**UniswapV3Ink-产品软件总体设计**

1. 2022.03.02 V1.0 WeiYong create by wangdong

# 概述及需求

在substrate链上创建一个基础的金融swap服务.该服务主要提供以下几点服务:

* 1. 该服务可以使普通用户作为流动性提供者提供资金流动性.并且流动性具有较高的资金使用率.
  2. 提供抽离流动性服务.可以部分抽取流动性.
  3. 提供的swap功能和当前价格的查询服务.根据用户提供的交易量和交易价格swap出不低于最小量的token.
  4. 提供交易的路由服务.提供一种token swap另一种token的交易路由.
  5. 提供最新价格服务.

# 系统结构

uniswapV3ink 的系统结构如下图所示：

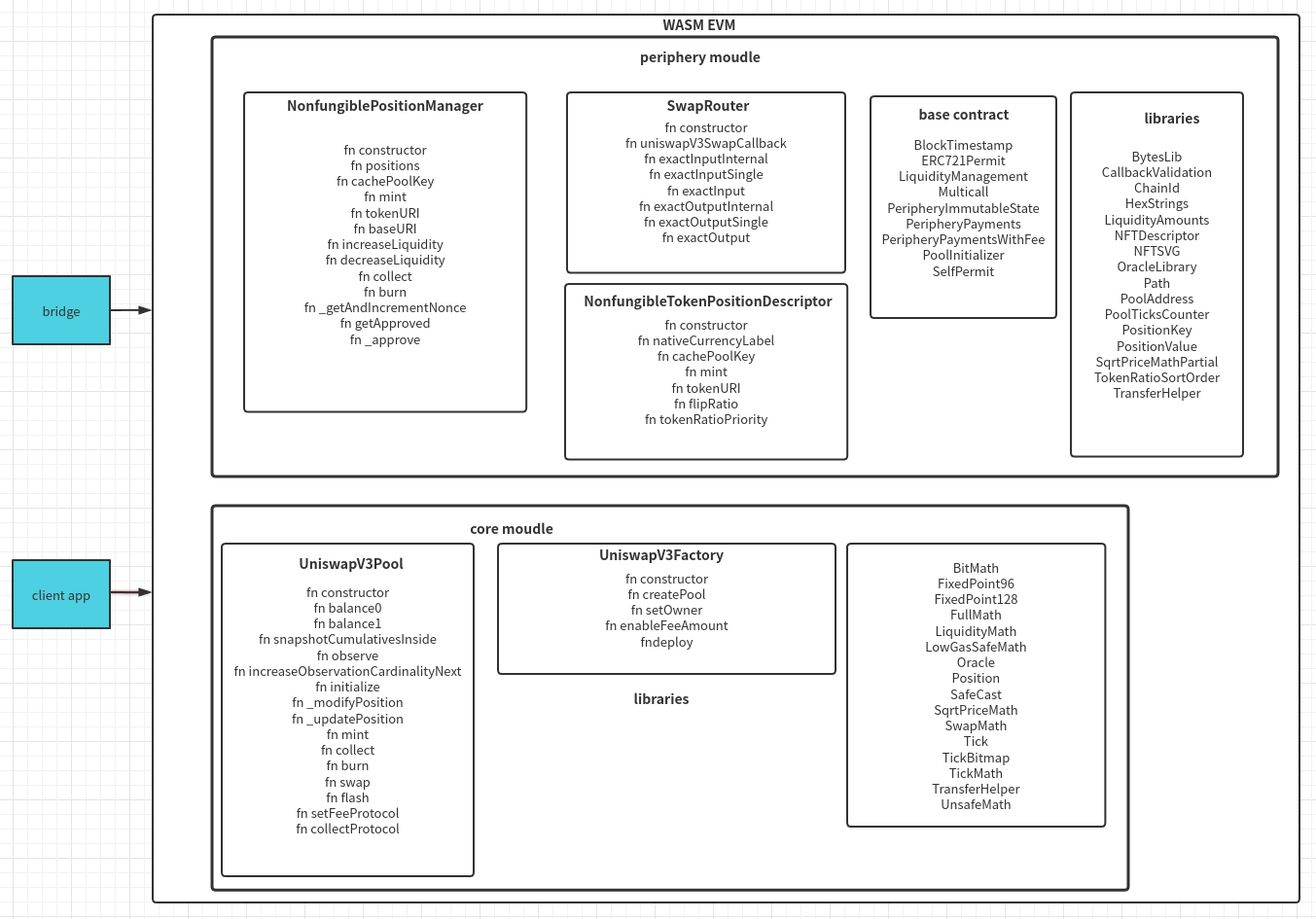


图1. uniswapV3ink系统结构

# 系统模块和功能

功能的具体流程图和计算公式见uniswap v3 白皮书,网址:https://www.jinse.com/news/blockchain/1057182.html

