作为一个区块链技术的学习者，个人认为自己有必要了解一下当前DeFi生态最为流行的去中心化交易所之一，Uniswap的运行机制。了解Uniswap先从它们的白皮书入手，先看看最初的Uniswap是什么样的（Uniswap已经出到了v3版本）。

简介

Uniswap是一种运行在以太坊上的可用于进行代币兑换的自动化代币兑换协议，与传统交易所基于订单簿并促进买卖双方的代币交易方式不同，Uniswap是基于各种代币的流动性储备来完成代币兑换交易的，这样的好处是买卖双方能快速直接地完成代币兑换而不必挂单等待。

代币兑换的价格由恒定乘积做市场来决定（x*y=k），这种自动做市的机制能使得代币的总体储备相对均衡。代币的储备由流动性提供者保障，流动性提供者会按代币交易费用的一定比例获取奖励。

Uniswap 的一个重要特征是利用工厂/注册（factory/registry）合约，为每个 ERC20 代币部署一个单独的代币兑换合约，并且这些交易合约各自持有 ETH 及其相关的 ERC20 代币来用于代币兑换。 代币兑换合约通过注册表链接，因此我们可以以ETH 作为中介直接进行ERC20代币到ERC20代币的兑换交易。

Uniswap由于设计较为简约，因此其使用gas的效率非常高。

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						

各交易所进行代币兑换所需消耗的gas (图片来源：Uniswap whitepaper V1)

创建代币兑换合约

uniswap_factory.vy是一个工厂合约，其中的函数createExchange()允许任何用户为任何未没创建代币兑换合约的ERC20代币创建并部署代币兑换合约。

```
exchangeTemplate: public(address)   #模板地址
token_to_exchange: address[address] #ERC20代币地址到兑换合约地址的映射
exchange_to_token: address[address] #兑换合约地址到ERC20代币地址的映射

@public
def __init__(template: address):    # 传入模板地址
    self.exchangeTemplate = template

@public
```

```
def createExchange(token: address) -> address:
    assert self.token_to_exchange[token] == ZERO_ADDRESS    #是否为0地址
    new_exchange: address = create_with_code_of(self.exchangeTemplate)  #创建兑换合约
    self.token_to_exchange[token] = new_exchange    #可用ERC20代币地址找到兑换合约地址
    self.exchange_to_token[new_exchange] = token    #可用兑换合约地址找到ERC20代币地址
    return new_exchange #返回兑换合约地址
```

uniswap_factory.vy合约中createExchange函数简略版本

所有的代币机器相关交易记录都存储在工厂合约中，并且ERC20代币地址和兑换合约地址之间可以相互查询。getExchange()函数可以根据传入的代币地址找到对应的兑换合约地址，getToken()函数可以根据传入的兑换合约地址找到相应的ERC20代币地址。

```
@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]
```

getExchange和getToken函数简略版本

工厂在生成兑换合约的时候不会执行任何检查，但是会强制要求每种代币只能生成一个兑换合约。

ETH和ERC20代币相互兑换

每个兑换合约（uniswap_exchange.vy）都与一个ERC20代币相关联，并且合约中都维护一个ETH和该代币的流动性资金池。ETH和ERC20代币之间的兑换比例主要取决于合约中流动性池里两个币的存储量，由公式ETH数量*代币数量=恒定值k决定。k在交易期间保持不变，只有在兑换合约中流动性发生改变时才改变。

ethToTokenSwap()为将ETH兑换为ERC20代币的兑换函数，下面展示一个简化版本：

```
eth_pool: uint256          #ETH锁仓量
token_pool: uint256        #ERC20代币锁仓量
token: address(ERC20)    #ERC20代币地址

@public
@payable
def ethToTokenSwap():
    fee: uint256 = msg.value / 500  #手续费，五百分之一的ETH
    invariant: uint256 = self.eth_pool * self.token_pool    #计算k
    new_eth_pool: uint256 = self.eth_pool + msg.value    #将ETH锁仓
    new_token_pool: uint256 = invariant / (new_eth_pool - fee)  #根据x*y=k代币剩余数量
    tokens_out: uint256 = self.token_pool - new_token_pool  #计算兑换所得代币数量
    self.eth_pool = new_eth_pool    #更新ETH锁仓量
```

```
    self.token_pool = new_token_pool     #更新代币锁仓量
    self.token.transfer(msg.sender, tokens_out) #将代币转出给用户
```

