作者：知乎-山上的石头:

这是我学习区块链入门时做的笔记（基于 0.8.7 版本），基本涵盖了编写合约所需常用知识，由于做智能合约安全方面的研究需要精通 Solidity 和 以太坊原理，因此做的笔记比较详实。

这些笔记基于阅读英文文档，参考中文文档和 stack overflow 以及相关教程，我根据学习者的接受新知识的顺序，对文章结构做了适当优化。

这篇文章既可以作为新手入门（因为啃英文文档并且搜索信息并不是容易的事情），也可以作为快捷的检索帮助文档（官方文档的翻译某些部分比较难以理解）。建议使用电脑端阅读。

初稿完成时，都还没学 C 语言，只是一知半解的边学边记。在大二上学期的寒假，我重新整理了一遍，修正了部分错误，将拗口的表述转化成习惯表述，补充了文档中缺少的范例，根据经验突出需要强调的注意事项，使得读者可以跟容易的学习。

本文共计接近7万字，如果觉得有帮助点赞关注呀，我将会继续写智能合约的攻击方式、以太坊虚拟机原理、字节码的深入探索等等，逐渐完善知识体系，并且会分享读论文时的前沿理论。

参考：

- Solidity 最新(0.8.0)中文文档
- Solidity - Solidity 0.8.12 documentation
- https://solidity-by-example.org

solidity基础

代码结构

直观理解代码结构，下面是铸造，生成代币的代码。

```
pragma solidity ^0.4;
contract Coin{
    //set the "address" type variable minter
    address public minter;
    /*convert "address"(for storing address or key )
    to the type of "uint" which is as subscrip of object balances*/
    mapping (address =>uint) public balances;
    // set an event so as to be seen publicly
    event Sent(address from,address to,uint amount);
    //constructor only run once when creating contract,unable to invoke
    //"msg" is the address of creator."msg.sender"  is
    constructor()public{
        minter=msg.sender;
    }
    //铸币
    //can only be called by creator
    function mint(address receiver,uint amount)public{
        require(msg.sender ==minter);
        balances[receiver]+=amount;
    }
    //转账
    function send(address receiver,uint amount)public{
        require(balances[msg.sender]>= amount);
        balances[msg.sender]-=amount;
        balances[receiver]+=amount;
```

```
        emit Sent(msg.sender,receiver,amount);
    }

}
```

版本标识

`pragma`

版本标识，是pragmatic information的简称，用于启动编译器检查，避免因为solidity更新后造成的不兼容和语法变动的错误。**只对本文件有效，如果导入其他文件，版本标识不会被导入，而是采用工作的文件自身的版本标识**

```
pragma solidity ^0.5.2;
```

这里`^`表示从0.5.2到0.6（不含）的版本

导入其他文件

`import "filename";` 这种导入方式会把导入文件的所有全局符号都导入到工作文件的全局作用域，会污染命名空间，不建议这么使用。

```
import * as symbolName from "filename";
//等价于
import "filename" as symbolName;
```

这样所有的全局符号都以`symbolName.symbol`的格式提供。

我们还可以设置别名，别名和重定义的符号名，都可以表示导入的文件里的全局符号。

```
import {symbol1 as alias, symbol2} from "filename";
```

支持从网络中导入，如：`import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.3/contracts/cryptography/ECDSA.sol";`

路径

路径的形式和Linux下的完全一致，但是要避免使用`..`。我们可以引入指定路径的文件，如`import "./filename" as simbolName`，是当前目录下的文件。引用的文件除了本地文件，也可以是网络资源。

实际solc编译器使用的时候可以指定路径的重映射，编译器可以从重映射的位置读取文件。尤其是使用网络文件的时候 例如，可以使`github.com/ethereum/dapp-bin/library`会被重映射到`/usr/local/dapp-bin/library`,格式如下。

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

更具体地会在solc编译器地部分说明。而truffle框架和remix就相对智能，可以通过网络获取文件。

注释

单行注释`//`,多行注释`/*.....*/`

一种natspec注释，他是用`///`或者`/**.....*/`，它里面可以使用Doxygen样式来给出相关地信息。

Doxygen样式地注释可以使特殊地注释形式变得可识别，方便读取和自动提取信息。主要有

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

/** @title Shape calculator.
 * @file（文件名）
 * @author John Doe <jdoe@example.com>（作者）
 * @version 1.0 （版本）
 * @details （细节）
 * @date （年-月-日）
 * @license （版权协议）
 * @brief （类的简单概述）
 * @section LICENSE（这一段的主要内容）
 * @param  Description of method's or function's input parameter（形式参数说明）
 * @return Description of the return value（返回说明）
 * @retval （返回值说明）
 * @attention（注意）
 * @warning（警告）
 * @var（变量声明）
 * @bug（代码缺陷）
 * @exception（异常）
 */
contract ShapeCalculator {
    /// @dev Calculates a rectangle's surface and perimeter.
    /// @param w Width of the rectangle.
    /// @param h Height of the rectangle.
    /// @return s The calculated surface.
    /// @return p The calculated perimeter.
    function rectangle(uint w, uint h) public pure returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

特别地，可以使用 `pragma abicoder v1` 或者 `pragma abicoder v1` 指定 ABI 的编码器和解码器版本，一般而言 0.8.0 以后，默认使用 v2 版本。

全局变量

状态变量是永久地存储在合约存储中的值，它具有数据的类型，也有可见性的属性。**在函数外的都是`storage`全局变量。**

```
pragma solidity >=0.4.0 <0.9.0;

contract TinyStorage {
    uint storedXlbData; // 状态变量
    // ...
}
```

函数

函数是代码的可执行单元。函数通常在合约内部定义，但也可以在合约外定义。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.7.0 <0.9.0;

contract TinyAuction {
    function Mybid() public payable { // 定义函数
        // ...
    }
}

// Helper function defined outside of a contract
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

函数调用可发生在合约内部或外部，且函数有严格的可见性限制，对于谁可以调用它有着明确的规定（可见性和 getter 函数）。

函数的返回值可以是元组，接收时需要一一对应。

函数修饰

函数修饰符用来修饰函数，比如添加函数执行前必须的先决条件。这样可以方便地实现代码复用。

```
contract Owner {
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

函数体会插入在修饰函数的下划线 `_` 的位置。所以只有当修饰条件满足之后才能执行这个函数，否则报错。

注意下面的用法。实际上常常会被继承，作为模块复用。

可以看到，使用的格式

`function funcName(params) 可见性修饰 函数属性修饰 函数修饰器 returns(params)`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8;

contract Test{
    uint public a;
    uint public b;
    function set(uint _a,uint _b) public{
        a=_a;
```

```
        b=_b;
    }
    modifier Func(uint _a)
    {
        require(a>_a,"error:a is so small.");
        _;
    }
    function f(uint _a) public view Func(_a) returns(uint) {
        return _a;
    }
}
```

事件

事件是能方便地调用以太坊虚拟机日志功能的接口，分为设置事件和触发事件。

设置事件只需要 `event 事件名(params)` 。

触发事件 `emit 事件名(实参)`，注意触发事件和设置事件的参数类型需要匹配。

```
pragma solidity >=0.4.21 <0.9.0;
contract TinyAuction {
    event HighestBidIncreased(address bidder, uint amount); // 事件

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // 触发事件
    }
}
```

合约

合约的构造函数至多一个，只在部署执行一次。

创建合约的方式可以是：Remix 这样的IDE、合约创建合约、用web3.js API.

部署的在区块链上的代码包括了所有可调用的函数或者是被其他函数调用的函数，但是不包括构造函数代码和只被构造函数调用的内部函数的代码。

构造函数的参数的ABI编码在合约的代码之后传递，web3.js可以略过这个细节。

支持合约类型和地址类型的强制转换。

函数和变量的可见性

可见性标识符在类型标识的后面。

`external`: 外部函数作为合约接口的一部分，可以被交易或者其他合约调用。 外部函数 `f` 不能以内部调用的方式调用（即 `f` 不起作用，但 `this.f()` 可以）。

`public`: public 函数是合约接口的一部分，可以在内部或通过消息调用。对于 public 状态变量， 会自动生成一个 getter 函数。

`internal` : 只能在当前合约内部或它派生合约中访问，不使用 `this` 调用。

`private`: private 函数和状态变量仅在当前定义它们的合约中使用，并且不能被派生合约使用（如继承）。

注意：区块链所有信息都是透明的，这里的可见性只是针对其他合约或者调用者的是否有权限，访问或者修改状态。

getter函数：`public` 的状态变量会自动生成一个 getter 函数，内部调用时相当于状态变量，外部调用时相当于一个函数。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // internal access
        return this.data(); // external access
    }
}
```

如果这个 `public` 的全局变量是一个数组，那么 getter 函数就只能通过下标访问单个元素，但是结构体中的数组或者是映射不能够返回。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

等效为

```
function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}
```

函数修饰器

函数修饰器会在函数执行前见擦汗条件，只有标记为`virtual`的情况下，才会被继承的合约覆盖。使用方法看下面的例子。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.7.0 <0.9.0;

contract owned {
```

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

    address owner;

    // 函数修饰器通过继承在派生合约中起作用。
    // 函数体会被插入到特殊符号 _; 的位置。
       modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract destructible is owned {
    //调用格式是在 可见性修饰符（或者view(payable)权限修饰符） 之后，returns之前
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // 修改器可以接收参数：
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, destructible {
    mapping (address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) { price = initialPrice; }

    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }
```

```
    // 这个函数受互斥量保护，这意味着 `msg.sender.call` 中的重入调用不能再次调用 `f`。
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

函数修饰器只能在当前合约或者是继承的合约中使用。库合约内的函数修饰器只能在库合约中定义及使用。

如果一个函数中有许多修饰器，写法上以空格隔开，执行时依次执行：首先进入第一个函数修饰器，然后一直执行到 `_;` 接着跳转回函数体，进入第二个修饰器，以此类推。到达最后一层时，一次返回到上一层修饰器的 `_;` 后。

修饰器不能够隐式地访问或者修改函数的变量，也不能够给函数提供返回值，只有规定的给修饰器的传入的参数才能够被修饰器使用。

显式地在修饰器中使用 `return` 不会影响函数地返回值，但是可能提前结束，就不会执行 `_;` 处地函数体了。修饰器和函数中的 `return` 都只会跳出当前的代码块，进入上一层的堆栈。

`_` 可以在修饰器中多次出现，每一处都会执行函数体（注意包括函数地其他修饰器）。

修饰器的参数可以是任意表达式，函数中可见的函数外的变量，在修饰器中都是可见的。但是修饰器中的变量对函数不可见。

构造函数

如果没有构造函数，就等同于有默认的构造函数 `constructor() {}`.

