目录

uniswap

uniswap是一个ERC20 token的交易协议，采用了AMM(Automated market maker)设计。传统市场主要采用订单簿来匹配交易

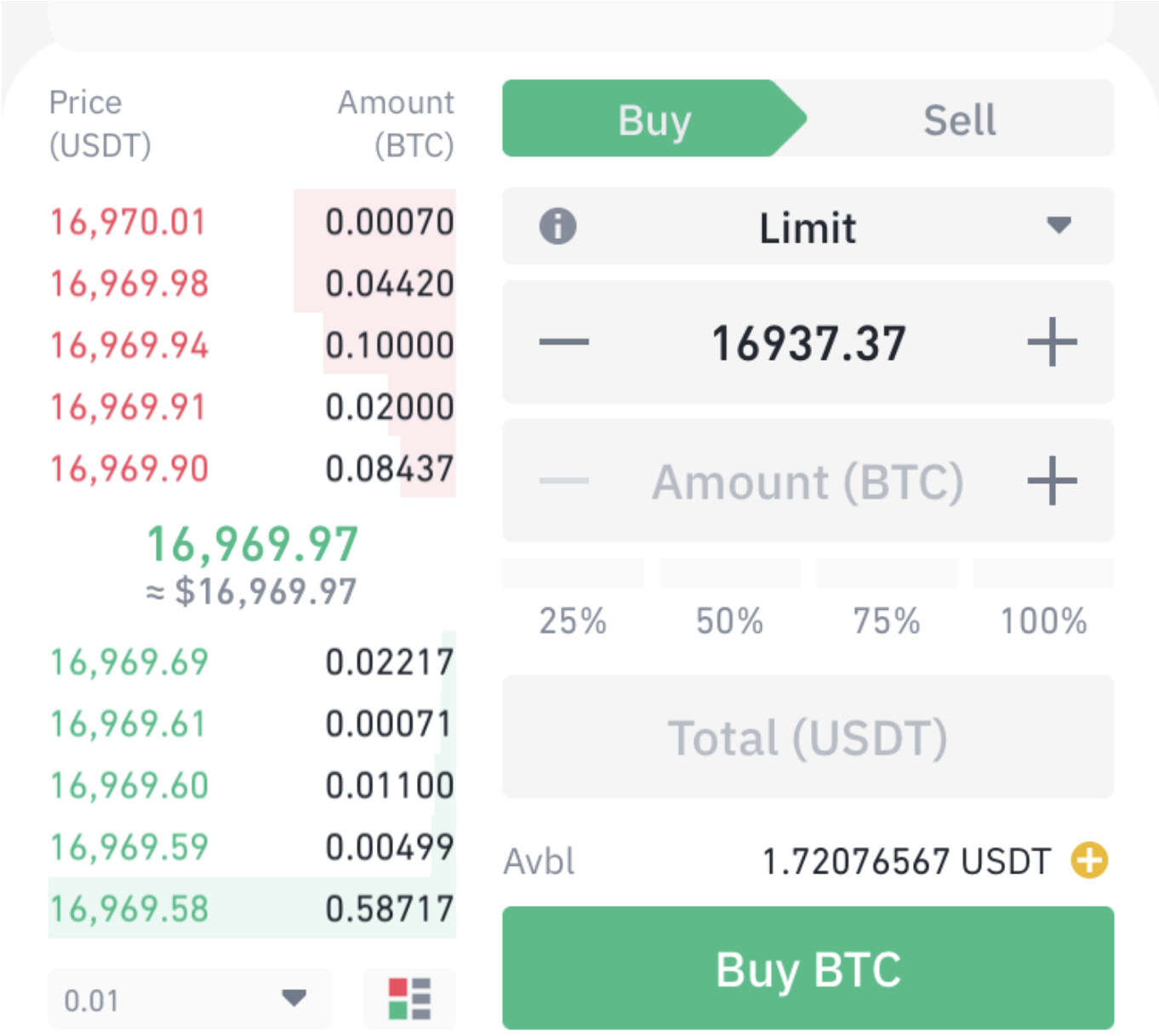
订单簿交易:

订单匹配

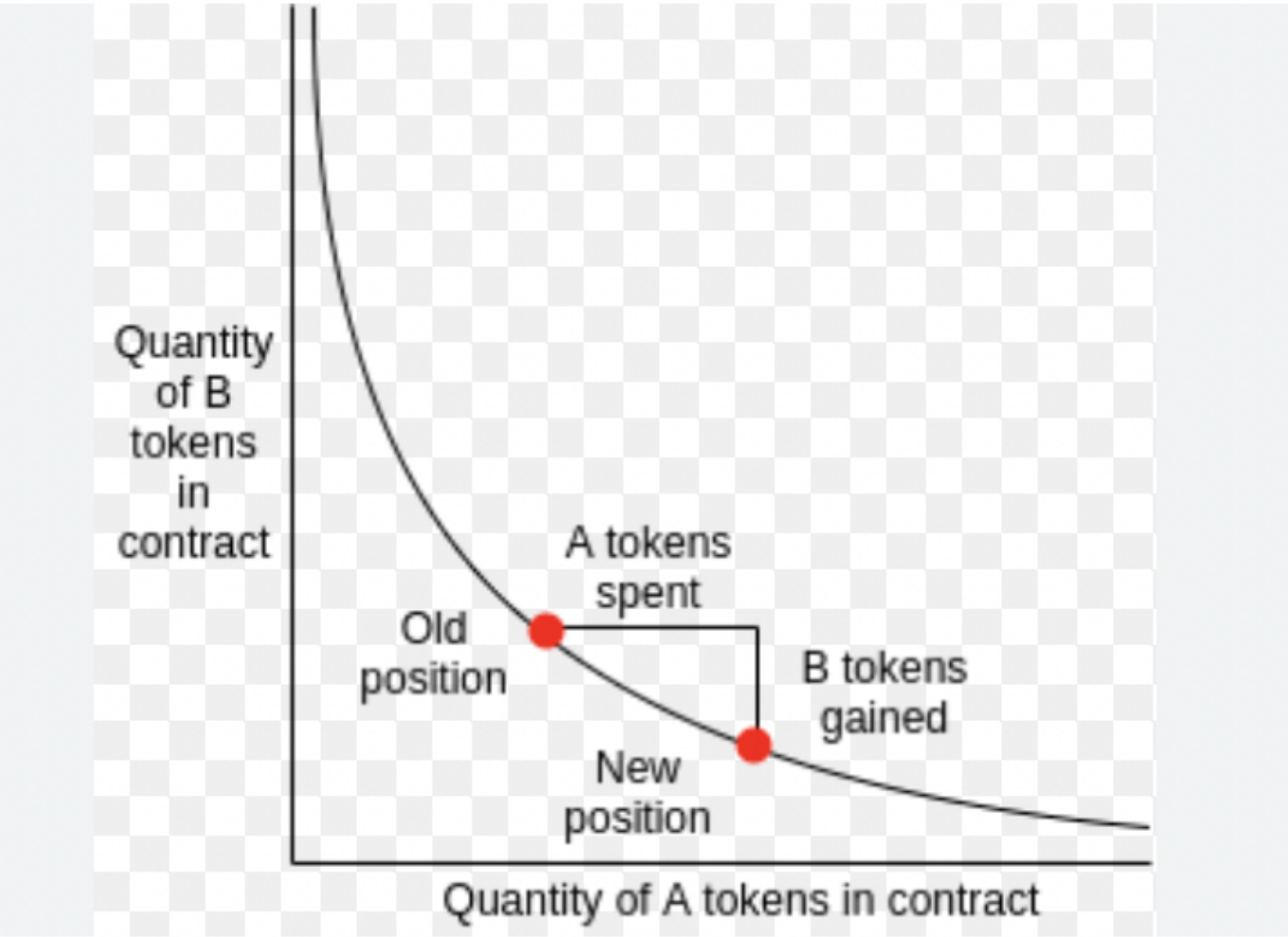
举个例子，如下图，是一个包含了前三行的中央限价订单簿（CLOB）。

- 订单簿中所有的订单，都是“剩余订单”（即还没人买的报价单和还没人卖的出价单）。
- 买卖数量（qty）即是这个价位所有订单的数量总和。

Level	Bid Qty	Bid Price	Spread	Offer Price	Offer Qty
1	2	9650	1	9651	1
2	1	9649	4	9653	**6**
3	5	9648			



uniswap2交易:

