UniswapV2Router02 源码深度学习笔记

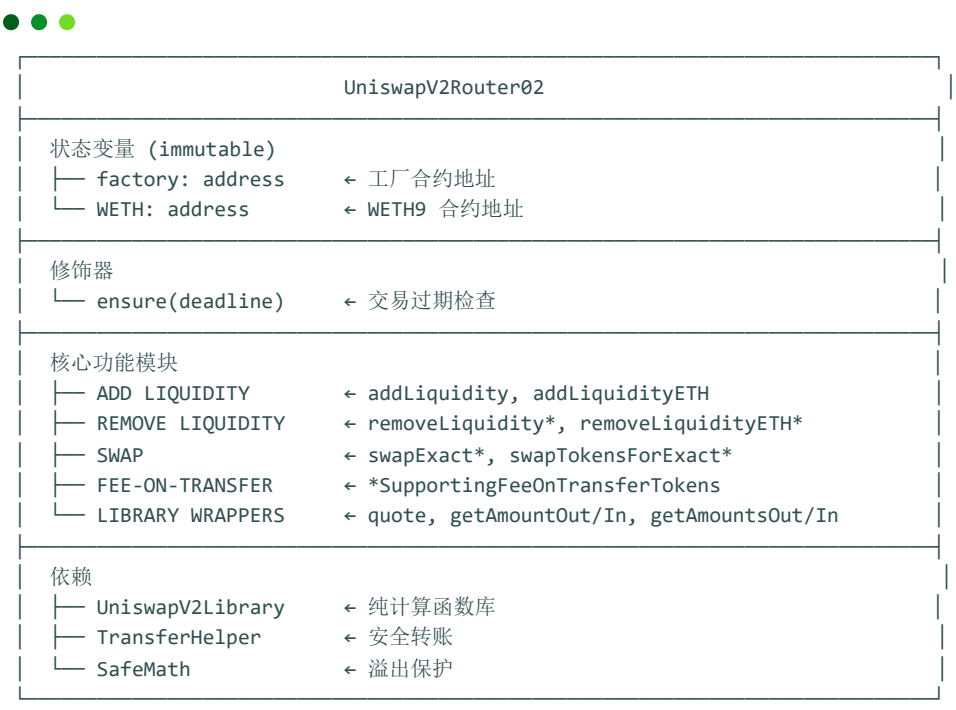
> 学习日期: 2024-12-14
> 源码路径: `src/v2-periphery/contracts/UniswapV2Router02.sol`
> 相关合约: UniswapV2Library, UniswapV2Pair, UniswapV2Factory

目录

---

1. 合约架构概览

1.1 整体结构图



1.2 函数分类表

类别	函数名	可见性	特点
添加流动性	`addLiquidity`	external	标准 ERC20 对
	`addLiquidityETH`	external payable	ETH + ERC20
移除流动性	`removeLiquidity`	public	基础版本
	`removeLiquidityETH`	public	返回 ETH
	`removeLiquidityWithPermit`	external	免 approve
	`removeLiquidityETHWithPermit`	external	免 approve + 返回 ETH
	`*SupportingFeeOnTransferTokens`	public/external	通缩币兼容
交换	`swapExactTokensForTokens`	external	固定输入
	`swapTokensForExactTokens`	external	固定输出
	`swapExactETHForTokens`	external payable	ETH → Token
	`swapTokensForExactETH`	external	Token → ETH
	`swapExactTokensForETH`	external	Token → ETH
	`swapETHForExactTokens`	external payable	ETH → Token
内部	`_addLiquidity`	internal	计算最优数量
	`_swap`	internal	多跳循环
	`_swapSupportingFeeOnTransferTokens`	internal	通缩币多跳

---

2. 核心状态变量与构造函数

2.1 源码分析

```
// @file: UniswapV2Router02.sol:15-26
address public immutable override factory;
address public immutable override WETH;

constructor(address _factory, address _WETH) public {
    factory = _factory;
    WETH = _WETH;
}
```

2.2 设计要点

为什么使用 `immutable`？

- **Gas 优化**：`immutable` 变量在部署时写入字节码，读取时无需 SLOAD（2100 gas），直接从代码读取（3 gas）
- **不可升级性**：一旦部署，factory 和 WETH 地址永远不变

- **安全性**: 防止恶意修改关键地址

🔴 疑问: 如果 factory 升级怎么办?

> Router 必须重新部署。这是有意为之的设计——Router 是无状态的，用户资金不存储在 Router 中，重新部署成本低。

2.3 receive() 函数

```
// @file: UniswapV2Router02.sol:28-30
receive() external payable {
    assert(msg.sender == WETH); // only accept ETH via fallback from the WETH contract
}
```

作用: 只接受来自 WETH 合约的 ETH（`WETH.withdraw()` 时触发）

🔴 疑问: 为什么用 assert 而不是 require?

