如何用solidity打造可升级智能合约

Gabriel

+关注

14 人赞同了该文章

前言

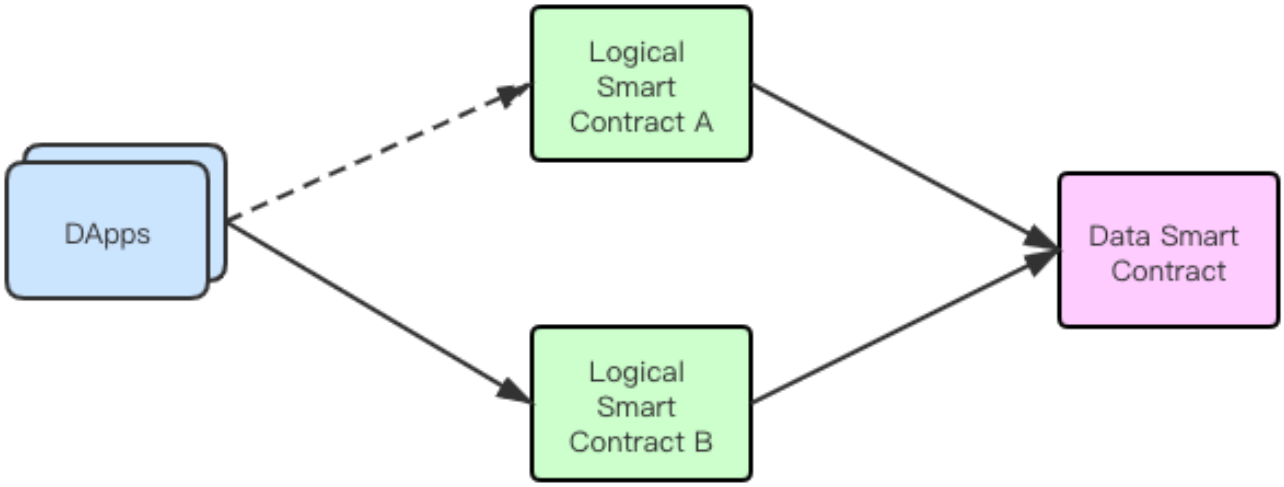
用过solidity语言的人都知道，如果智能合约有bug或者需要扩展新的特性，将是一个巨大的挑战。源码本身的修改程序员都能搞定，最大的挑战是solidity每次部署合约后，合约地址都会改变，这样就面临很多棘手的问题：

所有的用到了该合约的DApps都需要修改合约地址来适配新的合约
链上合约里的数据要迁移到新的合约里面，一般会对旧合约做快照，然后把数据导入到新合约中。这种方式的不足之处在于工作量大，需要脚本扫链，很容易出错。一旦出错，后果可能是无法承担的

因此，程序员就自然而然的思考能不能打造一种可升级的智能合约架构呢？答案是肯定的，本文将详细阐述如何做到这一点。

正文

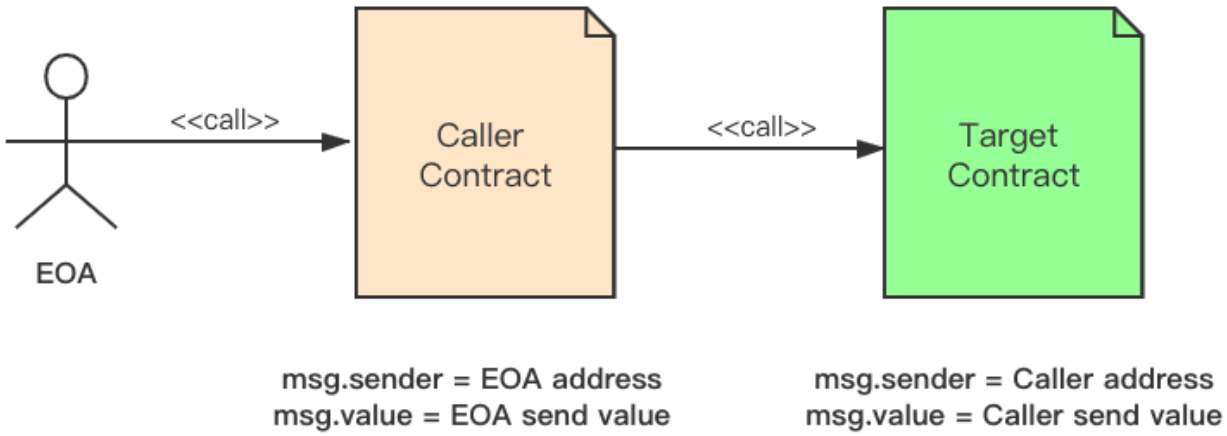
要打造可升级的智能合约架构，就必须做到数据和逻辑分离。数据保存在一个合约里面，该合约保持稳定，避免升级；逻辑保存在另外一个合约里面，该合约可以升级。毕竟每次升级都是修改的代码逻辑。这样的设计，正好完美的解决了上面提到的第二个痛点。

