**Solidity语言**

区块链1.0是以太坊，而区块链2.0是以太坊。区块链2.0和1.0的最大区别就是2.0支持智能合约。智能合约是运行在可复制、共享的账本上的计算机程序，可以处理信息，接收、存储和发送价值。



Solidity是一种智能合约高级语言，运行在以太坊虚拟机（EVM）上。它是一种面向对象的编程语言，其语法和C++、Javascript类似。但作为一种运行在以太坊区块链网络上的智能合约语言，其有如下几种特点:

* 以太坊底层是基于账户，而非UTXO的。其有一个特殊的Address数据类型，用于定位用户账号、合约账号以及合约代码。
* 在语言层面直接支持支付。使用关键字payable可让语言支持支付功能。
* 语言中数据的每一个状态都可以永久存储在以太坊区块链上。函数中定义的变量存储在内存中，而在合约对象的成员变量存储在区块链中。

# 1 智能合约示例

首先，让我们来快速熟悉一下智能合约的编写、编译、部署和调用的过程。本书均使用Solidity编写智能合约，本节使用Remix进行智能合约的编译、部署和调用。

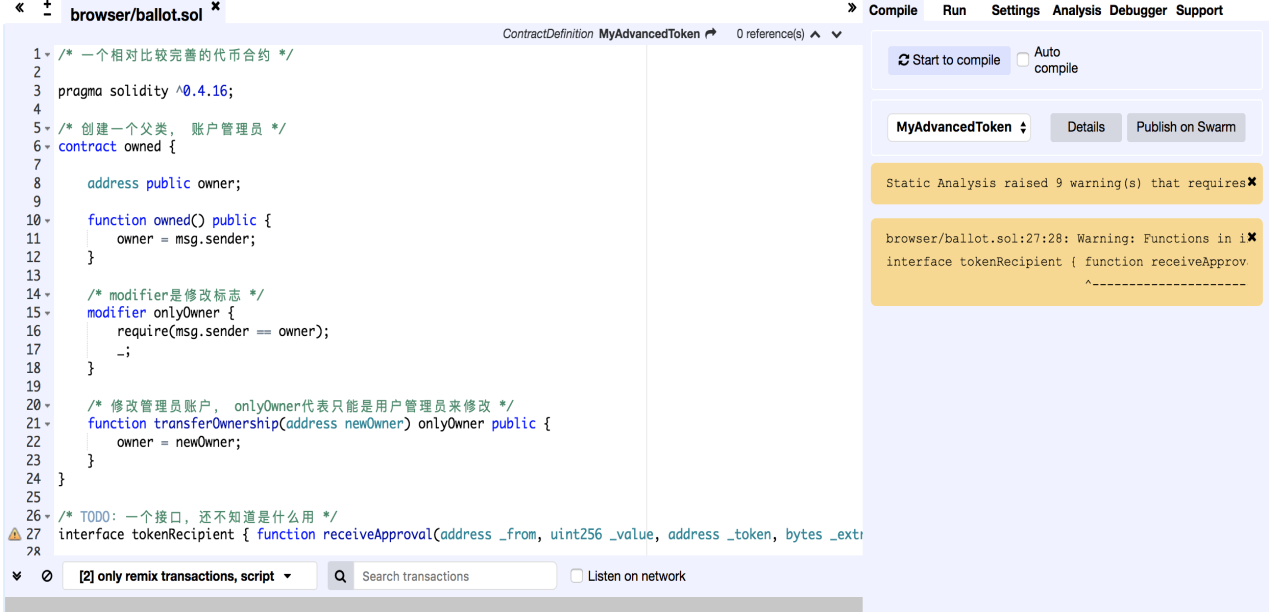
## 1.1 编写智能合约

以下智能合约代码实现了两个函数，函数Hello()用于设置欢迎语句内容，函数Say()用于获取欢迎语句内容。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Hello {  string greeting;  function Hello(string \_greeting) public {  greeting = \_greeting;  }  function Say() constant public returns (string){  return greeting;  }  } |

## 1.2 编译智能合约

使用浏览器打开网址http://remix.ethereum.org，并将刚才编写的智能合约代码粘贴到remix的代码编辑区域，并点击右侧的编译按钮进行编译。



## 1.3 部署智能合约

编译完成后，接下来就可以将智能合约部署到私有链上。

## 1.4 调用智能合约

部署完成后，可以得到智能合约的地址，通过合约地址又可以获得该合约实现的所有函数列表。

# 2 数据类型

## 2.1 整数类型

整数类型简称为整型。Solidity语言的整型的细分类型非常的丰富，整型变量的步长以8递增，有符号整型支持从int8，int16，int24，int32…逐步递增到int256, 无符号整型支持从uint8，uint16，uint24，uint32…逐步递增到uint256。另外，需要注意的是，int和uint默认分别代表int256和uint256。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Calc {  function add(int x, int y) public returns(int) {  return x + y;  }  } |

