AO 生态的胜利之匙：Web3 时代的微服务架构

作者：ArweaveOasis，来源：PermaDAO

本文讨论了采用微服务架构（或者说 Actor 模型）的优势，分析了它为应用开发带来的一定程度上合乎逻辑的复杂性。

@aoTheComputer 的发布无疑给整个 @ArweaveEco 生态乃至于整个 Web3 行业带来了一种全新的思考与实践。这不仅体现在广大投资者的关注度上，更体现在吸引大量优质开发者开始深度研究之上。

## 是什么阻碍了 Web3 的大规模采用？

很简单，因为值得人们使用的去中心化应用太少了。

基于 Web3 基础设施、开发工具、软件工程实践等方面的现状，很多类型的去中心化应用当前几乎是无法实现的。

在基础设施方面，我认为 AO 的出现填补了其中一部分重大的空白。但是，目前构建大型去中心化应用的工程的复杂性，仍然是令人望而生畏的。这使得我们无法在资源受限的情况下——在事物发展的初始阶段，通常如此——开发出更多样化的、更大规模、往往也意味着更棒、功能更丰富的去中心化应用。

不要相信那些类似“智能合约/链上程序应该就是很简单的，没有必要搞得太复杂”之类倒果为因的鬼话！

现实问题并不是“不想”，而是“不能”——臣妾做不到啊。

AO 是运行在 Arweave 上的计算机系统，旨在实现可验证的无限计算能力。它是 Actor Oriented（面向参与者）的简称。顾名思义，这说明运行在 AO 上的去中心化应用需要采用 Actor 模型为基础的设计和编程方法。

事实上，AO 并不是最早将 Actor 模型用于区块链（或者说“去中心化基础设施”）的。比如，TON 的智能合约就是使用 Actor 模型构建的。说到 TON，我个人觉得它和 AO 在某些方面颇有相似之处。

对于尚未深入了解 Web3 的 Web2 开发者来说，想要迅速理解 AO 或 TON 相对其他“单体区块链”的最大特色，一个方便的抓手是：把运行在它们之上的智能合约（链上程序）看作是“微服务”。而 AO 或 TON 是支持这些微服务运行的基础设施，比如 Kafka、Kubernetes 等。

作为一个 20 多年来主要专注于应用开发的一名资深 CRUD boy， 我个人非常乐见 AO、TON 这样的非单体区块链的出现，并对它们的发展充满期待。下面我想从一个应用开发者的视角，谈谈我对 AO 的看法，可能很多观点还不太成熟。也许部分应用开发者会心有戚戚焉，那就足矣。



## 将 Actor 模型应用于区块链，真的有必要吗？

答案是肯定的。看看已经取得“大规模采用”的 Web2 应用，你就会明白。

太多的架构师已经知道如何将 Web2 应用“搞大”：微服务架构（MSA）、事件驱动的架构（EDA）、消息通信机制、最终一致性模型、分片……这些东西，不管叫什么，总是与 Actor 模型共生共存的。其中的一些概念，甚至可以说只是一个事物的不同方面而已。所以在下面的行文中，我们不对“微服务”和 Actor 做区分，你可以认为它们就是同义词。

今日互联网的繁荣，离不开这些架构师的智慧。他们不断地探索、实践、总结，最终形成了一套完整的工程实践体系。

作为 Web3 基础设施，AO 做的很棒。至少，AO 作为（我眼中的）当前 Web3 领域的最佳去中心化消息代理，已经展现出巨大的潜力。我相信传统 Web2 应用的开发者由此可以马上理解其中的重大意义：倘若没有 Kafka 或者类 Kafka 的消息代理可用，你能想象现在很多大型的互联网应用“程序要怎么写”吗？

虽然 Actor 模型在很多方面具有理论上的优势，但是不管是 Actor 模型也好，微服务架构也好，在我看来，更多是开发者为了开发某些应用（特别是大型应用）所不得不承受之“痛”。

让我们用一个简单的例子来向非技术读者说明这一点。假设世界上所有银行都基于一个“世界计算机”来开展业务，而这个世界计算机是一个单体架构的系统。那么，当工商银行的客户“张三”向在招商银行开设账户的“李四”汇款 100 元的时候，开发者可以这样编写转账程序的代码：

1. 开始一个事务（或者说“交易”，它们在英文中是同一个词）；

2. 在张三的账户上扣减 100 元；

3. 在李四的账户上增加 100 元；

4. 提交事务。

以上步骤不管哪一步出现问题，比如说第三步，在李四的账户上增加金额，因为某种原因失败了，那么整个操作都会被回滚，就像什么都没有发生过一样。对了，程序这样写，我们称之为采用“强一致性”模型。

倘若这个世界计算机是个采用微服务架构（MSA）的系统呢？

那么，管理工商银行账户的那个微服务（或者说 Actor）与管理招商银行账户的那个微服务，几乎不太可能是同一个。我们先假设它们确实不是同一个，前者我们称为 Actor ICBC，后者我们称为 Actor CMB。此时，开发者可能需要这样编写转账的代码：

