polygon zkEVM

技术文件

无需信任的 L2 状态管理

v.1.1

2023 年 2 月 13 日

内容

1 简介

区块链三难困境，也称为可扩展性三难困境，是指区块链系统中去中心化、安全性和可扩展性之间存在的权衡。这个概念最早是由以太坊的联合创始人 Vitalik Buterin 提出的。由于网络吞吐量有限以及三难困境所描述的局限性，以太坊面临着可扩展性问题。为了解决这些问题，已经为以太坊开发了各种第 2 层 (L2) 可扩展性解决方案。这些解决方案旨在通过在不牺牲去中心化或安全性的情况下提高交易吞吐量和降低费用来提高以太坊网络的可扩展性。Polygon zkEVM 是一种 L2 rollup 解决方案，它结合了 Layer 1 (L1) 以太坊链中的数据可用性和执行验证，以确保 L2 状态转换的安全性和可靠性。本文档描述了 Polygon 为此目的设计和实现的基础设施。

2 协议组件

本文档旨在描述用于确保 Polygon zkEVM L2 rollup 中交易的最终性和状态转换的正确性的协议。首先我们将介绍协议的主要组件：

- **可信测序仪：**角色负责接收用户的 L2 交易，对其进行排序，生成批次，并以序列的形式提交到 L1 合约的存储槽中。这些批次将在之前执行并广播到 L2 网络节点，以实现快速终结并降低与高网络使用率相关的成本。必须在 Sequencer 模式下运行 zkEVM 节点并控制在 L1 合约中强制执行的特定以太坊帐户。

- **可信聚合器：**接受由 Sequencer 提交的 L2 批次并使用特殊的链下 EVM 解释器的角色，该解释器除了能够计算由交易批次执行产生的 L2 状态外，还生成计算完整性 (CI) 的零知识证明。该证明由 L1 合约逻辑（L1 安全继承）验证，其验证成功需要在 L1 合约中提交新的 L2 状态根（L2 状态的简洁密码摘要），这将是一个无可辩驳的证明批次序列导致特定的 L2 状态。必须在聚合器模式下运行 zkEVM 节点并控制在 L1 合约中强制执行的特定以太坊帐户。

- **PolygonZkEVM.sol:**Sequencer 使用 L1 合约来提交交易批次的序列，充当序列的历史存储库。此外，允许聚合器通过验证交易批次执行的证明来公开验证 L2 状态根转换。也充当 L2 状态根历史存储库。

如图 1 所示，批次由 Trusted Sequencer 生成，但为了实现 L2 交易的快速终结并避免等待下一个 L1 块的需要，它们通过广播通道与 L2 网络节点共享。每个节点将执行批处理以在本地计算生成的 L2 状态。然后，一旦 Trusted Sequencer 提交，L2 网络节点将再次执行直接从 L1 获取的批处理序列，因此它们不再需要信任它。最终，批次的链下执行将通过零知识证明的验证在链上进行验证，并将提交生成的 L2 状态根。作为协议的最后一步，新的 L2 状态根也将由 L2 网络节点直接从 L1 同步。

交易执行仅依赖于 L1 安全假设，并且在协议的最后阶段，节点将仅依赖于 L1 中存在的数据来与每个 L2 状态转换保持同步。

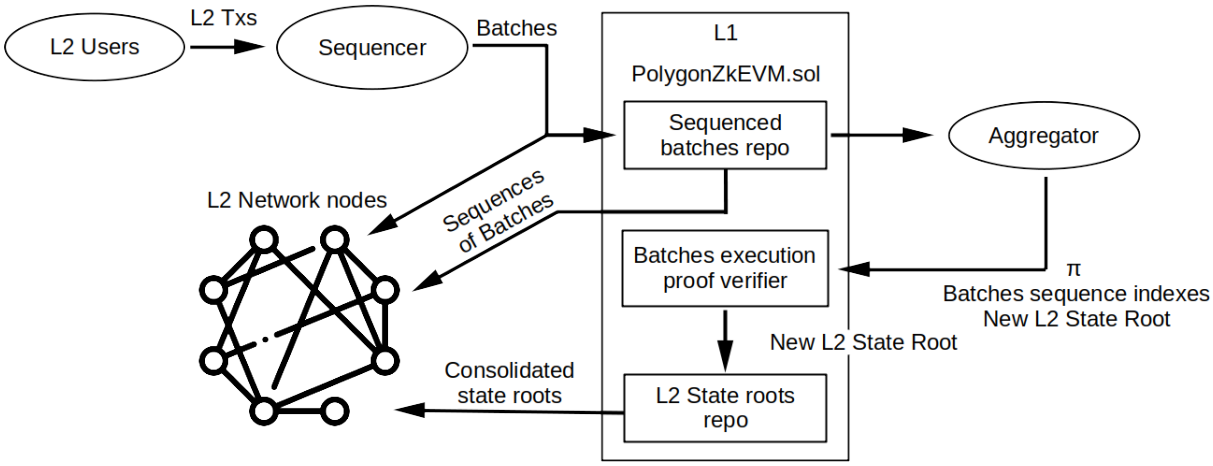

图1：Polygon zkEVM 无需信任的 L2 状态管理概述。

3 个 L2 状态阶段

由于 L2 网络节点更新/同步它们的本地状态总共 3 次，我们必须定义 L2 状态的 3 个可能阶段：

- **可信状态：**当 L2 网络节点在提交到 L1 之前接收并执行由可信排序器广播的交易批次时达到的状态。此阶段称为可信阶段，因为它仅基于对可信定序器的信任才被视为有效。

- **虚拟状态：**当 L2 网络节点处理已经由 Trusted Sequencer 提交给 L1 的批处理序列时，就会达到此状态。请注意，此状态是无需信任的，因为它依赖于 L1 安全假设。所有 L2 网络节点都可以通过执行来自 L1 的批处理序列来计算其本地 L2 状态，而无需再依赖可信排序器。然而，由于稍后将解释的不同因素，节点可能在可信状态和虚拟状态之间存在分歧。L2 网络节点准备好处理这些情况，并且只会将执行 L1 中提交的批处理序列所产生的状态视为有效。

- **合并状态：**当零知识证明在 L1 中成功验证后，L2 网络节点将其本地 L2 状态根与可信聚合器在 L1 中提交的状态根同步时达到的状态。

图 2 从批处理的角度显示了 L2 阶段时间线，以及触发其包含到下一个 L2 状态阶段的操作。

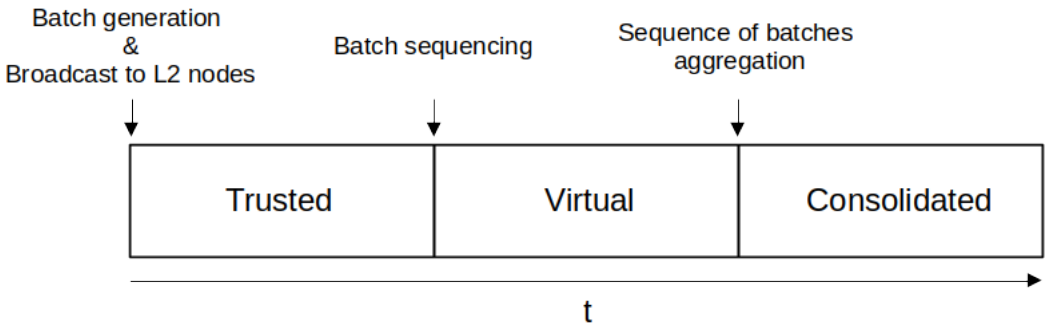

图 2：L2 状态阶段时间表。

4 种 zkEVM 节点执行模式

zkEVM 节点是包含运行 zkEVM 网络所需的所有组件的软件包。该节点可以以三种不同的模式运行：

- **音序器：**持有 L2 状态的实例，管理批量广播到其他 L2 网络节点，并有一个 API（应用程序编程接口）来处理 L2 用户的交互（交易请求和 L2 状态查询）。还有一个数据库来临时存储尚未订购和执行的交易（待定交易池），以及与 L1 交互所需的所有组件，以便对交易批次进行排序，并保持其本地 L2 状态的最新状态.

