链接：https://mp.weixin.qq.com/s?__biz=Mzg2NTUxMTY0NQ==&mid=2247483984&idx=1&sn=18ba6bcdc3180de3cbcdc793f0abd55f&chksm=ce59b3e2f92e3af4bae8abbfa8f7a09c0db677f45c3d44484f2a9e4ea5bc9b9ad6efe2b39211&scene=21#wechat_redirect

3月27日，Polygon zkEVM主网测试版本正式上线，Vitalik 在上面完成了第一笔交易。
本文是Polygon zkEVM系列文章的第一篇，**简要阐述了Polygon zkEVM的整体架构和交易执行流程，**
并且分析了Polygon zkEVM是如何实现计算扩容并同时继承以太坊的安全性的。
同时还会在接下来两篇文章里详细介绍Polygon zkEVM的**zkEVM Bridge和zkEVM的设计细节**，
以及Polygon zkEVM接下来的去中心化Sequencer的路线图。

一、Rollup为了给以太坊实现计算扩容

首先，我们需要明确Rollup的大概工作原理。
Rollup的出现是为了给Ethereum实现计算扩容，具体的实现方法是将交易的执行外包给Rollup，
然后将交易和交易执行后的状态(State)存储在 Ethereum 的合约内。
由于技术路线的不同演变出了两种类型的 Rollup：
Optimistic Rollup：乐观的认为发送到 Ethereum 的 Rollup 交易(Rollup Transaction)和对应的 Rollup 状态(Rollup State)都是正确的，
任何人都可以通过提供欺诈证明(Fraud Proof)对还处于挑战期的Rollup State进行挑战(Challenge)。

Zero-knowledge Rollup：ZK会为发送到 Ethereum 的 Rollup交易和对应的 Rollup 状态提供一个有效性证明
(由以太坊上的合约验证，来证明 Rollup 的执行对应交易后的状态是正确的)。

> 参考以太坊官方定义：https://ethereum.org/en/developers/docs/scaling/#rollups

Zero-knowledge Rollup 和 Optimistic Rollup 最大的区别就是由于验证状态有效性的不同方式导致达成 Finality 的时间不同：

Optimistic Rollup：乐观的认为提交到 Ethereum 上的交易和状态都是正确的，
所以存在7天的挑战期（达成Finality的时间是7天），期间任何人发现在 Ethereum 上的交易对应状态不正确都可以通过提交正确的状态进行挑战。

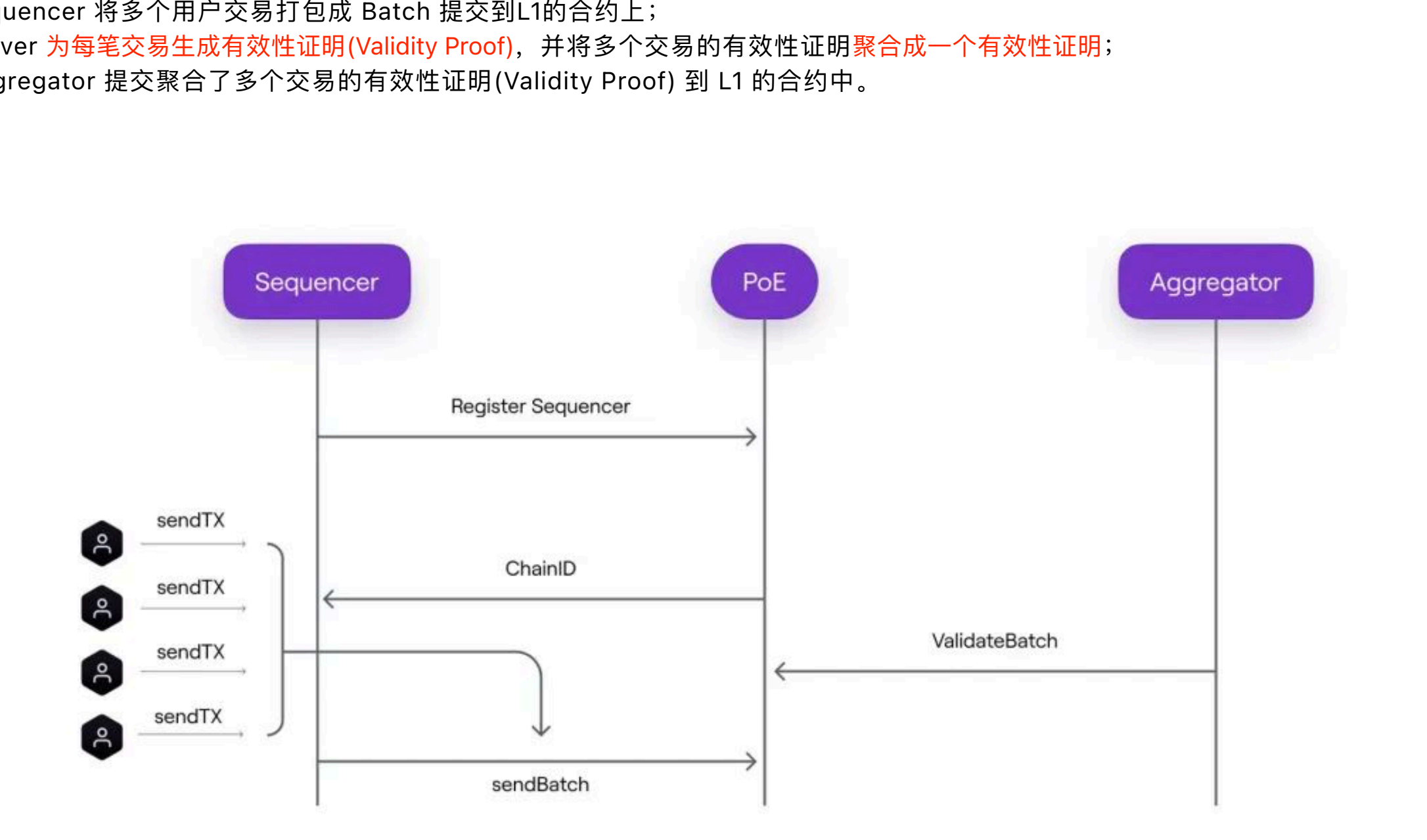
Zero-knowledge Rollup(zk-Rollup)：达成 Finality 的时间，则取决于：
交易对应的有效性证明(Validity Proof)提交到以太坊并且验证通过所花费的时间。
目前可能在1个小时左右的Finality居多(因为需要考虑到Gas成本问题)。

