用代理模式实现智能合约的可升级性

以太坊最大的优势之一是，转移资金的每一笔交易，部署的每一个合约，以及调用合约的每一笔交易，在我们称之为区块链的公共账本上是不可变的。我们无法隐瞒或修改任何交易。带来的巨大的好处是，以太坊网络上的任何节点都可以验证每一笔交易的有效性和状态，使以太坊成为一个非常健壮的去中心化系统。

但最大的缺点是，你不能更改你的智能合约部署后的源代码。开发中心化应用(如Facebook或Airbnb)的开发人员习惯于为了修复漏洞或引入新功能而频繁进行更新。这在以太坊的传统模式下是不可能做到的。

还记得臭名昭著的导致150000个ETH被偷的Parity Wallet Multisig黑客事件吗?在攻击期间，Parity multisig钱包合约中的一个漏洞被利用，钱包的资金被抽干。唯一能做的就是尝试比黑客更快，并利用同样的漏洞在攻击发生后攻击剩余的钱包，将ETH重新分配给它们的合法所有者。

如果有一种方法可以在部署智能合约后更新源代码就好了

* 代理模式（Proxy Pattern）的引入

虽然不可能升级已经部署的智能合约的代码，但可以搭建一个代理合约架构，允许你使用新部署的合约，就像你的主逻辑已经升级一样。

代理架构模式是这样的:所有消息调用都经过一个代理合约，该合约将它们重定向到最新部署的合约逻辑。要升级，将部署你的合约的新版本，并更新代理以引用/指向新合约地址。

图形用户界面, 文本, 应用程序, Word

描述已自动生成

作为实现zeppelin\_os的工作的一部分，Zeppelin一直在研究几种代理模式。所探讨的三个选择是

1. Inherited Storage（继承存储）
2. Eternal Storage（永久存储）
3. Unstructured Storage（非结构化存储）

这三种模式都依赖于底层delegatecalls，即委托调用。尽管Solidity提供了一个delegatecall函数，但它只返回true/false代表是否调用成功，并且不允许你管理返回的数据。（译者注：老版本的solidity是这样的，后来新的solidity版本是可以返回调用结果的）

在我们深入讨论之前，有两个重要的概念需要理解

* 当对一个合约不支持的函数进行调用时，将调用fallback函数。你可以编写一个自定义的fallback函数来处理这种情况。代理合约使用一个自定义的fallback函数将调用重定向到其他合约实现。
* 每当一个合约A委托调用另一个合约B时，它就会在合约A的上下文中执行合约B的代码。这意味着msg.value和msg.sender将被保留，每次存储修改都会影响合约A的存储。

Zeppelin的代理合约，被所有的代理模式共享，为了这个特殊的原因实现了它自己的delegatecall函数，它返回调用逻辑合约的结果值。如果你计划使用Zeppelin的代理合约代码，你应该充分了解详细的代码。让我们探索它是如何工作的，并理解它用来实现这一点的assembly opcodes。请参考Solidity的Assembly文档以获得更多信息。

文本

描述已自动生成

为了将调用委托给另一个solidity合约的函数，我们必须将代理合约接收到的msg.data传递给它。由于msg.data是bytes类型的，这是一种动态的数据结构，它的大小是可变的，其大小存储在msg.data的第一个word size(32字节)中。如果我们只想提取实际的数据，则需要跳过第一个word size(32字节)，并从msg.data的0x20(32字节)开始。但是，我们将使用两个操作码来代替此操作。我们将使用calldatasize来获取msg.data的大小，并使用calldatacopy将其复制到我们的ptr变量中。

注意我们是如何初始化ptr变量的。在solidity中，位于0x40位置的内存槽(memory slot)是特殊的，因为它包含下一个可用的空闲内存指针(free memory pointer)的值。每次将一个变量直接保存到内存中时，都应该检查0x40处的值，从而确定应该将其保存到哪里。现在我们知道了允许保存变量的位置，我们可以使用calldatacopy将大小为calldatasize的calldata从call data的0开始复制到ptr的地址。

图片包含 文本

描述已自动生成

让我们看看assembly块中使用委托调用操作码的下面一行



参数：

* 我们传入执行函数所需的gas
* \_impl调用的逻辑合约的地址
* ptr内存指针，数据从哪里开始
* calldatasize我们传递的数据的大小。
* 0表示数据输出，表示调用逻辑合约返回的值。这是没有用的，因为我们还不知道输出数据的大小，因此不能将其分配给一个变量。稍后我们仍然可以使用returndata opcode访问这些信息
* 0表示数据输出的大小。这是没有用的，因为我们没有机会创建一个临时变量来存储数据输出的大小，因为我们在调用另一个合约之前不知道它的大小。我们可以使用另一种方法来获得这个值，稍后调用returndatasize操作码

