

智能合约

一、“Code is Law”，不可篡改的智能合约

什么是合约？

什么是智能合约？

合约内容公开透明

合约内容不可篡改

永久运行

定义

简单来说，“智能合约”就是一段可以运行在以太坊上的代码。之所以被称作“合约”，是因为用户可以通过这段运

行在以太坊上的代码控制有价值的事物，例如ETH 或其他数字资产

智能合约优势

去信任

经济、高效

无需第三方仲裁

比特币脚本与以太坊智能合约

去中心化的代码

将现实世界的逻辑在区块链上实现

合约的内容和生命周期被共识确认，是大家认可的条款

在所有节点上保证逻辑的一致性

在所有节点上产生和维护一致的数据

合约可能有Bug，Bug也不可篡改

Code is Law，是个理想目标

分布式环境

传统方式，信任依赖第三方，代码运行在第三方的服务器上。

有了区块链，信任就可以放在分布式网络上，合约代码在所有节点上执行、验证

比特币脚本

- 比特币脚本存在一些严重的限制

缺少图灵完备性

缺少价值控制

缺少状态

- 比特币脚本 -- “交易” 代码片段

```

{
  "hash": "5a42590f0e0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b",
  "ver": 1,
  "vin_sz": 2,
  "vout_sz": 1,
  "lock_time": 0,
  "size": 404,
  "in": [
    {
      "prev_out": {
        "hash": "3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
        "n": 0
      },
      "scriptSig": "30440..."
    },
    {
      "prev_out": {
        "hash": "7508e6ab259b4df0fd5147bab0c949d81473db4518f81afc5c3f52f91ff6b34e",
        "n": 0
      },
      "scriptSig": "3f3a4..."
    }
  ],
  "out": [
    {
      "value": "10.12287097",
      "scriptPubKey": "OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e
        OP_EQUALVERIFY OP_CHECKSIG"
    }
  ]
}

```

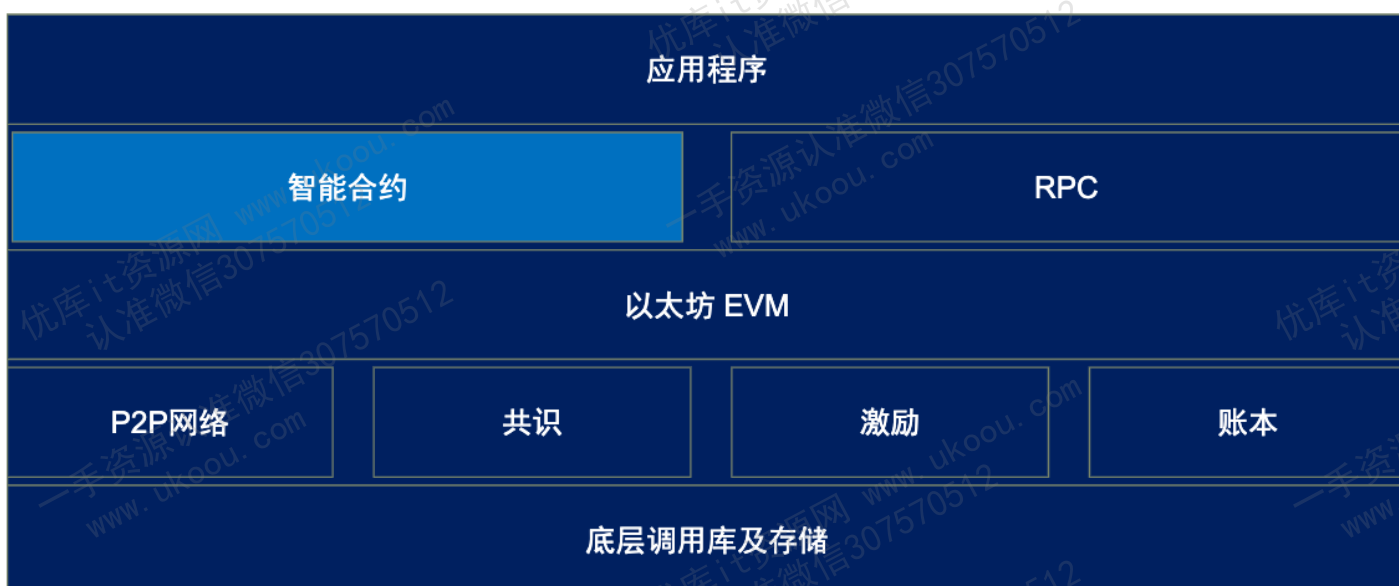
"hash": 交易ID
 "ver": 版本号
 "vin_sz": 输入的数量
 "vout_sz": 输出的数量
 "lock_time": 锁定时间
 "size": 交易规模

比特币脚本 -- 指令

数据指令	功能
<sig>	直接入栈
<pubkey>	直接入栈
操作符指令	功能
OP_DUP	复制堆栈顶端数据
OP_HASH160	计算哈希函数两次：第一次用 SHA – 256，第二次用 RIPEMD – 160
OP_EQUALVERIFY	如果输入是相同的，返回真 如果输入是不同的，返回假，整个交易作废
OP_CHECKSIG	检查输入的签名是否有效
OP_CHECKMULTISIG	检查在交易中 t 个公钥（地址）对应的 t 个签名是否有效
<sig> <pubkey>	数据指令

以太坊智能合约

Vitalik Buterin

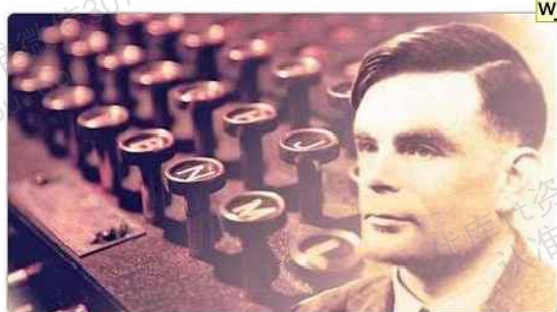


这个层次结构适合目前市面上所有的区块链平台

- 区块链层次结构

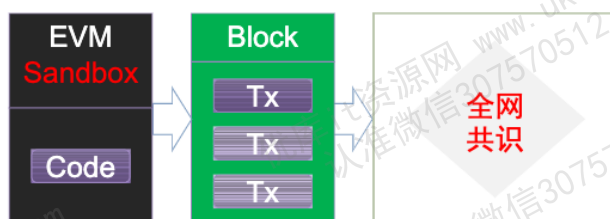
以太坊智能合约 -- 图灵完备

- 智能合约可以执行普通计算机可以执行的任何操作，尽管与常规计算机相比，区块链版本的运行速度要慢得多并且运行成本更高，这取决于区块链的设置。
- 理论上，可以将任何逻辑放入以太坊智能合约中，并由整个网络运行。
- 支持循环、跳转、判断、分支等语句。
- 支持多种数据类型。
- 支持面向对象编程。
- 可应对图灵停机问题。

