**智能合约入门参考**

目录

[1 概述 2](#_Toc517448939)

[2 solidity 数据类型 2](#_Toc517448940)

[2.1 值类型(Value Type) 2](#_Toc517448941)

[2.2引用类型(Reference Types) 2](#_Toc517448942)

[2.3数据位置 3](#_Toc517448943)

[2.3.1代码测试结果 3](#_Toc517448944)

[2.3.2参考： 4](#_Toc517448945)

[3合约相关部分需要注意的点 7](#_Toc517448946)

[3.1 合约构造函数 7](#_Toc517448947)

[3.2 合约析构函数 safedestruct 8](#_Toc517448948)

[3.3 Modifier函数 8](#_Toc517448949)

[3.4 继承，重载 9](#_Toc517448950)

[3.5 库 11](#_Toc517448951)

[3.6 payable 12](#_Toc517448952)

[3.7 fallback函数 12](#_Toc517448953)

[3.8 send/transfer区别 13](#_Toc517448954)

[4 特殊的变量和函数 13](#_Toc517448955)

[4.1块和交易属性 13](#_Toc517448956)

[4.2数学和加密功能： 14](#_Toc517448957)

[4.3合约相关的: 15](#_Toc517448958)

[5 Truffle 16](#_Toc517448959)

[6 remix 16](#_Toc517448960)

[7 智能合约的几种设计模式 17](#_Toc517448961)

[7.1 自毁合约 17](#_Toc517448962)

[7.2 工厂合约 18](#_Toc517448963)

[7.3 名称注册表 19](#_Toc517448964)

[7.4 映射表迭代器 20](#_Toc517448965)

[7.5 参考 21](#_Toc517448966)

[附录： 21](#_Toc517448967)

# 1 概述

Solidity 是一门面向合约的、为实现智能合约而创建的高级编程语言。这门语言受到了 C++，Python 和 Javascript 语言的影响，设计的目的是能在以太坊虚拟机（EVM）上运行。Solidity 是静态类型语言，支持继承、库和复杂的用户定义类型等特性。

# 2 solidity 数据类型

## 2.1 值类型(Value Type)

布尔(Booleans)

整型(Integer)

地址(Address)

定长字节数组(fixed byte arrays)

有理数和整型(Rational and Integer Literals)

枚举类型(Enums)

函数(Function Types)

值类型传值时，会临时拷贝一份内容出来，而不是拷贝指针，当修改新的变量时，不会影响原来的变量的值

## 2.2引用类型(Reference Types)

不定长字节数组（bytes）

字符串（string）

数组（Array）

结构体（Struts）

引用类型，赋值时可以值传递，也可以引用即地址传递，如果是值传递，和基本值类型一样，修改新变量时，不会影响原来的变量值，如果是引用传递，那么当修改新变量时，原来变量的值会跟着变化，这是因为新旧变量同时指向同一个地址的原因。

## 2.3数据位置

### 2.3.1代码测试结果





### 2.3.2参考：

在使用solidity开发以太坊智能合约时，可以为变量声明memory和storage关键字。那么，它们有什么不同之处？如果在声明合约变量时没有使用memory关键字，Solidity会尝试在storage中存储这个变量。

memory和storage的不同

根据Solidity首席工程师Chriseth的说法：“**你可以把storage想像成一个大数组，它有自己的结构，这个结构是由你的合约中的状态变量所决定的，因此在运行时不能改变**”。这就是说，storage的结构是在合约部署创建时，根据你的合约中状态变量的声明，就固定下来了，并且不能在将来的合约方法调用中改变这个结构。但是，storage中的内容是可以通过交易来改变的。这些交易调用因此将修改合约的状态，这也是为什么合约中的变量被称为状态变量的原因。因此在合约层面声明的一个uint8类型的storage变量， 它的值可以修改为任何0-255之间的有效uint8值，但是该变量在storage结构中的位置始终不会变化。

