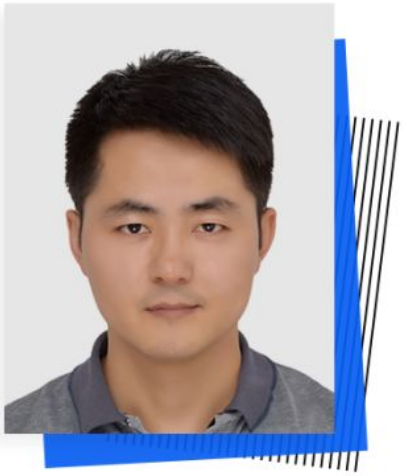
# 如何优雅地编写智能合约

原创 张龙 FISCO BCOS开源社区 2019-09-24



AUTHOR | 作者

张龙

微众银行区块链高级架构师

## 1

## 写在开头

众所周知，智能合约的出现，使得区块链不仅能够处理简单的转账功能，还能实现复杂的业务逻辑处理，其核心在于账户模型。

目前在众多区块链平台中，大多数集成了以太坊虚拟机，并使用Solidity作为智能合约的开发语言。Solidity语言不仅支持基础/复杂数据类型操作、逻辑操作，同时提供高级语言的相关特性，比如继承、重载等。

除此之外，Solidity语言还内置很多常用方法，比如成套的加密算法接口，使得数据加解密非常简单；提供事件Event，便于跟踪交易的执行状态，为业务的逻辑处理、监控和运维提供便利。

然而，我们在编写智能合约代码的时候，还是会碰到各种问题，这些问题包括：代码bug、可扩展性、可维护性、业务互操作的友好性等。同时，Solidity语言还不完善、需要执行在EVM上、语言本身及执行环境也会给我们带来一些坑。

基于此，我们结合之前的项目和经验进行梳理，希望将之前碰到的问题总结下来，为后续的开发提供借鉴依据。

⊙ **注**：智能合约安全不在本篇文章讨论范畴，文中智能合约代码为0.4版本写法。

## Solidity常见问题

### EVM栈溢出

EVM的栈深度为1024，但是EVM指令集最多访问深度为16，这给智能合约的编写带来很多限制，常见的报错为：stack overflows。

这个报错出现在智能合约编译阶段。我们知道EVM的栈用于存储临时变量或者局部变量，比如函数的参数或者函数内部的变量。优化一般也是从这两个方面出发。

下述代码片段可能存在栈溢出问题：

```
//如果课程超过14个，那么参数超过16个，则溢出
function addStudentScores(
  bytes32 studentId,
  bytes32 studentName,
  uint8 chineseScore;
  uint8 englishScore;
  ...
  uint8 mathScore
)
  public
    returns (bool)
{
    //TODO
}
```

函数参数和局部变量不能超过16个，一般建议不超过10个。参数过多会出现的问题：

1. 容易栈溢出；
2. 编写代码费劲容易出错；
3. 不利于业务理解和维护；
4. 不便于扩展。

常规做法是尽量减少函数参数，碰到实在无法减少的情况，建议采用数组。局部变量和函数参数类似，定义过多也会导致栈溢出，可以通过拼接数组，减少变量个数，一般会和数组入参结合使用，上述代码片段优化后如下所示：

```
function addStudentScores(
  bytes32[] studentInfo,
    uint8[] scores
)
  public
    returns (bool)
{
    //TODO
}
```

## BINARY字段超长

智能合约通过JAVA编译器编译后会生成对应的JAVA合约，在JAVA合约中有一个重要的常量字段BINARY，该字段为智能合约的编码，即合约代码。合约代码用于合约部署时签名，每一次合约的变更对应的BINARY都会不一样。

在编写智能合约时，如果单个智能合约代码很长，经过编译后的BINARY字段会很大。在JAVA合约中，BINARY字段用String类型存储，String类型的最大长度为65534，如果智能合约代码过多，会导致BINARY字段的长度超过String类型的最大长度，导致String类型溢出，从而报错。

