深入理解 Uniswap v2 白皮书

tags: uniswap uniswap-v2 whitepaper

前言

本文作为《深入理解Uniswap》系列的第一篇，将从Uniswap v2白皮书入手，讲解Uniswap v2协议的设计思路和数学公式推导过程。

网络上讲解Uniswap的文章已经很多了，为什么要再写一遍呢?

最初原因是为了记录我个人在学习Uniswap过程中的总结，这些总结不是简单的翻译，更多是对于白皮书知识点的延伸阅读、数学公式的推导以及合约代码的工程实现的学习思考，而这些在原版白皮书大多没有展开。

虽然目前Uniswap v3已经推出一段时间了，但是学习v2仍然是理解V3的基础；并且由于v3的License限制 (https://uniswap.org/blog/uniswap-v3#license)，其他EVM链AMM项目大多fork v2代码，因此深入学习Uniswap v2仍然很有必要。

此外，Uniswap作为DeFi的基础协议，无论是行业地位，还是理论基础及其工程实现，都是DeFi的经典范例，对于想要深入学习DeFi或者智能合约编程的同学，Uniswap v2是非常好的入门材料。

希望本文能够帮助你在理解Uniswap v2的过程中提供一点帮助。由于本人水平有限，文中难免出现错误，欢迎斧正。

下文将按照Uniswap v2 白皮书 (https://uniswap.org/whitepaper.pdf)章节结构进行翻译，同时将重点知识的延伸阅读和数学公式推导过程以注释形式说明。

Uniswap v2 Core

1 Introduction 介绍

Uniswap v1是一个以太坊链上智能合约系统，实现了基于$x \cdot y = k$的AMM（自动做市）协议。每一个Uniswap v1交易对池子包含两种代币，在提供流动性的过程中保证两种代币余额的乘积无法减少。交易者为每次交易支付0.3%的手续费给流动性提供者。v1的合约不可升级。

Uniswap v2是基于同一个公式的新版实现，包含许多令人期待的新特性。其中最重要的一个特性是可以支持任意ERC20代币的交易对，而不是v1只支持ERC20与ETH的交易对。此外，v2提供了价格预言机功能，其原理是在每个区块开始时累计两种代币的相对价格。这将允许其他以太坊合约可以获取任意时间段内两种代币的时间加权平均价格；最后，v2还提供“闪电贷”功能，这将允许用户在链上自由借出并使用代币，只需在该交易的最后归还这些代币并支付一定手续费即可。

虽然v2的合约也是不可升级的，但是它支持在工厂合约中修改一个变量，以便允许Uniswap协议针对每笔交易收取0.05%的手续费（即0.3%的$\frac{1}{6}$）。该手续费默认关闭，但是可以在未来被打开，在打开后流动性提供者将只能获取0.25%手续费，而非0.3%。

> 注：因为其中0.05%分给协议。
>
> 关于0.05%协议手续费这个开关，后续引发了Sushiswap和Uniswap的流动性大战，Sushiswap fork Uniswap代码，将0.05%协议手续费分给SUSHI持有者(https://docs.sushi.com/products/yield-farming/the-sushibar#what-is-xsushi)，一度要将Uniswap流动性抢走；并最终迫使Uniswap发行了自己的代币UNI。

在第三节，我们将介绍Uniswap v2同时修复了Uniswap v1的一些小问题，同时重构了合约实现，通过最小化（持有流动性资金的）core合约逻辑，降低了Uniswap被攻击的风险，并使得系统更加容易升级。

本文讨论了core合约和用来初始化交易对合约的工厂合约的结构。实际上，使用Uniswap v2需要通过router（路由）合约调用交易对合约，它将帮助计算在交易和提供流动性时需要向交易对合约转账的代币数量。

2 New features 新特性

2.1 ERC-20 pairs ERC-20 交易对

Uniswap v1使用ETH作为桥接代币，任何交易对都包含ETH作为其中一个代币。这使得路由更加简单，比如要想实现ABC和XYZ的交易，只需要分别使用ETH/ABC和ETH/XYZ交易对即可，这同时也减少了流动性分裂。