**函数中的变量**

如果你在合约函数中声明变量时没有使用memory关键字，那么solidity将会尝试使用storage结构，目前来讲，这样做可以通过编译，但是可能导致不可预期的结果。memory关键字告诉solidity应当在该函数运行时为变量创建一块空间，使其大小和结构满足函数运行的需要。

**在合约层面你不能为变量应用memory关键字。**

引用类型是一个复杂类型，占用的空间通常超过256位， 拷贝时开销很大，因此我们需要考虑将它们存储在什么位置，是memory（内存中，数据不是永久存在）还是storage（永久存储在区块链中），所有的复杂类型如数组(arrays)和数据结构(struct)有一个额外的属性：数据的存储位置（data location）。可为memory和storage。

根据上下文的不同，大多数时候数据位置有默认值，也通过指定关键字storage和memory修改它。

函数参数（包含返回的参数）默认是memory。

局部复杂类型变量（local variables）和 状态变量（state variables） 默认是storage。

**局部变量**：局部作用域（越过作用域即不可被访问，等待被回收）的变量，如函数内的变量。

**状态变量**：合约内声明的公有变量

还有一个存储位置是：calldata，用来存储函数参数，是只读的，不会永久存储的一个数据位置。外部函数的参数（不包括返回参数）被强制指定为calldata。效果与memory差不多。

数据位置指定非常重要，因为他们影响着赋值行为。

在memory和storage之间或与状态变量之间相互赋值，总是会创建一个完全独立的拷贝。

而将一个storage的状态变量，赋值给一个storage的局部变量，是通过引用传递。所以对于局部变量的修改，同时修改关联的状态变量。

另一方面，将一个memory的引用类型赋值给另一个memory的引用，不会创建拷贝（即：memory之间是引用传递）。

强制的数据位置(Forced data location)

外部函数(External function)的参数(不包括返回参数)强制为：calldata

状态变量(State variables)强制为: storage

默认数据位置（Default data location）

函数参数及返回参数：memory

复杂类型的局部变量：storage

**深入分析**

storage 存储结构是在合约创建的时候就确定好了的，它取决于合约所声明状态变量。但是内容可以被（交易）调用改变。

Solidity 称这个为状态改变，这也是合约级变量称为状态变量的原因。也可以更好的理解为什么状态变量都是storage存储。

memory 只能用于函数内部，memory 声明用来告知EVM在运行时创建一块（固定大小）内存区域给变量使用。

storage 在区块链中是用key/value的形式存储，而memory则表现为字节数组

**关于栈（stack）**

EVM是一个基于栈的语言，栈实际是在内存(memory)的一个数据结构，每个栈元素占为256位，栈最大长度为1024。

值类型的局部变量是存储在栈上。

**不同存储的消耗（gas消耗）：**

storage 会永久保存合约状态变量，开销最大

memory 仅保存临时变量，函数调用之后释放，开销很小

stack 保存很小的局部变量，几乎免费使用，但有数量限制。

**存储中状态变量的布局:**

静态尺寸大小的变量（除了映射和动态尺寸大小的数组类型（的其他类型变量））在存储中，是从位置0连续存储。如果可能的话，不足32个字节的多个条目被紧凑排列在一个单一的存储块，参见以下规则：

在存储块中的第一项是存储低阶对齐的。

基本类型只使用了正好存储它们的字节数。

如果一个基本类型不适合存储块的剩余部分，则移动到下一个存储块中。

结构和数组的数据总是开始一个新的块并且占整个块（根据这些规则，结构或数组项都是紧凑排列的）。

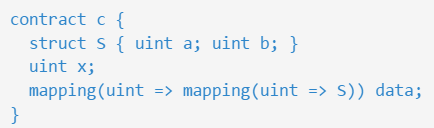
结构和数组元素是一个接着一个存储排列的，就如当初它们被声明的次序。

由于无法预知的分配的大小，映射和动态尺寸大小的数组类型（这两种类型）是使用sha3 计算来找到新的起始位置，来存放值或者数组数据。这些起始位置总是满栈块。

根据上述规则，映射或动态数组本身存放在（没有填满）的存储块位置p（或从映射到映射或数组递归应用此规则）。对于一个动态数组，存储块存储了数组元素的数目（字节数组和字符串是一个例外，见下文）。对于映射，存储块是未使用的（但它是需要的，因此，前后相邻的两个相同的映射，将使用一个不同的hash分布）。数组数据位于sha3(p)， 对应于一个映射key值k位于 sha3(k . p) （这里 . 是连接符）。如果该值又是一个非基本类型，位置的偏移量是sha3(k . p)。