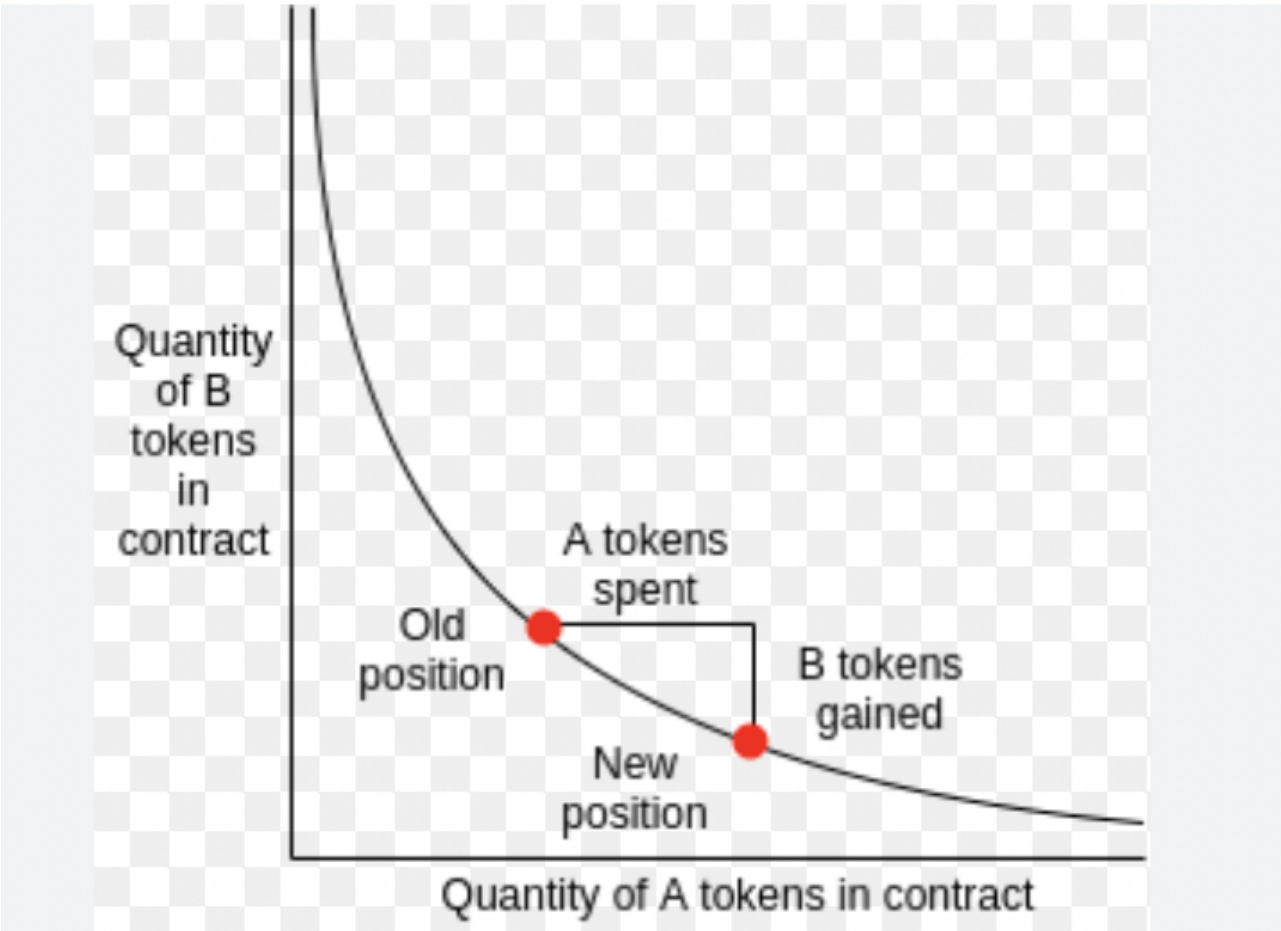


uniswap2

- 基础概念

  uniswap2协议采用恒定常数公式实现.

  $$x * y = k$$



x代表tokenA的数量,y代表tokenB的数量,k代表流动性

tokenA和tokenB的交易对(pair)也叫池子(pool)

那么tokenA的价格(以tokenB为参照物)即是$P_a = y / x $, tokenB的价格$P_b = x / y $

- swap接口(router02合约)
  - swapExactTokensForTokens

```
function swapExactTokensForTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint[]
memory amounts)
```

```
amountIn = 1e+18;
amountOutMin = 0.09249 * 1e+6;
path = [hiBAYC, USDT]
```

```
to = "0x0000000000000000000000000000000000000000"
deadline = time.now() + 60; //1670818353
```

- swapTokensForExactTokens

```
function swapTokensForExactTokens(
    uint amountOut,
    uint amountInMax,
    address[] calldata path,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint[]
memory amounts)
```

```
amountOut= 0.09249 * 1e+6;
amountInMax = 1e+18;
path = [hiBAYC, USDT]
to = "0x0000000000000000000000000000000000000000"
deadline = time.now() + 60; //1670818353
```

- swapExactETHForTokens

```
function swapExactETHForTokens(uint amountOutMin, address[]
calldata path, address to, uint deadline)
external
virtual
override
payable
ensure(deadline)
returns (uint[] memory amounts)
```

- swapTokensForExactETH

```
function swapTokensForExactETH(uint amountOut, uint amountInMax,
address[] calldata path, address to, uint deadline)
external
virtual
override
ensure(deadline)
returns (uint[] memory amounts)
```

- swapExactTokensForETH

```
function swapExactTokensForETH(uint amountIn, uint amountOutMin,
address[] calldata path, address to, uint deadline)
external
virtual
override
ensure(deadline)
returns (uint[] memory amounts)
```

- swapETHForExactTokens

```
function swapETHForExactTokens(uint amountOut, address[] calldata
path, address to, uint deadline)
external
virtual
override
payable
ensure(deadline)
returns (uint[] memory amounts)
```

- swapExactTokensForTokensSupportingFeeOnTransferTokens

```
function swapExactTokensForTokensSupportingFeeOnTransferTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
) external virtual override ensure(deadline) {
```

- quotes

```
function quote(uint amountA, uint reserveA, uint reserveB) public
pure virtual override returns (uint amountB)
```

- getAmountOut

```
function getAmountOut(uint amountIn, uint reserveIn, uint
reserveOut)
public
pure
virtual
override
returns (uint amountOut)
```