> 注：由于交易对总是包含ETH，相比v2任意ERC-20代币组合，v1的交易对数量大大减少，并且流动性都被吸收到ETH这一侧。

但是这样的规则给流动性提供者带来巨大成本。所有的流动性提供者都将面临ETH的风险敞口，并且在代币价格相对ETH价格波动时承受无常损失。

> 注：由于$x \cdot y = k$引入的滑点，流动性提供者在Uniswap做市时将承受无常损失，简单而言就是在代币价格单方面（上涨或下跌）波动时，做市者手中持有的代币总价值反而减少。关于无常损失的说明，可以参考币安的这篇博客(https://academy.binance.com/en/articles/impermanent-loss-explained)。

如果ABC和XYZ是两种关联的代币，比如都是USD稳定币，那么交易对ABC/XYZ的无常损失将小于ABC/ETH或XYZ/ETH。

> 注：因为ABC与XYZ价格相对于ETH朝同一方向运动，ABC/XYZ价格波动较小，而ABC/ETH或XYZ/ETH价格波动较大。

使用ETH作为强制的交易代币也会增加交易成本。相比直接使用ABC/XYZ交易对，他们将支付两倍的交易手续费，同时承受两倍的滑点。

> 注：因为在v1，要想从ABC交易到XYZ，必须依次交易ABC/ETH和ETH/XYZ，因此手续费和滑点都需要两倍。

Uniswap v2允许流动性提供者为任意两个ERC-20代币创建交易对合约。

交易对数量的激增将给寻找最优交易路径带来困难，但是路由问题可以在上层解决（比如通过链下或链上的路由器或聚合器）。

2.2 Price Oracle 价格预言机

在时间点t由Uniswap提供的边际价格（不包含手续费）可以通过代币a和代币b的数量相除得出：

$$p_t = \frac{r_t^a}{r_t^b} \tag{1}$$

当Uniswap提供的价格不正确时，套利者可以在Uniswap交易套利（通过足够数量代币以支付手续费），因此Uniswap提供的代币价格将跟随市场价格。这意味着Uniswap提供的代币价格可以作为一种近似的价格预言机。

> 注：同一个代币在不同市场的价格差异提供了套利机会，驱使套利者维持Uniswap市场价格与其他市场（如中心化交易所或DEX）价格一致。

然而，Uniswap v1无法提供安全的链上预言机，因为它的价格很容易被操控。假设其他合约使用当前ETH-DAI价格作为衍生品交易的基准价格。攻击者可以从ETH-DAI交易对买入ETH来操控价格，并触发衍生品合约的清算，接着再将ETH卖回以使价格回归正常。上述操作可以通过一个原子交易完成，或者被矿工通过排序同一区块中的不同的交易来实现。

> 注：由于采样的价格是瞬时的，因此很容易通过买入卖出大额代币来操纵实时价格。
>
> samczsun有一篇博客 (https://samczsun.com/taking-undercollateralized-loans-for-fun-and-for-profit/)介绍了这种攻击。

Uniswap v2改进了预言机功能，通过在每个区块的第一笔交易前计算和记录价格来实现（等价于上一个区块的最后一笔交易之后）。操纵这个价格会比操纵区块中任意时间点的价格要困难。如果攻击者通过在区块的最后阶段提交一笔交易来操纵价格，其他套利者（发现价格差异后）可以在同一区块中提交另一笔交易来将价格恢复正常。矿工（或者支付了足够gas费用填充整个区块的攻击者）可以在区块的末尾操控价格，但是除非他们同时挖出了下一个区块，否则他们没有特殊的优势可以进行套利。

> 注：由于价格预言机仅在每个区块记录一次，因此除非同一个人控制了两个区块的所有交易，否则他们将没有足够的套利优势。但是这从另一方面说明，Uniswap v2的预言机仍然是不够健壮的。我们在v3可以看到这方面的改进。

Uniswap v2通过在每个区块第一笔交易前记录累计价格实现预言机。每个价格会以时间权重记录（基于当前区块与上一次更新价格的区块的时间差）。这意味着在任意时间点，该累计价格将是此合约历史上每秒的现货价格之和。

$$a_t = \sum_{i=1}^{t} p_i \tag{2}$$

为了估算在t_1到t_2时间段内的时间加权平均价格（TWAP），外部调用者可以分别记录t_1和t_2的累计价格，将t_2价格减去t_1价格，并除以$t_2 - t_1$的时间差（需注意，合约本身不存储历史的累计价格，因此需要调用者在区间开始时调用合约，读取并保存当前的价格）。

$$p_{t_1,t_2} = \frac{\sum_{i=t_1}^{t_2} p_i}{t_2 - t_1} = \frac{\sum_{i=1}^{t_2} p_i - \sum_{i=1}^{t_1} p_i}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1} \tag{3}$$