如果bytes 和 string是短类型的，它们将和其长度存储在同一个存储块里。特别是：如果数据最长31字节，它被存储在高阶字节（左对齐）, 低字节存储length 2。 如果是长类型，主存储块存储 length 2 + 1, 数据存储在sha3(shot)。

**本合约片段如下：**

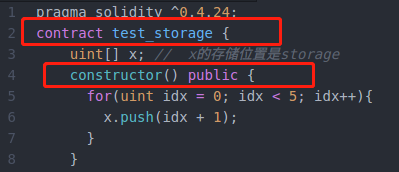


# 3合约相关部分需要注意的点

## 3.1 合约构造函数

合约构造函数不再使用function 加合约名，改为统一使用constructor。

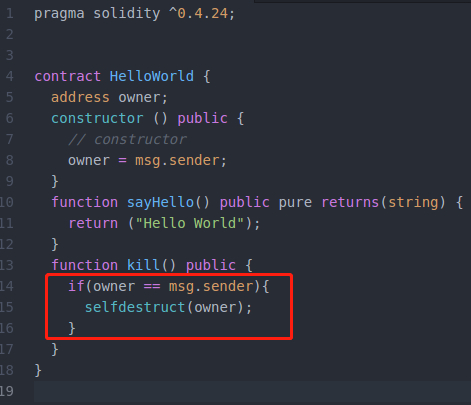
可在构造中记录合约的部署者：creator = msg.sender;



## 3.2 合约析构函数 safedestruct

selfdestruct(address recipient) 销毁当前合约，把剩余资金发送到指定地址。

调用kill后，合约销毁。



## 3.3 Modifier函数

1 modifier不能有返回值

2 modifyer可以带参数

3 如果代码中有return，则最终的函数直接返回，返回值为0或空

4 modifier中return之后的代码在函数返回后，再被执行

通过modifier可在具体函数代码执行前进行校验



## 3.4 继承，重载

继承通过代码备份实现，支持多重继承，来自父合约的代码总是复制到最终合约里。继承不节省gas。

继承链始于衍生最充分的合约，终于衍生最不充分的合约。

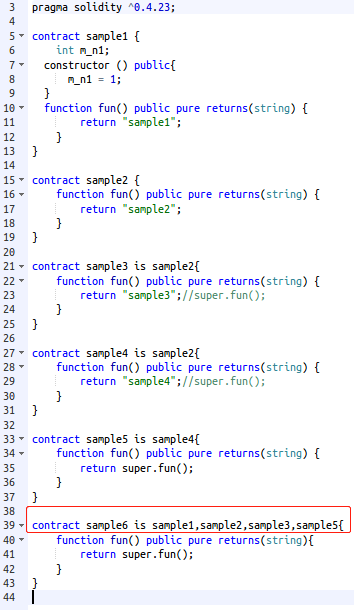
实际继承链顺序是：sample6，sample5，sample4，sample2，sample3，sample1