- **聚合器：**拥有执行事务批处理、计算结果 L2 状态和生成零知识 CI 证明所需的所有软件组件。此外，具有获取由可信排序器在 L1 中提交的事务批次所需的所有组件，并调用函数来公开验证 L1 上的 L2 状态转换。

- **远程过程调用：**持有更新的 L2 状态实例，首先是可信排序器广播的批次，然后是从 L1 合约中获取的批次序列。还具有与 L1 交互所需的所有组件，以保持其本地 L2 状态最新并检查 L2 状态根同步。

5 L2 交易生命流。

5.1 交易提交给Trusted Sequencer节点。

与发送到 L1 的交易一样，交易由用户的钱包创建并使用他们的私钥签名，事实上，作为 Polygon zkEVM 的 L2 EVM 的一个特性是它具有与以太坊 L1 完全相同的用户体验。

用户和 zkEVM 之间的通信是通过与以太坊 RPC 完全兼容的 JSON RPC 完成的。这种方法使得任何 EVM 兼容的应用程序，例如钱包软件，也与 zkEVM 原生兼容。

一旦交易生成并签名，它们就会通过他的 JSON RPC 接口发送到 Trusted Sequencer 的节点。交易将存储在待定交易池中，等待被排序器选择/丢弃以执行。

5.2 交易执行和可信状态。

最终，Trusted Sequencer 将从池中取出交易，将它们排序并添加到交易批次中，并通过执行这些批次来更新其本地 L2 状态。一旦交易批次被添加到可信排序器的 L2 状态实例中，它们就可以立即由广播服务共享给其他 zkEVM 节点，以允许它们访问可信状态。请注意，通过依赖可信排序器，我们可以归档快速交易终结性（比 L1 更快），但是生成的 L2 状态将处于可信状态，直到在 L1 合约中对批次进行排序。

用户通常会与受信任的 L2 State 进行交互，但是，由于某些协议特性（将在以下部分详细介绍），L2 交易的验证过程可能需要相对较长的时间，通常在 30 分钟左右，但在极少数情况下会长达 2 周. 因此，用户应该意识到与高价值交易相关的潜在风险，尤其是那些无法逆转的交易，例如在 L2 之外产生影响的交易，例如场外交易、场外交易、和替代桥梁。

5.3 交易批处理。

Trusted Sequencer 必须使用 L1 PolygonZkEVM.sol 合约中表达的特殊格式对交易进行批处理，如下所示**批量数据**结构：

```
1个   结构批处理数据{
2个           字节交易;
3个           字节 32     全球出口根;
4个           uint64    时间戳;
5个           uint64    minForcedTimestamp ;
6个     }
```

- **交易：**包含串联批处理事务的字节数组。每笔交易都使用 rlp（递归长度前缀）标准按照 Ethereum pre-EIP-115 或 EIP-115 格式进行编码，并与签名的值 v、r、s 连接。
  - EIP-155: rlp(nonce, gasprice, gasLimit, to, value, data, chainid, 0, 0,)vrs.
  - EIP-155 之前：rlp(nonce, gasprice, gasLimit, to, value, data)vrs。
- **全局退出根：**Bridge 的 Global Exit Merkle Tree 的根将在批处理开始时同步到 L2 State，允许 bridge claiming transactions 在 L2 中成功执行。Bridge 用于在 L1 和 L2 之间移动资产，并使用认领交易解锁目标网络中的资产。
- **时间戳：**Batch timestamp，Trusted Sequencer 可以为一个batch设置的最大batch timestamp是执行排序L1交易的块的时间戳。此外，批次的时间戳必须高于或等于最后一个排序批次的时间戳。这两个限制约束了批次要及时排序并与 L1 块同步。
- **minForcedTimestamp:**如果批次是强制批次，则此参数将不为零，强制批次用作审查对策（详见第五节）。

5.4 批次排序和虚拟状态。

对批次进行排序意味着成功地将一系列批次添加到 L1 PolygonZkEVM.sol 合约的映射**排序批次**，这是一个存储结构，包含定义虚拟状态的序列队列。

```
1个    // 序列批号-->序列化批数据
2个    映射(uint64=>序列批处理数据）民众排序批次;
```

图 3 显示了批次序列的逻辑结构，批次中可以包含的交易数量受交易字节数组长度限制为合约的值**MAX_TRANSACTIONS_BYTE_LENGTH**(300000) 公共常量。同样，序列中的批次数量受合同的限制 **MAX_VERIFY_BATCHES 个**(1000) 公共常量。

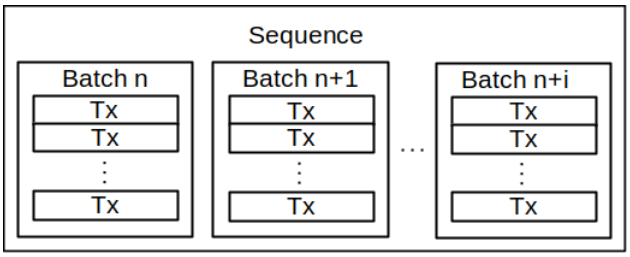

图 3：序列结构。

要对批次序列进行排序，可信排序器必须调用**序列批次**合同函数，作为参数给出一个数组，其中包含要排序的批次。

```
1个    功能序列批次（
2个        批量数据[]记忆批次
3个    ）  民众如果不是紧急状态                    只有可信的定序器
```

批次array 必须至少包含一个批次，最多包含 **MAX_VERIFY_BATCHES 个**持续的。**序列批次**只能由 Trusted Sequencer 的 Etherum 帐户调用。此外，合同不得处于紧急状态（详见第 IX 节）。如果不满足这些条件，函数调用将恢复。

这**序列批次**函数将遍历每批序列，检查其有效性。有效批次必须满足以下条件：

- 必须包括**全局退出根**中存在的值**GlobalExitRootMap** 桥的 L1 PolygonZkEVMGlobalExitRoot.sol。这意味着要排序的批次必须包含有效的 globalExitRoot。
- 交易字节数组长度必须小于值 **MAX_TRANSACTIONS_BYTE_LENGTH**持续的。
- 批次的时间戳必须大于或等于上一个排序的批次的时间戳，并且小于或等于执行排序的 L1 交易的块的时间戳。请注意，批次是按时间 戳排序的，并且它们永远不会超过执行排序事务的 L1 块的时间戳，因此 L1 块时间戳将作为 L2 批次的时钟源，以实现两个网络之间的时间同步。

