智能合约编写之Solidity的编程攻略

原创 毛嘉宇 FISCO BCOS开源社区 3月26日

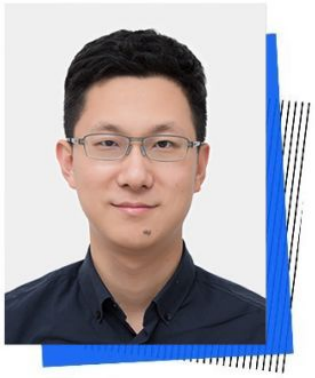
第1场 | 智能合约初探：概念与演变
第2场 | 智能合约编写之Solidity的基础特性
第3场 | 智能合约编写之Solidity的高级特性
第4场 | 智能合约编写之 Solidity的设计模式

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 的编程攻略

作者：毛嘉宇



毛嘉宇

FISCO BCOS核心开发者

have fun

前 言

作为一名搬砖多年的资深码农，刚开始接触Solidity便感觉无从下手：昂贵的计算和存储资源、简陋的语法特性、令人抓狂的debug体验、近乎贫瘠的类库支持、一言不合就插入汇编语句……让人不禁怀疑，这都已经过了9012年了，居然还有这种反人类的语言?

对于习惯使用各类日益“傻瓜化”的类库和自动化高级框架的码农而言，学习Solidity的过程就是一场一言难尽的劝退之旅。

但随着对区块链底层技术的深入学习，大家会慢慢理解作为运行在“The World Machine”上的Solidity语言，必须要严格遵循的设计原则以及权衡后必须付出的代价。

正如黑客帝国中那句著名的slogan：“Welcome to the dessert of the real”，在恶劣艰苦的环境面前，最重要的是学会如何适应环境、保存自身并快速进化。

本文总结了一些Solidity编程的攻略，期待各位读者不吝分享交流，达到抛砖引玉之效。

（图片摄于羚羊峡，远看之下，一片红土，草木稀少，平淡无奇。）

上链的原则

“如无必要，勿增实体”。

基于区块链技术及智能合约发展现状，数据的上链需遵循以下原则：

- 需要分布式协作的重要数据才上链，非必需数据不上链；
- 敏感数据脱敏或加密后上链（视数据保密程度选择符合隐私保护安全等级要求的加密算法）；
- 链上验证，链下授权。

在使用区块链时，开发者不需要将所有业务和数据都放到链上。相反，“好钢用在刀刃上”，智能合约更适合被用在分布式协作的业务场景中。

精简函数变量

如果在智能合约中定义了复杂的逻辑，特别是合约内定义了复杂的函数入参、变量和返回值，就会在编译的时候碰到以下错误：

```
Compiler error: Stack too deep, try removing local variables.
```

这也是社区中的高频技术问题之一。造成这个问题的原因就是EVM所设计用于最大的栈深度为16。

所有的计算都在一个栈内执行，对栈的访问只限于其顶端，限制方式为：允许拷贝最顶端16个元素中的一个到栈顶，或者将栈顶元素和下面16个元素中的一个交换。

所有其他操作都只能取最顶的几个元素，运算后，把结果压入栈顶。当然可以把栈上的元素放到存储或内存中。但无法只访问栈上指定深度的那个元素，除非先从栈顶移除其他元素。如果一个合约中，入参、返回值、内部变量的大小超过了16个，显然就超出了栈的最大深度。

因此，我们可以使用结构体或数组来封装入参或返回值，达到减少栈顶元素使用的目的，从而避免此错误。

例如以下代码，通过使用bytes数组来封装了原本16个bytes变量。

```
function doBiz(bytes[] paras) public {
        require(paras.length >= 16);
        // do something
}
```

保证参数和行为符合预期

心怀“Code is law”的远大理想，极客们设计和创造了区块链的智能合约。

在联盟链中，不同的参与者可以使用智能合约来定义和书写一部分业务或交互的逻辑，以完成部分社会或商业活动。

相比于传统软件开发，智能合约对函数参数和行为的安全性要求更为严格。在联盟链中提供了身份实名和CA证书等机制，可以有效定位和监管所有参与者。不过，智能合约缺乏对漏洞和攻击的事前干预机制。正所谓字字珠玑，如果不严谨地检查智能合约输入参数或行为，有可能会触发一些意想不到的bug。

