获取报价

AMM 自动做市商协议

简介

在流动性池中，存入与撤出代币的数量关系遵循 AMM 协议，可以简单理解为一个 xy = k 的函数曲线，该函数曲线拥有的最大特点就是：x 或者 y 轴所代表的代币数量永远不会减少到 0（这也是保持永久流动性的思想基础）。

AMM 将两种需要兑换的代币放到一个流动性池中，比如代币 a 和代币 b；当用户想要用代币 a 兑换代币 b 时，先将代币 a 存入流动性池中，然后使用一个计算公式得出需要从流动性池中直接转移给用户的代币 b 的数量，这样就完成了交易，而且不需要等待时间（保证了兑换完成后，代币 a 和代币 b 的数量之积仍然是一个恒定的常数）。

> 🦄 核心思想：智能合约使用约定好的规则自动处理交易，并调整价格

关键公式

> $x * y = k$

我们首先不考虑手续费，推导兑换代币的数量关系公式。

在一个常数为 k 的流动性池中，假设拥有两种代币 x 和 y，满足 x * y = k；在这种情况下，假设我们往流动性池中存入了 Δx 个 a 代币，那么可以兑换出多少个 b 代币呢？

我们知道兑换前后，k 值应该保持不变，所以应有以下公式成立：

$$(x + \Delta x) * (y - \Delta y) = k$$

其中 Δy 就是可以兑换的 y 代币的数量，整理得：

$$\Delta y = y - \frac{k}{x + \Delta x} \quad (⭐)$$

有了这个公式，我们举个例子示范一下：假设现在流动性池中有 5000 DAI 和 10 ETH （500 DAI : 1 ETH），满足的 k 值 = 5000 * 10 = 50000；我们现在想要按照市场价值使用 500DAI 从流动性池中兑换 1 个 ETH，但是我们根据上面推导出来公式计算出可以兑换的代币数量为：

$$(5000 + 500) * (10 - \Delta y) = 50000 \quad \Delta y = 0.909$$

500DAI 只能兑换出 0.909 个 ETH， 而这部分差值我们通常称为**滑点**。

这里的滑点达到了 10%，主要是因为流动性池中的流动性不足。

🏆 根据公式可以推知：交易者存入了更多的 DAI，取出了 ETH。流动性池中的 ETH 数量相对于 DAI 是减少的，那么以太的价格就升高了，同等数量的 DAI 币就无法兑换原来可以兑换的 ETH。

我们已经推导出在不考虑手续费情况下的兑换公式，现在我们继续推导考虑手续费情况下的兑换公式（uniswap 收取 0.3% 的手续费）。假设场景：用户想要使用 Δx 数量的 a 代币兑换 b 代币，那么用户需要向流动性池中的流动性提供者支付手续费（基于兑换的代币数量），不难得到如下公式：

$$(x + \Delta x - 0.003\Delta x) * (y - \Delta y) = k$$

继续推导得：

$$(x + 0.997\Delta x) * (y - \Delta y) = k$$

$$xy - x\Delta y + 0.997\Delta xy - 0.997\Delta x\Delta y = xy$$

$$\Delta y = \frac{0.997\Delta x * y}{x + 0.997\Delta x} = \frac{997\Delta x * y}{1000x + 997\Delta x} \ (⭐)$$

我们再与官方合约提供的 API 进行对比：

```
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) inernal pure rerturns(uint amount) {
    require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');

    // 997Δx
    uint amountInWithFee = amountIn.mul(997);
    // 分子 997Δx*y
    uint numerator = amountInWithFee.mul(resesrveOut);
    // 分母 1000x + 997Δx
    uint denominator = reserveIn.mul(1000).add(amountInWithFee);
    // Δy
    amountOut = numerator / denominator;
}
```

