智能合约编写之Solidity的高级特性

原创 毛嘉宇 FISCO BCOS开源社区 3月12日

第1场 | 智能合约初探：概念与演变

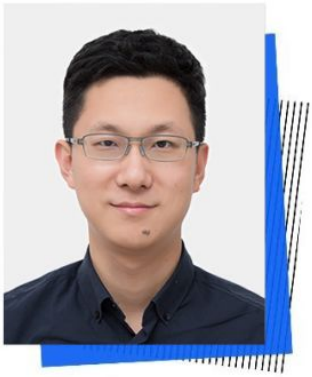
第2场 | 智能合约编写之Solidity的基础特性

FISCO BCOS

系列专题 | 超话区块链之智能合约专场

编写篇：智能合约编写之 Solidity 的高级特性

作者：毛嘉宇



毛嘉宇

FISCO BCOS核心开发者

have fun

前言

FISCO BCOS使用了Solidity语言进行智能合约开发。Solidity是一门面向区块链平台设计、图灵完备的编程语言，支持函数调用、修饰器、重载，事件、继承和库等多种高级语言的特性。

在本系列前两篇文章中，介绍了智能合约的概念与Solidity的基础特性。本文将介绍Solidity的一些高级特性，帮助读者快速入门，编写高质量、可复用的Solidity代码。

合理控制函数和变量的类型

基于最少知道原则（Least Knowledge Principle）中经典面向对象编程原则，一个对象应该对其他对象保持最少的了解。优秀的Solidity编程实践也应符合这一原则：每个合约都清晰、合理地定义函数的可见性，暴露最少的信息给外部，做好对内部函数可见性的管理。

同时，正确地修饰函数和变量的类型，可给合约内部数据提供不同级别的保护，以防止程序中非预期的操作导致数据产生错误；还能提升代码的可读性与质量，减少误解和bug；更有利于优化合约执行的成本，提升链上资源的使用效率。

守住函数操作的大门：函数可见性

Solidity有两种函数调用方式：

- 内部调用：又被称为『消息调用』。常见的有合约内部函数、父合约的函数以及库函数的调

用。（例如，假设A合约中存在f函数，则在A合约内部，其他函数调用f函数的调用方式为f()。）

- 外部调用：又被称为『EVM调用』。一般为跨合约的函数调用。在同一合约内部，也可以产生外部调用。（例如，假设A合约中存在f函数，则在B合约内可通过使用A.f()调用。在A合约内部，可以用this.f()来调用。）。

函数可以被指定为 external ， public ， internal 或者 private标识符来修饰。

标识符	作用
external	不可内部调用，在接收大量数据时更为高效。
public	同时支持内部和外部调用。
internal	只支持内部调用。
private	仅在当前合约使用，且不可被继承。

基于以上表格，我们可以得出函数的可见性 public > external > internal > private。

另外，如果函数不使用上述类型标识符，那么默认情况下函数类型为 public。

综上所述，我们可以总结一下以上标识符的不同使用场景：

- public，公有函数，系统默认。通常用于修饰**可对外暴露的函数，且该函数可能同时被内部调用。**
- external，外部函数，推荐**只向外部暴露**的函数使用。当函数的某个参数非常大时，如果显式地将函数标记为external，可以强制将函数存储的位置设置为calldata，这会节约函数执行时所需存储或计算资源。
- internal，内部函数，推荐所有合约内**不对合约外暴露**的函数使用，可以避免因权限暴露被攻击的风险。
- private，私有函数，在极少数严格保护合约函数**不对合约外部开放且不可被继承**的场景下使用。

不过，需要注意的是，无论用何种标识符，即使是private，整个函数执行的过程和数据是对所有

节点可见，其他节点可以验证和重放任意的历史函数。实际上，整个智能合约所有的数据对区块链的参与节点来说都是透明的。

刚接触区块链的用户常会误解，在区块链上可以通过权限控制操作来控制和保护上链数据的隐私。

这是一种错误的观点。事实上，在区块链业务数据未做特殊加密的前提下，区块链同一账本内的所有数据经过共识后落盘到所有节点上，链上数据是全局公开且相同的，智能合约只能控制和保护合约数据的执行权限。