下一行使用returndatasize操作码获取返回数据的大小

图片包含 图表

描述已自动生成

我们使用返回数据的大小将返回数据的内容复制到我们的ptr变量，使用一个辅助操作码函数returndatacopy

徽标, 公司名称

描述已自动生成

最后，switch语句要么返回返回的数据，要么在出错时抛出异常。



很好，现在我们有了一种从逻辑合约检索适当结果值的方法。

现在我们已经了解了代理合约的工作原理，让我们来看看Zeppelin提出的三个模式:使用继承存储、非结构化存储和永久存储的可升级性。

这三种方法有不同的方法来解决相同的技术难题:如何确保逻辑合约不会覆盖代理合约中用于可升级性的状态变量。

任何代理架构模式的主要关注点都是如何处理存储分配(storage allocation)。请记住，由于我们使用一个合约来负责存储，而用另一个合约来负责逻辑，因此它们中的任何一个都可能覆盖已经使用的存储槽。这意味着,如果代理合约的一个状态变量（在某个存储槽）用来跟踪最新的逻辑合约的地址，然而逻辑合约不知道它，然后逻辑合约可以在同一个槽存储其他数据从而重写代理合约存储的关键信息。Zeppelin的三种方法提供了不同的方法来构建你的系统，从而通过代理模式使你的合约可升级。

* 使用继承存储的可升级性

继承存储方法依赖于使逻辑合约包含代理合约所需的存储结构。代理合约和逻辑合约都继承相同的存储结构，以确保两者都可存储必要的代理合约的状态变量。

在探索这种方法时，我们尝试使用Registry合约来跟踪逻辑合约的不同版本。为了升级到一个新的逻辑合约，你将需要在Registry合约中将其注册为一个新版本，并要求代理合约升级到它。注意，Registry不会影响存储机制;事实上，它可以在这篇文章中展示的任何存储模式中实现。

图示

描述已自动生成

如何初始化：

* 部署一个Registry合约
* 部署合约的初始版本(v1)。确保它继承了Upgradeable契约
* 向Registry注册你的初始版本合约地址
* 要求Registry合约创建一个UpgradeabilityProxy实例
* 调用UpgradeabilityProxy来升级到合约的初始版本

如何升级：

* 部署从初始版本继承的合约新版本(v2)，以确保它保留代理合约的存储结构和合约初始版本中的存储结构。
* 将新版本的合约登记到Registry
* 调用UpgradeabilityProxy实例来升级到新注册的版本。

我们可以在未来部署的逻辑合约中引入升级的函数以及新函数和新状态变量，但仍然调用相同的UpgradeabilityProxy合约。

* 使用永久存储的可升级性

在永久存储模式中，存储在一个独立的合约中定义，代理合约和逻辑合约都继承这个合约。存储合约包含逻辑合约将需要的所有状态变量，由于代理合约也知道这些状态变量，它可以定义自己的状态变量以实现可升级性，而不必担心它们被覆盖。注意，逻辑合约的所有未来版本都不应该定义任何其他状态变量。逻辑合约的所有版本都必须始终使用开始时定义的永久存储结构。（译者注：局限性很大）

Zeppelin仓库中提供的这个实现还引入了代理所有权(proxy ownership)的概念。一个proxy owner是唯一可以升级代理合约以指向新的逻辑合约的地址，也是唯一可以转移所有权的地址。

图示

描述已自动生成

如何初始化：

* 部署一个EternalStorageProxy实例
* 部署合约的初始版本(v1)
* 调用你的EternalStorageProxy实例来升级到初始版本的地址
* 如果你的逻辑合约依赖于它的构造函数来设置一些初始状态，那将不得不在它链接到代理合约之后重新做，因为代理合约的存储不知道那些值。EternalStorageProxy有一个upgradeToAndCall函数，专门用于在代理升级到新版本合约之后，在你的逻辑合约上调用一些函数来重做设置。

