下面会对Uniswap V1版本的链上合约源码进行解读分析，主要是看Uniswap的各种功能是如何实现的。V1版本使用的不是Solidity而是Vyper语言编写，语言本身不关键，主要看实现逻辑和核心思路。

Uniswap V1版本比较简单，总共两份链上合约，分别是uniswap_exchange.vy代币兑换合约和uniswap_factory.vy工厂合约，其中兑换合约的功能就是实现基本的代币兑换操作，工厂合约的功能就是产生不同代币的代币兑换合约。

要看懂代码的操作目的就得明白每个功能的设计思想，因此**在看源码前推荐可以先去看Uniswap V1版本的白皮书。**

1 uniswap_exchange.vy代币兑换合约

该合约主要实现基本的代币兑换操作，并且维护一个流动性池以保障代币兑换的正常进行，一个代币兑换合约可以看作是专职于该代币兑换的交易所。

1.1 ETH和ERC20代币的相互兑换

先看用ERC20代币兑换ETH的实现。

```
# @notice Convert Tokens to ETH.
# @dev User specifies exact input and minimum output.
# @param tokens_sold Amount of Tokens sold.  tokens_sold为要支付的代币数量
# @param min_eth Minimum ETH purchased. min_eth为要购买的ETH的最小值
# @param deadline Time after which this transaction can no longer be executed.
# @return Amount of ETH bought. 返回最终购买到的ETH数量
# 函数功能：指定输入的代币数量，根据代币数量兑换ETH并发送给消息调用者
@public
def tokenToEthSwapInput(tokens_sold: uint256, min_eth: uint256(wei), deadline: timestamp) -> uint256(wei):
    return self.tokenToEthInput(tokens_sold, min_eth, deadline, msg.sender, msg.sender)

# @notice Convert Tokens to ETH and transfers ETH to recipient.
# @dev User specifies exact input and minimum output.
# @param tokens_sold Amount of Tokens sold.
# @param min_eth Minimum ETH purchased.
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output ETH.
# @return Amount of ETH bought.
# 函数功能：指定输入的代币数量，根据代币数量兑换ETH并发送给指定接收者
# 比tokenToEthSwapInput函数多了一个接收者，指定用来接收所兑换的ETH的地址
@public
def tokenToEthTransferInput(tokens_sold: uint256, min_eth: uint256(wei), deadline: timestamp, recipient: address) -> uint256(wei):
    assert recipient != self and recipient != ZERO_ADDRESS
    return self.tokenToEthInput(tokens_sold, min_eth, deadline, msg.sender, recipient)

# @notice Convert Tokens to ETH.
# @dev User specifies maximum input and exact output.
# @param eth_bought Amount of ETH purchased.    #想要购买的ETH的数量
# @param max_tokens Maximum Tokens sold.    #最大能接受的支付的代币数量
# @param deadline Time after which this transaction can no longer be executed.
# @return Amount of Tokens sold.    #最终消耗的代币数量
# 函数功能：指定所想要兑换到的ETH数量并将ETH发送给消息调用者，函数根据要兑换的ETH计算扣除代币
@public
def tokenToEthSwapOutput(eth_bought: uint256(wei), max_tokens: uint256,
deadline: timestamp) -> uint256:
    return self.tokenToEthOutput(eth_bought, max_tokens, deadline, msg.sender,
msg.sender)

# @notice Convert Tokens to ETH and transfers ETH to recipient.
# @dev User specifies maximum input and exact output.
# @param eth_bought Amount of ETH purchased.
# @param max_tokens Maximum Tokens sold.
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output ETH.
# @return Amount of Tokens sold.
# 函数功能：指定所想要兑换到的ETH数量并将ETH发送给指定接收者，函数根据要兑换的ETH计算扣除代币
@public
def tokenToEthTransferOutput(eth_bought: uint256(wei), max_tokens: uint256,
deadline: timestamp, recipient: address) -> uint256:
    assert recipient != self and recipient != ZERO_ADDRESS
    return self.tokenToEthOutput(eth_bought, max_tokens, deadline, msg.sender,
recipient)
```

ERC20代币兑换ETH的函数入口

用ERC20代币兑换ETH的入口有4个，如上所示。上面4个函数按照ETH接收者来划分可以分成Swap和Transfer，Swap代表消息调用者用自己的代币为自己兑换ETH，ETH会发送给消息调用者自身，Transfer表示消息调用者用自己的代币为别人兑换ETH，ETH会被发送给指定recipient；按照兑换数量计算方式来划分可以分成Input和Output，Input是根据自己花费的代币计算能够兑换得到的ETH，Output则是根据自己想要兑换的ETH数量计算自己会花费的代币数量。不同的划分直接体现在入口函数名字上，比如tokenToEthSwapInput函数就代表要用代币兑换ETH，为自己兑换，并且根据自己花费的代币计算我所能得到的ETH。

tokenToEthSwapInput和tokenToEthTransferInput都调用了函数tokenToEthInput进行ETH的兑换，区别是ETH的接收者不同，tokenToEthSwapOutput和tokenToEthTransferOutput的结构与前两者类似，不同的是输入换成了eth_bought，也就是想要兑换的ETH的目标数量，并且调用的函数是tokenToEthOutput。