**继承顺序有很大的影响**

**基类排列顺序，父子关系，基类在前，子类在后，**

**执行时，从右往左查找基类函数**

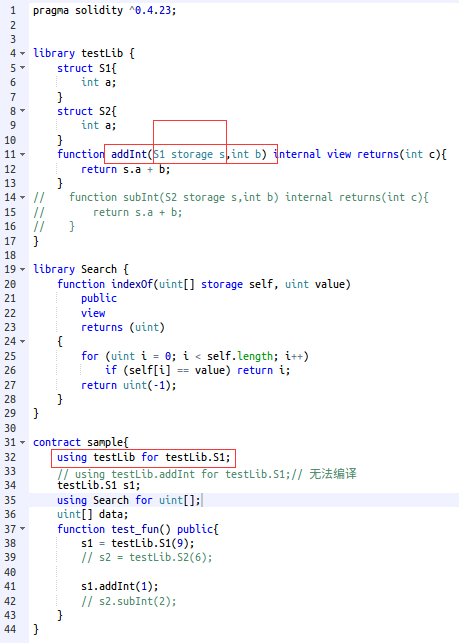
**调用sample6.fun返回：”sample4”**



## 3.5 库

库类似合约，只部署一次，供不同合约调用，相比继承，能节省gas；代码在调用合约中执行，this指向调用合约方法，只能调用合约的状态变量；库没有状态变量，不支持继承，不能接收币，可以包含结构和枚举；库部署后，知道其地址和源码即可调用；库可部分想象成基础合约，库里面带有内部可视性（internal/private）的函数不能被使用它的合约调用，此外库里的struct和enum被复制给调用这个库的合约，即：如果一个库只有内部函数或struct/enum，则不需要部署库，反正都会复制到合约里。库可用于给数据类型添加成员函数（using A for B,A的函数的第一个参数必须和B的数据类型一致）。

需要注意的是所有库调用都实际上是EVM函数调用。这意味着，如果你传的是memory类型的，或者是值类，那么仅会传一份拷贝，即使是self变量。变通之法就是使用存储类型的变量，这样就不会拷贝内容。



**Using for 如何使用:**

import是讲某个合约contract或者某个库lib导入到当前文件，它是using的前提；import后，当前文件内可以引用被引入文件内定义的library或contract。

using A for B,这里A通常是某个library里面定义的某个方法，B是某种数据类型，这句话是把A方法绑定到B类型上，相当于给B类型附加了一个A方法。（也有翻译为附着库的）

在上面的例子中，将LibContract里定义的方法绑定到所有的数据类型。但是一般我们不会在所有的类型实例上都去调用LibContract的方法，应该是要按需using的，这里偷懒就写\*。

在通俗一点的例子就是，

比如 using LibInt for uint，然后LibInt里面有定义一个toString方法。我们有一个uint a;那么可以这样调用a.toString()，toString方法在定义的时候，第一个参数会是一个uint类型的变量，表示调用者。

using A for B,A的函数的第一个参数必须和B的数据类型一致。

还有这个方法是可以重载的，你可以定义好几个同名的方法，但是第一个参数的类型不同，调用的时候自动的根据调用类型选择某一种方法。

## 3.6 payable

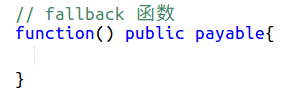
payable关键字，如果一个函数需要进行货币操作，必须要带上payable关键字，这样才能正常接收msg.value。

## 3.7 fallback函数

一个合约可以有唯一的未命名函数，称为回退函数（fallback function）。该函数不能有实参，不能返回任何值。如果其他函数都不能匹配给定的函数标识符，就在合约调用上执行回退函数。

当合约不用任何函数调用就接收以太币（即交易发送以太币给合约却不调用任何方法）时，也执行该函数。在此情况下，用于函数调用的gas通常很少（准确说是2300gas），所以回退函数尽可能便宜也很重要。

接收以太币但是却不定义回退函数的合约会抛出异常，把以太币发送回去。如果想让合约接收以太币，就必须要实现回退函数



## 3.8 send/transfer区别

**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。

**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。

**注意：**

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

# 4 特殊的变量和函数

有特殊的变量和函数总是存在于全局命名空间,主要用于提供关于blockchain的信息。

## 4.1块和交易属性

block.coinbase (address): :当前块的矿工的地址

block.difficulty (uint):当前块的难度系数

block.gaslimit (uint):当前块汽油限量

block.number (uint):当前块编号

block.blockhash (function(uint) returns (bytes32)):指定块的哈希值——最新的256个块的哈希值

block.timestamp (uint):当前块的时间戳

msg.data (bytes):完整的calldata

msg.gas (uint):剩余的汽油

msg.sender (address):消息的发送方(当前调用)

msg.sig (bytes4):calldata的前四个字节(即函数标识符)

msg.value (uint):所发送的消息中wei的数量

now (uint):当前块时间戳(block.timestamp的别名)

tx.gasprice (uint):交易的汽油价格

tx.origin (address):交易发送方(完整的调用链)