uniswap_exchange.vy合约中ethToTokenSwap函数简略版本

当用户调用ethToTokenSwap函数以ETH兑换代币时，会将ETH发送到代币兑换合约并使得eth_pool增加。为了保持k不变，代币的数量需要减少，代币较少的数量就是用户用ETH换得的数量。当兑换合约中其中一个币种数量持续减少时，用相同数量的另一个币种能换出来的该代币数量就越多，这也就意味着该代币的价格升高了，这种兑换机制能够激励人们反向兑换，一定程度上维持价格的相对稳定。

为了节约gas，上面的eth_pool和token_pool都不是存储变量（以太坊中更改存储变量花费的gas较多）。在实际应用中ETH的锁仓量用self.balance方法获得，代币锁仓量用self.token.balanceOf(self)获得。

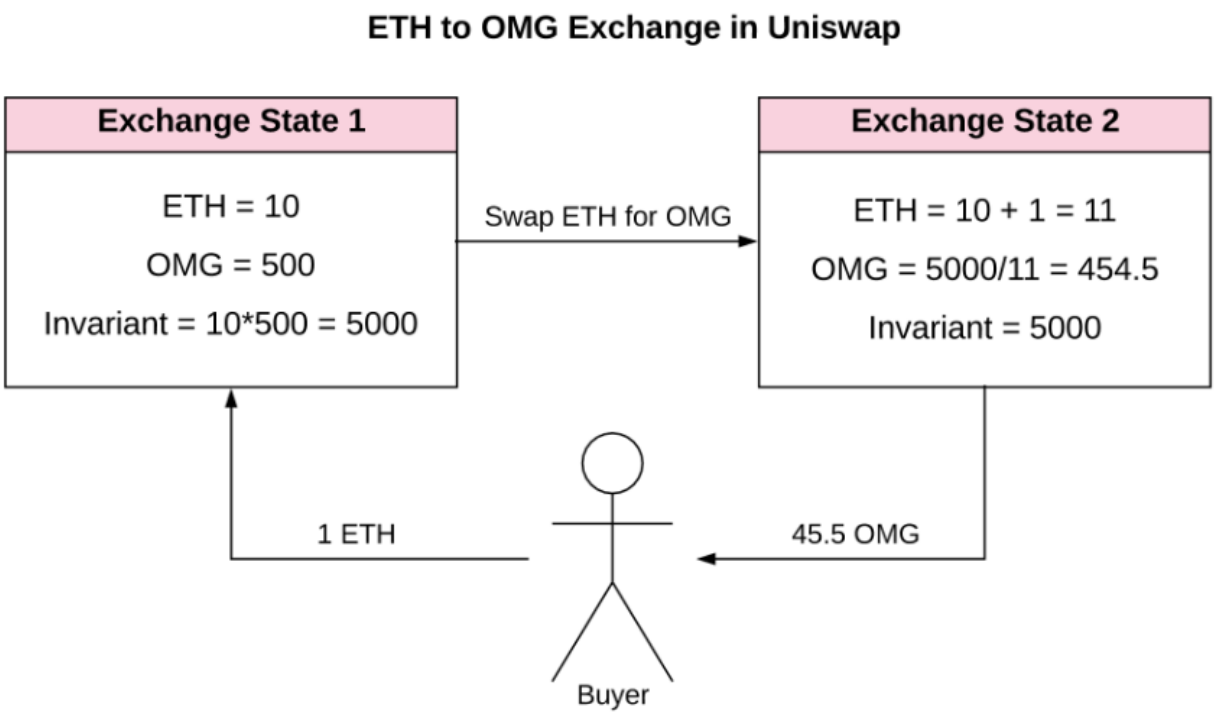
tokenToEthSwap()和ethToTokenSwap()相反，是把代币兑换为ETH的兑换函数，其简略版本如下：

```
@public
def tokenToEthSwap(tokens_in: uint256):
    fee: uint256 = tokens_in / 500  #收取五百分之一代币作为手续费
    invariant: uint256 = self.eth_pool * self.token_pool     #计算k值
    new_token_pool: uint256 = self.token_pool + tokens_in    #将代币锁仓
    new_eth_pool: uint256 = self.invariant / (new_token_pool - fee) #计算ETH剩余数量
    eth_out: uint256 = self.eth_pool - new_eth_pool #计算兑换所得ETH数量
    self.eth_pool = new_eth_pool     #更新ETH锁仓量
    self.token_pool = new_token_pool     #更新代币锁仓量
    self.token.transferFrom(msg.sender, self, tokens_out)    #收取用户代币
    send(msg.sender, eth_out)    #将ETH转出给用户
```