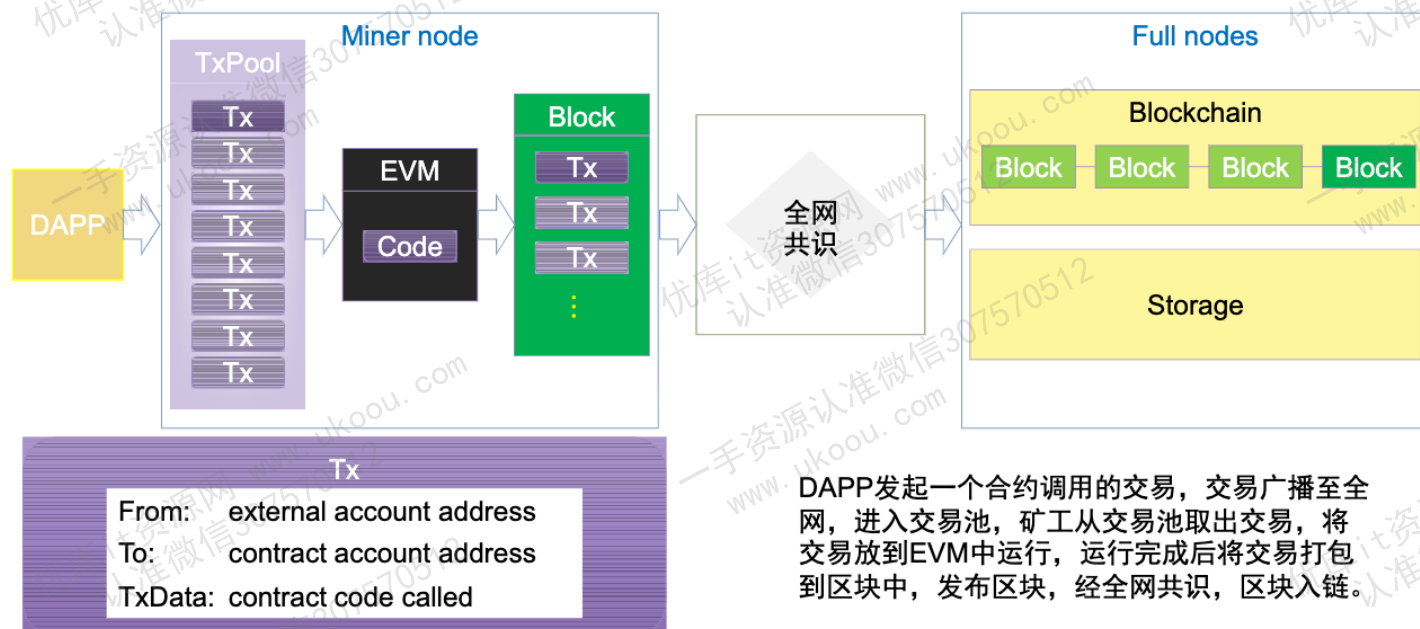


以太坊智能合约 -- 数据一致性

- 以太坊是一个交易驱动的状态机。
- 调用智能合约的交易发布到区块链上后，每个矿工都会执行这个交易，从当前状态确定性地转移到下一个状态。
- 智能合约在沙箱运行，不能访问时钟、网络、文件等不确定性系统。
- 所有节点的运行结果必须保持一致，经过共识的运行结果才能记录上链。

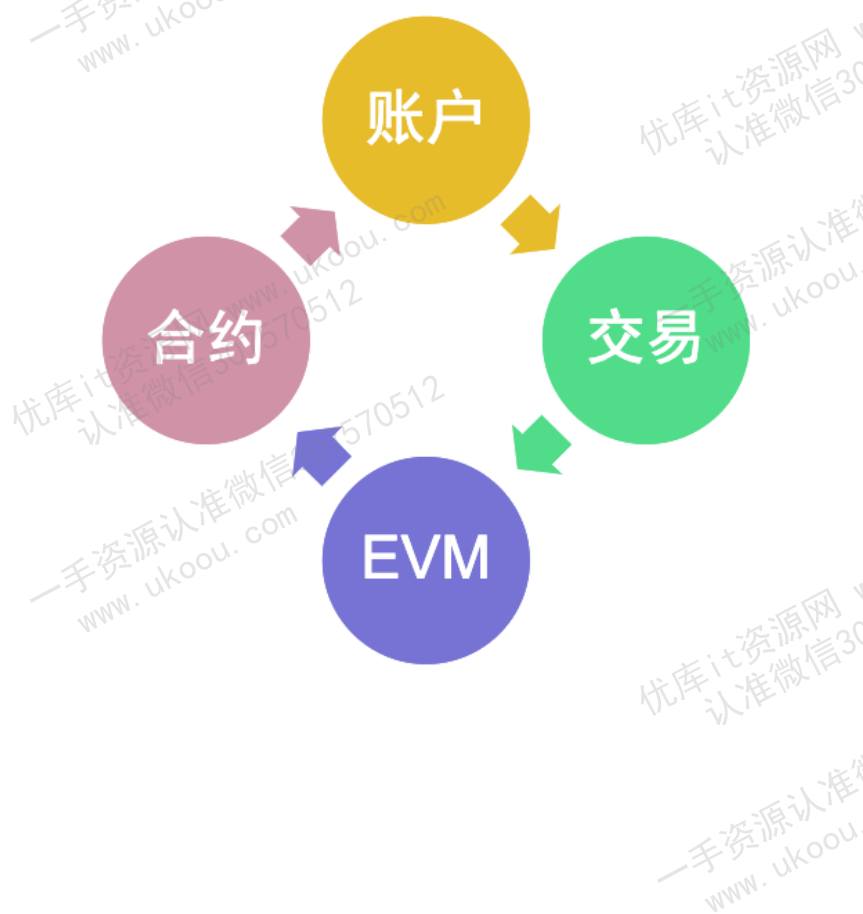


以太坊智能合约 -- 合约上链执行



以太坊智能合约 -- 驱动状态变化

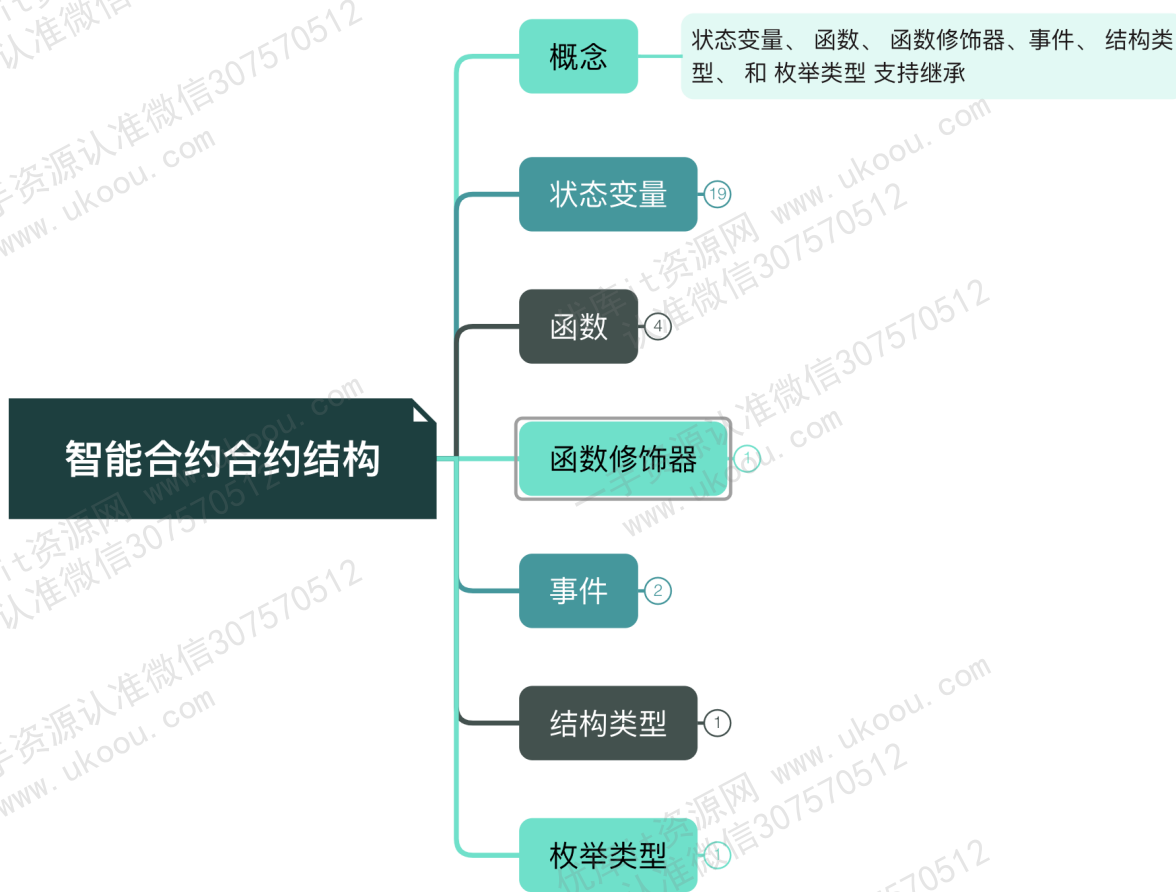
- 以太坊是一个状态机
- 由账户发起一笔交易
- 创建 EVM 虚拟机
- 装载合约代码并执行
- 合约执行结果使得账户状态发生改变