因此接下来我们需要分析tokenToEthInput和tokenToEthOutput。

```
@private
def tokenToEthInput(tokens_sold: uint256, min_eth: uint256(wei), deadline:
timestamp, buyer: address, recipient: address) -> uint256(wei):
    #判断输入数据的合理性，且当前时间还没超过限定的时间戳
    assert deadline >= block.timestamp and (tokens_sold > 0 and min_eth > 0)
    #获取当前兑换合约对应代币的储备量
    token_reserve: uint256 = self.token.balanceOf(self)
    #调用getInputPrice函数获取可以兑换到的eth数量（as_unitless_number用于去除wei单位）
    eth_bought: uint256 = self.getInputPrice(tokens_sold, token_reserve, as_unitless_number(self.balance))
    #调用as_wei_value函数将单位转换成wei
    wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')
    assert wei_bought >= min_eth    #兑换的eth不能低于设定最小值
    send(recipient, wei_bought) #调用send函数向recipient转移兑换得到的eth
    #调用代币合约的transferFrom函数从购买者收取应当支付的代币
    assert self.token.transferFrom(buyer, self, tokens_sold)
    log.EthPurchase(buyer, tokens_sold, wei_bought) #日志
    return wei_bought
```

tokenToEthInput函数实现

tokenToEthInput通过token.balanceOf获得代币兑换合约当前的代币ERC20代币存储量，然后用getInputPrice获得可兑换到的ETH的数量，接着用as_wei_value将单位转换成wei后用send函数将eth发送给接收者，最后再调用transferForm从buyer手中收取应当支付的代币。

我们需要关注的点有两个，分别是①token是在哪里被初始化的，②getInputPrice实现。下面看token的初始化。

```
# @dev This function acts as a contract constructor which is not currently supported in contracts deployed
# using create_with_code_of(). It is called once by the factory during contract creation.
@public
def setup(token_addr: address):
    # 当没有绑定工厂，则所支持兑换的代币还未绑定，并且传入的address不为0时该函数才能被调用
    assert (self.factory == ZERO_ADDRESS and self.token == ZERO_ADDRESS) and token_addr != ZERO_ADDRESS
    self.factory = msg.sender #记录创建该工厂的
    self.token = token_addr #初始化代币地址
    self.name = 0x556e69737761702056310000000000000000000000000000000000000000000000
    self.symbol = 0x554e492d5631000000000000000000000000000000000000000000000000000000
    self.decimals = 18
```

setup函数实现

self.token是在setup函数中被初始化的，直接将传入参数token_addr赋值给self.token。

```
# @dev Pricing function for converting between ETH and Tokens.
# @param input_amount Amount of ETH or Tokens being sold.
# @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
# @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
# @return Amount of ETH or Tokens bought.
@private
@constant
def getInputPrice(input_amount: uint256, input_reserve: uint256, output_reserve: uint256) -> uint256:
    assert input_reserve > 0 and output_reserve > 0 #需要两币的储备都大于0
    input_amount_with_fee: uint256 = input_amount * 997 #抽取千分之3手续费
    numerator: uint256 = input_amount_with_fee * output_reserve
    denominator: uint256 = (input_reserve * 1000) + input_amount_with_fee
    return numerator / denominator
```

getInputPrice函数实现

getInputPrice用于计算最终所兑换出来的币（ETH或代币）的数量，输入为两个币的储备量和用于兑换的币的数量，输出为能够兑换出来的币的数量。计算公式如下：

$$output = \frac{input\_amount * output\_reserve}{input\_reserve + input\_amount}$$

getInputPrice兑换出的数量计算公式（抽取手续费前）

至于为什么这么算这里不多做解释，可以去看我之前翻译的白皮书。但是由于UniswapV1会收取0.3%的手续费，因此input_amount要乘以0.997最终计算的公式如下：

$$output = \frac{input\_amount * output\_reserve * 997}{input\_reserve * 1000 + input\_amount * 997}$$

getInputPrice兑换出的数量计算公式（抽取手续费后）

将上述公式套入ERC20代币兑换ETH的例子里，则公式中的output为所兑换出的ETH的数量，input_amount为所花费的代币数量，input_reserve为代币的储备量，output_amount为ETH的储备量。

看完tokenToEthInput接着看tokenToEthOutput的实现。

```
@private
def tokenToEthOutput(eth_bought: uint256(wei), max_tokens: uint256, deadline: timestamp, buyer: address, recipient: address) -> uint256:
    assert deadline >= block.timestamp and eth_bought > 0
    token_reserve: uint256 = self.token.balanceOf(self) #获取代币储备量
    # 通过getOutputPrice计算所需要花费的代币数量
    tokens_sold: uint256 = self.getOutputPrice(as_unitless_number(eth_bought), token_reserve, as_unitless_number(self.balance))
    # tokens sold is always > 0
    assert max_tokens >= tokens_sold
    send(recipient, eth_bought) #向接收者发送所兑换得到的ETH
    assert self.token.transferFrom(buyer, self, tokens_sold)     #从购买者收取代币
    log.EthPurchase(buyer, tokens_sold, eth_bought)
    return tokens_sold
```

tokenToEthOutput函数实现

tokenToEthOutput通过getOutputPrice计算兑换eth_bought数量的ETH所需要花费的代币数量。

```
# @dev Pricing function for converting between ETH and Tokens.
# @param output_amount Amount of ETH or Tokens being bought.
# @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
# @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
# @return Amount of ETH or Tokens sold.
@private
@constant
def getOutputPrice(output_amount: uint256, input_reserve: uint256, output_reserve: uint256) -> uint256:
    assert input_reserve > 0 and output_reserve > 0
    numerator: uint256 = input_reserve * output_amount * 1000
    denominator: uint256 = (output_reserve - output_amount) * 997
    return numerator / denominator + 1
```

getOutputPrice函数实现

与getInputPrice函数一样，计算的过程中需要收取0.3%手续费，因此计算公式如下：

$$input = \frac{input\_reserve * output\_amount * 1000}{(output\_reserve - output\_amount) * 997}$$

getOutputPrice兑换出的数量计算公式（抽取手续费后）

看完公式再对比代码实现之后我们会发现getOutputPrice函数在计算完价格之后最后还加了个1，这是因为**uint256的除法会产生浮点数，向下取整后小数会被舍去**，因此兑换者实际需要支付的代币数量会比理论上少一点。为了避免每次交易后交易所产生亏损（会导致流动性池内资金越来越少），因此在最后的计算结果手动加1向上取整，不过因为结算单位是wei，所以向上取整给用户带来的损失可以忽略不计。