如果批次无效，事务将恢复并丢弃整个序列。否则，如果批次有效，则排序过程将继续。

最后一批顺序是一个存储变量，它将为每个批次排序而递增，充当批次计数器并为每个批次提供一个特定的索引号，该索引号将用作批次链中的位置值。

为确保批链的加密完整性，将使用一种机制将批次链接到它们之前的批次。将为每个排序的批次计算累积的哈希值。之所以称为累积，是因为将批次绑定到已排序的前一批次的累积哈希值。

对于特定批次，累积哈希计算如下：

```
keccak256(
    阿比 编码打包（
        当前账户输入哈希，
        当前事务哈希，当前批次。
        globalExitRoot，currentBatch。时间戳,
        消息.发件人))
```

- **currentAccInputHash (bytes32):**前一批序列的累积哈希值。
- **当前交易哈希（bytes32）：**当前批量交易字节数组的哈希摘要，keccak256(currentBatch.transactions)。
- **currentBatch.globalExitRoot (bytes32):**当前批量执行产生的 Bridge 全局出口默克尔树根。
- **当前批处理时间戳（uint64）：**当前批处理时间戳。
- **msg.sender（地址）**在 L1 中执行排序事务的 EOA。

从图 4 中可以看出，每个累积的哈希将确保批处理数据（交易、时间戳、globalExitRoot）及其所有前身的完整性，以及它们的排序顺序。请注意，批处理链中的任何更改都是不可能的，因为单个位的修改将导致不同的最后累积哈希。

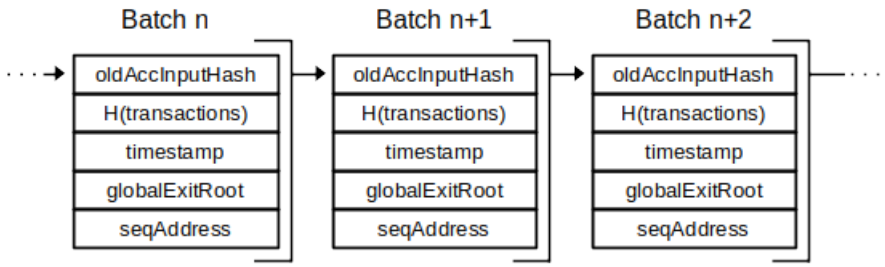

图 4：批处理链结构。

一旦验证了序列中所有批次的有效性并且计算了每个批次的累积哈希值，批次序列将被添加到**排序批次** 使用以下映射**序列化批数据**结构：

```
结构序列化批数据                    {
    字节 32     账户输入哈希;
    uint64    排序时间戳;
    uint64    previousLastBatchSequenced ;
}
```

- **accInputHash:**序列中最后一批的唯一批加密代表。

- **排序时间戳：**执行顺序 L1 事务的块的时间戳。

- **previosLastBatchSequenced：**当前序列的第一个批次之前的最后一个排序批次的索引，即前一个序列的最后一个批次。

序列中最后一批的索引号将用作键， **序列化批数据**当输入序列时，struct 将用作值**排序批次**映射。由于 L1 中的存储操作在气体消耗方面非常昂贵，因此尽可能减少其使用非常重要。为此，存储槽（映射条目）专门用于存储序列的承诺。映射中的每个条目将提交两个批次索引（前一个序列的最后一批作为值**序列化批数据**结构和当前序列的最后一批作为映射键）以及当前序列中最后一批的累积哈希和时间戳。

请注意，仅存储序列中最后一批的累积哈希，所有其他哈希仅在运行中计算以获得最后一个。事实上，正如上面所解释的那样，哈希摘要将是整个批处理链的承诺。此外，批次索引提供有用的信息，例如序列中的批次数量及其在批次链中的位置。时间戳将序列绑定到特定的时刻。

L2 交易的数据可用性是有保证的，因为每一个 batch 的数据都可以从排序交易的 calldata 中恢复出来，这些数据不是合约存储的一部分，而是 L1 State 的一部分。

完成排序交易执行的最后一个要求是排序者支付激励机制规定的排序费用（详见第五节）。

最后一个**序列批次**事件将被发出。

```
1个    事件序列批次（uint64索引批次数）
```

一旦批次在 L1 中成功排序，所有 zkEVM 节点都可以通过直接从 L1 PolygonZkEVM.sol 合约获取批次序列来同步其本地 L2 状态状态，而无需再信任 Trusted Sequencer，从而达到 L2 虚拟状态.

5.5 批量聚合和合并状态。

为避免将来的误解，有必要区分“合并”、“验证”和“汇总”等术语。合并指的是状态转换或生成的 L2 状态根，而验证和聚合指的是批次序列。此外，特定状态转换的合并意味着，首先是成功的验证，然后是代表该特定状态转换的特定批处理序列的成功聚合。

为了实现 L2 状态最后阶段（合并），可信聚合器最终应该聚合之前由可信排序器提交的批次序列。

聚合序列意味着成功地将生成的 L2 状态根添加到 L1 PolygonZkEVM.sol 合约的 **batchNumToStateRoot**映射，这是一个存储结构，它保存所有合并的 L2 状态根，这些根由每个聚合批次序列的最后一个批次索引键控。

```
1个    //批号-->状态根
2个    映射(uint64=>字节 32)民众batchNumToStateRoot;
```

批次序列的验证意味着批次执行序列的零知识 CI 证明的成功验证。底层的零知识验证模式是一种简洁的非交互式知识论证 (SNARK)，其关键属性之一是证明的简洁性和快速验证。因此，给定详尽的计算，可以使用原始计算所需的一小部分计算资源来验证其完整性。因此，通过使用 SNARK 模式，我们可以以一种高效的方式为详尽的链下计算提供链上安全性。

从图 5 中可以看出，一系列批处理的链下执行将假定 L2 状态转换，并因此更改为新的 L2 状态根。该执行的计算完整性 (CI) 证明将由聚合器生成，其链上验证 (L1) 将确保生成的 L2 状态根的有效性。

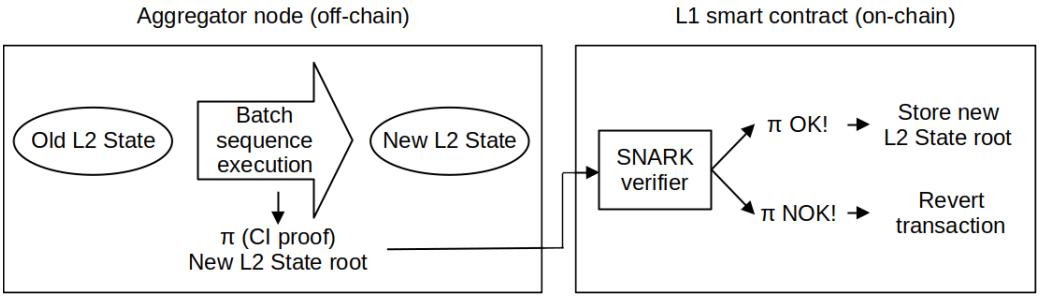

图 5：具有链上安全继承的链下 L2 状态转换。

为了聚合一系列批次，Trusted Aggregator 必须调用 **可信验证批次**合约功能：

```
1个    功能          trustedVerifyBatches (
2个        uint64    pendingStateNum ,
3个        uint64    初始化编号批次，
4个        uint64    最终新批次，
5个        字节 32     新本地出口根，
6个        字节 32     新国根，
7          uint256[2]       调用数据证明A，
8个        uint256[2][2]          调用数据证明 B ,
9          uint256[2]呼叫数据              证明C
10     )   民众仅受信任的聚合器
```