二、Polygon zkEVM 执行流程

接下来我们以一个简单的交易被确认流程来看看 Polygon zkEVM是怎么工作的，从而对整体协议有一个具体的理解。

它的整个过程可以主要分为三个步骤：

1. Sequencer 将多个用户交易打包成 Batch 提交到L1的合约上；
2. Prover 为每笔交易生成有效性证明(Validity Proof)，并将多个交易的有效性证明聚合成一个有效性证明；
3. Aggregator 提交聚合了多个交易的有效性证明(Validity Proof) 到 L1 的合约中。



1. Sequencer 将用户交易打包成 Batch 提交到 L1 合约上
1) 用户将交易发送给Sequencer，Sequencer会在本地按照收到交易的快慢顺序进行处理(FRFS)，
当Sequencer在本地将交易执行成功后，如果用户相信Sequencer是诚实的，那么他可以认为这个时候的交易已经达成了Finality。
这里需要注意，目前大多数Sequencer内部的Mempool(交易池)都是私有的，所以暂时可以获取的MEV是比较少的。

2) Sequencer 会将多笔交易打包进一个Batch里(目前是一个Batch里只包含一个交易) 然后在收集到多个Batches之后，
通过L1上的 PolygonZKEvm.sol的SequenceBatch()函数将多个Batches一起送到L1的交易Calldata上。