预言机的用户可以自行选择区间的开始和结束。选择一个更长的区间，意味着攻击者将花费更高的代价来操控该区间的时间加权平均价格，虽然这将导致该平均价格与实时价格相差较大。

注：公式(3)比较容易理解，这里就不展开。但是需注意，由于合约仅记录当前的累计价格，因此如果需要计算区间的平均价格，外部应用要自己记录并保存历史价格，合约本身不保存历史数据。

Uniswap v2的TWAP计算方式实际上使用的是（加权）算数平均数（Arithmetic Mean），这里我们需要了解几种平均数的概念和应用场景。

在数学上有一个毕达哥拉斯平均的概念，指的是三种经典平均数，分别是：算数平均数、几何平均数和调和平均数。

- 算数平均数 Arithematic Mean

$$A(x_1,\ldots,x_n)=\frac{1}{n}(x_1+\ldots+x_n)$$

- 几何平均数 Geometric Mean

$$G(x_1,\ldots,x_n)=\sqrt[n]{x_1\ldots x_n}$$

- 调和平均数 Harmonic Mean

$$H(x_1,\ldots,x_n)=\frac{n}{\frac{1}{x_1}+\ldots+\frac{1}{x_n}}$$

- 当 x 为正数时，三者的关系：

$$A(x_1,\ldots,x_n)\geq G(x_1,\ldots,x_n)\geq H(x_1,\ldots,x_n)$$

其中，算术平均数是最常见的一种平均数，其优点是计算简单，缺点是容易受到极端数据的影响，导致均值误差；几何平均数相比算术平均数，更适用于在金融市场场景，因为金融市场价格本身是一种布朗运动；调和平均数更易受到极小值的影响，一般应用于计算平均速率等场景。

从应用场景上，Uniswap价格均值应该使用几何平均数更合适，均值的误差更小，但由于几何平均数在以太坊合约上实现难度较大，所以Uniswap v2版本采用算数平均数；但是Uniswap v3则使用几何平均数计算价格预言机。

在3.4节，Uniswap v2使用几何平均数计算初始流动性代币数量。

一个难题：我们应该计算以B代币计价的A代币价格，还是以A代币计价的B代币价格？虽然在现货价格上，以B代币计价的A代币价格（B/A）与以A代币计价的B代币价格（A/B）总是互为倒数，但在计算某个时间区间的算数平均数时，二者却不是互为倒数关系。比如，假设在区块1的价格为

100 USD/ETH（B为USD，A为ETH），区块2的价格为300 USD/ETH，则其平均价格为200 USD/ETH，但ETH/USD的平均价格却是1/150 ETH/USD。因为合约无法知道交易对中哪一个代币将被用户用作计价单位，因此Uniswap v2同时记录了两个代币的价格。

> 注：两种代币计价的均值计算如下：
>
> - 以 USD 计价
>
> $$A(x_1, x_2) = \frac{100 + 300}{2} = 200 \text{ USD/ETH}$$
>
> - 以 ETH 计价
>
> $$A(\frac{1}{x_1}, \frac{1}{x_2}) = \frac{\frac{1}{100} + \frac{1}{300}}{2} = \frac{1}{150} \text{ ETH/USD}$$

另一个难题是用户可以不通过交易而直接向交易对合约发送代币（这将改变代币余额并影响价格），此时将无法触发预言机价格更新。

> 注：因为预言机价格需要在区块的第一笔交易之前更新，因此如果不交易，将绕开预言机更新。

如果合约只是简单地检查它的余额，并使用当前余额计算价格来更新预言机，那么攻击者可以在区块的第一笔交易之前，立即向合约发送代币来操控预言机价格。如果上一笔交易是在x秒之前的某个区块，合约将错误的使用（被操纵后的）新价格乘以x来累计，即使并没有人使用该价格交易过。

> 注：假设在上一个区块最后一笔交易后，交易对合约中两个代币A、B的余额分别为100、200，以A计价的B价格为200/100=2，在x秒后，下一个区块第一笔交易发生之前，应该累计的价格是$2x$，但是如果在第一笔交易发生之前，攻击者向合约发送了100个B，此时价格为200/200=1，合约将错误地以$1x$累计。

为了防止这个问题，core合约在每次交互后缓存了两种代币余额，并且使用缓存余额（而非实时余额）更新预言机价格。除了防止预言机价格被操控外，这个改动也带来了合约架构的重新设计，我们将在3.2节进行说明。