二、智能合约的合约结构

掌握了区块链的一些知识体系后，作为技术开发者，首先要学习的就是以太坊智能合约的开发。我们先从学习合约的开发开始，因为后面的技术栈中，我们需要用支持与以太坊交互的编程语言与以太坊交互，与合约交互，都是基于合约的代码逻辑来的，所以，接下来我们先从学会掌握 solidity 智能合约开发开始。

我们打开智能合约在线编辑器，编辑器的具体使用我就不具体教学了，可以自行百度或者谷歌学习编辑器的使用。我们在这里还是主要注重学习 solidity 语言的开发入门，在这里我们使用的都是 solidity 的 0.8 版本的特性。



概念

概念

状态变量、函数、函数修饰器、事件、结构类型、和 枚举类型 支持继承

状态变量

值类型

布尔类型

整型

定长浮点型

地址类型

地址类型成员变量

balance & transfer

send

call

定长字节数组

变长字节数组

地址字面常数

有理数和整数字面常数

字符串字面常数

枚举类型

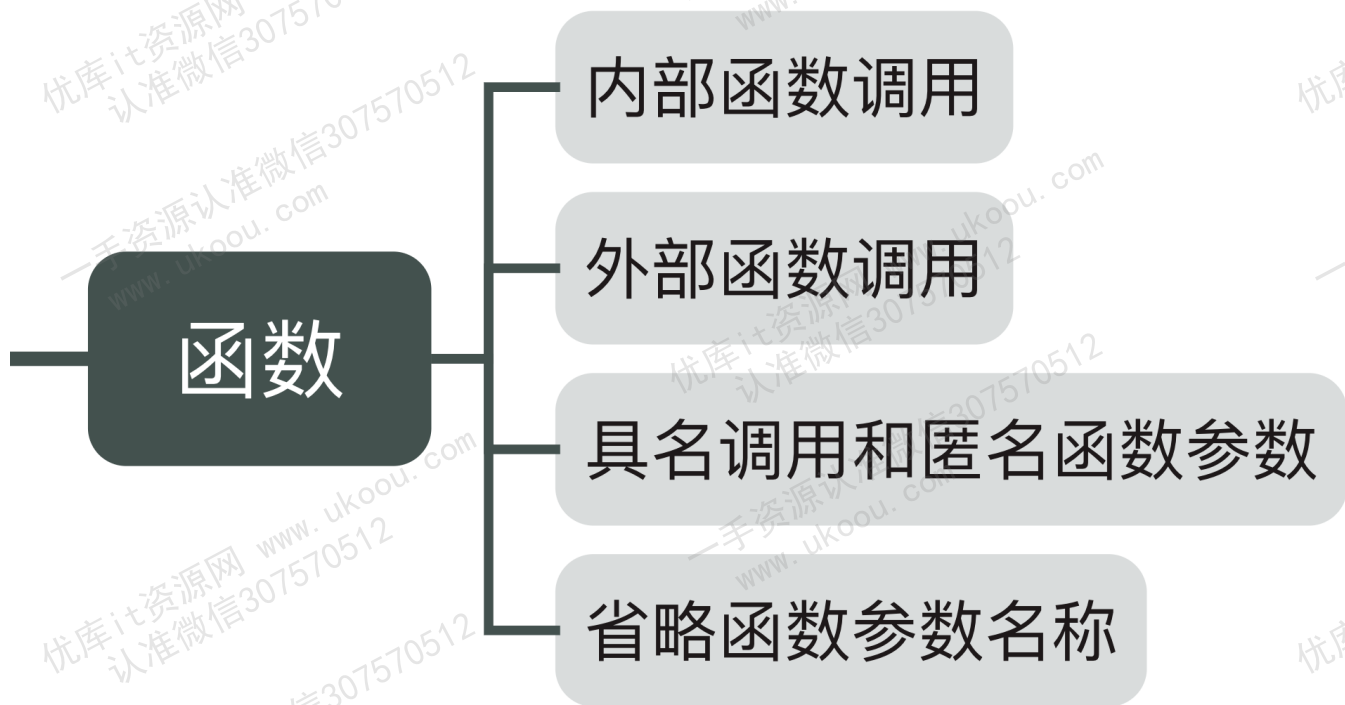
函数类型

引用类型

映射

Getter 函数

函数



函数修饰器

```
pragma solidity ^0.4.22;

contract Purchase {
    address public seller;

    modifier onlySeller() { // 修饰器
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public onlySeller { // Modifier usage
        // ...
    }
}
```

事件

```
pragma solidity ^0.4.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // 我们可以过滤对 `Deposit` 的调用, 从而用 Javascript API 来查明对这个函数的任何调用 (甚至是深度嵌套)
        Deposit(msg.sender, _id, msg.value);
    }
}
```

事件调用

```
var abi = /* abi 由编译器产生 */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* 地址 */);

var event = clientReceipt.Deposit();

// 监视变化
event.watch(function(error, result){
    // 结果包括对 `Deposit` 的调用参数在内的各种信息。
    if (!error)
        console.log(result);
});

// 或者通过回调立即开始观察
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

结构类型

```
pragma solidity ^0.4.0;

contract Ballot {
    struct Voter { // 结构
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