如何升级：

* 部署一个新版本的合约(v2)，确保它拥有永久的存储结构。
* 调用你的EternalStorageProxy实例来升级到新版本。

直观的方法，没有token逻辑合约的重大开销。未来的逻辑联系可以升级现有的方法并引入新的方法，但不应该引入新的状态变量。

* 使用非结构化存储的可升级性

非结构化存储模式与继承存储类似，但不需要逻辑合约来继承与可升级性相关的任何状态变量。此模式使用代理合约中定义的非结构化存储槽来存储可升级性所需的数据。

在代理合约中，我们定义了一个constant变量，当进行哈希处理时，它应该提供一个足够随机的存储位置来存储代理合约应该调用的逻辑合约的地址。（译者注：这里存储位置的随机性非常重要）

文本

中度可信度描述已自动生成

因为constant状态变量不占用存储槽，所以不需要担心implementationPosition被逻辑合约意外覆盖。基于Solidity在存储中布局状态变量的方式，这个存储槽被逻辑合约中定义的其他东西使用时发生冲突的可能性极小。

通过使用这种模式，任何逻辑合约版本都不需要知道代理合约的存储结构，但是所有未来的逻辑合约的版本都必须继承由其祖先版本声明的状态变量。就像在Inherited Storage模式（继承模式）中一样，未来升级的通证逻辑合约既可以升级现有的函数，也可以引入新的函数和新的状态变量。

Zeppelin实验室的代码库中提供的这个实现也使用了代理所有权（proxy ownership）的概念。代理所有者是唯一可以升级代理合约以指向新的逻辑合约的地址，也是唯一可以转移所有权的地址。

图示

描述已自动生成

如何初始化

* 部署一个OwnedUpgradeabilityProxy实例
* 部署逻辑合约的初始版本(v1)
* 调用OwnedUpgradeabilityProxy实例来升级指向到初始逻辑合约版本的地址
* 如果逻辑合约依赖于它的构造函数来设置一些初始状态，那将不得不在它链接到代理合约之后重新做，因为代理合约的存储不知道那些值。OwnedUpgradeabilityProxy有一个upgradeToAndCall函数，专门用来在你的逻辑合约上调用一些函数来在代理合约升级到它之后重做设置。

如何升级

* 部署逻辑合约的新版本v2，确保它继承了以前版本中使用的状态变量结构。
* 调用OwnedUpgradeabilityProxy实例来升级到新逻辑合约版本的地址。

总结：

这种方法很好，因为它不需要通证逻辑合约意识到它是代理合约系统的一部分。

* 关于可升级性

重要:如果你的逻辑合约依赖于它的构造函数来设置一些初始状态，这必须在代理合约升级到你的逻辑合约后重新做。例如，逻辑合约经常继承自Zeppelin的Ownable合约实现。当你的逻辑合约从Ownable继承时，它也继承了Ownable的构造函数，该构造函数在合约创建时设置了谁是owner。当你链接代理合约到逻辑合约以使用逻辑合约时，从代理合约的角度来看，owner的值将丢失。

升级代理合约的一个常见模式是，代理合约立即调用逻辑合约上的initialize方法。initialize方法应该模仿你通常在构造函数中放入的所有内容。你还需要包含一个标志位，以防止多次初始化逻辑合约。

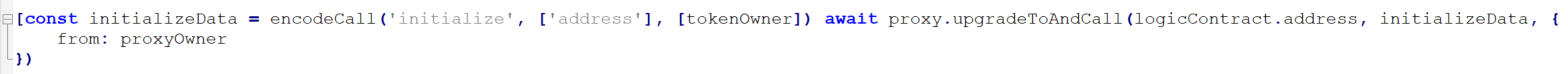
你的逻辑合约应该看起来像这样:

文本

描述已自动生成

根据你的部署策略，你可以有一个helper deployer合约，也可以单独部署代理合约和逻辑合约。

如果单独部署，则可以使用upgradeToAndCall将代理合约链接到逻辑合约，其内容大概如下所示



* 结论

代理模式的概念已经出现了一段时间，但由于其复杂性、担心引入安全漏洞以及绕过区块链的不可变性而引起的争议，还没有得到广泛的采用。过去的解决方案也相当不灵活，需要严格限制未来的逻辑合约可以修改和添加的内容。然而，从开发人员的角度来看，很明显，对合约可升级的能力有很大的需求。Zeppelin为探索的三种模式提供了代码和测试，以帮助开发人员构建他们的项目，使其具备可升级性。

尽管代理模式的概念并不新鲜，但它的采用仍处于早期阶段，看到更先进的DApp架构正在使用这种模式实现是令人兴奋的。

原文链接：

https://blog.openzeppelin.com/proxy-patterns/