**注：**

msg的所有成员的值,包括msg.sender和msg.value可以在每个 external函数调用中改变。这包括调用库函数。

如果你想在库函数实现访问限制使用msg.sender, 你必须手动设置msg.sender作为参数。

注意谨慎使用block.timestamp, now and block.blockhash，因为他们都是有可能被篡改的。

写Solidity最大的不同在于，我们要随时计算好我们的gas消耗，方法的复杂度，变量类型的存储位置（memory，storage等等）都会决定gas的消耗量。？？？？？？？？？？？？？

使用event可以获得比storage更便宜的gas消耗。

总结一下event，就是如果你的Dapp客户端web3.js想调用智能合约内部的函数，则使用event作为桥梁，它能方便执行异步调用同时又节约gas消耗。

**注：**

由于所有块可伸缩性的原因，（所有）块的Hash值就拿不到。你只能访问最近的256块的Hash值,其他值为零。

## 4.2数学和加密功能：

addmod(uint x, uint y, uint k) returns (uint):

计算 (x + y) % k （按指定的精度，不能超过2\*\*256）

mulmod(uint x, uint y, uint k) returns (uint):

compute (x \* y) % k where the multiplication is performed with arbitrary precision and does not wrap around at 2256. （按指定的精度，不能超过2\*\*256）

计算compute (x \* y) % k

sha3(...) returns (bytes32):

计算（紧凑排列的）参数的Ethereum-SHA-3 的Hash值值

sha256(...) returns (bytes32):

计算（紧凑排列的）参数的SHA-256 的Hash值

ripemd160(...) returns (bytes20):

计算（紧凑排列的）参数的 RIPEMD-160 的Hash值

ecrecover(bytes32, byte, bytes32, bytes32) returns (address):

恢复椭圆曲线特征的公钥-参数为(data, v, r, s)

在上述中，“紧凑排列”，意思是没有填充的参数的连续排列，也就是下面表达式是没有区别的

sha3("ab", "c")

sha3("abc")

sha3(0x616263)

sha3(6382179)

sha3(97, 98, 99)

如果需要填充，要用显示的形式表示： sha3(“x00x12”) 和 sha3(uint16(0x12))是相同的。

在一个私有的blockchain里，你可能（在使用）sha256, ripemd160 或 ecrecover (的时候) 碰到"Out-of-Gas"（的问题） 。原因在于这个仅仅是预编译的合约，合约要在他们接到的第一个消息以后才真正的生成（虽然他们的合约代码是硬编码的）。对于没有真正生成的合约的消息是非常昂贵的，这时就会碰到“Out-of-Gas”的问题。 这一问题的解决方法是事先把1wei 发送到各个你当前使用的各个合约上。这不是官方或测试网的问题。