接着看ETH兑换ERC20代币的实现，由于实现方式和ERC20代币兑换ETH一致，因此特殊的地方会在代码的注释里说明，不再过多介绍。

```
# @notice Convert ETH to Tokens.
# @dev User specifies exact input (msg.value).
# @dev User cannot specify minimum output or deadline.
# 用ETH兑换代币的默认函数，用户只需要指定输入的ETH的数量
@public
@payable
def __default__():
    self.ethToTokenInput(msg.value, 1, block.timestamp, msg.sender, msg.sender)

# @notice Convert ETH to Tokens.
# @dev User specifies exact input (msg.value) and minimum output.所以接收的代币最小值
# @param min_tokens Minimum Tokens bought.
# @param deadline Time after which this transaction can no longer be executed.
# @return Amount of Tokens bought.
# ETH通过msg.value的方式发送给代币合约，代币合约根据所接收到的ETH计算所需支付的代币
@public
@payable
def ethToTokenSwapInput(min_tokens: uint256, deadline: timestamp) -> uint256:
    return self.ethToTokenInput(msg.value, min_tokens, deadline, msg.sender, msg.sender)

# @notice Convert ETH to Tokens and transfers Tokens to recipient.
# @dev User specifies exact input (msg.value) and minimum output
# @param min_tokens Minimum Tokens bought.
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output Tokens.
# @return Amount of Tokens bought.
@public
@payable
def ethToTokenTransferInput(min_tokens: uint256, deadline: timestamp, recipient: address) -> uint256:
    assert recipient != self and recipient != ZERO_ADDRESS
    return self.ethToTokenInput(msg.value, min_tokens, deadline, msg.sender, recipient)

# @notice Convert ETH to Tokens.
# @dev User specifies maximum input (msg.value) and exact output.
# @param tokens_bought Amount of tokens bought.
# @param deadline Time after which this transaction can no longer be executed.
# @return Amount of ETH sold.
@public
@payable
def ethToTokenSwapOutput(tokens_bought: uint256, deadline: timestamp) -> uint256(wei):
    return self.ethToTokenOutput(tokens_bought, msg.value, deadline, msg.sender, msg.sender)

```

```
# @notice Convert ETH to Tokens and transfers Tokens to recipient.
# @dev User specifies maximum input (msg.value) and exact output.
# @param tokens_bought Amount of tokens bought.
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output Tokens.
# @return Amount of ETH sold.
@public
@payable
def ethToTokenTransferOutput(tokens_bought: uint256, deadline: timestamp, 
recipient: address) -> uint256(wei):
    assert recipient != self and recipient != ZERO_ADDRESS
    return self.ethToTokenOutput(tokens_bought, msg.value, deadline, 
msg.sender, recipient)

@private
def ethToTokenInput(eth_sold: uint256(wei), min_tokens: uint256, deadline: 
timestamp, buyer: address, recipient: address) -> uint256:
    assert deadline >= block.timestamp and (eth_sold > 0 and min_tokens > 0)
    token_reserve: uint256 = self.token.balanceOf(self) #获取代币储备量
    #用getInputPrice获得所购买得到的代币数量
    #因为交易是先转账再执行合约，所以获得ETH储备量的时候需要先减去该买者已经发送的ETH
    tokens_bought: uint256 = self.getInputPrice(as_unitless_number(eth_sold), 
as_unitless_number(self.balance - eth_sold), token_reserve)
    assert tokens_bought >= min_tokens
    assert self.token.transfer(recipient, tokens_bought)#向接收者发送代币
    log.TokenPurchase(buyer, eth_sold, tokens_bought)
    return tokens_bought

@private
def ethToTokenOutput(tokens_bought: uint256, max_eth: uint256(wei), deadline: 
timestamp, buyer: address, recipient: address) -> uint256(wei):
    assert deadline >= block.timestamp and (tokens_bought > 0 and max_eth > 0)
    token_reserve: uint256 = self.token.balanceOf(self) #获取代币储备
    #用getOutputPrice获得所需支付的ETH数量
    #因为交易是先转账再执行合约代码，所以调用该合约时ETH已经转到兑换合约中，
    #而入口函数会直接将msg.value作为max_eth传入，所以ETH储备量为self.balance-max_eth
    eth_sold: uint256 = self.getOutputPrice(tokens_bought, 
as_unitless_number(self.balance - max_eth), token_reserve)
    # Throws if eth_sold > max_eth
    # 计算需要退还给用户的ETH
    eth_refund: uint256(wei) = max_eth - as_wei_value(eth_sold, 'wei')
    if eth_refund > 0:
        send(buyer, eth_refund)#如果需要退还的ETH大于0，转账。
    assert self.token.transfer(recipient, tokens_bought)#向用户发送代币
    log.TokenPurchase(buyer, as_wei_value(eth_sold, 'wei'), tokens_bought)
    return as_wei_value(eth_sold, 'wei')
```