> `assert` 用于检查不应该发生的情况（内部错误），失败会消耗所有 gas。这里用 assert 表示：如果有人直接向 Router 发送 ETH，这是一个严重的使用错误，应该被惩罚。

---

3. pairFor() 离线地址计算

3.1 源码分析

```
// @file: UniswapV2Library.sol:17-26
function pairFor(address factory, address tokenA, address tokenB)
    internal pure returns (address pair)
{
    (address token0, address token1) = sortTokens(tokenA, tokenB);
    pair = address(uint(keccak256(abi.encodePacked(
            hex'ff',
            factory,
            keccak256(abi.encodePacked(token0, token1)),
            hex'25aad938d8616b6e59148d3e701e4966de4418a752233589352d7c616a256568' // init code hash
        ))));
}
```

3.2 CREATE2 地址计算公式

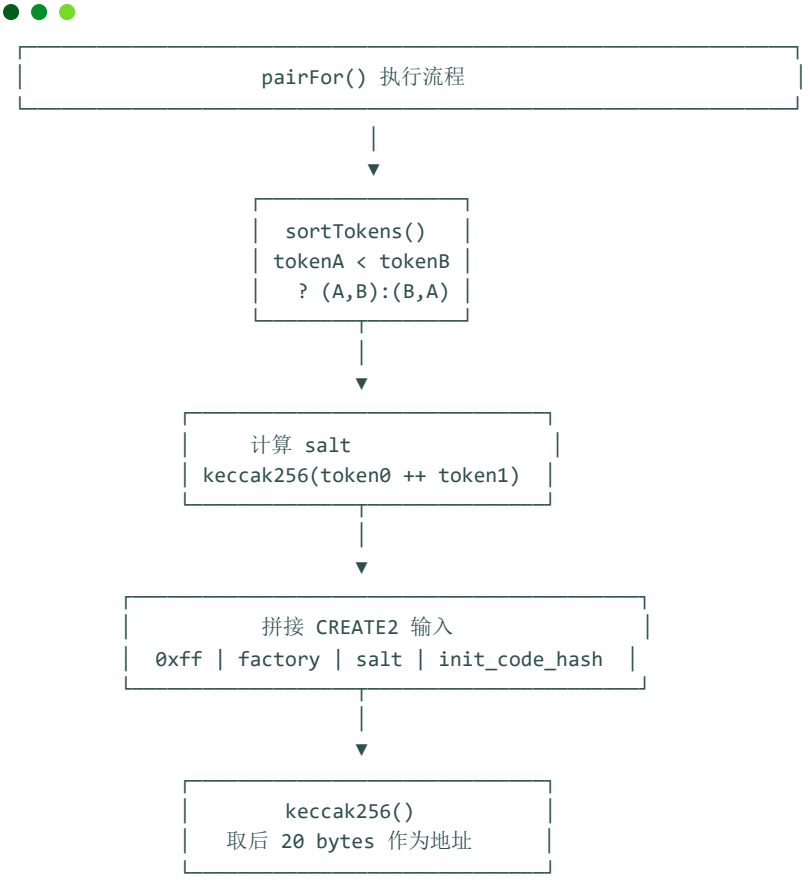
```
address = keccak256(0xff ++ factory_address ++ salt ++ init_code_hash)[12:]
```

组成部分:

字段	值	说明
`0xff`	固定前缀	CREATE2 标识符
`factory`	工厂地址	20 bytes
`salt`	`keccak256(token0, token1)`	32 bytes

字段	值	说明
`init_code_hash`	Pair 合约创建字节码的 hash	32 bytes

3.3 流程图



3.4 sortTokens() 地址升序规则

```solidity
// @file: UniswapV2Library.sol:11-15
function sortTokens(address tokenA, address tokenB)
    internal pure returns (address token0, address token1)
{
    require(tokenA != tokenB, 'UniswapV2Library: IDENTICAL_ADDRESSES');
    (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
    require(token0 != address(0), 'UniswapV2Library: ZERO_ADDRESS');
}
```

设计目的:

1. **确定性**: 无论传入顺序如何，始终返回相同的 (token0, token1)
2. **唯一性**: 每对代币只有一个 Pair 合约
3. **零地址检查**: 只检查 token0，因为如果 token0 != 0，token1 必然 != 0（token1 > token0）

🔴 **疑问: 为什么只检查 token0 != address(0)?**

> 因为 `token0 < token1`，如果 token0 不是零地址，token1 必然也不是。这是一个 gas 优化。

---

4. getAmountOut/getAmountIn 公式推导

4.1 常数积公式基础

Uniswap V2 的核心是 **常数积做市商 (CPMM)**:

```
x * y = k (恒定)
```

其中 x, y 是两种代币的储备量。

4.2 getAmountOut 源码与推导

```
// @file: UniswapV2Library.sol:43-50
function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut)
    internal pure returns (uint amountOut)
{
    require(amountIn > 0, 'UniswapV2Library: INSUFFICIENT_INPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
    uint amountInWithFee = amountIn.mul(997);           // 扣除 0.3% 手续费
    uint numerator = amountInWithFee.mul(reserveOut);   // 分子
    uint denominator = reserveIn.mul(1000).add(amountInWithFee); // 分母
    amountOut = numerator / denominator;
}
```

