Архитектура Uniswap V3

Указанный протокол представляет собой совокупность смартконтрактов объединенных по функциональному назначению в два основных пакета:

- математическое ядро CFMM и базовые интерфейсы для пула ликвидности, фабрики пулов, интерфейсы событийной модели: https://github.com/Uniswap/uniswap-v3-core
- интерфейсы взаимодействия с пользователем, хранения и изменения состояния пулов ликвидности: https://github.com/Uniswap/uniswap-v3-periphery

Обобщенная схема протокола приведена на рис. 1.

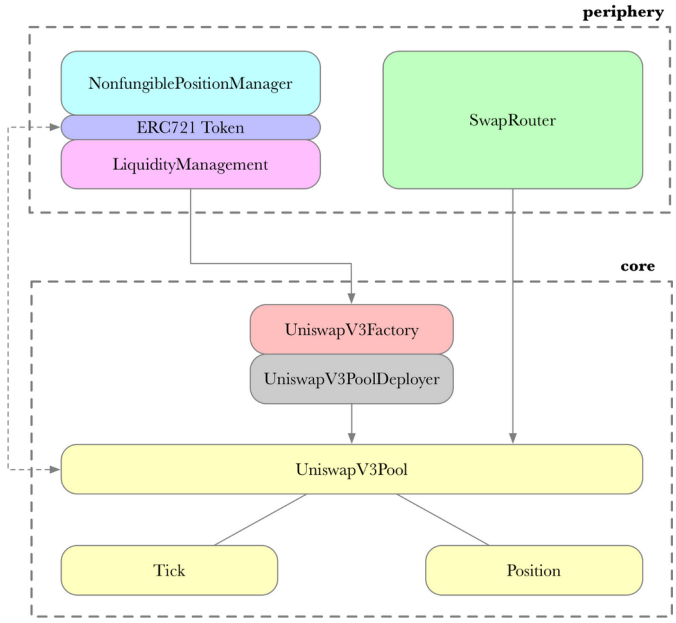

Рис. 1 – Обощенная схема Uniswap V3

Периферия также состоит из двух частей: маршрутизатор обмена (Swap Router) и менеджер позиций (Position Manager). Менеджер позиций отвечает за взаимодействие пользователя со смартконтрактами ядра в части создания пула транзакций и добавления/удаления ликвидности. Маршрутизатор обмена - это средство управления процессом обмена активами. UniswapV3Factory - это унифицированный интерфейс пула транзакций UniswapV3Pool. UniswapV3Pool настраивается и управляется UniswapV3PoolDeployer. UniswapV3Pool - включает логику управления активами и позициями (Position), реализует функции управления ликвидностью и обмена в пуле транзакций. В каждом пуле позиция представляет собой токен ERC721 (имеет свой независимый идентификатор токена ERC721).

Особенность протокола по сравнению с V2 заключается в том, что ликвидность в пуле распределена не равномерно на всей числовой прямой, а в границах диапазона, заданного пользователем в рамках позиции. Таким образом, пул содержит множество как пересекающихся, так и не пересекающих позиций ликвидности, учет которых при операциях обмена, предоставления и снятия активов осуществляется с применением концепции тиков.

Концепция тиков

Пространство возможных цен разграничено дискретными тиками. Поставщики ликвидности могут предоставлять ликвидность в диапазоне между любыми двумя произвольными тиками. Каждый диапазон может быть указан как пара целочисленных индексов тиков со знаком (нижний и верхний). Тики представляют собой цены, при которых может измениться виртуальная ликвидность контракта. Квадратный корень текущей цены вычисляется через текущий тик $\sqrt{p(i)} = \sqrt{1.0001^i}$. Когда ликвидность добавляется к диапазону, если один или оба тика еще не используются в качестве границы в существующей позиции, этот тик инициализируется. Пул создается с параметром tickSpacing; только тики с индексами, которые делятся на tickSpacing без остатка, могут быть инициализированы.

Всякий раз, когда цена пересекает инициализированный тик, активируется или исчезает виртуальная ликвидность. Стоимость газа при инициализированном пересечении тиков постоянна и не зависит от количества открываемых или закрываемых позиций в этот тик. Контракт пула использует переменные для отслеживания состояния на глобальном уровне (для каждого пула), на уровне тиков и на уровне позиций.