因此，在编写智能合约时，一定要注意对合约参数和行为的检查，尤其是那些对外部开放的合约函数。

Solidity提供了require、revert、assert等关键字来进行异常的检测和处理。一旦检测并发现错误，整个函数调用会被回滚，所有状态修改都会被回退，就像从未调用过函数一样。

以下分别使用了三个关键字，实现了相同的语义。

```
require(_data == data, "require data is valid");

if(_data != data) { revert("require data is valid"); }

assert(_data == data);
```

不过，这三个关键字一般适用于不同的使用场景：

- require：最常用的检测关键字，用来验证输入参数和调用函数结果是否合法。
- revert：适用在某个分支判断的场景下。
- assert: 检查结果是否正确、合法，一般用于函数结尾。

在一个合约的函数中，可以使用函数修饰器来抽象部分参数和条件的检查。在函数体内，可以对运行状态使用if-else等判断语句进行检查，对异常的分支使用revert回退。在函数运行结束前，可

以使用assert对执行结果或中间状态进行断言检查。

在实践中，推荐使用require关键字，并将条件检查移到函数修饰器中去；这样可以让函数的职责更为单一，更专注到业务逻辑中。同时，函数修饰器等条件代码也更容易被复用，合约也会更加安全、层次化。

在本文中，我们以一个水果店库存管理系统为例，设计一个水果超市的合约。这个合约只包含了对店内所有水果品类和库存数量的管理，setFruitStock函数提供了对应水果库存设置的函数。在这个合约中，我们需要检查传入的参数，即水果名称不能为空。

```
pragma solidity ^0.4.25;

contract FruitStore {
    mapping(bytes => uint) _fruitStock;
    modifier validFruitName(bytes fruitName) {
        require(fruitName.length > 0, "fruite name is invalid!");
        _;
    }
    function setFruitStock(bytes fruitName, uint stock) validFruitNam
        _fruitStock[fruitName] = stock;
    }
}
```

如上所述，我们添加了函数执行前的参数检查的函数修饰器。同理，通过使用函数执行前和函数执行后检查的函数修饰器，可以保证智能合约更加安全、清晰。智能合约的编写需要设置严格的前置和后置函数检查，来保证其安全性。

严控函数的执行权限

如果说智能合约的参数和行为检测提供了静态的合约安全措施，那么合约权限控制的模式则提供了动态访问行为的控制。

由于智能合约是发布到区块链上，所有数据和函数对所有参与者都是公开透明的，任一节点参与者都可发起交易，无法保证合约的隐私。因此，合约发布者必须对函数设计严格的访问限制机制。

Solidity提供了函数可见性修饰符、修饰器等语法，灵活地使用这些语法，可帮助构建起合法授权、受控调用的智能合约系统。

还是以刚才的水果合约为例。现在getStock提供了查询具体水果库存数量的函数。

```
pragma solidity ^0.4.25;

contract FruitStore {
    mapping(bytes => uint) _fruitStock;
    modifier validFruitName(bytes fruitName) {
        require(fruitName.length > 0, "fruite name is invalid!");
        _;
    }
    function getStock(bytes fruit) external view returns(uint) {
        return _fruitStock[fruit];
    }
    function setFruitStock(bytes fruitName, uint stock) validFruitNam
        _fruitStock[fruitName] = stock;
    }
}
```

水果店老板将这个合约发布到了链上。但是，发布之后，setFruitStock函数可被任何其他联盟链的参与者调用。

虽然联盟链的参与者是实名认证且可事后追责；但一旦有恶意攻击者对水果店发起攻击，调用setFruitStock函数就能任意修改水果库存，甚至将所有水果库存清零，这将对水果店正常经营管理产生严重后果。