官方合约的 API 实现思路和我们推导的计算公式是一致的（分子分母同时乘以1000是因为 solidity 中不支持浮点数运算）。

实战

我们在第一部分简单介绍了获取报价的原理，这一部分我们就实际调用一下合约提供的 API ，我们将使用 **Uniswap V2: Router 2** 合约提供的 API ，计算往池中存入一种代币可以取出的另一种代币的数量，又或者是从池中撤出一定量的代币需要存入的另一种代币的数量（比如存入 1 ETH 可以兑换出 1831 USDT）。

> 合约地址：0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D

1. 在 https://etherscan.io/ 搜索 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D 信息
2. 获取合约 ABI
3. 编写 js 脚本
4. 执行 js 脚本获取报价

附录

- 代码部分

```
// uniswapV2 Contract's ABI
const uniswapV2ABI = require("./uniswapV2.json");

// import web3.js
var Web3 = require("web3");

// initiate web3 instance (connected to hosted mainnet node)
const web3 = new Web3("https://cloudflare-eth.com");

// uniswapV2Address
const uniswapV2Address = "0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D"
// uniswapV2Address Object
var priceContractInstance = new web3.eth.Contract(uniswapV2ABI, uniswapV2Address
// call (Use Promise)

// DAI 精度 18 位
// USDT 精度 6 位

// 场景 1：存入 1 DAI，计算可以撤出的 USDT 的数量，因此需要使用方法 getAmountsOut
// DAI 为存入的代币，所以调用形式为：getAmountsOut("1000000000000000000", ['DAI_ADDI
// return amount['store', 'retrieve'] ['dai', 'usdt']
priceContractInstance.methods.getAmountsOut("1000000000000000000", ['0x6B175474E
.then((result) => {
    console.log(`store ${result[0]/1e18} DAI coin, retrieve ${result[1]/1e6} USD
})
.catch((error) => {
    console.error(error);
```

```
})

// ['weth', 'usdt']
priceContractInstance.methods.getAmountsOut("1000000000000000000", ['0xC02aaA39b
.then((result) => {
    console.log(`store ${result[0]/1e18} WETH coin, retrieve ${result[1]/1e6} US
})
.catch((error) => {
    console.error(error);
})
// 场景 2：取出 997651 USDT，计算需要往池中存入的 DAI 的数量，所以需要使用方法 getAmoun
// DAI 为存入代币，USDT 为撤出代币，所以调用形式：getAmountsIn("997651", ['DAI_ADDRES
// return [ 'DAI' '997651000000']
priceContractInstance.methods.getAmountsIn("997651000000", ['0x6B175474E89094C44
.then((result) => {
    console.log(`retrieve ${result[1]/1e6} USDT , store ${result[0]/1e18} DAI`);
})
.catch((error) => {
    console.error(error);
})

// 场景 3：存入 997651 USDT，计算可以撤出的 DAI 的数量，因此需要使用方法 getAmountsOut
// 存入的代币位于前面，所以调用方式为：getAmountsOut('997651000000', ['USDT', 'DAI'],
// return [997651, DAI_Out]
priceContractInstance.methods.getAmountsOut("997651000000", ['0xA0b86991c6218b36
.then((result) => {
    console.log(`store ${result[0]/1e6} USDT coin, retrieve ${result[1]/1e18} DA
})
.catch((error) => {
    console.error(error);
})

// 场景 4：取出 DAI 1000000000000000000，计算需要存入的 USDT 的数量，所以需要调用 getA
// 因为 DAI 为取出代币，所以位于地址数组后面，调用形式为：getAmountsIn('10000000000000
// [USDT, 1000000000000000000]
priceContractInstance.methods.getAmountsIn("1000000000000000000", ['0xA0b86991c6
.then((result) => {
    console.log(`retrieve ${result[1]/1e18} DAI , store ${result[0]/1e6} USDT`);
})
.catch((error) => {
    console.error(error);
})
```

- 运行结果

```
store 1 WETH coin, retrieve 1822.611914 USDT
retrieve 997651 USDT , store 1101733.4898315906 DAI
store 997651 USDT coin, retrieve 911391.4771374861 DAI
store 1 DAI coin, retrieve 0.996945 USDT
retrieve 1 DAI , store 1.002955 USDT
```