枚举

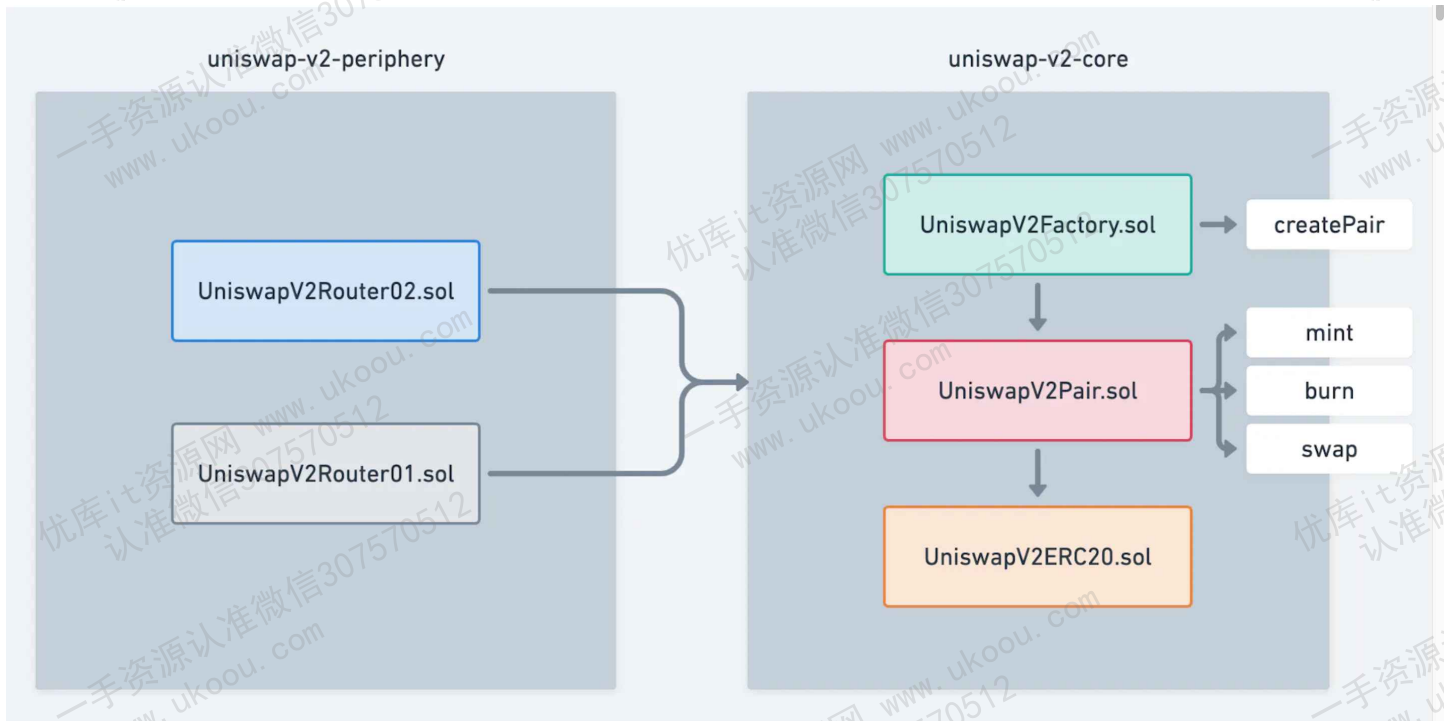
```
pragma solidity ^0.4.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // 枚举
}
```

三、解读 Uniswap v2 合约代码

合约架构

Uniswap v2 的合约主要分为两类：core 合约和 periphery 合约。其中，core 合约仅包含最基础的交易功能，核心代码仅 200 行左右，由于用户资金都存储在 core 合约里，因此需要保证 core 合约最简化，避免引入 bug；periphery 合约则针对用户使用场景提供多种封装方法，比如支持原生 ETH 交易（自动转为 WETH），多路径交换（一个方法同时执行 $A \rightarrow B \rightarrow C$ 交易）等，其底层调用的是 core 合约。我们在 app.uniswap.org 界面操作时用的就是 periphery 合约。



我们先介绍几个主要合约的功能：

- uniswap-v2-core
 - UniswapV2Factory：工厂合约，用于创建 Pair 合约（以及设置协议手续费接收地址）
 - UniswapV2Pair：Pair（交易对）合约，定义和交易有关的几个最基础方法，如 swap/mint/burn，价格预言机等功能，其本身是一个 ERC20 合约，继承 UniswapV2ERC20
 - UniswapV2ERC20：实现 ERC20 标准方法
- uniswap-v2-periphery

- UniswapV2Router02: 最新版的路由合约, 相比 UniswapV2Router01 增加了对 FeeOnTransfer 代币的支持; 实现 Uniswap v2 最常用的接口, 比如添加/移除流动性, 使用代币 A 交换代币 B, 使用 ETH 交换代币等
- UniswapV1Router01: 旧版本 Router 实现, 与 Router02 类似, 但不支持 FeeOnTransferTokens, 目前已不使用

uniswap-v2-core

代码地址: <https://github.com/Uniswap/v2-core>

UniswapV2Factory

在工厂合约中最重要的是 createPair 方法:

```
1 function createPair(address tokenA, address tokenB) external returns (address
   pair) {
2     require(tokenA != tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
3     (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) :
   (tokenB, tokenA);
4     require(token0 != address(0), 'UniswapV2: ZERO_ADDRESS');
5     require(getPair[token0][token1] == address(0), 'UniswapV2: PAIR_EXISTS');
   // single check is sufficient
6     bytes memory bytecode = type(UniswapV2Pair).creationCode;
7     bytes32 salt = keccak256(abi.encodePacked(token0, token1));
8     assembly {
9         pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
10    }
11    IUniswapV2Pair(pair).initialize(token0, token1);
12    getPair[token0][token1] = pair;
13    getPair[token1][token0] = pair; // populate mapping in the reverse
   direction
14    allPairs.push(pair);
15    emit PairCreated(token0, token1, pair, allPairs.length);
16 }
```

首先将 token0 token1 按照顺序排序, 确保 token0 字面地址小于 token1。接着使用 assembly + create2 创建合约。assembly 可以在 Solidity 中使用 Yul 语言直接操作 EVM, 是较底层的操作方法。我们在《深入理解 Uniswap v2 白皮书》中讲到, create2 主要用于创建确定性的交易对合约地址, 目的是根据两个代币地址直接计算 pair 地址, 而无需调用链上合约查询。

CREATE2 出自 EIP-1014, 根据规范, 这里能够影响最终生成地址的是用户自定义的 salt 值, 只需要保证每次生成交易对合约时提供的 salt 值不同即可, 对于同一个交易对的两种代币, 其 salt 值应该一样; 这里很容易想到应该使用交易对的两种代币地址, 我们希望提供 A/B 地址的时候可以直接算出 pair(A,B), 而两个地址又受顺序影响, 因此在合约开始时先对两种代币进行排序, 确保其按照从小到大的顺序生成 salt 值。

实际上在最新版的 EMV 中, 已经直接支持给 new 方法传递 salt 参数, 如下所示:

```
1 pair = new UniswapV2Pair{salt: salt}();
```

因为 Uniswap v2 合约在开发时还没有这个功能, 所以使用 assembly create2。

根据 Yul 规范, create2 的定义如下:

```
1 create2(v, p, n, s)
2
3 create new contract with code mem[p...(p+n)) at address keccak256(0xff . this .
  s . keccak256(mem[p...(p+n))) and send v wei and return the new address, where
  0xff is a 1 byte value, this is the current contract's address as a 20 byte
  value and s is a big-endian 256-bit value; returns 0 on error
```

源码中调用 create2 方法:

```
1 pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
```


因此，这几个参数含义如下：

- `v=0`：向新创建的 pair 合约中发送的 ETH 代币数量（单位 wei）
- `p=add(bytecode, 32)`：合约字节码的起始位置
 - 此处为什么要 add 32 呢？因为 bytecode 类型为 bytes，根据 ABI 规范，bytes 为变长类型，在编码时前 32 个字节存储 bytecode 的长度，接着才是 bytecode 的真正内容，因此合约字节码的起始位置在 bytecode+32 字节
- `n=mload(bytecode)`：合约字节码总字节长度
 - 根据上述说明，bytecode 前 32 个字节存储合约字节码的真正长度（以字节为单位），而 mload 的作用正是读出传入参数的前 32 个字节的值，因此 mload(bytecode)就等于 n
- `s=salt`：s 为自定义传入的 salt，即 token0 和 token1 合并编码

UniswapV2ERC20

这个合约主要定义了 UniswapV2 的 ERC20 标准实现，代码比较简单。这里介绍下 permit 方法：

```
1 function permit(address owner, address spender, uint value, uint deadline,
  uint8 v, bytes32 r, bytes32 s) external {
2   require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
3   bytes32 digest = keccak256(
4     abi.encodePacked(
5       '\x19\x01',
6       DOMAIN_SEPARATOR,
7       keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
  nonces[owner]++, deadline))
8     )
9   );
10  address recoveredAddress = ecrecover(digest, v, r, s);
11  require(recoveredAddress != address(0) && recoveredAddress == owner,
  'UniswapV2: INVALID_SIGNATURE');
12  _approve(owner, spender, value);
13 }
```

permit 方法实现的就是白皮书 2.5 节中介绍的“Meta transactions for pool shares 元交易”功能。EIP-712 定义了离线签名的规范，即 digest 的格式定义，用户签名的内容是其（owner）授权（approve）某个合约（spender）可以在截止时间（deadline）之前花掉一定数量（value）的代币（Pair 流动性代币），应用（periphery 合约）拿着签名的原始信息和签名后生成的 v, r, s，可以调用 Pair 合约的 permit 方法获得授权，permit 方法使用 ecrecover 还原出签名地址为代币所有人，验证通过则批准授权。