数学推导:

设:

- `Δx` = amountIn (输入量)
- `Δy` = amountOut (输出量)
- `x` = reserveIn
- `y` = reserveOut
- 手续费率 = 0.3% = 3/1000

Step 1: 扣除手续费后的有效输入

```
Δx_effective = Δx * (1 - 0.003) = Δx * 0.997 = Δx * 997/1000
```

Step 2: 应用常数积公式

```
x * y = (x + Δx_effective) * (y - Δy)
```

Step 3: 求解 Δy

```
x * y = (x + Δx * 997/1000) * (y - Δy)
x * y = x*y - x*Δy + Δx*997/1000*y - Δx*997/1000*Δy
0 = -x*Δy + Δx*997/1000*y - Δx*997/1000*Δy
x*Δy + Δx*997/1000*Δy = Δx*997/1000*y
Δy * (x + Δx*997/1000) = Δx*997/1000*y
Δy = (Δx * 997 * y) / (x * 1000 + Δx * 997)
```

代码对应:

```
amountOut = (amountIn * 997 * reserveOut) / (reserveIn * 1000 + amountIn * 997)
          = numerator / denominator
```

4.3 getAmountIn 源码与推导

```
// @file: UniswapV2Library.sol:53-59
function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut)
    internal pure returns (uint amountIn)
{
    require(amountOut > 0, 'UniswapV2Library: INSUFFICIENT_OUTPUT_AMOUNT');
    require(reserveIn > 0 && reserveOut > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
    uint numerator = reserveIn.mul(amountOut).mul(1000);
    uint denominator = reserveOut.sub(amountOut).mul(997);
    amountIn = (numerator / denominator).add(1);  // 向上取整
}
```

推导 (从 getAmountOut 公式反推):

```
Δy = (Δx * 997 * y) / (x * 1000 + Δx * 997)

解 Δx:
Δy * (x * 1000 + Δx * 997) = Δx * 997 * y
Δy * x * 1000 + Δy * Δx * 997 = Δx * 997 * y
Δy * x * 1000 = Δx * 997 * y - Δy * Δx * 997
Δy * x * 1000 = Δx * 997 * (y - Δy)
Δx = (Δy * x * 1000) / (997 * (y - Δy))
```

🔴 疑问: 为什么 getAmountIn 最后要 +1?

> 这是**向上取整**，确保用户支付足够的输入量。由于整数除法会截断，如果不 +1，可能导致实际输出略少于预期。

4.4 quote() 与 getAmountOut() 对比

```
// @file: UniswapV2Library.sol:36-40
function quote(uint amountA, uint reserveA, uint reserveB)
    internal pure returns (uint amountB)
{
    require(amountA > 0, 'UniswapV2Library: INSUFFICIENT_AMOUNT');
    require(reserveA > 0 && reserveB > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
    amountB = amountA.mul(reserveB) / reserveA;  // 无手续费!
}
```

函数	公式	手续费	用途
`quote()`	`amountB = amountA * reserveB / reserveA`	❌ 无	添加流动性时计算等价数量
`getAmountOut()`	见上文	✅ 0.3%	交换时计算输出

🔴 疑问: 为什么 addLiquidity 用 quote 而不是 getAmountOut?