uniswap_exchange.vy合约中tokenToEthSwap函数简略版本

tokenToEthSwap()函数和ethToTokenSwap()函数实现形式但兑换方向相反，不过多解释。

下面列举一个ETH兑换OMG代币的样例：



ETH-OMG代币兑换样例图(图片来源：Uniswap whitepaper V1)

现在Buyer准备在Uniswap中用1ETH换取OMG代币。兑换前Uniswap的ETH-OMG代币池中ETH的数量为10，OMG的数量为500，则k（图中的invariant）为10*500=500，现假设Uniswap的手续费为0.25%。

Buyer将1ETH交给兑换合约后，兑换合约会先扣除手续费1*0.25%=0.0025ETH，这部分手续费会作为流动性提供者的奖励，剩下的0.9975ETH会被加入代币池并对兑换处OMG代币。当ETH加入代币池后，代币池的ETH存量就变为10.9975，又因为k=5000恒定不变，所以OMG的存量需要变为5000/10.9975=454.6487，取整后为454.65，因此Buyer能兑换出500-454.65=45.35个OMG代币。由于计算过程中会出现小数，因此当交易完成后k值实际上还是有可能发生微小的变动。

上面的样例图为不考虑手续费情况下的兑换情况。

ERC20代币和ERC20代币相互兑换

由于ETH被用作所有代币的公共对，它可以被用来作为代币与代币之间兑换的桥梁。比如如果想要用OMG代币兑换KNC，则可以先用OMG兑换成ETH，然后再用ETH兑换成KNC。

下面是用OMG代币兑换KNC代币的函数示例：

```
#获得代币兑换合约地址
contract Factory():
    def getExchange(token_addr: address) -> address: constant

#ETH兑换代币函数
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)    #获取KNC代币的兑换合约
    fee: uint256 = tokens_sold / 500    #手续费
    invariant: uint256 = self.eth_pool * self.token_pool    #计算k（ETH和OMG的k）
    new_token_pool: uint256 = self.token_pool + tokens_sold #锁仓OMG
    new_eth_pool: uint256 = invariant / (new_token_pool - fee)  #计算OMG池中ETH数量
    eth_out: uint256 = self.eth_pool - new_eth_pool #计算出OMG能兑换出的ETH数量eth_out
    self.eth_pool = new_eth_pool    #更新OMG池中ETH数量
    self.token_pool = new_token_pool    #更新OMG池中OMG数量
    #将eth_out作为输入调用KNC兑换合约的ETH兑换KNC函数，并将KNC发送给msg.sender（这里是买家）
    Exchange(exchange).ethToTokenTransfer(msg.sender, value=eth_out)
```