## 2.2 小数类型

## 2.3 布尔类型

使用bool表示布尔类型，其取值范围只有两个值:真（true）和假（false）。通常用来判断条件是否成立。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Cmp {  function isEqual(int x, int y) public returns(bool) {  return (x == y);  }  } |

## 2.4 地址类型

使用address表示地址类型，其可用于定位用户账号、合约账号以及合约代码。以太坊地址占用20个字节（160bit）空间，因此，可使用uint160编码。

**1.当地址作为用户地址时**

当地址作为普通用户账号时，我们可通过地址获取当前用户以太币的余额。

|  |
| --- |
| pragma solidity ^0.4.18;  contract UAccount {  function getBalance(address addr) returns(uint256) {  return addr.balance;  }  } |

**2.当地址作为合约账号时**

当地址作为合约账号时，我们可通过地址获取当前合约以太币的余额。

|  |
| --- |
| pragma solidity ^0.4.18;  contract CAccount {  function getBalance() returns(uint256) {  return this.balance;  }  } |

对于合约来说，地址代表的就是合约本身，合约对象默认继承自地址对象，故其内部有地址的属性。

**3.当地址作为合约代码时**

当地址作为合约代码时，我们可通过该地址获得该合约定义的函数，并可调用合约定义的函数。

例如:我们部署某合约后，得到合约代码地址”0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF”。再将该地址拷贝粘贴到Remix的合约地址栏，就可以获得合约函数列表。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Calc {  // 加法运算  function add(int x, int y) returns(int) {  return (x + y);  }  // 减法运算  function sub(int x, int y) returns(int) {  return (x - y);  }  } |

## 2.5 数组类型

数组是存放类型相同的对象容器，这些对象本身没有名字，需要通过其所在位置访问。可以在声明时可以指定数组长度，也可以不指定数组长度。

对于storage的数组而言，元素类型可以是任意数据类型，如: 元素类型可以是数组、字典类型、数据结构、其他基础类型等。对于memory的数组而言，如果是函数内部使用的数组，则其元素类型可以是任意数据类型；如果外部可访问函数的参数中有数组，则其元素类型不能是字典类型，只能是支持ABI的类型。

一个类型为T，长度为k的数组，可以声明为T[k]，而一个变长的数组则声明为T[]。

**1.创建数组**

可使用new关键字创建一个数组。对于storage型数组，创建时虽然指定了数组长度，但是可通过修改\*.length的值来调整数组长度。而对于memory型数组，创建时指定了数组长度，长度便不可修改。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Array {  uint[] a; // 声明storage型数组  function f1() {  // 创建storage型长度为5的数组  a = new uint[](5);  // 合法: 能通过修改\*.length的值来调整数组长度  a.length = 10;  a[9] = 99;  }  function f2() {  // 创建memory型长度为5的数组  uint[] memory b = new uint[](5);  // 非法: 不能通过修改\*.length的值来调整数组长度  b.length = 10;  }  } |

**2.数组字面量**

数组字面量是指以表达式方式隐式声明的一个数组，并作为一个数组变量的使用方式。通过数组字面量创建是memory型数组，且是定长数组。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Array {  function f() {  g([uint(1), 2, 3]); // 数组字面量  }  function g(uint[3] data) {  // ...  }  } |

如果未进行类型转化，数组字面量的元素类型则刚好能存储的数组元素的数据类型。比如: 以上代码中[1, 2, 3]，只需uint8类型即可存储。函数g()要求数组参数为uint类型，需使用uint(1)进行类型转换。

**3.length属性**

可通过数组\*.length属性获取或设置数组的长度。

* 当为storage型数组时，可通过\*.length属性获取数组长度，同时可通过对其赋值调整数组长度。
* 当为memory型数组时，可通过\*.length属性获取数组长度，但不可通过对其赋值调整数组长度。

|  |
| --- |
| *注意: 对于memory型数组，可通过参数灵活指定数组长度。但数组一旦被创建，其长度不可调整。对于变长数组，可通过参数在编译期间指定数组长度。* |

**4.push方法**

变长的storage数组可使用push()方法将一个新元素附加到数组末端，返回值为数组当前长度。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Array {  uint[] list;  function f() {  uint len = list.push(1);  }  } |

|  |
| --- |
| *注意:*   * *对于memory型数组，不支持push()方法。* * *对于bytes数据类型，支持push()方法。* * *对于string数据类型，不支持push()方法。* |