## 代码架构主要分为两个大的集合:

**core moudle**

**periphery moudle**

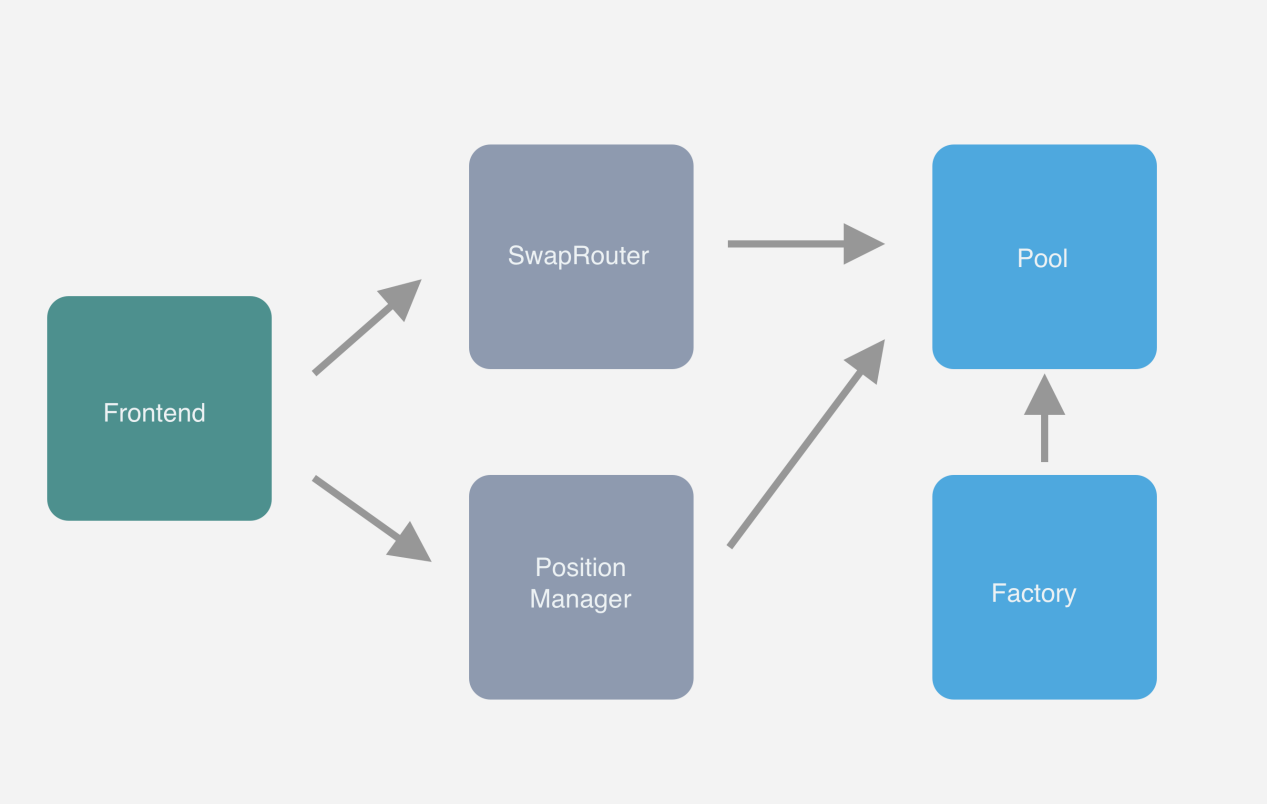
core 仓库的功能主要包含在以下 2 个合约中：

* **UniswapV3Factory**: 提供创建 pool 的接口，并且追踪所有的 pool
* **UniswapV3Pool**: 实现代币交易，流动性管理，交易手续费的收取，oracle 数据管理。接口的实现粒度比较低，不适合普通用户使用，错误的调用其中的接口可能会造成经济上的损失。

peirphery 仓库的功能主要包含在以下 2 个合约：

* **SwapRouter**: 提供代币交易的接口，它是对 UniswapV3Pool 合约中交易相关接口的进一步封装，前端界面主要与这个合约来进行对接。
* **NonfungiblePositionManager**: 用来增加/移除/修改 Pool 的流动性，并且通过 NFT token 将流动性代币化。使用 ERC721 token（v2 使用的是 ERC20）的原因是同一个池的多个流动性并不能等价替换（v3 的集中流性动功能）。

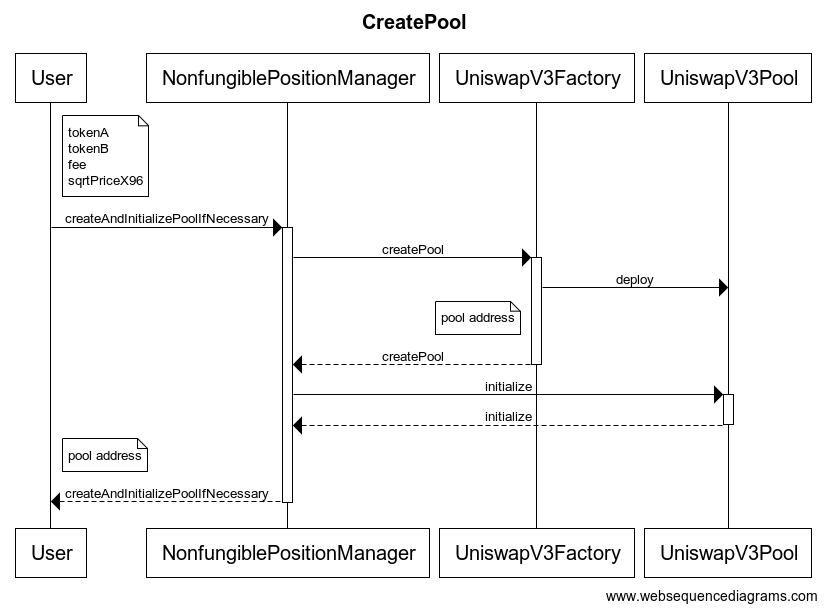
这些合约间的关系大致如下图：

* 

### ****NonfungiblePositionManager**** Module

* + 1. 创建交易对.

创建交易对的调用流程如下：



用户首先调用 NonfungiblePositionManager 合约的 createAndInitializePoolIfNecessary 方法创建交易对，传入的参数为交易对的 token0, token1, fee 和初始价格 .

NonfungiblePositionManager 合约内部通过调用 UniswapV3Factory 的 createPool 方法完成交易对的创建，然后对交易对进行初始化，初始化的作用就是给交易对设置一个初始的价格。

createAndInitializePoolIfNecessary 如下：

首先调用 UniswapV3Factory.getPool 方法查看交易对是否已经创建，使用 (tokenA, tokenB, fee)的 元组来作为一个交易对的键，即相同代币，不同费率之间的流动池不一样。另外对于给定的 tokenA 和 tokenB，会先将其地址排序，将地址值更小的放在前，这样方便后续交易池的查询和计算。

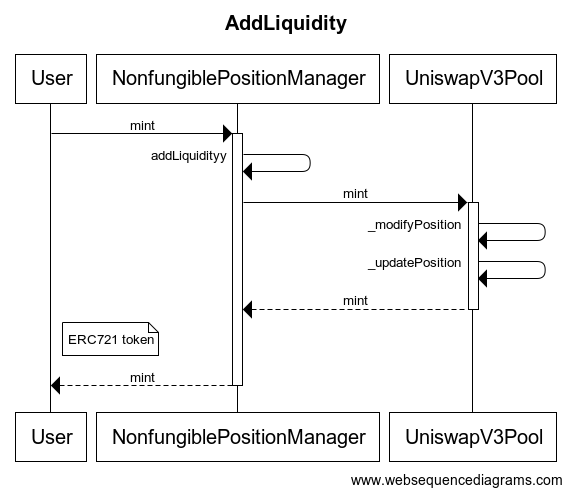
原有的solidity采用固定的initcode和salt方法可以直接线下算出来pool的地址.ink!使用codehash和salt也可以算出固定的pool的地址.再将pool存放到pool\_map中,其中的key为(token0,token1,fee),value为pool的地址.ink!中暂时不知道pool地址的算法,故需要获取pool的时候,仍然从factory通过(token0,token1,fee)获取pool的地址.