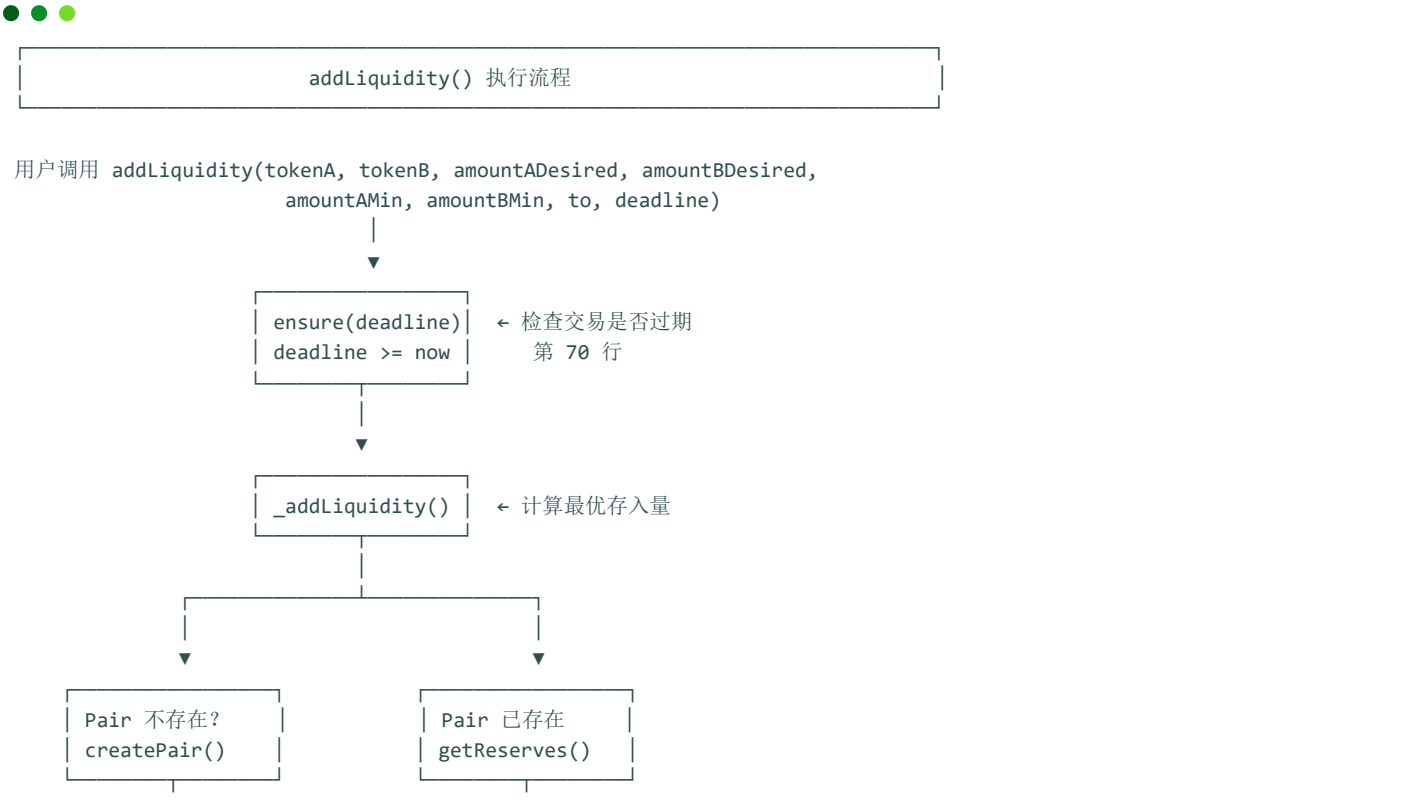
> 添加流动性不是交换，**不应该收取手续费**。LP 按当前价格比例存入两种代币。