如何正确地选择函数修饰符是合约编程实践中的『必修课』，只有掌握此节真谛方可自如地控制合约函数访问权限，提升合约安全性。

对外暴露最少的必要信息：变量的可见性

与函数一样，对于状态变量，也需要注意可见性修饰符。状态变量的修饰符默认是internal，不能设置为external。此外，当状态变量被修饰为public，编译器会生成一个与该状态变量同名的函数。

具体可参考以下示例：

```
pragma solidity ^0.4.0;

contract TestContract {
    uint public year = 2020;
}

contract Caller {
    TestContract c = new TestContract();
    function f() public {
        uint local = c.year();
        //expected to be 2020
    }
}
```

这个机制有点像Java语言里lombok库所提供的@Getter注解，默认为一个POJO类变量生成get函数，大大简化了某些合约代码的书写。

同样，变量的可见性也需要被合理地修饰，不该公开的变量果断用private修饰，使合约代码更符合『最少知道』的设计原则。

精确地将函数分类：函数的类型

函数可以被声明为pure、view，两者的作用可见下图。

函数类型	作用
pure	承诺不读取或修改状态。
view	保证不修改状态。

那么，什么是读取或修改状态呢？简单来说，两个状态就是读取或修改了账本相关的数据。

在FISCO BCOS中，读取状态可能是：

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

而修改状态可能是：

1. 修改状态变量。
2. 产生事件。
3. 创建其它合约。
4. 使用 selfdestruct。
5. 调用任何没有标记为 view 或者 pure 的函数。

6. 使用底层调用。

7. 使用包含特定操作码的内联汇编。

需要注意的是，在某些版本编译器中，并没有对这两个关键字进行强制的语法检查。

推荐尽可能使用pure和view来声明函数，例如将没有读取或修改任何状态的库函数声明为pure，这样既提升了代码可读性，也使其更赏心悦目，何乐而不为?

编译时就确定的值：状态常量

所谓的状态常量是指被声明为constant的状态变量。

一旦某个状态变量被声明为constant，那么该变量值只能为编译时确定的值，无法被修改。编译器一般会在编译状态计算出此变量实际值，不会给变量预留储存空间。所以，constant只支持修饰值类型和字符串。

状态常量一般用于定义含义明确的业务常量值。

面向切片编程：函数修饰器（modifier）

Solidity提供了强大的改变函数行为的语法：函数修饰器（modifier）。一旦某个函数加上了修饰器，修饰器内定义的代码就可以作为该函数的装饰被执行，类似其他高级语言中装饰器的概念。

这样说起来很抽象，让我们来看一个具体的例子：

```
1  pragma solidity ^0.4.11;
2
3  contract owned {
4      function owned() public { owner = msg.sender; }
5      address owner;
6
7      // 修饰器所修饰的函数体会被插入到特殊符号 _; 的位置。
8      modifier onlyOwner {
9          require(msg.sender == owner);
10         _;
11     }
12
13     // 使用onlyOwner修饰器所修饰，执行changeOwner函数前需要首先执行onlyOwner"
14     function changeOwner(address _owner) public onlyOwner {
15         owner = _owner;
16     }
17 }
```

如上所示，定义onlyOwner修饰器后，在修饰器内，require语句要求msg.sender必须等于owner。后面的"_;"表示所修饰函数中的代码。

所以，代码实际执行顺序变成了：

1. 执行onlyOwner修饰器的语句，先执行require语句。（执行第9行）
2. 执行changeOwner函数的语句。（执行第15行）

由于changeOwner函数加上了onlyOwner的修饰，故只有当msg.sender是owner才能成功调用此函数，否则会报错回滚。

同时，修饰器还能传入参数，例如上述的修饰器也可写成：

```
modifier onlyOwner(address sender) {
    require(sender == owner);
    _;
}

function changeOwner(address _owner) public onlyOwner(msg.sender) {
        owner = _owner;
}
```