- **pendingStateNum:**待合并的状态转换数，只要可信聚合器正常运行，就为 0。Pending 状态是一种安全机制，当 L2 状态由一个独立的 Aggregator 合并时使用（详见第 VII 节）。

- **初始批次：**最后一个聚合序列中最后一批的索引。
- **最终新批次：**正在聚合的序列中最后一批的索引。
- **新本地出口根：**在序列执行结束时，Bridge 的 L2 Exit Merkle Tree 的根将用于在聚合序列时计算新的 Global Exit Root，从而使 bridge claiming transactions 在 L1 中成功执行。

- **新状态根：**在较旧的 L2 状态上执行批处理序列产生的 L2 StateRoot。

- **证明（A、B 和 C）：**批处理执行顺序的零知识 CI 证明。

可信验证批次只能由 Trusted Aggregator 帐户调用。最初 **可信验证批次**函数将调用**_验证和奖励批次**内部功能。 **_verifyAndRewardBatches**采用与相同的参数**可信验证批次**并实现逻辑来验证给定批次执行序列的零知识 CI 证明，如果验证成功，则向聚合器支付激励机制规定的奖励（详见第五节）。

要成功验证一系列批次，必须满足以下条件：

- **初始批次**参数必须是已经聚合的批次的索引，也就是说，必须有一个 L2 状态根 **batchNumToStateRoot**映射。
- **初始批次**参数必须小于或等于最后验证的批次索引。
- **最终新批次**参数必须高于最后验证的批次索引。
- **初始批次**和**最终新批次**参数必须是有序的批次，即在**排序批次**映射。

- 必须成功验证零知识CI 证明。

Executor 和 Prover 是 Aggregator 节点的服务，它们执行批处理并生成零知识证明。它们的架构超出了本文的范围，所以现在我们将它们视为以太坊虚拟机 (EVM)“黑匣子”解释器，在当前 L2 状态上执行一系列事务批处理，计算生成的 L2 状态根，并为执行生成零知识 CI 证明。

证明/验证系统的设计方式是，如果证明验证成功，则通过密码证明在特定 L2 状态上执行给定的批次序列会导致 L2 状态，由**新状态根**争论。

以下代码是验证零知识证明的 PolygonZkEVM.sol 合约的一部分：

```
1个   // 获取 snark 字节
2个   字节记忆snarkHashBytes = getInputSnarkBytes (
3个       初始化编号批次，
4个       最终新批次，
5个       新本地出口根，
6个       旧状态根，
7         新状态根
8个   );
9
10      //计算snark输入
11      uint256inputSnark =uint256(sha256(snarkHashBytes )) % _RFIELD ;
12
13      // 验证证明
14      要求(
15          汇总验证器。verifyProof (proofA, proofB, proofC, [inputSnark]), “ PolygonZkEVM ::
16          _verifyBatches： 无效证明”
17      );
```

汇总验证器是一个具有函数的外部合约**验证证明**需要一个证明（proofA，proofB，proofC）和一个值**输入Snark**并返回一个布尔值，如果证明有效则为真，否则为假。

证明的成功验证只是确认了计算的完整性，而不是使用了正确的输入以及它们产生了正确的输出值。公开论证用于公开披露被证明的计算的关键点，以证明它是使用正确的输入执行的并揭示输出。

这样，在证明验证过程中，L1 智能合约将设置公共参数，以确保被证明的状态转换对应于 Trusted Sequencer 提交的批次的执行。

输入Snark是特定 L2 状态转换的 256 位唯一密码表示，用作公共参数。计算为 sha256 mod % _RFIELD 字节字符串的散列，称为**snarkHashBytes**(由于 SNARK 中使用了数学原语，因此需要模运算符）。**snarkHashBytes**数组由称为智能合约的函数计算**getInputSnarkBytes**它是以下值的 ABI 编码打包字符串：

- **消息发件人：**可信聚合器的地址。
- **旧状态根：**L2 State Root，表示要证明的状态转换之前的 L2 State。
- **旧的 AccInputHash：**最后一批汇总的累积哈希值。
- **初始批次：**最后一批汇总的索引。
- **链号：**唯一链标识符。
- **新状态根：**L2 State Root 表示正在证明的状态转换后的 L2 状态。
- **新的 AccInputHash：**正在聚合的序列中最后一批的累积哈希值。
- **新本地出口根：**序列执行结束时 Bridge 的 L2 Exit Merkle Tree 的根。
- **最终新批次：**执行范围内最后一批的编号。

输入蛇鲨将代表特定 L2 状态转换的所有 L2 交易，以特定顺序执行，在特定 L2（链 id）中，并由特定的可信聚合器（msg.sender）证明。这**可信验证批次**函数不仅验证零知识证明的有效性，而且还检查**输入蛇鲨** 对应于待聚合的 L2 状态转换。

如果内部调用**_验证和奖励批次**返回 true 将意味着批次序列已成功验证，然后**新状态根**参数将被添加到**batchNumToStateRoot**映射。序列中最后一批的索引将用作条目的键。

最后一个**可信验证批次**事件将被发出。

```
1个   事件TrustedVerifyBatches (
2个       uint64    索引数量，
3个       字节 32     状态根，
4个       地址        索引聚合器
5个   );
```

一旦批次在 L1 中成功聚合，所有 zkEVM 节点都可以通过直接从 L1 PolygonZkEVM.sol 合约获取和检查合并根来检查其本地 L2 状态的有效性，从而达到 L2 合并状态。

6 激励机制

为了保持系统的可持续性，必须激励参与者正确地履行他们的角色并最终确定协议。

6.1 L2 交易手续费和排序手续费

Bridged Ether，起源于L1，是L2使用的原生货币，即用于支付L2交易手续费的货币。它可以以 1:1 的交换比率从 L1 桥接到 L2，反之亦然。由于 L2 帐户默认情况下没有以太币来支付交易费用，因此当从 L1 索取桥接资产时，调用桥接索取功能的 L2 索取交易由协议资助，不需要支付 gas 费用。

排序器将赚取用户在 L2 中支付的交易费用，因此将直接以桥接以太币支付。支付的费用金额将取决于天然气价格，天然气价格由用户根据他们愿意为执行交易支付的费用设定。

为了激励聚合器，对于每个排序的批次，排序器必须在 L1 PolygonZkEVM.sol 合约中锁定与序列中的批次数量成比例的 MATIC 代币数量。**批量费**存储变量包含每个批次排序必须锁定的 MATIC 代币数量。