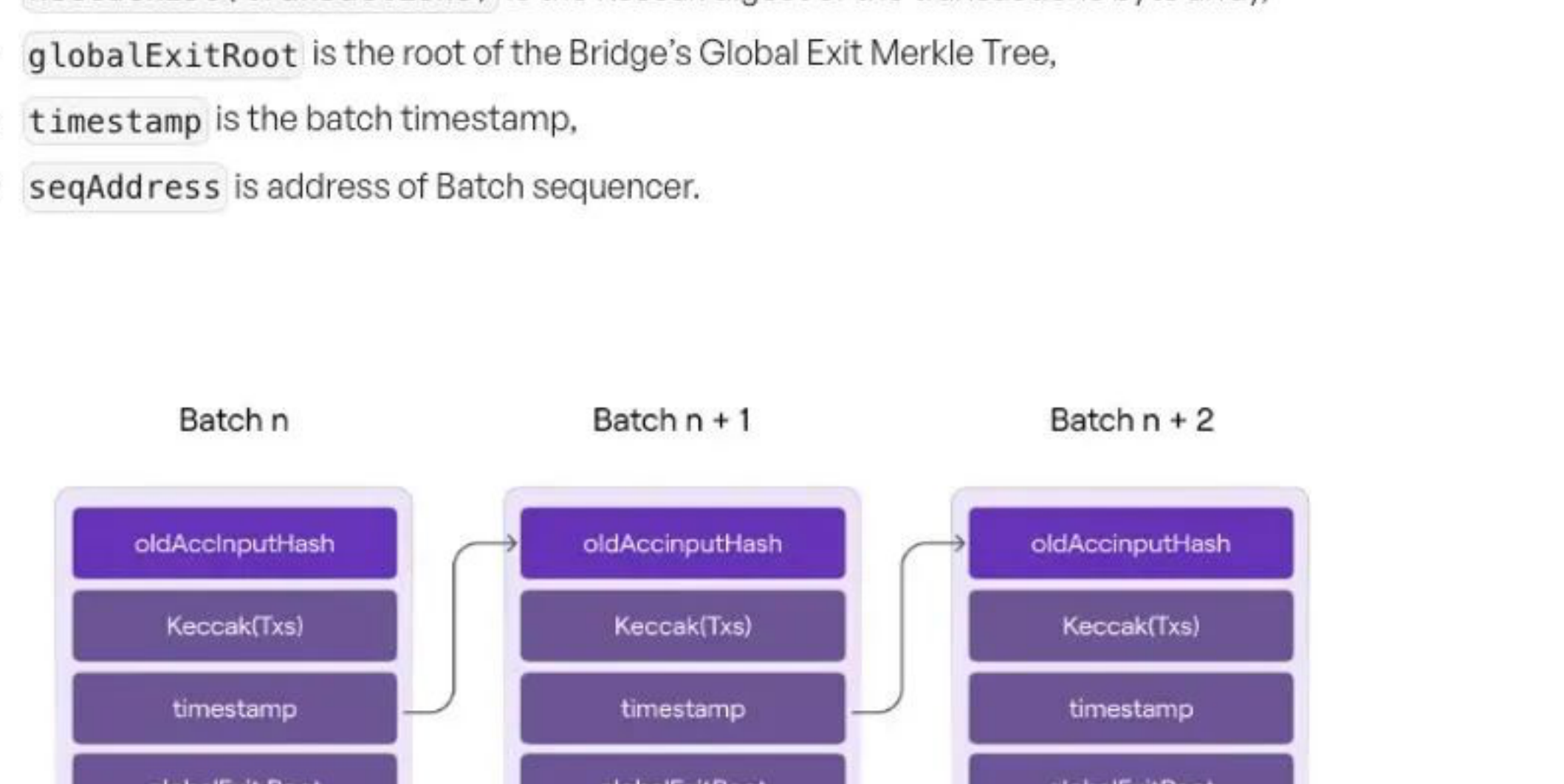
```
function sequenceBatches(
        BatchData[] memory batches
    ) public ifNotEmergencyState onlyTrustedSequencer
```

(需要注意这里一次性提交多个Batches是为了尽可能减少L1的Gas消耗)

3) 当 PolygonZkEvm.sol 收到 Sequencer 提供的 Batches 后，

它会依次在合约内计算每个Batch的哈希，然后在后一个Batch里记录前一个Batch的哈希，于是我们就得到了下图的Batch结构。

- `oldAccInputHash` is the accumulated hash of the previous sequenced batch,
- `keccack256(transactions)` is the Keccak digest of the transactions byte array,
- `globalExitRoot` is the root of the Bridge's Global Exit Merkle Tree,
- `timestamp` is the batch timestamp,
- `seqAddress` is address of Batch sequencer.



4) 每个Batch里的交易顺序也是确定的，所以当Batch的顺序确定之后，
我们认为所有被包含在Batch里提交到L1的 Polygon zkEVM合约的交易的顺序都被确定了。

The below figure shows the logic structure of a sequence of batches.

以上实际过程也是L1充当Rollup DA层需要完成的工作(这个时候并没有完成任何状态检验或推进的工作)。

2. Aggregator 为多个Batch的交易生成 Validity Proof
1) 当Aggregator监听到L1的 PolyonZKEVM.sol 合约中已经有新的 Batch 被成功的提交之后，
它会把这些 Batch 同步到自己的节点里，然后给 **zkProver** 发送这些交易。

2) zkProver 接收到这些交易之后会并行为每笔交易生成 **Validity Proof**，
再将多个Batch包含的交易的 Validity Proof再聚合成一个有效性证明(Validity Proof)。

3) zkProver 将聚合多个交易的Validity Proof发送给 Aggregator。

3. Aggregator 提交聚合证明到 L1 的合约：
Aggregator 会将这个有效性证明(Validity Proof)以及对应的这些 Batch 执行后的状态一起提交到 L1 的 PolygonzkEvm.sol 合约内，
通过调用以下方法：

```
~/Web3/zk/polygon-zkEVM/zkevm-contracts/contracts/mocks
 * @notice Allows an aggregator to verify multiple batches
 * @param pendingStateNum Init pending state, 0 when consolidated state is used
 * @param initNumBatch Batch which the aggregator starts the verification
 * @param finalNewBatch Last batch aggregator intends to verify
 * @param newLocalExitRoot  New local exit root once the batch is processed
 * @param newStateRoot New State root once the batch is processed
 * @param proofA zk-snark input
 * @param proofB zk-snark input
 * @param proofC zk-snark input
 */
ftrace | funcSig
function trustedVerifyBatches(
    uint64 pendingStateNum,
    uint64 initNumBatch,
    uint64 finalNewBatch,
    bytes32 newLocalExitRoot,
    bytes32 newStateRoot,
    uint256[2] calldata proofA,
    uint256[2][2] calldata proofB,
    uint256[2] calldata proofC
) public onlyTrustedAggregator {
```