因此，设置某些预防和授权的措施很必要：对于修改库存的函数setFruitStock，可在函数执行前对调用者进行鉴权。

类似的，这些检查可能会被多个修改数据的函数复用，使用一个onlyOwner的修饰器就可以抽象此检查。_owner字段代表了合约的所有者，会在合约构造函数中被初始化。使用public修饰getter查询函数，就可以通过_owner()函数查询合约的所有者。

```
contract FruitStore {
    address public  _owner;
    mapping(bytes => uint) _fruitStock;

    constructor() public {
        _owner = msg.sender;
    }

    modifier validFruitName(bytes fruitName) {
        require(fruitName.length > 0, "fruite name is invalid!");
        _;
    }
    // 鉴权函数修饰器
    modifier onlyOwner() {
        require(msg.sender == _owner, "Auth: only owner is authorized
        _;
    }
    function getStock(bytes fruit) external view returns(uint) {
        return _fruitStock[fruit];
    }
    // 添加了onlyOwner修饰器
    function setFruitStock(bytes fruitName, uint stock)
        onlyOwner validFruitName(fruitName) external {
        _fruitStock[fruitName] = stock;
    }
}
```

这样一来，我们可以将相应的函数调用权限检查封装到修饰器中，智能合约会自动发起对调用者身份验证检查，并且只允许合约部署者来调用setFruitStock函数，以此保证合约函数向指定调用者开放。

抽象通用的业务逻辑

分析上述FruitStore合约，我们发现合约里似乎混入了奇怪的东西。参考单一职责的编程原则，水果店库存管理合约多了上述函数功能检查的逻辑，使合约无法将所有代码专注在自身业务逻辑中。

对此，我们可以抽象出可复用的功能，利用Solidity的继承机制继承最终抽象的合约。

基于上述FruitStore合约，可抽象出一个BasicAuth合约，此合约包含之前onlyOwner的修饰器和相关功能接口。

```
contract BasicAuth {
    address public _owner;

    constructor() public {
        _owner = msg.sender;
    }

    function setOwner(address owner)
        public
        onlyOwner
{
        _owner = owner;
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "BasicAuth: only owner is autho
        _;
    }
}
```

FruitStore可以复用这个修饰器，并将合约代码收敛到自身业务逻辑中。

```
import "./BasicAuth.sol";

contract FruitStore is BasicAuth {
    mapping(bytes => uint) _fruitStock;

    function setFruitStock(bytes fruitName, uint stock)
        onlyOwner validFruitName(fruitName) external {
        _fruitStock[fruitName] = stock;
    }
}
```

这样一来，FruitStore的逻辑被大大简化，合约代码更精简、聚焦和清晰。

预防私钥的丢失

在区块链中调用合约函数的方式有两种：内部调用和外部调用。

出于隐私保护和权限控制，业务合约会定义一个合约所有者。假设用户A部署了FruitStore合约，那上述合约owner就是部署者A的外部账户地址。这个地址由外部账户的私钥计算生成。

但是，在现实世界中，私钥泄露、丢失的现象比比皆是。一个商用区块链DAPP需要严肃考虑私钥的替换和重置等问题。

这个问题最为简单直观的解决方法是添加一个备用私钥。这个备用私钥可支持权限合约修改owner的操作，代码如下：

```
contract BasicAuth {
    address public  _owner;
    address public _bakOwner;

    constructor(address bakOwner) public {
        _owner = msg.sender;
        _bakOwner = bakOwner;
    }

    function setOwner(address owner)
        public
        canSetOwner
{
        _owner = owner;
    }

    function setBakOwner(address owner)
        public
        canSetOwner
{
        _bakOwner = owner;
    }

    // ...

    modifier isAuthorized() {
        require(msg.sender == _owner || msg.sender == _bakOwner, "Bas
        _;
    }
}
```

