Layer2 架构设计，关键模块实现

> 本文尝试从演化角度讨论 Rollup Layer2 的发展以及演进，主要解答以下几个问题：
>
> 1. Rollup 是如何工作的
> 2. Rollup 的模块化演进
> 3. 模块化带来的可能性
> 4. 模块化应用的技术趋势
> 5. 总结

Rollup 是如何工作的

区块链的“三难问题”一直是困扰业界的一个难题，如果我们认为 Layer1 区块链应该首先保证“去中心化”和“安全”，那将“扩展性”方案从 Layer1 迁移出来就是自然的选择了，于是有了 Layer2。那新的难题就是如何通过 Layer1 来保证 Layer2 的安全。

最初有一种想法是定时将 Layer2 应用的状态树根写到 Layer1，这样可以通过状态证明来校验应用的状态，类似于交易所储备金证明。但这种方式第三方无法通过公开的方式验证两次状态转换是正确的。

为了更深入的探讨这个问题，我们抽象一下，任何程序的状态都可以通过一个状态转换公式表达：

```
1 σt+1 ≡ Y(σt, T)
```

这个公式来自于 Ethereum 黄皮书，但它可以代表任意的程序。在这里 Y 代表程序，σ 代表状态。状态 σt+1 由程序 Y 通过状态 σt 和交易 T 计算得出。交易 T 代表程序的输入。**任意时候，如果 σt 是确定的，程序 Y 是确定的，T 是确定的，那 σt+1 就是确定的。**

所以要提供公开的可验证性，关键是 Y 要公开可用，历史上所有的 T 要公开可用并且顺序确定，中间的状态可通过 Y 和 T 重新计算得到。而程序的公开可用我们可以通过开源来实现，关键是 T 公开可用如何保证，这就引入了数据可用性（DA）的概念。

数据可用性需要有个公开的不可篡改的账本来记录应用的交易。自然想到，区块链账本就是这样一个系统，于是将 Layer2 的交易写回 Layer1，保证数据可用性，这也就是 Rollup 名称的来源。

所以 Layer2 系统中需要有个角色收集用户的交易，进行排序并写入到 DA，这个角色叫 **定序器（Sequencer）**。这里的交易序列叫 **Canonical Transaction Chain**。

保证了数据的可用性，每个人都可以通过自己运行程序执行交易来得到最终的状态。但这里并没有达成共识，因为每个人不确定自己得到的结果是否和其他人的结果一致，毕竟软件或者硬件故障也可能

导致数据不一致。所以需要另外一个角色将交易执行后的状态树根定时公布出来，供大家校验自己的状态，这个角色叫 **提案者（Proposer）**。这里每次提交的状态也构成了一个状态序列，和交易序列对应，叫 **State Commitment Chain**。

到这里，我们达到了应用的可验证性。如果某个人运行的结果和 Proposer 提交的状态不一致，并确定不是自己的问题，那就是 Proposer 作弊或者出错了，那怎么让别人也知道呢？这就需要引入**仲裁者（Arbitrator）**的角色。仲裁者需要是一个可信第三方，链上合约正好可以承担这个角色。

仲裁有两个方案：

1. Proposer 每次提交状态的时候，同时提供与前一次状态之间的状态转换有效证明（Validity Proof），链上的仲裁合约进行校验。有效证明一般通过 Zero knowledge 技术生成，这种叫 ZK Rollup。
2. 先假定 Proposer 的结果是对的，但如果发现不一致，则提交欺诈证明（Fraud Proof），由仲裁合约进行判定。如果仲裁合约判定 Proposer 作弊，则对 Proposer 进行惩罚，并将 State Commitment Chain 回滚到欺诈交易之前的状态。当然，为了保证安全，一般会设置一个比较长的挑战周期来达到链上交易结算的最终确定性。这种叫 Optimistic Rollup。

我们还需要实现 Layer1 和 Layer2 之间的资产互通。于是构建一个 Layer1 到 Layer2 之间的桥，通过状态证明来进行资产结算。而 Layer2 在 Layer1 的状态根由 Layer1 的仲裁合约保证，我们可以认为这个桥的安全也受仲裁合约保证。

至此，我们得到了一个由 Layer1 保证安全，并且可以和 Layer1 进行资产互通的 Rollup Layer2 方案。

当然，Rollup 方案也做了一些妥协：

1. 将交易写入 Layer1，也就代表 Layer2 的扩展性依然受 Layer1 区块大小限制。以 Ethereum 为例，某个 Layer2 完全占据 Ethereum 的所有区块，能提供的平均 TPS 也才数百，扩展性受 DA 限制。
2. 为了节省 Gas 费，Sequencer 会将交易批量写入 DA，而在写入 DA 之前，Sequencer 有可能通过调整交易的顺序来作弊。