2.2.1 Precision 精度

因为Solidity原生不支持非整数数据类型，Uniswap v2使用了简单的二进制定点制进行编码和操作价格。确切地说，任意时间的价格都被保存为UQ112.112格式的数据，它表示在小数点的左右两边都有112位比特表示精度，无符号（注：非负数）。这个格式能表示的范围为$[0, 2^{112} - 1]$，精度为$\frac{1}{2^{112}}$。

> 注：UQ112.112的理论最大值为：$2^{112} - \frac{1}{2^{112}}$，但由于Uniswap v2使用两个uint112相除得到UQ112.112，因此其最大值为$2^{112} - 1$。

选择UQ112.112格式是出于（Solidty合约）编程实践的考虑，因为这些格式的数字能够使用一个uint224类型（占用224位比特，28个字节）的变量表示，在一个256位（比特）的存储槽（注：EVM中一个Storage Slot是256位）中正好剩余32位可用。而对于缓存的代币余额变量，每一个代币余额可以使用一个uint112类型（112比特位，14个字节）的变量，（在声明时）也正好在256位的存储槽中剩余32位可用。这些剩余空间可用于上述的累计运算使用。具体来说，代币余额与最近一个有交易区块的时间戳一起保存，该时间戳针对2^{32}取模，以确保可以使用32位表示。此外，虽然在任意时间点的价格（使用UQ112.112格式的数字）一定符合224位，但是一段时间的累计价格却不是这样。在存储槽末尾的多余32位空间将用于存储由于重复累计价格导致的溢出数据。这样的设计意味着价格预言机仅仅在每个区块的第一笔交易增加了3个SSTORE操作（当前消耗15,000 gas）。

> 注：这里我们可以看到Uniswap在设计上是非常小心的，价格预言机虽然是一个很有用的功能，但是却不能因为过度设计而给用户增加成本，因此如何在确保对用户影响最小的同时把功能实现，就成为设计的核心所在。为了避免每次交易都更新预言机给用户带来额外交易成本，Uniswap v2才设计成只在每个区块的第一笔交易之前更新。
>
> 每个代币余额使用uint112表示，时间戳使用32位表示，总共112+112+32=256位，正好占用一个storage slot。更少的storage slot意味着交互时需要花费的gas更小，有利于减少用户操作成本。累计价格则采用256位表示。

这个设计最主要的缺点是32位无法确保时间戳永不溢出。事实上，Unix时间戳溢出32位（可表示的最大值）将发生在02/07/2106。为了确保系统能够在该时间后正常工作，同时在每一轮32位溢出后（$2^{32}-1$秒）也能正常工作，预言机需要至少在每一轮（大约136年）被调用查询一次。因为累计计算的核心方法是溢出安全的，这意味着即使交易跨越了时间戳溢出的时间点，它也是可以被正常累计的，只要预言机使用了正确的溢出算法来检查时间间隔。

注：这里我们主要关注在时间戳溢出边界，使用公式（3）是否能够正确算出预言机价格的平均数。假设在2106年2月7日附近，t_1, t_2, t_3分别表示三个连续的区块时间，其中t_1未发生时间戳溢出（差1秒），而t_2,t_3则发生溢出，可以算出即使在溢出后，$t_2 - t_1$仍然可以计算出正确的时间差（3秒）；同理可以计算即使当累计价格a_{t_3}发生溢出后，只要调用者保存了a_{t_1}的值，即可计算出二者正确的差值。p_{t_1,t_3}为t_1到t_3时间区间的平均价格，按照公式（3）可推出如下计算：

$$uint32.max = 2^{32} - 1 = 4,294,967,295$$

$$t_1 = 4,294,967,294$$

$$t_2 = 4,294,967,297 \% 4,294,967,296 = 1$$

$$t_3 = 4,294,967,301 \% 4,294,967,296 = 5$$

$$\Delta t_1, t_2 = 1 - 4,294,967,294 = -4,294,967,293 = 3$$

$$\Delta t_2, t_3 = 5 - 1 = 4$$

$$\Delta t_1, t_3 = 5 - 4,294,967,294 = -4,294,967,289 = 7$$

$$p_{t_1} = 100$$

$$p_{t_2} = 110$$

$$a_{t_1} = 1000$$

$$a_{t_2} = a_{t_1} + p_{t_1} * \Delta t_1, t_2 = 1000 + 100 * 3 = 1300$$

$$a_{t_3} = a_{t_2} + p_{t_2} * \Delta t_2, t_3 = 1300 + 110 * 4 = 1740$$

$$p_{t_1,t_3} = \frac{a_{t_3} - a_{t_1}}{t_3 - t_1} = \frac{\Delta a_{t_1}, a_{t_3}}{\Delta t_1, t_3} = \frac{740}{7} = 105.71$$

