Solidity Ethereum Uniswap

Uniswap v3 详解（五）：Oracle 预言机

Posted on April 5, 2021

Uniswap v2 的 Oracle

Uniswap v2 版本起，就已经提供了预言机功能，用来计算交易对的时间加权平均价（TWAP, Time Weighted Average Price），但是存在下面几个问题：

- v2 版本的预言机使用门槛比较高，官方合约中只提供了最新的价格累积值，开发者需要自己实现历史价格的记录和抓取
- 在使用 DEX 预言机时，深度是一个比较重要的考量指标，深度决定了此交易对预言机被攻击的难易程度，但是在 v2 版本我们无法通过简单的方式获取交易对深度变化的历史数据

Uniswap v2 版本中，Pair 合约中记录了 `price0CumulativeLast`，`price0CumulativeLast` 两个状态值，分别表示上一次价格更新时，价格的时间累积值，计算公式为：

$$a_t = \sum_{i=1}^{t} p_i$$

通过访问不同时间点的价格时间累积值，我们可以计算一个交易对在过去一段时间内的价格平均值。假设当前时间为 t_1，5min 前的时间为 t_2，那么可以计算过去 5min 交易对的平均价格：

$$p_{t_2,t_1} = \frac{\sum_{i=t_1}^{t_2} p}{t_2 - t_1} = \frac{\sum_{i=1}^{t_2} p - \sum_{i=1}^{t_1} p}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

这给开发者带来了一个问题是：开发者需要自己记录历史时间点中的价格累积值 a_t. 而这些工作是相对比较复杂的。Gtihub 中有人开源过通用的 Uniswap v2 Oracle 实现，Uniswap v2 官方也给出了一个相对简单的 Oracle 合约实现，最近大热的 Fei Protocol 中使用的 Oracle 就是参考这个样例实现的。

Uniswap v3 的 Oracle

Uniswap v3 版本针对 v2 版本 Oracle 的痛点，进行了改进：

- 合约中默认还是存储一个最近价格的时间累积值，但是可以根据需要，扩展为存储最近 N 个历史价格的时间累积值，最多支持 65535 个最近历史价格信息（还可以包含当前未被写入的价格信息，这样就是 65536 个值），这样第三方开发者不再需要自己实现合约存储历史信息
- Oracle 中不光记录了价格信息，还记录了对应流动性的时间累积值，因为 v3 中相同交易对在不同费率时时不同的交易池，这样在使用 Oracle 时，可以选择流动性较大的池最为价格参考来源
- Uniswap v2 中可以计算出时间加权平均价格（算术平均值），而 v3 中计算出来的是时间加权价时几何平均值，团队称几何平均值比算术平均值更适合，但我个人认为这个主要是考虑到工程实现的复杂度所做出的改变

算术平均值与几何平均值

在 v2 版本，计算平均价格时，使用累积值相减，再除以时间差值：

$$P_{t_1,t_2} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

这里求得的是算术平均值。

在 v3 版本中，实际在 Oracle 中记录的是 tick index 的时间累积值，而并不是价格的时间累积值。tick index 与价格的关系为：

$$i = log_{\sqrt{1.0001}} \sqrt{P_i} = log_{1.0001} P_i$$

即 tick index i 为其对应价格的对数值。在 v3 Oracle 中记录的是此值的时间累积值，即：

$$a_t = \sum_{i=1}^{t} log_{1.0001} P_i$$

在使用时，我们求这个值的算术平均值：

$$\frac{\sum_{i=t_1}^{t_2} log_{1.0001} P_i}{t_2 - t_1}$$

可以转换称为价格 P 的几何平均值形式（几何平均值其实是其对数域上的算数平均值）：

$$\frac{\sum_{i=t_1}^{t_2} log_{1.0001} P_i}{t_2 - t_1} = log_{1.0001} \left(\prod_{i=t_1}^{t_2} P_i \right)^{\frac{1}{t_2 - t_1}}$$

其中 $(\prod_{i=t_1}^{t_2} P_i)^{\frac{1}{t_2 - t_1}}$ 即为价格 P 在 $[t_1, t_2]$ 时间段的几何平均值。那么我们就可以通过 Uniswap v3 的 Oracle 接口求出一段时间内交易对价格的时间加权几何平均值。

