Uniswap 白皮书（中文版）

25 SEPTEMBER 2021 on uniswap

介绍

Uniswap 是一个基于以太坊的自动代币交易协议。它的设计目标是更易用，gas 高利用率，限制审查和无手续费抽成。它对交易者很有用，部分功能也作为组件适用于那些需要保证链上资产资产流动性的智能合约。

多数交易所维护一个交易委托账本来帮助撮合买卖双方。Uniswap 智能合约持有各种代币的流动性准备金，用户会直接跟这些准备金进行对手交易。价格会使用恒定产品(`x*y=k`)做市商机制自动设定**，它会保证整体准备金的相对平衡。流动性提供者组成一个网络汇集准备金，他们向系统提供交易代币从而获取一定比例手续费份额。

Uniswap 的一个重要特性是利用一个工厂-注册合约来为每个 ERC20 代币部署一个独立的交易合约。这些交易合约同时持有 ETH 和他们关联的 ERC20 代币构成的准备金。这可以实现两个基于相关供应的交易对之间的交易。交易合约被注册串联在一起，从而可以以 ETH 作为媒介实现 ERC20 代币之间的互相交易。

这篇文档列出了 Uniswap 的核心结构和技术细节。一些代码为了可读性做了简化。诸如溢出检查和最低购买限额之类的安全特性被忽略了。完整的源代码可以在 GitHub 上找到。

协议网站: uniswap.io

文档: docs.uniswap.io

代码: ** github.com/Uniswap

形式模型: https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf

Gas 基准测试

得益于最小化设计，Uniswap 的 gas 利用率很高。对于 ETH 对 ERC20 代币的兑换，它的 消耗的 gas 是 Bancor 的十分之一。它在实现 ERC20 代币之间交易效率比 0x 更高，对比链上交易委托账本的交易所，例如 EtherDelta 和 IDEX， 显著的减少了 gas 消耗。

Exchange	Uniswap	EtherDelta	Bancor	Radar Relay (0x)	IDEX	Airswap
ETH to ERC20	46,000	108,000	440,000	113,000*	143,000	90,000
ERC20 to ETH	60,000	93,000	403,000	113,000*	143,000	120,000*
ERC20 to ERC20	88,000	no	538,000	113,000	no	no

*wrapped ETH

直接 ERC20 代币转账的费用是36,000 gas - 比在 Uniswap 中用 ETH 兑换 ERC20 代币的费用少20%左右。🔥🔥🔥

创建交易所

`uniswap_factory.vy` 这个智能合约是Uniswap 交易所的工厂和注册表。公共函数 `createExchange()` 可以让以太坊用户为任意尚未注册的 ERC20 代币部署一个交易所合约。

```
exchangeTemplate: public(address)
token_to_exchange: address[address]
exchange_to_token: address[address]

@public
def __init__(template: address):
    self.exchangeTemplate = template

@public
def createExchange(token: address) -> address:
    assert self.token_to_exchange[token] == ZERO_ADDRESS
    new_exchange: address =
create_with_code_of(self.exchangeTemplate)
    self.token_to_exchange[token] = new_exchange
    self.exchange_to_token[new_exchange] = token
    return new_exchange
```

所有代币和它们相关交易所的记录都被存储在工厂中。通过提交代币或者交易所地址，使用函数 `getExchange()` 或 `getToken()`来查询另一个信息。

```
@public
@constant
def getExchange(token: address) -> address:
    return self.token_to_exchange[token]

@public
@constant
def getToken(exchange: address) -> address:
    return self.exchange_to_token[exchange]
```

载入一个交易所合约的时候，工厂不会对代币做任何检查，尤其是强制交易所和代币一一对应。用户和前端应当跟他们所信任代币的交易所进行交互。

ETH ⇄ ERC20 交易

每个交易所合约(`uniswap_exchange.vy`)都会跟一个 ERC20 代币关联，并且带有一个ETH和这个代币的流动性池。ETH 和 ERC20 之前的兑换比率给予它们在合约中的流动性池的相对大小。这通过维持 `eth_pool（ETH 数量） * token_pool（代币数量） = invariant（不变量）` 这个关系来实现。 这个不变量在交易中保持恒定，只有在流动性被添加到市场或者从市场移除的时候才会变化。

以下代码是 `ethToTokenSwap()`这个函数转换 ETH 到 ERC20 代币的简单实现:

```
eth_pool: uint256
token_pool: uint256
token: address(ERC20)

@public
@payable
def ethToTokenSwap():
    fee: uint256 = msg.value / 500
    invariant: uint256 = self.eth_pool * self.token_pool
    new_eth_pool: uint256 = self.eth_pool + msg.value
    new_token_pool: uint256 = invariant / (new_eth_pool
- fee)
    tokens_out: uint256 = self.token_pool -
new_token_pool
    self.eth_pool = new_eth_pool
```

```
    self.token_pool = new_token_pool
    self.token.transfer(msg.sender, tokens_out)
```