UniswapV2Pair

Pair 合约主要实现了三个方法：mint（添加流动性）、burn（移除流动性）、swap（兑换）。

mint

该方法实现添加流动性功能。

```
1 // this low-level function should be called from a contract which performs
  important safety checks
2 function mint(address to) external lock returns (uint liquidity) {
3     (uint112 _reserve0, uint112 _reserve1) = getReserves(); // gas savings
4     uint balance0 = IERC20(token0).balanceOf(address(this));
5     uint balance1 = IERC20(token1).balanceOf(address(this));
6     uint amount0 = balance0.sub(_reserve0);
7     uint amount1 = balance1.sub(_reserve1);
8
9     bool feeOn = _mintFee(_reserve0, _reserve1);
10    uint _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
11    if (_totalSupply == 0) {
12        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
13        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
        MINIMUM_LIQUIDITY tokens
14    } else {
15        liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0,
        amount1.mul(_totalSupply) / _reserve1);
16    }
17    require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
18    _mint(to, liquidity);
```

```

19
20 _update(balance0, balance1, _reserve0, _reserve1);
21 if (fee0n) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1
    are up-to-date
22     emit Mint(msg.sender, amount0, amount1);
23 }

```

首先, getReserves() 获取两种代币的缓存余额。在白皮书中提到, 保存缓存余额是为了防止攻击者操控价格预言机。此处还用于计算协议手续费, 并通过当前余额与缓存余额相减获得转账的代币数量。

_mintFee 用于计算协议手续费:

```

1 // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
2 function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool
    fee0n) {
3     address feeTo = IUniswapV2Factory(factory).feeTo();
4     fee0n = feeTo != address(0);
5     uint _kLast = kLast; // gas savings
6     if (fee0n) {
7         if (_kLast != 0) {
8             uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
9             uint rootKLast = Math.sqrt(_kLast);
10            if (rootK > rootKLast) {
11                uint numerator = totalSupply.mul(rootK.sub(rootKLast));
12                uint denominator = rootK.mul(5).add(rootKLast);
13                uint liquidity = numerator / denominator;
14                if (liquidity > 0) _mint(feeTo, liquidity);
15            }
16        }
17    } else if (_kLast != 0) {
18        kLast = 0;
19    }
20 }

```

关于协议手续费的计算公式可以参考白皮书。

mint 方法中判断,如果是首次提供该交易对的流动性,则根据根号 xy 生成流动性代币,并销毁其中的 MINIMUM_LIQUIDITY (即 1000wei); 否则根据转入的代币价值与当前流动性价值比例铸造流动性代币。

burn

该方法实现移除流动性功能。

```
1 // this low-level function should be called from a contract which performs
  important safety checks
2 function burn(address to) external lock returns (uint amount0, uint amount1) {
3     (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
4     address _token0 = token0; // gas savings
5     address _token1 = token1; // gas savings
6     uint balance0 = IERC20(_token0).balanceOf(address(this));
7     uint balance1 = IERC20(_token1).balanceOf(address(this));
8     uint liquidity = balanceOf[address(this)];
9
10    bool feeOn = _mintFee(_reserve0, _reserve1);
11    uint _totalSupply = totalSupply; // gas savings, must be defined here
    since totalSupply can update in _mintFee
12    amount0 = liquidity.mul(balance0) / _totalSupply; // using balances
    ensures pro-rata distribution
13    amount1 = liquidity.mul(balance1) / _totalSupply; // using balances
    ensures pro-rata distribution
14    require(amount0 > 0 && amount1 > 0, 'UniswapV2:
    INSUFFICIENT_LIQUIDITY_BURNED');
15    _burn(address(this), liquidity);
16    _safeTransfer(_token0, to, amount0);
17    _safeTransfer(_token1, to, amount1);
18    balance0 = IERC20(_token0).balanceOf(address(this));
19    balance1 = IERC20(_token1).balanceOf(address(this));
20
21    _update(balance0, balance1, _reserve0, _reserve1);
22    if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1
    are up-to-date
23    emit Burn(msg.sender, amount0, amount1, to);
24 }
```

与 mint 类似, burn 方法也会先计算协议手续费。

参考白皮书, 为了节省交易手续费, Uniswap v2 只在 mint/burn 流动性时收取累计的协议手续费。

移除流动性后, 根据销毁的流动性代币占总量的比例获得对应的两种代币。

swap

该方法实现两种代币的交换 (交易) 功能。

```
1 // this low-level function should be called from a contract which performs
  important safety checks
2 function swap(uint amount0Out, uint amount1Out, address to, bytes calldata
  data) external lock {
3     require(amount0Out > 0 || amount1Out > 0, 'UniswapV2:
  INSUFFICIENT_OUTPUT_AMOUNT');
4     (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
5     require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2:
  INSUFFICIENT_LIQUIDITY');
6
7     uint balance0;
8     uint balance1;
9     { // scope for _token{0,1}, avoids stack too deep errors
10        address _token0 = token0;
11        address _token1 = token1;
12        require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
13        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); //
  optimistically transfer tokens
14        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); //
  optimistically transfer tokens
15        if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender,
  amount0Out, amount1Out, data);
16        balance0 = IERC20(_token0).balanceOf(address(this));
17        balance1 = IERC20(_token1).balanceOf(address(this));
18    }
19    uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0
  - amount0Out) : 0;
20    uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1
  - amount1Out) : 0;
```



```

21     require(amount0In > 0 || amount1In > 0, 'UniswapV2:
    INSUFFICIENT_INPUT_AMOUNT');
22     { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
23         uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
24         uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
25         require(balance0Adjusted.mul(balance1Adjusted) >=
    uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
26     }
27
28     _update(balance0, balance1, _reserve0, _reserve1);
29     emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
30 }

```

为了兼容闪电贷功能，以及不依赖特定代币的 transfer 方法，整个 swap 方法并没有类似 amountIn 的参数，而是通过比较当前余额与缓存余额的差值来得出转入的代币数量。

由于在 swap 方法最后会检查余额（扣掉手续费后）符合 k 恒等式约束（参考白皮书公式），因此合约可以先将用户希望获得的代币转出，如果用户之前并没有向合约转入用于交易的代币，则相当于借币（即闪电贷）；如果使用闪电贷，则需要在自定义的 uniswapV2Call 方法中将借出的代币归还。

在 swap 方法最后会使用缓存余额更新价格预言机所需的累计价格，最后更新缓存余额为当前余额。

```

1 // update reserves and, on the first call per block, price accumulators
2 function _update(uint balance0, uint balance1, uint112 _reserve0, uint112
    _reserve1) private {
3     require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'UniswapV2:
    OVERFLOW');
4     uint32 blockTimestamp = uint32(block.timestamp % 2**32);
5     uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is
    desired
6     if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
7         // * never overflows, and + overflow is desired
8         price0CumulativeLast +=
    uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
9         price1CumulativeLast +=
    uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
10    }
11    reserve0 = uint112(balance0);
12    reserve1 = uint112(balance1);

```

```
13     blockTimestampLast = blockTimestamp;  
14     emit Sync(reserve0, reserve1);  
15 }
```