令 P_{t_1,t_2} 为 $[t_1, t_2]$ 区间的交易对价格的几何平均值，a_1, a_2 分别为 t_1, t_2 时间点 tick index 的时间加权累积值，那么 P_{t_1,t_2} 的计算方式为为：

$$log_{1.0001}(P_{t_1,t_2}) = \frac{\sum_{i=t_1}^{t_2} log_{1.0001} P_i}{t_2 - t_1}$$

$$log_{1.0001}(P_{t_1,t_2}) = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

$$P_{t_1,t_2} = 1.0001^{\frac{a_{t_2} - a_{t_1}}{t_2 - t_1}}$$

通过上述公式，第三方开发者可以计算过去一段时间内，某个交易对价格的时间加权平均值，其结果是几何平均值。

为什么使用几何平均值

算术平均数的优势是其简单性，也是最符合直觉的平均数。当用于计算算术平均数的数据系列越长，其算术平均值就有越高的机率接近期望值，这种现象在统计学中被称为「大数法则」。

几何平均数一般来说都会小于算术平均数，因此但是对于波动较大的数字而言，使用几何平均数，其受波动性的影响会更小。

具体到 Uniswap 项目中：

- 从工程实现角度看
- 合约中使用 tick index 来表示价格区间，存储 tick index 的时间加权累积值，使得 Oracle 的实现更为简单直接，避免了在存储 Oracle 数据过程中进行指数计算，在计算时可以更节省 gas
- 在 v2 版本中，需要对 token0 和 token1 的 Oracle 分别进行存储（因为价格的算术平均数并不是互为倒数的）。而 v3 使用了几何平均数，token0 和 token1 的价格的几何平均数互为倒数，这样合约只存储代币对中一个代币的的 Oracle 值即可，在存储时也更加节省 gas 费用
- 从金融数学角度看
- 交易对价格的走势可以通过几何布朗运动模型化，在几何布朗运动中算术平均数，会给予高价格更高的权重，平均价格更容易受到波动性的影响。

攻击成本

一个攻击者在攻击代币对的 Oracle 时，为了成本最小化，通常会采用以下策略：

- 攻击者在同一个区块中，在 Oracle 写入的前后先买入再卖出某种资产，以实现低成本操纵 Oracle 数据的目的

防御措施

Uniswap v3 沿用了一些 v2 版本中的设计，以增加对 Oracle 进行攻击的成本。具体如下：

1. 在同一个区块内，Oracle 的写入操作只会发生在这个区块中的第一笔对价格发生了改变的交易中
2. 在写入 Oracle 时，写入的并不是本次交易结束时的价格，而是交易前的价格，即上一次交易的最终价格。又因为第一条的原因，这样实际上在更新本次 Oracle 数据时使用的是上一次交易所在区块中最后一个交易的价格。

这样做之后：

- 攻击者无法在一个区块内完成对 Oracle 数据的操纵，只能分多笔交易跨区块才有可能实现 Oracle 数据的操纵
- 多笔跨区块的交易还必须要发生在对应区块的末尾和起始交易，这也是很难实现的

这些限制无疑增加了攻击者对代币对进行攻击的成本，使得低成本攻击变得困难。

当然，随着 flashbots/MiningDAO 之类的工具的流行，这种攻击也不是不可能实现，但是这是另外的话题了，不在本文讨论范围内。

代码实现

Oracle 实现的代码都在 uniswap-v3-core 的合约中实现。Oracle 相关的操作主要在以下情况下发生：

- 初始化交易池时，初始化 Oracle，但是此时 Oracle 中只有一个槽位，即只保存最近的一份数据
- 发生交易时，价格会变动，此时需要更新 Oracle
- 外部开发者可以调用接口获取历史 Oracle 数据，计算出 TWAP
- 对交易对历史 Oracle 数据由需求的开发者可以调用交易池提供的接口扩展 Oracle 数据存储的数量

存储

Oracle 数据使用一个结构体 `Observation` 来表示：

```
struct Observation {
    // 记录区块的时间戳
    uint32 blockTimestamp;
    // tick index 的时间加权累积值
    int56 tickCumulative;
    // 价格所在区间的流动性的时间加权累积值
    uint160 liquidityCumulative;
    // 是否已经被初始化
    bool initialized;
}
```