2.3 Flash Swaps 闪电贷

在Uniswap v1，用户如果想使用XYZ购买ABC，则需要先将XYZ发送到合约才能收到ABC。这将给那些希望使用ABC购买XYZ的用户带来不便。比如，当Uniswap与其他合约出现套利机会时，用户可能希望使用ABC在别的合约购买XYZ；或者用户希望通过卖出抵押物来释放他们在Maker或Compound的头寸，以此偿还Uniswap的借款。

Uniswap v2增加了一个新特性，允许用户在支付费用前先收到并使用代币，只要他们在同一个交易中完成支付。swap方法会在转出代币和检查k值两个步骤之间，调用一个可选的用户指定的回调合约。一旦回调完成，Uniswap合约会检查当前代币余额，并且确认其满足k值条件（在扣除手续费后）。如果当前合约没有足够的余额，整个交易将被回滚。

用户可以只归还原始代币，而不需要执行交易操作。这个功能将使得任何人可以闪电借出Uniswap池子中的任意数量的代币（闪电贷手续费与交易手续费一致，都是0.30%）。

> 注：闪电贷在DeFi领域非常实用，对于TVL较高的协议，协议可以通过闪电贷获取手续费收入。比如dYdX和Aave，都推出了闪电贷功能。Uniswap v2合约中的闪电贷与交易功能实际上使用同一个swap方法。

2.4 Protocol fee 协议手续费

Uniswap v2包含一个0.05%的协议手续费开关。如果打开，该手续费将被发送到合约中的feeTo地址。

默认情况下没有设置feeTo地址，因此不收取协议手续费。预定义的feeToSetter地址可以调用Uniswap v2工厂合约中的setFeeTo方法来修改feeTo地址。feeToSetter也可以调用setFeeToSetter修改合约中feeToSetter地址。

如果feeTo地址被设置了，协议将开始收取5个基点（0.05%）的手续费，也就是流动性提供者收取的30个基点（0.30%）手续费中的1/6将分配给协议。这意味着交易者将继续为每一笔交易支付0.30%的交易手续费，83.3%（5/6）的手续费（整笔交易的0.25%）将分配给流动性提供者，剩余的16.6%（手续费的1/6，整笔交易的0.05%）将分配给feeTo地址。

如果在每笔交易时收取0.05%的手续费，将带来额外的gas消耗。为了避免这个问题，累计的手续费只在提供或销毁流动性时收取。合约计算累计手续费，并且在流动性代币铸造或销毁的时候，为手续费受益者（feeTo地址）铸造新的流动性代币。

总累计手续费可以通过计算从上次收取手续费后，以$\sqrt{k}$（也就是$\sqrt{x \cdot y}$）计价的增长量。可计算从t_1到t_2的累计手续费，与t_2时刻的流动性的百分比如下：

$$f_{1,2} = 1 - \frac{\sqrt{k_1}}{\sqrt{k_2}} \tag{4}$$

注：这里不太好理解，为什么手续费是以$\sqrt{x \cdot y}$的形式给出的呢？如果看完白皮书【3.4 初始化流动性代币供应】就明白了。第一次流动性铸造的代币数量是以$\sqrt{x \cdot y}$算出的，在t_1,t_2不同时刻，（不考虑mint/burn流动性时）其流动性代币数量始终等于$\sqrt{x_1 \cdot y_1}$与$\sqrt{x_2 \cdot y_2}$，其增长部分即为手续费，因此公式（4）可按照如下推导得出：

$l_1 = \sqrt{x_1 \cdot y_1}$

$l_2 = \sqrt{x_2 \cdot y_2}$

$fee = l_2 - l_1$

$$f_{1,2} = \frac{l_2 - l_1}{l_2} = 1 - \frac{\sqrt{x_1 \cdot y_1}}{\sqrt{x_2 \cdot y_2}} = 1 - \frac{\sqrt{k_1}}{\sqrt{k_2}}$$