OMG代币兑换合约中的tokenToTokenSwap函数简略版本

上面是已部署的OMG代币兑换合约的函数，输入参数token_addr是KNC代币的地址，tokens_sold是OMG代币的数量。该函数先调用工厂合约的getExchange函数获取KNC代币兑换合约的地址，然后将OMG兑换为ETH，接着该函数没有将兑换出的ETH返还给买家，而是调用了KNC兑换合约中的ethToTokenTransfer函数，将ETH兑换为KNC。

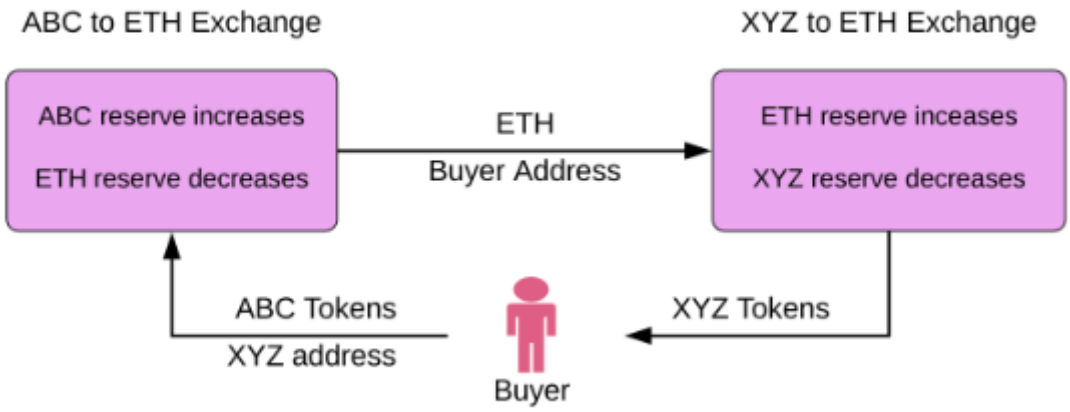
ethToTokenTransfer()函数实现如下：

```
@public
@payable
def ethToTokenTransfer(recipent: address):
    fee: uint256 = msg.value / 500  #手续费
    invariant: uint256 = self.eth_pool * self.token_pool    #计算k
    new_eth_pool: uint256 = self.eth_pool + msg.value   #锁仓ETH
    new_token_pool: uint256 = invariant / (new_eth_pool - fee)  #计算剩余的KNC数量
    tokens_out: uint256 = self.token_pool - new_token_pool  #计算能够兑换出的KNC数量
    self.eth_pool = new_eth_pool    #更新ETH锁仓量
    self.token_pool = new_token_pool    #更新KNC锁仓量
    self.invariant = new_eth_pool * new_token_pool  #更新k
    self.token.transfer(recipent, tokens_out)   #将KNC支付给买家
```

KNC代币兑换合约中的ethToTokenTransfer函数简略版本

ethToTokenTransfer函数接收ETH和买家地址，并且验证调用该函数的msg.sender（这里是OMG兑换合约）是从注册表中的兑换合约地址发出的。

下面为ERC20代币之间兑换的示意图：



ERC-20代币兑换ERC-20代币示意图(图片来源：Uniswap whitepaper V1)

添加流动性

添加流动性需要将等值的ETH和ERC20代币存入该代币的兑换合约中。第一个加入池子的流动性提供者可以通过存入他认为等值的ETH和ERC20代币来设置该池子的初始汇率。

流动性提供者在添加流动性（往池子里添加币）的时候会获得流动性代币作为添加凭证。流动性代币能够记录每个流动性提供者贡献的流动性占总储备的比例，他们是高度可分割的，并且可以随时销毁以按照市场流动性所占比例拿回ETH和代币（按照提供者拥有的流动性代币比例拿回对应的份额，因此实际拿回来的ETH和代币数量会发生变化，存在亏损风险）

