak256("admin");

    // 0xcb61ad33d3763aed2bc16c0f57ff251ac638d3d03ab7550adfd3e166c2e7adb6
    bytes32 constant private USER = keccak256("user");

    // 授权合约部署者 ADMIN 权限
    constructor() {
        _grantRole(ADMIN, msg.sender);
    }

    modifier onlyAdmin(address _account) {
        require(roles[ADMIN][_account], "not authorized");
```

```
        _;
    }

    function _grantRole(bytes32 _role, address _account) internal {
        roles[_role][_account] = true;
    }

    // 授权
    function grantRole(bytes32 _role, address _account) external onlyAdmin(_account) {
        _grantRole(_role, _account);
    }

    function _revokeRole(bytes32 _role, address _account) internal {
        roles[_role][_account] = false;
    }

    // 撤销授权
    function revokeRole(bytes32 _role, address _account) external onlyAdmin(_account) {
        _revokeRole(_role, _account);
    }
}
```

12. 39 验证签名

Solidity有一个`ecrecover`指令，可以根据消息`hash`和签名，返回签名者的地址：

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
```

根据恢复的签名地址，与验证地址对比，就可以验证签名。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Signature {
    function verify(address _signer, string memory _message, bytes memory _signature)
    external pure returns(bool) {
        bytes32 hash = getHash(_message);
        bytes32 ethSignedHash = getEthHash(hash);
        return recover(ethSignedHash,_signature) == _signer;
    }

    function getHash(string memory _message) public pure returns(bytes32) {
        return keccak256(abi.encodePacked(_message));
    }

    function getEthHash(bytes32 _hash) public pure returns(bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", _hash));
    }

    function recover(bytes32 ethSignedHash, bytes memory _signature) public pure returns(address) {
        (bytes32 r, bytes32 s, uint8 v) = _split(_signature);
        return ecrecover(ethSignedHash, v, r, s);
    }
```

```
    function _split(bytes memory _signature) internal pure returns(bytes32 r, bytes32 s, uint8 v){
        require(_signature.length == 65, "invalid signaure length");
        assembly {
            r := mload(add(_signature, 32))
            s := mload(add(_signature, 64))
            v := byte(0, mload(add(_signature, 96)))
        }
    }
}
```