解决方案也非常简单：

1. 尽可能复用代码，比如某些判断在不同的方法中多次出现，可以抽取出来，这样也便于后续的维护；

2. 合约拆分，将一个合约拆分成为多个合约，一般出现String越界，基本上可以说明合约设计不合理。

## 慎用string类型

string类型是一个比较特殊的动态字节数组，无法直接和定长数组进行转化，其解析和数组转化也非常复杂。

除此之外，string类型浪费空间、非常昂贵（消耗大量gas），且不能进行合约间传递（新的实验性ABI编译器除外），所以建议用bytes代替，特殊场景例外，比如未知长度字节数组或预留字段。

⊙ **备注：**string类型可以通过在合约中添加新的实验性ABI编译器（如下代码）进行合约间传递。

```
pragma experimental ABIEncoderV2;
```


## 智能合约编写

### 分层设计

网上多数智能合约的例子，比如著名的ERC20等，通常做法是写在一个智能合约文件中，这种写法本身没有什么问题，但面临复杂的业务，这种写法无可避免地会出现：

1. 代码全部写在一个文件中，这个文件就非常大，不便于查看和理解，修改容易出错；
2. 不便于多人协作和维护，尤其是业务发生变动或代码出现漏洞时，需要重新升级部署合约，导致之前的合约作废，相关业务数据或资产也就没有了。

那么，有没有一种方法可以使得智能合约升级又不影响原有账户（地址）？

先给答案：没有！（基于底层的分布式存储的CRUD除外，目前FISCO BCOS 2.0支持分布式存储，可直接通过CRUD操作数据库进行合约升级。）

但是！没有并不意味着不能升级，智能合约升级之后最大的问题是数据，所以只要保证数据完整就可以了。

举个例子：我们需要对学生信息上链，常规写法如下所示：

```
contract Students {
  struct StudentInfo {
        uint32 _studentId;
        bytes32 _studentName;
    }
    mapping (uint32 => StudentInfo) private _studentMapping;
    function addStudent(uint32 studentId, bytes32 studentName) public
      //TODO:
    }
}
```

这种写法，代码全部在一个智能合约中，如果现有的智能合约已经不能满足业务诉求，比如类型为uint32字段需升级为为uint64，或者合约中添加一个新的字段，比如sex，那这个智能合约就没有用了，需要重新部署。但因为重新部署，合约地址变了，无法访问到之前的数据。

一种做法是对合约进行分层，将业务逻辑和数据分离，如下所示：

```
contract StudentController {
  mapping (uint32 => address) private _studentMapping;
    function addStudent(uint32 studentId, bytes32 studentName) public
      //TODO:
    }
}
contract Student {
  uint32 _studentId;
  bytes32 _studentName;
    //uint8 sex;
}
```

这种写法使得逻辑和数据分离，当需要新增一个性别sex字段时，原始数据可以编写两个

StudentController合约，通过版本区分，新的Student采用新的逻辑，需要业务层面做兼容性处理，其最大的问题是对于原有数据的交互性操作，需要跨合约完成，非常不方便，比如查询所有学生信息。

我们再次进行分层，多出一个map层，专门用于合约数据管理，即使业务逻辑层和数据层都出现问题，也没有关系，只需要重新编写业务逻辑层和数据层，并对原有数据进行特殊处理就可以做到兼容。不过，这种做法需要提前在数据合约中做好版本控制（version），针对不同的数据，采用不同的逻辑。

这种做法最大的好处是数据全部保存在StudentMap中，数据合约和逻辑合约的变更都不会影响到数据，且在后续的升级中，可以通过一个controller合约做到对新老数据的兼容，如下所示：

```
contract StudentController {
  mapping (uint32 => address) private _studentMapping;
  constructor(address studentMapping) public {
      _studentMapping = studentMapping;
    }
    function addStudent(uint version, uint32 studentId, bytes32 stude
      if(version == 1){
            //TODO
        }else if(version == 2){
            //TODO
        }
    }
}
contract StudentMap {
  mapping (uint32 => address) private _studentMapping;
    function getStudentMap() public constant returns(address){
      return _studentMapping;
    }
}
contract Student {
  uint8 version;
  uint32 _studentId;
  bytes32 _studentName;
    //uint8 sex;
}
```

## 统一接口

智能合约尽管具备很多高级语言的特性，但是本身还是存在很多限制。对于业务的精准处理，需要采用Event事件进行跟踪，对于不同的合约和方法，可以编写不同的Event事件，如下：