ETH兑换ERC20代币相关函数实现

1.2 ERC20代币和ERC20代币的相互兑换

Uniswap V1中ERC20代币的相互兑换主要是以ETH为中介进行，也就是说其中一种ERC20代币兑换成ETH，再由ETH兑换成目标ERC20代币

```
# @notice Convert Tokens (self.token) to Tokens (token_addr).
# @dev User specifies exact input and minimum output.
# @param tokens_sold Amount of Tokens sold.支付的代币数量
# @param min_tokens_bought Minimum Tokens (token_addr) purchased.购买的代币的最小值
# @param min_eth_bought Minimum ETH purchased as intermediary.作为中介的ETH的最小值
# @param deadline Time after which this transaction can no longer be executed.
# @param token_addr The address of the token being purchased.目标代币的ERC20合约地址
# @return Amount of Tokens (token_addr) bought.最终购买的代币数量
# 根据输入的代币数量兑换相应数量的目标代币
@public
def tokenToTokenSwapInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline: timestamp, token_addr: address) -> uint256:
    # 获得目标代币的兑换合约地址
    exchange_addr: address = self.factory.getExchange(token_addr)
    return self.tokenToTokenInput(tokens_sold, min_tokens_bought, min_eth_bought, deadline, msg.sender, msg.sender, exchange_addr)

# @notice Convert Tokens (self.token) to Tokens (token_addr) and transfers
#         Tokens (token_addr) to recipient.
# @dev User specifies exact input and minimum output.
# @param tokens_sold Amount of Tokens sold.支付的代币数量
# @param min_tokens_bought Minimum Tokens (token_addr) purchased.购买的代币的最小值
# @param min_eth_bought Minimum ETH purchased as intermediary.作为中介的ETH的最小值
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output ETH.目标代币的接收者地址
# @param token_addr The address of the token being purchased.目标代币的ERC20合约地址
# @return Amount of Tokens (token_addr) bought.最终购买的代币数量
# 根据输入的代币数量兑换相应数量的目标代币，并将目标代币发送给指定接收者
@public
def tokenToTokenTransferInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline: timestamp, recipient: address, token_addr: address) -> uint256:
    exchange_addr: address = self.factory.getExchange(token_addr)
    return self.tokenToTokenInput(tokens_sold, min_tokens_bought, min_eth_bought, deadline, msg.sender, recipient, exchange_addr)

# @notice Convert Tokens (self.token) to Tokens (token_addr).
# @dev User specifies maximum input and exact output.
# @param tokens_bought Amount of Tokens (token_addr) bought.所要购买的代币数量
# @param max_tokens_sold Maximum Tokens (self.token) sold. 所要支付的代币的最大值
# @param max_eth_sold Maximum ETH purchased as intermediary.作为中介的ETH的最大值
# @param deadline Time after which this transaction can no longer be executed.
# @param token_addr The address of the token being purchased.目标代币的ERC20合约地址
# @return Amount of Tokens (self.token) sold.最终所需要支付的代币数量
# 根据所要购买的目标代币数量支付相应数量的持有代币
@public
def tokenToTokenSwapOutput(tokens_bought: uint256, max_tokens_sold: uint256, max_eth_sold: uint256(wei), deadline: timestamp, token_addr: address) -> uint256:
    exchange_addr: address = self.factory.getExchange(token_addr)
    return self.tokenToTokenOutput(tokens_bought, max_tokens_sold, max_eth_sold, deadline, msg.sender, msg.sender, exchange_addr)

# @notice Convert Tokens (self.token) to Tokens (token_addr) and transfers
#         Tokens (token_addr) to recipient.
# @dev User specifies maximum input and exact output.
# @param tokens_bought Amount of Tokens (token_addr) bought.所要购买的代币数量
# @param max_tokens_sold Maximum Tokens (self.token) sold.所要支付的代币的最大值
# @param max_eth_sold Maximum ETH purchased as intermediary.作为中介的ETH的最大值
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output ETH.目标代币的接收者地址
# @param token_addr The address of the token being purchased.目标代币的ERC20合约地址
# @return Amount of Tokens (self.token) sold.最终所需要支付的代币数量
# 根据所要购买的目标代币数量支付相应数量的持有代币，并将所兑换的目标代币发送给指定接收者
@public
def tokenToTokenTransferOutput(tokens_bought: uint256, max_tokens_sold: uint256, max_eth_sold: uint256(wei), deadline: timestamp, recipient: address, token_addr: address) -> uint256:
    exchange_addr: address = self.factory.getExchange(token_addr)
    return self.tokenToTokenOutput(tokens_bought, max_tokens_sold, max_eth_sold, deadline, msg.sender, recipient, exchange_addr)
```

ERC20代币兑换ERC20代币的函数入口

ERC20代币兑换ERC20代币的函数入口和ERC20代币兑换ETH的函数入口类似，不同的是ERC20代币兑换ERC20代币的函数入口在调用进行兑换的业务函数前会先用创建自身的工厂合约实现的getExchange函数来获取目标代币所在的兑换合约地址，然后再向目标兑换合约地址发送兑换请求，将在本合约兑换得到的ETH兑换成目标代币。

函数入口调用到的函数就两种，分别是根据输入代币计算输出代币的tokenToTokenInput，以及根据输出代币计算输入代币的tokenToTokenOutput。