这里总结一下 Layer2 的安全以及交易的最终确定性：

1. 如果用户自己运行了一个 Layer2 的节点，并且忠实地按照 DA 的交易顺序执行，用户可以认为交易是即时确认并且达到最终确定的，因为如果用户执行的结果和 Proposer 不一样，说明 Proposer 作弊，需要回滚链上的状态，最终会和用户自己的节点执行的结果一样。 这里主要的风险点在于前面提到的，如果实时从 Sequencer 同步数据， Sequencer 调整尚未写入 DA 的交易的顺序带来的风险。
2. 如果用户自己无法运行节点，需要依赖一个 RPC 提供方，用户需要承担一定的信任风险。但这个风险和用户信任 Layer1 的 RPC 节点带来的风险类似。这里额外的风险依然是 Sequencer 丢弃交易或者重排交易带来的风险。
3. 如果 Proposer 出错，但没有节点发起挑战，超过了挑战期，这时候错误的状态无法回滚，只能通过社会共识硬分叉方式来修复状态。

Rollup 的模块化演进

根据前面的分析，Rollup 解决方案中，链上的多个合约承担不同的职能，代表不同的模块。那自然想到，能否将模块拆分到多个链，从而获得更高的扩展性。这就是模块化区块链以及模块化 Rollup 的思路。

模块化在这里有两层含义：

1. 通过模块化设计，让系统变为一个可拔插的系统。开发者可以通过模块的组装，满足不同的应用场景需求。
2. 基于 1 提供的能力，模块层的实现并不绑定在同一个 Layer1 上，从而得到更好的扩展性。

我们可以认为有三个主要的模块层：

- **数据可用层（Data Availability）**： 保证执行层的交易数据可以通过公开的方式获取，以及保证交易的序列。
- **结算层（Settlement）**：实现 Layer1 和 Layer2 之间的资产和状态结算。它包含 State Commitment Chain 和 Bridge。
- **仲裁层（Arbitration）**：校验欺诈证明，并做出裁决（Optimistic）或者校验有效证明（ZK）。仲裁层要有能力操控 State Commitment Chain。

DA 模块化

将 DA 职能迁移出来，用一个独立的解决方案，获得的首要好处是 Layer2 的交易 Gas 费至少降低一个数量级。

从安全方面来看，即便是 DA 链的去中心化弱于 Ethereum，但 DA 层对安全的保证主要是挑战期内的交易，过了挑战期后，DA 主要是为了方便其他节点同步数据，对安全并没有保障作用，所以去中心化的要求可以降低一个层次。

DA 专用链可以提供更高的存储带宽和更低的存储成本，并且针对多个应用共享 DA 进行专门的设计。这也是当前如 Celestia、Polygon Avail 这样的 DA 链的立足点。

将 DA 层拆分出去后，我们得到了下图的架构：

上图中由 DA 来承担保存 Canonical Transaction Chain 的职责，而给 Layer1 留了一个 L1ToL2 Transaction Queue 来实现 Layer1 和 Layer2 之间的消息通信，用户也可以直接写交易给这个 Queue，确保 Layer2 的 Permissionless，Sequencer 无法审核用户或者交易。

但这里会引入一个新的难题，如果 Sequencer 写入 DA 的交易序列和 Proposer 执行的交易序列不一致，仲裁合约如何判定？一种方案是 DA 链和仲裁链之间有一个跨链桥，实现在仲裁合约中校验 DA 链提供的数据证明。但这种方案依赖 DA 和其他链之间的跨链桥的实现，DA 的方案选型会受限制。另外一种方案是引入排序证明。

排序证明（Sequence Proof）

我们可以认为 Sequencer 实际上也属于 DA 方案的一部分，它相当于一个 App-Specific 的 DA，主要基于以下理由：

1. Sequencer 需要提供批量写入 DA 链之前这段时间内的 DA 保证。
2. Sequencer 需要负责交易的验证，排序，以及最终写入 DA。

如果要求 Sequencer 给每个交易生成一个 Sequence Proof，则可以解决两个问题：

1. 对尚未写入 DA 链的交易提供了保证，使 Sequencer 不敢随意调整交易的顺序或者丢弃交易。
2. 如果 DA 链和仲裁链之间没有跨链桥，则可以通过 Sequence Proof 的挑战机制来保证数据可用。