在继承中，构造函数有两种写法，一种是继承时直接给参数，形如 `is Base(7)`；另外一种是在子合约的构造函数中定义，这很适用于依赖子合约状态给父合约的构造函数赋值的情况。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint _x) { x = _x; }
}

contract Derived1 is Base(7) {
    constructor() {}
}

contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) {}
}
```

常量和不可变量

全局变量如果有 `constant` 或者 `immutable` 标识，表示他们在合约创建后不可改变（但是可以在创建时可以使用使用 `constructor` 修饰。他们的区别在于：

- `constant` 的值必须是全局变量，且声明时就要确定，且不可在构造函数中修改，因为它是**在编译时就确定且固定的。**而且在构造函数中，给 `constant` 赋值的表达式必须是返回固定的值，不能是运行时才确定的值。
- `immutable` 既可以在全局变量声明时确定（此后不可用构造函数修改），也可以在构造函数中确定（但只能赋值一次），因为**在构建时才确定并且固定的。**创建 `immutable` 变量发生在返回合约的 `creation code` 之前，编译器会发生值替换，修改合约的 `runtime code` ，这会造成区块链上实际存储的代码和 `runtime code` 有些差异。

在编译时，编译器不会给这些变量留储存位置，而是把常量和不可变量当作常量表达式，因此相比于常规的全局变量，消耗的gas少得多。

`constant` 的常量将会把赋值给它的表达式复制到所有访问它的位置，然后再进行求值的运算，类似于 C 语言的 `#define a (7*5)`。`immutable` 的不变量则是将表达式的值复制到访问它的位置，但是占用固定的32个字节，类似于 `#define a (35)` 。因此，不可变量占用空间较多，而且实际计算表达式时会优化，`constant` 的常量可能更加省gas

只有值类型或者常量字符串 `string` 才支持 `constant` 和 `immutable` 的标识。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint _decimals, address _reference) {
        decimals = _decimals;
        // Assignments to immutables can even access the environment.
        maxBalance = _reference.balance;
    }

    function isBalanceTooHigh(address _other) public view returns (bool) {
        return _other.balance > maxBalance;
    }
}
```

函数

自由函数

函数既可以定义在合约内，也可以定义在合约外。

定义在合约外的函数叫做自由函数，一定是 `internal` 类型，就像一个内部函数库一样，会包含在所有调用他们的合约内，就像写在对应位置一样。但是自由函数不能直接访问全局变量和其他不在作用域下的函数（比如，需要通过地址引入合约，再使用合约内的函数）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.7.0 <0.9.0;

function sum(uint[] memory _arr) pure returns (uint s) {
    for (uint i = 0; i < _arr.length; i++)
        s += _arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory _arr) public {
        // This calls the free function internally.
        // The compiler will add its code to the contract.
        uint s = sum(_arr);
        require(s >= 10);
        found = true;
    }
}
```

参数和返回值

外部函数 不可以接受多维数组作为参数，除非原文件加入 `pragma abicoder v2;`，以启用启用ABI v2 版编码功能。 （注：在 0.7.0 之前是使用`pragma experimental ABIEncoderV2;`）

非内部函数无法返回多维动态数组、结构体、映射。如果添加 `pragma abicoder v2;` 启用 ABI V2 编码器，则是可以的返回更多类型，不过 `mapping` 仍然是受限的。

内部函数默认可以接受多维数组作为参数。

返回值的变量名可以出现，也可以省略。当变量名出现时，可以不写明`return`，但是如果和全局变量重名，就会局部覆盖。

`view` 函数

`view` 函数不能产生任何修改。由于操作码的原因， `view` 库函数不会在运行时阻止状态改变，不过编译时静态检查器会发现这个问题。

以下行为都视为修改状态：

1. 修改状态变量。
2. 触发事件。
3. 创建其它合约。
4. 使用 `selfdestruct`。
5. 通过调用发送以太币。
6. 调用任何没有标记为 `view` 或者 `pure` 的函数。
7. 使用低级调用。
8. 使用包含特定操作码的内联汇编。

注意： `constant` 之前是 `view` 的别名，在0.5.0之后移除。

注意： `getter`方法会自动标记为`view`。

注意：在0.5.0前， `view`函数仍然可能产生状态修改。

`pure`函数

`pure` 函数不会读取状态，也不会改变状态。但是由于EVM的更新，也可能读取状态，而且无法在虚拟机水平上强制不读取状态。

以下行为视为读取状态：

1. 读取状态变量。
2. 访问 `address(this).balance` 或者 `<address>.balance`。
3. 访问 `block`， `tx`， `msg` 中任意成员 （除 `msg.sig` 和 `msg.data` 之外）。
4. 调用任何未标记为 `pure` 的函数。
5. 使用包含某些操作码的内联汇编。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

在`try/catch`中的回滚，不会视作状态改变。

事件

事件是对EVM日志的简短总结，可以通过RPC接口监听。触发事件时，设置好的参数就会记录在区块链的交易日志中，永久的保存，但是合约本身是不可以访问这些日志的。可以通过带有日志的Merkle证明的合约，来检查日志是否存在于区块链上。由于合约中仅能访问最近的 256 个区块哈希，所以还需要提供区块头信息。

也可以对事件中至多三个参数附加 `indexed` 属性，他们就会成为 `topics` 数据结构的一部分（详细请查看 ABI 部分编码的方式）。一个`topic`只可以容纳32个字节，对于`indexed`的引用类型会把值的 Keccak-256 hash 储存在一个`topic`。 `topic` 允许通过过滤器来搜索事件，比如出发事件的合约地址。

没有 `indexed` 的参数就会被ABI编码后存储在日志。

如果没有使用 `anonymous` 标识符的话，事件的签名的哈希值就会是一个`topic`，如果使用了的话就无法通过除了触发它的合约地址之外的方式（如：事件的参数）去筛选事件。但是匿名事件在部署和调用时更节省gas，而且可以使用四个`index`(虽然没啥用了)。

```
pragma solidity  >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // 事件使用 emit 触发事件。
```

```
        // 我们可以过滤对 `Deposit` 的调用，从而用 Javascript API 来查明对这个函数的任何调用（甚至是深度嵌套调用）。
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

类型

Solidity的值传递和引用传递有自己的规则，通过不同的存储域决定，后面详述。

默认值：Solidity中不存在`undefined`或`null`，每种变量都有自己的默认值，一般是"零状态"。

[运算符优先级]

布尔类型

`bool`:常量值为`true`和`false`

整型

`int` / `uint` ：分别表示有符号和无符号的不同位数的整型变量。 支持关键字 `uint8` 到 `uint256` （无符号，从 8 位到 256 位）以及 `int8` 到 `int256`，以 `8` 位为步长递增。 `uint` 和 `int` 分别是 `uint256` 和 `int256` 的别名。

可以用`type(x).min` `type(x).max`来获取这个类型地最小值和最大值。

位运算

在二进制地补码上操作，特别的 `~int256（0）== int256（-1）`

移位:

左移则会截断最高位；右移操作数必须是无符号地整型，否则会编译错误。

- `x<<y`相当于`x*2**y`,(其实这里体现了`**`的优先级比较高)
- 如果`x>0`:`x>>y`相当于`x/2**y`
- 如果`x<0`:`x>>y`相当于`(x+1)/2**y - 1`(如果不是整数，则向下取整)（注意：0.5.0之前是向上取整)

加减乘除

在`0.8.0`之后加入了溢出检查，值超过上限或者下限则会回滚，我们可以使用`unchecked{}`来取消检查。在此之前需要使用`OpenZepplin SafeMath`库。

注意：`unchecked{}`不能替代码块的花括号，而且不支持嵌套，只对花括号内的语句有效，且对其中调用的函数无效，并且花括号内不能出现`_`。

除 0 或者模 0 会报错。`type(int).min / (-1)`是唯一的整除向上溢出的情况。

注意移位操作符造成的溢出并不会报错，需要额外注意溢出问题。

幂运算只适用于无符号的整型，有时为了减少gas消耗，编译器会建议用`x*x*x`来代替`x**3`。

定义`0**0=1`

定长浮点型

由于 EVM 只支持整数运算并且需要严格控制计算资源，因此浮点数的计算的实现有一定的挑战，采用了严格限制整数位数和小数位数的方式。

`fixed` / `ufixed`：表示各种大小的有符号和无符号的定长浮点型。 在关键字 `ufixedMxN` 和 `fixedMxN` 中，`M` 表示该类型占用的位数，`N` 表示可用的小数位数。 `M` 必须能整除 8，即 8 到 256 位。 `N` 则可以是从 0 到 80 之间的任意数。 `ufixed` 和 `fixed` 分别是 `ufixed128x19` 和 `fixed128x19` 的别名。

注意：solidity还没有完全的支持定长浮点型，**只能声明，但是不可以给他赋值，也不能用它给其他变量赋值**，只可以下面那样。用的很少。

```
fixed8x4 a;
```

地址类型

这是比较特殊的类似，其他语言没有，实际上是储存字节。

地址类型有两种，

- `address`：保存一个20字节的值（以太坊地址的大小），**不支持作为转账地址。**
- `address payable`： **可参与转账的地址**，与 `address` 相同，不过有成员函数 `transfer` 和 `send` 。

注意：`address` 和 `address payable` 的区别是在 0.5.0 版本引入的***

地址成员：

地址类型有默认的成员，方便查看它的属性。

- `<address>.balance` 返回 `uint256`

  以 Wei 为单位的余额。

- `<address>.code` 返回 `bytes memory`

  地址上的字节码(可以为空)

- `<address>.codehash` (`bytes32`)

  地址上的字节码哈希

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

  向该地址发送数量为 amount 的 Wei，失败时抛出异常，并且会回滚。使用固定（不可调节）的 2300 gas 的矿工费。

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

  向该地址发送数量为 amount 的 Wei，失败时返回 `false`，发送 2300 gas 的矿工费用，不可调节。

  注意：`send` 安全等级比较低，他失败时（比如因为堆栈在1024或者gas不足）不会发生异常，因此往往要检查它的返回值，或者直接用`transfer`

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
pragma solidity ^0.8.3;
contract HelloWorld {
    string public greet = "Hello World!";
    address public myAddress=address(this);
    uint public myBalance = myAddress.balance;
    bytes public myCode = myAddress.code;
    bytes32 public myCodehash = myAddress.codehash;
    function getstr() public view returns (string memory){
        return greet;
    }
}
```

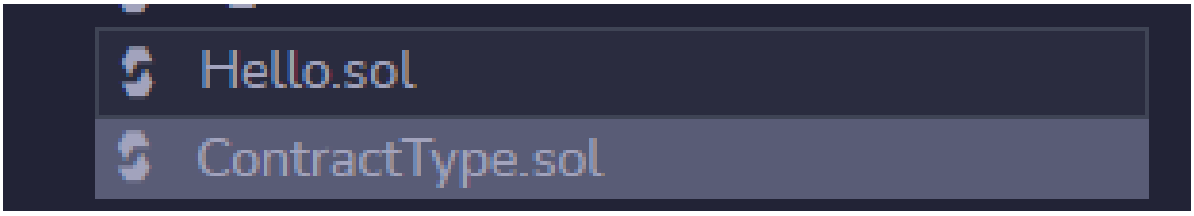
合约类型

每一个合约都有自己的类型，也可以用合约名定义其他变量，相当于创建了一个接口。

合约可以通过`address(x)`转换成`address`类型；只有可支付的合约（具有receive函数或者payable fallback函数），才可以使用`payable(address(x))`转换成`address payable`类型（0.5.0版本之后才区分`address`和`payable address`）

下面是示例用法：

新建两个文件放在同个文件夹下：