Один тик может пересекать несколько позиций. При расчете комиссии за транзакцию она должна быть равномерно распределена на общую ликвидность всех позиций в тике. На границе каждого тика происходит обновление состояния тика на величину liquidityDelta (сумма всех позиций с этим тиком в качестве границы). Когда цена колеблется и проходит один определенный тик, ликвидность будет увеличиваться или уменьшаться (зависит от того, в какую сторону колеблется цена). Например, цена пересекает одну позицию слева направо; когда она пересекает первый тик в позиции, ликвидность должна увеличиться. По мере прохождения последнего тика ликвидность уменьшается.

Реализация процедуры обновления состояния тика приведена ниже. Здесь upper - флаг для обновления позиций выше данного тика, info – структура обновляемого тика, flipped – флаг сигнализирующий, что данный тик теперь инициализирован.

```
uint128 liquidityGrossBefore = info.liquidityGross;
uint128 liquidityGrossAfter = LiquidityMath.addDelta(liquidityGrossBefore, liquidityDelta);

flipped = (liquidityGrossAfter == 0) != (liquidityGrossBefore == 0);

if (liquidityGrossBefore == 0) {
  // by convention, we assume that all growth before a tick was initialized happened _below_ the tick
  if (tick <= tickCurrent) {
    info.feeGrowthOutside0X128 = feeGrowthGlobal0X128;
    info.feeGrowthOutside1X128 = feeGrowthGlobal1X128;
    info.secondsPerLiquidityOutsideX128 = secondsPerLiquidityCumulativeX128;
    info.tickCumulativeOutside = tickCumulative;
    info.secondsOutside = time;
  }
  info.initialized = true;
}

info.liquidityGross = liquidityGrossAfter;

// when the lower (upper) tick is crossed left to right (right to left), liquidity must be added (removed)
info.liquidityNet = upper
  ? int256(info.liquidityNet).sub(liquidityDelta).toInt128()
  : int256(info.liquidityNet).add(liquidityDelta).toInt128();
```

Если тик не используется в качестве конечной точки диапазона с какой-либо ликвидностью в нем, то есть если тик не инициализирован, то этот тик можно пропустить во время свопов. В качестве оптимизации, чтобы сделать поиск следующего инициализированного тика более эффективным, пул отслеживает битовую карту tickBitmap инициализированных тиков. Позиция в карте, соответствующая индексу метки, устанавливается в 1, если метка инициализирована, и в 0, если она не инициализирована. Когда тик используется в качестве конечной точки для новой позиции и этот тик в настоящее время не используется какой-либо другой ликвидностью, тик инициализируется, а соответствующий бит в битовой карте устанавливается в 1. Инициализированный тик может снова стать неинициализированным, если вся ликвидность, для которой он является конечной точкой, удаляется, и в этом случае позиция этого тика в карте обнуляется. Навигация внутри карты осуществляется с помощью пары (int16 wordPos, uint8 bitPos) форируемой на основе значения tick / tickSpacing (https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/TickBitmap.sol).

Особенности арифметики тиков

Арифметика тиков приведена в https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/TickMath.sol, где представлены две основополагающие функции преобразования тика в корень из цены по формуле sqrt(1.0001^tick) * 2^96 [формула 6.2] и преобразования тика из корня из цены log_{sqrt(1.0001)}sqrt(p) [формула 6.8].

Функция преобразования тика в корень из цены

Ядром данной функции является процедура быстрого возведения в степень – https://ru.wikipedia.org/wiki/Алгоритмы_быстрого_возведения_в_степень.

Суть алгоритма сводится к следующему:

1) тик (int24) рассматривается как число в двоичной системе счисления tick = m_k * 2^k + m_k-1 * 2^k-1 + … m_1 * 2 + m_0, где m_k принадлежит множеству {0, 1}.
2) с учетом такого разложения константа sqrt(1.0001) возводится в степень 1 << k, если m_k == 1, и результат умножается на результат предыдущей операции,
3) пункт 2 повторяется для всех разрядов tick;
4) если число тиков меньше 0, то требуется результат представить как MAX_UINT_256 / результат;
5) осуществить приведение результата к нужной разрядности и корректировку полученного значения если обнаружена ситуация появления ошибок округления.