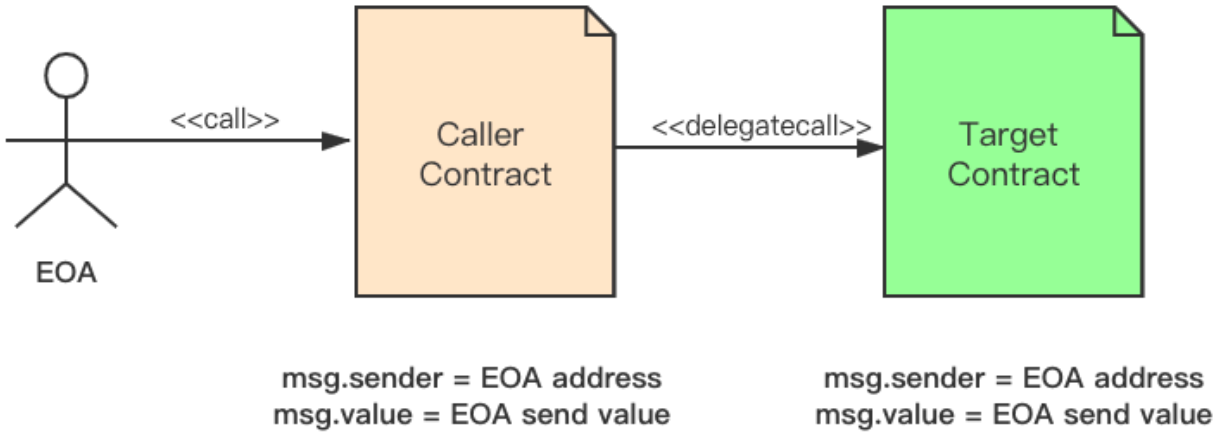


如图 1所示，Data合约和Logical合约A开发完成，部署到链上，Logical合约A需要用到的数据都通过合约间调用存储在Data合约里面。两个合约形成了一个整体服务。DApps只需要调用Logical合约A的ABI即可。突然有一天发现了严重的bug，于是修复完bug后，部署Logical合约B到链上，修改DApps指向新的Logical合约B，从而达到了升级合约的目的，完美的避免了数据的迁移。

有细心的程序员这时候会提出质疑，上述方法虽然能解决数据迁移的痛点，但是要修改所有使用该智能合约的DApps，还是非常的麻烦。那么别着急，下面介绍一种更高级的数据和逻辑分离的方法，同时解决上述的两个痛点。

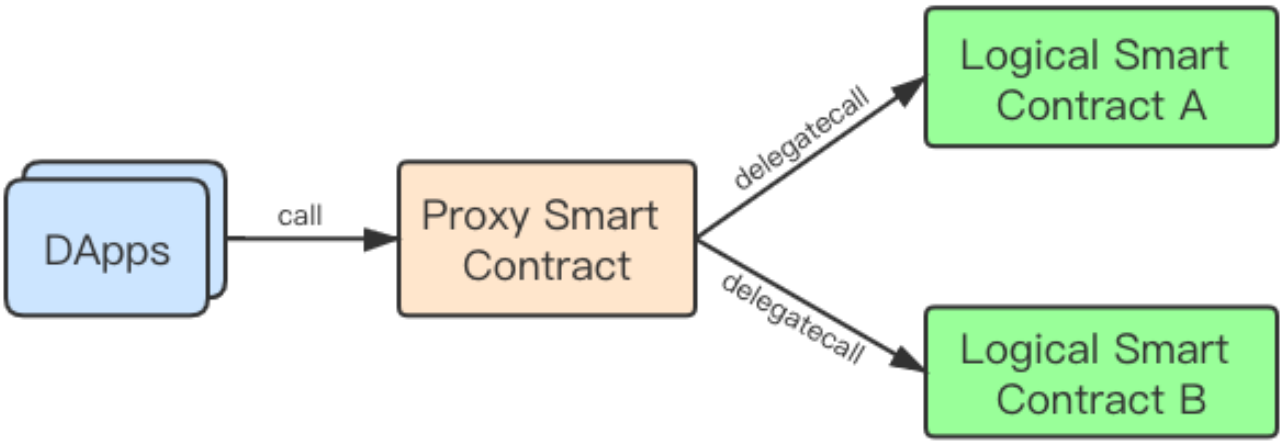
在开始介绍新的方法之前，要引入一种solidity里面的概念：delegate call。Solidity提供了几种合约之间调用的方法，最常用的是call()，大部分合约都是使用的call()。现在说的是不太常见的delegatecall()，这里只是简单的说明一下call()和delegatecall()的区别，详细的介绍请大家自行Google。