PS：你也可以采用require的方式进行处理，不过require方式不支持动态变量，每个require处理后需要填入特定的报错内容，在SDK层面耦合性太重，且不便于扩展。

```
contract StudentController {
  //other code
    event addStudentSuccessEvent(...); //省略参数，下同
    event addStudentFailEvent(...);

    function addStudent(bytes32 studentId, bytes32 studentName) publi
      if(add success){
          addStudentSuccessEvent(...);
            return true;
        }else {
          addStudentFailEvent(...);
            return false;
        }
    }
}
```

这种做法也没有问题，不过我们需要编写大量的Event事件，增加了智能合约的复杂性。如果每次新增加一个方法或者处理逻辑，我们都需要编写一个专门的事件进行追踪，代码侵入性太强，容易出错。

除此之外，基于智能合约的SDK开发，对于每一个交易（方法）由于Event事件不同，需要编写大量的不可复用的代码，解析Event事件。这种写法，对于代码的理解和维护性都是非常差的。要解决这个问题，我们只需要编写一个基合约CommonLib，如下所示：

```
contract CommonLib {
  //tx code
  bytes32 constant public ADD_STUDENT = "1";
    bytes32 constant public MODIFY_STUDENT_NAME = "2";

    //return code
    bytes32 constant public STUDENT_EXIST = "1001";
    bytes32 constant public STUDENT_NOT_EXIST = "1002";
    bytes32 constant public TX_SUCCESS = "0000";

  event commonEvent(bytes id, bytes32 txCode, bytes32 rtnCode);
}

contract StudentController is CommonLib {
     function addStudent(bytes32 studentId, bytes32 studenntName) pub
      //process add student
        if(add success){
          commonEvent(studentId, ADD_STUDENT, TX_SUCCESS);
            return true;
        }else {
          commonEvent(studentId, ADD_STUDENT, STUDENT_EXIST);
            return false;
        }
    }
    function modifyStudentName(bytes32 studentId, bytes32 studentName
      //TODO:
    }
}
```

当新增一个modifyStudentName方法或其他合约时，原有的做法是根据方法可能出现的情况定义多个Event事件，然后在SDK中针对不同的Event编写解析方法，工作量很大。现在只需要在CommonLib中定义一对常量即可，SDK的代码可以完全复用，几乎没有任何新增的工作。

⊙ **注：**在上述例子中，commonEvent包含三个参数，其中txCode为交易类型，即调用的哪个交易方法，rtnCode为返回代码，表示在执行txCode所代表的交易方法时出现什么情况，这两个参

数是必须的。在commonEvent中还有一个Id字段，用于关联业务字段studentId，在具体的项目中，关联的业务字段可以自行定义和调整。

## 代码细节

代码细节能体验一个coder的能力和职业操守。在业务比较赶的情况下，经常会忽略代码细节，同时代码细节（风格）因人而异。对于一个多人协作的项目，统一的代码风格、代码规范，能极大提升研发效率、降低研发及维护成本、降低代码错误率。

### 命名规范

智能合约命名并没有一个标准，不过团队内部可以按照一个行业共识的规范执行。经过实战，推荐以下风格（不强制），如下代码块。

1. 合约命名：采用驼峰命名、首字母大写、且能表达对应的业务含义；
2. 方法命名：采用驼峰命名、首字母小写、且能表达对应的业务含义；
3. 事件命名：采用驼峰命名、首字母小写、且能表达对应的业务含义，以Event结尾；
4. 合约变量：采用驼峰命名、以_开头，首字母小写、且能表达对应的业务含义；
5. 方法入参：采用驼峰命名、首字母小写、且能表达对应的业务含义；
6. 方法出参：建议只写出参类型，无需命名，特殊情况例外；
7. 事件参数：同方法入参；
8. 局部变量：同方法入参。

```
contract Student {
  bytes32 _studentId;
    bytes32 _studentName;
  event setStudentNameEvent(bytes32 studentId, bytes32 studentName);
  function setStudentName(bytes32 studentName) public returns(bool){}
    //other code
}
```