这样，当发现私钥丢失或泄露时，我们可以使用备用外部账户调用setOwner重置账号，恢复、保障业务正常运行。

面向接口编程

上述私钥备份理念值得推崇，不过其具体实现方式存在一定局限性，在很多业务场景下，显得过于简单粗暴。

对于实际的商业场景，私钥的备份和保存需要考虑的维度和因素要复杂得多，对应密钥备份策略也更多元化。

以水果店为例，有的连锁水果店可能希望通过品牌总部来管理私钥，也有的可能通过社交关系重置帐号，还有的可能会绑定一个社交平台的管理帐号……

面向接口编程，而不依赖具体的实现细节，可以有效规避这个问题。例如，我们利用接口功能首先定义一个判断权限的抽象接口：

```
contract Authority {
    function canCall(
        address src, address dst, bytes4 sig
    ) public view returns (bool);
}
```

这个canCall函数涵盖了函数调用者地址、目标调用合约的地址和函数签名，函数返回一个bool的结果。这包含了合约鉴权所有必要的参数。

我们可进一步修改之前的权限管理合约，并在合约中依赖Authority接口，当鉴权时，修饰器会调用接口中的抽象方法：

```
contract BasicAuth {
    Authority  public  _authority;

    function setAuthority(Authority authority)
        public
        auth
    {
        _authority = authority;
    }

    modifier isAuthorized() {
        require(auth(msg.sender, msg.sig), "BasicAuth: only owner or
        _;
    }

    function auth(address src, bytes4 sig) public view returns (bool)
{
        if (src == address(this)) {
            return true;
        } else if (src == _owner) {
            return true;
        } else if (_authority == Authority(0)) {
            return false;
        } else {
            return _authority.canCall(src, this, sig);
        }
    }
}
```

这样，我们只需要灵活定义实现了canCall接口的合约，在合约的canCall方法中定义具体判断逻辑。而业务合约，例如FruitStore继承BasicAuth合约，在创建时只要传入具体的实现合约，就可以实现不同判断逻辑。

合理预留事件

迄今为止，我们已实现强大灵活的权限管理机制，只有预先授权的外部账户才能修改合约owner属性。

不过，仅通过上述合约代码，我们无法记录和查询修改、调用函数的历史记录和明细信息。而这样的需求在实际业务场景中比比皆是。比如，FruitStore水果店需要通过查询历史库存修改记录，计算出不同季节的畅销与滞销水果。

一种方法是依托链下维护独立的台账机制。不过，这种方法存在很多问题：保持链下台账和链上记录一致的成本开销非常高；同时，智能合约面向链上所有参与者开放，一旦其他参与者调用了合约函数，相关交易信息就存在不能同步的风险。

针对此类场景，Solidity提供了event语法。event不仅具备可供SDK监听回调的机制，还能用较低的gas成本将事件参数等信息完整记录、保存到区块中。FISCO BCOS社区中，也有WEBASE-Collect-Bee这样的工具，在事后实现区块历史事件信息的完整导出。

WEBASE-Collect-Bee工具参考链接如下：
https://webasedoc.readthedocs.io/zh_CN/latest/docs/WeBASE-Collect-Bee/index.html

基于上述权限管理合约，我们可以定义相应的修改权限事件，其他事件以此类推。

```
event LogSetAuthority (Authority indexed authority, address indexed fr
}
```

接下来，可以调用相应的事件：

```
function setAuthority(Authority authority)
        public
        auth
{
        _authority = authority;
        emit LogSetAuthority(authority, msg.sender);
    }
```

当setAuthority函数被调用时，会同时触发LogSetAuthority，将事件中定义的Authority合约地址以及调用者地址记录到区块链交易回执中。当通过控制台调用setAuthority方法时，对应事件LogSetAuthority也会被打印出来。

基于WEBASE-Collect-Bee，我们可以导出所有该函数的历史信息到数据库中。也可基于WEBASE-Collect-Bee进行二次开发，实现复杂的数据查询、大数据分析和数据可视化等功能。

遵循安全编程规范

每一门语言都有其相应的编码规范，我们需要尽可能严格地遵循Solidity官方编程风格指南，使代码更利于阅读、理解和维护，有效地减少合约的bug数量。

Solidity官方编程风格指南参考链接如下：
https://solidity.readthedocs.io/en/latest/style-guide.html