如图 2所示，是典型的call()调用，重点关注两个合约的上下文。Caller合约的msg.sender和msg.value都是用户（EOA）发送的交易中的数据。Target合约的上下文变了，msg.sender和msg.value变成了Caller合约的交易数据，不再是用户（EOA）的了。这个还是很好理解的。

我们再看图 3，Caller合约跟图 2没有差别，但Target的msg.sender和msg.value居然还是用户（EOA）的交易数据，很神奇。这就是delegatecall()跟call()的本质区别，delegatecall()并不会切换合约的上下文，也就是Target合约使用的是Caller的存储空间，修改的也是Caller的存储空间。



言归正传，我们回到如何打造可升级智能合约这个话题上来。如图 4所示，我们使用一个Proxy合约来存储数据，Proxy合约使用delegatecall去调用Logical合约A。这样一来，Logical合约A读取和写入的数据都是在Proxy合约里面。升级合约也变得简单起来。部署完Logical合约B后，只需要发一条交易到Proxy，修改Proxy指向新的Logical合约B即可。此升级过程对DApps透明，DApps完全感知不到合约进行了升级，于是完美解决了本文开头提到的两个痛点，既不要升级DApps，也不需要进行数据迁移。

这个时候，挑剔的程序员可能会提出新的挑战，如果要新增特性，需要修改数据结构怎么办？上面说到的只是修改逻辑，而现实中修改数据结构也是很常见的需求。这个也是有办法解决的，就是在设计数据结构的时候全部用Key-Value的Map数据结构，这种结构的好处是原则上可以无限扩展。当需要保存新的数据，只要重新定义一个新的Key就可以了。这样做的代价就是代码的可读性下降不少，本来一些可以用结构体很容易做到的，却要生生拆成Key-Value的Map数据。所以我的建议是可以使用折中方案，就是开发第一版的时候还是使用常见的数据结构，可读性好，开发周期也短一些。但同时定义一些常见类型的Key-Value的Map用于扩展。当升级合约的时候需要新增数据，就使用这些预留好的Map去扩展，毕竟这样的需求并不是那么常见。

最后需要提醒一下程序员，使用Proxy-Delegate的架构实现可升级智能合约，是有约束条件的，就是Proxy和Delegate合约的**状态变量**的**定义**要保持一致，**顺序**也要保持一致。切记切记，否则出现问题是很难调试的。

结束语

到此为止，相信程序员已经知道如何去设计自己的可升级solidity智能合约了。但还是需要实践才能领悟得更深刻。Wanchain的EOS跨链智能合约用的就是Proxy-Delegate架构，可以作为很好的参考。

github.com/wanchain/wan

附录

Proxy合约模板：

```
pragma solidity ^0.4.24;

contract Proxy {
address public owner;
    event Upgraded(address indexed implementation);
    address internal _implementation;

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

    modifier onlyOwner() {
 require(msg.sender == owner, "Not owner");
 _;
}
    function implementation() public view returns (address) {
        return _implementation;
}

    function upgradeTo(address impl) public onlyOwner {
        require(impl != address(0), "Cannot upgrade to invalid address");
        require(impl != _implementation, "Cannot upgrade to the same implementation");
        _implementation = impl;
        emit Upgraded(impl);
}

    function () external payable {
        address _impl = _implementation;
        require(_impl != address(0), "implementation contract not set");

        assembly {
            let ptr := mload(0x40)
```

```
            calldatacopy(ptr, 0, calldatasize)
            let result := delegatecall(gas, _impl, ptr, calldatasize, 0, 0)
            let size := returndatasize
            returndatacopy(ptr, 0, size)

            switch result
            case 0 { revert(ptr, size) }
            default { return(ptr, size) }
        }
    }
}
```