注意，其中区块时间戳和累计价格都是溢出安全的。（具体推导过程请参考白皮书）

uniswap-v2-periphery

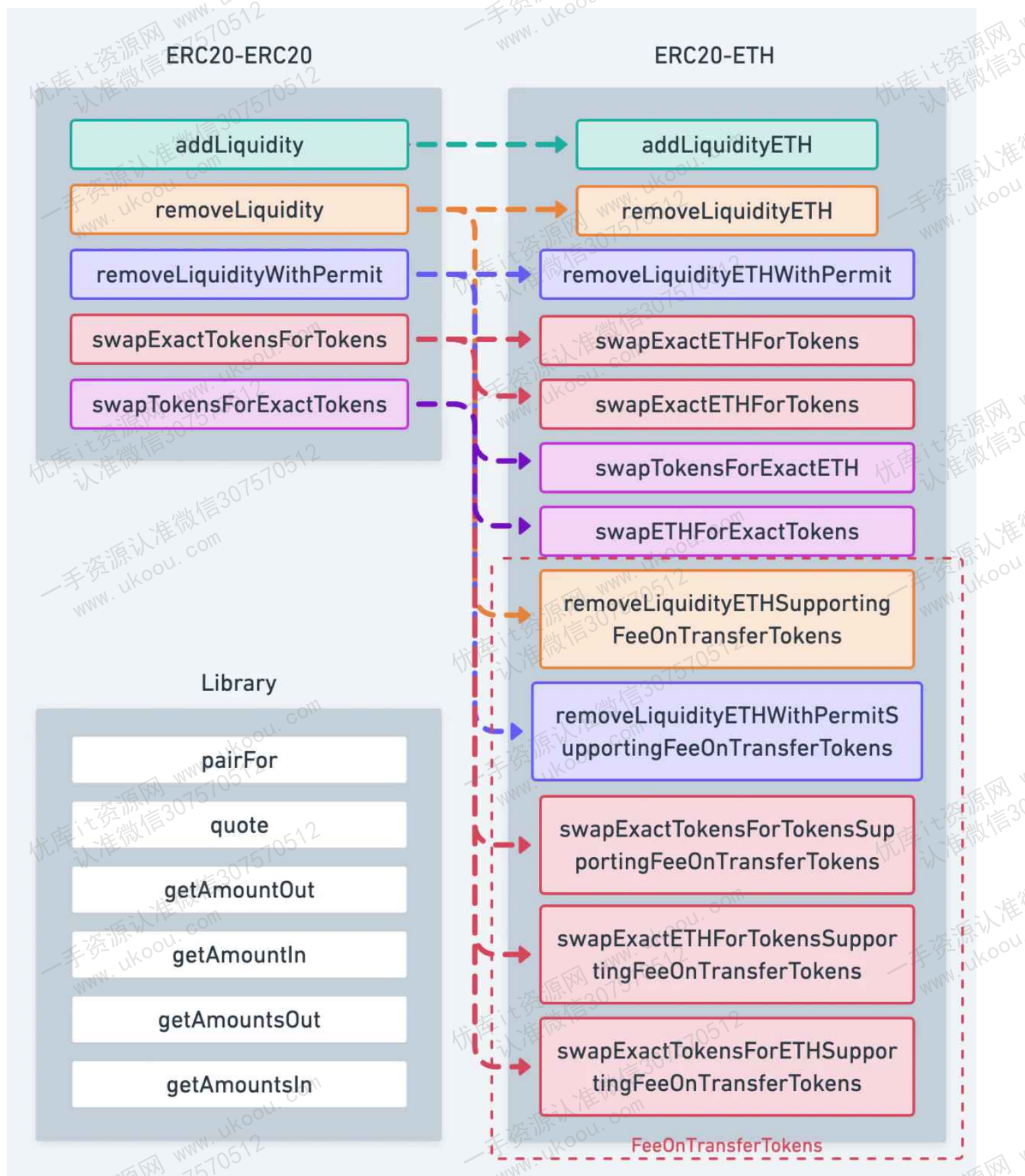
由于 UniswapV2Router01 在处理 FeeOnTransferTokens 时有 bug，目前已不再使用。此处我们仅介绍最新版的 UniswapV2Router02 合约。

代码地址：

<https://github.com/Uniswap/v2-periphery>

UniswapV2Router02

Router02 封装了最常用的几个交易接口；为了满足原生 ETH 交易需求，大部分接口都支持 ETH 版本；同时，相比 Router01，部分接口增加了 FeeOnTransferTokens 的支持。



我们将主要介绍 ERC20 版本的代码，因为 ETH 版本只是将 ETH 与 WETH 做转换，逻辑与 ERC20 一致。

在介绍具体 ERC20 方法前，我们先介绍 Library 合约中的几个常用方法，以及它们的数学公式推导。

Library

代码地址:

<https://github.com/Uniswap/v2-periphery/blob/master/contracts/libraries/UniswapV2Library.sol>

pairFor

输入工厂地址和两个代币地址, 计算这两个代币的交易对地址。

```
1 // calculates the CREATE2 address for a pair without making any external calls
2 function pairFor(address factory, address tokenA, address tokenB) internal
  pure returns (address pair) {
3     (address token0, address token1) = sortTokens(tokenA, tokenB);
4     pair = address(uint(keccak256(abi.encodePacked(
5         hex'ff',
6         factory,
7         keccak256(abi.encodePacked(token0, token1)),
8         hex'96e8ac4277198ff8b6f785478aa9a39f403cb768dd02cbee326c3e7da348845f' // init
9         code hash
10    ))));
```

上文提到, 由于使用 CREATE2 操作码, 交易对地址可以直接根据规范算出, 而无需调用链上合约进行查询。

```
1 create2(v, p, n, s)
2
3 create new contract with code mem[p...(p+n)) at address keccak256(0xff . this .
  s . keccak256(mem[p...(p+n))) and send v wei and return the new address, where
  0xff is a 1 byte value, this is the current contract's address as a 20 byte
  value and s is a big-endian 256-bit value; returns 0 on error
```

其中, 新创建的 pair 合约的地址计算方法为: $\text{keccak256}(0\text{xff} + \text{this} + \text{salt} + \text{keccak256}(\text{mem}[\text{p} \dots (\text{p} + \text{n})]))$:

- this: 工厂合约地址
- salt: $\text{keccak256}(\text{abi.encodePacked}(\text{token0}, \text{token1}))$
- $\text{keccak256}(\text{mem}[\text{p} \dots (\text{p} + \text{n})])$:
0x96e8ac4277198ff8b6f785478aa9a39f403cb768dd02cbee326c3e7da348845f

由于每个交易对都使用 UniswapV2Pair 合约创建, 因此 init code hash 都是一样的。我们可以在 UniswapV2Factory 写一个 Solidity 方法计算 hash:

```
1 function initCodeHash() external pure returns (bytes32) {
2     bytes memory bytecode = type(UniswapV2Pair).creationCode;
3     bytes32 hash;
4     assembly {
5         hash := keccak256(add(bytecode, 32), mload(bytecode))
6     }
7     return hash;
8 }
```

quote

quote 方法将数量为 amountA 的代币 A, 按照合约中两种代币余额比例, 换算成另一个代币 B。此时不考虑手续费, 因为仅是计价单位的换算。

```
1 // given some amount of an asset and pair reserves, returns an equivalent
   amount of the other asset
2 function quote(uint amountA, uint reserveA, uint reserveB) internal pure
   returns (uint amountB) {
3     require(amountA > 0, 'UniswapV2Library: INSUFFICIENT_AMOUNT');
```



```
4     require(reserveA > 0 && reserveB > 0, 'UniswapV2Library:
INSUFFICIENT_LIQUIDITY');
5     amountB = amountA.mul(reserveB) / reserveA;
6 }
```

getAmountOut

该方法计算：输入一定数量（amountIn）代币 A，根据池子中代币余额，能得到多少数量（amountOut）代币 B。

```
1 // given an input amount of an asset and pair reserves, returns the maximum
output amount of the other asset
2 function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal
pure returns (uint amountOut) {
3     require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
4     require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
INSUFFICIENT_LIQUIDITY');
5     uint amountInWithFee = amountIn.mul(997);
6     uint numerator = amountInWithFee.mul(reserveOut);
7     uint denominator = reserveIn.mul(1000).add(amountInWithFee);
8     amountOut = numerator / denominator;
9 }
```