下面先看tokenToTokenInput。

```
@private
def tokenToTokenInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline: timestamp, buyer: address, recipient: address, exchange_addr: address) -> uint256:
    assert (deadline >= block.timestamp and tokens_sold > 0) and (min_tokens_bought > 0 and min_eth_bought > 0)
    assert exchange_addr != self and exchange_addr != ZERO_ADDRESS
    token_reserve: uint256 = self.token.balanceOf(self) #获得支付代币的储备量
    #用getInputPrice计算所能兑换到的ETH
    eth_bought: uint256 = self.getInputPrice(tokens_sold, token_reserve, as_unitless_number(self.balance))
    wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')#将单位转换成wei
    assert wei_bought >= min_eth_bought
    assert self.token.transferFrom(buyer, self, tokens_sold)#收取支付代币
    # 调用目标兑换合约地址的ethToTokenTransferInput函数，将ETH兑换成目标代币
    tokens_bought: uint256 = 
Exchange(exchange_addr).ethToTokenTransferInput(min_tokens_bought, deadline, 
recipient, value=wei_bought)
    log.EthPurchase(buyer, tokens_sold, wei_bought)
    return tokens_bought
```

tokenToTokenInput函数实现

tokenToTokenInput在将支付代币兑换成ETH后，就将ETH发送到目标代币的兑换合约地址并调用其ethToTokenTransferInput函数来将ETH兑换成目标代币。

接着看tokenToTokenOutput。

```
@private
def tokenToTokenOutput(tokens_bought: uint256, max_tokens_sold: uint256, 
max_eth_sold: uint256(wei), deadline: timestamp, buyer: address, recipient: 
address, exchange_addr: address) -> uint256:
    assert deadline >= block.timestamp and (tokens_bought > 0 and max_eth_sold 
> 0)
    assert exchange_addr != self and exchange_addr != ZERO_ADDRESS
    # 调用目标兑换合约的getEthToTokenOutputPrice来根据目标代币数量计算所需的中介ETH的数量
    eth_bought: uint256(wei) = 
Exchange(exchange_addr).getEthToTokenOutputPrice(tokens_bought)
    token_reserve: uint256 = self.token.balanceOf(self)#获得支付代币的储备量
    # 根据得到的eth_bought代入getOutputPrice计算所需支付的代币数量
    tokens_sold: uint256 = self.getOutputPrice(as_unitless_number(eth_bought), 
token_reserve, as_unitless_number(self.balance))
    # tokens sold is always > 0
    assert max_tokens_sold >= tokens_sold and max_eth_sold >= eth_bought
    assert self.token.transferFrom(buyer, self, tokens_sold)#收取支付代币
    # 调用目标兑换合约地址的ethToTokenTransferOutput函数，将ETH兑换成目标代币
    eth_sold: uint256(wei) = 
Exchange(exchange_addr).ethToTokenTransferOutput(tokens_bought, deadline, 
recipient, value=eth_bought)
    log.EthPurchase(buyer, tokens_sold, eth_bought)
    return tokens_sold
```

tokenToTokenOutput函数实现

tokenToTokenOutput的实现与tokenToEthOutput类似，不同的在于由于支付代币的兑换合约不知道所需兑换到的中介ETH数量是多少，因此需要先调用getEthToTokenOutputPrice来计算兑换出目标数量的目标代币需要多少中介ETH。

```
# @notice Public price function for ETH to Token trades with an exact output.
# @param tokens_bought Amount of Tokens bought.
# @return Amount of ETH needed to buy output Tokens.
# 根据所需兑换的代币数量计算需要支付的ETH数量
@public
@constant
def getEthToTokenOutputPrice(tokens_bought: uint256) -> uint256(wei):
    assert tokens_bought > 0
    token_reserve: uint256 = self.token.balanceOf(self) #获取本合约的代币储备量
    #计算所需支付的ETH
    eth_sold: uint256 = self.getOutputPrice(tokens_bought,
as_unitless_number(self.balance), token_reserve)
    return as_wei_value(eth_sold, 'wei')
```

getEthToTokenOutputPrice函数实现

getEthToTokenOutputPrice函数和ethToTokenSwapOutput相比就是同样都用getOutputPrice获取支付代币的数量，但是getEthToTokenOutputPrice不执行转账操作，因此可以用它来提前获取支付代币数量。

类似的只计算不转账的功能函数还有3个，如下所示：

```
# @notice Public price function for ETH to Token trades with an exact input.
# @param eth_sold Amount of ETH sold.
# @return Amount of Tokens that can be bought with input ETH.
# 根据支付的ETH计算可以购买到的代币数量
@public
@constant
def getEthToTokenInputPrice(eth_sold: uint256(wei)) -> uint256:
    assert eth_sold > 0
    token_reserve: uint256 = self.token.balanceOf(self)
    return self.getInputPrice(as_unitless_number(eth_sold),
as_unitless_number(self.balance), token_reserve)

# @notice Public price function for Token to ETH trades with an exact input.
# @param tokens_sold Amount of Tokens sold.
# @return Amount of ETH that can be bought with input Tokens.
# 根据支付的代币数量计算可以购买到的ETH数量
@public
@constant
def getTokenToEthInputPrice(tokens_sold: uint256) -> uint256(wei):
    assert tokens_sold > 0
    token_reserve: uint256 = self.token.balanceOf(self)
    eth_bought: uint256 = self.getInputPrice(tokens_sold, token_reserve,
as_unitless_number(self.balance))
    return as_wei_value(eth_bought, 'wei')

# @notice Public price function for Token to ETH trades with an exact output.
# @param eth_bought Amount of output ETH.
# @return Amount of Tokens needed to buy output ETH.
# 根据所要购买的ETH数量计算所需要支付的代币数量
```

```
@public
@constant
def getTokenToEthOutputPrice(eth_bought: uint256(wei)) -> uint256:
    assert eth_bought > 0
    token_reserve: uint256 = self.token.balanceOf(self)
    return self.getOutputPrice(as_unitless_number(eth_bought), token_reserve, as_unitless_number(self.balance))
```