```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
contract HelloWorld {
    string public greet = "Hello World!";
    function getstr() public view returns (string memory){
        return greet;
    }
}
```

```
// SPDX-License-Identifier: GPL-3.0
import "./Hello.sol";
pragma solidity >=0.5.0 <0.9.0;
contract CallHello {
    HelloWorld public hello;
    constructor(address _addr){
        hello = HelloWorld(_addr);
    }
    function f()public view returns(string memory){
        return hello.getstr();
    }
    function g()public view returns(address){
        return address(hello);
```

```
    }
}
```

枚举类型

枚举类型至少需要一个成员，且不能多于256个成员。整数类型和枚举类型只能显式转化，不能隐式转化。整数转枚举时需要在枚举值的范围内，否则会引发`Panic error`。

可以使用`type(NameOfEnum).min`和`type(NameOfEnum).max`获取这个枚举类型的最小值和最大值

```
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }
```

函数类型

函数可以作为类型，可以被赋值，而且也可以作为其他函数的参数或者返回值，这一点和 Go 语言是一致的。

```
pragma solidity ^0.8.3;
contract A{

    function foo() external pure returns(uint){
    uint a =5;
    return a;
    }

    function () external returns(uint) f=this.foo;//注意，访问函数类型，一定要从 this 访问
   // f=this.foo;注意无法这样赋值，只能初始化时赋值
```

函数类型实际上包括一个 20 个字节的地址和 4 个字节的函数选择器，等价于`byte24`类型

函数类型的调用限制

有两种：

- 内部（*internal*） 函数类型，只能在当前合约内被调用（包括内部库函数和继承的函数），不能在合约的上下文外执行。调用内部函数时通过跳转到函数的标签片段实现。
- 外部（*external*） 函数类型，由一个地址和函数签名组成，在调用时会被视作`function`类型，函数的地址后面后紧跟函数标识符一起编码成`bytes24`类型。

下面是函数的类型表示：

```
function (<parameter types>) {internal|external} [pure|constant|view|payable]
[returns (<return types>)]
```

函数类型默认是内部函数，但是**在合约内定义的函数可见性必须明确声明。在合约内定义函数的位置时任意的，可以调用后面才定义的函数。**

函数类型的成员

public（或 external）函数都有下面的成员：

- `.address` 返回函数的合约地址。
- `.selector` 返回 ABI 函数选择器

注意在过去还有两个成员：`.gas(uint)` 和 `.value(uint)` 在0.6.2中弃用了，在 0.7.0 中移除了。 用 `{gas: ...}` 和 `{value: ...}` 代替。

```
pragma solidity ^0.8.3;
contract A{

    function foo() public pure returns(uint){
    uint a =5;
    return a;
    }

    function getAddr() public view returns(address){

    return this.foo.address;
    }

    function getSekector() public pure returns(bytes4){
    return this.foo.selector;
    }
}
```

内部函数的例子：(这里采用了库函数)

```
library ArrayUtils {
  // 内部函数可以在内部库函数中使用，
  // 因为它们会成为同一代码上下文的一部分
  function map(uint[] memory self, function (uint) pure returns (uint) f)
    internal
    pure
    returns (uint[] memory r)
  {
    r = new uint[](self.length);
    for (uint i = 0; i < self.length; i++) {
      r[i] = f(self[i]);
    }
  }
  function reduce(
    uint[] memory self,
    function (uint, uint) pure returns (uint) f
  )
    internal
    pure
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
```

```
    }
  }
  function range(uint length) internal pure returns (uint[] memory r) {
    r = new uint[](length);
    for (uint i = 0; i < r.length; i++) {
      r[i] = i;
    }
  }
}

contract Pyramid {
  using ArrayUtils for *;
  function pyramid(uint l) public pure returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);//前一个的返回值作为后一个的参
数
  }
  function square(uint x) internal pure returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
  }
}
```

使用外部函数的例子：(对于不习惯将函数当作类型的读者，可能会比较陌生)

```
pragma solidity >=0.4.22  <0.9.0;

contract Oracle {
  struct Request {
    bytes data;
    function(uint) external callback;
  }
  Request[] private requests;
  event NewRequest(uint);
  function query(bytes memory data, function(uint) external callback) public {
    requests.push(Request(data, callback));
    emit NewRequest(requests.length - 1);
  }
  function reply(uint requestID, uint response) public {
    // 这里检查回复来自可信来源
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant private ORACLE_CONST =
Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa)); // known contract
  uint private exchangeRate;
  function buySomething() public {
    ORACLE_CONST.query("USD", this.oracleResponse);
  }
  function oracleResponse(uint response) public {
    require(
        msg.sender == address(ORACLE_CONST),
        "Only oracle can call this."
    );
```

```
    exchangeRate = response;
  }
}
```

引用类型

引用类型可以通过不同变量名来修改指向的同一个值。目前的引用类型包括：结构体、数组和映射。

在使用引用类型时，需要指明这个类型存储在哪个数据域（data area）

- memory:存储在内存里，只在函数内部使用，**函数内变量不做特殊说明为`memory`类型。**
- storage:相当于全局变量。**函数外合约内的都是`storage`类型。**
- calldata:保存函数的参数的特殊储存位置，只读，大多数时候和`memory`相似。

如果可以的话，尽可能使用`calldata`临时存储传入函数的参数，因为它既不会复制，也不能修改，而且还可以作为函数的返回值。

数据的赋值

更改位置或者类型转化是拷贝；同一位置赋值一般是引用

- `storage`和`memory`之间的赋值或者用`calldata`对它们赋值，都是产生独立的拷贝，不修改原来的值。
- `memory`之间的赋值，是引用。
- `storage`给合约的全局变量赋值总是引用。
- 其他向`storage`赋值是拷贝。
- 结构体里面的赋值是一个拷贝。

```
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint[] x; //函数外变量都默认 storage

    // 函数内变量都是 memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // memory 给函数外的storage变量赋值，拷贝
        uint[] storage y = x; // storage 之间 指针传递，节省内存
        y.pop(); // 同时修改X
        delete x; // 重置X,同时修改Y
        g(x); // 函数传参时,也遵守规则，这里是传引用
        h(x); //这里传复制
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

数组

- 创建多维数组时，下标的用法有些不一样，`a[2][4]`表示4个子数列，每个子数列里2个元素，所以`a.length`等于4。但是访问数组时下标的顺序和大多数语言相同。
- `a[3]`，其中`a`也可以是数组，即`a[3]`是多维数组。

- 多维数组的声明不要求写明长度，初始化如下`uint[][] a=[[1,2,3],[4,5,6]];`，当然也可以`uint[5][7] a=[[1,2,3],[4,5,6]];`，不够的位置用0来补上。
- 动态数组也支持切片访问，`x[start:end]`其中的`start`和`end`会隐式的转化成`uint256`类型，`start`默认是0，`end`默认到最后，因此可以省略其中一个。切片不能够使用数组的操作成员，但是会隐式地转换成新的数组，支持进一步地按索引访问。目前，**只有`calldata`的数组才支持切片。**
- 数组可以是任何类型，包括映射和结构体。但是**数组中的映射只能是storage类型。**
- `bytes.concat`函数可以把`bytes`或者`bytes1 ... bytes32`拼接起来，返回一个`bytes memory`的数组。
- `.push`在数列末尾添加元素，返回值是对这个元素的引用。
- 使用`new`创建的`memory`类型的数组，内存一旦分配就是固定的，因此，不能够使用`.push`改变数组的大小。

```
pragma solidity >=0.4.16 <0.9.0;

contract TX {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);

        assert(a.length == 7);
        assert(b.length == len);

        a[6] = 8;
    }
}
```

定长字节数组

`bytes1`， `bytes2`， `bytes3`， …， `bytes32`是存放1，2，3，直到 32 个字节的字节序列。它们看成是数组。比较特别的是，它们也可以比较大小，移位，但是不能够进行四则运算。

对于多个字节序列，可以使用`bytes32[k]`之类的数组存储，但是这样使用很浪费空间，往往还是当成整体来使用，太长时就用下面将介绍的`bytes`类型.

- `byte`作为`bytes1`的别名（在0.8.0之前）
- 可以使用`.length`获取字节数，即字节数组长度。

变长字节数组

`bytes`和`string`，当然还有一般的数组类型如`uint[]`

Solidity 没有字符串操作函数但是可以使用第三方的字符串库，不过可以使用keccak256-hash来比较两个字符串`keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))`，或者使用`bytes.concat(bytes(s1), bytes(s2))`来连接两个字符串。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
contract Hello {
    string public greet = "Hello, ";
    function getstr(string calldata a) public view  returns (string memory){
        return string(bytes.concat(bytes(greet),bytes(a)));
    }
}
```

`bytes` 和 `string` 是特殊的数组，一方面元素的内存是紧密连续存放的，不是按照32个字节一单元的方式存放。其中 `bytes` 可以通过下标访问（`bytes(Name)[index]` 或者 `Name[index]`），返回的是底层的 `bytes` 类型的 UTF-8 码； `string` 不能够通过下标访问。我们一般是用固定的 `bytes` 类型(如 `bytes1`, `bytes2` ,...., `bytes32`)，因为 `byte[]` 类型的可变长数组每个元素是占32个字节，一个单元用不完会自动填充0，消耗更多的gas。

数组的赋值和字面常量

数组字面常量是在方括号中（ `[...]` ） 包含一个或多个逗号分隔的表达式。 例如 `[1, a, f(3)]` 。

它是静态（固定大小）的memory类型的数组，长度由元素个数确定，数组的基本类型这样确定：

通过字面量创建的数组以列表中第一个元素的类型为准，其他的元素会隐式转化，但是这种转换需要合法。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] memory) public pure {
        // ...
    }
}
```