流动性提供者能够通过调用addLiquidity函数存入ETH和ERC-20代币并获得相应份额的流动性代币。

```
@public
```

```
@payable
def addLiquidity():
    eth_added: uint256 = msg.value  #记录流动性添加者添加的ETH数量
    #计算所需要铸造的流动性代币数量（体现此次添加的流动性占总流动性的比例）
    #铸造的流动性代币数量=当前兑换合约总流动性*（新添加的ETH占池子内总ETH的比例）
    shares_minted: uint256 = (eth_added * self.total_shares) / self.eth_pool
    #计算此次添加流动性操作所需加入的代币数量（等价添加，在池子内价值由比例体现，所以是等比例添加）
    tokens_added: uint256 = (shares_minted * self.token_pool) / self.total_shares)
    #将铸造的流动性代币发放给流动性添加者
    self.shares[msg.sender] = self.shares[msg.sender] + shares_minted
    self.total_shares = self.total_shares + shares_minted   #更新兑换合约总流动性
    self.eth_pool = self.eth_pool + eth_added   #更新池子内ETH锁仓量
    self.token_pool = self.token_pool + tokens_added    #更新代币锁仓量
    self.token.transferFrom(msg.sender, self, tokens_added) #从流动性添加者处收取代币
```

addLiquidity添加流动性函数

铸造的流动性代币数量由发送到兑换合约的ETH数量决定，可以用下列公式表示：

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

流动性代币铸造数量计算公式

该公式对应到addLiquidity函数中即为shares_minted的计算。

添加流动性的时候还需要加入与ETH等值的ERC20代币（在兑换合约中价的格就是两个币种之间的兑换比例，所以添加流动性时的等价添加其实就是等比例添加或者说是等份额添加），因为这样根据x*y=k的计算公式才能保持代币之间的兑换价格不变，需添加的代币数量可以用下列公式计算：

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

所需添加的代币数量计算公式

该公式对应到addLiquidity函数中即为tokens_added的计算。

移除流动性

流动性添加者可以随时通过销毁他们的流动性代币，以从代币兑换池中按他们所持流动性代币所占比例提取相同比例的ETH和ERC20代币。所能提取的ETH和ERC20代币计算公式如下：

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

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

移除流动性时ETH和ERC20代币提取数量计算公式

ETH和ERC20代币按照代币兑换池当前的汇率（比例）提取，而不是流动性提供者提供流动性时的汇率，因此流动性提供者在移除流动性时可能会因为市场波动而蒙受损失。

进行代币兑换时产生的手续费会被添加入池子中，但是兑换合约不会为这部分手续费铸造流动性代币。

流动性代币

Uniswap流动性代币代表流动性提供者对代币兑换池（或者叫流动性池）做的贡献。它本身就是一种ERC-20代币，包括了对EIP-20的完整实现，因此流动性提供者可以出售或转移他们的流动性代币。

流动性代币是特定于具体的某个兑换合约的，这也就意味着不同的兑换合约之间的流动性代币并不互通，这是因为流动性代币本身就代表了流动性提供者对某个特定的代币兑换池所做的流动性贡献。

Uniswap的手续费

ETH兑换ERC20代币

用0.3%的ETH支付手续费

ERC20代币兑换ETH

用0.3%的ERC20代币支付手续费

ERC20代币兑换ERC20代币

用0.3%的ERC20代币支付从ERC20到ETH的手续费

用0.3%的ETH代币支付从ETH到ERC20的手续费（相对于扣除完第一次手续费之后的0.3%）

最终的手续费比率为0.5991%

ERC20代币到ERC20代币的交易包括ERC20代币到ETH和ETH到ERC20代币两笔兑换，因此手续费是分别支付给两个兑换合约的，Uniwap没有收取额外的平台费用。

手续费会直接被加入池子且没有额外铸造流动性代币，这相当于增加了所有流动性代币的价值，因此流动性提供者会因为用户在该池子进行代币兑换而获益，这部分收益最终可以通过销毁流动性代币来获得。

最后

现在回过头来看Uniswap V1版本的设计，不得不说有一种简洁的美感。Uniswap V1版本奠定了整个Uniswap交易所的核心业务思路，后续两个版本都围绕着这样的兑换体系展开并丰富，业务也更加复杂，更加难懂。

资料来源

- Uniswap Whitepaper V1