在交易池的合约中，使用一个数组来存储交易池最近的的 Oracle 数据：

```
contract UniswapV3Pool is IUniswapV3Pool, NoDelegateCall {
    ...
    // Oracle 相关操作的库
    using Oracle for Oracle.Observation[65535];
    ...
    // 使用数据记录 Oracle 的值
    Oracle.Observation[65535] public override observations;
    ...
    struct Slot0 {
        ...
        // 记录了最近一次 Oracle 记录在 Oracle 数组中的索引位置
        uint16 observationIndex;
        // 已经存储的 Oracle 数量
        uint16 observationCardinality;
        // 可用的 Oracle 空间，此值初始时会被设置为 1，后续根据需要来可以扩展
        uint16 observationCardinalityNext;
    }
    Slot0 public override slot0;
}
```

数组的大小为 65535，但是实际上在初始阶段这个数据并不会被全部使用，而仅使用其中一部分空间（初始为 1）。这样做的目的是，如果没有必要，那么仅存储最近一份 Oracle 数据即可，因为写入数据到数组中需要比较高昂的 gas 费用（`SSTORE` 操作）。

当第三方对某个交易池的 Oracle 有需求时，可以主动调用合约的接口扩展这个数据的可用空间，这样后续合约会存储更多的 Oracle 数据。

同时在这个接口中，会对数组中数据进行写入，这样后续交易发生时，不需要进行存储空间的扩展，而仅需要更新其中的值。这样也将扩展 Oracle 空间需要的手续费从代币交易者转移到了 Oracle 的实际需求方。这方面内容会在后文介绍。

当数组可用大小写满之后，它会重新从 0 开始写入，即使用上类似一个 ring buffer.

初始化

在之前创建交易对的文章中说过，创建交易对时，`UniswapV3Factory` 会调用新创建交易对的 `UniswapV3Pool.initialize()` 函数对合约进行初始化：

```
function initialize(uint160 sqrtPriceX96) external override {
    ...

    // 初始化 Oracle
    (uint16 cardinality, uint16 cardinalityNext) = observations.initialize(_blockTimestamp());

    ...
}
```

在初始化的代码中，调用了 `observations.initialize(_blockTimestamp())` 来进行 Oracle 的初始化：

```
function initialize(Observation[65535] storage self, uint32 time)
    internal
    returns (uint16 cardinality, uint16 cardinalityNext)
{
    self[0] = Observation({blockTimestamp: time, tickCumulative: 0, liquidityCumulative: 0, initi
    // 返回当前 Oracle 的个数和最大可用个数
    return (1, 1);
}
```

可以看到初始化就是写入一个空值 Oracle 数据，并且返回了当前 Oracle 的个数和最大可用个数。最大可用个数即 `cardinalityNext` 为 1，表示合约初始化时，只会记录最近的一次 Oracle 数据。

Oracle 更新

Oracle 数据的更新发生在价格变动的时候，前文说过，为了防止攻击，同一个区块内，只会在第一次发生价格变动时写入 Oracle 信息。在 `UniswapV3Pool.swap()` 函数中：

```
// 检查价格是否发生了变化，当价格变化时，需要写入一个 Oracle 数据
if (state.tick != slot0Start.tick) {
    // 写入 Oracle 数据
    (uint16 observationIndex, uint16 observationCardinality) =
        observations.write(
            // 交易前的最新 Oracle 索引
            slot0Start.observationIndex,
            // 当前区块的时间
            cache.blockTimestamp,
            // 交易前的价格的 tick ，如前文所述，这样做是为了防止攻击
            slot0Start.tick,
            // 交易前的价格对应的流动性
            cache.liquidityStart,
            // 当前的 Oracle 数量
            slot0Start.observationCardinality,
            // 可用的 Oracle 数量
            slot0Start.observationCardinalityNext
        );
    // 更新最新 Oracle 指向的索引信息以及当前 Oracle 数据的总数目
    (slot0.sqrtPriceX96, slot0.tick, slot0.observationIndex, slot0.observationCardinality) = (
        state.sqrtPriceX96,
        state.tick,
        observationIndex,
        observationCardinality
    );
} else {
    ...
}
```