上面就是 `uint` 类型的数组字面常量。 `[1,-1]` 就是不合法的，因为正整数默认是 `uint8` 类型，而第二个元素是 `-1` ，是 `int8` 类型，数组字面常量的元素的类型就不一致了。 `[int8(1),-1]` 就是合法的。

更进一步，在2多维数组中，每个子列表的第一个元素都要是同样的类型:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffff, 2], [uint24(0xff),
3], [uint24(0xffff), 4]];
        // The following does not work, because some of the inner arrays are not
of the right type.
        // uint[2][4] memory x = [[0x1, 1], [0xffffff, 2], [0xff, 3], [0xffff,
4]];
        return x;
    }
}
```

通过数组的字面常量创建数组，不支持动态分配内存，必须预设数组大小。`uint[] memory x = [uint(1), 3, 4];`报错，必须写成`uint[3] memory x = [uint(1), 3, 4];`。这个考虑移除这个特性，但是会造成ABI中数组传参的一些麻烦。

如果是先创建 memory 的数组，再传参，也不能通过数组的字面常量赋值，必须单独给元素赋值:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        uint[] memory x = new uint[](3);
        x[0] = 1;
        x[1] = 3;
        x[2] = 4;
    }
}
```

数组的成员

- `.length`: 返回当前数组的长度。
- `.push()`: 除了`string`类型，其他的动态storage数组和`bytes`都可以使用这个函数在数组的末尾添加一个元素，这个元素默认是0，**返回对这个元素的引用**，`x.push() = b`，修改`b`即可实现对数组元素的修改。
- `.push(x)`: 将`x`添加到数组末尾，没有返回值。
- `.pop()`:除了`string`类型，其他的动态数组和`bytes`都可以使用这个函数删除数组的最后一个元素，相当于隐式地`delete`这个元素。（注意 delete 的效果，并不是删除）

可以看出，`push`增加一个元素的gas是固定的，因为储存单元的大小是确定的，但是使用`pop()`等同执行`delete`操作，擦除大量的空间可能会消耗很多gas。

注意：如果需要在外部（external）函数中使用多维数组，这需要启用 ABI coder v2 (在合约最开头加上`pragma experimental ABIEncoderV2;`，这是为了方便 ABI 编码）。 公有（public）函数中默认支持的使用多维数组。

注意：在Byzantium（在2017-10-16日4370000区块上进行硬分叉升级）之前的EVM版本中，无法访问从函数调用返回动态数组。 如果要调用返回动态数组的函数，请确保 EVM 在拜占庭模式或者之后的模式上运行。

结构体

结构体的辖域

- 定义在合约之外的结构体类型，可以被所有合约引用。
- 合约内定义的结构体，只能在合约内或者是继承后的合约内可见。
- 结构体的使用和C语言类似，但是注意，结构体不能使用自身。

注意：在 Solidity 0.70 以前`memory`结构体只有`storage`的成员。

结构体赋值办法：

`structName(para1, para2, para3, para4)`或者`structName(paraName1:para1, paraName2:para2, paraName3:para3)`

映射

映射类型在声明时的形式为 `mapping(_KeyType => _ValueType)`。声明映射类型的变量的形式为 `mapping(_KeyType => _ValueType) _VariableName`.

其中 `_KeyType` 可以是任何内置的类型，包括 `bytes`、`string` 以及合约类型和枚举类型，但是不能是自定义的复杂类型，映射、结构体以及数组。`_ValueType` 可以是任何类型。但是，映射实际上是哈希表，`key` 存储的是 `keccak256` 的哈希值而不是真实的 `key`。因此，底层存储方式上并不是键值对的集合。

映射只能被声明为 `storage` 类型，不可以作为 `public` 函数的参数或返回值。如果结构体或者数组含有映射类型，也需要满足这个规则。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

可迭代的映射

我们使用嵌套映射和结构体，来实现复杂的数据结构，比如链表。以下例子有些难懂，这是通过位置（索引）和关键词构成的链式结构。理解思想即可，需要用到再深入学习。

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

//在合约外定义的变量，是全局变量。

struct IndexValue { uint keyIndex; uint value; } //关键词对应的索引和对应的值
struct KeyFlag { uint key; bool deleted; } //标记关键词是否删除

//这类似于链表。data 用于从当前位置传递到下一个位置，每一个位置都有关键词的索引和值，构成链式结构。
//而KeyFlag 用于记录每个节点（关键词+值）是否删除
//size 标记链表长度
struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

//这是库，里面很多函数可用
```

```
library IterableMapping {
    //插入
    function insert(itmap storage self, uint key, uint value) internal returns
(bool replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;//已经存在
        else {
            keyIndex = self.keys.length;

            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    //删除
    function remove(itmap storage self, uint key) internal returns (bool
success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    //是否包含某个元素
    function contains(itmap storage self, uint key) internal view returns (bool)
{
        return self.data[key].keyIndex > 0;
    }

    function iterate_start(itmap storage self) internal view returns (uint
keyIndex) {
        return iterate_next(self, type(uint).max);
    }

    function iterate_valid(itmap storage self, uint keyIndex) internal view
returns (bool) {
        return keyIndex < self.keys.length;
    }

    function iterate_next(itmap storage self, uint keyIndex) internal view
returns (uint r_keyIndex) {
        keyIndex++;
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return keyIndex;
    }

    function iterate_get(itmap storage self, uint keyIndex) internal view
returns (uint key, uint value) {
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
```

```
    }
}

// 如何使用
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(i);
            s += value;
        }
    }
}
```

类型转换

自定义类型

注意 `type UFixed256x18 is uint256` 的定义方式

`UFixed256x18.unwrap(a)` 从自定义类型，解封装成内置类型

`UFixed256x18.wrap(a * multiplier)` 从内置类型封装成自定义类型。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// Represent a 18 decimal, 256 bit wide fixed point type using a user defined
value type.
type UFixed256x18 is uint256;

/// A minimal library to do fixed point operations on UFixed256x18.
library FixedMath {
    uint constant multiplier = 10**18;

    /// Adds two UFixed256x18 numbers. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns
(UFixed256x18) {
```

```
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) +
UFixed256x18.unwrap(b));
    }
    /// Multiplies UFixed256x18 and uint256. Reverts on overflow, relying on
checked
    /// arithmetic on uint256.
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18)
{
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }
    /// Take the floor of a UFixed256x18 number.
    /// @return the largest integer that does not exceed `a`.
    function floor(UFixed256x18 a) internal pure returns (uint256) {
        return UFixed256x18.unwrap(a) / multiplier;
    }
    /// Turns a uint256 into a UFixed256x18 of the same value.
    /// Reverts if the integer is too large.
    function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(a * multiplier);
    }
}
```

基本类型转换

隐式转换：隐式转换发生在编译时期，如果不出现信息丢失，其实都可以进行隐式转换，比如`uint8`可以转成`uint16`。隐式转换常发生在不同的操作数一起用操作符操作时发生。

显式转换：如果编译器不允许隐式转换，而你足够自信没问题，那么就去尝试显示转换，但是这很容易造成安全问题。

高版本的Solidity不支持常量的不符合编译器的显式转换，但是允许变量之间进行显式转换。对于`int`转`uint`就是找补码，负数可以理解为下溢。如果是`uint`或者`int`同类型强制转换，就是从最低位截断(十六进制下，或者从最高位补0。

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

对于`bytes`类型就是从最低位补0或者从最高位开始保留，这样就没有改变原来的下标。

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b will be 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

只有具有相同字节数的整数和`bytes`类型才允许之间的强制转换，不同长度的需要中间过渡。注意：`bytes32`,表示32个字节，一个字节是8位；`int256`这样指的是二进制位。

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b will be 0x00001234
uint32 c = uint32(bytes4(a)); // c will be 0x12340000
uint8 d = uint8(uint16(a)); // d will be 0x34
uint8 e = uint8(bytes1(a)); // e will be 0x12
```

`bytes`数组和`calldate`的`bytes`的切片转换成`bytes32`这样的定长字节类型，截断和填充和定长`bytes`一致。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16,
bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m);  // if length of m is greater than 16,
truncation will happen
        b = bytes16(s);  // padded on the right, so result is
"abcdefgh\0\0\0\0\0\0\0\0"
        bytes3 b1 = bytes3(s); // truncated, b1 equals to "abc"
        b = bytes16(c[:8]);  // also padded with zeros
        return (b, b1);
    }
}
```

地址类型转换

`address payable`可以完成到`address`的隐式转换，但是从`address`到`address payable`必须显式的转换, 通过`payable(<address>)`进行转换。 某些函数会严格限制采用哪一种类型。

实际上，合约类型、`uint160`、整数字面常量、`bytes20`都可以与`address`类型互相转换。

- 如果有需要截断的情况，byte类型需要转换成`uint`之后才能转换成地址类型。`bytes32`就会被截断，且在`0.4.24`之后需要做显式处理`address(uint(bytes20(b)))`。
- 合约类型如果已经绑定到已部署的合约，可以显式转换成已部署合约的地址。
- 字面常量是"0xabc...."的字符串，可以当作地址类型直接使用。

字面常量类型转换

- 0.8.0以后整型的字面产常量的强转必须在满足隐式转化的条件之上，而且整数的隐式转换非常严格，不存在截断。
- 字节型的字面常量只支持同等大小的十六进制数转化，不能由十进制转化。但是如果字面常量是十进制的0或者十六进制的0，那么就允许转换成任何的定长字节类型。

```
bytes2 a = 54321; // not allowed
bytes2 b = 0x12; // not allowed
bytes2 c = 0x123; // not allowed
bytes2 d = 0x1234; // fine
bytes2 e = 0x0012; // fine
bytes4 f = 0; // fine
bytes4 g = 0x0; // fine
```

- 字符串字面常量转换成定长字节类型也需要大小相同。

```
bytes2 a = hex"1234"; // fine
bytes2 b = "xy"; // fine
bytes2 c = hex"12"; // not allowed
bytes2 d = hex"123"; // not allowed
bytes2 e = "x"; // not allowed
bytes2 f = "xyz"; // not allowed
```

- 只有大小正确（40位十六进制，160个字节）的满足检验和的十六进制常量才能转换成地址类型。

函数可见性类型转化

函数的可见性类型可以发生隐式的转化，**规则是**：只能变得比以前更严格，不能改变原来的限制条件，只能增加更多的限制条件。

有且仅有以下三种转化：

- `pure` 函数可以转换为 `view` 和 `non-payable` 函数
- `view` 函数可以转换为 `non-payable` 函数
- `payable` 函数可以转换为 `non-payable` 函数