最后，对创建的交易对合约进行初始化：

初始化主要是设置了交易池的初始价格（注意，此时池子中还没有流动性），以及费率，tick 等相关变量的初始化。完成之后一个交易池就创建好了。

3.1.2 提供流动性

在合约内保存所有用户的流动性，代码内称作 Position，提供流动性的调用流程如下：



用户还是首先和 NonfungiblePositionManager 合约交互。v3 这次将 LP token 改成了 ERC721 token，并且将 token 功能放到 NonfungiblePositionManager 合约中。这个合约替代用户完成提供流动性操作，然后根据将流动性的数据元记录下来，并给用户铸造一个 NFT Token.

这里有几点值得注意：

传入的 lower/upper 价格是以 tick index 来表示的，因此需要在链下先计算好价格所对应的 tick index.

传入的是流动性 L 的大小，这个也需要在链下先计算好，计算过程见下面

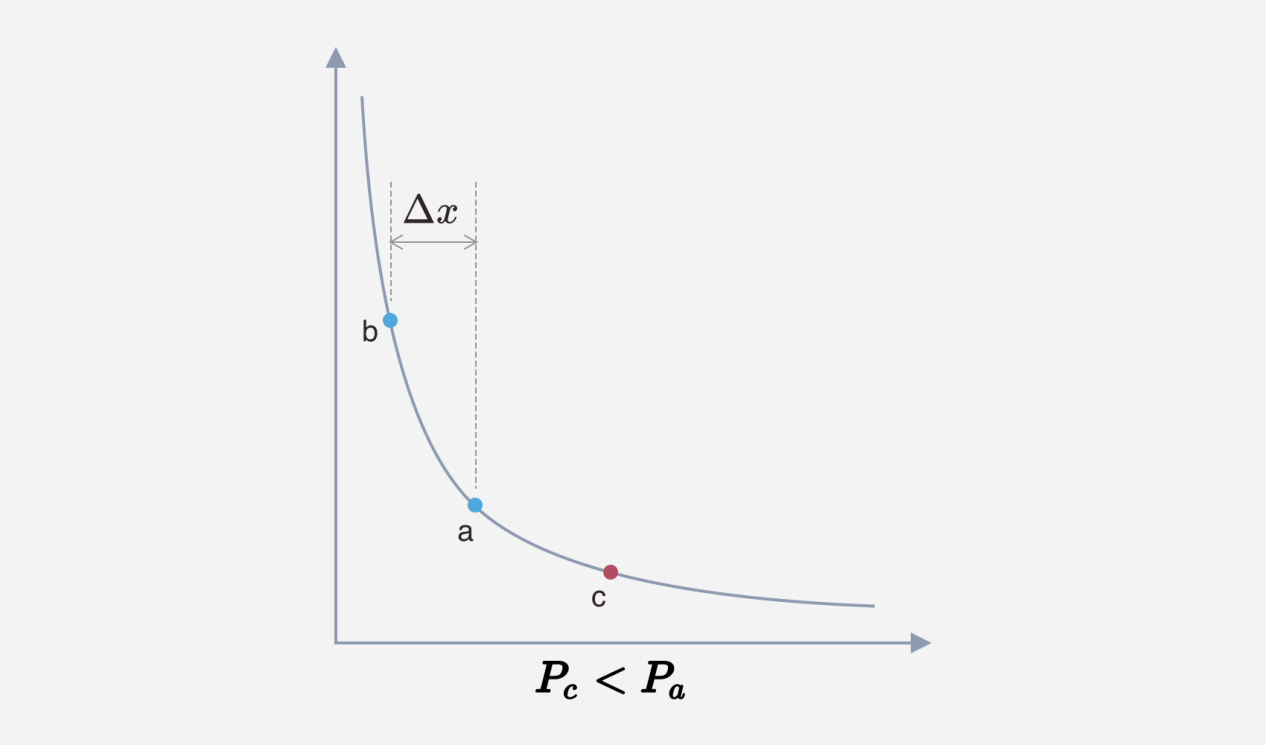
这里有一个回调函数的参数。v3 使用回调函数来完成进行流动性 token 的支付操作，原因见下面

### **从 token 数计算流动性 L**

如前所述，因为合约的参数接受的是流动性 L 的值，我们需要在链下通过用户愿意提供流动性包含的 token 数，计算出 L.