除了编程规范，业界也总结了很多安全编程指南，例如重入漏洞、数据结构溢出、随机数误区、构造函数失控、为初始化的存储指针等等。重视和防范此类风险，采用业界推荐的安全编程规范至关重要，例如Solidity官方安全编程指南。参考链接如下：
https://solidity.readthedocs.io/en/latest/security-considerations.html

同时，在合约发布上线后，还需要注意关注、订阅Solidity社区内安全组织或机构发布的各类安全漏洞、攻击手法，一旦出现问题，及时做到亡羊补牢。

对于重要的智能合约，有必要引入审计。现有的审计包括了人工审计、机器审计等方法，通过代码分析、规则验证、语义验证和形式化验证等方法保证合约安全性。

虽然本文通篇都在强调，模块化和重用被严格审查并广泛验证的智能合约是最佳的实践策略。但在实际开发过程，这种假设过于理想化，每个项目或多或少都会引入新的代码，甚至从零开始。

不过，我们仍然可以视代码的复用程度进行审计分级，显式地标注出引用的代码，将审计和检查的重点放在新代码上，以节省审计成本。

最后，“前事不忘后事之师”，我们需要不断总结和学习前人的最佳实践，动态和可持续地提升编码工程水平，并不断应用到具体实践中。

积累和复用成熟的代码

前文面向接口编程中的思想可降低代码耦合，使合约更容易扩展、利于维护。在遵循这条规则之外，还有另外一条忠告：尽可能地复用现有代码库。

智能合约发布后难以修改或撤回，而且发布到公开透明的区块链环境上，就意味着一旦出现bug造成的损失和风险更甚于传统软件。因此，复用一些更好更安全的轮子远胜过重新造轮子。

在开源社区中，已经存在大量的业务合约和库可供使用，例如OpenZeppelin等优秀的库。

如果在开源世界和过去团队的代码库里找不到合适的可复用代码，建议在编写新代码时尽可能地测试和完善代码设计。此外，还要定期分析和审查历史合约代码，将其模板化，以便于扩展和复用。

例如，针对上面的BasicAuth，参考防火墙经典的ACL(Access Control List)设计，我们可以进一步地继承和扩展BasicAuth，抽象出ACL合约控制的实现。

```
contract AclGuard is BasicAuth {
    bytes4 constant public ANY_SIG = bytes4(uint(-1));
    address constant public ANY_ADDRESS = address(bytes20(uint(-1)));
    mapping (address => mapping (address => mapping (bytes4 => bool))

    function canCall(
```

```
        address src, address dst, bytes4 sig
) public view returns (bool) {
        return _acl[src][dst][sig]
            || _acl[src][dst][ANY_SIG]
            || _acl[src][ANY_ADDRESS][sig]
            || _acl[src][ANY_ADDRESS][ANY_SIG]
            || _acl[ANY_ADDRESS][dst][sig]
            || _acl[ANY_ADDRESS][dst][ANY_SIG]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][sig]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][ANY_SIG];
    }

    function permit(address src, address dst, bytes4 sig) public only
        _acl[src][dst][sig] = true;
        emit LogPermit(src, dst, sig);
    }

    function forbid(address src, address dst, bytes4 sig) public only
        _acl[src][dst][sig] = false;
        emit LogForbid(src, dst, sig);
    }

    function permit(address src, address dst, string sig) external {
        permit(src, dst, bytes4(keccak256(sig)));
    }

    function forbid(address src, address dst, string sig) external {
        forbid(src, dst, bytes4(keccak256(sig)));
    }

    function permitAny(address src, address dst) external {
        permit(src, dst, ANY_SIG);
    }

    function forbidAny(address src, address dst) external {
        forbid(src, dst, ANY_SIG);
    }
```

```
44  }
```

在这个合约里，有调用者地址、被调用合约地址和函数签名三个主要参数。通过配置ACL的访问策略，可以精确地定义和控制函数访问行为及权限。合约内置了ANY的常量，匹配任意函数，使访问粒度的控制更加便捷。这个模板合约实现了强大灵活的功能，足以满足所有类似权限控制场景的需求。