用于代币或ETH数量计算的辅助功能函数实现

上面3个函数在兑换合约的其它地方没有被使用到，被用于外部调用。

1.2开头提到的ERC20代币兑换ERC20代币的4个函数入口仅支持由同一个工厂部署的兑换合约之间进行ERC20代币的兑换，但是Uniswap V1实际上也支持不同工厂部署的代币兑换合约之间进行兑换，入口函数如下：

```
# @notice Convert Tokens (self.token) to Tokens (exchange_addr.token).
# @dev Allows trades through contracts that were not deployed from the same factory.
# @dev User specifies exact input and minimum output.
# @param tokens_sold Amount of Tokens sold.
# @param min_tokens_bought Minimum Tokens (token_addr) purchased.
# @param min_eth_bought Minimum ETH purchased as intermediary.
# @param deadline Time after which this transaction can no longer be executed.
# @param exchange_addr The address of the exchange for the token being purchased.
# @return Amount of Tokens (exchange_addr.token) bought.
@public
def tokenToExchangeSwapInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline: timestamp, exchange_addr: address) -> uint256:
    return self.tokenToTokenInput(tokens_sold, min_tokens_bought, min_eth_bought, deadline, msg.sender, msg.sender, exchange_addr)

# @notice Convert Tokens (self.token) to Tokens (exchange_addr.token) and transfers
#         Tokens (exchange_addr.token) to recipient.
# @dev Allows trades through contracts that were not deployed from the same factory.
# @dev User specifies exact input and minimum output.
# @param tokens_sold Amount of Tokens sold.
# @param min_tokens_bought Minimum Tokens (token_addr) purchased.
# @param min_eth_bought Minimum ETH purchased as intermediary.
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output ETH.
# @param exchange_addr The address of the exchange for the token being purchased.
# @return Amount of Tokens (exchange_addr.token) bought.
@public
def tokenToExchangeTransferInput(tokens_sold: uint256, min_tokens_bought: uint256, min_eth_bought: uint256(wei), deadline: timestamp, recipient: address, exchange_addr: address) -> uint256:
    assert recipient != self
    return self.tokenToTokenInput(tokens_sold, min_tokens_bought,
min_eth_bought, deadline, msg.sender, recipient, exchange_addr)

# @notice Convert Tokens (self.token) to Tokens (exchange_addr.token).
# @dev Allows trades through contracts that were not deployed from the same
factory.
# @dev User specifies maximum input and exact output.
# @param tokens_bought Amount of Tokens (token_addr) bought.
# @param max_tokens_sold Maximum Tokens (self.token) sold.
# @param max_eth_sold Maximum ETH purchased as intermediary.
# @param deadline Time after which this transaction can no longer be executed.
# @param exchange_addr The address of the exchange for the token being
purchased.
# @return Amount of Tokens (self.token) sold.
@public
def tokenToExchangeSwapOutput(tokens_bought: uint256, max_tokens_sold: uint256,
max_eth_sold: uint256(wei), deadline: timestamp, exchange_addr: address) ->
uint256:
    return self.tokenToTokenOutput(tokens_bought, max_tokens_sold,
max_eth_sold, deadline, msg.sender, msg.sender, exchange_addr)

# @notice Convert Tokens (self.token) to Tokens (exchange_addr.token) and
transfers
#         Tokens (exchange_addr.token) to recipient.
# @dev Allows trades through contracts that were not deployed from the same
factory.
# @dev User specifies maximum input and exact output.
# @param tokens_bought Amount of Tokens (token_addr) bought.
# @param max_tokens_sold Maximum Tokens (self.token) sold.
# @param max_eth_sold Maximum ETH purchased as intermediary.
# @param deadline Time after which this transaction can no longer be executed.
# @param recipient The address that receives output ETH.
# @param token_addr The address of the token being purchased.
# @return Amount of Tokens (self.token) sold.
@public
def tokenToExchangeTransferOutput(tokens_bought: uint256, max_tokens_sold:
uint256, max_eth_sold: uint256(wei), deadline: timestamp, recipient: address,
exchange_addr: address) -> uint256:
    assert recipient != self
    return self.tokenToTokenOutput(tokens_bought, max_tokens_sold,
max_eth_sold, deadline, msg.sender, recipient, exchange_addr)
```

跨工厂代币兑换函数入口

不作过多解释，和1.2开头的函数入口的区别就是不再通过getExchange获得代币兑换合约（因为部署兑换合约的工厂不同，所以无法直接获得），而是直接将别的工厂部署的代币兑换合约地址作为传入参数。

1.3 交易池流动性相关

Uniswap V1通过向流动性添加者发放流动性代币来记录流动性添加者在该交兑换合约的交易池内所持有的份额，流动性代币也是一种ERC20代币，不同的兑换合约之间的流动性代币不互通。（关于流动性池和流动性代币的相关概念在白皮书里）

```
#流动性代币名称
name: public(bytes32)                             # Uniswap V1
#流动性代币符号
symbol: public(bytes32)                           # UNI-V1
#精度
decimals: public(uint256)                         # 18
#流动性代币总供应量
totalSupply: public(uint256)                      # total number of UNI in existence
#余额映射
balances: uint256[address]                        # UNI balance of an address
#余额使用授权列表
allowances: (uint256[address])[address]           # UNI allowance of one address on another

# @dev This function acts as a contract constructor which is not currently supported in contracts deployed
#      using create_with_code_of(). It is called once by the factory during contract creation.
@public
def setup(token_addr: address):
    assert (self.factory == ZERO_ADDRESS and self.token == ZERO_ADDRESS) and token_addr != ZERO_ADDRESS
    self.factory = msg.sender
    self.token = token_addr
    self.name = 0x556e69737761702056310000000000000000000000000000000000000000000
    self.symbol = 0x554e492d563100000000000000000000000000000000000000000000000000000
    self.decimals = 18
```