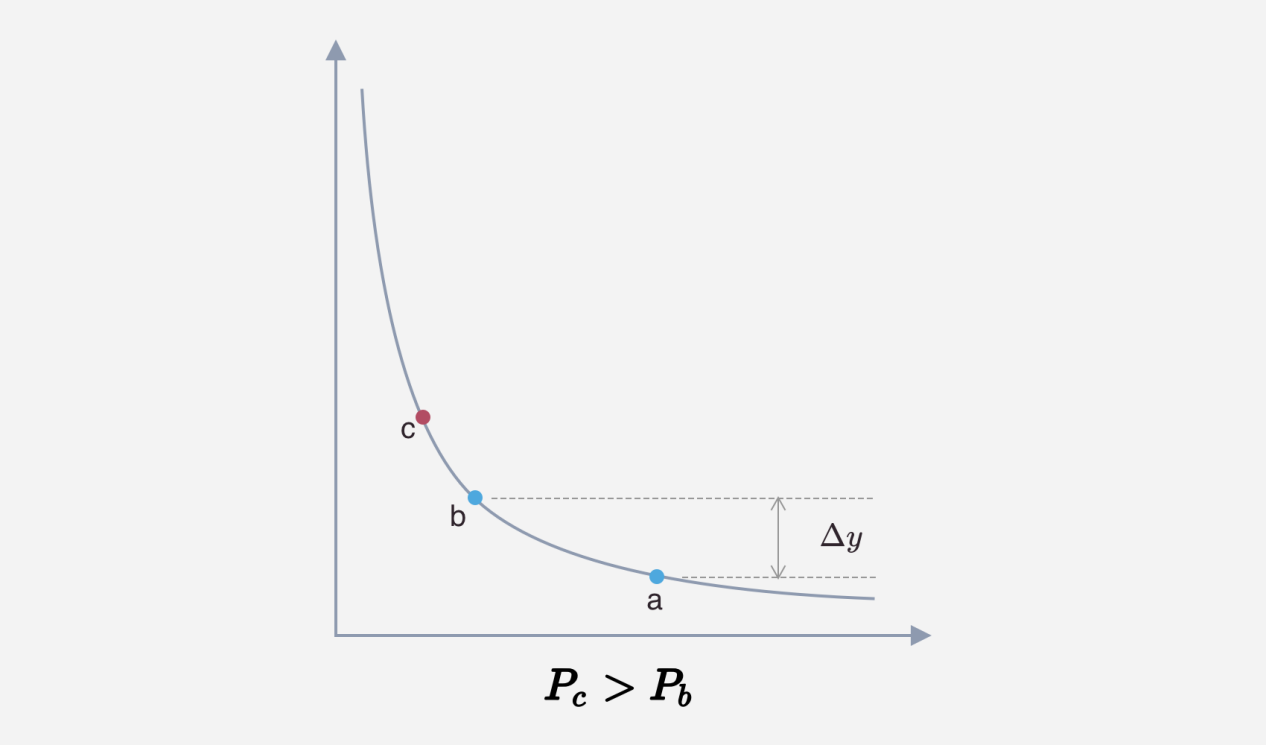
假设用户提供流动性的价格范围是：[Pa, Pb] (Pa<Pb)，代币池中的当前价格为 Pc，可以分成三种情况来计算流动性 L的值：

当前池中的价格 Pc<Pa,如下图：

* 

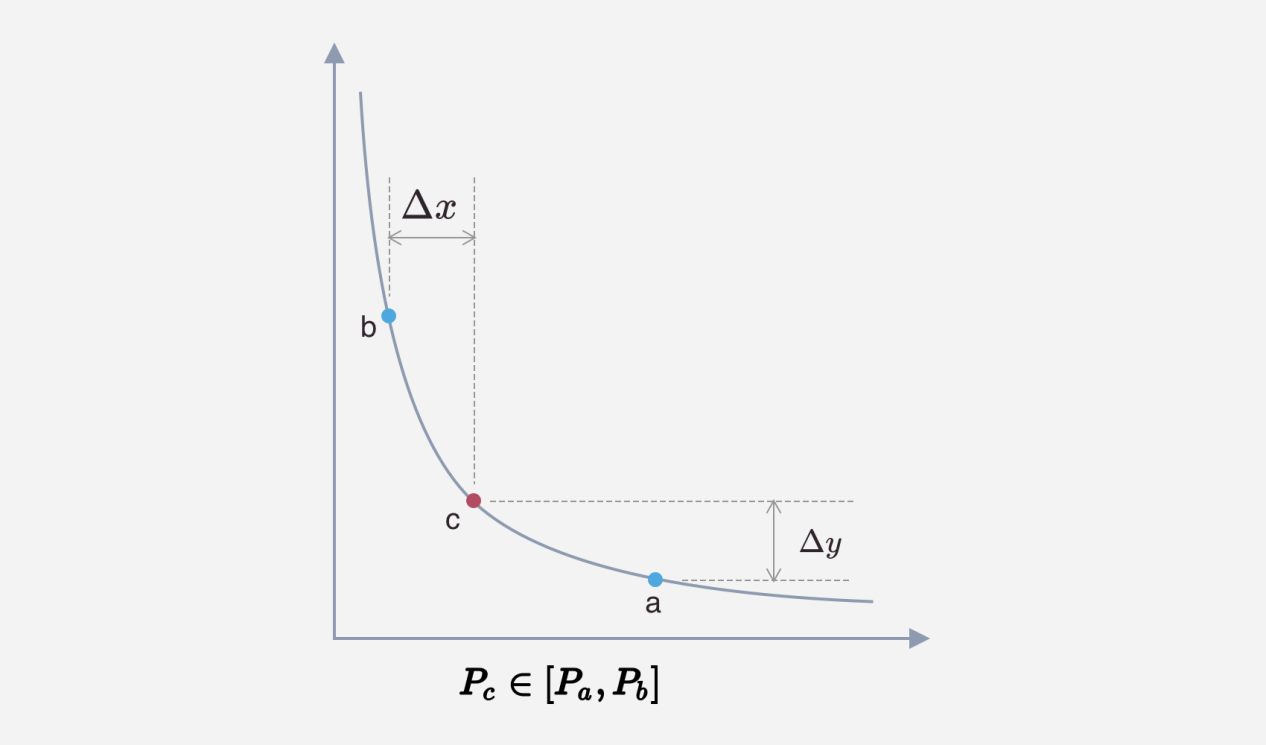
此时添加的流动性全部为 x token，计算 L:

当前池中的价格 Pc>Pb



此时添加的流动性全部为 y token，计算 L：

当前池子中的价格 Pc∈[Pa,Pb]，如下图：



此时添加的流动性包含两个币种，可以通过任意一个 token 数量计算出 L:

### **回调函数**

使用回调函数原因是，将 Position 的 owner 和实际流动性 token 支付者解耦。这样可以让中间合约来管理用户的流动性，并将流动性 token 化。关于 token 化，Uniswap v3 默认实现了 ERC721 token（因为即使是同一个池子，流动性之间差异也也很大）。

例如，当用户通过 NonfungiblePositionManager 来提供流动性时，对于 UniswapV3Pool 合约来说，这个 Position 的 owner 是 NonfungiblePositionManager，而 NonfungiblePositionManager 再通过 NFT Token 将 Position 与用户关联起来。这样用户就可以将 LP token 进行转账或者抵押类操作。

在 NonfungiblePositionManager 中回调函数的实现了转账操作.