合约内接下来会执行以下操作来验证状态转换是否正确。

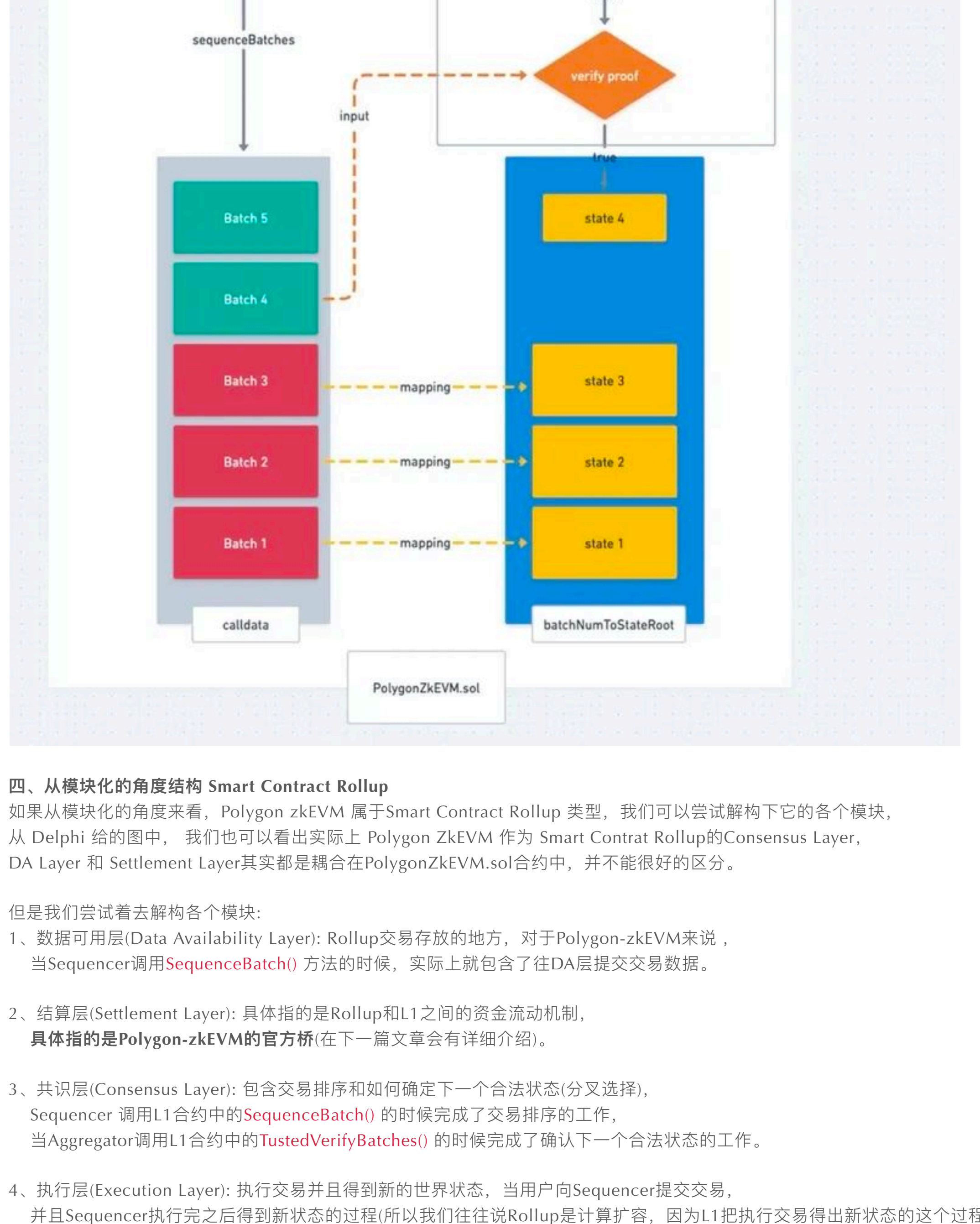
```
    // Get snark bytes
    bytes memory snarkHashBytes = getInputSnarkBytes(
        initNumBatch,
        finalNewBatch,
        newLocalExitRoot,
        oldStateRoot,
        newStateRoot
    );

    // Calulate the snark input
    uint256 inputSnark = uint256(sha256(snarkHashBytes)) % _RFIELD;

    // Verify proof
    require(
        rollupVerifier.verifyProof(proofA, proofB, proofC, [inputSnark]),
        "PolygonZkEVM::_verifyBatches: Invalid proof"
    );
```