提升存储和计算的效率

迄今为止，在上述的推演过程中，更多的是对智能合约编程做加法。但相比传统软件环境，智能合约上的存储和计算资源更加宝贵。因此，如何对合约做减法也是用好Solidity的必修课程之一。

选取合适的变量类型

显式的问题可通过EVM编译器检测出来并报错；但大量的性能问题可能被隐藏在代码的细节中。

Solidity提供了非常多精确的基础类型，这与传统的编程语言大相径庭。下面有几个关于Solidity基础类型的小技巧。

在C语言中，可以用short\int\long按需定义整数类型，而到了Solidity，不仅区分int和uint，甚至还能定义uint的长度，比如uint8是一个字节，uint256是32个字节。这种设计告诫我们，能用uint8搞定的，绝对不要用uint16！

几乎所有Solidity的基本类型，都能在声明时指定其大小。开发者一定要有效利用这一语法特性，编写代码时只要满足需求就尽可能选取小的变量类型。

数据类型bytes32可存放 32 个（原始）字节，但除非数据是bytes32或bytes16这类定长的数据类型，否则更推荐使用长度可以变化的bytes。bytes类似byte[]，但在外部函数中会自动压缩打包，更节省空间。

如果变量内容是英文的，不需要采用UTF-8编码，在这里，推荐bytes而不是string。string默认采用UTF-8编码，所以相同字符串的存储成本会高很多。

紧凑状态变量打包

除了尽可能使用较小的数据类型来定义变量，有的时候，变量的排列顺序也非常重要，可能会影响到程序执行和存储效率。

其中根本原因还是EVM，不管是EVM存储插槽（Storage Slot）还是栈，每个元素长度是一个字（256位，32字节）。

分配存储时，所有变量（除了映射和动态数组等非静态类型）都会按声明顺序从位置0开始依次写下。

在处理状态变量和结构体成员变量时，EVM会将多个元素打包到一个存储插槽中，从而将多个读或写合并到一次对存储的操作中。

值得注意的是，使用小于32 字节的元素时，合约的gas使用量可能高于使用32字节元素时。这是因为EVM每次会操作32个字节，所以如果元素比32字节小，必须使用更多的操作才能将其大小缩减到所需。这也解释了Solidity中最常见的数据类型，例如int，uint，byte32，为何都刚好占用32个字节。

所以，当合约或结构体声明多个状态变量时，能否合理地组合安排多个存储状态变量和结构体成员变量，使之占用更少的存储位置就十分重要。

例如，在以下两个合约中，经过实际测试，Test1合约比Test2合约占用更少的存储和计算资源。

```
contract Test1 {
    //占据2个slot, "gasUsed":188873
    struct S {
        bytes1 b1;
        bytes31 b31;
        bytes32 b32;
    }
    S s;
    function f() public {
        S memory tmp = S("a","b","c");
        s = tmp;
    }
}

contract Test2 {
    //占据1个slot, "gasUsed":188937
    struct S {
        bytes31 b31;
        bytes32 b32;
        bytes1 b1;
    }
    // ……
}
```

优化查询接口

查询接口的优化点很多，比如一定要在只负责查询的函数声明中添加view修饰符，否则查询函数会被当成交易打包并发送到共识队列，被全网执行并被记录在区块中；这将大大增加区块链的负担，占用宝贵的链上资源。

再如，不要在智能合约中添加复杂的查询逻辑，因为任何复杂查询代码都会使整个合约变得更长更复杂。读者可使用上文提及的WeBASE数据导出组件，将链上数据导出到数据库中，在链下进行查询和分析。

缩减合约binary长度

开发者编写的Solidity代码会被编译为binary code，而部署智能合约的过程实际上就是通过一个transaction将binary code存储在链上，并取得专属于该合约的地址。

缩减binary code的长度可节省网络传输、共识打包数据存储的开销。例如，在典型的存证业务场景中，每次客户存证都会新建一个存证合约，因此，应当尽可能地缩减binary code的长度。