为了推导该方法的数学公式，我们需要先回顾白皮书以及 core 合约中对于 swap 交换后两种代币的约束：

其中，x0, y0 为交换前的两种代币余额，x1, y1 为交换后的两种代币余额，xin 为输入的代币 A 数量，因为只提供代币 A，因此 yin=0；yout 为需要计算的代币 B 数量。

可推导数学公式如下：

$$y_{in} = 0$$

$$x_1 = x_0 + x_{in}$$

$$y_1 = y_0 - y_{out}$$

$$(x_1 - 0.003 * x_{in}) * (y_1 - 0.003 * y_{in}) = x_0 * y_0$$

$$(x_1 - 0.003 * x_{in}) * y_1 = x_0 * y_0$$

$$(x_0 + x_{in} - 0.003 * x_{in}) * (y_0 - y_{out}) = x_0 * y_0$$

$$(x_0 + 0.997 * x_{in}) * (y_0 - y_{out}) = x_0 * y_0$$

$$y_{out} = y_0 - \frac{x_0 * y_0}{x_0 + 0.997 * x_{in}}$$

$$y_{out} = \frac{0.997 * x_{in} * y_0}{x_0 + 0.997 * x_{in}}$$

由于 Solidity 不支持浮点数，因此可以换算成如下公式：

$$y_{out} = \frac{997 * x_{in} * y_0}{1000 * x_0 + 997 * x_{in}}$$

可以看出，该计算结果即为 getAmountOut 方法中的 amountOut，其中，

$$\begin{aligned} \text{amountIn} &= x_{in} \\ \text{reserveIn} &= x_0 \\ \text{reserveOut} &= y_0 \\ \text{amountOut} &= y_{out} \end{aligned}$$

getAmountIn

该方法计算当希望获得一定数量 (amountOut) 的代币 B 时, 应该输入多少数量 (amountIn) 的代币 A。

```
1 // given an output amount of an asset and pair reserves, returns a required
   input amount of the other asset
2 function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal
   pure returns (uint amountIn) {
3     require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
4     require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library:
   INSUFFICIENT_LIQUIDITY');
5     uint numerator = reserveIn.mul(amountOut).mul(1000);
6     uint denominator = reserveOut.sub(amountOut).mul(997);
7     amountIn = (numerator / denominator).add(1);
8 }
```

getAmountOut 是已知 x_{in} , 计算 y_{out} ; 相应地, getAmountIn 则是已知 y_{out} , 计算 x_{in} 。根据上述公式可以推导出:

计算结果即为合约中代码所示, 注意最后有一个 add(1), 这是为了防止 amountIn 为小数的情况, 加 1 可以保证输入的数 (amountIn) 不小于理论的最小值。

getAmountsOut

该方法用于计算在使用多个交易对时,输入一定数量 (amountIn) 的第一种代币,最终能收到多少数量的最后一种代币 (amounts)。amounts 数组中的第一个元素表示 amountIn,最后一个元素表示该目标代币对应的数量。该方法实际上是循环调用 getAmountIn 方法。

```
1 // performs chained getAmountOut calculations on any number of pairs
2 function getAmountsOut(address factory, uint amountIn, address[] memory path)
  internal view returns (uint[] memory amounts) {
3   require(path.length >= 2, 'UniswapV2Library: INVALID_PATH');
4   amounts = new uint[](path.length);
5   amounts[0] = amountIn;
6   for (uint i; i < path.length - 1; i++) {
7     (uint reserveIn, uint reserveOut) = getReserves(factory, path[i],
      path[i + 1]);
8     amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut);
9   }
10 }
```

getAmountsIn

与 getAmountsOut 相对, getAmountsIn 用于计算当希望收到一定数量 (amountOut) 的目标代币,应该分别输入多少数量的中间代币。计算方法也是循环调用 getAmountIn。

```
1 // performs chained getAmountIn calculations on any number of pairs
2 function getAmountsIn(address factory, uint amountOut, address[] memory path)
  internal view returns (uint[] memory amounts) {
3   require(path.length >= 2, 'UniswapV2Library: INVALID_PATH');
4   amounts = new uint[](path.length);
5   amounts[amounts.length - 1] = amountOut;
6   for (uint i = path.length - 1; i > 0; i--) {
7     (uint reserveIn, uint reserveOut) = getReserves(factory, path[i - 1],
      path[i]);
8     amounts[i - 1] = getAmountIn(amounts[i], reserveIn, reserveOut);
9   }
10 }
```

```
9     }  
10 }
```

ERC20-ERC20

addLiquidity 添加流动性

```
1 function addLiquidity(  
2     address tokenA,  
3     address tokenB,  
4     uint amountADesired,  
5     uint amountBDesired,  
6     uint amountAMin,  
7     uint amountBMin,  
8     address to,  
9     uint deadline  
10 ) external virtual override ensure(deadline) returns (uint amountA, uint  
    amountB, uint liquidity) {  
11     (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,  
        amountBDesired, amountAMin, amountBMin);  
12     address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);  
13     TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);  
14     TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);  
15     liquidity = IUniswapV2Pair(pair).mint(to);  
16 }
```

由于 Router02 是直接与用户交互的，因此接口设计需要从用户使用场景考虑。addLiquidity 提供了 8 个参数：

- address tokenA：代币 A
- address tokenB：代币 B
- uint amountADesired：希望存入的代币 A 数量
- uint amountBDesired：希望存入的代币 B 数量
- uint amountAMin：最少存入的代币 A 数量

- uint amountBMin: 最少存入的代币 B 数量
- address to: 流动性代币接收地址
- uint deadline: 请求失效时间

用户提交交易后, 该交易被矿工打包的时间是不确定的, 因此提交时的代币价格与交易打包时的价格可能不同, 通过 amountMin 可以控制价格的浮动范围, 防止被矿工或机器人套利; 同样, deadline 可以确保该交易在超过指定时间后将失效。

在 core 合约中提到, 如果用户提供流动性时的代币价格与实际价格有差距, 则只会按照较低的汇率得到流动性代币, 多余的代币将贡献给整个池子。_addLiquidity 可以帮助计算最佳汇率。如果是首次添加流动性, 则会先创建交易对合约; 否则根据当前池子余额计算应该注入的最佳代币数量。

```
1 /**
2  * ADD LIQUIDITY
3  * create the pair if it does not exist yet
4  */
5 function _addLiquidity(
6     address tokenA,
7     address tokenB,
8     uint amountADesired,
9     uint amountBDesired,
10    uint amountAMin,
11    uint amountBMin
12 ) internal virtual returns (uint amountA, uint amountB) {
13     if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
14         IUniswapV2Factory(factory).createPair(tokenA, tokenB);
15     }
16     (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory,
17     tokenA, tokenB);
18     if (reserveA == 0 && reserveB == 0) {
19         (amountA, amountB) = (amountADesired, amountBDesired);
20     } else {
21         uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
22         reserveB);
23         if (amountBOptimal <= amountBDesired) {
24             require(amountBOptimal >= amountBMin, 'UniswapV2Router:
25             INSUFFICIENT_B_AMOUNT');
26             (amountA, amountB) = (amountADesired, amountBOptimal);
27         } else {
28             uint amountAOptimal = UniswapV2Library.quote(amountBDesired,
29             reserveB, reserveA);
30             require(amountAOptimal >= amountAMin, 'UniswapV2Router:
31             INSUFFICIENT_A_AMOUNT');
32             (amountA, amountB) = (amountAOptimal, amountBDesired);
33         }
34     }
35 }
```

```
26     assert(amountAOptimal <= amountADesired);
27     require(amountAOptimal >= amountAMin, 'UniswapV2Router:
    INSUFFICIENT_A_AMOUNT');
28     (amountA, amountB) = (amountAOptimal, amountBDesired);
29 }
30 }
31 }
```