同一个函数可有多个修饰器，中间以空格间隔，修饰器依次检查执行。此外，修饰器还可以被继承和重写。

由于其所提供的强大功能，修饰器也常被用来实现权限控制、输入检查、日志记录等。

比如，我们可以定义一个跟踪函数执行的修饰器：

```
event LogStartMethod();
event LogEndMethod();

modifier logMethod {
    emit LogStartMethod();
    _;
    emit LogEndMethod();
}
```

这样，任何用logMethod修饰器来修饰的函数都可记录其函数执行前后的日志，实现日志环绕效果。如果你已经习惯了使用Spring框架的AOP，也可以试试用modifier实现一个简单的AOP功能。

modifier最常见的打开方式是通过提供函数的校验器。在实践中，合约代码的一些检查语句常会被抽象并定义为一个modifier，如上述例子中的onlyOwner就是个最经典的权限校验器。这样一来，连检查的逻辑也能被快速复用，用户也不用再为智能合约里到处都是参数检查或其他校验类代码而苦恼。

可以debug的日志：合约里的事件（Event）

介绍完函数和变量，我们来聊聊Solidity其中一个较为独有的高级特性——事件机制。

事件允许我们方便地使用 EVM 的日志基础设施，而Solidity的事件有以下作用：

1. 记录事件定义的参数，存储到区块链交易的日志中，提供廉价的存储。
2. 提供一种回调机制，在事件执行成功后，由节点向注册监听的SDK发送回调通知，触发回调函数被执行。
3. 提供一个过滤器，支持参数的检索和过滤。

事件的使用方法非常简单，两步即可玩转。

- 第一步，使用关键字『event』来定义一个事件。建议事件的命名以特定前缀开始或以特定后缀结束，这样更便于和函数区分，在本文中我们将统一以『Log』前缀来命名事件。下面，我们用『event』来定义一个函数调用跟踪的事件：

```
event LogCallTrace(address indexed from, address indexed to, bool resu
```

事件在合约中可被继承。当他们被调用时，会将参数存储到交易的日志中。这些日志被保存到区块链中，与地址相关联。在上述例子中，用indexed标记参数被搜索，否则，这些参数被存储到日志的数据中，无法被搜索。

- 第二步，在对应的函数内触发定义事件。调用事件的时候，在事件名前加上『emit』关键字：

```
function f() public {
    emit LogCallTrace(msg.sender, this, true);
}
```

这样，当函数体被执行的时候，会触发执行LogCallTrace。

最后，在FISCO BCOS的Java SDK中，合约事件推送功能提供了合约事件的异步推送机制，客户端向节点发送注册请求，在请求中携带客户端关注的合约事件参数，节点根据请求参数对请求区块范围的Event Log进行过滤，将结果分次推送给客户端。更多细节可以参考合约事件推送功能文档。在SDK中，可以根据事件的indexed属性，根据特定值进行搜索。

合约事件推送功能文档：
https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/sdk/java_sdk.html#id14

不过，日志和事件无法被直接访问，甚至在创建的合约中也无法被直接访问。

但好消息是日志的定义和声明非常利于在『事后』进行追溯和导出。

例如，我们可以在合约的编写中，定义和埋入足够的事件，通过WeBASE的数据导出子系统我们可以将所有日志导出到MySQL等数据库中。这特别适用于生成对账文件、生成报表、复杂业务的OLTP查询等场景。此外，WeBASE提供了一个专用的代码生成子系统帮助分析具体的业务合约，自动生成相应的代码。

WeBASE的数据导出子系统：
https://webasedoc.readthedocs.io/zh_CN/latest/docs/WeBASE-Collect-Bee/index.html
代码生成子系统：
https://webasedoc.readthedocs.io/zh_CN/latest/docs/WeBASE-Codegen-Monkey/index.html