当这一步在L1合约内执行成功时，这部分batch包含的所有交易也就真正达成了Finality（对应OP的7天挑战期结束）。

三、Ethereum 在 Polygon-zkEVM 中充当的角色
上文我们已经了解了Polygon zkEVM的整体流程, 可以回顾下Ethereum 为 Rollup 做了哪些工作:

第一步，Sequencer 将 Rollup 的交易收集起来打包成 Batch 之后，提交到L1的合约中。
L1不仅仅提供了DA层的功能，实际上还完成了一部分交易排序的功能；
当你把交易提交到Sequencer时，交易是没有真正被定序的，因为Sequencer有权力可以随便改变交易的顺序，
但是当交易被包含在Batch里提交到L1的合约上之后，任何人都没有权利再修改其中的交易顺序。

第二步，Aggregator 将Validity Proof 提到L1合约上来达成新的状态,Aggregator则是类似**Proposer**的角色，合约则类似Validator的角色：
Aggregator 提供了一个Validity Proof来证明一个新的状态是正确的，
并告诉Validator我提供的Validity Proof涉及哪些交易Batch，他们都存在了L1的哪个位置。
接着Validator从合约中提取对应的Batch，与Validity Proof结合在一起就可以验证状态转换的合法性了。
如果验证成功实际上合约内也会更新到对应Validity Proof的新状态。



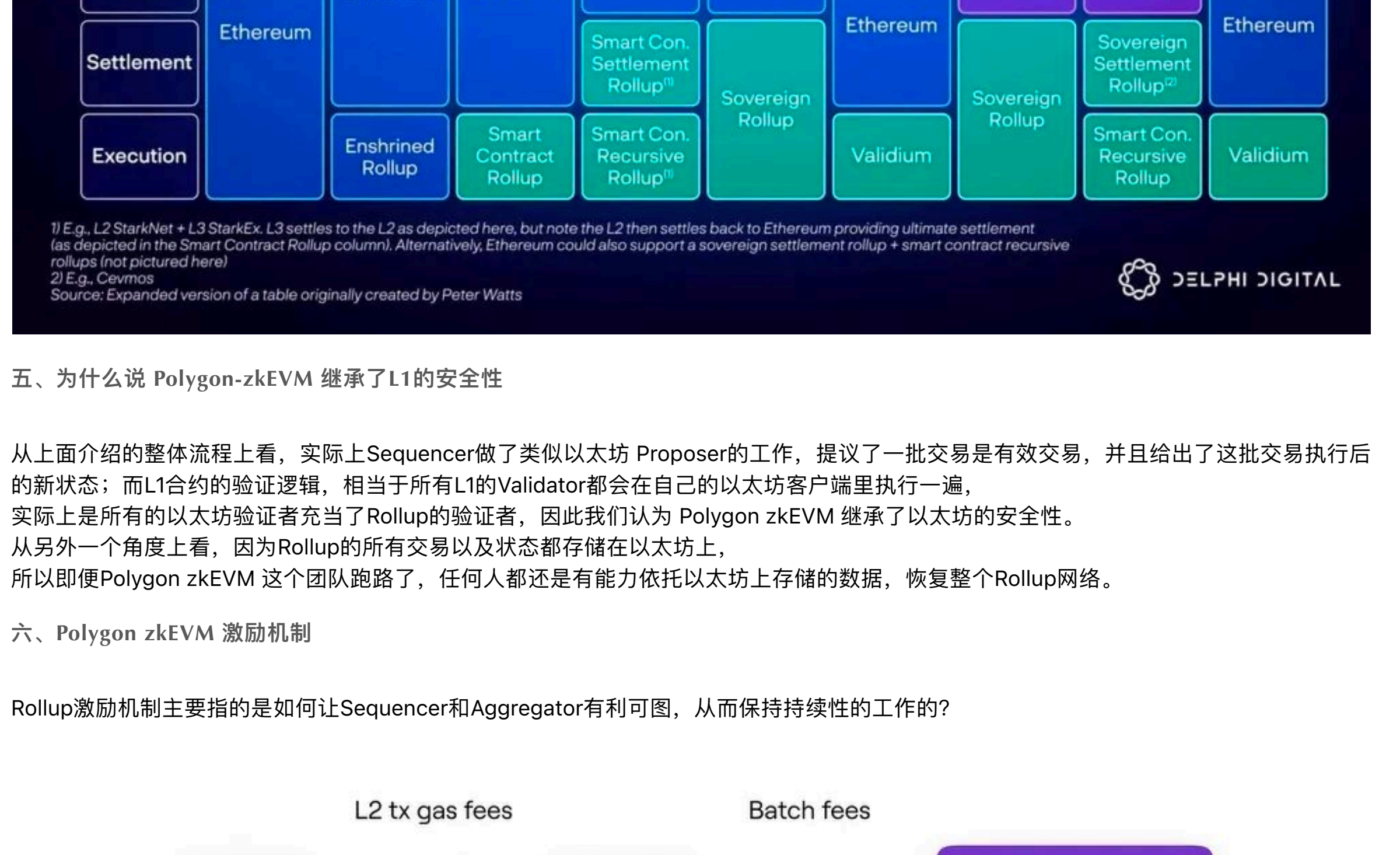
四、从模块化的角度结构 Smart Contract Rollup
如果从模块化的角度来看，Polygon zkEVM 属于Smart Contract Rollup 类型，我们可以尝试解构下它的各个模块，
从 Delphi 给的图中，我们也可以看出实际上 Polygon ZkEVM 作为 Smart Contrat Rollup的Consensus Layer，
DA Layer 和 Settlement Layer其实都是耦合在PolygonZkEVM.sol合约中，并不能很好的区分。