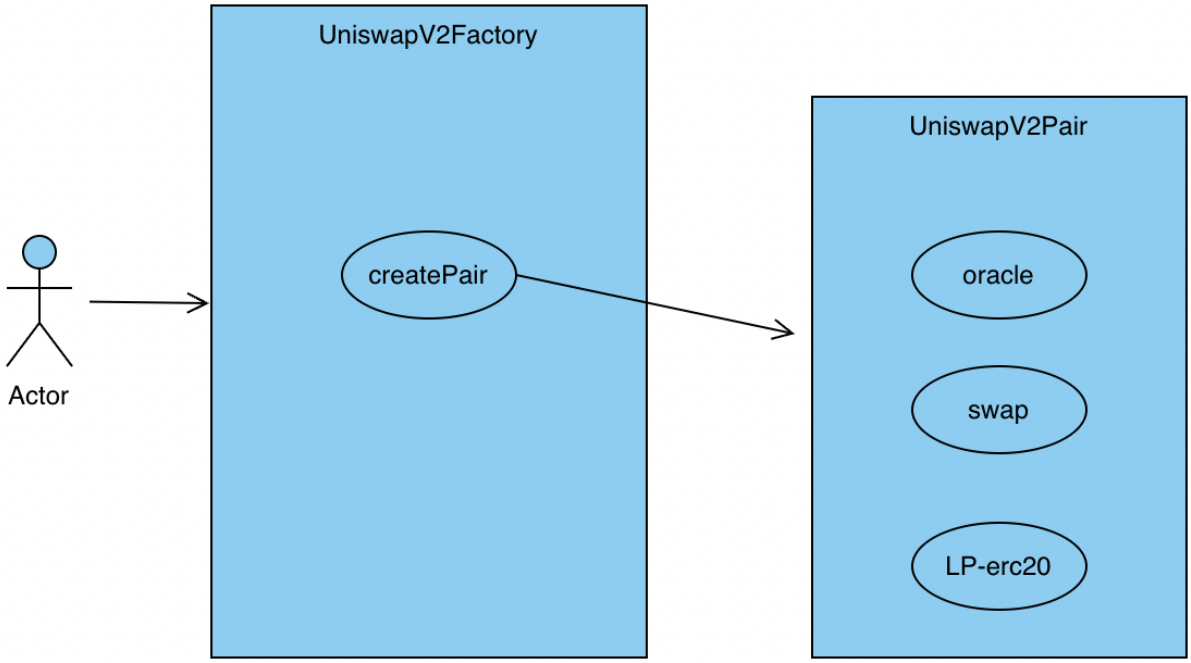
给出`input`数量, 获得最大的`amountOut`

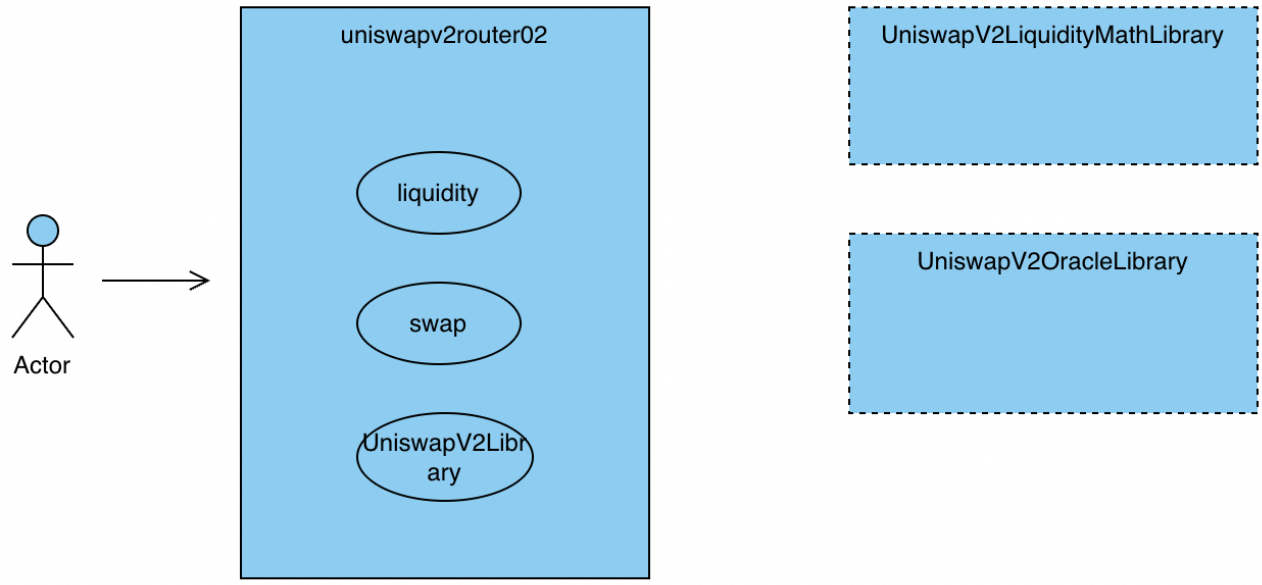
- getAmountIn

```
function getAmountIn(uint amountOut, uint reserveIn, uint
reserveOut)
public
pure
virtual
override
returns (uint amountIn)
```

返回需要购买`amountOut`的`minimum input`数量(包含fee)

- uniswap合约结构





讲述UniswapV2Factoary.sol

讲述`UniswapV2Router02.sol: addLiquidity`接口

- createPair流程
- 初始化流动性过程
- 已有流动性继续添加流动性过程
- erc20凭证如何计算

  $liquitdity = \sqrt{x*y} $

  `MINIMUM_LIQUIDITY`的作用: erc20的价格会随着时间增长而增长，可能导致小流动性提供者无法提供流动性. 增加最小流动性后，如果为了erc20价格增加100刀, 那么攻击者需要 `100刀 * MINIMUM_LIQUIDITY` = 100,000 刀

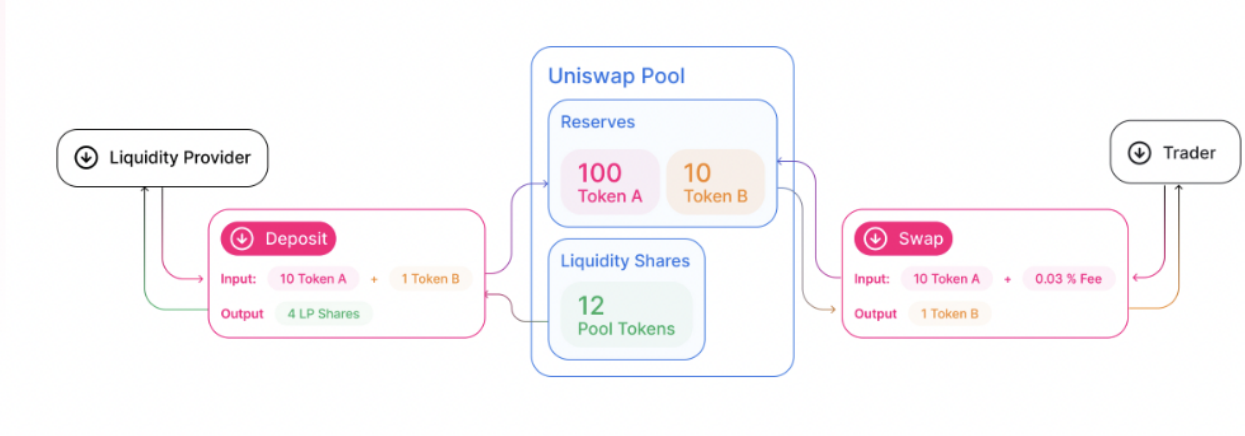
  $liquidity = \sqrt{amount0*totalSupply/reserve0}$

$liquidity = \sqrt{amount0*reserve0/totalSupply}$

使用safeTransfer而不是用IERC20的原因(不等同于reqire(IERC20).transfer()):

- 如果success = false, 调用失败revert
- 如果success = true且没有返回数据，调用成功
- 如果success = true但有数据返回(数据类型解析成bool)
  - 数据解析为true, 调用成功
  - 数据解析为false, 调用失败

讲述`UniswapV2Router02.sol: removeLiquidity`接口



讲述`UniswapV2Router02.sol: swapExactTokensForTokens`接口

- sortToken目的
- lock作用(某些erc20有callback)
- 流动性计算

  因为引入了`flash swap`, amount0In和amount1In可能同时存在,所以公式为

  $ (X_1 - 0.003X_{in}) * (Y_1 - 0.003Y_{in}) >= X_0 * Y_0 $

  $ (1000X_1 - 3X_{in}) * (1000Y_1 - 3Y_{in}) >= X_0 * Y_0 * 1000 * 1000 $

- sync和skim

  sync()函数是一个恢复机制，用来重新调整交易对价格。将reserve更新为balance

  skim()也是一个恢复机制，将`balance-reserve`部分发送给某个地址

- Oracle

```
uint public price0CumulativeLast;
uint public price1CumulativeLast;
```