:::info **备注：** 为了 gas 效率，`eth_pool` 和 `token_pool` 不是被存储的变量。他们可以用 `self.balance` 来获取，然后通过外部调用 `self.token.balanceOf(self)` 来实现。 :::

发送 ETH 到这个函数的时候，`eth_pool` 会增加。为了维持 `eth_pool（ETH 数量） * token_pool（代币数量） = invariant（不变量）` 这个关系，`token pool` 将会减少相应比例。`token_pool` 减少的数量就是用户交易得到的数量。准备金比率的改变会导致 ETH 对 ERC20 兑换率的改变，这会刺激反向交易。

使用 `tokenToEthSwap()` 兑换代币到 ETH：

```
@public
def tokenToEthSwap(tokens_in: uint256):
    fee: uint256 = tokens_in / 500
    invariant: uint256 = self.eth_pool * self.token_pool
    new_token_pool: uint256 = self.token_pool +
tokens_in
    new_eth_pool: uint256 = self.invariant /
(new_token_pool - fee)
    eth_out: uint256 = self.eth_pool - new_eth_pool
    self.eth_pool = new_eth_pool
    self.token_pool = new_token_pool
    self.token.transferFrom(msg.sender, self,
tokens_out)
    send(msg.sender, eth_out)
```

`token_pool`增加，`eth_pool`减少，导致价格反向移动。下面示例展示 ETH 如何兑换 OMG 代币。

示例: ETH → OMG

10 ETH 和 500 OMG（ERC20）被流动性提供者投入到智能合约中。不变量将通过 `eth_pool（ETH 数量） * OMG_pool（代币数量） = invariant（不变量）` 这个公式自动设置。

> *ETH_pool（ETH 资金池） = 10*
>
> *OMG_pool（OMG 资金池） = 500*
>
> *invariant（不变量） = 10 * 500 = 5000*

OMG 购买者发送 1 ETH 到这个合约。其中 0.25% 的手续费会被划分给流动性提供者，剩余的 0.9975 ETH 会被放入 `ETH_pool`。不变量除以流动性池里 ETH 新增后的数量，得到新的 `OMG_pool` 的代币数量。剩下来的 OMG 会转给购买者。

> *购买者发送: 1 ETH*
>
> *Fee（费用） = 1 ETH / 500 = 0.0025 ETH*
>
> *ETH_pool（ETH 资金池） = 10 + 1 - 0.0025 = 10.9975*
>
> *OMG_pool（OMG 资金池） = 5000/10.9975 = 454.65*
>
> *购买者收到: 500 - 454.65 = 45.35 OMG*

手续费会被放回流动性资金池池里，它会作为从市场中移除流动性的费用支付给流动性提供者。由于手续费会在价格计算后被添加到资金池中，所以不变量会随着每次交易稍微变大，为流动性提供者提供系统性