## 4.3合约相关的:

this (current contract’s type):当前的合约,显示可转换地址

selfdestruct(address)::销毁当前合约,其资金发送给指定的地址

此外,当前合同的所有函数均可以被直接调用（包括当前函数）。

# 5 Truffle

Truffle是一个世界级的开发环境，测试框架，以太坊的资源管理通道，致力于让以太坊上的开发变得简单，Truffle有以下：

1)内置的智能合约编译，链接，部署和二进制文件的管理。

2)快速开发下的自动合约测试。

3)脚本化的，可扩展的部署与发布框架。

4)部署到不管多少的公网或私网的网络环境管理功能

5)使用EthPM&NPM提供的包管理，使用ERC190标准。

6)与合约直接通信的直接交互控制台（写完合约就可以命令行里验证了）。

7)可配的构建流程，支持紧密集成。

8)在Truffle环境里支持执行外部的脚本。

sudo npm install -g solc

sudo npm install -g ethereumjs-testrpc truffle

**注意：**

1 通过truffle创建合约文件，版本号，构造函数，可见性，都是旧版本形式，需要修改，最新支持solidity 0.4.24

0.4.4

Truffle create conctract llll

2 编译时强制重新编译所有：truffle compile --compile-all

3 如果部署过一次，修改后不强制重新部署，链上的合约不一定是最新的，测试可能不对，需强制重新部署： truffle migrate -reset

# 6 remix

Remix 是以太坊提供的一个开发Solidity智能合约的网络版开发软件。合约的开发者在Remix里提供的JavaScript虚拟机上开发，调试好合约后，可以发布到以太坊，或者任何支持Solidity智能合约的区块链上。

**安装：**

sudo npm install node-gyp

sudo npm install remix-ide -g

sudo npm install remixd -g

**执行出错，修改**：

/usr/local/lib/node\_modules/remix-ide/node\_modules/scrypt/index.js

将require("./build/Release/scrypt")改为require("scrypt")



**注：**remix目前支持solidity 版本0.4.23

# 7 智能合约的几种设计模式

## 7.1 自毁合约

合约自毁模式用于终止一个合约，这意味着将从区块链上永久删除这个合约。 一旦被销毁，就不可能 调用合约的功能，也不会在账本中记录交易。

现在的问题是：“为什么我要销毁合约？”。

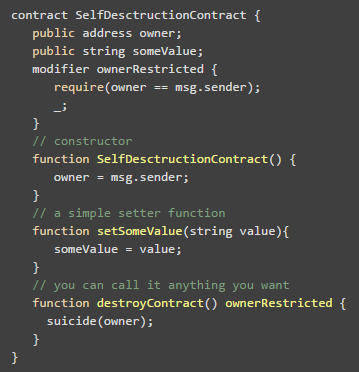
有很多原因，比如某些定时合约，或者那些一旦达到里程碑就必须终止的合约。 一个典型的案例 是贷款合约，它应当在贷款还清后自动销毁；另一个案例是基于时间的拍卖合约，它应当在拍卖结束后 终止 —— 假设我们不需要在链上保存拍卖的历史记录。

在处理一个被销毁的合约时，有一些需要注意的问题：

合约销毁后，发送给该合约的交易将失败

任何发送给被销毁合约的资金，都将永远丢失

为避免资金损失，应当在发送资金前确保目标合约仍然存在，移除所有对已销毁合约的引用。

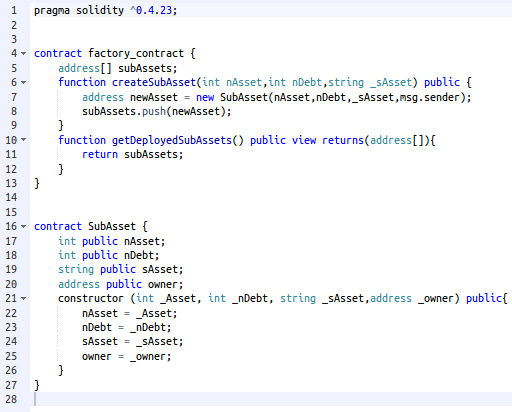


## 7.2 工厂合约

工厂合约用于创建和部署“子”合约。 这些子合约可以被称为“资产”，可以表示现实生活中的房子或汽车。

工厂用于存储子合约的地址，以便在必要时提取使用。 你可能会问，为什么不把它们存在Web应用数据库里？ 这是因为将这些地址数据存在工厂合约里，就意味着是存在区块链上，因此更加安全，而数据库的损坏 可能会造成资产地址的丢失，从而导致丢失对这些资产合约的引用。 除此之外，你还需要跟踪所有新 创建的子合约以便同步更新数据库。