如果在{internal,external}的位置是`public`，那么函数既可以当作内部函数，也可以当作外部函数使用，如果只想当内部函数使用，就用`f`（函数名）调用，如果想当作外部函数调用，使用`this.f`（地址+函数名，合约对象.函数名）

字面常量

地址字面常量

地址的字面常量有EIP-55的标准（区分大写字母和小写字母来验证），只有经过校验后才能作为address变量。

有理数和整数的字面常量

十进制的小数字面常量都会带一个`.`比如`1.` `.1` `1.2`。。`.5*8`的结果是整型的`4`。

也支持科学计数法，但是指数部分需要是整数（防止无理数出现）。`2e-10`

为了提高可读性，数字之间可以添加下划线，编译器会自动忽略。但是下划线不允许连续出现，而且只能出现在数字之间。`1_2e345_678`

数值字面常量支持任意精度和长度，也支持对应类型的所有运算，字面常量的运算结果还是字面常量。但是如果出现在变量表达式就会发生隐式转化，并且不同类型的字面常量和变量运算不能通过编译。也就是说`2**800+1-2**800`在字面常量中是允许的

（在`0.4.0`之前，整数的字面常量会被截断即`5/2=2`，但是之后是`2.5`）

字面类型的运算还是字面常量，和非字面常量运算，就会隐式转化成普通类型

字符串字面常量及类型

字符串字面常量（"foo"或者'bar'这样），可以分段写(`"foo""bar"`等效为`"foobar"`)

字符串`"foo"`相当于3个字节，而不是4个字节，它不像C语言里以`\0`结尾。

字符串字面常量可以隐式的转换成`bytes1`,... `bytes32`。在合适的情况下，可以转换成`bytes`以及`string`

字符串字面常量只包含可打印的ASCII字符和下面的转义字符：

- `\<newline>` (转义实际换行)
- `\\` (反斜杠)
- `\'` (单引号)
- `\"` (双引号)
- `\b` (退格)
- `\f` (换页)
- `\n` (换行符)
- `\r` (回车)
- `\t` (标签 tab)
- `\v` (垂直标签)
- `\xNN` (十六进制转义，表示一个十六进制的值，)
- `\uNNNN` (unicode 转义，转换成UTF-8的序列)

十六进制的字面常量

十六进制字面常量以关键字`hex`打头，后面紧跟着用单引号或双引号引起来的字符串（例如，`hex"001122FF"`）。 字符串的内容必须是一个十六进制的字符串，它们的值将使用二进制表示。

用空格分隔的多个十六进制字面常量被合并为一个字面常量：`hex"00112233" hex"44556677"`等同于`hex"0011223344556677"`

内置函数和变量

单位

- 币的单位默认是`wei`，也可以添加单位。

```
1 wei == 1;
1 gwei == 1e9;
1 ether == 1e18;
```

- 时间单位，默认是秒。但是需要注意闰秒和闰年的影响，这里的统计的时间并不是完全和日历上的时间相同。

```
1 == 1 seconds`
1 minutes == 60 seconds
1 hours == 60 minutes
1 days == 24 hours
1 weeks == 7 days
```

区块和交易的属性

括号内表示返回值类型

- `blockhash(uint blockNumber) returns (bytes32)`：指定区块的区块哈希，但是仅可用于最新的 256 个区块且不包括当前区块，否则返回0.
- `block.chainid` (`uint`): 当前链 id
- `block.coinbase` (`address`): 挖出当前区块的矿工地址
- `block.difficulty` (`uint`): 当前区块难度
- `block.gaslimit` (`uint`): 当前区块 gas 限额
- `block.number` (`uint`): 当前区块号
- `block.timestamp` (`uint`): 自 unix epoch 起始当前区块以秒计的时间戳
- `gasleft() returns (uint256)` ： 剩余的 gas
- `msg.data` (`bytes`): 完整的 calldata
- `msg.sender` (`address`): 消息发送者（当前调用）
- `msg.sig` (`bytes4`): calldata 的前 4 字节（也就是函数标识符）
- `msg.value` (`uint`): 随消息发送的 wei 的数量
- `tx.gasprice` (`uint`): 交易的 gas 价格
- `tx.origin` (`address payable`): 交易发起者（完全的调用链）

注意几大变化：

- `gasleft` 原来是 `msg.gas`
- `block.timestamp` 原来是 `now`
- `blockhash` 原来是 `block.blockhash`

delete

`delete a`不是常规意义上的删除，而是给`a`赋默认值（即返回不带参数的声明的状态），比如`a`是整数，那么等同于`a=0`。对用动态数组是将数组长度变为0；对于静态数组是将每一个元素初始化；对于结构体就把每一个成员初始化；对于映射在原理上无效（不会影响映射关系），但是会删除其他的成员，如值。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[]
is a complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local
variables
        // referencing storage objects can only be made from existing storage
objects.
        assert(y.length == 0);
    }
}
```

ABI 编码及解码函数

详细原理和应用见下一节 应用二进制接口，需要明白 ABI 编码的含义才懂这些函数的用法。

- `abi.decode(bytes memory encodedData, (...)) returns (...)`：对给定的数据进行ABI解码，而数据的类型在括号中第二个参数给出 。 例如: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes)`： 对给定参数进行编码

例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract A {
    bytes public c = abi.encode(5,-1);
    uint public d;//5
    int public e;//-1

    constructor(){
        (d,e) = abi.decode(c,(uint,int));
    }
}
```

- `abi.encodePacked(...) returns (bytes)`：对给定参数执行 紧打包编码 （即编码时不够 32 字节，不用补0了。
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes)`： ABI - 对给定第二个开始的参数进行编码，并以给定的函数选择器作为起始的 4 字节数据一起返回
- `abi.encodeWithSignature(string signature, ...) returns (bytes)`：等价于 `abi.encodeWithSelector(bytes4(keccak256(signature), ...)`

用法如下：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract A {
    uint public a;

    function add(uint b,uint c) public returns (uint) {
        a=a+b+c;
        return a;
    }

    bytes public encodedABI=abi.encodeWithSelector(this.add.selector,5,1);

    function callFunc()public returns(bool,bytes memory,uint) {
        bool flag=false;
        bytes memory result;
       (flag,result) = address(this).call(encodedABI);
       return (flag,result,a);
    }
    fallback() external{

    }
```

```
}
```

错误处理

`assert(bool condition)`,`require(bool condition)`,`require(bool condition, string memory message)`均是条件为假然后回滚。

`revert()`,`revert(string memory reason)` 立即回滚。

数学和密码学函数

- `addmod(uint x, uint y, uint k) returns (uint)`: 计算 `(x + y) % k`，加法会在任意精度下执行，并且加法的结果即使超过 `2**256` 也不会被截取。从 0.5.0 版本的编译器开始会加入对 `k != 0` 的校验（assert）。
- `mulmod(uint x, uint y, uint k) returns (uint)`: 计算 `(x * y) % k`，乘法会在任意精度下执行，并且乘法的结果即使超过 `2**256` 也不会被截取。从 0.5.0 版本的编译器开始会加入对 `k != 0` 的校验（assert）。
- `keccak256((bytes memory) returns (bytes32)`: 计算 Keccak-256 哈希。0.5.0以前有别名 `sha3`.它一般用于：生成输入信息的独一无二的标识。

例如：函数选择器即函数签名

```
pragma solidity >=0.5.0 <0.9.0;
contract A {
    uint public a;
    function add(uint b) public {
        a+=b;
    }
    bytes4 public selector = this.transfer.selector;
    bytes4 public genSelector = bytes4(keccak256("add(uint256)"));
    bool public isequal = (selector==genSelector);
}
```

- `sha256(bytes memory) returns (bytes32)`: 计算参数的 SHA-256 哈希。
- `ripemd160(bytes memory) returns (bytes20)`: 计算参数的 RIPEMD-160 哈希。
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)` 利用椭圆曲线签名恢复与公钥相关的地址。

函数参数对应于 ECDSA签名的值:

`r` = 签名的前 32 字节

`s` = 签名的第2个32 字节

`v` = 签名的最后一个字节

(以后还需要补充许多密码学知识)

合约相关

`this`：表示当前合约的实例。

`selfdestruct(address payable recipient)`：在交易成功结束后，销毁合约，并且把余额转到指定地址。接受的合约不会运行。

类型信息

`type(X)` 返回 `X` 的类型，目前只支持整型和合约类型，未来计划会拓展。

用于合约类型 `C` 支持以下属性:

- `type(C).name`:

  获得合约名

- `type(C).creationCode`:

  获得包含创建合约的字节码的`memory byte[]`数组。只能在内联汇编中使用。

- `type(C).runtimeCode`

  获得合同的运行时字节码的内存字节数组，通常在构造函数的内联汇编中使用。

在**接口类型** `I` 下可使用:

- `type(I).interfaceId`:

  返回接口 `I` 的 `bytes4` 类型的接口 ID，接口 ID 参考： EIP-165 定义的， 接口 ID 被定义为 接口内所有的函数的函数选择器（除继承的函数）的 `XOR` （异或）。

对于**整型** `T` 有下面的属性可访问:

- `type(T).min`

  `T` 的最小值。

- `type(T).max`

  `T` 的最大值。

应用二进制接口(ABI)

在地址类型的介绍中提到了底层调用的 call函数，这里将会介绍它的用法，以及和ABI函数的配合。

ABI 全名 Application Binary Interface，翻译为应用二进制接口。它定义了与合约交互的规范，因此底层函数 (如 call) 直接给合约发消息前，需要了解 ABI。

ABI 是由合约生成的，规定与合约交互方式的规则，它是一个接口，常被 web3 等库调用。熟悉 REST API 的读者应该能很快理解。

接口含义

例如下面是智能合约:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Owner
 * @dev 设置和改变所有者
 */
contract Owner {

    address private owner;

    // 设置事件
    event OwnerSet(address indexed oldOwner, address indexed newOwner);
```

```
    // 函数修饰器，限制调用者必须是所有者
    modifier isOwner() {
        // 如果require 第一个参数为false，就回滚，并且日志中包含作为错误信息的第二个参数。它常用于限制合约调用是否合法。
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    /**
     * @dev 构造函数默认部署者为所有者
     */
    constructor() {
        owner = msg.sender; // 'msg.sender' is sender of current call, contract deployer for a constructor
        emit OwnerSet(address(0), owner);
    }


    function changeOwner(address newOwner) public isOwner {
        emit OwnerSet(owner, newOwner);
        owner = newOwner;
    }

    function getOwner() external view returns (address) {
        return owner;
    }
}
```