图 6 显示了协议中每个参与者的收入和结果。

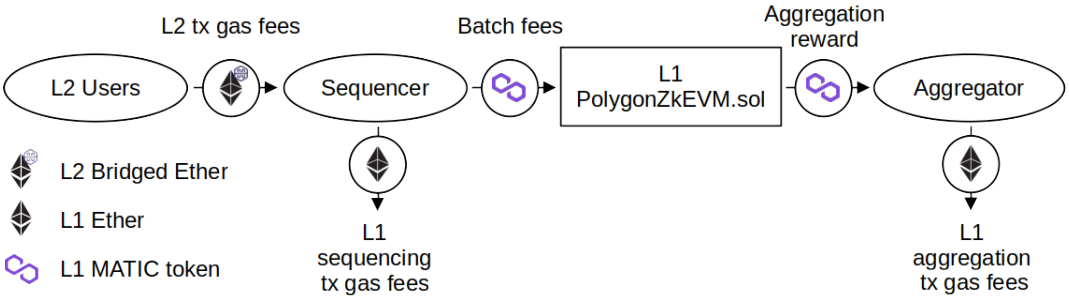

图 6：协议中每个参与者的收入和结果。

请注意，Sequencer 将优先处理 gas 价格较高的交易，以最大化其收入。此外，还有一个阈值，低于该阈值，Sequencer 执行交易将无利可图，因为从 L2 用户那里赚取的费用将少于作为排序费用支付的 MATIC 金额加上 L1 排序交易费用的等价以太币。为了激励 Sequencer，用户应该将他们的交易费用设置为高于此阈值的水平，否则 Sequencer 将不会被激励来处理他们的交易。以下表达式表示 Sequencer 对一系列批次进行排序所获得的以太币净值：

$$\text{“}Sequencernetetherincome\text{”} = \text{总}L2\text{个}TxGasFee - (\text{大号}1\text{个}SeqTxGasFees + \frac{\text{批量费} * n\text{批次}}{\text{比特币/以太币}})$$

在哪里：

- **总 L2TxGas 费用：**从批序列中包含的所有 L2 交易中收集的费用总和。
- **L1SeqTxGasFee：**L1 中支付的排序交易 gas 费。
- **批量费用：**PolygonZkEVM.sol**批量费**存储变量。

- **n批次：**序列中的批次数。
- **马蒂克/以太坊：**MATIC 代币的价格以以太币表示。

6.2 聚合奖励

每次聚合器聚合一个序列时，赚取的 MATIC 代币数量将取决于合约的总 MATIC 余额和聚合的批次数。每批聚合赚取的 MATIC 数量由 L1 PolygonZkEVM.sol 合约在序列聚合之前使用以下表达式计算：

$$\text{批量奖励} = \frac{\text{“合约 MATIC 余额”}}{\text{“尚未聚合的批次数量”}}$$

因此，以下表达式表示聚合器将通过聚合一系列批次获得的以太币价值总量。

$$\text{“聚合器净以太币收入”} = \frac{\text{批量奖励} * n\text{批次}}{\text{比特币/以太币}} - \text{大号1个} AggTxGasFee$$

在哪里：

- **L1AggTxGasFee：**在 L1 中支付的聚合交易 gas 费。
- **批量奖励：**每批汇总获得的 MATIC 数量。
- **n批次：**序列中的批次数。
- **马蒂克/以太坊：**MATIC 代币的价格以以太币表示。

6.3 batchFee变量重新调整

批量费每次由独立聚合器聚合序列时，都会自动调整。当 Trusted Aggregator 无法正常工作（详见第 VII 节）时，就会出现这种情况，并且**批量费**变量将被修改以激励聚集。

_updateBatchFee内部函数用于调整**批量费**存储变量。

```
1个    功能_更新批量费用（uint64newLastVerifiedBatch )内部的
```

管理员定义的两个存储变量（详见第 IX 节）用于调整费用调整功能：

- **veryBatchTimeTarget：**批次验证的时间目标，**批量费** 将更新变量以实现此目标。
- **乘数批量费用：**具有 3 位小数的批量费用乘数，范围为 1000 - 1024。

_updateBatchFee首先将评估有多少被聚合的批次迟到了，也就是说它们还没有被聚合，并且**veryBatchTimeTarget**时间过去了。 **差异批次**变量表示后期批次与低于目标的批次之间的差异，但其值受限于**MAX_BATCH_MULTIPLIER**常量 (12)。

如果在被聚合的序列中，迟到的批次多于低于目标的其他批次，则将应用以下公式**批量费**存储变量：

$$\text{”}新批次费\text{”} = \text{“}旧批费\text{”} \frac{乘数批处理费^{差异批次}}{10^{3个差异批次}}$$

图 7 显示**批量费**% 的变量变化取决于**差异批次**不同值的值**乘数批量费用**当后期批次主导序列时。请注意，目标是增加聚合奖励以激励聚合。

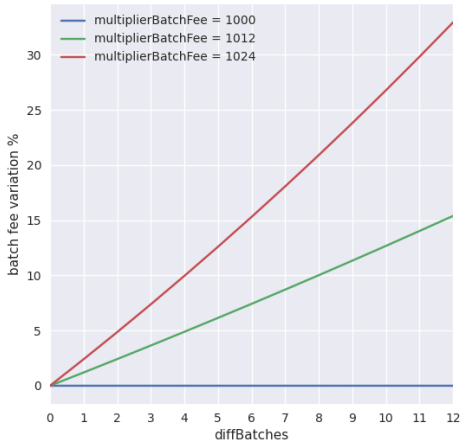

图 7：%当后期批次占主导地位时，批次费用变化的影响。

在另一种情况下，如果低于目标的批次多于迟到的批次，则将应用以下公式**批量费**存储变量：

$$\text{”}新批次费\text{”} = \text{“}旧批费\text{”} \frac{10^{3个差异批次}}{乘数批处理费^{差异批次}}$$

图 8 显示**批量费**% 的变量变化取决于**差异批次**不同值的值**乘数批量费用**当低于时间目标的批次支配序列时。请注意，目标是减少聚合奖励。

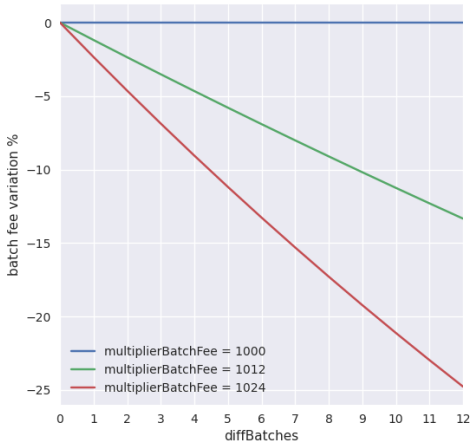

图 8：%当低于时间目标的批次在序列中占主导地位时，批次费用的变化。

总结一下调整**veryBatchTimeTarget**和**乘数批量费用** 管理员可以调整**批量费**变量重新调整并激励协议的参与者以平均目标为目标**veryBatchTimeTarget**.