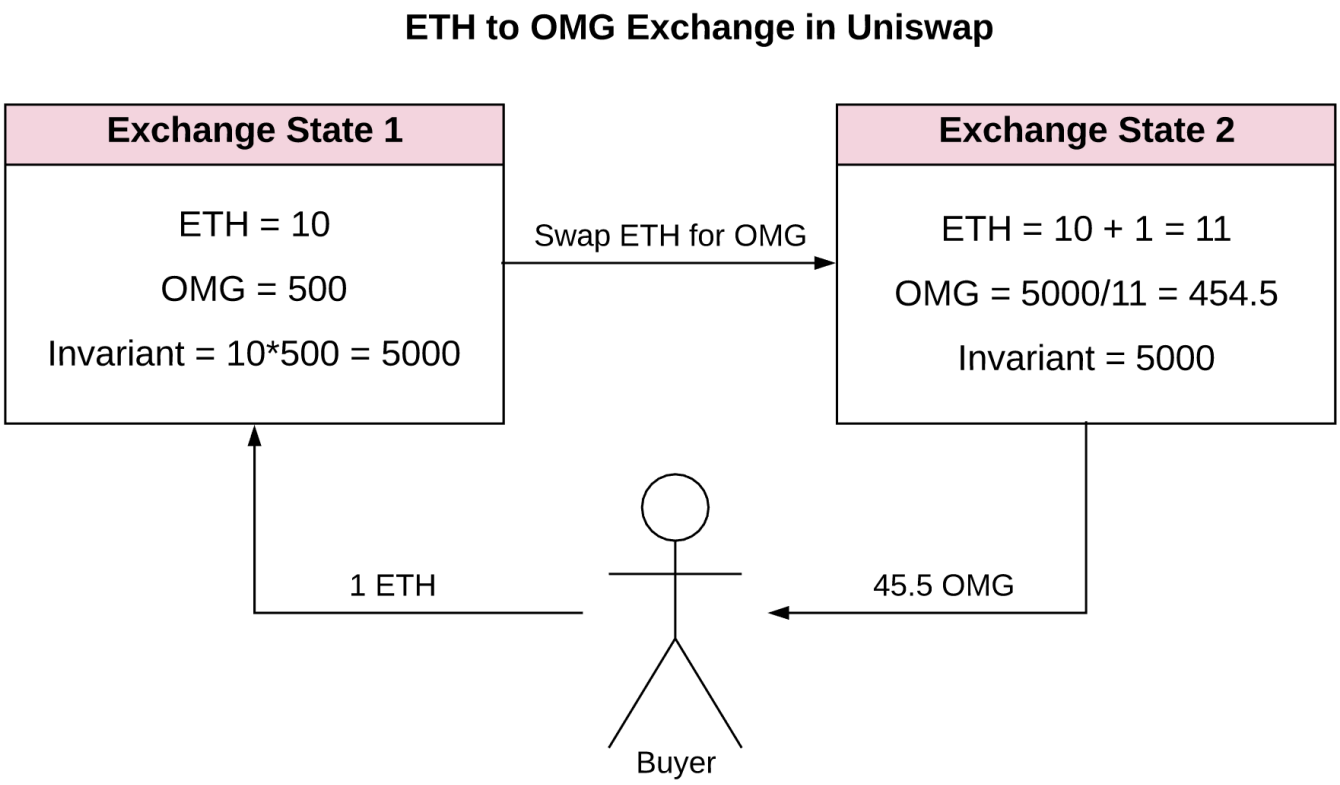
盈利。故此，不变量表示的是上次交易结束后`ETH_pool（ETH 资金池） * OMG_pool（OMG 资金池）`的值。

> *ETH_pool（ETH 资金池） = 10.9975 + 0.0025 = 11*
>
> *OMG_pool（OMG 资金池） = 454.65*
>
> *new invariant（新的不变量） = 11 * 454.65 = 5,001.15*

在这个示例中，购买者接受的兑换比率是 45.35 OMG / 1 ETH。然而价格已经发生了变动。如果另一个购买者做了同样方向的交易，他将会得到一个稍微差一些的 OMG/ETH 兑换比率。可是如果一个购买者做了一个反向的交易，他则会得到一个稍微好一些的 ETH/OMG 兑换比率。

> *1 ETH 转入 44.5 OMG 转出 兑换比率 = 45.35 OMG/ETH*

相对于流动性资金池较大规模的购买会导致价格明显下滑。不过在活跃的市场中，套利操作会确保价格不会和其他交易所偏移太多。



ERC20 ⇄ ERC20 交易

由于 ETH 被用作所有 ERC20 代币的通用交易对，所以它可以当作 ERC20 代币之间直接交易的媒介。例如，可以把 OMG 兑换成 ETH 这个交易和下面 ETH 兑换 KNC 的交易放在一个事务中。

比如：为了把 OMG 兑换成 KNC，购买者可以调用 OMG 交易所合约的 `tokenToTokenSwap()` 函数：

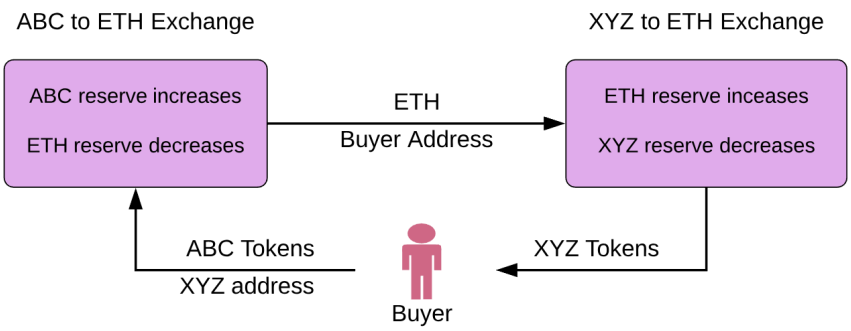
```
contract Factory():
    def getExchange(token_addr: address) -> address:
constant

contract Exchange():
    def ethToTokenTransfer(recipent: address) -> bool:
modifying

factory: Factory

@public
def tokenToTokenSwap(token_addr: address, tokens_sold:
uint256):
    exchange: address =
self.factory.getExchange(token_addr)
    fee: uint256 = tokens_sold / 500
    invariant: uint256 = self.eth_pool * self.token_pool
    new_token_pool: uint256 = self.token_pool +
tokens_sold
    new_eth_pool: uint256 = invariant / (new_token_pool
- fee)
    eth_out: uint256 = self.eth_pool - new_eth_pool
    self.eth_pool = new_eth_pool
```

```
    self.token_pool = new_token_pool
    Exchange(exchange).ethToTokenTransfer(msg.sender,
value=eth_out)
```