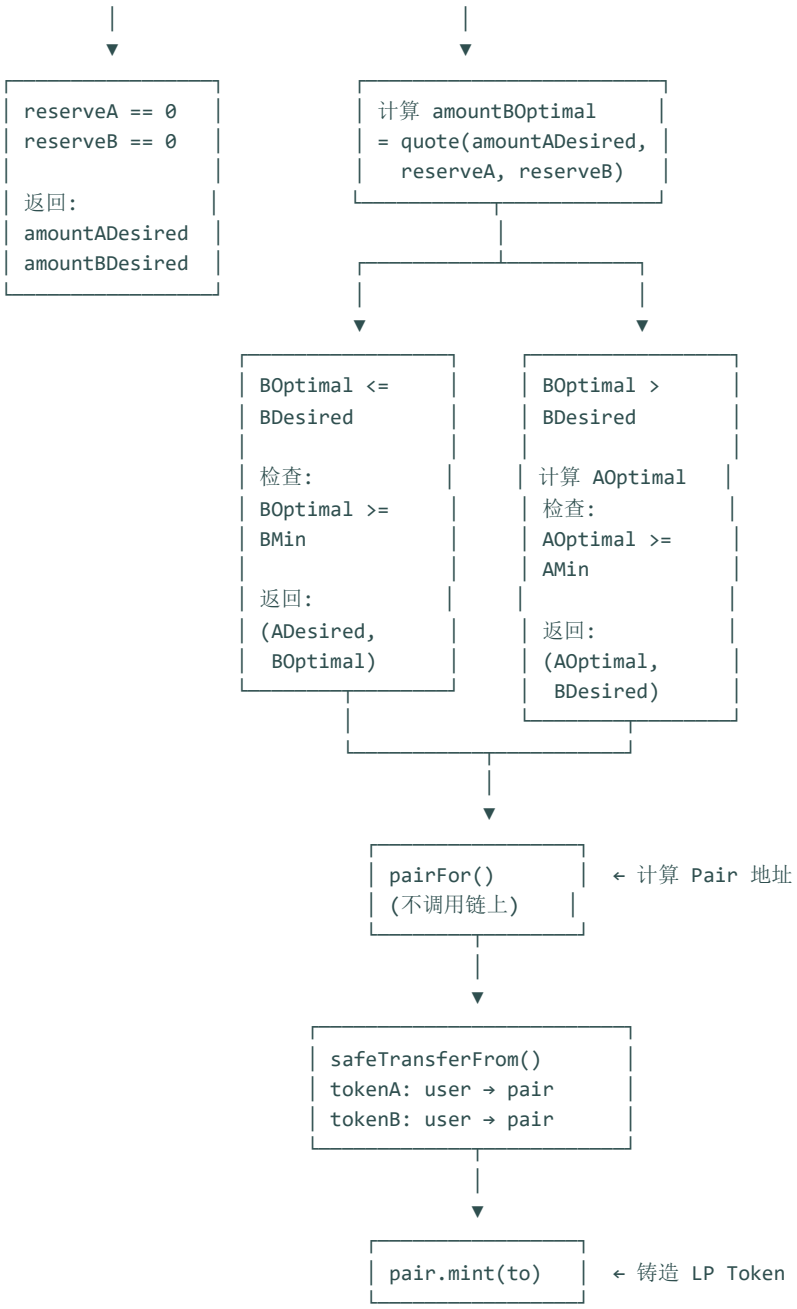
4.5 手续费位置分析

---

5. addLiquidity 流程与滑点保护

5.1 完整流程图





5.2 滑点保护机制

参数说明:

参数	含义	保护作用
`amountADesired`	期望存入的 A 数量	上限
`amountBDesired`	期望存入的 B 数量	上限
`amountAMin`	最少接受的 A 数量	下限，防止滑点过大
`amountBMin`	最少接受的 B 数量	下限，防止滑点过大
`deadline`	交易截止时间	防止交易被延迟执行

滑点检查位置:

```
// @file: UniswapV2Router02.sol:51
require(amountBOptimal >= amountBMin, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');

// @file: UniswapV2Router02.sol:56
require(amountAOptimal >= amountAMin, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
```

5.3 deadline 与 MEV

deadline 如何缓解 MEV?

- 限制交易有效期，矿工无法无限期延迟执行
- 用户可以设置较短的 deadline（如 20 分钟）

为何无法完全消除三明治攻击?

- 三明治攻击发生在**同一区块内**
- deadline 只能防止跨区块延迟，无法阻止同区块的抢跑

```
区块 N:
  ├── 攻击者 frontrun tx (买入，抬高价格)
  ├── 用户 tx (以更差价格买入)     ← deadline 检查通过
  └── 攻击者 backrun tx (卖出获利)
```

---

6. swapExactTokensForTokens 多跳实现

6.1 源码分析

```
// @file: UniswapV2Router02.sol:224-237
function swapExactTokensForTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint[] memory amounts) {
    // 1. 计算每一跳的输入输出量
    amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);

    // 2. 滑点检查：最终输出 >= 最小期望
    require(amounts[amounts.length - 1] >= amountOutMin,
            'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');

    // 3. 将初始代币转入第一个 Pair
    TransferHelper.safeTransferFrom(
        path[0], msg.sender,
        UniswapV2Library.pairFor(factory, path[0], path[1]),
        amounts[0]
    );

    // 4. 执行多跳交换
    _swap(amounts, path, to);
}
```

6.2 _swap 内部循环

```solidity
// @file: UniswapV2Router02.sol:212-223
function _swap(uint[] memory amounts, address[] memory path, address _to) internal virtual {
    for (uint i; i < path.length - 1; i++) {
        (address input, address output) = (path[i], path[i + 1]);
        (address token0,) = UniswapV2Library.sortTokens(input, output);
        uint amountOut = amounts[i + 1];

        // 确定 amount0Out 和 amount1Out
        (uint amount0Out, uint amount1Out) = input == token0
            ? (uint(0), amountOut)
            : (amountOut, uint(0));

        // 确定接收地址：中间跳 → 下一个 Pair，最后一跳 → 用户
        address to = i < path.length - 2
            ? UniswapV2Library.pairFor(factory, output, path[i + 2])
            : _to;

        // 调用 Pair.swap
        IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output))
            .swap(amount0Out, amount1Out, to, new bytes(0));
    }
}
```

6.3 多跳流程图 (3 跳示例: A → B → C → D)

6.4 getAmountsOut 多跳计算

```
// @file: UniswapV2Library.sol:62-70
function getAmountsOut(address factory, uint amountIn, address[] memory path)
    internal view returns (uint[] memory amounts)
{
    require(path.length >= 2, 'UniswapV2Library: INVALID_PATH');
    amounts = new uint[](path.length);
    amounts[0] = amountIn;
    for (uint i; i < path.length - 1; i++) {
        (uint reserveIn, uint reserveOut) = getReserves(factory, path[i], path[i + 1]);
        amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut);
    }
}
```