$P_0 = P_0 + reserve1 / reserve0 * timeElapsed$

$A_t = \sum_{i=1}^{t}P_i$

所以[t1,t2]之间的价格平均数为 $P_{t1,t2} = \frac {(A_{t2} - A_{t1})}{(t_2 - t_1)}$

这里计算的是几何平均数

- Router01和Router02区别

  getAmountOut和getAmountIn为何如此计算

  - getAmountOut

    $\frac {amountIn}{amountOut} = \frac{reserveIn}{reserveOut}$

    $ 997 * amountIn = \frac{reserveIn * amountOut * 1000}{reserveOut}$

    $ amountOut = \frac{997 * amountIn * reserveOut}{reserveIn * 1000}$

    $ amountOut = \frac{997 * amountIn * reserveOut}{reserveIn * 1000 + 997*amountIn}$

  - getAmountIn

    $\frac {amountIn}{amountOut} = \frac{reserveIn}{reserveOut}$

    $ 997 * amountIn = \frac{reserveIn * amountOut * 1000}{reserveOut}$

    $ amountIn = \frac{reserveIn * amountOut * 1000}{reserveOut*997}$

    $ amountIn = \frac{reserveIn * amountOut * 1000}{(reserveOut - amountOut)*997}$

- FlashSwap

  ```
  function uniswapV2Call(address sender, uint amount0, uint amount1,
  bytes calldata data) external;
  ```

  EIP-3156(Flash Loans): https://eips.ethereum.org/EIPS/eip-3156

  借贷callback参数:

  - msg.sender: 作为认证使用
  - token: 告诉借贷的币种
  - amount: 借贷的数量
  - fee: 借贷手续费
  - data: 闪电贷中执行的其它操作

uniswapv3

- 基本概念

  池子的概念:uniswapv2池子的标识符是(tokenA, tokenB), v3的标志符是(tokenA, tokenB, fee)，不同的fee池子互不相通

  价格的概念:uniswapv3只有一个价格

  ```

  ```

```
address tokenA;
address tokenB;

(token0, token1) = tokenA < tokenB? (tokenA, tokenB): (tokenB, tokenA)

P = token0 / token1
```

uniswapv3保存价格通过

$$sqrtPriceLimitX96 = \sqrt{P} * 2^{96} $$

- swap接口
  - exactInputSingle

```
struct ExactInputSingleParams {
    address tokenIn;
    address tokenOut;
    uint24 fee;
    address recipient;
    uint256 deadline;
    uint256 amountIn;
    uint256 amountOutMinimum;
    uint160 sqrtPriceLimitX96;
}

function exactInputSingle(ExactInputSingleParams calldata params)
    external
    payable
    override
    checkDeadline(params.deadline)
    returns (uint256 amountOut)

function exactInput(ExactInputParams memory params)
    external
    payable
    override
    checkDeadline(params.deadline)
    returns (uint256 amountOut)
```

  - exactOutputSingle

```
struct ExactOutputSingleParams {
    address tokenIn;
    address tokenOut;
    uint24 fee;
    address recipient;
    uint256 deadline;
    uint256 amountOut;
```

```
        uint256 amountInMaximum;
        uint160 sqrtPriceLimitX96;
    }

    function exactOutputSingle(ExactOutputSingleParams calldata
    params)
        external
        payable
        override
        checkDeadline(params.deadline)
        returns (uint256 amountIn)

    function exactOutput(ExactOutputParams calldata params)
        external
        payable
        override
        checkDeadline(params.deadline)
        returns (uint256 amountIn)
```

- 基本概念2
  - 流动性

    $$ x * y = k$$

    uniswapv2中保存`x, y`来表示曲线, 那么tokenA的价格(以tokenB为参照物)即是$P_a = y / x $, tokenB的价格$P_b = x / y $

    uniswapv3中采用另一种表示曲线方法, 已知`x * y = k` 且 `y = px`, 那么:

    - $x = \sqrt{\frac{k}{p}}$
    - $y = \sqrt{k*p}$

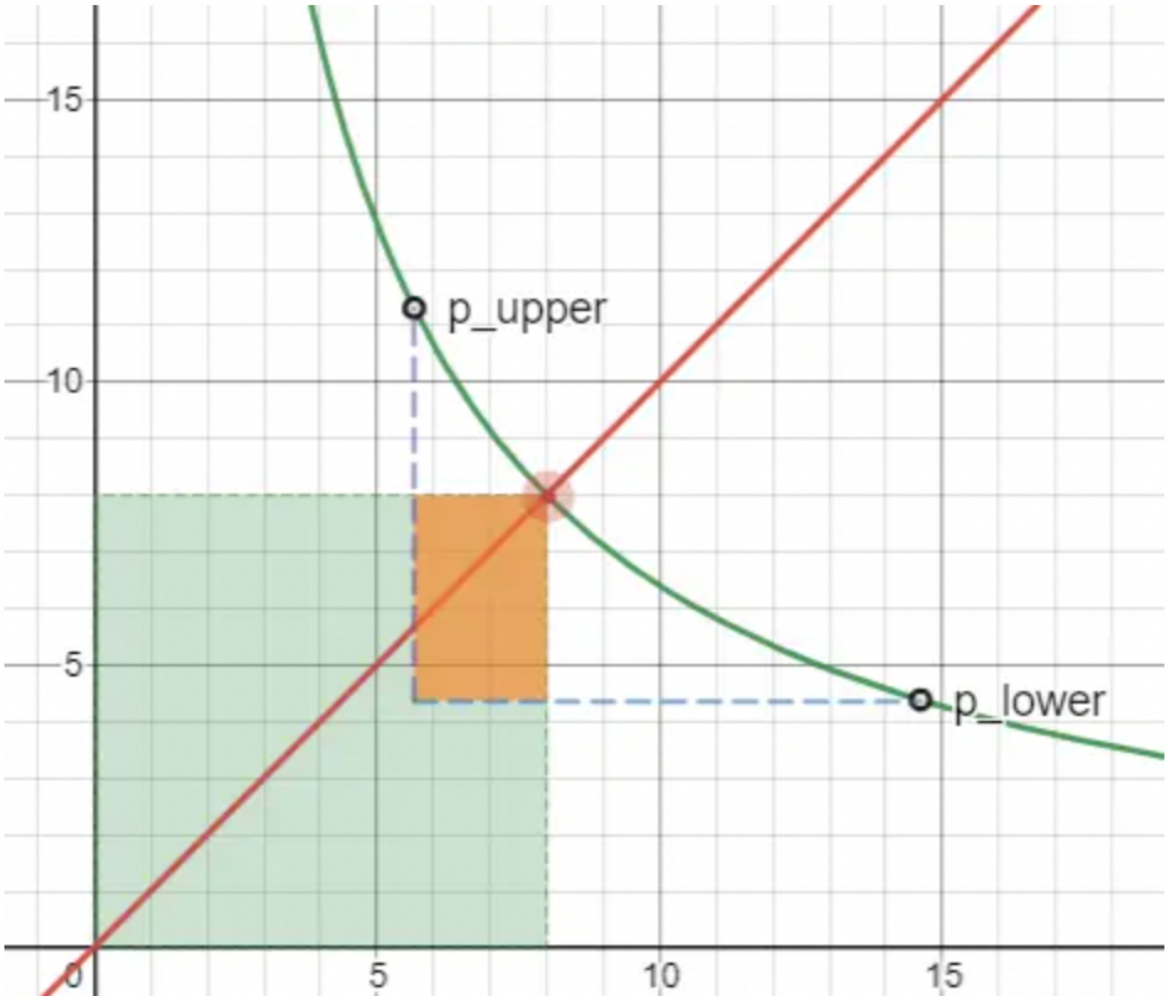
    设置`k = L^2`, 那么:

    - $x = \frac{L}{\sqrt{p}}$
    - $y = L * \sqrt{p}$

    v3中保存 L和$\sqrt{p}$ 来表示曲线, 其中L^2表示流动性, p表示价格

    L 在合约中使用 `uint128`保存, $\sqrt{p}$ 使用 `uint160`保存

  - v3的聚集流动性



从图中可以看到, 价格 $p \in [p_{upper},p_{lower}]$时，提供的流动性为$k = x * y$, 实际使用的流动性为$\Delta{L} = \Delta{x} * \Delta {y}$, 显然流动性利用率很低。

实际上用户提供Δ{x}和Δ{y}的流动性就可以满足。

所以在uniswapv3中，用户提供某个价格范围内的流动性，用户可以选择不同的价格范围提供流动性。反应在曲线中即用户只提供曲线中的虚拟流动性

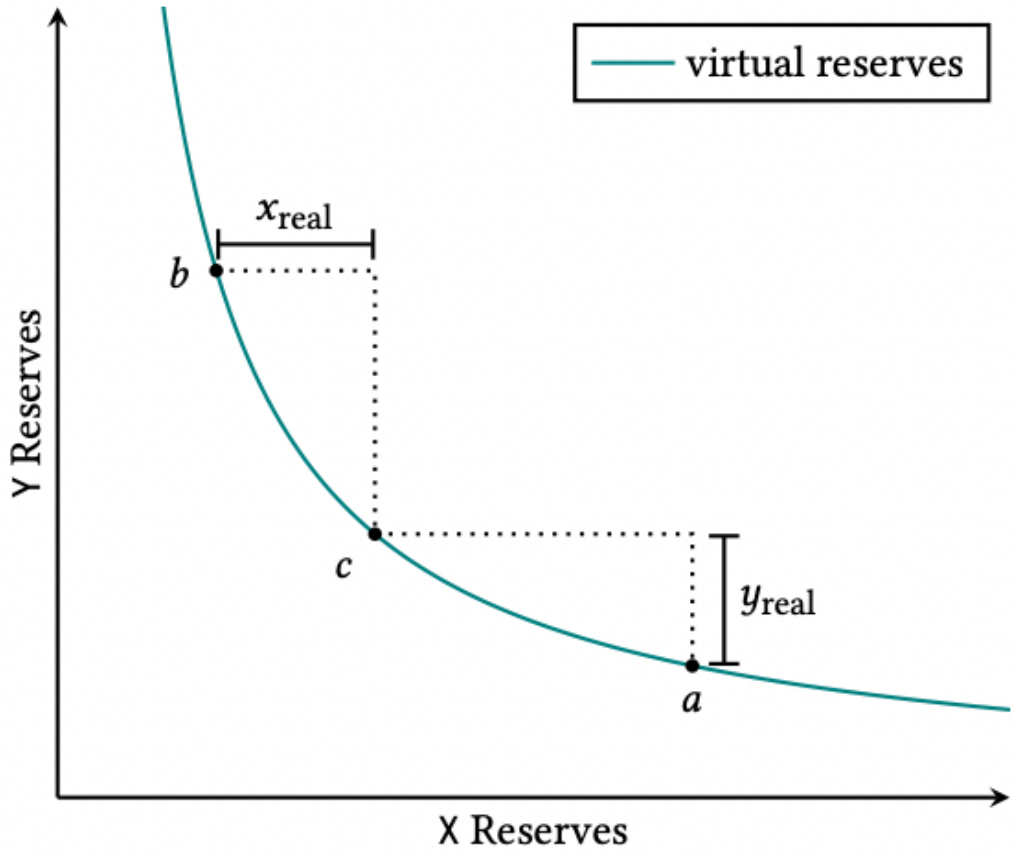


Figure 1: Simulation of Virtual Liquidity

- 价格

  在uniswapv2中，

  - 添加流动性通过`amount0,amount1`参数添加
  - 默认添加价格范围为(0, ∞)
  - 流动性份额使用erc20来表示, 通过等比公式求出. //流动性份额用来计算赎回token

  在uniswapv3中，因为用户可以选择不同价格范围添加，所以不同价格产生的流动性份额也不等价

  - 添加流动性通过`amount0, amount1`参数添加
  - 添加价格范围通过`tickLower, tickUpper`指定
  - 流动性份额通过L表示 //流动性份额用来计算手续费

  uniswapv2中，因为添加的价格范围是一样的，通过更改`reserve0, reserve1`即可表示曲线，

  求上述参数也是求曲线表示方法.

  先看看uniswapv3流动性曲线，

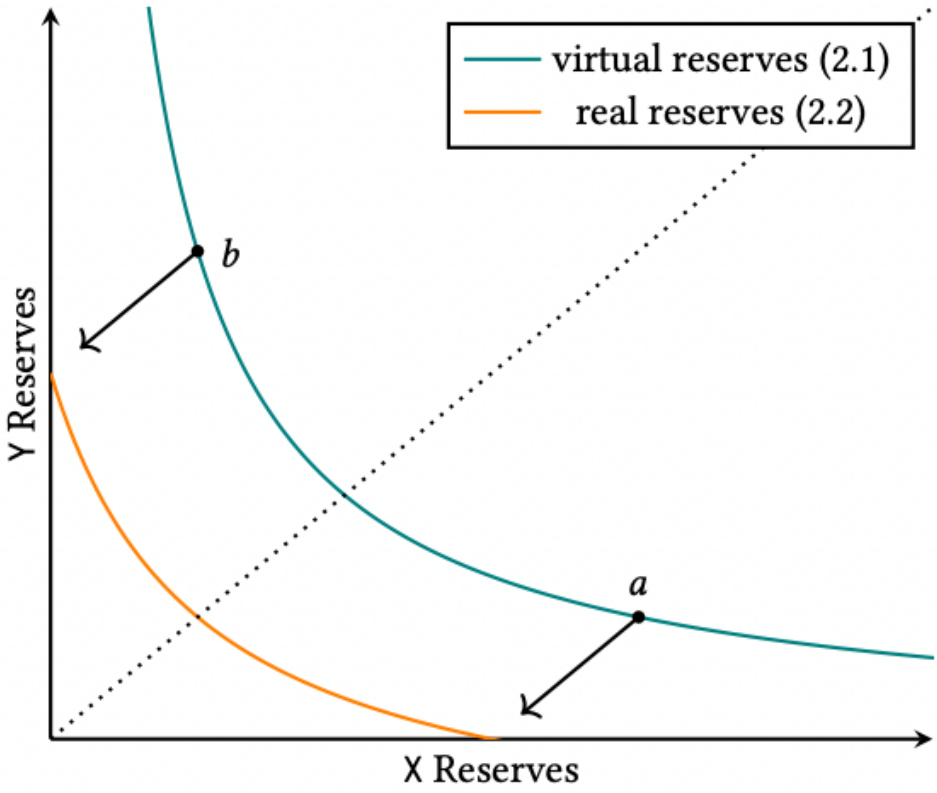


Figure 2: Real Reserves

图上曲线可以看作坐标轴平移.