`token_addr` 是 KNC 代币的地址，`tokens_sold` 是 OMG 要卖出的数量。这个函数首先会检查工厂合约以检索 KNC 的交易所地址。然后交易所会把转入的 OMG 转换成 ETH。但是此时并没有把转换出来的 ETH 转给购买者，这个函数在这次的KNC交易所中会调用函数`ethToTokenTransfer()`来支付。

```
@public
@payable
def ethToTokenTransfer(recipent: address):
    fee: uint256 = msg.value / 500
    invariant: uint256 = self.eth_pool * self.token_pool
    new_eth_pool: uint256 = self.eth_pool + msg.value
    new_token_pool: uint256 = invariant / (new_eth_pool
- fee)
    tokens_out: uint256 = self.token_pool -
new_token_pool
    self.eth_pool = new_eth_pool
    self.token_pool = new_token_pool
    self.invariant = new_eth_pool * new_token_pool
    self.token.transfer(recipent, tokens_out)
```

`ethToTokenTransfer()` 接收 ETH 和购买者地址，首先验证调用的交易所是否存在于注册表，然后把 ETH 兑换成 KNC，最后把 KNC 转发给购买者。`ethToTokenTransfer()` 函数和 `ethToTokenSwap()` 函数相同，但是带有额外的入参`recipient: address`。在本例中就是 OMG 的兑换中，这是用来把购买的代币转发给购买者而不是`msg.sender`。



Swaps 对比 Transfers

`ethToTokenSwap()`，`tokenToEthSwap()` 和 `tokenToTokenSwap()` 函数会返回购买的代币给购买者的地址。

`ethToTokenTransfer()`，`tokenToEthTransfer()` 和 `tokenToTokenTransfer()` 函数允许购买者执行交易后然后把购买到的代币转发到指定接收的地址。

提供流动性

增加流动性

增加流动性需要投放等值的 ETH 和 ERC20 代币到 ERC20 代币的交易合约里。

第一个流动性提供者加入到资金池中，通过存入他们所认同的等值的 ETH 和 ERC20 代币来设置初始的兑换汇率。如果这一比率下降，套利交易者将以最初的流动性提供者为代价，使价格达到均衡。如果比率是错的，套利交易者会把价格带到合理位置，而让初始的流动性提供者付出代价。

所有后来的流动性提供者存入 ETH 和 ERC20 时会使用他们当时的交易比率。如果交易比率不好，那么就存在有利可图的套利机会使得价格恢复正常。

流动性代币

流动性代币发行是为了追踪每个流动性提供者贡献的相对于总保证金的相对比例。他们是高度可分离的，并且可以在任何时候销毁，返还相应份额的市场流动性给提供者。

流动性提供者调用 `addLiquidity()` 函数，存入准备金以获取新发行的流动性代币：

```
@public
@payable
def addLiquidity():
    token_amount: uint256 = msg.value * token_pool /
eth_pool
    liquidity_minted: uint256 = msg.value *
total_liquidity / eth_pool



    eth_added: uint256 = msg.value
    shares_minted: uint256 = (eth_added *
self.total_shares) / self.eth_pool
    tokens_added: uint256 = (shares_minted *
self.token_pool) / self.total_shares)
    self.shares[msg.sender] = self.shares[msg.sender] +
shares_minted
    self.total_shares = self.total_shares +
shares_minted
    self.eth_pool = self.eth_pool + eth_added
    self.token_pool = self.token_pool + tokens_added
    self.token.transferFrom(msg.sender, self,
tokens_added)
```