但是我们尝试着去解构各个模块：
1、数据可用层(Data Availability Layer)：Rollup交易存放的地方，对于Polygon-zkEVM来说，
当Sequencer调用SequenceBatch() 方法的时候，实际上就包含了往DA层提交交易数据。

2、结算层(Settlement Layer)：具体指的是Rollup和L1之间的资金流动机制，
具体指的是Polygon-zkEVM的官方桥(在下一篇文章会有详细介绍)。

3、共识层(Consensus Layer)：包含交易排序和如何确定下一个合法状态(分叉选择)，
Sequencer 调用L1合约中的SequenceBatch() 的时候完成了交易排序的工作，
当Aggregator调用L1合约中的TustedVerifyBatches() 的时候完成了确认下一个合法状态的工作。

4、执行层(Execution Layer)：执行交易并且得到新的世界状态，当用户向Sequencer提交交易，
并且Sequencer执行完之后得到新状态的过程(所以我们往往说Rollup是计算扩容，因为L1把执行交易得出新状态的这个过程外包给了Rollup，
同时Sequencer会通过Aggregator委托zkProver帮忙生成Validity Proof。



五、为什么说 Polygon-zkEVM 继承了L1的安全性

从上面介绍的整体流程上看，实际上Sequencer做了类似以太坊 Proposer的工作，提议了一批交易是有效交易，并且给出了这批交易执行后的新状态；而L1合约的验证逻辑，相当于所有L1的Validator都会在自己的以太坊客户端里执行一遍，
实际上是所有的以太坊验证者充当了Rollup的验证者，因此我们认为 Polygon zkEVM 继承了以太坊的安全性。
从另外一个角度上看，因为Rollup的所有交易以及状态都存储在以太坊上，
所以即便Polygon zkEVM 这个团队跑路了，任何人都还是有能力依托以太坊上存储的数据，恢复整个Rollup网络。

六、Polygon zkEVM 激励机制

Rollup激励机制主要指的是如何让Sequencer和Aggregator有利可图，从而保持持续性的工作的?

首先用户需要支付自己在Rollup上的交易手续费，这部分的手续费是采用ETH计价的，用Bridged ETH支付。
Sequencer 则需要支付这些包含Rollup交易的Batch上传到L1交易的Calldata上的成本(调用SequenceBatch(batches())的成本)，
同时需要在上传Batch的同时支付一定的Matic到L1合约中，用于之后支付Aggregator为这些Batches提供Validity Proof的成本。

Aggregator 在调用Trusted VerifyBatches 为L1合约内还没有被Finality的Batches提供Validity Proof的同时，
也可以取出Sequencer提前支付在合约内的MATIC代币，作为提供Validity Proof的报酬。