1. Actor ICBC 先记录好以下信息：“张三向李四转账 100 元”；Actor ICBC 在张三的账户上扣减 100 元，并向 Actor CMB 发送一条消息：“张三向李四转账100 元”；

2. Actor CMB 收到消息，在李四的账户上增加 100 元，然后向 Actor ICBC 发送一条消息“李四已收到张三汇入的 100 元”；

3. Actor ICBC 收到消息，记录好：“张三向李四转账 100 元，已成功”。

上面只是“一切都好”的过程。但是，如果某个步骤，比如说第二个步骤，“在李四的账户上增加 100 元”，出现了问题，怎么办？

开发者需要针对这个可能发生的问题，编写这样的处理逻辑：

Actor CMB 向 Actor ICBC 发送一条消息：“张三向李四转账 100 元，处理失败”。

Actor ICBC 收到消息，在张三的账户上增加 100 元，并记录：“张三向李四转账 100 元，已失败”。

程序这样写，我们称之为采用最终一致性模型。

以上，非技术读者应该能直观感受到开发单体架构的应用与开发 MSA 应用之间在工作量上的巨大差异了吧？要知道，上面所说的转账示例只是一个非常简单的应用而已，如果我们把它称之为应用，而不是功能的话。大型应用里面的功能往往比这样的例子要复杂的太多。

## 这个微服务应该多大？

换句话说，"这个微服务是不是太大了，应该一分为二？”

很不幸，这个问题没有标准答案，它是一门艺术。微服务越小，就越容易通过创建新实例并按需移动它们来优化系统。但是，微服务越小，开发人员就越难实施复杂的流程，正如上面展示的那样。

对了，将一个应用拆分为多个微服务，从数据库设计角度看，即所谓的“分片（Sharding）”。微服务架构的最佳实践之一，就是每个微服务仅使用一个属于自己的本地数据库。简单来说，分片允许水平扩展。当数据集变得太大，无法通过传统方式处理时，除了将它们拆分成更小的片段以外，别无他法（来进行扩展）。

回到微服务的拆分问题。为了更好的践行这门艺术，我们需要掌握一些思维工具的使用。DDD（领域驱动设计）的 “聚合（Aggregate）”就是这样一件你必须拥有的“大杀器”。我的意思是，它能帮助你摧毁软件设计中的“核心复杂性”。

我认为聚合是 DDD 在战术层面最为重要的一个概念。

什么是聚合？聚合在对象之间，特别是实体与实体之间划出边界。一个聚合一定包含且仅包含一个聚合根实体，以及可能包含不定数量的聚合内部实体（或者叫非聚合根实体）。

我们可以使用聚合这一概念对应用所服务的领域进行分析和建模；然后在编码的时候，就可以按照聚合来切分微服务。最简单的做法，就是将每个聚合实现为一个微服务。

不过，即使你的手艺再娴熟，这种事情你也不能保证第一次就做对。这个时候，一件让你可以尽快对建模结果进行验证、不行就推倒重来的工具，对你来说就弥足珍贵了。

## 还有什么东西可能构成大型 Web2 应用迁移到 AO 生态的障碍？

我想谈谈语言和程序运行时的问题。

AO 是一个数据协议。你可以认为它是一套定义 AO 网络中的各个“单元”如何实现协作的接口规范。目前，AO 的官方实现包含了一个基于 WASM 的虚拟机环境，以及一个编译为 WASM 的 Lua 运行时环境（ao-lib），旨在简化 AO 进程的开发。

Lua 是一种小而美的语言。一般认为，Lua 的优势在于它的轻量级和易于嵌入其他语言，这使得它在特定场景（比如游戏开发）中特别有用。但是，对于开发大型互联网应用来说，Lua 语言并不是主流的选择。大型的互联网应用开发通常倾向于使用如 Java、C#、PHP、Python、JavaScript、Ruby 等语言，因为这些语言提供了更全面的生态系统和工具链，以及更广泛的社区支持。

有人可能要争论，这些语言都可以编译成 WASM 字节码，在 WASM 虚拟机里运行。但是事实上，虽然 WASM 在 Web 前端开发领域的表现很强势，但目前互联网应用采用 WASM 作为后端的运行环境并不是一个主流选择。注意，智能合约（链上程序）是 Web3 时代的“新后端”。

## 总结

综上，我们已经讨论了采用微服务架构（或者说 Actor 模型）的优势，以及它为应用开发带来的复杂性。有些复杂性是不可避免的。比如，即使在工程化更成熟的 Web2 环境中，基于消息通信来实现“最终一致性”对于许多开发者而言已经是不小的挑战。在新生的 AO 平台上开发 Dapp，这个挑战似乎还要更加明显——当然这是完全可以理解的。以下链接文章的开篇就展示了一个例子。

https://github.com/dddappp/A-AO-Demo?tab=readme-ov-file#an-ao-dapp-development-demo-with-a-low-code-approach

我们都知道，公链之争，其实是争夺应用开发者的战争。那么，在这种情况下 AO 要如何赢得开发者？

我认为需要继续向已经获得“大规模采用”的 Web2 学习。这不仅包括学习其基础设施，还包括开发方法论、开发工具和软件工程实践等各个方面。在下一篇文章中，我会为大家展示我坚信的一种解决方案：低代码开发。