### **postion 更新**

接着我们看 UniswapV3Pool 是如何添加流动性的。流动性的添加主要在 UniswapV3Pool.\_modifyPosition 中，这个函会先调用 \_updatePosition 来创建或修改一个用户的 Position.

先忽略费率相关的操作，这个函数所做的操作是：

* 添加/移除流动性时，先更新这个 Positon 对应的 lower/upper tick 中记录的元数据
* 更新 position
* 根据需要更新 tick 位图

Postion 是以 owner, lower tick, uppper tick 作为键来存储的，注意这里的 owner 实际上是 NonfungiblePositionManager 合约的地址。这样当多个用户在同一个价格区间提供流动性时，在底层的 UniswapV3Pool 合约中会将他们合并存储。而在 NonfungiblePositionManager 合约中会按用户来区别每个用户拥有的 Position.

Postion 中包含的字段中，除去费率相关的字段，只有一个即流动性 L：

### **tick 管理**

我们再来看 tick 相关的管理，在 UniswapV3Pool 合约中有两个状态变量记录了 tick 相关的信息：

*// 记录了一个 tick 包含的元数据，这里只会包含所有 Position 的 lower/upper ticks* **mapping**(**int24** **=>** Tick.Info) **public** **override** ticks;

*// tick 位图，因为这个位图比较长（一共有 887272x2 个位），大部分的位不需要初始化*

*// 因此分成两级来管理，每 256 位为一个单位，一个单位称为一个 word*

*// map 中的键是 word 的索引*

**mapping**(**int16** **=>** **uint256**) **public** **override** tickBitmap;

tick 中和流动性相关的字段有两个 liquidityGross，liquidityNet。

liquidityNet 表示当价格从左至右经过此 tick 时整体流动性需要变化的净值。在单个流动性中，对于 lower tick 来说，它的值为正，对于 upper tick 来说它的值为 负。

如果有两个 position 中的流动性相等，例如 L = 500，并且这两个 position 同时引用了一个 tick，其中一个为 lower tick ，另一个为 upper tick，那么对于这个 tick，它的 liquidityNet = 0。此时我们就需要有一种机制来判断一个 tick 是否仍然在被引用中。这里使用 liquidityGross 记录流动性的增值（不考虑 lower/upper），我们可以就通过流动性变化前后 liquidityGross 是否等于 0 来判断这个 tick 是否仍被引用。

当价格变动导致 tickcurrent 越过一个 position 的 lower/upper tick 时，我们需要根据 tick 中记录的值来更新当前价格所对应的总体流动性。假设 position 的流动性值为 ΔL，会有以下四种情况：

1. token0 价格上升，即从左至右越过一个 lower tick 时， L=Lcurrent+ΔL
2. token0 价格上升，即从左至右越过一个 upper tick 时， L=Lcurrent−ΔL
3. token0 价格下降，即从右至左越过一个 upper tick 时， L=Lcurrent+ΔL
4. token0 价格下降，即从右至左越过一个 lower tick 时， L=Lcurrent−ΔL

liquidityNet 中记录的就是当从左至右穿过这个 tick 时，需要增减的流动性，当其为 lower tick 时，其值为正，当其为 upper tick 时，其值为负。对于从右至左穿过的情况，只需将 liquidityNet 的值取反即可完成计算。

我再来看如何更新 tick 元数据，以下是 tick.update 函数：

此函数返回的 flipped 表示此 tick 的引用状态是否发生变化，之前的 \_updatePosition 中的代码会根据这个返回值去更新 tick 位图。

### **tick 位图**

tick 位图用于记录所有被引用的 lower/upper tick index，我们可以用过 tick 位图，从当前价格找到下一个（从左至右或者从右至左）被引用的 tick index。关于 tick 位图的管理，在 \_updatePosition 中的：

这里不做进一步的说明，具体代码实现在[TickBitmap库](https://github.com/Uniswap/uniswap-v3-core/blob/2dc1eb9f251bad1c260d22dd392d8cedb2c6a4b5/contracts/libraries/TickBitmap.sol)中。tick 位图有以下几个特性：

* 对于不存在的 tick，不需要初始值，因为访问 map 中不存在的 key 默认值就是 0
* 通过对位图的每个 word(uint256) 建立索引来管理位图，即访问路径为 word index -> word -> tick bit

### **token 数确认**

\_modifyPosition 函数在调用 \_updatePosition 更新完 Position 后，会计算出此次提供流动性具体所需的 x token 和 y token 数量。

这里插入一个题外话，这一行代码：

Slot0 **memory** \_slot0 **=** slot0; *// SLOAD for gas optimization*

因为后续需要多次访问 slot0，这里将其读入内存中，后续的访问就可以使用 MLOAD 而不用使用 SLOAD，可以节省 gas（****SLOAD 的成本比 MLOAD 高很多****）。Uniswap v2 和 v3 大量使用了这个技巧。

这个函数在更新完 position 之后，主要做的就是通过 L 和  计算出用户需要支付的 token 数量，我们之前已经讲过 [从 token 数计算流动性 L](https://liaoph.com/uniswap-v3-1/" \l "%E4%BB%8E-token-%E6%95%B0%E8%AE%A1%E7%AE%97%E6%B5%81%E5%8A%A8%E6%80%A7-l)的三种情况，这里其实就是之前计算的逆运算，即通过 L 计算 x token 和 y token 的数量.