在Solidity中，事件是一个非常有用的机制，如果说智能合约开发最大的难点是debug，那善用事件机制可以让你快速制伏Solidity开发。

面向对象之重载

重载是指合约具有多个不同参数的同名函数。对于调用者来说，可使用相同函数名来调用功能相同，但参数不同的多个函数。在某些场景下，这种操作可使代码更清晰、易于理解，相信有一定编程经验的读者对此一定深有体会。

下面将展示一个典型的重载语法：

```
pragma solidity ^0.4.25;

contract Test {
    function f(uint _in) public pure returns (uint out) {
        out = 1;
    }

    function f(uint _in, bytes32 _key) public pure returns (uint out)
        out = 2;
    }
}
```

需要注意的是，每个合约只有一个构造函数，这也意味着合约的构造函数是不支持重载的。

我们可以想像一个没有重载的世界，程序员一定绞尽脑汁、想方设法给函数起名，大家的头发可能又要多掉几根。

面向对象之继承

Solidity使用『is』作为继承关键字。因此，以下这段代码表示的是，合约B继承了合约A：

```
pragma solidity ^0.4.25;

contract A {
}

contract B is A {
}
```

而继承的合约B可以访问被继承合约A的所有非private函数和状态变量。

在Solidity中，继承的底层实现原理为：当一个合约从多个合约继承时，在区块链上只有一个合约被创建，所有基类合约的代码被复制到创建的合约中。

相比于C++或Java等语言的继承机制，Solidity的继承机制有点类似于Python，支持多重继承机制。因此，Solidity中可以使用一个合约来继承多个合约。

在某些高级语言中，比如Java，出于安全性和可靠性的考虑，只支持单重继承，通过使用接口机制来实现多重继承。对于大多数场景而言，单继承的机制就可以满足需求了。

多继承会带来很多复杂的技术问题，例如所谓的『钻石继承』等，建议在实践中尽可能规避复杂的多继承。

继承简化了人们对抽象合约模型的认识和描述，清晰体现了相关合约间的层次结构关系，并且提供软件复用功能。这样，能避免代码和数据冗余，增加程序的重用性。

面向对象之抽象类和接口

根据依赖倒置原则，智能合约应该尽可能地面向接口编程，而不依赖具体实现细节。

Solidity支持抽象合约和接口的机制。

如果一个合约，存在未实现的方法，那么它就是抽象合约。例如：

```
pragma solidity ^0.4.25;

contract Vehicle {
    //抽象方法
    function brand() public returns (bytes32);
}
```

抽象合约无法被成功编译，但可以被继承。

接口使用关键字interface，上面的抽象也可以被定义为一个接口。

```
pragma solidity ^0.4.25;

interface Vehicle {
    //抽象方法
    function brand() public returns (bytes32);
}
```

接口类似于抽象合约，但不能实现任何函数，同时，还有进一步的限制：

1. 无法继承其他合约或接口。
2. 无法定义构造函数。
3. 无法定义变量。
4. 无法定义结构体
5. 无法定义枚举。

合适地使用接口或抽象合约有助于增强合约设计的可扩展性。但是，由于区块链EVM上计算和存储资源的限制，切忌过度设计，这也是从高级语言技术栈转到Solidity开发的老司机常常会陷入的天坑。

避免重复造轮子：库(Library)

在软件开发中，很多经典原则可以提升软件的质量，其中最为经典的就是尽可能复用久经考验、反复打磨、严格测试的高质量代码。此外，复用成熟的库代码还可以提升代码的可读性、可维护性，甚至是可扩展性。

和所有主流语言一样，Solidity也提供了库（Library）的机制。Solidity的库有以下基本特点：

1. 用户可以像使用合约一样使用关键词library来创建合约。
2. 库既不能继承也不能被继承。
3. 库的internal函数对调用者都是可见的。
4. 库是无状态的，无法定义状态变量，但是可以访问和修改调用合约所明确提供的状态变量。