{
记平移后的坐标原点$O^{'}(h, k)$, 取坐标系上任一点$P(x, y)$, 利用平面向量, 有 $$(x, y) = (h, k) + (x^{'}, y^{'})$$ }

求得图上曲线$(x + \frac{L}{\sqrt{p_{upper}}}) * (y + L * \sqrt{p_{lower}}) = L^2 $

在V3中添加流动性，是需要用户自己设置需要做市的价格空间的。v3中创建者不同或者价格区间不同都是不同的流动性头寸(position)

- 价格区间的风险和收益

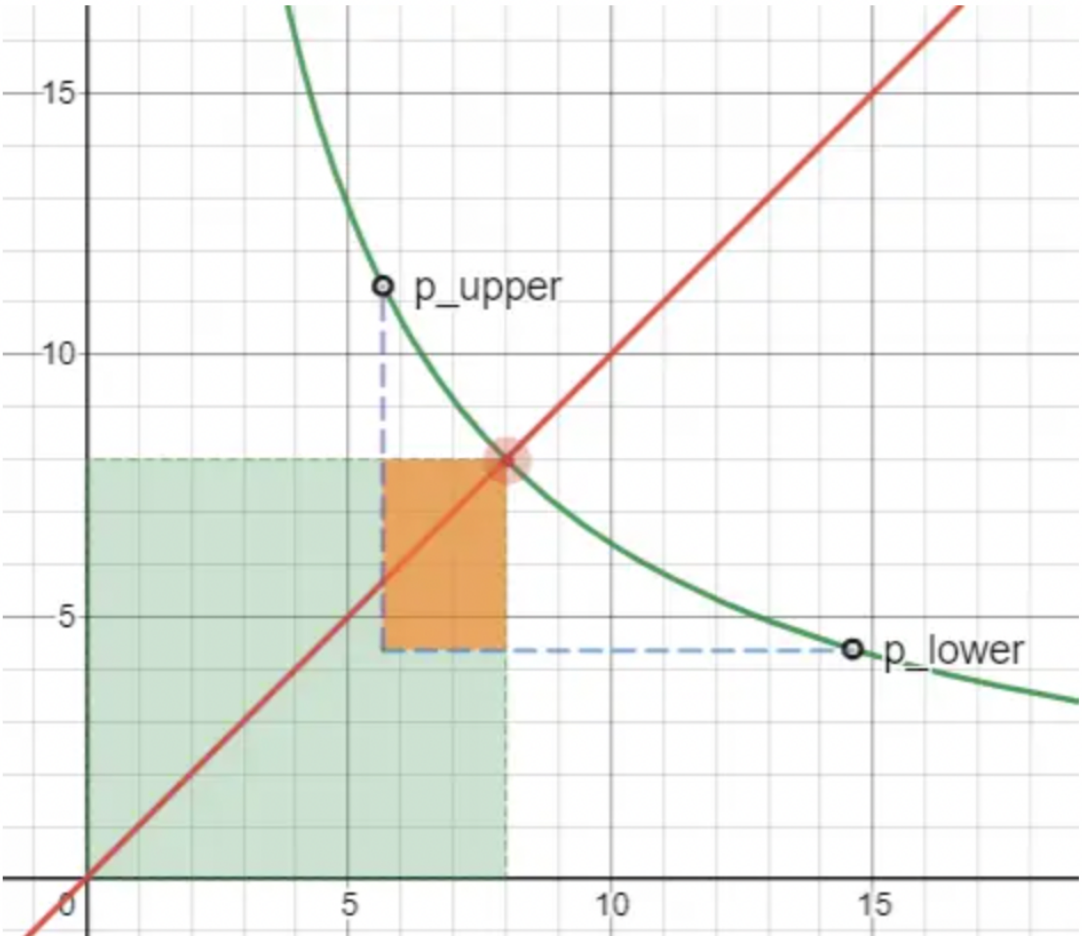
  当出现价格在区间以外的情况下，此时流动性头寸不但不能继续赚取手续费，同时必定变成单一资产，且一定是当时市场中处于弱势一方的资产。比如当资产Y涨价，将会有大量订单用资产X从池子中换取Y,于是池子中的X越来越多，而Y最终会清零。因为AMM自动做市，实际上也是一种被动的做市，永远需要和市场中的订单做对手盘.

  也就是说，价格区间越窄，价格移出区间的概率越大，风险越大，而区间越宽，风险就越小。如果你厌恶这种价格移出区间的风险，那么大可以直接将价格区间设置为$(0, \infty)$

- 添加流动性和移除流动性

  添加流动性的计算过程，是已知当前价格和输入的其中一种资产数量，计算另一种资产数量和添加的流动性数量。

- 价格p在区间内



计算橙色部分。 橙色部分宽高称为Δ{x}, $\Delta{y}。则

$\Delta{x} = L / \sqrt{p} - L / \sqrt{p_{upper}} = L * (\sqrt{p_{upper}} - \sqrt{p}) / \sqrt{p_{upper}} * \sqrt{p}$

$\Delta{y} = L * \sqrt{p} - L * \sqrt{p_{lower}} = L * (\sqrt{p} - \sqrt{p_{lower}})$