工厂合约的一个常见用例是销售资产并跟踪这些资产（例如，谁是资产的所有者）。 需要向负责部署资产的 函数添加payable修饰符以便销售资产

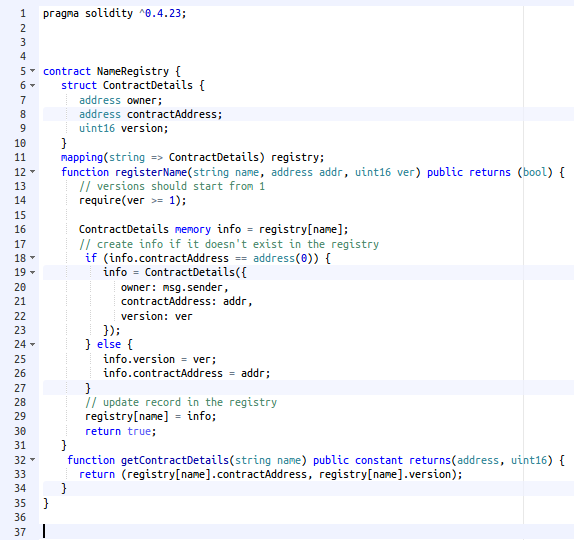


## 7.3 名称注册表

假设你正在构建一个依赖与多个合约的DApp，例如一个基于区块链的在线商城，这个DApp使用了 ClothesFactoryContract、GamesFactoryContract、BooksFactoryContract等多个合约。

现在想象一下，将所有这些合约的地址写在你的应用代码中。 如果这些合约的地址随着时间的推移而变化，那该怎么办？

这就是名称注册表的作用，这个模式允许你只在代码中固定一个合约的地址，而不是数十、数百甚至数千个 地址。它的原理是使用一个合约名称 => 合约地址的映射表，因此可以通过调用getAddress("ClothesFactory") 从DApp内查找每个合约的地址。 使用名称注册表的好处是，即使更新那些合约，DApp也不会受到任何影响，因为 我们只需要修改映射表中合约的地址。

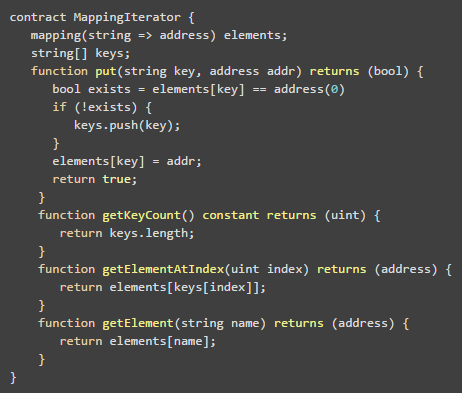


**注：**

名称注册表和工厂合约结合起来使用，根据名称查到工厂合约的地址，再根据工厂合约创造具体的合约

## 7.4 映射表迭代器

很多时候我们需要对一个映射表进行迭代操作 ，但由于Solidity中的映射表只能存储值， 并不支持迭代，因此映射表迭代器模式非常有用。 需要指出的是，随着成员数量的增加， 迭代操作的复杂性会增加，存储成本也会增加，因此请尽可能地避免迭代。



**群星项目的智能合约，有类似功能**

## 7.5 参考

部分solidity合约代码参考

<https://github.com/OpenZeppelin/openzeppelin-solidity>

wanchain token智能合约参考

https://github.com/wanchain/wanchain-token

# 附录：

1 solidity 中 Var 不建议使用

2 Throw 不建议使用，建议使用： "revert()", "require()" and "assert()".

3 错误处理

assert(bool condition): 如果条件不满足，则抛出 - 用于内部错误。

require(bool condition): 如果条件不满足，则抛出 - 用于输入或外部组件中的错误。

revert(): 中止执行并恢复状态更改

4 pure / view

view 修饰的函数 ，只能读取storage变量的值，不能写入。

pure 修饰的函数 ， 不能对storage变量进行读写。

5 对合约状态变量有修改的函数，不直接返回值，需要通过其他途径查看最终结果