代码将计算的过程封装在了 SqrtPriceMath 库中，getAmount0Delta 和 getAmount1Delta 分别对应公式 和

在具体的计算过程中，又分成了 RoundUp 和 RoundDown 两种情况，简单来说：

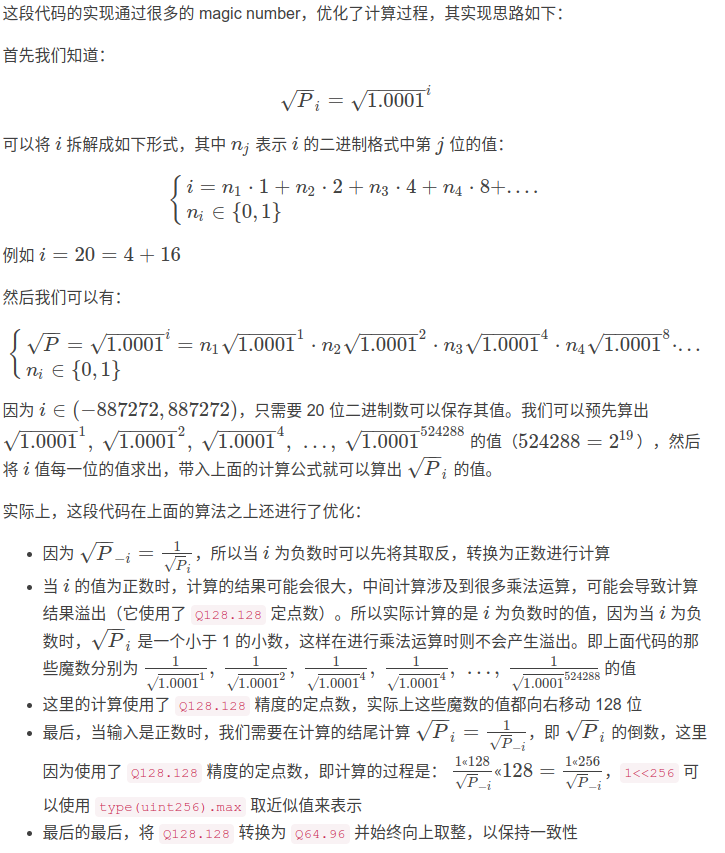
1. 当提供/增加流动性时，会使用 RoundUp，这样可以保证增加数量为 L 的流动性时，用户提供足够的 token 到 pool 中
2. 当移除/减少流动性时，会使用 RoundDown，这样可以保证减少数量为 L 的流动性时，不会从 pool 中给用户多余的 token

通过上述两个条件可以保证 pool 在流动性增加/移除的操作中，不会出现坏账的情况。除了流动性操作之外，swap 操作也会使用类似机制，保证 pool 不会出现坏账。

同时，Uniswap v3 参考[这里](https://xn--2-umb.com/21/muldiv/index.html)实现了一个精度较高的 的算法，封装在 FullMath 库中。

### **tick index ->**

上面的代码还使用了 TickMath 库中的 getSqrtRatioAtTick 来通过 tick index 计算其所对应的价格，实现为：



### **-> tick index**

这里顺带提一下，在交易计算中会需要进行上述计算的逆计算，给定 P−−√P，需要计算出对应的 tick index，即 log1.0001√P−−√log1.0001P 的计算。在代码中为：TickMath.getTickAtSqrtRatio.

### **完成流动性添加**

\_modifyPosition 调用完成后，会返回 x token, 和 y token 的数量。再来看 UniswapV3Pool.mint 的代码：

这个函数关键的步骤就是通过回调函数，让调用方发送指定数量的 x token 和 y token 至合约中。

我们再来看 NonfungiblePositionManager.mint 的代码：

可以看到这个函数主要是将用户的 Position 保存起来，并给用户铸造 NFT token，代表其所持有的流动性。至此提供流动性的步骤就完成了。

* + 1. **流动性的移除**.

移除流动性就是上述操作的逆操作，在 core 合约中：

移除流动性时，还是使用之前的公式计算出移出的 token 数，但是并不会直接将移出的 token 数发送给用户，而是记录在了 position 的 tokensOwed0 和 tokensOwed1 上。这样做应该是为了遵循实践：[Favor pull over push for external calls](https://consensys.github.io/smart-contract-best-practices/recommendations/" \l "favor-pull-over-push-for-external-calls).

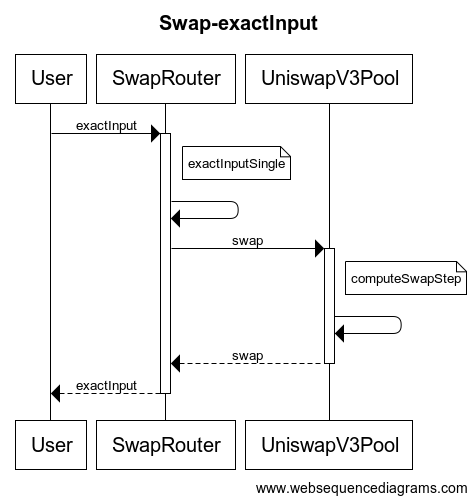
### Router Module

* + 1. **交易过程**

v3 的 UniswapV3Pool 提供了比较底层的交易接口，而在 SwapRouter 合约中封装了面向用户的交易接口：

* exactInput：指定交易对路径，付出的 x token 数和预期得到的最小 y token 数（x, y 可以互换）
* exactOutput：指定交易路径，付出的 x token 最大数和预期得到的 y token 数（x, y 可以互换）

这里我们讲解 exactInput 这个接口，调用流程如下：



### **路径选择**

在进行两个代币交易时，是首先需要在链下计算出交易的路径，例如使用 ETH -> DAI ：

* 可以直接通过 ETH/DAI 的交易池完成
* 也可以通过 ETH -> USDC -> DAI 路径，即经过 ETH/USDC, USDC/DAI 两个交易池完成交易

Uniswap 的前端会帮用户实时计算出最优路径（即交易的收益最高），作为参数传给合约调用。前端中这部分计算的具体实现在[这里](https://github.com/Uniswap/uniswap-interface/blob/3aa045303a4aeefe4067688e3916ecf36b2f7f75/src/hooks/useBestV3Trade.ts" \l "L17-L96)，具体过程为先用需要交易的输入代币，输出代币，以及一系列可用的中间代币（代码中叫 Base token）生成所有的路径（当然为了降低复杂度，路径中最多包含3个代币），然后遍历每个路径输出的输出代币数量，最后选取最佳路径。