Sequence Proof 具有以下特性：

1. 它携带 Sequencer 的签名，证明它是某个 Sequencer 发出的。
2. 它可以证明某个交易在全部交易序列中的位置。
3. 它是累加器（Accumulator）证明的一种。每个交易累加后会生成新的累加结果，与该交易之前所有历史交易相关，使得其难以篡改。累加器的可选方案之一是默克尔累加器（Merkle Accumulator），累加结果表现为默克尔树的根。

Sequence Proof 的工作原理：

用户或者执行节点提交交易给 Sequencer，Sequencer 将 Sequence Proof 返回给用户，同时同步给其他节点。如果 Sequencer 在提交给 DA 之前丢弃或者篡改了交易的顺序，用户或者其他节点可以提交 Sequence Proof 给仲裁合约，从而惩罚 Sequencer。仲裁合约需要从 State Commitment Chain 合约中读取交易累加器的根，从而校验 Sequence Proof。

分场景探讨一下：

1. Sequencer 丢弃或者重排了用户交易。这会导致 Sequencer 在同一个位置生成了两个 Sequence Proof。用户提交 Sequence Proof 给仲裁合约，Sequencer 需要提供该交易被包含在最新的交易累加器的根中的证明，如果不能给出，则惩罚 Sequencer。
2. Sequencer 没有正确地将交易写入 DA 链，和 Proposer 合谋隐藏交易。如果仲裁链和 DA 链有桥，则通过桥来验证，惩罚 Sequencer。否则用户可以发起挑战，要求 Sequencer 给出某个位置的交易的证明以及原始信息。但这种情况仲裁合约无法判断用户是否是恶意挑战，所以如果 Sequencer 给出数据则不惩罚 Sequencer。而对用户来说，恶意挑战损人损己，也缺少经济动力。

我们通过引入 Sequence Proof 让 Layer2 的协议变得更安全。

交易流水线（Transaction Pipeline）以及并行执行（Parallel Execution）

将 Sequencer 划分给 DA，只负责交易的验证和排序，带来的另外一个好处是容易实现交易的流水线以及并行执行。

验证交易时，需要验证签名和是否有足够的 Gas 费，而 Gas 费的校验需要依赖状态。如果我们为了保证验证交易不会被执行交易阻塞，允许 Sequencer 验证交易依赖的状态和最新状态之间有一定的延迟（秒级），会导致 Gas 校验会不太准确，有被 DDoS 攻击的风险。

但我们认为 Sequencer 属于 DA 是一个正确的方向，所以值得我们进一步研究。比如可以将交易费中 DA 部分拆分出来，通过 UTXO（Sui Move Object） 表达，则可以降低 Gas 费检测成本。

Sequencer 给交易排序然后输出成交易流水线，然后同步给 Proposer 以及其他节点。每个节点可以根据自己的服务器情况来选择并行方案。每个节点需要保证：只并行没有因果关系的交易，有因果关系的交易必须按 Sequencer 的顺序执行，那最终的结果就是一致的。

Proposer 需要定时提交状态树的根，以及累加器的根到链上的 State Commitment Chain 合约中。

于是我们得到了一个低 Gas 费，高 TPS，并且更安全的模块化 Layer2： **Rooch**。

- MoveOS：它包含 MoveVM 以及 StateDB，是系统的执行以及状态存储引擎。StateDB 由两层稀疏默克尔树构建，可以提供状态证明。根据前面的分析可得出，状态树以及状态证明是 Rollup 应用不可或缺的组件。
- RPC：对外提供查询，提交交易，以及订阅服务。可以通过代理方式兼容其他链的 RPC 接口。
- Sequencer：验证交易，给交易排序，提供 Sequence Proof，将交易流式输出到 Transaction Pipeline。
- Proposer：从 Transaction Pipeline 获取交易，批量执行，定期提交到链上的 State Commitment Chain。
- Challenger：从 Transaction Pipeline 获取交易，批量执行，和 State Commitment Chain 比较，决定是否发起挑战。
- DA & Settlement & Arbitration Interface：对不同的模块层的抽象和封装，保证在不同的实现之间切换时不影响上层业务逻辑。

支持多链的交互式欺诈证明

Optimistic Rollup 方案中，链上的仲裁合约如何判定链下的交易执行出错，一直是一个难题。最初的想法是 Layer1 上重新执行一遍 Layer2 的交易，但这种方案的难题是 Layer1 的合约要模拟 Layer2 的交易执行，成本很高，也会限制 Layer2 交易的复杂度。