最后调用 core 合约 mint 方法铸造流动性代币。

removeLiquidity 移除流动性

首先将流动性代币发送到 pair 合约，根据收到的流动性代币占全部代币比例，计算该流动性代表的两种代币数量。合约销毁流动性代币后，用户将收到对应比例的代币。如果低于用户设定的最低预期（amountAMin/amountBMin），则回滚交易。

```
1 // **** REMOVE LIQUIDITY ****
2 function removeLiquidity(
3     address tokenA,
4     address tokenB,
5     uint liquidity,
6     uint amountAMin,
7     uint amountBMin,
8     address to,
9     uint deadline
10 ) public virtual override ensure(deadline) returns (uint amountA, uint
    amountB) {
11     address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
12     IUniswapV2Pair(pair).transferFrom(msg.sender, pair, liquidity); // send
    liquidity to pair
13     (uint amount0, uint amount1) = IUniswapV2Pair(pair).burn(to);
14     (address token0,) = UniswapV2Library.sortTokens(tokenA, tokenB);
15     (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1,
    amount0);
16     require(amountA >= amountAMin, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
17     require(amountB >= amountBMin, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');
18 }
```

removeLiquidityWithPermit 使用签名移除流动性

用户正常移除流动性时，需要两个操作：

1. approve：授权 Router 合约花费自己的流动性代币
2. removeLiquidity：调用 Router 合约移除流动性

除非第一次授权了最大限额的代币，否则每次移除流动性都需要两次交互，这意味着用户需要支付两次手续费。而使用 removeLiquidityWithPermit 方法，用户可以通过签名方式授权 Router 合约花费自己的代币，无需单独调用 approve，只需要调用一次移除流动性方法即可完成操作，节省了 gas 费用。同时，由于离线签名不需要花费 gas，因此可以每次签名仅授权一定额度的代币，提高安全性。

```
1 function removeLiquidityWithPermit(  
2     address tokenA,  
3     address tokenB,  
4     uint liquidity,  
5     uint amountAMin,  
6     uint amountBMin,  
7     address to,  
8     uint deadline,  
9     bool approveMax, uint8 v, bytes32 r, bytes32 s  
10 ) external virtual override returns (uint amountA, uint amountB) {  
11     address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);  
12     uint value = approveMax ? uint(-1) : liquidity;  
13     IUniswapV2Pair(pair).permit(msg.sender, address(this), value, deadline, v,  
14     r, s);  
14     (amountA, amountB) = removeLiquidity(tokenA, tokenB, liquidity,  
15     amountAMin, amountBMin, to, deadline);  
15 }
```

swapExactTokensForTokens

交易时的两个常见场景：

1. 使用指定数量的代币 A（输入），尽可能兑换最多数量的代币 B（输出）

2. 获得指定数量的代币 B（输出），尽可能使用最少数量的代币 A（输入）

本方法实现第一个场景，即根据指定的输入代币，获得最多的输出代币。

```

1 function swapExactTokensForTokens(
2     uint amountIn,
3     uint amountOutMin,
4     address[] calldata path,
5     address to,
6     uint deadline
7 ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
8     amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
9     require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router:
    INSUFFICIENT_OUTPUT_AMOUNT');
10    TransferHelper.safeTransferFrom(
11        path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0],
12        path[1]), amounts[0]
13    );
14    _swap(amounts, path, to);
15 }
```

首先使用 Library 合约中的 `getAmountsOut` 方法，根据兑换路径计算每一次交易的输出代币数量，确认最后一次交易得到的数量（`amounts[amounts.length - 1]`）不小于预期最少输出（`amountOutMin`）；将代币发送到第一个交易对地址，开始执行整个兑换交易。

假设用户希望使用 WETH 兑换 DYDX，链下计算的最佳兑换路径为 WETH → USDC → DYDX，则 `amountIn` 为 WETH 数量，`amountOutMin` 为希望获得最少 DYDX 数量，`path` 为 [WETH address, USDC address, DYDX address]，`amounts` 为 [amountIn, USDC amount, DYDX amount]。在 `_swap` 执行交易的过程中，每次中间交易获得的中间代币将被发送到下一个交易对地址，以此类推，直到最后一个交易完成，`_to` 地址将收到最后一次交易的输出代币。

```

1 //requires the initial amount to have already been sent to the first pair
2 function _swap(uint[] memory amounts, address[] memory path, address _to)
    internal virtual {
3     for (uint i; i < path.length - 1; i++) {
4         (address input, address output) = (path[i], path[i + 1]);
5         (address token0,) = UniswapV2Library.sortTokens(input, output);
6         uint amountOut = amounts[i + 1];
```

```
7      (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0),
      amountOut) : (amountOut, uint(0));
8      address to = i < path.length - 2 ? UniswapV2Library.pairFor(factory,
      output, path[i + 2]) : _to;
9      IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output)).swap(
10     amount0Out, amount1Out, to, new bytes(0)
11   );
12   }
13 }
```

swapTokensForExactTokens

该方法实现交易的第二个场景，根据指定的输出代币，使用最少的输入代币完成兑换。

```
1 function swapTokensForExactTokens(
2     uint amountOut,
3     uint amountInMax,
4     address[] calldata path,
5     address to,
6     uint deadline
7 ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
8     amounts = UniswapV2Library.getAmountsIn(factory, amountOut, path);
9     require(amounts[0] <= amountInMax, 'UniswapV2Router:
10    EXCESSIVE_INPUT_AMOUNT');
11     TransferHelper.safeTransferFrom(
12     path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0],
13     path[1]), amounts[0]
14   );
15     _swap(amounts, path, to);
16 }
```

与上面类似，这里先使用 Library 的 `getAmountsIn` 方法反向计算每一次兑换所需的最少输入代币数量，确认计算得出的（扣除手续费后）第一个代币所需的最少代币数不大于用户愿意提供的最大代币数（`amountInMax`）；将代币发送到第一个交易对地址，调用 `_swap` 开始执行整个兑换交易。

ERC20-ETH

ETH Support

由于 core 合约只支持 ERC20 代币交易，为了支持 ETH 交易，periphery 合约需要将 ETH 与 WETH 做转换；并为大部分方法提供了 ETH 版本。兑换主要涉及两种操作：

地址转换：由于 ETH 没有合约地址，因此需要使用 WETH 合约的 deposit 和 withdraw 方法完成 ETH 与 WETH 的兑换

代币数量转换：ETH 的代币需要通过 msg.value 获取，可根据该值计算对应的 WETH 数量，而后使用标准 ERC20 接口即可

FeeOnTransferTokens

由于某些代币会在转账（transfer）过程中收取手续费，转账数量与实际收到的数量有差异，因此无法直接通过计算得出中间兑换过程中所需的代币数量，此时应该通过 balanceOf 方法（而非 transfer 方法）判断实际收到的代币数量。Router02 新增了对 Inclusive Fee On Transfer Tokens 的支持，更具体说明可以参考官方文档

<https://docs.uniswap.org/contracts/v2/reference/smart-contracts/common-errors>。