那么当mint/burn流动性时，如何计算手续费呢？实际上在每次mint/burn流动性之前，都会先结算未领取的累计手续费，所以在mint/burn之后，可以重新按照上面的公式计算手续费。因此上述公式只需关注仅当发生swap交易时，累计手续费如何计算这一问题。

如果协议手续费在t_1之前被激活，那么在t_1, t_2时段，feeTo地址应该收取$\frac{1}{6}$的手续费作为协议手续费。因此，我们需要为feeTo地址铸造新的流动性代币以代表该时段手续费，这里等于$\frac{1}{6} \cdot f_{1,2}$。

假设协议手续费对应的流动性代币数量为s_m，s_1为t_1时刻的流动性代币数量，则有以下等式：

$$\frac{s_m}{s_m + s_1} = \phi \cdot f_{1,2} \tag{5}$$

使用公式（4）替换$f_{1,2}$，经过计算可以得出s_m为：

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{(\frac{1}{\phi} - 1) \cdot \sqrt{k_2} + \sqrt{k_1}} \cdot s_1 \tag{6}$$

使用$\frac{1}{6}$替换其中的比例部分，可得：

$$s_m = \frac{\sqrt{k_2} - \sqrt{k_1}}{5 \cdot \sqrt{k_2} + \sqrt{k_1}} \cdot s_1 \tag{7}$$

假设初始流动性提供者存入100 DAI和1 ETH，获得10个流动性代币。一段时间后（假设没有其他流动性提供者），当feeTo希望取出协议手续费时，两种代币余额分别为96 DAI和1.5 ETH。分别代入公式（7）可得：

$$s_m = \frac{\sqrt{1.5 \cdot 96} - \sqrt{1 \cdot 100}}{5 \cdot \sqrt{1.5 \cdot 96} + \sqrt{1 \cdot 100}} \cdot 10 \approx 0.0286 \tag{8}$$

> 注：当没有mint/burn流动性时，只是单纯swap，池子的k值是不断变大的，原因就在于手续费沉淀，因为此时流动性代币总量（shares）不变，但交易对池子中两种代币余额不断增加。如上所述，
> $k_1 = 100DAI \cdot 1ETH = 100$，$k_2 = 96DAI \cdot 1.5ETH = 144$，$k_2 > k_1$。

2.5 Meta transactions for pool shares 元交易

Uniswap v2的池子份额（即流动性代币）天然支持元交易。这意味着用户可以通过签名授权第三方转移其持有的流动性代币，而无需通过他们自己发起链上交易。任何人都可以通过调用permit方法来代替该用户提交签名，支付gas费用，并且可以在同一交易中执行其他操作。

> 注：这里的元交易实际上指的是通过离线签名方式，由第三方代替用户发起链上交易。在某些场景下很实用，比如用户的钱包没有ETH，可以由第三方代付gas。
>
> 在Uniswap v2 core合约中的签名功能是授权转账流动性代币；这个签名是在外围的router合约中使用，因为v2将合约分为core（最核心的swap/mint/burn功能）和periphery（外围应用）合约，而应用一般直接调用periphery合约，通过签名方式可以减少用户与core合约的链上交互，只需使用离线签名与periphery合约交互一次即可移除流动性。签名也便于其他合约与core合约集成。
>
> 这里涉及两个EIP，分别是EIP-712 (https://eips.ethereum.org/EIPS/eip-712)与EIP-2612 (https://eips.ethereum.org/EIPS/eip-2612)，我们在另外的文章再具体说明这两个EIP。简单而言，EIP-712定义了针对结构数据的签名方式，在以前只能针对一串hash签名，实际上我们并不知道签名的内容是什么，容易引发安全问题，比如误将代币授权给恶意合约；通过EIP-712，我们可以在签名时检查具体的签名内容，如授权转账的额度，截止时间等信息。（但从实际使用上，大部分用户仍然并不知道自己签名会带来什么影响）。EIP-2612则是关于使用EIP-712的permit方法的Solidity编码规范，该提案是在Uniswap v2以后才出来的，目前仍处于Review阶段。

3 Other changes 其他改动

3.1 Solidity

Uniswap v1使用Vyper语言实现，这是一个类Python的智能合约语言。Uniswap v2使用更流行的Solidity语言实现，因为v2依赖一些（在开发时）Vyper语言还不具有的能力，比如解析非标准ERC-20代币的返回值，通过内联的assembly语法访问一些新操作码，如chainid。

