升级智能合约

X 上次修改时间: @Xin Cheng, 2024年2月21日

查看贡献者

以太坊上的智能合约是在以太坊虚拟机 (EVM) 中运行的自执行程序。 这些程序在设计上不可变，这意味着一旦部署合约，就无法对业务逻辑进行任何更新。
尽管不可变性对于智能合约的去信任性、去中心化和安全性是必要的，但在某些情况下，它可能具有一定的缺点。 例如，不可变的代码可能会使开发者无法修复存在漏洞的合约。
然而，随着对改进智能合约的研究力度不断加强，引入了几种升级模式。 这些升级模式使开发者能够通过将业务逻辑放置在不同的合约中来升级智能合约（同时保持不可变性）。

前提条件

你应当充分了解智能合约、智能合约结构以及以太坊虚拟机 (EVM)。 本指南还假定读者掌握了编写智能合约的知识。

什么是智能合约升级?

智能合约升级涉及更改智能合约的业务逻辑，同时保留合约的状态。 重要的是要澄清，可升级性和可变性并不是相同的概念，尤其是在智能合约的背景下。
你仍然无法更改在以太坊网络地址上部署的程序。 但是你可以更改与用户交互时执行的智能合约代码。
这可以通过以下方式完成：

1. 创建智能合约的多个版本并从旧合约向新合约的实例迁移状态（即数据）。
2. 创建独立的合约以存储业务逻辑和状态。
3. 使用代理模式将不可修改的代理合约中的函数调用委托给可修改的逻辑合约。
4. 创建一个不可变的主合约，与灵活的卫星合约进行接口交互，并依赖这些卫星合约来执行特定的功能。
5. 使用钻石模式将函数调用从代理合约委托给逻辑合约。

升级机制 #1：合约迁移

合约迁移是基于版本控制的概念，即创建和管理相同软件的独特状态。 合约迁移涉及部署现有智能合约的新实例，并将存储和余额转移到新合约中。
新部署的合约将有空的存储空间，使你能够从旧合约中恢复数据并将其写入新的实现。 之后，你需要更新与旧合约交互的所有合约，以反映新地址的变更。
合约迁移的最后一步是说服用户改为使用新合约。 新合约版本将保留用户的余额和地址，从而保持不可变性。 如果这是一个基于代币的合约，你还需要与交易所联系，废弃旧合约并使用新合约。
合约迁移是一种相对简单且安全的方法，用于升级智能合约而不影响用户交互。 然而，手动迁移用户存储和余额到新合约非常耗时，且可能产生高额的燃料费用。
更多关于合约迁移的信息。

升级机制 #2：数据分离

升级智能合约的另一种方法是将业务逻辑和数据存储分离到不同的合约中。 这意味着用户与逻辑合约进行交互，而数据存储在存储合约中。
逻辑合约包含当用户与应用程序进行交互时执行的代码。 它还保存了存储合约的地址，并与之进行交互以获取和设置数据。
同时，存储合约保存与智能合约相关的状态，例如用户的余额和地址。 请注意，存储合约由逻辑合约拥有，并在部署时配置为后者的地址。 这可以防止未经授权的合约调用存储合约或更新其数据。
默认情况下，存储合约是不可变的，但你可以用新的实现替换它所指向的逻辑合约。 这将修改在以太坊虚拟机中运行的代码，同时保持存储和余额不变。

使用这种升级方法需要在存储合约中更新逻辑合约的地址。 你还必须基于上文解释的原因，配置新的逻辑合约，并提供存储合约的地址。

与合约迁移相比，数据分离模式在实现上按理说更容易。 然而，你将需要管理多个合约并实施复杂的授权方案，以保护智能合约免受恶意升级的影响。

升级机制 #3：代理模式

代理模式还使用数据分离，将业务逻辑和数据保存在单独的合约中。 不过，在代理模式中，存储合约（称为代理）会在代码执行过程中调用逻辑合约。 这与数据分离方式相反，即逻辑合约调用存储合约。

下面介绍代理模式的原理：

1. 用户与代理合约进行交互，代理合约存储数据，但不保存业务逻辑。
2. 代理合约存储逻辑合约的地址，并使用 delegatecall 函数将所有函数调用委托给逻辑合约（逻辑合约保存业务逻辑）。
3. 调用转移到逻辑合约后，逻辑合约返回的数据将被检索并返回给用户。

使用代理模式需要了解 **delegatecall** 函数。 基本上，delegatecall 是一个操作码，它允许一个合约调用另一个合约，而实际的代码执行在调用合约过程中进行。 在代理模式中使用 delegatecall 的意义是，代理合约会读写其存储，并执行存储在逻辑合约中的逻辑，就像调用内部函数一样。

摘自 Solidity 文档：

*存在一种消息调用的特殊变体，名为 **delegatecall**，它与消息调用相同，但是目标地址的代码在调用合约的语境（即地址）下执行，并且 msg.sender 和 msg.value 不会更改其值。__这意味着合约在运行时可以从不同的地址动态加载代码。 存储、当前地址和余额仍参考调用合约，只是代码取自被调用地址。*

每当用户调用函数时，代理合约就会调用 delegatecall，因为它内置了一个 fallback 函数。 在 Solidity 编程中，当函数调用与合约中指定的函数不匹配时，将执行回退函数。