这里首先检查价格是否发生了变化，当价格变化时，需要写入 Oracle 数据，调用的是 `observations.write` 函数：

```
function write(
    Observation[65535] storage self,
    uint16 index,
    uint32 blockTimestamp,
    int24 tick,
    uint128 liquidity,
    uint16 cardinality,
    uint16 cardinalityNext
) internal returns (uint16 indexUpdated, uint16 cardinalityUpdated) {
    // 获取当前的 Oracle 数据
    Observation memory last = self[index];

    // 如前文所述，同一个区块内，只会在第一笔交易中写入 Oracle 数据
    if (last.blockTimestamp == blockTimestamp) return (index, cardinality);

    // 检查是否需要使用新的数组空间
    if (cardinalityNext > cardinality && index == (cardinality - 1)) {
        cardinalityUpdated = cardinalityNext;
    } else {
        cardinalityUpdated = cardinality;
    }

    // 本次写入的索引，使用余数实现 ring buffer
    indexUpdated = (index + 1) % cardinalityUpdated;
    // 写入 Oracle 数据
    self[indexUpdated] = transform(last, blockTimestamp, tick, liquidity);
}
```

写入时会计算出需要使用的索引数，如果可用空间用满会重新从头开始写入。Oracle 数据使用 `transform` 函数生成：

```
function transform(
    Observation memory last,
    uint32 blockTimestamp,
    int24 tick,
    uint128 liquidity
) private pure returns (Observation memory) {
    // 上次 Oracle 数据和本次的时间差
    uint32 delta = blockTimestamp - last.blockTimestamp;
    return
        Observation({
            blockTimestamp: blockTimestamp,
            // 计算 tick index 的时间加权累积值
            tickCumulative: last.tickCumulative + int56(tick) * delta,
            // 计算时间加权累积值
            liquidityCumulative: last.liquidityCumulative + uint160(liquidity) * delta,
            initialized: true
        });
}
```

这样就完成了一次 Oracle 数据的写入。

Oracle 的使用

第三方开发者要使用 Oracle，调用交易池的 `UniswapV3Pool.observe()` 函数即可：

```
function observe(uint32[] calldata secondsAgos)
    external
    view
    override
    noDelegateCall
    returns (int56[] memory tickCumulatives, uint160[] memory liquidityCumulatives)
{
    return
        observations.observe(
            _blockTimestamp(),
            secondsAgos,
            slot0.tick,
            slot0.observationIndex,
            liquidity,
            slot0.observationCardinality
        );
}
```

传入的参数 `secondsAgos` 是一个动态数组，顾名思义表示请求 N 秒前的数据，使用数组可以一次请求多个历史数据。返回的 `tickCumulatives` 和 `liquidityCumulatives` 也是动态数组，记录了请求参数中对应时间戳的 tick index 累积值和流动性累积值。数据的处理是在 `observations.observe()` 中完成的：

```
function observe(
    Observation[65535] storage self,
    uint32 time,
    uint32[] memory secondsAgos,
    int24 tick,
    uint16 index,
    uint128 liquidity,
    uint16 cardinality
) internal view returns (int56[] memory tickCumulatives, uint160[] memory liquidityCumulatives) {
    require(cardinality > 0, 'I');

    tickCumulatives = new int56[](secondsAgos.length);
    liquidityCumulatives = new uint160[](secondsAgos.length);
    // 遍历传入的时间参数，获取结果
    for (uint256 i = 0; i < secondsAgos.length; i++) {
        (tickCumulatives[i], liquidityCumulatives[i]) = observeSingle(
            self,
            time,
            secondsAgos[i],
            tick,
            index,
            liquidity,
            cardinality
        );
    }
}
```