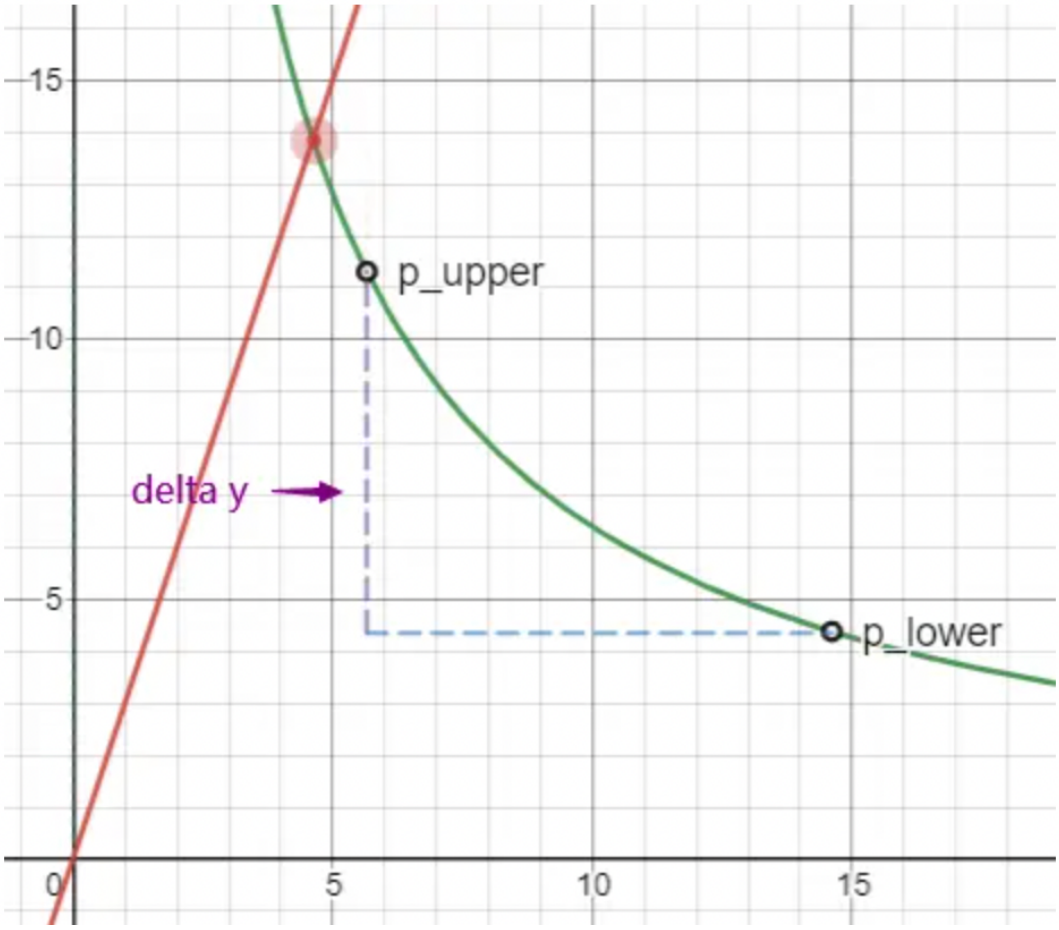
则流动性L等式为

$ L = \Delta{x} * \sqrt{p_{upper}} * \sqrt{p} / (\sqrt{p_{upper}} - \sqrt{p})$

$ L = \Delta{y} / (\sqrt{p} - \sqrt{p_{lower}})$

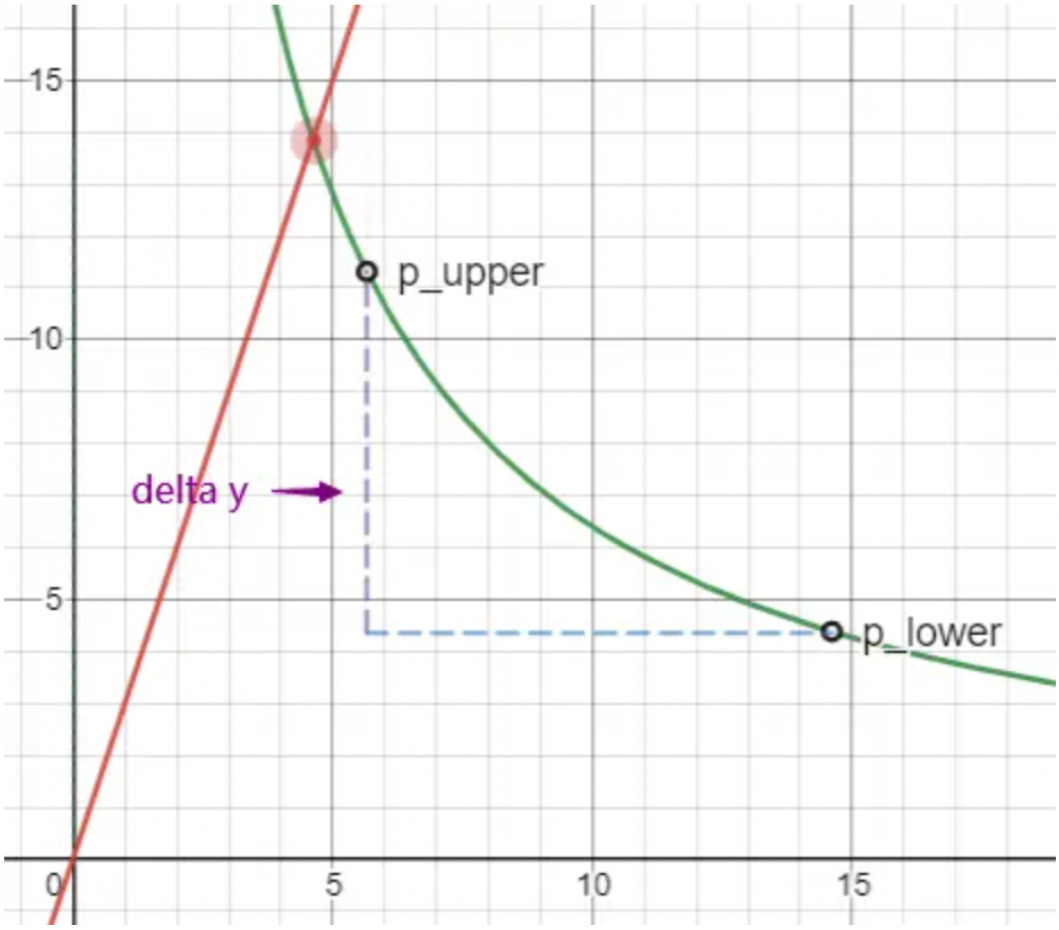
其中Δ{x}和Δ{y}就是其中将要注入流动性的资产数量

- 价格p大于区间



$ L = \Delta{y} / \sqrt{p_{upper}} - \sqrt{p_{lower}}$

- 价格p小于区间



$ L = \Delta{x} * \sqrt{p_{upper}} * \sqrt{p_{lower}} / (\sqrt{p_{upper}} - \sqrt{p_{lower}})$

- 价格的tick表示

  在v2中，提供的流动性是统一的手续费标准(0.3%), 统一的区间价格$(0, \infty)$, 所以大家的资金都是均匀分布在整个价格轴之上的，因此在每一笔交易的过程中，大家收取手续费的权重计算比例也应该是相等的。即每一笔手续费的收益分配，只基于用户提供的流动性相对于总流动性相对于总流动性的比例来分配，出资多的得到的收益多。

  然而v3的价格区间做市机制让v2的分配机制不再公平，因为流动性都有不同的做市区间，使用的流动性是不同的，再用出资比例来分配是不合理的。

  正确的方式应该是记录每一笔交易，都是用了哪些流动性头寸`position`，再将这些头寸按比例分配。

  这样在理论上来说是合理的，如果是中心化的网络这么干的成本可能承受的起，但放到区块链的运行环境中，频繁的进行高昂的读写操作是不可行的，会为用户带来极高的 gas 费用，使得交易的摩擦成本飙升，变得不划算。

  优化gas的第一步是采用离散的价格轴。一个价格轴上由离散的价格点组成，离散函数可以是等差数列、等比数列、幂函数。

  v3使用的是幂函数, 即 $p_{i} = 1.0001^{tick}$， 那么 $tick = log_{1.0001}{p_{i}}$

  v3合约实现价格范围是$[2^{-128}, 2^{128}]$, 求出对应tick范围 $[-887272, 887272]$, 使用`int24`足以表示

一个交易对不需要所有的tick点，于是引入tickSpacing. tickSpacing是两个有效tick之间的距离

```
feeAmountTickSpacing[500] = 10;
feeAmountTickSpacing[3000] = 60;
feeAmountTickSpacing[10000] = 200;
```

- fee手续费原理

在之前，已经拿到了流动性分布情况，那么每笔交易进行过程中，就可以根据相应的交易量对应相应的流动性头寸，将其产生的手续费数量计算出来

- 一笔交易在已有流动性的tick之间，逐个进行
- 当价格来到一个tick(已有流动性的tick), 记录在这个tick上的交易量，根据费率算出该区间的总手续费
- 找出所有包含该tick流动性头寸的position, 将其数量汇总
- 再按照出资比例逐个分配手续费，将数值累加到每个头寸的待收取手续费变量上

这个实现方案十分消耗gas

- 相同position之间的分配

  为了简化问题，我们假设这个池子内所有的流动性价格区间都是 (0, ∞)。

  对于计算流动性大小的比例（出资比例），不得不在交易过程中去遍历的原因在于，池内所有流动性头寸 position ，他们的数量和头寸之间的大小比例是会不断变化的。你无法缓存一个固定的比例，然后在每次交易时按照这个固定的比例分配。

  因为流动性头寸可能随时会移除或注入，这个比例是无法保证一直正确的，每次交易都需要遍历计算。

  所以我们应该放弃计算流动性大小的比例，转而去计算每单位流动性能够获取的手续费数量（即 1 个流动性产生的手续费收益）

  在提供流动性的用户提取手续费的时候，我们是可以获取他提供的流动性数量的，那么只要将其数量和每单位获取的手续费收益相乘，就是他应得的手续费数量。每单位流动性应得的手续费数量只需要用全局的总手续费除以全局的流动性总量即可。

  $ fee = \Delta{L} * (fee_{global} / liquidity_{global})$

  记录一个全局的每单位赚取流动性手续费累计值`feeGrowthGlobal`,另一个相关变量是`liquidity`

  - feeGrowthGlobal: 每当有交易发生，就会把每单位流动性赚取的手续费累加到该变量上，这是一个随时间推移单调递增的变量。（这个变量其实有两个，因为手续费的收取是两种资产分别收取的，所以累计的变量也对应了两个）
  - liquidity: 池内当前处于激活状态的总流动性数量。(**注意并非是注入的所有流动性总量，而是只计算价格区间包含当前价格的流动性数量，这个变量的含义非常容易误解**)

$feeGrowthGlobal = feeGrowthGlobal + \Delta{fee} / liquidity$

$ \Delta{fee} = \Delta{feeGrowthGlobal} * liuquidity$

- 不同区间`position`之间的分配

  这里处理的问题是如何把`feeGrowthGlobal`公平且省gas的分配给不同区间

  手续费产生根源: 每当价格移动到某个tick上，会不断消耗其上的流动性，这是产生手续费的根源。

  那么，将每个tick上产生的手续费增量记录到tick上，用户收取手续费再汇总其区间所有tick数据（需要遍历，十分消耗gas)

  打破注意力的局限性，把目光移动到价格区间之外，有一种变量不需要遍历计算，**区间外的手续费**。

  $feeGrowthInside = feeGrowthGlobal - feeGrowthOutside$

  交易在(a, b)价格区间内进行，手续费会在区间不断累加，但(0,a) 和 (b, ∞)区间内的手续费并没有增加，那么

  $ feeGrowthInside = feeGrowthGlobal - feeGrowthOutside_{below} - feeGrowthOutside_{above}$

  在tick上新增一个`feeGrowthOutside`变量，记录该tick作为流动性边界时，区间外的手续费总量.