事实上因为 v3 引入了费率的原因，在路径选择的过程中还需要考虑费率的因素。

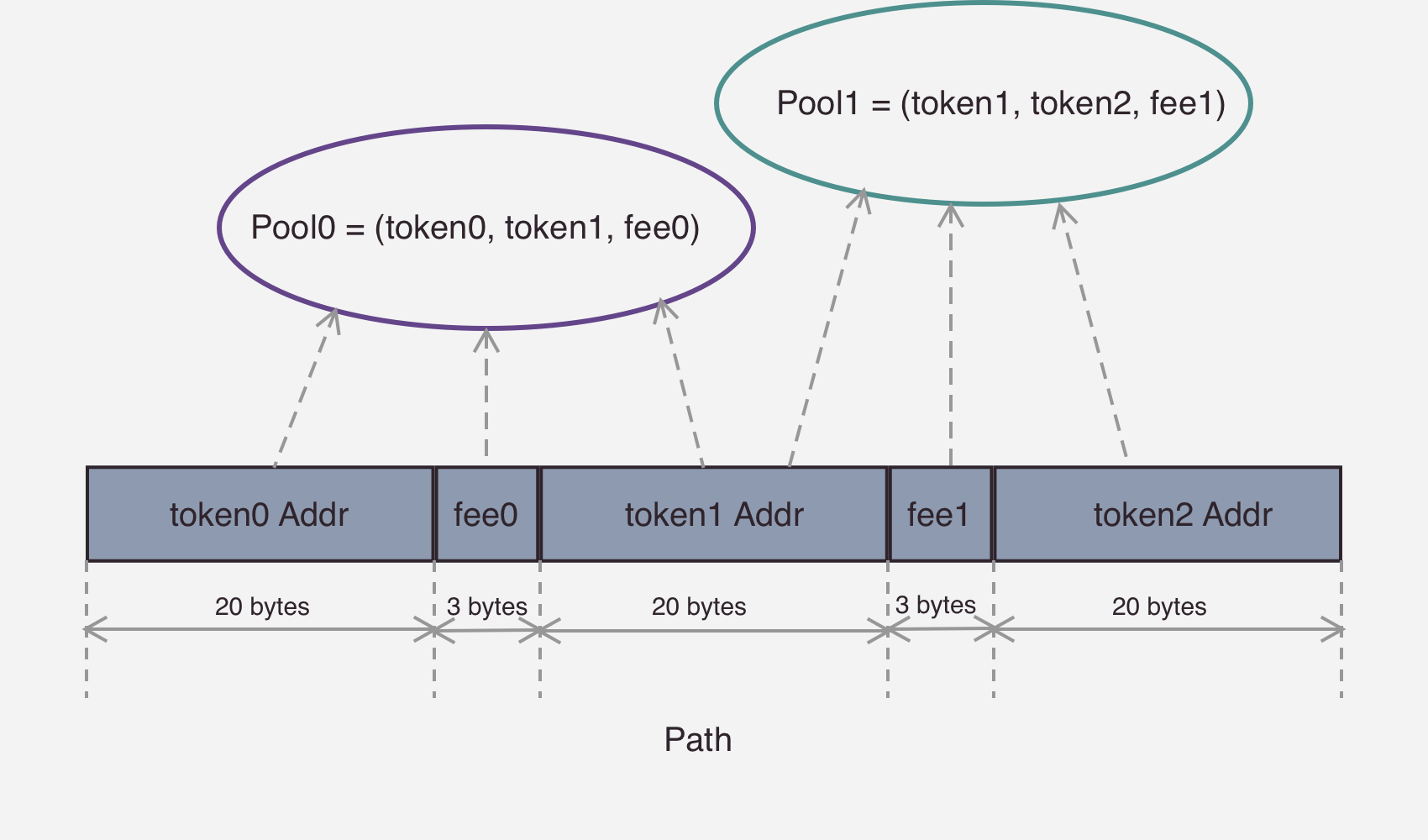
### **交易入口**

交易的入口函数是 exactInput 函数，代码如下：

这里使用一个循环遍历传入的路径，路径中包含了交易过程中所有的 token，每相邻的两个 token 组成了一个交易对。例如当需要通过 ETH -> USDC -> DAI 路径进行交易时，会经过两个池：ETH/USDC 和 USDC/DAI，最终得到 DAI 代币。如前所述，这里其实还包含了每个交易对所选择的费率。

### **路径编码/解码**

上面输入的参数中 path 字段是 bytes 类型，通过这种类型可以实现更紧凑的编码。Uniswap 会将 bytes 作为一个数组使用，bytes 类型就是一连串的 byte1，但是不会对每一个成员使用一个 word，因此相比普通数组其结构更加紧凑。在 Uniswap V3 中， path 内部编码结构如下图：



图中展示了一个包含 2个路径（pool0, 和 pool1）的 path 编码。Uniswap 将编码解码操作封装在了 Path 库中，本文不再赘述其过程。每次交易时，会取出头部的 tokenIn, tokenOut, fee，使用这三个参数找到对应的交易池，完成交易。

### **单个池的交易过程**

单个池的交易在 exactInputSingle 函数中：

交易过程就是先获取交易池，然后需要确定本次交易输入的是交易池的 x token, 还是 y token，这是因为交易池中只保存了 x 的价格 ，x token 和 y token 的计价公式是不一样的。最后调用 UniswapV3Pool 的 swap 函数完成交易。

### **交易分解**

UniswapV3Pool.swap 函数比较长，这里先简要描述其交易步骤：

假设支付的 token 为 x

1. 根据买入/卖出行为， 会随着交易下降或上升，即 tick 减小或增大
2. 在 tickBitmap 中找到和当前 tick 对应的 ic 在一个 word 中的下一个 tick 对应的 in，根据买入/卖出行为，这里分成向下查找和向上查找两种情况
3. 如果当前 word 中没有记录其他 tick index ，那么取这个 word 的最小/最大 tick index，这么做的目的是，让单步交易中 tick 的跨度不至于太大，以减少计算中溢出的可能性（计算中会需要使用 ）。
4. 在 [ic,in] 价格区间内，流动性 L 的值是不变的，我们可以根据 L 的值计算出交易运行到 in时，所需要最多的 Δx数量
5. 根据上一步计算的 Δx 数量，如果满足 Δx<xremaining，那么将 i 设置为 in，并将 xremaining减去需要支付的 Δx，随后跳至第 2 步继续计算（这里需要将 i±tickSpace 使其进入位图中的下一个 word），计算之前还需要根据元数据修改当前的流动性 L=L±ΔL
6. 如果上一步计算 Δx，满足 Δx≥xremaining，则表示 x token 将被耗尽，则交易在此结束。
7. 记录下结束时的价格 ，将所有交易阶段的 tokenOut 数量总和返回，即为用户得到的 token 数量.
8. 上一步的计算过程还需要考虑费率的因素，为了让计算简单化，可能会多收费

我们逐步拆解 swap 函数中的代码：

上面的代码即交易的主循环，实现思路即以一个 tickBitmap 的 word 为最大单位，在此单位内计算相同流动性区间的交易数值，如果交易没有完成，那么更新流动性的值，进入下一个流动性区间计算，如果 tick index 移动到 word 的边界，那么步进到下一个 word.