最后业界摸索出一种交互式证明的方案。因为任何复杂的交易，最终会转换成机器指令执行，如果我们找到产生分歧的指令，则只需要在链上模拟执行指令即可。

还用上面那个状态转换公式：

```
1 σt+1 ≡ Y(σt, T)
```

这里 Y 代表指令，T 代表指令输入，σ 代表指令所依赖的内存状态。如果在执行过程中，给每个 σ 都生成一个状态证明。 控辩双方可以通过交互，发现双方的分歧点 m，将 m-1 的状态 σ 以及指令 m 提交到链上仲裁合约模拟执行，仲裁合约执行后就可以给出判定。

所以剩下的问题就是通过什么方式来生成证明，主要有两个方案：

1. 直接在合约语言虚拟机中实现，比如 Arbitrum 的 AVM，Fuel 的 FuelVM。
2. 基于已有的指令集实现一个模拟器，在模拟器中提供证明能力。如 Optimism 的基于 MIPS 指令的 cannon，Arbitrum 新的基于 WASM 指令的 Nitro，以及 Rooch 的基于 MIPS 指令的 flexEmu。

flexEmu 是一个拥有单步状态证明能力的通用字节码模拟器，为多链执行环境设计。有了 flexEmu 的支持，可以实现仲裁层的模块化。任意支持图灵完备合约的链，都可以在合约中模拟 flexEmu 的指令，作为仲裁层。

ZK + Optimistic 组合方案

业界一直在争论 Optimistic Rollup 和 ZK Rollup 孰优孰劣，但我们认为将二者结合起来可以兼得两种方案的优点。

在前面的 Optimistic 方案基础上，我们再引入一个新的角色，ZK Prover。它批量给 Proposer 提交的交易状态生成有效证明，并提交给仲裁合约。仲裁合约验证后，就可以判定该交易在 Layer1 上达到了最终确定性，可以进行 Layer2 到 Layer1 的提款交易的结算。

这种方案的优点：

1. 不会因为 ZK 的性能问题限制 Layer2 的整体吞吐。
2. 可以通过 ZK 缩短 Optimistic 的挑战周期，提升用户体验。

在 ZK 的方案以及硬件加速成熟之前，我们可以先通过 Optimistic 构建生态，同时模块化方案可以让 ZK 最后无缝接入进来。

多链结算

如果我们进一步思考模块化的趋势，自然想到，既然 DA 可以迁移到别的链，那结算层是否也可以部署到别的链？

Layer1 和 Layer2 之间的资产结算主要依赖两个组件，一个是 Bridge，一个是 State Commitment Chain，从 Bridge 结算的时候，需要依赖 State Commitment Chain 校验 Layer2 的状态证明。Bridge 当然可以部署到多个链，但 State Commitment Chain 只能有一个权威的版本，由仲裁合约保证安全。

这个方向还需要深入研究，但有个初步的方案。其他链上的 State Commitment Chain 都是仲裁链（Ethereum）上的镜像。这个镜像并不需要同步全部的 Layer2 State Root 到其他链，而是用户按需通过 Ethereum 的状态证明做映射。

当然，其他链上还需要能校验 Ethereum 上的状态证明，所以需要知道 Ethereum 上的状态根。当前，将 Ethereum 上的状态根同步到其他节点有两个方案：1. 依赖 Oracle。2. 嵌入 Ethereum 轻节点，校验 Ethereum 的区块头。

这样我们就可以得到一个支持多链结算，但安全由 Ethereum 保证的 Layer2 方案。

这种方案和跨链的区别：

1. 如果是依赖中继链的跨链方案，可以认为 Layer2 替代了中继链，是一个安全受仲裁合约保证的中继层。

2. 如果是链间互相校验状态证明的跨链方案，多链结算方案和它共享状态根同步的技术方案，但简化了许多。因为在多链结算方案中，状态根的同步需求是单向的，只需要从仲裁链同步到其他链，不是两两互相同步。