它的 ABI 如下：

```
[
            {
                "inputs": [],
                "stateMutability": "nonpayable",
                "type": "constructor"
            },
            {
                "anonymous": false,
                "inputs": [
                    {
                        "indexed": true,
                        "internalType": "address",
                        "name": "oldOwner",
                        "type": "address"
                    },
                    {
                        "indexed": true,
                        "internalType": "address",
                        "name": "newOwner",
                        "type": "address"
                    }
                ],
                "name": "OwnerSet",
                "type": "event"
            },
            {
```

```
                "inputs": [
                    {
                        "internalType": "address",
                        "name": "newOwner",
                        "type": "address"
                    }
                ],
                "name": "changeOwner",
                "outputs": [],
                "stateMutability": "nonpayable",
                "type": "function"
            },
            {
                "inputs": [],
                "name": "getOwner",
                "outputs": [
                    {
                        "internalType": "address",
                        "name": "",
                        "type": "address"
                    }
                ],
                "stateMutability": "view",
                "type": "function"
            }
        ]
```

对于函数：

- `type`: `"function"`, `"constructor"` (可以省略，默认是 `"function"`; 也可能是`"fallback"`);
- `name`: 函数的名字;
- `constant`: `true` 表示该函数调用不修改区块链状态，只读或者只生成调用后销毁的 memory 变量;
- `payable`: `true` 表示可以接收以太币, 默认是 `false`;
- `stateMutability`:四种结果， `pure` (不读取也不修改状态), `view` (不修改状态，和上面的 `constant` 是等价的), `nonpayable` and `payable` (否，是接收以太币);
- `inputs`: 对象的数组，包括:
  - `name`: 参数的名字
  - `type`: 参数的类型
- `outputs`: 和 `inputs` 类似, 如果没有输出可为空.

对于下面这个函数的解读：

input 输入变量是内置类型 (internalType) 中的地址类型 (address)，类型 (type) 是 地址 (address)。

output 返回值为空。

该函数的属性标记 (stateMutability) 是不可转账 (nonpayable) ，类型是函数 (function)

```
        {
            "inputs": [
                {
                    "internalType": "address",
                    "name": "newOwner",
                    "type": "address"
                }
            ],
            "name": "changeOwner",
            "outputs": [],
            "stateMutability": "nonpayable",
            "type": "function"
        },
```

对于事件:

- `type`: `"event"`
- `name`:事件名字;
- `inputs`:对象的数组，包括:
  - `name`: 参数的名字
  - `type`: 参数的类型
  - `indexed`: `true` 表示是特殊结构 `topics` 的一部分（见事件的 indexed 修饰），`false` 表示日志文件.
- `anonymous`: `true` 表示事件被声明为 `anonymous`.

对下面这个事件的解读:

非匿名，输出参数有两个，花括号类型标记，其余与函数差别不大。

```
        {
            "anonymous": false,
            "inputs": [
                {
                    "indexed": true,
                    "internalType": "address",
                    "name": "oldOwner",
                    "type": "address"
                },
                {
                    "indexed": true,
                    "internalType": "address",
                    "name": "newOwner",
                    "type": "address"
                }
            ],
            "name": "OwnerSet",
            "type": "event"
        },
```

至于全局变量 `address private owner`，由于 private 限制访问，所以不在 ABI 中。

从上面的格式中，可以看到 ABI 和编码很相关，发送的数据应当 ABI 的方式组织，同样的也需要对应的编码格式。

函数选择器

函数选择器也和接口强相关，因为在 call 之类的底层调用中，需要根据函数签名匹配函数。函数调用依靠调用数据的前四个字节匹配函数，这四个字节是函数签名的哈希值的前四个字节。调用函数的数据的编码格式按照顺序是函数名和带圆括号的参数类型列表，参数之间只靠逗号分隔。注意函数返回值并不参与哈希，这样可以进一步解耦，更灵活地重载。

详细选择函数过程需要深入执行过程，可见博客，后面我也会学习。

ABI 的参数编码

编码规则如下:

设 `X` 是编码前的值，对于静态类型 `a` (内置的类型)，定义 `len(a)` 是 `a` 转化成二进制数后的位数(注：所有类型底层最终是由二进制数表示)；对于动态类型 `a` (如数组、元组，`bytes`、`string`、`T[k]`)，我们常用编码后的长度 `len(enc(a))`。`enc()` 是我们定义的函数，它输入参数是类型 (包括静态类型和动态类型)，返回值是二进制序列 (一串二进制数，但是含义不在于数值而是字符排列顺序)。

我们的核心就在于如何定义编码函数 `enc()`。首先设定编码的基本格式

1. 对于元组，表示如下。不同的 `head()` 放在一块，表示二进制代码直接拼接。

`enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`，函数的参数列表就是元组。

定义 `head` 和 `tail` 如下：

- 若 `X` 是静态类型，`head(X(i)) = enc(X(i))`，`tail(X(i)) = ""`（空字符串）。因为静态类型是唯一的，可以直接编码，无需额外的参数说明。
- 若 `X` 是动态类型: `head(X) = enc(len(head(X) tail(X)))`，`tail(X(i)) = enc(X(i))`，即需要在实际编码值前面添加编码后的长度。一来方便读取，也明确了动态的类型的确切类型（如 `T` 类型的数组 `T[k]`，确切类型是长度为 `k`，类型为 `T` 的数组）。

2. 对于一般变量的编码规则
    1. `T[k]` 对于任意 `T` 和 `k`：数组当作同类型变量凑在一起的元组

       `enc(X) = enc((X[0], ..., X[k-1]))`
    2. `T[]` 当 `X` 有 `k` 个元素（`k` 默认是类型 `uint256`）：不定长数组多了元素的个数作为前缀。

       `enc(X) = enc(k) enc([X[1], ..., X[k]])`
    3. 具有 `k`（呈现为类型 `uint256`）个字节的 `bytes`：不定长数组多了元素的个数作为前缀，如果 `byte` 数组，然后直接抄下来。

       `enc(X) = enc(k) pad_right(X)`，`pad_right(x)` 的意思是在左边把原来的字节序列添加上去，填充在右边，`enc(k)` 参靠下面 `uint` 类型的编码方式。
    4. `string`：先把 `string` 类型转成 `bytes` 类型，注意 `k` 是指转化后的字节数。

       `enc(X) = enc(enc_utf8(X))`，`enc_utf8(X)` 指将`string` 类型转成 `bytes` 类型。
    5. `uint<M>`：`enc(X)` 是在 `X` 的高位补充若干 0 直到长度为 32 字节。
    6. `address`：与 `uint160` 的情况相同。
    7. `int<M>`：`enc(X)` 是在 `X` 补码的高位添加若干字节，直到长度为 32 字节；
        - 如果 `X` 是负数，在高位一直添加值为 `0xff`（16 进制转二进制，实际上就是 8 个 1。注意 `int` 和 `uint` 这两类的位数都是 8 的倍数)
        - 对于 `X` 是非负数，在高位添加 `0x00` 值（即 8 位全为 0），直到为32个字节。

8. `bool`：与 `uint8` 的情况相同，`1` 用来表示 `true`，`0` 表示 `false`。
9. `fixed<M>x<N>`：`enc(X)` 就是 `enc(X * 10**N)`，其中 `X * 10**N` 可以理解为 `int256`。
10. `fixed`：与 `fixed128x18` 的情况相同。
11. `ufixed<M>x<N>`：`enc(X)` 就是 `enc(X * 10**N)`，其中 `X * 10**N` 可以理解为 `uint256`。
12. `ufixed`：与 `ufixed128x18` 的情况相同。
13. `bytes<M>`：`enc(X)` 就是 `X` 的字节序列加上为使长度成为 32 字节而添加的若干 0 值字节。

ABI编码的实际格式

在编写合约时，或者需要验证签名时，对于多个参数往往需要打包多个参数。调用函数时发送的数据从第五个字节开始就是参数的编码了。格式如下：`function_selector(f) enc((a_1, ..., a_n))`。

函数选择器即函数签名的 keccak256 哈希，函数签名即是我们写代码时见到的函数名和参数类型，如 `myFunc(string, uint8, uint8, uint8)`。函数选择器即这个哈希值的前 4 个字节。具体可以用 web3 库或者合约中的密码学函数实现。

具体例子：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 ||
y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

可得 `baz` 函数选择器：`0xcdcd77c0`

参数都是静态类型，所以

```
enc((uint32 39,bool true))
= head(uint32) head(bool) tail(uint32) tail(bool)
= enc(uint32 39) enc(bool true)
= enc(uint32 39) enc(uint32 1)
=0x0000000000000000000000000000000000000000000000000000000000000045+0x0000000000
0000000000000000000000000000000000000000000000000000000000000001 //注释：注意实际没有加号+，这
里为了方便说明两段是如何组合的

最终 ABI 编码如下：
0xcdcd77c000000000000000000000000000000000000000000000000000000000000045000000
0000000000000000000000000000000000000000000000000000000000000001`
```

更多的例子可见文档。

事件

事件是从 EVM 日志中提取出来的片段，为了方便解析它，事件类似于函数的 ABI 。事件有事件名和参数列表，编码时把参数列表分成两份，一份是带有 `indexed` 标识的参数列表（对于非匿名事件里面至多三个参数，对于匿名事件至多4个参数，在编写合约时也有这样的限制），另一部分则是无这个标识的参数列表。标有 `indexed` 的参数列表会和事件签名的 Keccak 哈希共同构成日志中的特殊数据结构 `topics`(这种数据结构便于检索)。无 `indexed` 标识的参数列表会根据普通类型的编码规则，生成序列。

详细地，事件的结构如下：

- `address`：由 EVM 自动提供的事件所在合约的地址；
- `topics[0]`：`keccak(事件名+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")`

  `EVENT_ARGS.map(canonical_type_of).join(",")` 表示事件的每个参数对应的类型，类型之间用逗号分开。例如，对 `event myevent(uint indexed foo,int b)`，那么 `topics[0]=keccak(myevent(uint,int))`。

  如果事件被声明为 `anonymous`，那么 `topics[0]` 不会被生成；
- `topics[n]`：`EVENT_INDEXED_ARGS[n - 1]`

  `EVENT_INDEXED_ARGS[n-1]` 是带有 `indexed` 标识的参数列表中下标为 `n-1` 的参数，即第 `n` 个参数；这个式子表示每个 topics 结构里面的内容是什么。
- `data`：`abi_serialise(EVENT_NON_INDEXED_ARGS)`

  `EVENT_NON_INDEXED_ARGS` 是不带有 `indexed` 标识的事件参数，`abi_serialise()` 把参数列表 ABI 编码，相当于前面提到的 `enc()` 编码函数。