这个函数就是通过遍历请求参数，获取每一个请求时间点的 Oracle 数据，具体数据通过 `observeSingle()` 函数来获取：

```
function observeSingle(
    Observation[65535] storage self,
    uint32 time,
    uint32 secondsAgo,
    int24 tick,
    uint16 index,
    uint128 liquidity,
    uint16 cardinality
) private view returns (int56 tickCumulative, uint160 liquidityCumulative) {
    // secondsAgo 为 0 表示当前的最新 Oracle 数据
    if (secondsAgo == 0) {
        Observation memory last = self[index];
        if (last.blockTimestamp != time) last = transform(last, time, tick, liquidity);
        return (last.tickCumulative, last.liquidityCumulative);
    }

    // 计算出请求的时间戳
    uint32 target = time - secondsAgo;

    // 计算出请求时间戳最近的两个 Oracle 数据
    (Observation memory beforeOrAt, Observation memory atOrAfter) =
        getSurroundingObservations(self, time, target, tick, index, liquidity, cardinality);

    Oracle.Observation memory at;
    // 如果请求时间和返回的左侧时间戳吻合，那么可以直接使用
    if (target == beforeOrAt.blockTimestamp) {
        at = beforeOrAt;
    // 如果请求时间和返回的右侧时间戳吻合，那么可以直接使用
    } else if (target == atOrAfter.blockTimestamp) {
        at = atOrAfter;
    } else {
        // 当请求的时间在中间时，计算根据增长率计算出请求的时间点的 Oracle 值并返回
        uint32 delta = atOrAfter.blockTimestamp - beforeOrAt.blockTimestamp;
        int24 tickDerived = int24((atOrAfter.tickCumulative - beforeOrAt.tickCumulative) / delta)
        uint128 liquidityDerived =
            uint128((atOrAfter.liquidityCumulative - beforeOrAt.liquidityCumulative) / delta);
        at = transform(beforeOrAt, target, tickDerived, liquidityDerived);
    }

    return (at.tickCumulative, at.liquidityCumulative);
}
```

在这函数中，会先调用 `getSurroundingObservations()` 找出的时间点前后，最近的两个 Oracle 数据。然后通过时间差的比较计算出需要返回的数据：

- 如果和其中的某一个的时间戳相等，那么可以直接返回
- 如果在两个时间点的中间，那么通过计算增长率的方式，计算出请求时间点的 Oracle 数据并返回

`getSurroundingObservations()` 函数的作用是在已记录的 Oracle 数组中，找到时间戳离其最近的两个 Oracle 数据：

```
function getSurroundingObservations(
    Observation[65535] storage self,
    uint32 time,
    uint32 target,
    int24 tick,
    uint16 index,
    uint128 liquidity,
    uint16 cardinality
) private view returns (Observation memory beforeOrAt, Observation memory atOrAfter) {
    // 先把 beforeOrAt 设置为当前最新数据
    beforeOrAt = self[index];

    // 检查 beforeOrAt 是否 <= target
    if (lte(time, beforeOrAt.blockTimestamp, target)) {
        if (beforeOrAt.blockTimestamp == target) {
            // 如果时间戳相等，那么可以忽略 atOrAfter 直接返回
            return (beforeOrAt, atOrAfter);
        } else {
            // 当前区块中发生代币对的交易之前请求此函数时可能会发生这种情况
            // 需要将当前还未持久化的数据，封装成一个 Oracle 数据返回，
            return (beforeOrAt, transform(beforeOrAt, target, tick, liquidity));
        }
    }

    // 将 beforeOrAt 调整至 Oracle 数组中最老的数据
    // 即为当前 index 的下一个数据，或者 index 为 0 的数据
    beforeOrAt = self[(index + 1) % cardinality];
    if (!beforeOrAt.initialized) beforeOrAt = self[0];

    // ensure that the target is chronologically at or after the oldest observation
    require(lte(time, beforeOrAt.blockTimestamp, target), 'OLD');

    // 然后通过二分查找的方式找到离目标时间点最近的前后两个 Oracle 数据
    return binarySearch(self, time, target, index, cardinality);
}
```

这个函数会调用 `binarySearch()` 通过二分查找的方式，找到目标离目标时间点最近的前后两个 Oracle 数据，其中的具体实现这里就不再描述了。

最终，`UniswapV3Pool.observe()` 将会返回请求者所请求的每一个时间点的 Oracle 数据，请求者可以根据这些数据计算出交易对的 TWAP（时间加权平均价，几何平均数），计算公式在前文。

同时因为 Oracle 数据中还包含了流动性的时间累积值，还可以计算出交易池在一段时间内的 TWAL（时间加权平均流动性，算是平均数）。

Oracle 数组扩容

之前说过，虽然合约定义了 Oracle 使用 65535 长度的数组，但是并不会在一开始就使用这么多的空间，这样做是因为：

- 向空数组中写入 Oracle 数据是比较昂贵的操作（SSTORE）
- 写入 Oracle 数据的操作发生在交易的操作中
- 这些操作如果由交易者负担，是不公平的，因为代币交易者并不一定是 Oracle 的使用者

因此 Uniswap v3 在初始时 Oracle 数组仅可以写入一个数据，这个是通过交易池合约的 `slot0.observationCardinalityNext` 变量控制的。