流动性代币相关定义和初始化

从上面的实现来看实质上流动性代币就是一种ERC20代币，并且在setup函数中对部分参数进行初始化。

下面先看看添加流动性的函数实现。

```
# @notice Deposit ETH and Tokens (self.token) at current ratio to mint UNI tokens.
# @dev min_liquidity does nothing when total UNI supply is 0.
# @param min_liquidity Minimum number of UNI sender will mint if total UNI supply is greater than 0. 用户能接受的最少流动性代币
# @param max_tokens Maximum number of tokens deposited. Deposits max amount if total UNI supply is 0. 用户想要提供的代币数量最大值。
# @param deadline Time after which this transaction can no longer be executed.
# @return The amount of UNI minted. 所铸造的流动性代币数量
# 根据流动性池中ETH和代币的比例等比例添加两种币，并获得等比例份额的流动性代币
@public
@payable
def addLiquidity(min_liquidity: uint256, max_tokens: uint256, deadline: timestamp) -> uint256:
    assert deadline > block.timestamp and (max_tokens > 0 and msg.value > 0)
    total_liquidity: uint256 = self.totalSupply #获得流动性代币总供应量
    if total_liquidity > 0: #非该池子第一次添加流动性
        assert min_liquidity > 0 #添加的流动性最小也要大于0
        eth_reserve: uint256(wei) = self.balance - msg.value #获得ETH储备量
        token_reserve: uint256 = self.token.balanceOf(self) #获得代币储备量
        #根据投入的ETH数量计算需要投入的代币数量
        #最后+1是手动向上取整，防止默认的向下取整减少流动性池应收的代币数量，进而逐渐稀释份额
        token_amount: uint256 = msg.value * token_reserve / eth_reserve + 1
        #计算需要铸造的流动性代币数量
        #这里不向上取整是为了保证铸造的流动性代币价值<代币价值以防止流动性代币价值的稀释
        liquidity_minted: uint256 = msg.value * total_liquidity / eth_reserve
        assert max_tokens >= token_amount and liquidity_minted >= min_liquidity
        self.balances[msg.sender] += liquidity_minted#铸造流动性代币并发放给提供者
        self.totalSupply = total_liquidity + liquidity_minted#更新流动性代币总供应量
        assert self.token.transferFrom(msg.sender, self, token_amount)#收取代币
        log.AddLiquidity(msg.sender, msg.value, token_amount)
        log.Transfer(ZERO_ADDRESS, msg.sender, liquidity_minted)
        return liquidity_minted
    else: #该池子第一次添加流动性时
        assert (self.factory != ZERO_ADDRESS and self.token != ZERO_ADDRESS) and msg.value >= 1000000000
        #检查兑换合约地址和代币地址是否正确且对应
        assert self.factory.getExchange(self.token) == self
        token_amount: uint256 = max_tokens#直接将用户的代币全部投入池子
        #获取当前兑换合约的ETH余额数量，因为第一个人可以自行决定所要投入的代币和ETH，因此拥有定价权
        initial_liquidity: uint256 = as_unitless_number(self.balance)
        self.totalSupply = initial_liquidity#将ETH余额数量赋予给总供应量
        self.balances[msg.sender] = initial_liquidity#为第一个添加流动性的人发放流动性代币
        #收取添加的代币
        assert self.token.transferFrom(msg.sender, self, token_amount)
        log.AddLiquidity(msg.sender, msg.value, token_amount)
        log.Transfer(ZERO_ADDRESS, msg.sender, initial_liquidity)
        return initial_liquidity
```

addLiquidity添加流动性

添加流动性主要分两种情况，第一种情况是该池子第一次添加流动性时，兑换合约会直接铸造与合约ETH余额数量相等的流动性代币并发放给流动性添加者，并且第一次添加流动性时合约不对代币的添加数量做限制，也就意味着第一个流动性添加者有该代币的定价权，但是无法干预后续代币的价格变动。