关于设计原理的说明：

对于复杂的类型（超过 32 个字节或者时动态类型），比如结构体、`bytes`、`string`，编码的前面有 `keccak` 能够高效的检索这样的类型变量，但是也增加了解析的复杂度。因此，要精心设计将需要检索的参数标上 `indexed`，不需要检索而定位后直接获取的变量就不带 `indexed`。当然也可以制造冗余，每个变量都设置一个 带 `indexed` 的参数和不带 `indexed` 的参数，但是部署合约时 gas 会更高，调用消耗也会更高。

错误处理的编码

当人为设置回滚函数时，有时需要提供错误的描述性信息。这样的参数也会参与 ABI 的编码。

如下所示，`error` 是新增的类型，和事件类似，但是用于 `revert` 操作，提供提示信息。这里触发的 `error` 类型，将会以函数的编码方式编码，在以后可能会改变，将 `error` 的函数选择器，改成 错误选择器 error selector，固定为 四个字节的全 0（`0x00000000`）或者全 1（`0xffffffff`）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

底层函数交互

特殊交互方式

call 是底层的调用（没有封装过），直接发送消息给合约。方式如下：

1. 所有的参数，都会打包成一串32个字节，连续存放的序列。
2. 若第一个参数是函数的签名（函数哈希之后的前4个字节），则第二、第三这些后面的参数是函数的参数。如：`nameReg.call(bytes4(keccak256("fun(uint256)")), a);`

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

  用给定的合约发出低级 `CALL` 调用，返回成功状态及返回数据，发送所有可用 gas，也可以调节 gas。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract Receiver {
    event Received(address caller, uint amount, string message);

    fallback() external payable {//回退函数
        emit Received(msg.sender, msg.value, "Fallback was called");
    }

    function foo(string memory _message, uint _x) public payable returns (uint)
{
        emit Received(msg.sender, msg.value, _message);

        return _x + 1;
    }
}

contract Caller {
    event Response(bool success, bytes data);

    function testCallFoo(address payable _addr) public payable {
        // 注意观察调用的格式，结合前面学习的 ABI 编码方式。
        (bool success, bytes memory data) = _addr.call{value: msg.value, gas:
5000}(
            abi.encodeWithSignature("foo(string,uint256)", "call foo", 123)
        );

        emit Response(success, data);
    }

    // 不存在的函数调用会失败，但是同样会触发回调函数。
    function testCallDoesNotExist(address _addr) public {
        (bool success, bytes memory data) = _addr.call(
            abi.encodeWithSignature("doesNotExist()")
        );

        emit Response(success, data);
    }
}
```

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

  用给定的合约发出低级 `DELEGATECALL` 调用，返回成功状态并返回数据，失败时返回 `false`。上下文属于发出调用的合约。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

// NOTE:先部署这个合约
contract B {
    // NOTE: storage layout must be the same as contract A
    uint public num;
    address public sender;
    uint public value;

    function setVars(uint _num) public payable {
        num = _num;
        sender = msg.sender;
        value = msg.value;
    }
}

contract A {
    uint public num;
    address public sender;
    uint public value;

    function setVars(address _contract, uint _num) public payable {
        // 只改变了合约A的值，因为上下文属于合约A。
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("setVars(uint256)", _num)
        );
    }
}
```

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

  用给定的有效载荷 发出低级 `STATICCALL` 调用，如果改变了被调用合约的状态，立即回滚。

注意：`.call`会绕过类型检查，函数存在检查和参数打包。

注意：`send`调用栈深度达到1024就会失败。

注意：0.5.0以后不允许通过合约实例来访问地址成员`this.balance`。0.5.0以前，底层调用只会返回是否成功不会返回数据。

注意：因为EVM不会检查调用的合约是否存在，并且总是把调用不存在的合约视为成功，因此提供了`extcodesize`的操作码，确认合约存在（即合约地址内有代码），否则引起异常。注意底层调用不会触发。

注意：底层的调用略去了很多检查，使得他们更加节省gas但是安全性更低。

表达式和控制结构

Solidity 支持`if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`这些和C语言一样的关键字。

Solidity还支持 `try`/`catch` 语句形式的异常处理， 但仅用于 外部函数调用 和 合约创建调用.

由于不支持非布尔类型值转换成布尔类型，因此`if(1){}`是不合法的。

函数调用

内部调用

内部调用再EVM中只是简单的跳转（不会产生实际的消息调用），传递当前的内存的引用，效率很高。但是仍然要避免过多的递归，因为每次进入内部函数都会占用一个堆栈槽，而最多只有1024个堆栈槽。

外部调用

- 只有`external`或者`public`的函数才可以通过消息调用而不是单纯的跳转调用，外部函数的参数会暂时复制在内存中。
- `this`**不可以出现在构造函数里，因为此时合约还没有完成。**
- **调用时可以指定 value 和 gas** 。这里导入合约使用的时初始化合约实例然后赋予地址。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info{value: 10, gas: 800}(); }
}
```

需要注意到，`function callFeed() public { feed.info{value: 10, gas: 800}(); }`，花括号`{ feed.info{value: 10, gas: 800}`里的只是修饰，实际调用的时圆括号`()`里的内容。再0.7.0前，使用的时`f.value(x).gas(g)()`。

一般我们不推荐使用call调用除了`fallback`函数之外的函数，但是在考虑节省gas和保证安全性的前提下可以尝试。

函数参数写法

调用函数时参数还有一种写法：与函数声明的的名字对应。当然，最常见的还是按照顺序，忽略函数参数的名字。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }
```

```
}
```

用new创建合约实例

在已知一个合约完整的代码的前提下（比如写在同一个文件内），就可以使用`contractName newContractInstance{value:initial value}(constructor para)`，（注意无法限定gas，但是可以写明发送多少以太币过去）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

合约创建的新合约地址

合约的地址时由创建时交易的nonce和创建者的地址决定，但是还可以选择一个32个字节的`salt`来改变生成合约地址的方式，合约地址将会由创建者的地址、给定`salt`、被创建合约的字节码及参数共同决定。下面是计算方法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;

contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        /// 这个复杂的表达式只是告诉我们，如何预先计算合约地址。
        /// 这里仅仅用来说明。
        /// 实际上，你仅仅需要 ``new D{salt: salt}(arg)``.
```

```
        address predictedAddress =
address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                arg
            ))
        )))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

这一特性使得在销毁合约之后在重新在同一地址生成代码相同的合约。但是，尽管创建的字节码相同，但是由于编译器会检查外部的状态变化，`deploy bytecode`可能会不一样。

下面是创建多个合约的例子：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

contract Car {
    address public owner;
    string public model;

    constructor(address _owner, string memory _model) payable {
        owner = _owner;
        model = _model;
    }
}

contract CarFactory {
    Car[] public cars;

    function create(address _owner, string memory _model) public {
        Car car = new Car(_owner, _model);
        cars.push(car);
    }

    function createAndSendEther(address _owner, string memory _model)
        public
        payable
    {
        Car car = (new Car){value: msg.value}(_owner, _model);
        cars.push(car);
    }

    function getCar(uint _index)
        public
        view
        returns (address owner, string memory model, uint balance)
    {
        Car car = cars[_index];
```

```
        return (car.owner(), car.model(), address(car).balance);
    }
}
```

特别提到，调用已部署的合约，应当先引入合约的接口（或者源代码），然后 `合约名 a=合约名(地址)`

元组的赋值行为

函数的返回值可以是元组，因此就可以用元组的形式接收，但是必须按照顺序排列。在0.5.0之后，两个元组的大小必须相同，用逗号表示间隔，可以**空着省略元素**。注意，不允许赋值和声明都出现在元组里，比如 `(x, uint y) = (1, 2);` 不合法。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Common trick to swap values -- does not work for non-value storage
types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

注意：元组的赋值行为，它仍然保留了引用类型。

错误处理

调用和因这次调用而形成的调用链出现异常就会回滚所有更改，但是可以使用 `try` 或者 `catch` 只回滚到这一层（回滚不会到底，如 A 调用B, B 调用 C, 如果B 调用 C 时出错导致回滚，不会消除 A 调用 B 造成的影响）。

底层函数错误是不会回滚的，而是返回 `false` 和 `error instance`.

有两种错误类型，一种是 `error`，表示常规的错误。而 `Panic` 则表示代码没有问题，

`assert` 函数，用于检查内部错误，返回 `Panic(uint256)`，错误代码分别表示：

1. 0x00: 由编译器本身导致的Panic.
2. 0x01: `assert` 的参数（表达式）结果为 false 。
3. 0x11: 在 `unchecked { … }` 外，算术运算结果向上或向下溢出。
4. 0x12: 除以0或者模0.
5. 0x21: 不合适的枚举类型转换。
6. 0x22: 访问一个没有正确编码的 `storage` byte数组.
7. 0x31: 对空数组 `.pop()` 。
8. 0x32: 数组的索引越界或为负数。

9. 0x41: 分配了太多的内存或创建的数组过大。
10. 0x51: 如果你调用了零初始化内部函数类型变量。

`Error(string)`的异常（错误提示信息）由编译器产生，有以下情况：

1. `require` 的参数为 `false` 。
2. 触发 `revert` 或者 `revert("discription")`
3. 执行外部函数调用合约没有代码。
4. `payable` 修饰的函数（包括构造函数和 fallback 函数），接收以太币。
5. 合约通过 getter 函数接收以太币 。

以下即可能是`Panic`也可能是`error`

1. `.transfer()` 失败。
2. 通过消息调用调用某个函数，但该函数没有正确结束（例如, 它耗尽了 gas，没有对应的函数，或者本身抛出一个异常）。低级操作不会抛出异常，而通过返回 `false` 来指示失败。
3. 如果你使用 `new` 关键字创建未完成的合约 。

注意： `Panic`异常使用的是`invalid`操作码，会消耗所有可用gas. 在 都会 版本之前，require 也是这样。

注意： `revert errorInstance` 其中的`errorInstance`是自定义的错误实例，用`errorInstance`的名字来表示错误，在编码的时候只占4个字节（如果带参数的话可能不一样），因此，远比`Error(string)`的方式节省gas。错误实例和函数调用与错误实例同名且同参数的函数的函数的ABI编码相同，也就是说错误实例的数据是由ABI编码后的4个字节的选择器组成的。而这个选择器是错误实例的签名的keccak256-hash 的前4个字节。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        payable(msg.sender).transfer(address(this).balance);
    }
}
```

注意： `require` 是可执行的函数，在 `require(condition, f())` 里，函数 `f` 会被执行，即便 `condition` 为 True .