3.2 Contract re-architecture 合约重构

Uniswap v2的一个设计重点在于最小化core交易对合约的对外接口范围和复杂度（core合约存放流动性提供者的代币资产）。在core合约上发现的任何bug都可能是灾难性的，因为这可能会导致数百万美元的流动性资产被盗走或冻结。

在评估core合约的安全性时，最重要的问题是它是否能保护流动性提供者的资产不被盗走或冻结。任何增强或保护交易者的功能特性，而不是允许池子里资产交换这种最基本的功能，都应该被抽取放到router（路由）合约。

> core合约仅保留最基础最重要的功能，以保证安全性，因为所有流动性资产将存放在core合约中。代码越少，改动越小，出现bug的概率也越小。实际上core合约的核心代码只有200行左右。

事实上，甚至部分swap功能的代码也可以被提到router合约中。如前所述，Uniswap v2保存每种代币最后的余额记录（为了防止攻击者操纵预言机机制）。新的架构在此基础上针对Uniswap v1做了进一步简化。

在Uniswap v2，卖方在执行swap方法前，会发送代币到core合约。合约将通过比较缓存余额和当前余额来判断收到多少代币。这意味着core合约无法知道交易者是通过什么方式发送代币。事实上，他可以通过离线签名的元交易方式，或者其他未来授权ERC-20代币转移的机制，而不只是transferFrom方法。

3.2.1 Adjustment for fee 手续费调整

Uniswap v1的交易手续费是通过减少存入合约的代币数量来实现，在比较k常值函数之前，需要先减去0.3%的交易手续费。合约隐式约束如下：

$$(x_1 - 0.003 \cdot x_{in}) \cdot y_1 \geq x_0 \cdot y_0 \tag{9}$$

> 注：扣除手续费以后的两种代币余额，符合k常值函数。

通过闪电贷功能，Uniswap v2引入了一种可能性，即xin和yin可能同时不为0（当一个用户希望通过归还借出的代币，而不是做交易时）。为了处理这种情况下的手续费问题，合约强制要求如下约束：

$$(x_1 - 0.003 \cdot x_{in}) \cdot (y_1 - 0.003 \cdot y_{in}) \geq x_0 \cdot y_0 \tag{10}$$

> 注：Uniswap的swap方法可以同时支持闪电贷和交易功能，当通过闪电贷同时借出x和y两种代币时，需要分别对x和y收取0.3%的手续费，因此需要先扣除手续费，再保证余额满足k值约束。

为了简化链上计算，我们可以为公式（10）两边同时乘以1,000,000，得出：

$$(1000 \cdot x_1 - 3 \cdot x_{in}) \cdot (1000 \cdot y_1 - 3 \cdot y_{in}) \geq 1000000 \cdot x_0 \cdot y_0 \tag{11}$$

> 因为Solidity不支持浮点数，因此通过同步放大来简化计算。

3.2.2 sync() 和 skim()

为了防止某些可以修改交易对合约余额的定制代币，同时也为了更优雅地解决那些总量超过2^{112}的代币，Uniswap v2提供了两个方法：sync()和skim()。

当某种代币异步通缩时，sync()可以作为一种恢复手段。在这种场景下，交易将获得次优的兑换率，如果没有流动性提供者愿意纠正这种状态，交易对将难以继续工作。sync()方法可以将合约中缓存的代币余额设置为当前实际余额，以帮助系统从这种状态中恢复。

当发送大量代币导致交易对的代币余额溢出（超过uint112最大值）时，交易将失败，skim()可以作为这种情况的恢复手段。skim()允许任意用户取出多余的代币（代币实际余额与$2^{112}-1$的差值）。

> 简单来说，由于某些（非Uniswap导致的）外部因素，交易对合约中的缓存余额与实际余额可能出现算法外的不一致问题。sync()方法可以更新缓存余额到实际余额，skim()方法可以更新实际余额到缓存余额，从而保证系统继续运行。任何人都可以执行这两个方法。Alpha Leak：如果有人误将交易对中的代币转入合约，任何人都可以取出这些代币。

3.3 Handling non-standard and unusual tokens 处理非标准和罕见代币

ERC-20标准要求transfer()和transferFrom()返回一个布尔值表示该请求是否成功。然而某些代币在实现这两个（或其中一个）方法时并没有返回值，比如USDT和BNB。Uniswap v1在解析无返回值的方法时，将其当作失败处理，因此将回滚交易，从而导致交易失败。