**5.多维数组**

与其他语言(如:C/C++/Java)一样Solidity也支持多维数组。一个类型为T的二维数组，行为m，列为n，则可表示为T[n][m]；一个类型为T的二维数组，行为m，列变长，则可表示为T[][m]。需要注意的是，Solidity多维数组各维长度声明的顺序正好与其他语言相反。

虽然Solidity多维数组各维长度声明的顺序正好与其他语言相反，但是访问多维数组成员时，各维顺序又与其他语言相同。例如: 访问多维数组T[n][m](m行 n列)的第5行第6列的元素，则可使用T[4][5]进行访问。

**6.数组限制**

当前在外部函数中，不能使用多维数组。并且因为EVM的限制，不能通过外部函数返回变长数组中的数据元素。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Array {  function f() returns (uint[]) {  return new uint[](1);  }  } |

在上面的代码中，函数f()返回的是变长数组。使用Solidity的外部函数的访问方式，将会出现如下编译错误提示信息:

|  |
| --- |
| Return argument type inaccessible dynamic type is not implicitly convertible to expected type. |

对于以上问题可使用返回定长数组的方式来解决。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Array {  function f() returns (uint[1]) {  return [uint[](1)];  }  } |

|  |
| --- |
| *注意: bytes和string是特殊的数组。*  *1.bytes类似byte[]，但在外部函数作为参数调用中，会使用压缩打包，更省空间。所以，尽量使用bytes类型。*  *2.string类似bytes，但不能提供长度和按序号的访问方式。*  *3.bytes与string可以相互转换。可将字符串str通过bytes(str)转换为bytes类型。* |

## 2.6 字符串类型

字符串(string)是特殊的数组类型，但不能按照序号的方式访问其字符元素。string与bytes可以相互转换，可将字符串str通过bytes(str)转换为bytes类型。

|  |
| --- |
| pragma solidity ^0.4.18;  contract String {  function f(){  string memory str;  str = “This is just a test!”;  // string转化为bytes类型  bytes memory b = bytes(str);  }  } |

## 2.7 字节数组

字节数组是特殊的数组类型，其分为定长字节数组和变长字节数组。定长字节数组包括以步长1递增的byte1,byte2,…,byte32类型。byte默认表示byte1类型。变长字节数组的长度不固定，使用bytes表示。

字节数组支持通过序号的方式访问字节元素。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Byte {  function f(){  bytes memory b;  str = “This is just a test!”;  // string转化为bytes类型  bytes memory b = bytes(str);  }  } |

|  |
| --- |
| *原则:*  *1.bytes用来存储任意长度的字节数据，string用来存储任意长度的UTF-8编码的字符串数据。*  *2.如果字节长度可以确定，尽量使用定长字节类型（byte1~byte32），可以节省更多空间。* |

## 2.8 字典类型

字典类型又称为映射类型，其是一种键值对的映射关系存储结构。字典定义形式为:mapping(K => V)。K表示键，其类型允许除字典外的所有类型。V表示值，其类型可为任何数据类型，无任何限制。

可以将字典视为一个哈希表，但字典中并不存储键的数据，仅仅存储它kecak256哈希值。因此，字典没有长度、键集合（或列表）、值集合（或列表）概念。如果查找的键在字典中不存在，将会返回一个值类型的默认值（二进制表示的零）。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Mapping {  mapping(address => uint) public scores;  function update(uint score) returns(address addr) {  scores[msg.sender] = score;  return msg.sender;  }  } |

## 2.9 十六进制字面量

十六进制字面量以关键字hex开头，后面紧跟使用单引号或双引号包裹的字符串。一个hex元素由[0-9A-F]两个字符组成，占用一个字节空间。与其他语言类似，任何数据类型(数字、字符串等)均可通过十六进制来表示。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Hex {  function f() returns(bytes4, bytes1, bytes1) {  bytes4 b = hex”1A2B3C4D”;  return (b, b[0], b[1]);  }  } |

## 2.10 枚举类型

枚举类型是一种用户自定义类型，可与整数进行显示转换，但不能隐式转换。进行显示转换时，将会进行数值范围的检测。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Enum {  enum Action {LEFT, RIGHT, DOWN, UP}  Action action;  Action const DEFAULT\_ACTION = Action.UP;  // 设置  function setAction(Action a) returns(Action) {  action = a;  return action;  }  // 获取  function getAction() returns(Action) {  return action;  }  // 获取默认值  function getDefaultAction() returns(uint) {  return uint(action);  }  } |

## 2.11 引用类型

## 2.12 数据位置

智能合约中的变量拥有数据位置的属性: memory, storage和calldata。

|  |  |
| --- | --- |
| **数据位置** | **描述** |
| memory | 数据存储在内存中，不会永久存在。  内部函数的参数（包括返回参数）的默认数据位置 |
| storage | 数据存储在区块链中，将会永久存在。  局部变量和状态变量的默认数据位置。 |
| 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）强制设定为该数据位置，不可调整。 |