合约初始化期间设置的值：

- **批量费**=10_{18}(1 马蒂奇）。
- **veryBatchTimeTarget**=30分钟。
- **乘数批量费用**=1002。

7 抵制 Trusted Sequencer 审查或故障

在上述方案中，用户需要依赖一个可信排序器来让他们的交易在 L2 中执行。如果用户无法通过可信排序器执行他们的交易，他们可以将它们包含在强制批处理中。强制批处理是用户可以提交给 L1 的一批 L2 交易，作为公开声明他们执行这些交易的意图的一种方式。

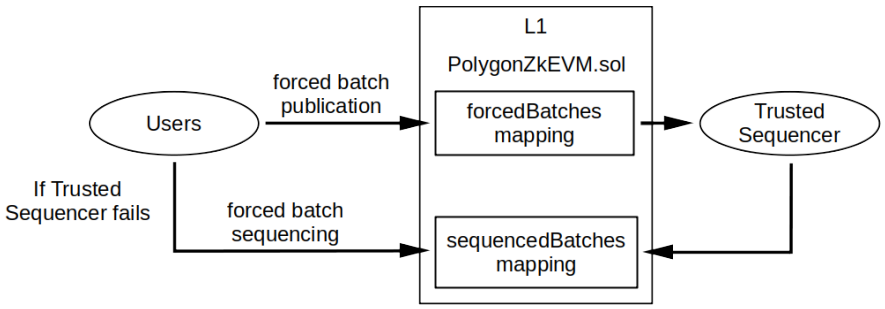

图 9：强制批量排序流程。

如图 9 所示，PolygonZkEVM.sol 合约有一个**强制批次**用户可以在其中提交要强制执行的事务批次的映射。**强制批次**映射充当不可变的公告板，其中强制批次将被标记时间戳并发布等待包含在序列中。为了保持其作为可信实体的地位，可信排序器将在未来的序列中包含这些强制批次。否则，用户将能够证明他们正在被审查，Trusted Sequencer 将失去其可信状态。

```
1个    // 强制批号-->hashedForcedBatchData 映射(uint64=>字节 32)民众强制
2个    批次;
```

尽管受信任的排序者被激励对发布在**强制批次**映射，这不能保证这些批次中交易执行的最终性。为了确保在 Trusted Sequencer 故障的情况下的最终性，L1 PolygonZkEVM.sol 合约有一个替代的批量排序功能，称为**sequenceForceBatches**. 此功能允许任何人对已发布的强制批次进行排序**强制批次**公共常量指定时间段的映射**FORCE_BATCH_TIMEOUT**(5 天）而且它们还没有被测序。

任何用户都可以通过直接调用来发布要强制执行的批处理**批处理**功能：

```
1个   功能强制批量（
2个   字节记忆          交易，
3个   uint256    数量
4个   ）  民众      如果不是紧急状态            isForceBatchAllowed
```

- **交易：**包含连接的批量交易的字节数组（与正常的批量交易格式相同）。

- **数量：**用户愿意作为强制批量发布费用支付的最大 MATIC 代币数量。强制批次发表费用与测序费用相同，设置在**批量费**存储变量。由于费用是在强制批发布时支付的，因此在批次测序时不会再次支付。

成功发布强制批量到**强制批次**映射必须满足以下条件，否则事务将被还原：

- 合同不得处于紧急状态。
- 必须允许强制批处理。
- 这**数量**参数必须高于每批次的 matic 费用。
- 交易字节数组长度必须小于值 **MAX_TRANSACTIONS_BYTE_LENGTH**常数（300000）。

强制批次将输入**强制批次**由其强制批次索引键控的映射。**lastForceBatch**是一个存储变量，它将为每个发布的强制批次递增，充当强制批次计数器并给出特定的索引号。输入的值是 ABI 编码打包的哈希摘要**强制批处理数据**结构字段。

```
1个   结构      强制批处理数据         {
2个      字节     交易;
3个      字节 32     全球出口根;
4个      uint64    minForcedTimestamp ;
5个   }
```

```
1个   keccak256(
2个      阿比 编码打包（
3个         keccak256(字节        交易），
4个         字节 32     全球出口根，
5个         unint64     最小时间戳
6个      )
7     );
```

请注意，出于存储使用优化的原因，存储槽（映射条目）仅用于存储强制批处理的承诺。数据可用性得到保证，因为它可以从交易调用数据中恢复。**最小时间戳**将由合约设置为 L1 块时间戳，因此这将是强制批量发布的时间。

在 Trusted Sequencer 发生故障的极端情况下，任何用户都可以调用**sequenceForceBatches**函数对一系列强制批次进行排序。

```
1个   功能sequenceForceBatches (
2个      ForcedBatchData[]记忆批次
3个   )  民众ifNotEmergencyState isForceBatchAllowed
```

sequenceForceBatches功能就像**序列批次**函数，不同之处在于，如果允许批量强制，则可以（由任何人）调用它。这**sequenceForce-Batches**函数还将检查提交序列中的每个批次是否已发布到**强制批次**映射时间超过**FORCE_BATCH_TIMEOUT**. 由于 MATIC 批量费用已在发布时支付，因此无需再次支付。

如果强制批次序列满足所有要排序的条件，它将被添加到 **排序批次**映射为常规映射。最后一个**SequenceForceBatches**事件将被发出。

```
1个    事件SequenceForceBatches (uint64索引批号）；
```

请注意，由于使用**sequenceForce-Batches**功能永远不会处于受信任状态，节点的本地受信任 L2 状态与 L1 PolygonZkEVM.sol 合约中提交的虚拟 L2 状态之间将存在差异。节点软件准备检测和处理这种情况，并将将从 L1 获取的 L2 State 视为有效状态，以重组其本地 L2 State 实例。

图 10 显示了在对强制批处理序列进行排序时将出现的受信任和虚拟 L2 状态之间的差异。

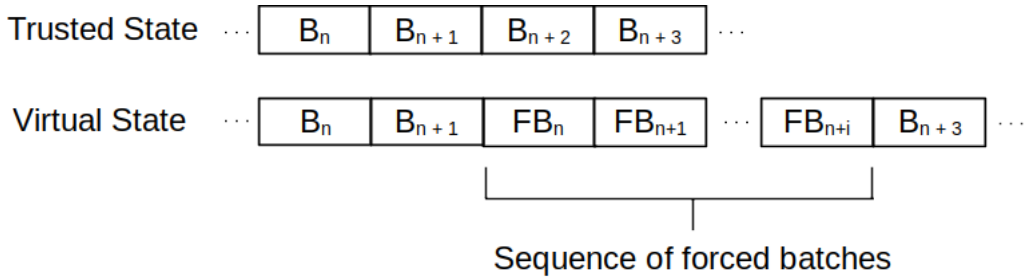

图 10：可信和虚拟 L2 状态之间的差异。

8个抵抗受信任的聚合器不活动或故障

与批次排序一样，如果 Trusted Aggregator 不存在或出现故障，上述系统将不会具有最终性，因为 L2 状态转换永远不会在 L1 中得到巩固。为此，L1 PolygonZkEVM.sol 合约有一个名为**验证批次**这允许任何人聚合批次序列。

```
1个    功能          验证批次 (
2个       uint64     pendingStateNum ,
3个       uint64     初始化编号批次，
4个       uint64     最终新批次，
5个       字节 32      新本地出口根，
6个       字节 32      新国根，
7         uint256[2]       调用数据证明A，
8个       uint256[2][2]          调用数据证明 B ,
9         uint256[2]呼叫数据              证明C
10     )  民众如果不是紧急状态
```

可以看出，**验证批次**函数采用与相同的参数**可信验证批次**， 尽管如此，**验证批次**对于要聚合的序列有两个更多的限制，进一步引入一个新的 L2 State 阶段，名为 pending state。此外

到所需的条件**可信验证批次**还必须满足以下条件**验证批次**:

- 合同不得处于紧急状态。
- A**trustedAggregatorTimeout**存储变量 delay 必须从序列中最后一批的时间戳开始传递（批次排序的时间戳）。**trustedAggregatorTimeout**变量由合约的管理员设置。