> 扩展阅读： EIP-20: ERC-20代币标准 (https://eips.ethereum.org/EIPS/eip-20) USDT合约地址 (https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code) BNB合约地址 (https://etherscan.io/address/0xB8c77482e45F1F44dE1745F52C74426C631bDD52#code)

Uniswap v2针对非标准ERC-20代币的实现，则使用不一样的处理方法。当transfer()方法没有返回值时，Uniswap v2认为它表示执行成功，而非失败。这个改动不会影响任何实现标准ERC-20的代币（因为他们的transfer()方法有返回值）。

同样，Uniswap v1假设transfer()和transferFrom()不能触发重入交易对合约的方法。这种假设会和某些ERC-20代币冲突，包括那些支持ERC-777标准hooks的代币。为了完全支持这些代币，Uniswap v2引入了"lock"机制用来解决所有公开修改状态方法的重入问题。这也可以防止在闪电贷中用户自定义回调的重入问题。

> 注：lock实际上是一个Solidity modifer，通过一个unlock变量控制同步锁。

3.4 Initialization of liquidity token supply 初始化流动性代币供应

当一个新的流动性提供者将代币存入一个已存在的Uniswap交易对，新铸造的流动性代币数量可根据当前代币数量计算：

$$s_{minted} = \frac{x_{deposited}}{x_{starting}} \cdot s_{starting} \tag{12}$$

> 注：因为流动性代币本身是一种ERC-20代币，持有流动性代币数量即表示占有该池子代币的份额（shares）。因此对于已存在的交易对，即已经有该交易对的流动性代币存在，那么存入的代币价值与总价值的比例，与其得到的流动性代币数量与总数量的比例应相等：
>
> $$\frac{s_{minted}}{s_{starting}} = \frac{x_{deposited}}{x_{starting}}$$

但如果他们是第一个流动性提供者呢？在这种情况下，$x_{starting}$是0，因此上述公式无法适用。

Uniswap v1将首次流动性代币数量等同于存入的ETH数量（以wei为单位）。这有一定的合理性，因为如果首次流动性是以正确的价格存入的，那么1个流动性份额（如ETH是一种有18位小数的代币）将代表大约2ETH的价值。

> 注：因为Uniswap v1/v2提供流动性时需要注入两边等值的代币，如果份额等同于ETH数量，则1份额表示需要存入1ETH，而在价格正确时，另一个代币的价值也同样是1ETH，因此1个流动性份额的流动性总价值是2ETH。

然而，这意味着流动性份额的价值需要依赖首次注入流动性时的价格比例，而这个价格是可以被认为控制的，我们无法保证首次注入流动性时的两种代币的比例能够正确反映真实价格。此外，由于Uniswap v2支持任意代币的交易对，因此将有更多的交易对不包含ETH。

与v1不同，Uniswap v2规定首次铸造流动性代币的数量等于存入的两种代币数量的几何平均数：

$$s_{minted} = \sqrt{x_{deposited} \cdot y_{deposited}} \tag{13}$$

该公式确保在任意时刻，流动性份额的价值与其存入代币的价格比例无关。比如，假设当前1 ABC的价格是100 XYZ，如果首次存入2 ABC和200 XYZ（对应的比例为1:100），则流动性提供者将收到$\sqrt{2 \cdot 200} = 20$个流动性代币。这些代币价值2 ABC和200 XYZ，以及对应的累计手续费。

如果首次存入2 ABC和800 XYZ（对应比例1:400），则流动性提供者将收到$\sqrt{2 \cdot 800} = 40$个流动性代币。

以上公式确保1个流动性份额（代币）的价值将不少于池子中两种代币余额的几何平均数。然后，1个流动性代币的价值将可能随着时间持续增长，比如通过累计交易手续费，或者通过其他人“捐赠”代币到池子里。

理论上可能存在这种情况，最小的流动性代币单位（10^{-18}，即1 wei）的价值太高，以至于无法让其他（小）流动性提供者加入。

为了解决这个问题，Uniswap v2销毁首次铸造10^{-15}（最小代币单位的1000倍）流动性代币。这个损耗对于大部分交易对而言都是微不足道的。但是这将极大提到首次铸币攻击的代价。为了将每个流动性代币价格提高到100美元，攻击者需要捐赠10万美元的代币到池子中，这些代币将被作为流动性而永久锁定。