Магические константы, к примеру 0xfff97272373d413259a46990580e213a, 0xfff2e50f5f656932ef12357cf3c7fdcc, есть не что иное, как соответствующие степени константы sqrt(1.0001). Убедиться в этом можно запустив следующий код (JS):

```
(0xfffcb933bd6fad37aa2d162d1a594001n >> 32n).toString() / 2 ** 96
// 0.9999500037496876, что есть не что иное, как 1 / √ 1.0001

Math.sqrt((0xfff97272373d413259a46990580e213an >> 32n).toString() / 2 ** 96)
// 0.9999500037496876, что есть не что иное, как 1 / √ 1.0001

Math.sqrt(Math.sqrt((0xfff2e50f5f656932ef12357cf3c7fdccn >> 32n).toString() / 2 ** 96))
// 0.9999500037496876, что есть не что иное, как 1 / √ 1.0001,
соответственно 0xfff2e50f5f656932ef12357cf3c7fdccn – константа связанная с sqrt(1.0001)^-4
```

Реализация данного алгоритма на Go приведена в исходниках.

Функция преобразования тика из корня из цены

Достаточно полное объяснение приведено в https://ethereum.stackexchange.com/questions/113844/how-does-uniswap-v3s-logarithm-library-tickmath-sol-work.

Суть алгоритма сводится к следующему:

1) преобразовать формулу log_{sqrt(1.0001)}sqrt(p) к виду log2(sqrt(p))/log2(sqrt(1.0001)) и в последующем к виду log2(sqrt(p))*log_{sqrt(1.0001)}(2) в соответствии с формулами 12 и 13 https://zaochnik.com/spravochnik/matematika/vyrazhenija/preobrazovanie-vyrazhenij-s-ispolzovaniem-svojstv/
2) осуществить грубый расчет целочисленного значения log2(sqrt(p)) используя битовые операции https://docs.soliditylang.org/en/v0.4.24/assembly.html# .
3) уточнить грубое решение до 14 битовых разрядов,
4) осуществить расчет log2(sqrt(p))*log_{sqrt(1.0001)}(2), используя магическую константу 255738958999603826347141 ~ console.log(BigInt((1.0 / Math.log2(Math.sqrt(1.0001))) * 2 ** 64).toString())
5) далее осуществить корректировку полученного значения если обнаружена ситуация появления ошибок округления:

```
int24 tickLow = int24((log_sqrt10001 - 3402992956809132418596140100660247210) >> 128);
int24 tickHi = int24((log_sqrt10001 + 291339464771989622907027621153398088495) >> 128);

tick = tickLow == tickHi ? tickLow : getSqrtRatioAtTick(tickHi) <= sqrtPriceX96 ? tickHi : tickLow;
```

Способы расчета логарифма: https://ethereum.stackexchange.com/questions/8086/logarithm-math-operation-in-solidity/32900#32900

Реализация данного алгоритма на Go приведена в исходниках.

Расчет роста собранной комиссии

Реализация рассчета комиссии за транзакцию, соответствующая формулам 6.17-6.20, приведена ниже. Здесь feeGrowthGlobal0X128 – глобальная величина роста комиссии на единицу ликвидности в терминах токена 0, feeGrowthGlobal1X128 – аналогично feeGrowthGlobal0X128 в терминах токена 1, feeGrowthInside0X128 – величина роста комиссии на единицу ликвидности в терминах токена 0 внутри заданных границ, определенных с помощью tickLower и tickUpper, feeGrowthInside1X128 – аналогично feeGrowthInside0X128 в терминах токена 1.

```
// 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;
```

Чтобы рассчитать ставку комиссии за транзакцию в позиции, мы вычитаем ставку за пределами позиции из общей суммы. На граничном тике отмечаем feeGrowthOutside0X128. feeGrowthOutside0X128 означает общую комиссию «противоположного» направления. К примеру, когда цена проходит через тик слева направо, feeGrowthOutside0X128 относится к комиссии за транзакции всех позиций слева от тика. Поскольку это общая ставка комиссии транзакций в обоих направлениях за тик, сумма ставок в обоих направлениях должна быть равна feeGrowthGlobal0X128. Поэтому при прохождении тика feeGrowthBelow0X128 и feeGrowthAbove1X128 необходимо “флипнуть”.