模块化带来的可能性

通过模块化，开发者可以通过 Rooch 组合出不同的应用。

1. **Rooch Ethereum Layer2** = Rooch + Ethereum(Settlement+Arbitration) + DA
   这是 Rooch 首先要运行的网络。提供一个由 Ethereum 安全保证的，可以和 Ethereum 上的资产互通的 Move 运行平台。未来可以扩展到多链结算。
2. **Rooch Layer3 Rollup DApp** = Rooch + DApp Move Contract + Rooch Ethereum Layer2(Settlement + Arbitration) + DA 如果应用把自己的结算和仲裁部署到 Rooch Layer2，它就是一个 Rooch 的 Layer3 应用。
3. **XChain Rollup DApp** = Rooch + DApp Move Contract + XChain(Settlement + Arbitration) + DA
   任意链都可以通过 Rooch 来给开发者提供一套基于 Move 语言的 Rollup DApp 工具包。开发者只需要通过 Move 语言编写自己的应用逻辑，就可以运行一个安全受 XChain 保障的，资产可以和 XChain 互通的，独立环境的 Rollup 应用。当然这个需要和各公链的开发者来协同开发。
4. **Sovereign Rollup DApp** = Rooch + DApp Move Contract + DA 应用也可以将 Rooch 作为 Sovereign Rollup SDK，不部署 Bridge 以及 Arbitration 合约，State Commitment Chain 也保存在 DA，保证可验证性，由社会共识保证安全。
5. **Arweave SCP DApp** = Rooch + DApp Move Contract + DA（Arweave） SCP 和 Sovereign Rollup 思路类似，SCP 要求应用程序的代码也要保存到 DA。而 Rooch 中合约部署和升级都是交易，合约代码在交易中，都会写到 DA 层，所以我们认为符合 SCP 的标准。
6. **Move DApp Chain** = Cosmos SDK + MoveOS + DApp Move Contract MoveOS 可以作为一个独立的 Move 运行环境嵌入到任意的链的运行环境中，去构建应用链或者新的公链。
7. 非区块链项目 非区块链项目，可以把 MoveOS 作为一个可以带有数据校验能力以及存储证明能力的数据库使用。比如用它做一个本地的博客系统，数据结构和业务逻辑通过 Move 表达。等未来基础设施成熟，则可以直接和区块链生态对接起来。再比如可以用它做云计算中的 FaaS 服务，开发者通过 Move 编写 FaaS 中的 Function，平台托管状态，用户间的 Function 还可以互相组合调用。更多的可能性需要大家探索。

Rooch 的模块化方案可以适应于不同类型以及阶段的应用。比如开发者可以先通过部署合约在 Rooch Ethereum Layer2 上验证自己的想法，等成长起来后，将应用迁移到独立的基于 Rooch 搭建的 App-Specific Rollup 中。

再或者开发者直接通过 Sovereign Rollup 方式启动应用，因为应用早期对安全性要求不高，也没有和其他链互通资产的需求，先做到可验证。等应用成长起来，有了互通资产的需求，对安全性要求变高，这时候可以启用结算以及仲裁模块从而保证资产的安全。

模块化应用的技术趋势

DA 层潜力尚待挖掘

由前面的分析可以看出，无论哪种组合方式，都依赖 DA。DA 在去中心化应用中扮演的角色类似于 Web2 系统的日志平台，可以用来做审计，支持大数据分析，进行 AI 训练等。未来会有很多应用和服务围绕 DA 建立起来。当前已有 Celestia，Polygoin avail，未来还会有 EigenLayer，Ethereum danksharding 等。

根据前面的分析，我们得出 Sequencer 的角色应该属于 DA 的一部分，如果 DA 层能为应用提供交易校验能力，并且有足够的性能，实际上完全可以由 DA 来承担 Sequencer 的职责，用户直接写交易到 DA。当然能否使用应用的 Token 付 DA 的 Gas 费是另外一个需要解决的问题。

DApp 编程语言会爆发

新的应用形态会促使新的编程语言爆发，这在 Web2 时代已经验证。而 Move 会成为构建 Web3 DApp 的最佳语言。除了 Move 本身的语言特性外，还基于以下理由：

1. DApp 用同一种语言可以快速积累应用所需要的基础库，形成生态聚集效应。所以一开始支持多语言不是个好策略。
2. 去中心化应用至少要保证可验证性，而智能合约语言可以让开发者在保证可验证性方面减少许多心智负担。
3. Move 的平台无关性，可以让它很容易适配到不同的平台，不同的应用中。
4. Move 的状态是结构化的，有利于 DApp 的数据结构表达以及存储检索。

总结

我在 17 年底进入区块链领域，当时业界有非常多的团队尝试在区块链领域构建应用。可惜当时基础设施尚不完备，业界尚未摸索出一个可复制的构建应用的模式，大多数应用类项目以失败告终，打击了开发者和投资者。区块链上的应用应该如何构建出来？这个问题一直让我思考了五年。

而现在，随着 Layer1，Layer2 以及智能合约，模块化基础设施的成熟，这个问题的答案也逐渐清晰起来。

希望在即将到来的 Web3 DApp 爆发潮中， Rooch 可以助开发者一臂之力，让应用更快的构建，真正的落地。