一定数量的 ETH 发送到这个函数，相应数量的流动性代币发行就被确定了。它可以用这个等式来计算：

$amountMinted=totalAmount*\frac{ethDeposited}{ethPool}$

存入 ETH 到保证金需要同等价值数量的 ERC20 代币。可以用下面这个等式计算：

$tokensDeposited=tokenPool*\frac{ethDeposited}{ethPool}$

移除流动性

提供者可以在任何时候销毁他们的流动性代币以从资金池中提取他们相应份额的 ETH 和 ERC20 代币。

$ethWithdrawn=ethPool*\frac{amountBurned}{totalAmount}$

$tokensWithdrawn=tokenPool*\frac{amountBurned}{totalAmount}$

ETH 和 ERC20 代币被提取的时候是以当前的交易比率（保证金比率），而不是他们当初投资的比率。这是因为一些价值因为市场波动和套利而丢失。

交易中收取的手续费被添加到总流动性资金池但是不会产生新的流动性代币。所以从流动性首次被投入开始，`ethWithdrawn`和`tokensWithdrawn`就包含了所有收取手续费中的相应比例份额。

流动性代币

Uniswap 流动性代币代表了一个流动性提供者对一个 ETH-ERC20 交易对的贡献度。他们本身就是 ERC20 代币并且拥有 EIP-20 的完整实现。

这允许流动性提供者可以卖掉他们的流动性代币，或者不需要从资金池中移动流动性直接转账给不同账户。流动性代币是特定于某个

ETH⇄ERC20 交易所。这个项目中并不存在一个通用 ERC20 代币。

手续费构成

- ETH 兑换 ERC20
  - 手续费为 ETH 的 0.3%
- ERC20 兑换 ETH
  - 手续费为 ERC20 代币的 0.3%
- ERC20 兑换 ERC20
  - 输入 ERC20 到 ETH 交易，手续费为 ERC20 代币的 0.3%。
  - 输出 ETH 到 ERC20 交易，手续费为 ETH 的 0.3%。
  - 实际手续费为输入 ERC20 代币 的 0.5991%

ETH 和 ERC20 代币兑换中需要收取 0.3% 的手续费。这个手续费会根据流动性提供者对流动性准备金的贡献度来按比例分配。因为 ERC20 对 ERC20 的交易包含了 ERC20 到 ETH 的兑换和 ETH 到 ERC20 的兑换，所以手续费会收取两次。但是并不存在平台手续费。

兑换手续费是被立即存入到流动性准备金中。由于总准备金增加了，但是没有增加额外的份额代币，所以所有的份额代币增加了同等价值。这相当于向流动性提供者支付了一笔费用，流动性提供者可以通过销毁份额来提取。

由于手续费被添加到流动性资金池中，不变量在每次交易结束时都有增长。在一次交易中，`invariant` 不变量表示的是上一次交易结束后的 `eth_pool（ETH资金池） * token_pool（代币资金池）`

自定义资金池

ERC20 到交易所

附加函数 `tokenToExchangeSwap()` 和 `tokenToExchangeTransfer()` 增加了 Uniswap 的灵活性。这些函数转换 ERC20 代币到 ETH ，并且尝试

用用户输入的地址调用`ethToTokenTransfer()`。只要是实现了相应的接口，即使是不同工厂之间的自定义 uniswap 交易所，也可以实现 ERC20 到 ERC20 的交易。自定义交易所可以有不同的曲线、管理者、私人流动资金池、基于 FOMO 的资金盘，或者任何你能想到的东西。

选择性升级

面对不断升级的审查，实现去中心化的智能合约很难。有希望的是，Uniswap 1.0 虽然很完美，但还不是。如果改良版的Uniswap 2.0 设计被创建出来，那么就可以部署一个新的工厂合约。流动性提供者可以选择转移到新系统或者留在老系统中。

`tokenToExchange` 函数支持和不同的工厂推出的交易所进行交易。这个可以用来实现向后兼容。使用 `tokenToToken` and `tokenToExchange` 函数可能让ERC20 到 ERC20 在同版本之间交易。可是，跨版本的话，只有 `tokenToExchange` 能正常运行。所有的升级都是可选择的并且向后兼容的。💖💖💖

提前交易

Uniswap 可以在一定程度上实现提前交易。用户可以使用最大最小值和交易截止日期来限制。

白皮书中的 DEX

DEX： https://uniswap.exchange

原文链接：https://hackmd.io/@477aQ9OrQTCbVR3fq1Qzxg/HJ9jLsfTz?type=view (2019-07-14)

翻译：Shawn Xie

Shawn Xie

Read more posts by this author.

Share this post

0条评论 登录 ▼

开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号

姓名

♡ 分享 最佳 最新 最早

来做第一个留言的人吧！

订阅 隐私 不要出售我的数据

YOU MIGHT ENJOY

使用智能合约实现自动分账

FIBOS 是结合了 EOS 和 JavaScript 的区块链网络。它底层使用了EOS 来保障性能…