🔴 **疑问: 最大 7 跳栈深度限制?**

> EVM 栈深度限制为 1024，但实际限制来自 gas。每跳约消耗 ~60,000 gas，7 跳约 420,000 gas。超过 8 跳通常不经济。

---

7. ETH wrap/unwrap 流程

7.1 ETH → Token (swapExactETHForTokens)

```
// @file: UniswapV2Router02.sol:252-266
function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadline)
    external virtual override payable ensure(deadline)
    returns (uint[] memory amounts)
{
    require(path[0] == WETH, 'UniswapV2Router: INVALID_PATH');
    amounts = UniswapV2Library.getAmountsOut(factory, msg.value, path);
    require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');

    // 1. 将 ETH 存入 WETH 合约，获得 WETH
    IWETH(WETH).deposit{value: amounts[0]}();

    // 2. 将 WETH 转入第一个 Pair
    assert(IWETH(WETH).transfer(UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0]));

    // 3. 执行交换
    _swap(amounts, path, to);
}
```

7.2 Token → ETH (swapExactTokensForETH)

```
// @file: UniswapV2Router02.sol:284-300
function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline)
    external virtual override ensure(deadline)
    returns (uint[] memory amounts)
{
    require(path[path.length - 1] == WETH, 'UniswapV2Router: INVALID_PATH');
    amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
    require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');

    TransferHelper.safeTransferFrom(
```

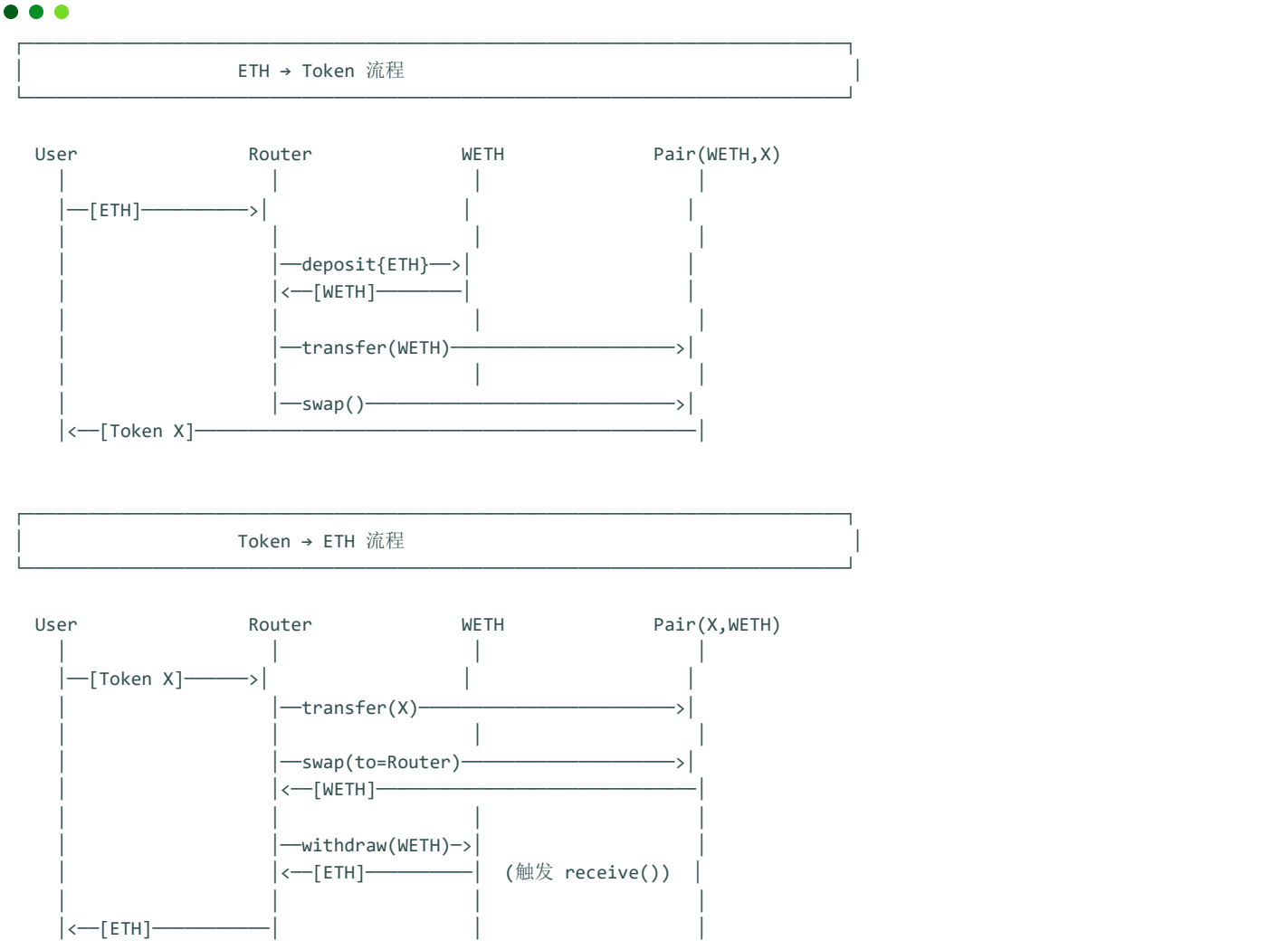
```
        path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0]
    );

    // 交换输出到 Router 自身
    _swap(amounts, path, address(this));

    // 将 WETH 转换为 ETH
    IWETH(WETH).withdraw(amounts[amounts.length - 1]);

    // 将 ETH 发送给用户
    TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);
}
```