### 条件判断

在智能合约中，可以通过逻辑控制进行条件判断，比如if语句，也可以采用solidity语言提供的内置方法，比如require等。

两者在执行时存在一些差异，一般情况下，使用require没有问题，但是require不支持传参，如果业务需要在异常情况下给出明确的异常提示，则推荐使用if语句结合Event使用，如下。

```
event commonEvent(bytes id, bytes32 txCode, bytes32 rtnCode);
//require(!_studentMapping.studentExist(studentId),"student does not e
if(_studentMapping.studentExist(studentId)){
  commonEvent(studentId, ADD_STUDENT, STUDENT_EXIST);
  return false;
}
```

### 常量及注释

在智能合约中，常量和其他编程语言一样，需要采用大写加下划线方式命名，且命名需具备业务含义，同时需要采用constant关键词修饰，建议放置在合约开头。

常量也需要区分，对外接口常量采用public修饰，放置在基合约中。业务相关常量采用private修饰，放置在具体的业务逻辑合约中。如下所示：

```
contract CommonLib {
    //tx code
  bytes32 constant public ADD_STUDENT = "1";
    bytes32 constant public MODIFY_STUDENT_NAME = "2";
    ...
}

contract StudentController is CommonLib {
  /** student status */
    bytes32 constant private STUDENT_REGISTED = "A";
    bytes32 constant private STUDENT_CANCELED = "C";

    //other code
}
```

智能合约的注释同大部分编程语言，没有很严格的要求。对于一些特殊字段、常量、数组中的每个变量及特定逻辑，需进行说明，方法及Event可以使用/** comments */，特定字段及逻辑说明可采用//。如下所示：

```
/**
* stundent controller
*/
contract StudentController {
  /** add student */
    function addStudent(
      //[0]-seqNo;[1]-studentId;[2]-studentName;
      bytes32[3] studentInfos
)
      public returns(bool)
{
      //TODO:
    }
}
```

## 兜底方案

在智能合约设计过程中，谁都无法保证自己的代码一定满足业务诉求，因为业务的变动是绝对的。同时，谁也无法保证业务及操作人员一定不会犯错，比如业务对某些字段未做校验导致链上出现非法数据，或者因为业务操作人员手误、恶意操作等，导致链上出现错误数据。

区块链系统不像其他传统系统，可以通过手动修改库或文件对数据进行修正，区块链必须通过交易对数据进行修正。

针对业务变更，在编写智能合约时可以适当增加一些保留字段，用于后续可能存在的业务变更。一般定义为一个通用化的数据类型比较合适，比如string，一方面string类型存储容量大，另一方面几乎啥都可以存。

我们可以在SDK层面通过数据处理将扩展数据存入string字段，在使用时提供相应的数据处理反向操作解析数据，比如在Student合约中，新增reserved字段，如下所示。当前阶段，reserved没有任何作用，在智能合约中为空。

```
contract Student {
  //other code
    string _reserved;

    function getReserved() constant public returns(string){
    return _reserved;
  }

  function setReserved(string reserved) onlyOwner public returns(bool
    _reserved = reserved;
    return true;
  }
}
```

针对手误或者非法操作导致的数据错误，务必预留相关的接口，以便在紧急情况下可以不修改合

约，而通过更新SDK对链上数据进行修复（SDK中可以先不实现）。比如针对Student合约中的owner字段，添加set操作。

```
contract Student {
  //other code;
    address _owner;
  function setOwner(address owner) onlyOwner public returns(bool){
    _owner = owner;
    return true;
  }
}
```

需要特别注意的是，对于预留字段和预留方法，必须确保其操作权限，防止引入更多问题。同时预留字段和预留方法都是一种非正常情况下的设计，具备超前意识，但一定要避免过度设计，这样会导致智能合约的存储空间非常浪费，同时预留方法使用不当会给业务的安全性带来隐患。


## 写在最后

区块链应用的开发涉及很多方面，智能合约是核心，本篇给出了开发智能合约过程中的一些建议和优化方法，但并不是完整和完美的，且本质上无法杜绝bug的出现，但通过优化方法，可以让代码变得更加健壮和易维护，从这点上来讲，已具备业界的基本良心要求了。

**下载地址↓↓↓**

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