如果满足所有条件，该函数将通过调用来验证零知识 CI 证明**_验证和奖励批次**内部函数，如果验证成功，不像什么函数**可信验证批次**会做，序列不会立即聚合。验证的序列将被添加到**pendingStateTransitions** 在由设置的时间延迟期间等待聚合的映射**pendingStateTimeout**.

```
1个    // pendingStateNumber-->待定状态
2个    映射(uint256=>待定状态）民众pendingStateTransitions ;
```

```
1个    结构待定状态                    {
2个        uint64     时间戳;
3个        uint64     最后验证批次;
4个        字节 32      出口根;
5个        字节 32      状态根;
6个    }
```

已验证的批次序列将处于称为挂起状态的中间状态，其中它们的状态转换尚未合并，因此，新的 L2 状态根尚未添加到**batchNumToStateRoot**映射和网桥的新全局出口根都没有更新。这**最后一个挂起状态**storage 变量将跟踪等待合并的挂起状态转换的数量，并将用作映射中条目的键。由于零知识证明已经过验证，独立聚合者仍将获得聚合奖励。

图 11 从批处理的角度显示了 L2 阶段时间线，以及当批处理序列通过以下方式聚合时触发其包含到下一个 L2 状态阶段的操作**验证批次**功能。

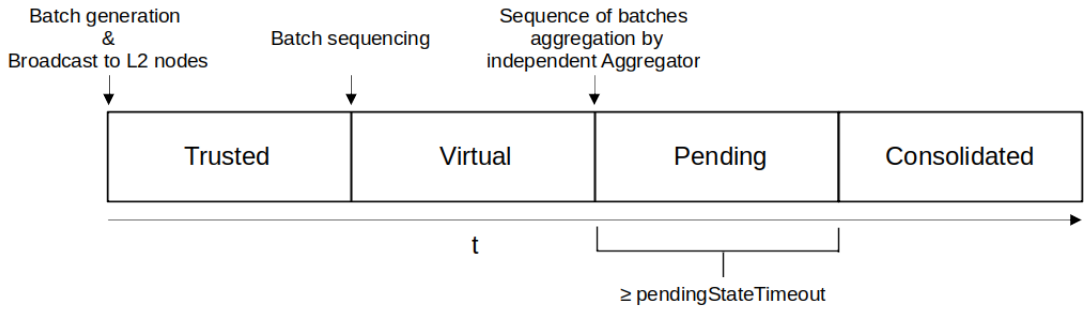

图 11：L2 State stages timeline with pending 状态。

处于待处理状态的批处理序列的存在不会影响协议的正确工作，因为将在待处理的序列之前验证进一步的序列。这**最后验证批次**存储变量将跟踪最后一批验证和聚合的索引。因此，当假装验证了一系列批次时，将通过调用的函数查询最后验证批次的索引**获取最后验证批次**. 这

如果存在未决状态转换，函数将返回处于未决状态的最后一批的索引，或者**最后验证批次**如果没有。

```
1个    功能getLastVerifiedBatch ()公众观点回报(uint64)
```

每次**序列批次**调用函数时，将通过调用来尝试合并挂起状态**_tryConsolidatePending状态**内部功能。 **_tryConsolidatePendingState**将检查是否**pendingStateTimeout**已通过批次验证的未决状态序列，如果它，将合并未决状态转换。零知识 CI 证明已经验证过了，不需要再次验证其有效性。

此外，任何人都可以通过调用来触发挂起状态的合并**consolidatePending状态**外部功能。如果来电者帐户**consolidatePending状态** 是受信任的聚合器帐户，批处理的待处理序列将直接聚合，即使**pendingStateTimeout**自其验证以来未通过。否则，如果调用方帐户不是受信任的聚合器帐户**consolidatePending状态**功能将检查是否**pendingStateTimeout**已通过批次验证的未决状态序列，如果它，将合并未决状态转换。

```
1个    功能consolidatePendingState (uint64pendingStateNum )外部的
```

这种机制旨在为 Polygon 团队提供余地，以防在零知识证明验证系统中检测到健全性漏洞利用，并保护资产不被恶意行为者桥接至 L2。

9 可升级性

为了允许将来对协议实现进行更新（在添加新功能、修复错误或优化升级的情况下），使用透明可升级代理（TUP）模式部署以下合约：

- **PolygonZkEVM.sol。**
- **PolygonZkEVMGlobalExitRoot.sol。**
- **PolygonZkEVMBridge.sol。**

为了继承安全性并避免延长和使审计过程变得更加复杂，Polygon 团队选择使用 OpenZeppelin 的 openzeppelin-upgrades 库来实现此功能。OpenZeppelin 因其对以太坊标准实施的审计和开源库而在业界赢得了知名品牌的声誉，其 openzeppelin-upgrades 库已经过审计和实战测试。此外，openzeppelin-upgrades 不仅是一套合约，还有Hardhat 和 Truffle 插件来支持代理的部署、升级和管理员权限管理。

如图12所示，Open Zeppelin的TUP模式，通过委托调用和fallback函数的使用，将存储变量的协议实现分离出来，从而提供了在不改变存储状态或改变public的情况下更新实现代码的能力合同的地址。

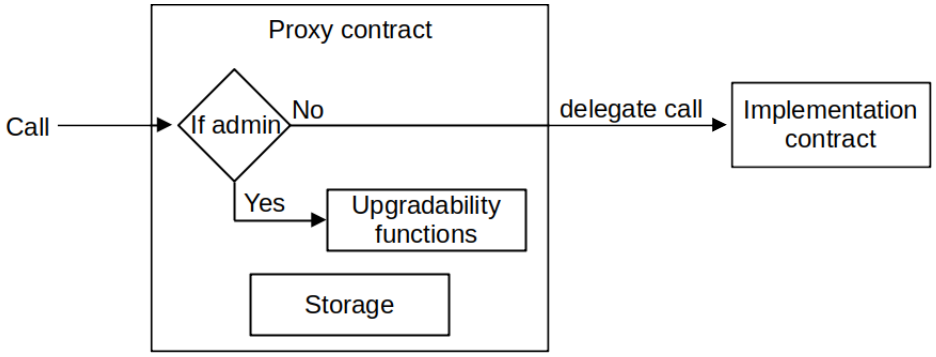

图 12：打开 Zeppelin 透明可升级代理模式架构。

按照 OpenZeppelin 的建议，部署了一个名为 ProxyAdmin.sol 的合约实例，它也包含在 openzeppelin-upgrades 库中，并将其地址设置为代理合约的管理员。使用 Hardhat 和 Truffle 插件可以安全轻松地完成这些操作。每个 ProxyAdmin.sol 实例充当每个代理的实际管理界面，每个 ProxyAdmin.sol 实例的所有者将是管理帐户。ProxyAdmin.sol 所有权将在部署期间转移到协议的管理员角色（详见第 IX 节）。

10 管理角色和治理系统

Admin 是管理整个协议的以太坊帐户。是唯一可以调用PolygonZkEVM.sol合约以下函数集的账户：

- **设置可信排序器。**
- **setForceBatchAllowed。**
- **setTrustedSequencerURL。**
- **设置可信聚合器。**
- **setTrustedAggregatorTimeout。**
- **设置挂起状态超时。**
- **setMultiplierBatchFee。**
- **setVeryBatchTimeTarget。**
- **设置管理员。**
- **停用紧急状态。**