接下来，我们来看一个简单的例子，以下是FISCO BCOS社区中一个LibSafeMath的代码库。我们对此进行了精简，只保留了加法的功能：

```
pragma solidity ^0.4.25;

library LibSafeMath {
  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal returns (uint256 c) {
    c = a + b;
    assert(c >= a);
    return c;
  }
}
```

我们只需在合约中import库的文件，然后使用L.f()的方式来调用函数，（例如LibSafeMath.add(a,b)）。

接下来，我们编写调用这个库的测试合约，合约内容如下：

```
pragma solidity ^0.4.25;

import "./LibSafeMath.sol";

contract TestAdd {

  function testAdd(uint256 a, uint256 b) external returns (uint256 c)
    c = LibSafeMath.add(a,b);
  }
}
```

在FISCO BCOS控制台中，我们可以测试合约的结果（控制台的介绍文章详见FISCO BCOS 控制台详解，飞一般的区块链体验），运行结果如下：

```
================================================================
Welcome to FISCO BCOS console(1.0.8)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ ______  ______   ______   ______       _______   ______   _
|        |      \/      \ /      \ /      \     |       \ /      \ /
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$     | $
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \

================================================================
[group:1]> deploy TestAdd
contract address: 0xe2af1fd7ecd91eb7e0b16b5c754515b775b25fd2

[group:1]> call TestAdd 0xe2af1fd7ecd91eb7e0b16b5c754515b775b25fd2 te
transaction hash: 0x136ce66603aa6e7fd9e4750fcf25302b13171abba8c6b2109
----------------------------------------------------------------
Output
function: testAdd(uint256,uint256)
return type: (uint256)
return value: (2020)
----------------------------------------------------------------

[group:1]>
```

通过以上示例，我们可清晰了解在Solidity中应如何使用库。

类似Python，在某些场景下，指令『using A for B;』可用于附加库函数（从库 A）到任何类型（B）。这些函数将接收到调用它们的对象作为第一个参数（像 Python 的 self 变量）。这个功能使库的使用更加简单、直观。

例如，我们对代码进行如下简单修改：

```
pragma solidity ^0.4.25;

import "./LibSafeMath.sol";

contract TestAdd {
  // 添加using ... for ... 语句，库 LibSafeMath 中的函数被附加在uint256的类
  using LibSafeMath for uint256;

  function testAdd(uint256 a, uint256 b) external returns (uint256 c)
        //c = LibSafeMath.add(a,b);
        c = a.add(b);
        //对象a直接被作为add方法的首个参数传入。
  }
}
```

验证一下结果依然是正确的。

```
================================================================================
Welcome to FISCO BCOS console(1.0.8)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ ______  ______   ______   ______       _______   ______   _
|        |      \/      \ /      \ /      \     |       \ /      \ /
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$     | $
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \

================================================================================
[group:1]> deploy TestAdd
contract address: 0xf82c19709a9057d8e32c19c23e891b29b708c01a

[group:1]> call TestAdd 0xf82c19709a9057d8e32c19c23e891b29b708c01a te
transaction hash: 0xcc44a80784404831d8522dde2a8855606924696957503491e
--------------------------------------------------------------------------------
Output
function: testAdd(uint256,uint256)
return type: (uint256)
return value: (2020)
--------------------------------------------------------------------------------

[group:1]>
```

更好地使用Solidity library有助于开发者更好地复用代码。除了Solidity社区提供的大量开源、高质量的代码库外，FISCO BCOS社区也计划推出全新的Solidity代码库，开放给社区用户，敬请期待。

当然，你也可以自己动手，编写可复用的代码库组件，并分享到社区。

总结

本文介绍了Solidity合约编写的若干高级语法特性，旨在抛砖引玉，帮助读者快速沉浸到Solidity编程世界。

编写高质量、可复用的Solidity代码的诀窍在于：多看社区优秀的代码，多动手实践编码，多总结并不断进化。期待更多朋友在社区里分享Solidity的宝贵经验和精彩故事，have fun :)

下期预告

FISCO BCOS的代码完全开源且免费

下载地址↓↓↓

https://github.com/FISCO-BCOS/FISCO-BCOS