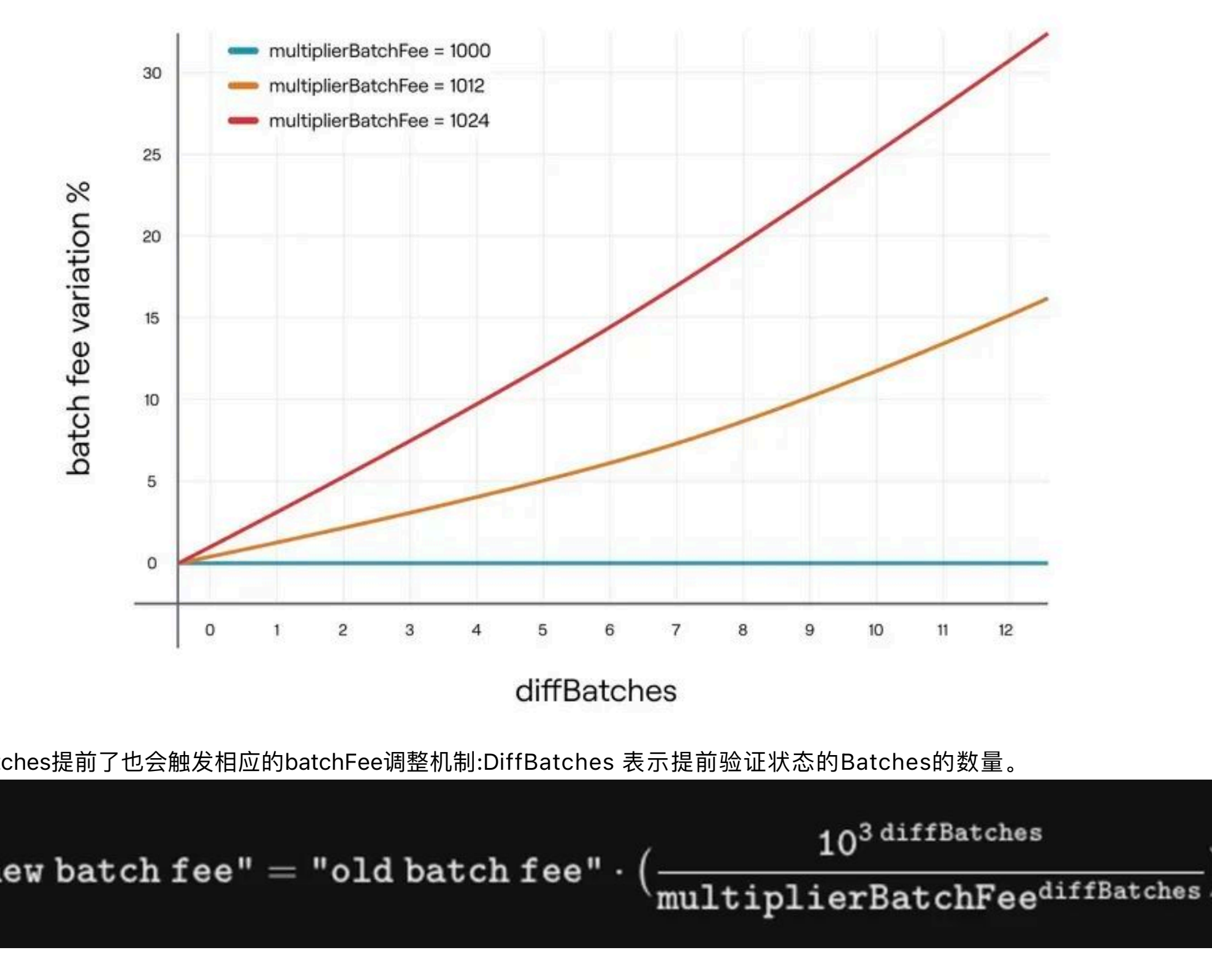
Sequencer的收入 = Rollup所有交易的Gas费用 - 将Batches上传到L1花费的L1网络Gas费用 - 支付给Aggregator的证明费用(MATIC计价)。
Aggregator的收入 = Sequencer支付的MATIC报酬 - 提交到Validity Proof到 L1的Gas费用 - Validity Proof生成花费的硬件费用。

调整支付给Aggregator的证明费用，同时为了避免Sequencer因为无利可图罢工，提供了以下的机制来调整Sequencer支付给Aggregator的证明费用。

合约中存在这样一个方法用来调整为Batch提供证明的费用:
function _updateBatchFee(uint64 newLastVerifiedBatch) internal它会更改合约中一个名为BatchFee的变量，
而这个变量决定了Sequencer为每个Batch支付的MATIC代币数量。