  如果以某个tick为中轴,价格点所在一侧时内侧，而相对的另一侧是外侧。当价格中轴穿过tick时，更新`feeGrowthOutside`

  当没有流动性将该tick作为边界时，不用考虑这个变量。这里说下记录tick流动性的两个变量

  - liquidityGross: 每当有流动性将该tick设为价格区间时，liquidityGross增加。即`liduitidyGross > 0`说明tick已经初始化，有流动性; `liquidityGross == 0`则表示没有流动性
  - liquidityNet: 当价格中轴穿过该tick时，处于激活的流动性需要变化的数量
    - addLiquidity, tick是价格下限: liquidityNet 增加 l
    - addLiquidity, tick是价格上限: liquidityNet 减少 l
    - removeLiquidity, tick是价格下限: liquidityNet 减少 l
    - removeLiquidity, tick是价格上限: liquidityNet 增加 l

  流动性有激活和非激活两个状态，一个池子内处于激活的流动性数量总是随着价格$\sqrt{p}$变动而变动

  回到手续费(a,b)区间的计算过程

  $feeGrowthOutside_{below} = feeGrowthOutside_{a}$

  $feeGrowthOutside_{above} = feeGrowthGlobal - feeGrowthOutside_{b}$

$feeGrowthInside = feeGrowthGlobal - feeGrowthOutside_{a} - (feeGrowthGlobal - feeGrowthOutside_{b})$

$feeGrowthInside = feeGrowthOutside_{b} - feeGrowthOutside{a}$

//todo:

- v3手续费完整流程
  - 首先记录池子内全局的每单位流动性收取手续费的数量的累加变量`feeGrowthGlobal`
  - 每个有流动性tick上记录外侧的手续费`feeGrowthOutside`
  - 任意区间，利用公式计算出区间内的手续费总量`feeGrowthInside`

    $$feeGrowthInside = feeGrowthGlobal - feeGrowthOutside_{lower} - feeGrowthOutside_{upper}$$
  - 该区间存在不同用户注入的不同流动性头寸position，在position流动性数量发生变化时，累加距离上次变化到现在这段时间的手续费数量增量到`tokensOwed`变量

    $$tokensOwed = tokensOwed + feeGrowthInside * liuquidity_{position}$$
  - 用户通过`collect`收取手续费，从`tokensOwed`变量中扣除

tips1: v3的position有两种，一种是Pool合约的，只要价格区间相同，就是同一种position;而另一种是PositionManager合约中的，该position会区分不同用户。

tips2：v2的手续费计算流程是隐式的，合并在交易和移除流动性的过程中。v2没有考虑做市时长的因素，也就是说后添加的流动性会摊薄之前流动性的手续费收益

```
function getFeeGrowthInside(
    mapping(int24 => Tick.Info) storage self,
    int24 tickLower,
    int24 tickUpper,
    int24 tickCurrent,
    uint256 feeGrowthGlobal0X128,
    uint256 feeGrowthGlobal1X128
) internal view returns (uint256 feeGrowthInside0X128,
uint256 feeGrowthInside1X128) {
    Info storage lower = self[tickLower];
    Info storage upper = self[tickUpper];

    // calculate fee growth below
    uint256 feeGrowthBelow0X128;
    uint256 feeGrowthBelow1X128;
    if (tickCurrent >= tickLower) {
        feeGrowthBelow0X128 = lower.feeGrowthOutside0X128;
        feeGrowthBelow1X128 = lower.feeGrowthOutside1X128;
    } else {
        feeGrowthBelow0X128 = feeGrowthGlobal0X128 -
lower.feeGrowthOutside0X128;
```

```
            feeGrowthBelow1X128 = feeGrowthGlobal1X128 -
lower.feeGrowthOutside1X128;
        }

        // calculate fee growth above
        uint256 feeGrowthAbove0X128;
        uint256 feeGrowthAbove1X128;
        if (tickCurrent < tickUpper) {
            feeGrowthAbove0X128 = upper.feeGrowthOutside0X128;
            feeGrowthAbove1X128 = upper.feeGrowthOutside1X128;
        } else {
            feeGrowthAbove0X128 = feeGrowthGlobal0X128 -
upper.feeGrowthOutside0X128;
            feeGrowthAbove1X128 = feeGrowthGlobal1X128 -
upper.feeGrowthOutside1X128;
        }

        feeGrowthInside0X128 = feeGrowthGlobal0X128 -
feeGrowthBelow0X128 - feeGrowthAbove0X128;
        feeGrowthInside1X128 = feeGrowthGlobal1X128 -
feeGrowthBelow1X128 - feeGrowthAbove1X128;
    }
```

上述原理讲述完后，liquidityGross, liquidityNet, liquidity, 穿过后feeOutside变化 有些不明.

这里先说下feeOutside变化, 每次越过刻度时需要修改feeOutside。当刻度从任意方向被越过时，它的feeOutside做如下修改:

$f_{o}(i)$ = f_{g} - $f_{o}(i)$

- 添加流动性和移除流动性
  - mint

    讲述`NonfungiblePositionManager.sol:128 mint()`函数

    - addLiquidity
      - 计算 poolKey, 根据(token0, token1, fee)
      - 根据当前价格范围计算流动性l
      - 调用pool.mint()
        - _modifyPosition
          - _updatePosition
            - 计算出 position, 根据(owner(==this-manager), tickLower, tickUpper)
            - 更新tick上下边界值
            - 更新tickmap
            - 更新pool合约中positions[key]的流动性份额和feeGrowth
          - 根据流动性l计算对应amount，更新pool合约中的liquidity
        - 获取pool当前token余额
        - transferFrom(msg.sender, pool, amount)

            - 余额判断, balance0Before + amount0 <= balance0After
        - _mint()
        - 计算positionKey, 根据(address(this-manager), tickLower, tickUpper), 获取feeGrowth通过pool.positions(positionKey)
        - 计算poolId, 根据(pool, token0, token1, fee)
        - 给Manager合约中的_posisions[tokenId]赋值
    - increaseLiquidity
        - 获取_positions[tokenId]
        - addLiquidity
        - 计算出positionKey, 获取feeGrowth
        - 更新_positions[tokenId]的tokensOwed, 增加 feeGrowth 和 liquidity
    - decreaseLiquidity
    - collect
    - burn
- swap
    - exactInputSingle
        - pool.swap(to, direction, amount, p, data)
            - 判断p值
            - cache(liquidity, blockTimestamp)
            - sate(amount, ${p_{c}}$, ...)
            - while循环
                - 从当前价格向轴上左方向寻找
            - **SwapMath.computeSwapStep**
    - exactOutputSingle
- 其它改进
    - 限价单

      v3通过设置狭窄的价格区间，可以实现类似限价单。跟传统限价单的区别在于.

        - 当价格处于该价格区间内，限价订单会被部分执行
        - 当超出头寸的价格区间时，需要被提取出去。否则当价格重新落回价格区间，资产类别会被逆转
    - 交易对

v3中一个交易对可以多个流动对,(tokenA, tokenB, fee)

- 关于预言机

  在v2中使用预言机的话，需要另外一个合约或链下方式记录其`price0CumulativeLast`历史值。

  v3中新增一个循环数组来记录其历史值，使得v3合约的预言机可直接被使用。

  uniswapv2记录的是累计价格和，从而允许用户计算时间加权的算术平均数；而uniswapv3跟踪的是价格的对数和，从而允许用户计算时间加权的几何平均数

  关于为何采用时间加权几何平均价格能更真实反映平均价格的理论依据，参考度量几何布朗运动过程(几何布朗运动过程的算术平均值，倾向于加大高价格的权重[同样百分比的变化，高价格的变化绝对值更大])