7.3 ETH wrap/unwrap 流程图



7.4 receive() 与 WETH.withdraw() 的关系

```
// Router 的 receive()
receive() external payable {
    assert(msg.sender == WETH); // 只接受 WETH 合约的 ETH
}
```

当调用 `WETH.withdraw(amount)` 时：

1. WETH 合约销毁用户的 WETH

2. WETH 合约向调用者（Router）发送 ETH
3. Router 的 `receive()` 被触发
4. Router 再将 ETH 转给用户

---

8. removeLiquidityETHWithPermit 省 gas 技巧

8.1 源码分析

```
// @file: UniswapV2Router02.sol:156-169
function removeLiquidityETHWithPermit(
    address token,
    uint liquidity,
    uint amountTokenMin,
    uint amountETHMin,
    address to,
    uint deadline,
    bool approveMax, uint8 v, bytes32 r, bytes32 s
) external virtual override returns (uint amountToken, uint amountETH) {
    address pair = UniswapV2Library.pairFor(factory, token, WETH);
    uint value = approveMax ? uint(-1) : liquidity;

    // EIP-2612 permit: 用签名代替 approve 交易
    IUniswapV2Pair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);

    (amountToken, amountETH) = removeLiquidityETH(token, liquidity, amountTokenMin, amountETHMin, to, deadline);
}
```

8.2 EIP-2612 Permit 机制

传统流程 (2 笔交易):

```
Tx1: LP Token.approve(Router, amount)  ← 需要 gas
Tx2: Router.removeLiquidity(...)
```

Permit 流程 (1 笔交易):

```
Tx1: Router.removeLiquidityWithPermit(... v, r, s)
     ├── pair.permit(owner, spender, value, deadline, v, r, s)  ← 链上验证签名
     └── removeLiquidity(...)
```

8.3 Permit 签名验证 (UniswapV2ERC20)

```
// @file: UniswapV2ERC20.sol:81-93
function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external {
    require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
    bytes32 digest = keccak256(
        abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
        )
```

```
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'UniswapV2: INVALID_SIGNATURE');
        _approve(owner, spender, value);
    }
```

8.4 Gas 节省分析

操作	Gas 消耗
传统 approve 交易	~46,000 (21,000 base + SSTORE)
permit 验证	~3,000 (ecrecover + SSTORE)
节省	**~43,000 gas**

🔴 疑问: permit 失败时回滚到哪一层?

> 回滚到 `removeLiquidityETHWithPermit` 的调用层。整个交易失败，用户的 LP Token 不会被转移。

---

9. 通缩币支持机制

9.1 问题背景

通缩币（Fee-on-Transfer Token）在每次转账时会销毁或抽取一部分代币。例如：

- 转账 100 Token，实际到账 98 Token（2% 销毁）

9.2 标准 swap 的问题

```
// 标准 _swap 预计算 amounts
amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
// amounts[1] 基于 amounts[0] 计算，但通缩币实际到账可能更少
```

9.3 通缩币版本的解决方案

```
// @file: UniswapV2Router02.sol:321-338
function _swapSupportingFeeOnTransferTokens(address[] memory path, address _to) internal virtual {
    for (uint i; i < path.length - 1; i++) {
        (address input, address output) = (path[i], path[i + 1]);
        (address token0,) = UniswapV2Library.sortTokens(input, output);
        IUniswapV2Pair pair = IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output));
        uint amountInput;
        uint amountOutput;
        { // scope to avoid stack too deep errors
            (uint reserve0, uint reserve1,) = pair.getReserves();
            (uint reserveInput, uint reserveOutput) = input == token0 ? (reserve0, reserve1) : (reserve1, reserve0);

            // 关键：使用实际余额差计算输入量
            amountInput = IERC20(input).balanceOf(address(pair)).sub(reserveInput);
            amountOutput = UniswapV2Library.getAmountOut(amountInput, reserveInput, reserveOutput);
        }
        (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOutput) : (amountOutput, uint(0));
```

```
        address to = i < path.length - 2 ? UniswapV2Library.pairFor(factory, output, path[i + 2]) : _to;
        pair.swap(amount0Out, amount1Out, to, new bytes(0));
    }
}
```

9.4 关键差异对比

特性	标准 _swap	_swapSupportingFeeOnTransferTokens
输入量计算	预计算 amounts[]	实时读取 `balanceOf - reserve`
输出量计算	预计算 amounts[]	实时计算 getAmountOut
滑点检查	检查 amounts[last]	检查实际余额变化
Gas 消耗	较低	较高（多次 balanceOf 调用）

9.5 通缩币滑点检查

```
// @file: UniswapV2Router02.sol:349-354
uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
_swapSupportingFeeOnTransferTokens(path, to);
require(
    IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
    'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT'
);
```

🔴 疑问: addLiquidity 与 swapExact 哪个会多扣币?