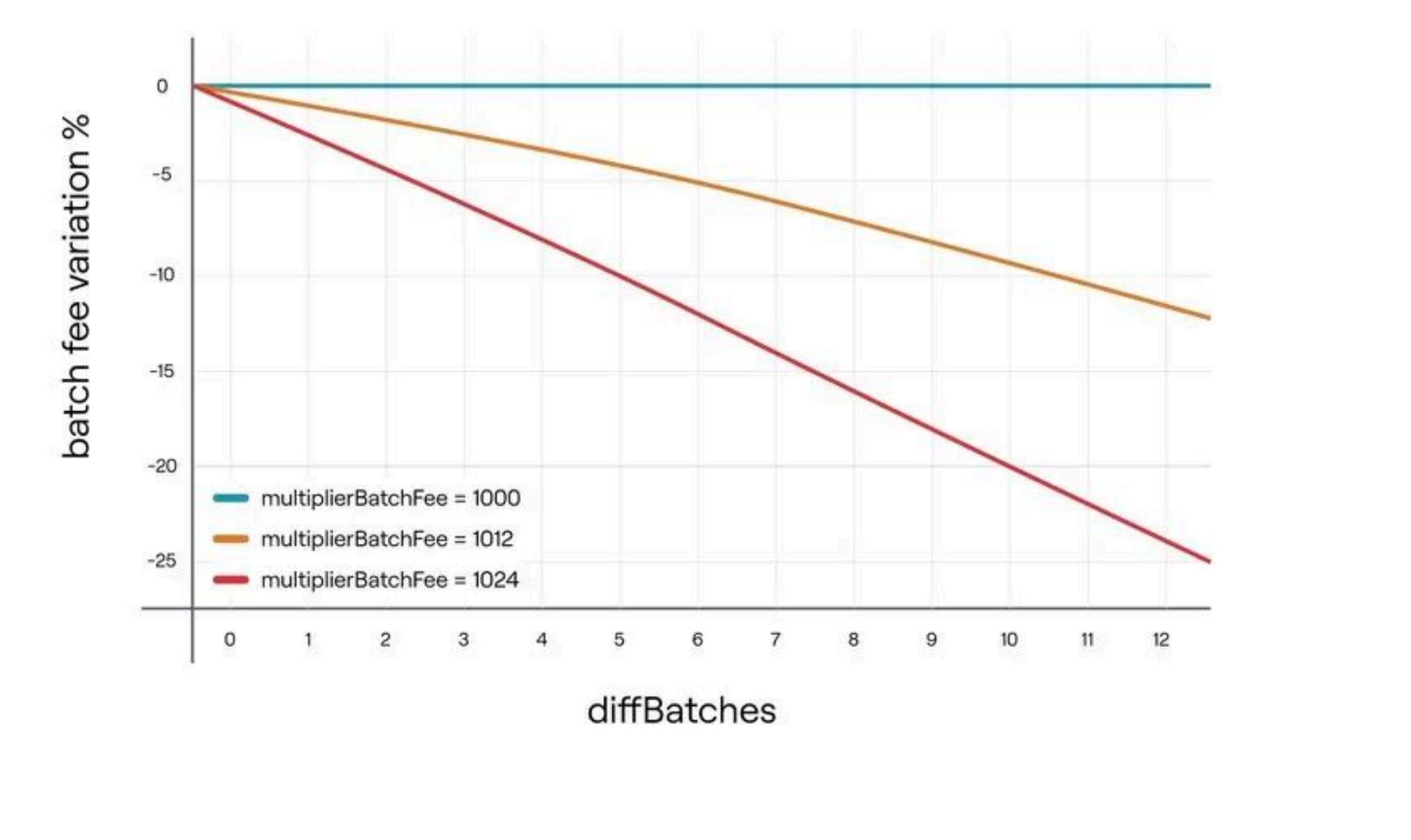
更改机制如下：
合约中维护了这样一个变量VeryBatchTimeTarget，代表每个Batch被Sequencer提交到L2之后期望在这个时间内被验证状态。
合约内会记录所有超过了VeryBatchTimeTarget之后还没有被验证状态的Batches, 并且将这些Batches的总数量记为DiffBatches。
于是当有Batches迟到的时候，会用以下公式来调整BatchFee:MultiplierBatchFee 是一个被限制在1000~1024范围的数，
可以通过函数setMultiplierBatchFee() 由合约管理员更改：
Function setMultiplier BatchFee (uint16newMultiplierBatchFee) public onlyAdmin需要注意这里的
采用MultiplierBatchFee 和10^3是为了实现3个小数点后的调整精度。

$$\text{"new batch fee"} = \text{"old batch fee"} \cdot \left(\frac{\text{multiplierBatchFee}^{\text{diffBatches}}}{10^{3\,\text{diffBatches}}}\right)$$



同理假如Batches提前了也会触发相应的batchFee调整机制:DiffBatches 表示提前验证状态的Batches的数量。

$$\text{"new batch fee"} = \text{"old batch fee"} \cdot \left(\frac{10^{3\,\text{diffBatches}}}{\text{multiplierBatchFee}^{\text{diffBatches}}}\right)$$



总结

在这篇文章里我们梳理了Polygon zkEVM的核心机制，并分析了它实现以太坊计算扩容的可行性。
有了一个整体的大纲后，在接下来的文章里我们会深入到协议内部，
依次解析zkEVM Bridge的设计细节以及Sequencer的去中心化路线，zkProver的实现以及zkEVM的设计原理。