根据上下文的不同，大数据时候数据位置有默认值，但可通过使用关键字storage和memory调整数据位置。

|  |
| --- |
| *变量说明:*  *1.状态变量：合约内声明的公有变量。*  *2.局部变量：局部作用域的变量，越过作用域即不可被访问，等待被回收。如：函数内定义的变量。* |

|  |
| --- |
| *注意事项：*  *1.在memory和storage之间相互赋值，将会进行值拷贝。*  *2.状态变量之间相互赋值，将会进行值拷贝。*  *3.将storage的状态变量，赋值给storage局部变量，将会使用引用传递。对该局部变量的修改，也将同时修改对应状态变量的值。*  *4.将一个memory的引用类型，赋值给另一个memory的引用类型，不会进行值拷贝。*  *5.不能将memory赋值给局部变量。*  *6.对于值类型，进行赋值时总会进行值拷贝。* |

|  |
| --- |
| pragma solidity ^0.4.18;  contract DataLocation {  uint[] x; // x的存储位置是storage  // 参数m的存储位置是 memory  function f(uint[] m) public {  x = m; // 从 memory 复制到 storage  var y = x; // storage 引用传递局部变量y（y 是一个 storage 引用）  y[7]; // 返回第8个元素  y.length = 2; // x同样会被修改  delete x; // y同样会被修改  // 错误， 不能将memory赋值给局部变量  // y = m;  // 错误，不能通过引用销毁storage  // delete y;  g(x); // 引用传递， g可以改变x的内容  h(x); // 拷贝到memory， h无法改变x的内容  }  function g(uint[] storage storageArray) internal {}  function h(uint[] memoryArray) public {}  } |

|  |
| --- |
| pragma solidity ^0.4.18;  contract DataLocation {  uint[] x; // the data location of x is storage  // the data location of memoryArray is memory  function f(uint[] memoryArray) {  x = memoryArray; // works, copies the whole array to storage  var y = x; // works, assigns a pointer, data location of y is storage  y[7]; // fine, returns the 8th element  y.length = 2; // fine, modifies x through y  delete x; // fine, clears the array, also modifies y  // The following does not work; it would need to create a new temporary /  // unnamed array in storage, but storage is "statically" allocated:  // y = memoryArray;  // This does not work either, since it would "reset" the pointer, but there  // is no sensible location it could point to.  // delete y;  g(x); // calls g, handing over a reference to x  h(x); // calls h and creates an independent, temporary copy in memory  }  function g(uint[] storage storageArray) internal {}  function h(uint[] memoryArray) {}  } |

## 2.13 函数类型

## 2.14 结构体

## 2.15 类型推断

## 2.16 类型转换

# 4 单位转换

## 4.1 时间单位

智能合约当前支持以下几种时间单位，相互之间可以按照规定比例进行转换。

|  |  |  |
| --- | --- | --- |
| **序号** | **单位** | **备注** |
| 01 | seconds | 秒（1 == 1 seconds） |
| 02 | minutes | 分（1 minutes == 60 seconds） |
| 03 | hours | 时（1 hours == 60 minutes） |
| 04 | days | 天（1 days == 24 hours） |
| 05 | weeks | 周（1 weeks == 7 days） |
| 06 | years | 年（1 years == 365days） |

在使用时间单位时，我们需要注意以下两点内容:

* 日期转换: 如果需要将这些单位进行日期计算时，这时我们需要特别的注意，因为有的年份是365天，而有的年份是366天。
* 变量后缀: 时间单位不能作为变量的后缀。如果想说明变量的时间单位，可让该变量乘以’1 时间单位’的方式来表达。

|  |
| --- |
| pragma solidity ^0.4.18;  contract Time {  function time(uint now, uint day) uint public {  return now + day \* 1 days;  }  } |

|  |
| --- |
| *注意: 并不是每分钟都是60秒，还存在闰秒的情况。也就是说当出现闰秒的情况时，这1分钟将会61秒。* |

## 4.2 货币单位

# 5 运算符

# 6 控制结构

# 7 函数

## 7.1 函数结构

## 7.2 入参出参

## 7.3 权限控制

## 7.4 函数调用

## 7.5 访问函数

## 7.6 回退函数

## 7.7 函数修改器

# 8 高级特性

## 8.1 特殊变量

## 8.2 库

## 8.3 接口

## 8.4 抽象

## 8.5 继承

## 8.6 事件

## 8.7 异常

## 8.8 内联汇编

## 8.9 创建合约实例