关于 tickBitmap 中下一个可用价格 tick index 的查找，在函数 TickBitmap 中实现，这里不做详细描述。

### **拆分后的交易计算**

交易是否能够结束的关键计算在 SwapMath.computeSwapStep 中完成，这里计算了交易是否能在目标价格范围内结束，以及消耗的 tokenIn 和得到的 tokenOut.

函数的输入参数是当前价格，目标价格，当前的流动性，以及 tokenIn 的余额。

这里再次调用了 SqrtPriceMath.getAmount0Delta 或者 SqrtPriceMath.getAmount1Delta 来计算到达目标价是所需的 token 数量。即已知 , , L,求 Δx 和 Δy. 计算的过程在上一章已经讲过了，运用的公式是：

假设交易是输入 x token ，余额为 x（预先扣除最大所需的手续费后的余额，以防止手续费不足），在计算得到 Δx 后，比较：

* 当 x≥Δx 时，表示交易可以到达目标价格
* 当 x<Δx 时，表示交易不足以到达目标价格，此时还需要进一步当前余额 Xremaining 全部耗尽时所能够达到的价格.

如果 x<Δx，我们需要计算 x 耗尽时的价格，即已知 Δx, ,L,求 . 根据：

得出：

具体上述公式计算仅对通过 x token 余额求出下一个价格的公式进行了推导，如果输入的时 y token，也可以额进行类似的推导。代码中具体的实现已经封装在在 SqrtPriceMath.getNextSqrtPriceFromInput 函数中，这里不再进一步详细解释。我们接着看 computeSwapStep 的剩余步骤：

后续的步骤即重新计算了需要支付的手续费用和付出的 tokenIn, tokenOut 数量，这一步的交易就结束了，函数会将手续费，到达的目标价以及 tokenIn, tokenOut 返回。

在进行交易输入/输出的计算时，和流动性的计算一样，也会遇到 rounding 的问题，处理的原则是：

1. 当计算 output 时，使用 RoundDown，保证 pool 不会出现坏账
2. 当计算 input 时，使用 RoundUp，保证 pool 不会出现坏账
3. 当通过 input 计算  时，如果会减少，那么使用 RoundUp，这样可以保证 被 RoundDown，在后续计算 output 时不会使 pool 出现坏账。反之 如果  会增大， 那么使用 RoundDown
4. 当通过 output 计算 时，如果  会减少，那么使用 RoundDown，这样可以保证  被 RoundUp，在后续计算 input 时不会使 pool 出现坏账。反之 如果  会增大， 那么使用 RoundUp.

### **交易收尾阶段**

我们再回到 swap 函数中循环检查条件：

**while** (state.amountSpecifiedRemaining **!=** 0 **&&** state.sqrtPriceX96 **!=** sqrtPriceLimitX96) { ... }

即通过通过 tokenIn 是否还有余额来判断是否还需要继续循环，进入下一步的进行交易计算。当 tokenIn 全部被耗尽后，交易就结束了。当交易结束后，我们还需要做这些事情：

* 更新预言机
* 更新当前交易对的价格，流动性 L
* 更新手续费累计值
* 扣除用户需要支付的 token

这里还是通过回调完成用户支付 token 的费用。因为发送用户 token 是在回调函数之前完成的，因此这个 swap 函数是可以被当作 flash swap 来使用的。

需要注意，如果本次交易是交易路径中的一次中间交易，那么扣除的 token 是从 SwapRouter 中扣除的，交易完成获得的 token 也会发送给 SwapRouter 以便其进行下一步的交易，我们回到 SwapRouter 中的 exactInput 函数：

params.amountIn **=** exactInputSingle( params.amountIn,

*// 这里会判断是否是最后一次交易，当是最后一次交易时，获取的 token 的地址才是用户的指定的地址*

hasPools **?** **address**(this) **:** params.recipient, SwapData({ path**:** params.path.getFirstPool(), payer**:** msg.sender }) );

回调完成后，swap 函数会返回本次交易得到的代币数量。exactInput 将判断是否进行下一个路径的交易，直至所有的交易完成，进行输入约束的检查：

require(amountOut **>=** params.amountOutMinimum, 'Too little received');

如果交易的获得 token 数满足约束，则本次交易结束。

本文仅对 exactInput 这一种交易情况进行了分析，理解了这个交易的整个流程后，就可以触类旁通理解 exactOutput 的交易过程。

### **交易预计算**

当用户和 uniswap 前端进行交互时，前端需要预先计算出用户输入 token 能够预期得到的 token 数量。

这个功能在 uniswap v2 有非常简单的[实现](https://github.com/Uniswap/uniswap-v2-periphery/blob/dda62473e2da448bc9cb8f4514dadda4aeede5f4/contracts/libraries/UniswapV2Library.sol" \l "L42-L59)，只需要查询处合约中两个代币的余额就可以完成预计算。

但是在 v3 版本中，由于交易的计算需要使用合约内的 tick 信息，预计算只能由 uniswap v3 pool 合约来完成，但是 pool 合约中的计算函数都是会更改合约

## **手续费的计算和存储**

每一个流动性的提供者都可以设置独立的价格范围区间，这个被称为 positon. 当我们计算交易的手续费时，我们需要计算如下值：

* 每一个 position 收取的手续费（token0, token1 需要分别单独计算）
* 用户如果提取了手续费，需要记录用户已提取的数值

### **手续费的提取**

手续费的提取也是以 position 为单位进行提取的.

根据 position 中已经记录的手续费和用户请求的数额，发送指定数额的手续费给用户。

但是这里 posiiton 中的手续费可能并不是最新的（上面说过手续费总数只会在 position 的流动性更新时更新）。因此在提取手续费前，需要主动触发一次手续费的更新，这些操作已经在 uniswap-v3-periphery 仓库中进行了封装。

## **Oracle 计算**

* 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.