Когда текущая цена больше, чем цена тика вся комиссия происходит ниже цены тика. Если цена тика больше текущей цены, и цена не преодолела тик, так как мы предположили, что вся предыдущая комиссия произошла ниже цены тика, комиссия выше тика не взимается.

Алгоритм обмена активами с использованием концепции тиков

Логика процесса обмена представлена в https://github.com/Uniswap/v3-periphery/blob/main/contracts/SwapRouter.sol. Определено два разных типа функций:

- exactInputSingle/exactInput,
- exactOutputSingle/exactOutput,

exactInputSingle, exactOutputSingle - это функции обмена в едином пуле транзакций. Одна функция осуществляет определение входного значения актива в обмен на определенное количество выходного актива, в то время как другая функция осуществляет определение выходного значения по заданному входному.

Опуская из последующего изложения особенности реализации указанных функций перейдем к описанию алгоритма основополагающей функции обмена ядра Uniswap V3, а именно (https://github.com/Uniswap/v3-core/blob/main/contracts/UniswapV3Pool.sol):

```
function swap(
        address recipient,
        bool zeroForOne,
        int256 amountSpecified,
        uint160 sqrtPriceLimitX96,
        bytes data) external returns (int256 amount0, int256 amount1)
```

где *recipient* – адрес получателя;

zeroForOne – направление обмена: true для token0 -> token1, false для token1 -> token0;
amountSpecified – количество токенов, которое будет обменено – положительное для exactInput , отрицательное для exactOutput;
sqrtPriceLimitX96 – пороговое значение цены, “выше” которой итерационный процесс обмена прекратится;
amount0 – дельта баланса token0 (округленное значение);
amount1 – дельта баланса token1 (округленное значение).

Алгоритм функции swap() обмена активами:

1) задаются начальные значения для текущего тика, ликвидности и корня из цены, определяется направление движения вдоль тиков (слева направо, либо справа налево), а также параметр exactInput;
2) определяется следующий инициализированный тик с использованием карты тиков. В случае, если тик выходит за границы допустимых значений, то он устанавливается в ближайшее допустимое граничное значение;
3) формируется следующее значение корня из цены с учетом значения полученного тика по формуле [6.1, whitepaper-v3], а также значения *sqrtPriceLimitX96;*
4) формируются величины входного/выходного количества токенов и цены после операции обмена *amountIn*, *amountOut* и *sqrtRatioNextX96* (с учетом округления и переполнения):
   a) если порядок расчета exactIn и направление расчета $0 \rightarrow 1$, то приращение к токену 0 равно liquidity * (sqrt(upper) - sqrt(lower)) / (sqrt(upper) * sqrt(lower)), где upper – большее из step.sqrtPriceNextX96 и sqrtPriceLimitX96, для lower – наоборот;
   b) если порядок расчета exactIn и направление расчета $1 \rightarrow 0$, то приращение к токену 1 равно liquidity * (sqrt(upper) – sqrt(lower));
   c) если порядок расчета exactOut и направление расчета $0 \rightarrow 1$, то приращение к токену 0 равно liquidity * (sqrt(upper) – sqrt(lower));
   d) если порядок расчета exactOut и направление расчета $1 \rightarrow 0$, то приращение к токену 1 равно liquidity * (sqrt(upper) - sqrt(lower)) / (sqrt(upper) * sqrt(lower));
   e) если порядок расчета exactIn, то sqrtRatioNextX96 определяется из соотношения liquidity * sqrtPX96 / (liquidity +- amount * sqrtPX96) для $0 \rightarrow 1$, либо sqrtPX96 +- amount / liquidity для $1 \rightarrow 0$;
   f) если порядок расчета exactIn, то sqrtRatioNextX96 определяется из соотношения liquidity * sqrtPX96 / (liquidity +- amount * sqrtPX96) для $1 \rightarrow 0$, либо sqrtPX96 +- amount / liquidity для $0 \rightarrow 1$;
5) рассчитывается величина комиссии feeAmount;
6) исключается комиссия провайдера;
7) корректируется величина доступной ликвидности после перехода через тик;
8) шаги с 2 по 7 повторяются, пока еще имеется актив для обмена и не достигнуто предельное значение корня из цены *sqrtPriceLimitX96;*
9) формируются значения *amountIn*, *amountOut:*

```
if zeroForOne = = exactInput {
        amount0.Sub(amountSpecified, state.amountSpecifiedRemaining)
        amount1.Set(state.amountCalculated)
} else {
        amount0.Set(state.amountCalculated)
        amount1.Sub(amountSpecified, state.amountSpecifiedRemaining)
}
```