第二种情况就是常规的流动性添加，兑换合约根据流动性添加者添加的ETH等比例收取代币，并根据添加的ETH所占比例铸造流动性代币并发放给添加者。

接着看移除流动性。

```
# @dev Burn UNI tokens to withdraw ETH and Tokens at current ratio.
# @param amount Amount of UNI burned.要销毁的流动性代币数量
# @param min_eth Minimum ETH withdrawn.提现的ETH最小值
# @param min_tokens Minimum Tokens withdrawn.提现的代币最小值
# @param deadline Time after which this transaction can no longer be executed.
# @return The amount of ETH and Tokens withdrawn.最终体现的ETH和代币最小值
@public
def removeLiquidity(amount: uint256, min_eth: uint256(wei), min_tokens: uint256, deadline: timestamp) -> (uint256(wei), uint256):
    assert (amount > 0 and deadline > block.timestamp) and (min_eth > 0 and min_tokens > 0)
    total_liquidity: uint256 = self.totalSupply #获取当前流动性代币总供应量
    assert total_liquidity > 0 #总供应量要大于0
    token_reserve: uint256 = self.token.balanceOf(self) #获取代币储备
    #根据移除的流动性占比等比例计算能提现的ETH余额，交易所不亏损所以不向上取整
    eth_amount: uint256(wei) = amount * self.balance / total_liquidity
    #等比例计算能提现的token余额
    token_amount: uint256 = amount * token_reserve / total_liquidity
    assert eth_amount >= min_eth and token_amount >= min_tokens #ETH和代币数量要大于预期
    self.balances[msg.sender] -= amount #扣除流动性移除者流动性代币
    self.totalSupply = total_liquidity - amount #销毁流动性代币
    send(msg.sender, eth_amount) #向移除者发送ETH
    assert self.token.transfer(msg.sender, token_amount) #向移除者发送代币
    log.RemoveLiquidity(msg.sender, eth_amount, token_amount)
    log.Transfer(msg.sender, ZERO_ADDRESS, amount)
    return eth_amount, token_amount
```

removeLiquidity添加流动性

移除流动性的操作也比较简单，等比例计算完之后向移除者转账并销毁流动性代币即可。

1.4 其它函数实现

代币兑换合约的主要功能就是上面提到的代币兑换和流动性操作，除此之外还有一些其它函数。

```
# @return Address of Token that is sold on this exchange.
# 获得代币地址
@public
@constant
def tokenAddress() -> address:
    return self.token

# @return Address of factory that created this exchange.
# 获得创建自身合约的工厂地址
@public
@constant
def factoryAddress() -> address(Factory):
    return self.factory
```

```

# ERC20 compatibility for exchange liquidity modified from
# https://github.com/ethereum/vyper/blob/master/examples/tokens/ERC20.vy
# 获取自身的流动性代币余额
@public
@constant
def balanceOf(_owner : address) -> uint256:
    return self.balances[_owner]

# 发送流动性代币
@public
def transfer(_to : address, _value : uint256) -> bool:
    self.balances[msg.sender] -= _value
    self.balances[_to] += _value
    log.Transfer(msg.sender, _to, _value)
    return True

# 授权进行流动性代币转账
@public
def transferFrom(_from : address, _to : address, _value : uint256) -> bool:
    self.balances[_from] -= _value
    self.balances[_to] += _value
    self.allowances[_from][msg.sender] -= _value
    log.Transfer(_from, _to, _value)
    return True

# 流动性代币使用授权
@public
def approve(_spender : address, _value : uint256) -> bool:
    self.allowances[msg.sender][_spender] = _value
    log.Approval(msg.sender, _spender, _value)
    return True

# 授权额度
@public
@constant
def allowance(_owner : address, _spender : address) -> uint256:
    return self.allowances[_owner][_spender]
```

兑换合约其它函数实现

可以看到基本都一些基于流动性代币的标准ERC20代币功能实现。

2 uniswap_factory.vy代币兑换合约

该合约主要实现兑换合约的部署，或者换句话说该工程主要用来部署不同代币的流动性池，实现如下：

```
contract Exchange(): #代币兑换合约接口
    def setup(token_addr: address): modifying

```

```
NewExchange: event({token: indexed(address), exchange: indexed(address)})

exchangeTemplate: public(address) #兑换合约模板地址
tokenCount: public(uint256) #已部署的代币兑换合约数量
token_to_exchange: address[address] #代币地址-兑换合约地址的映射
exchange_to_token: address[address] #代币兑换合约-代币地址的映射
id_to_token: address[uint256] #代币id到代币地址的映射

#初始化兑换合约地址模板，只能运行一次，当合约模板存在时无法再调用
@public
def initializeFactory(template: address):
    assert self.exchangeTemplate == ZERO_ADDRESS
    assert template != ZERO_ADDRESS
    self.exchangeTemplate = template

#创建代币兑换合约，传入代币地址
@public
def createExchange(token: address) -> address:
    assert token != ZERO_ADDRESS    #代币地址不能是0地址
    assert self.exchangeTemplate != ZERO_ADDRESS #合约模板不能为空
    assert self.token_to_exchange[token] == ZERO_ADDRESS #该代币需要未创建过兑换合约
    exchange: address = create_with_code_of(self.exchangeTemplate) #创建对比兑换合约
    Exchange(exchange).setup(token) #初始化代币兑换合约
    self.token_to_exchange[token] = exchange #记录代币兑换合约地址
    self.exchange_to_token[exchange] = token #记录代币地址
    token_id: uint256 = self.tokenCount + 1 #已部署的兑换合约数量+1并作为代币id
    self.tokenCount = token_id
    self.id_to_token[token_id] = token
    log.NewExchange(token, exchange)
    return exchange

#根据代币地址找到代币兑换合约地址
@public
@constant
def getExchange(token: address) -> address:
    return self.token_to_exchange[token]

#根据兑换合约地址找到代币地址
@public
@constant
def getToken(exchange: address) -> address:
    return self.exchange_to_token[exchange]

#根据代币id找到代币地址
@public
@constant
def getTokenWithId(token_id: uint256) -> address:
    return self.id_to_token[token_id]
```

Uniswap V1工厂合约实现

整个工厂合约的代码实现非常简单，唯一需要提的点就是工厂合约会通过create_with_code_of函数创建新的代币兑换合约。该函数的功能就是根据传入的合约地址找到已经部署上链的合约代码，然后将代码进行拷贝并部署，最后返回新合约的地址。

资料来源

[v1-contracts/contracts at master · Uniswap/v1-contracts (github.com)]