要使代理模式正常工作，需要编写一个自定义的回退函数，指定代理合约应如何处理它不支持的函数调用。 在这种情况下，代理的回退函数被编程为启动委托调用，并将用户的请求转发给当前的逻辑合约实现。

代理合约在默认情况下是不可变的，但可以创建更新了业务逻辑的新逻辑合约。 然后，只需更改代理合约中引用的逻辑合约的地址即可执行升级。

通过将代理合约指向新的逻辑合约，用户调用代理合约函数时执行的代码就会发生变化。 这样，我们就可以在不要求用户与新合约进行交互的情况下，升级合约的逻辑。

代理模式是一种流行的智能合约升级方法，因为它消除了与合约迁移相关的困难。 但是，代理模式的使用更为复杂，如果使用不当，可能会带来严重缺陷，例如函数选择器冲突。

更多关于代理模式的信息。

升级机制 #4：策略模式

这项技术受到策略模式的影响，该模式鼓励创建与其他程序进行接口交互的软件程序，以实现特定功能。 将策略模式应用于以太坊开发意味着构建一个智能合约，该合约可以调用其他合约的函数。

在这种情况下，主合约包含核心业务逻辑，但与其他智能合约（“卫星合约”）进行接口交互，以执行某些功能。 该主合约还存储每个卫星合约的地址，并可在卫星合约的不同实现之间切换。

你可以构建一个新的卫星合约，并为主合约配置新地址。 这允许你更改智能合约的*策略*（即实现新的逻辑）。

虽然策略模式与前面讨论的代理模式类似，但不同之处在于，与用户交互的主合约中包含了业务逻辑。 使用这种模式可以让你有机会在不影响核心基础架构的情况下对智能合约进行有限的更改。

这种模式的主要缺点是它主要适用于推出小规模升级。 此外，如果主合约被泄露（如被黑客攻击），则无法使用此升级方法。

升级机制 #5：钻石模式

钻石模式可以说是代理模式的改进。 钻石模式不同于代理模式，因为钻石代理合约可以将函数调用委托给多个逻辑合约。
钻石模式中的逻辑合约被称为 *“切面”* 。 要使钻石模式发挥作用，你需要在代理合约中创建一个映射，将函数选择器映射到不同的“切面”地址。
当用户调用函数时，代理合约会检查映射，以找到负责执行该函数的“切面”。 然后，它会调用 delegatecall（使用回退函数），并将调用重定向到相应的逻辑合约。
与传统的代理升级模式相比，钻石升级模式有一些优势：

1. 它允许你在不更改所有代码的情况下升级合约的一小部分。 使用代理模式进行升级需要创建一个全新的逻辑合约，即使是小规模升级也是如此。
2. 所有智能合约（包括代理模式下使用的逻辑合约）的大小限制为 24KB，这可能是一个限制 — 特别是对于需要更多函数的复杂合约。 钻石模式通过在多个逻辑合约中拆分函数，轻松解决了这一问题。
3. 代理模式采用了一种一揽子的访问控制方法。 可访问升级功能的实体能够更改*整个*合约。 但是，钻石模式支持模块化权限方法，可以限制实体升级智能合约中的某些功能。

更多关于钻石模式的信息。

升级智能合约的优缺点

优点	缺点
智能合约升级可以更轻松地修复部署后阶段发现的漏洞。	智能合约的升级否定了代码不变性的理念，这对去中心化和安全性都有影响。
开发者可以使用逻辑升级为去中心化应用程序添加新功能。	用户必须相信开发者不会随意修改智能合约。
由于漏洞可以快速修复，因而智能合约升级可以提高最终用户的安全性。	将升级功能编程到智能合约中又增加了一层复杂性，并增加了出现严重缺陷的可能性。
合约升级为开发者提供了更广阔的空间来试验不同的功能和不断改进去中心化应用程序。	升级智能合约的机会可能会促使开发者更快启动项目，但在开发阶段不进行尽职审查。 智能合约中不安全的访问控制或中心化会让恶意行为者更容易执行未经授权的升级。

升级智能合约的考量

1. 使用安全的访问控制或授权机制，防止未经授权的智能合约升级，尤其是在使用代理模式、策略模式或数据分离的情况下。 例如，限制升级功能的访问权限，只有合约所有者才能调用该功能。
2. 升级智能合约是一项复杂的活动，需要高度谨慎以防止引入漏洞。
3. 通过分散实施升级的流程，减少信任假设。 可行策略包括使用多重签名钱包合约来控制升级，或要求去中心化自治组织的成员投票批准升级。
4. 了解合约升级所涉及的费用。 例如，在合约迁移过程中，将状态（如用户余额）从旧合约复制到新合约可能需要不止一次交易，这意味着更多的燃料费用。
5. 考虑实施**时间锁**来保护用户。 时间锁指的是对系统变更强制执行的延迟。 时间锁可与多重签名治理系统相结合，以控制升级：如果拟议的操作达到了所需的批准阈值，则该操作将在预定的延迟期过后才会执行。

如果用户不同意拟议的更改（如逻辑升级或新的收费方案），时间锁会给他们一些时间退出系统。
如果没有时间锁，用户就需要相信开发者不会在没有事先通知的情况下对智能合约进行任意更改。
缺点是，时间锁限制了快速修补漏洞的能力。