注意： `Error(string)`函数会返回16进制的 错误提示信息。

注意： `throw`等同于`reverse()` 但是在0.5.0废除了。

`try/catch`

`try`**后面只能接外部函数调用或者是创建合约**`new ContractName`**的表达式**，并且花括号里面的错误会立即回滚，当花括号调用合约以外的函数（或者以外部调用的形式调用函数，如用 `this`）出现错不会造成当前合约回滚。用 `try` 尝试调用的外部函数如果需要返回参数，就要在 `returns` 后面声明返回参数的类型，如果外部调用执行成功就可以获取返回值，继续执行花括号内的语句，花括号的语句都完全成功了，就会跳过后面的 `catch`；但是如果失败就会根据错误类型跳转到对应的 `catch` 里面。如下面的代码：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
//接口类型，后面会介绍，如果熟悉 Golang 的接口则很容易理解。
interface DataFeed { function getData(address token) external returns (uint
value); }

contract FeedConsumer {
    DataFeed feed;//从接口创建合约
    uint errorCount;//记录错误次数
    function rate(address token) public returns (uint value, bool success) {
        // 如果有十个及以上的错误就回滚
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {//尝试调用 外部的 getData 函数，执
行成功就获得返回值，然后继续执行花括号内的内容
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // 执行 revert 语句造成的回滚，返回错误提示信息
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // Panic类型错误。
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // 无返回提示的底层错误。
            errorCount++;
            return (0, false);
        }
    }
}
```

Solidity支持不同的`catch`代码块：

- `catch Error(string memory reason) { ... }`: 对应的执行条件是 `revert("reasonString")` or `require(false, "reasonString")` 或者是执行时内部的错误.
- `catch Panic(uint errorCode) { ... }`: 用于接收 Panic 类型的错误，比如用了 `assert`，数组下标越界，除以0，这些语言层面的错误。
- `catch (bytes memory lowLevelData) { ... }`: 如果发送错误类型不是前两种，比如无错误提示信息，或者是返回的错误提示信息无法解码（比如由编译器版本变迁造成），这个语句就会提供底层的编码后的错误提示信息。
- `catch { ... }`: 接收所有错误类型，但是不能出现前面的判断错误类型的分句。

注意：为了接收所有方式的错误，最后要使用 `catch { ...}` 或者 `catch (bytes memory lowLevelData) { ... }`.

注意：调用失败的原因多种多样，错误消息可能是来自调用链中的某一环，不一定来自被调用的合约。比如gas不足。在调用时会保留1/64的gas，以保证当前合约顺利执行。

合约的高级特性

receive 函数

一个合约至多有一个 receive 函数，形如 `receive() external payable { ... }`，注意：

- 没有 `function` 的标识
- 没有参数
- 只能是 `external` 和 `payable` 标识
- 可以有函数修饰器
- 支持重载。

`receive` **函数在调用数据为空时（如用 `call` 传入空字节，或者转账）执行**,如果没有设置 `receive` 函数，那么就会执行 `fallback` 函数，如果这两个函数都不存在，合约就不能通过交易的形式获取以太币。

注意 `receive` 函数只有2300gas可用，因此它进行其他操作的空间很小。 以下功能都因为超过消耗的 gas 而不能够实现。

- 写入存储
- 创建合约
- 调用消耗大量 gas 的外部函数
- 发送以太币

每一步都会消耗2300gas.

我们建议**只使用 `receive` 函数来接收以太币。**

回退函数

一个合约至多一个回退函数，格式如：`fallback () external [payable]` 或者 `fallback (bytes calldata _input) external [payable] returns (bytes memory _output)`，后者的函数参数会接收完整的调用数据（`msg.data`），返回未经过ABI编码的原始数据。

- 回退函数只当没有与调用数据匹配的函数签名时执行。
- 可以重载，也可以被修饰器修饰。
- 在函数调用时，如果**没有与之匹配的函数签名或者调用数据为空且无 `receive` 函数**，就会调用 `fallbakc` 函数。
- 如果回退函数代替了 `receive` 函数完成接收以太币的功能，那么仍然只有2300gas可用。

继承

继承的机制和python的非常相似，但是存在差异。一般而言使用过 C++, 基本已经掌握。

当合约继承其他的合约时，只会在区块链上生成一个合约，**所有相关的合约都会编译进这个合约，调用机制和写在一个合约上一致。**

继承时，全局变量无法覆盖，如果出现可见的同名变量会编译错误。通过例子来体会细节，重点理解语法，而不是程序逻辑。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;


contract Owned {
    constructor() { owner = payable(msg.sender); }//构造函数中的msg.sender 是部署者
    address payable owner;
```

```
}

    // `is` 是继承的关键词. 子合约可以接受父合约所有非 private 的东西.
contract Destructible is Owned {
    //  `virtual` 表示函数可以被重写
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);//只有调用函数的人是部署者，才能执行自毁操作
    }
}

// abstract用于提取合约的 接口，重写后实现更多的功能
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}

abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}

// 允许从多个合约继承.
contract Named is Owned, Destructible {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);//从地址创建 满足接口的 Condig 合约实例，用于调用
        NameReg(config.lookup(1)).register(name);//这里并未重写 lookup函数，因此返回值都是默认零值，这里创建0地址上的NameReg合约实例，然后注册管理者
    }

    // 将重写的函数需要使用overridden的标识，并且被重写的函数之前有virtual标识。
    //注意重写函数的名字，参数以及返回值类型都不能变。
    function destroy() public virtual override {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            Destructible.destroy();
        }
    }
}

// 如果父合约有构造函数，则需要填上参数。
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // 如果从多个合约继承了同名的可重写函数，需要在override后面指明所有同名函数所在的合约。
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

但是，继承是从右到左深度优先搜索来寻找同名函数（搜索的顺序是按 "辈分" 从小到大，而且继承多个合约时也要按着从右到左的顺序填上，如下图继承链是 D, C, B, A），一旦找到同名函数就停止，不会执行后面重复出现的重名函数。所以如果继承了多个合约，希望把上一级父合约的同名函数都执行一遍，就需要 `super` 关键词。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

/* Inheritance tree
   A
 /  \
B   C
 \ /
  D
*/

contract A {
    event Log(string message);

    function foo() public virtual {
        emit Log("A.foo called");
    }

    function bar() public virtual {
        emit Log("A.bar called");
    }
}

contract B is A {
    function foo() public virtual override {
        emit Log("B.foo called");
        A.foo();
    }

    function bar() public virtual override {
        emit Log("B.bar called");
        super.bar();
    }
}

contract C is A {
    function foo() public virtual override {
        emit Log("C.foo called");
        A.foo();
    }

    function bar() public virtual override {
        emit Log("C.bar called");
        super.bar();
    }
}

contract D is B, C {
    // Try:
    // - Call D.foo and check the transaction logs.
    //   Although D inherits A, B and C, it only called C and then A.
    // - Call D.bar and check the transaction logs
```

```
    //   D called C, then B, and finally A.
    //   Although super was called twice (by B and C) it only called A once.

    function foo() public override(B, C) {
        super.foo();
    }

    function bar() public override(B, C) {
        super.bar();
    }
}
```

更多的介绍请见官方文档。

函数重写

父合约中被标记为`virtual`的非private函数可以在子合约中用`override`重写。

重写可以改变函数的标识符，规则如下：

- 可见性只能单向从`external`更改为`public`。
- `nonpayable`可以被`view`和`pure`覆盖。
- `view`可以被`pure`覆盖。
- `payable`不可被覆盖。

如果有多个父合约有相同定义的函数，`override`关键字后必须指定所有父合约的名字，且这些父合约没有被继承链上的其他合约重写。

接口会自动作为`virtual`。

注意：特殊的，如果`external`函数的参数和返回值和`public`全局变量一致的话，可以把函数重写全局变量。

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

contract A
{
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

注意：函数修饰器也支持重写，且和函数重写规则一致。

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

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

抽象合约

如果合约至少有一个函数没有完成(例如：`function foo(address) external returns (address);`)，则该合约会被视为抽象合约，需要用 `abstract` 标明。

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

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

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

如果子合约没有重写父合约中所有未完成的函数，那么子合约也需要标注`abstract`

注意：声明函数类型的变量和未实现的函数的不同：

```
function(address) external returns (address) foo;//函数类型变量
function foo(address) external returns (address);//抽象合约的函数
```

抽象合约可以将定义合约和实现合约的过程分离开，具有更佳的可拓展性。

接口

接口和抽象合约的作用很类似，但是它的每一个函数都没有实现，而且不可以作为其他合约的子合约，只能作为父合约被继承。

接口中所有的函数必须是`external`，且**不包含构造函数和全局变量**。接口的所有函数都会隐式标记为`external`，可以重写。多次重写的规则和多继承的规则和一般函数重写规则一致。

```
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

库

库与合约类似，但是它们只在某个合约地址部署一次，并且通过 EVM 的`DELEGATECALL`（为了实现上下文更改）来实现复用。

当库中的函数被调用时，它的代码在当前合约的上下文中执行，并且只可以访问调用时显式提供的调用合约的状态变量。库本身没有状态记录（如 全局变量）。

如果库被继承的话，库函数在子合约是可见的，也可以直接使用，和普通的继承相同（属于库的内部调用方式）。为了改变状态，**内部的库（即不是通过地址引入的库）所有`data a rea`的传参需要都是传递一个引用**（库函数使用`storage`标识），在EVM中，编译也是直接把库包含进调用合约。

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

struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // 注意到这里使用的是storage引用类型
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // 如果已经存在停止插入
        self.flags[value] = true;
        return true;
    }

    function remove(Data storage self, uint value)
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // 如果不存在就比用移除
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    Data knownValues;

    function register(uint value) public {
        require(Set.insert(knownValues, value));
    }
```

```
    // In this contract, we can also directly access knownValues.flags, if we
want.
}
```

库具有以下特性：

- 没有状态变量
- 不能够继承或被继承
- 不能接收以太币
- 不可以被销毁

Using For

`using A for B;` 可用于附加库函数（从库 `A`）到任何类型（`B`）

`using A for *;` 的效果是，库 `A` 中的函数被附加在任意的类型上，这个类型可以使用A内的函数。

```
pragma solidity >=0.6.0 <0.9.0;

// 这是和之前一样的代码，只是没有注释。
struct Data { mapping(uint => bool) flags; }

library Set {

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
        return false; // 已经存在
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // 不存在
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      public
      view
      returns (bool)
  {
      return self.flags[value];
  }
}

contract C {
    using Set for Data; // 这里是关键的修改
    Data knownValues;
```

```
    function register(uint value) public {
        // Here, all variables of type Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
        // 这里， Data 类型的所有变量都有与之相对应的成员函数。
        // 下面的函数调用和 `Set.insert(knownValues, value)` 的效果完全相同。
        require(knownValues.insert(value));
    }
}
```

引用存储变量或者 internal 库调用 是唯一不会发生拷贝的情况。