常见思路是裁剪不必要的逻辑，删掉冗余代码。特别是在复用代码时，可能引入一些非刚需代码。以上文ACL合约为例，支持控制合约函数粒度的权限。

```
function canCall(
        address src, address dst, bytes4 sig
    ) public view returns (bool) {
        return _acl[src][dst][sig]
            || _acl[src][dst][ANY_SIG]
            || _acl[src][ANY_ADDRESS][sig]
            || _acl[src][ANY_ADDRESS][ANY_SIG]
            || _acl[ANY_ADDRESS][dst][sig]
            || _acl[ANY_ADDRESS][dst][ANY_SIG]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][sig]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][ANY_SIG];
    }
```

但在具体业务场景中，只需要控制合约访问者即可，通过删除相应代码，进一步简化使用逻辑。这样一来，对应合约的binary code长度会大大缩小。

```
function canCall(
        address src, address dst
) public view returns (bool) {
        return _acl[src][dst]
            || _acl[src][ANY_ADDRESS]
            || _acl[ANY_ADDRESS][dst];
    }
```

另一种缩减binary code的思路是采用更紧凑的写法。

经实测，采取如上短路原则的判断语句，其binary长度会比采用if-else语法的更短。同样，采用if-else的结构，也会比if-if-if的结构生成更短的binary code。

最后，在对binary code长度有极致要求的场景中，应当尽可能避免在合约中新建合约，这会显著增加binary的长度。例如，某个合约中有如下的构造函数：

```
constructor() public {
        // 在构造器内新建一个新对象
        _a = new A();
}
```

我们可以采用在链下构造A对象，并基于address传输和固定校验的方式，来规避这一问题。

```
constructor(address a) public {
        A _a = A(a);
        require(_a._owner == address(this));
}
```

当然，这样也可能会使合约交互方式变得复杂。但其提供了有效缩短binary code长度的捷径，需要在具体业务场景中做权衡取舍。

保证合约可升级

经典的三层结构

通过前文方式，我们尽最大努力保持合约设计的灵活性；翻箱倒柜复用了轮子；也对发布合约进行全方位、无死角的测试。除此之外，随着业务需求变化，我们还将面临一个问题：如何保证合

约平滑、顺利的升级?

作为一门高级编程语言，Solidity支持运行一些复杂控制和计算逻辑，也支持存储智能合约运行后的状态和业务数据。不同于WEB开发等场景的应用-数据库分层架构，Solidity语言甚至没有抽象出一层独立的数据存储结构，数据都被保存到了合约中。

但是，一旦合约需要升级，这种模式就会出现瓶颈。

在Solidity中，一旦合约部署发布后，其代码就无法被修改，只能通过发布新合约去改动代码。假如数据存储在老合约，就会出现所谓的“孤儿数据”问题，新合约将丢失之前运行的历史业务数据。

这种情况，开发者可以考虑将老合约数据迁移到新合约中，但此操作至少存在两个问题：

1. 迁移数据会加重区块链的负担，产生资源浪费和消耗，甚至引入安全问题；
2. 牵一发而动全身，会引入额外的迁移数据逻辑，增加合约复杂度。

一种更合理的方式是抽象一层独立的合约存储层。这个存储层只提供合约读写的最基本方法，而不包含任何业务逻辑。

在这种模式中，存在三种合约角色：

- 数据合约：在合约中保存数据，并提供数据的操作接口。
- 管理合约：设置控制权限，保证只有控制合约才有权限修改数据合约。
- 控制合约：真正需要对数据发起操作的合约。

具体的代码示例如下：

数据合约：

```
contract FruitStore is BasicAuth {
    address _latestVersion;
    mapping(bytes => uint) _fruitStock;

    modifier onlyLatestVersion() {
       require(msg.sender == _latestVersion);
        _;
    }

    function upgradeVersion(address newVersion) public {
        require(msg.sender == _owner);
        _latestVersion = newVersion;
    }

    function setFruitStock(bytes fruit, uint stock) onlyLatestVersion
        _fruitStock[fruit] = stock;
    }
}
```