Концепция позиций

Каждая позиция смартконтракта представляет собой виртуальную ликвидность, приращение комиссии на интервале, заданном верхним и нижним тиком, а также адрес владельца ликвидности. Хранение позиции осуществляется в мапе, где ключем выступает значение *keccak256*(*abi.encodePacked*(owner, tickLower, tickUpper)).

Количество токенов двух активов, необходимых для открытия позиции рассчитывается по формулам 6.29 и 6.30 [whitepaper-v3]. Эмиссия и сжигание токенов осуществляется посредством вызовов функций mint() и burn() ядра, при этом обновляется карта тиков, пересчитывается значение комиссий и величина ликвидности.

Заключение

Проведен анализ архитектуры Uniswap V3 и получены следующие результаты:

1) в протоколе UniswapV3 по отношению к V2 введены концепции тиков и позиций, в рамках которых ликвидность в пуле распределена не равномерно на всей числовой прямой, а в границах диапазона, заданного пользователем в рамках позиции. Таким образом, пул содержит множество как пересекающихся, так и не пересекающих позиций ликвидности, учет которых при операциях обмена, предоставления и снятия активов осуществляется с применением концепции тиков;
2) для оптимизации расчетов в части “тяжелых” математических операций, таких как процедуры взятия корня, они исключены из кода ядра, т.е., к примеру, все формулы whitepaper-v3 приведены к виду, когда на вход функций подается значение корня из цены. В коде смартконтрактов в хранилищах также представлено текущее значения корня из цены, а соответствующие значения цены для диапазонов приведены к значениям тиков и определены взаимно однозначные процедуры перевода, включая функцию преобразования тика в корень из цены на основе алгоритма быстрого возведения в степень. Требуемые параметры для инициализации пула, а также предельные значения корня из цены формируются на фронтенде (пример: https://docs.uniswap.org/sdk/guides/fetching-prices);
3) концепция тиков подразумевает разделение диапазона значений цены на равноотстоящие узлы, где между двумя отстоящими узлами сконцентрирована некоторая величина ликвидности, которая продставляет собой позицию, уникальную для каждого пользователя (провайдера ликвидности). Ликвидность для каждой позиции распределена равномерно между границами тиков позиции. При добавлении/удалении позиции осуществляется инициализация/деинициализация тиков. Заработанная комиссия хранится в виде распределенной на единицу ликвидности и пересчитывается при обновлении соответствующих тиков;
4) разработан пакет базовых операций с тиками, ликвидностью, ценой и осуществления операций обмена внутри одного пула. Архитектура пакета в большей степени сформирована в соответствии с архитектурой протокола с минимальными изменениями. Ревизии подвергся механизм маппинга тиков, а также математические операции смартконтракта, такие как MulDiv и пр., поскольку использованы разные типы представления целочисленных значений в коде на Solidity и Go.

Список использованных источников

1. **https://uniswap.org/whitepaper-v3.pdf**
2. **http://atiselsts.github.io/pdfs/uniswap-v3-liquidity-math.pdf**

3. https://github.com/Uniswap/v3-core
4. https://github.com/Uniswap/v3-periphery
5. https://github.com/Uniswap/v3-subgraph
6. https://docs.uniswap.org/
7. https://starli.medium.com/uniswap-deep-dive-into-v3s-source-code-b141c1754bae
8. https://starli.medium.com/uniswap-deep-dive-into-v3-technical-white-paper-2fe2b5c90d2
9. https://web.stanford.edu/~guillean/papers/uniswap_analysis.pdf
10. https://arxiv.org/pdf/2003.10001.pdf
11. http://reports-archive.adm.cs.cmu.edu/anon/2012/CMU-CS-12-123.pdf
12. https://jamesbachini.com/new-token/
13. https://github.com/atiselsts/uniswap-v3-liquidity-math/blob/master/subgraph-liquidity-range-example.py