此外，Admin 帐户是所有 ProxyAdmin.sol 实例的所有者，即可以执行协议合约实现升级操作的唯一帐户。

此外，管理员帐户拥有所有代理的所有权，这意味着它是允许对协议的合同实现执行升级的唯一帐户。

为了提高用户在使用该协议时的安全性和信心，实施了时间锁控制器。时间锁控制器是一种合约，允许设置延迟，以便在应用潜在危险的维护操作之前为用户提供退出的余地。时间锁控制器允许管理员安排和提交

L1 中的维护操作事务，当给定**最小延迟**时间到期，可以触发时间锁来执行维护操作事务。

为了继承安全性并避免延长审计过程并使其变得更加复杂，Polygon 团队选择使用 OpenZeppelin 久经考验的 TimelockController.sol 合约，但使用**得到最小延迟**函数被覆盖，这个 OpenZeppelin 实现的自定义版本被命名为 PolygonZkEVMTimelock.sol。新的**得到最小延迟** 将设定时间**最小延迟**在 zkevm 合约系统的紧急模式处于活动状态时为 0（详见第 X 节）。该协议的 Admin 角色在部署期间设置为 PolygonZkEVMTimelock.sol 合约地址的一个实例。

管理员角色需要承担重大责任，不能单独分配给一个帐户。出于这个原因，PolygonZkEVMTimelock.sol 合约实例的 Admin Ethereum 帐户被分配给一个多签合约，该合约充当协议的治理工具，将管理权分散给多个受信任的实体。

图 13 显示了 Polygon zkEVM L1 合约的治理树的形状。

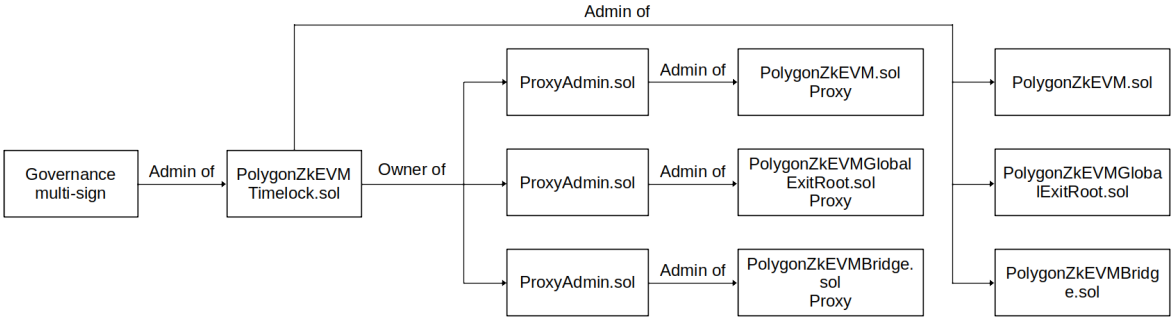

图 13：Polygon zkEVM L1 合约的治理树。

总之，协议维护操作只能按照以下步骤进行：

1. 提出维护操作交易并存储到治理多签合约中。多边形团队就是否应用这些操作达成共识。投票继承了 L1 的安全性。

2. 一旦做出决定，如果结果有利于执行维护操作，则可以触发治理多选来调度事务，一旦时间延迟过去，使用 PolygonZkEVMTimelock.sol 合约实例执行交易。

3. 一旦时间延迟过去，可以触发 PolygonZkEVMTimelock.sol 合约实例以执行预定交易并完成维护操作。

请注意，由于合约之间的治理链，代表管理员角色的任何交易只能通过上述步骤完成。

11 稳健的抗攻击性和紧急状态

紧急状态是 PolygonZkEVM.sol 和 PolygonZkEVMBridge.sol L1 合约状态，当它被激活时，批量排序和桥接操作将停止。此状态的目的是为 Polygon 团队提供解决健全性漏洞利用或智能合约漏洞利用案例的余地，并保护 L2 用户的资产。

合约处于紧急状态时，以下功能集被锁定：

- **序列批次**.
- **验证批次**.
- **consolidatePending状态**(仅当调用者不同于受信任的聚合器帐户时）。

- **强制批量**.
- **sequenceForceBatches**.
- **proveNonDeterministicPendingState**.

请注意，当合约处于紧急状态时，Sequencer 无法对批次进行排序。但是，受信任的聚合器仍将能够合并进一步的状态转换或覆盖可以证明是非确定性的未决状态转换。

当使用两个不同的结果 L2 状态根值成功验证相同的批处理序列时，将发生非确定性状态转换。这种情况可能是由于利用了零知识 CI 证明的验证系统中的健全性漏洞。

紧急状态只能由两个合约函数触发： **激活紧急状态**如果被合约所有者调用，将直接激活紧急状态。合约所有者是与管理员帐户不同的以太坊帐户，最终将被废除，因为它代表了治理链中的旁路。但是，它只能触发紧急状态激活。也可以被大家调用 **HALT_AGGREGATION_TIMEOUT**自从对应于**序列号**参数已经排序，但尚未得到验证。请注意，这种情况意味着没有人在聚合批次序列，目的是暂时停止协议，直到聚合活动可以再次恢复。

```
1个   功能激活紧急状态（uint64sequencedBatchNum )外部的
```

此外，如果能够证明某些未决状态是不确定的，任何人都可以触发紧急状态，使用 **proveNonDeterministicPendingState** 功能。

```
1个   功能          proveNonDeterministicPendingState
2个      uint64     ( initPendingStateNum ,
3个      uint64     finalPendingStateNum ,
4个      uint64     初始化编号批次，
5个      uint64     最终新批次，
6个      字节 32      新本地出口根，
7        字节 32      新国根，
8个      uint256[2]       调用数据证明A，
9        uint256[2][2]          调用数据证明 B ,
10       uint256[2]呼叫数据              证明C
11    ) 民众如果不是紧急状态
```

如果检测到稳健性漏洞利用，Trusted Aggregator 将能够覆盖非确定性挂起状态。**覆盖挂起状态**函数用于该命题。由于 Trusted Aggregator 是系统的可信实体，在存在非确定性状态转换的情况下，只有 Trusted Aggregator 提供的 L2 状态根才会被视为对合并有效。

```
1个   功能          覆盖挂起状态（
2个      uint64     初始化挂起状态数，
3个      uint64     finalPendingStateNum ,
```

```
 4个       uint64    初始化编号批次，
 5个       uint64    最终新批次，
 6个       字节 32     新本地出口根，
 7         字节 32     新国根，
 8个       uint256[2]       调用数据证明A，
 9         uint256[2][2]          调用数据证明 B ，
10         uint256[2]呼叫数据              证明C
11      ) 民众仅受信任的聚合器
```

要成功覆盖挂起状态，Trusted Aggregator 必须提交一份证明，该证明将按照**proveNonDeterministicPendingState**功能，如果成功，挂起的状态转换将被擦除，并直接合并一个新的状态转换。

综上所述，可以触发紧急状态：

- 当合同所有人认为合适时。
- 当聚合活动停止时**HALT_AGGREGATION_TIMEOUT** 时间段（1周））
- 当任何人都能够证明挂起状态是不确定的时。