管理合约：

```
contract Admin is BasicAuth {
    function upgradeContract(FruitStore fruitStore, address newControl
        fruitStore.upgradeVersion(newController);
    }
}
```

控制合约：

```
contract FruitStoreController is BasicAuth {
    function upgradeStock(bytes fruit, uint stock) isAuthorized extern
        fruitStore.setFruitStock(fruit, stock);
    }
}
```

一旦函数的控制逻辑需要变更，开发者只需修改FruitStoreController控制合约逻辑，部署一个新合约，然后使用管理合约Admin修改新的合约地址参数就可轻松完成合约升级。这种方法可消除合约升级中因业务控制逻辑改变而导致的数据迁移隐患。

但天下没有免费的午餐，这种操作需要在可扩展性和复杂性之间需要做基本的权衡。首先，数据和逻辑的分离降低了运行性能。其次，进一步封装增加了程序复杂度。最后，越是复杂的合约越会增加潜在攻击面，简单的合约比复杂的合约更安全。

通用数据结构

到目前为止，还存在一个问题，假如数据合约中的数据结构本身需要升级怎么办?

例如，在FruitStore中，原本只保存了库存信息，现在由于水果销售店生意发展壮大，一共开了十家分店，需要记录每家分店、每种水果的库存和售出信息。

在这种情况下，一种解决方案是采用外部关联管理方式：创建一个新的ChainStore合约，在这个合约中创建一个mapping，建立分店名和FruitStore的关系。

此外，不同分店需要创建一个FruitStore的合约。为了记录新增的售出信息等数据，我们还需要新建一个合约来管理。

假如在FruitStore中可预设一些不同类型的reserved字段，可帮助规避新建售出信息合约的开销，仍然复用FruitStore合约。但这种方式在最开始会增加存储开销。

一种更好的思路是抽象一层更为底层和通用的存储结构。

代码如下：

```
contract commonDB  is BasicAuth {
    mapping(bytes => uint) _uintMapping;

    function getUint(bytes key) external view returns(uint) {
        return _uintMapping[key];
    }

    function setUint(bytes key, uint value) isAuthorized onlyLatestVe
        _uintMapping[key] = value;
    }

}
```

类似的，我们可加入所有数据类型变量，帮助commonDB应对和满足不同的数据类型存储需求。

相应的控制合约可修改如下：

```
contract FruitStoreControllerV2 is BasicAuth {
    function upgradeStock(bytes32 storeName, bytes32 fruit, uint stock
        isAuthorized external {
        commonDB.setUint(sha256(storeName, fruit), stock);
        uint result = commonDB.getUint(sha256(storeName, fruit));
    }
}
```

使用以上存储的设计模式，可显著提升合约数据存储灵活性，保证合约可升级。

众所周知，Solidity既不支持数据库，使用代码作为存储entity，也无法提供更改schema的灵活性。但是，通过这种KV设计，可以使存储本身获得强大的可扩展性。

总之，**没有一个策略是完美的，优秀的架构师善于权衡**。智能合约设计者需要充分了解各种方案的利弊，并基于实际情况选择合适的设计方案。

（近看羚羊峡，深藏玄机，变幻莫测，曲径通幽。）

总 结

文至于此，希望激起读者对在Solidity世界生存与进化的兴趣。“若有完美，必有谎言”，软件开发的世界没有银弹。本文行文过程就是从最简单的合约逐步完善和进化的过程。

在Solidity编程世界中，生存与进化都离不开三个关键词：安全、可复用、高效。生命不息，进化不止。短短一篇小文难以穷尽所有生存进化之术，希望这三个关键词能帮助大家在Solidity的世界里翱翔畅游，并不断书写辉煌的故事和传说：）

特别感谢：文中羚羊峡拍摄者Jeniffer

下期预告

FISCO BCOS

FISCO BCOS的代码完全开源且免费

下载地址↓↓↓

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


FISCO BCOS
长按二维码关注
下载最新区块链应用案例集