当初始设置不满足需求时，合约提供了单独接口，让对 Oracle 历史数据有需求的开发者，自行调用接口来扩展交易池 Oracle 中存储数据的上限。这样就将 Oracle 数组存储空间初始化操作的 gas 费转移到了 Oracle 的需求方，而不是由代币交易者承担。

通过 `increaseObservationCardinalityNext()` 可以扩展交易池的 Oracle 数组可用容量，传入的参数为期望存储的历史数据个数。

```
function increaseObservationCardinalityNext(uint16 observationCardinalityNext)
    external
    override
    lock
    noDelegateCall
{
    uint16 observationCardinalityNextOld = slot0.observationCardinalityNext; // for the event
    uint16 observationCardinalityNextNew =
        observations.grow(observationCardinalityNextOld, observationCardinalityNext);
    slot0.observationCardinalityNext = observationCardinalityNextNew;
    if (observationCardinalityNextOld != observationCardinalityNextNew)
        emit IncreaseObservationCardinalityNext(observationCardinalityNextOld, observationCardina
}
```

这个函数调用了 `observations.grow()` 完成底层存储空间的初始化：

```
function grow(
    Observation[65535] storage self,
    uint16 current,
    uint16 next
) internal returns (uint16) {
    require(current > 0, 'I');
    // no-op if the passed next value isn't greater than the current next value
    if (next <= current) return current;
    // 对数组中将来可能会用到的槽位进行写入，以初始化其空间，避免在 swap 中初始化
    for (uint16 i = current; i < next; i++) self[i].blockTimestamp = 1;
    return next;
}
```

这里可以看到，通过循环的方式，将 Oracle 数组中未来可能被使用的空间中写入数据。这样做的目的是将数据进行初始化，这样在代币交易写入新的 Oracle 数据时，不需要再进行初始化，可以让交易时更新 Oracle 不至于花费太多的 gas，`SSTORE` 指令由 20000 降至 5000。可以参考：EIP-1087, EIP-2200, EIP-2929，具体实现：core/vm/gas_table.go。

Oracle 数据容量

当 Oracle 数据可使用空间被扩容至最大，即 65535 时，假设平均出块时间为 13 秒，那么此时至少可以存储最近 9.8 天的历史数据。

至此，关于 Uniswap v3 Oracle 的所有操作就介绍完毕了。

Uniswap v3 详解系列

本系列所有文章：

- Uniswap v3 详解（一）：设计原理
- Uniswap v3 详解（二）：创建交易对/提供流动性
- Uniswap v3 详解（三）：交易过程
- Uniswap v3 详解（四）：交易手续费
- Uniswap v3 详解（五）：Oracle 预言机
- Uniswap v3 详解（六）：闪电贷

PREVIOUS
UNISWAP V3 详解（四）：交易手续费

NEXT
UNISWAP V3 详解（六）：闪电贷

Rowing Saylor
3 个月前 edited
That smack in the beginning though 😂😂😂
GS.125365585.SPACE\v5977x
43 0 回复 • 分享 ›

Paco 管理员
2 年前
关于 oracle 使用中典型的一个谬误：


这里面有几个问题要考虑：
1. 首先计算出的 P 的值表示的是 token0/token1，即 token1 的价格，如果需要求 token0 的价格，则为 1/P
2. 每个 erc20 token 的 decimal 数是不一样的，对应你所说的 ETH/USDC 交易对中，WETH decimal 为 18，而 USDC decimal 为 6，那么计算的正确步骤应该是：
P(eth) = 1/(1.0001^198017) * 10^(18-6) = 2515.71197754
和实际情况相符。
关于 oracle 的使用，建议封装/调用官方的 library: https://github.com/Uniswap/uniswap-v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol
On Jun 10, 2021, at 17:46,
您好，看了您的https://paco0x.com/uniswap-v3-5/ 文章。
想请教v3在当作预言机对外提供价格时，调用observe函数，
比如输入5，10，表示5s到10s的价格平均值，
获取到两个对应时间戳的tick累积值
617833432231
617832442146
相减然后/5
990085/5=198017
然后1.0001^198017得出结果
计算得到397501784.358472175175487
以eth-usdc交易对为例
这个结果和实际价格不同，您能讲一下吗
感谢

0 0 回复 • 分享 ›

FEATURED TAGS
Life Basic Performance Tuning Operating System OpenStack Python Web Solidity Ethereum Uniswap DeFi