> `addLiquidity` 会多扣币。因为 `_addLiquidity` 计算的是期望存入量，但通缩币实际到账更少，导致 LP 获得的流动性代币减少。Router 没有专门的 `addLiquiditySupportingFeeOnTransferTokens`。

---

10. 源码疑问与设计思考

10.1 为什么 Library 声明为 contract 而非 library?

```
// @file: UniswapV2Library.sol:7
library UniswapV2Library {  // 实际上声明为 library
```

注意: 源码中确实声明为 `library`，所有函数都是 `internal`。

internal 函数的优势:

- **内联**: 编译时直接嵌入调用合约，无需 DELEGATECALL
- **Gas 节省**: 避免外部调用开销（~700 gas）
- **无部署成本**: library 不需要单独部署

10.2 refundETH() 为何用 call 而非 transfer?

```
// TransferHelper.safeTransferETH 实现
function safeTransferETH(address to, uint value) internal {
    (bool success,) = to.call{value: value}("");
    require(success, 'TransferHelper: ETH_TRANSFER_FAILED');
}
```

原因:

1. `transfer` 固定 2300 gas，可能不够接收合约执行 fallback
2. `call` 转发所有剩余 gas，更灵活
3. EIP-1884 后 SLOAD 成本增加，2300 gas 可能不够

10.3 Router 零余额原则

Router 设计为**无状态**，不应持有任何代币或 ETH:

- 所有操作完成后，资产应全部转出
- `refundETH()` 和退还多余 ETH 的逻辑确保这一点

🔴 **疑问: 若 Router 被误转 WETH，如何提取?**

> 任何人都可以调用 Router 的公开函数，但 Router 没有 `unwrapWETH()` 这样的公开函数。误转的 WETH 可能永久锁定，除非有人通过 swap 路径间接取出。

10.4 tx.origin 使用位置

在 UniswapV2 整套合约中，`tx.origin` 使用非常谨慎:

位置 1: 不在 Router 或 Library 中使用

位置 2: 可能在某些辅助合约中用于防止合约调用

设计目的: 避免 tx.origin 的安全风险（钓鱼攻击）

---

11. 验证问题解答

问题 1: 手写离线计算 Pair 地址的完整 Solidity 代码

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract PairAddressCalculator {
    bytes32 public constant INIT_CODE_HASH =
        0x25aad938d8616b6e59148d3e701e4966de4418a752233589352d7c616a256568;

    function sortTokens(address tokenA, address tokenB)
        internal pure returns (address token0, address token1)
    {
        require(tokenA != tokenB, "IDENTICAL_ADDRESSES");
        (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), "ZERO_ADDRESS");
    }
```

```
    function pairFor(address factory, address tokenA, address tokenB)
        public pure returns (address pair)
    {
        (address token0, address token1) = sortTokens(tokenA, tokenB);
        pair = address(uint160(uint256(keccak256(abi.encodePacked(
            hex'ff',
            factory,
            keccak256(abi.encodePacked(token0, token1)),
            INIT_CODE_HASH
        )))));
    }
}
```

问题 2: factory 升级导致 init_code_hash 变化，旧 Router 还能算出正确地址吗?

答案: ❌ 不能

- `init_code_hash` 是 Pair 合约创建字节码的 hash
- 如果 Factory 升级，新 Pair 的字节码可能改变
- 旧 Router 硬编码了旧的 `init_code_hash`
- 计算出的地址将是错误的，交易会失败

问题 3: amountOutMin 在源码第几行被校验？回滚错误签名是什么?

答案:

- **行号**: 第 232 行 (`swapExactTokensForTokens`)
- **错误签名**: `'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT'`

```
// @file: UniswapV2Router02.sol:232
require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
```

问题 4: deadline 如何缓解 MEV，为何无法完全消除三明治?

缓解方式:

- 限制交易有效期，矿工无法无限延迟
- 用户可设置短 deadline（如 5-20 分钟）

无法消除三明治的原因:

- 三明治攻击在**同一区块内**完成
- deadline 只检查 `block.timestamp` ，同区块内所有交易 timestamp 相同
- 攻击者可以在 deadline 内完成 frontrun + backrun

问题 5: 通缩币场景下 addLiquidity 与 swapExact 哪个会多扣币?

答案: `addLiquidity` 会多扣币

原因:

- `addLiquidity` 没有通缩币版本
- 计算的 `amountA/amountB` 是期望值，实际到账更少

- LP 获得的流动性代币按实际到账计算，会减少

swapExact 有 `SupportingFeeOnTransferTokens` 版本，使用实际余额差计算。

问题 6: 三跳交换比单跳多消耗多少 gas?

估算:

- 单跳 swap: ~100,000 gas
- 三跳 swap: ~220,000 gas
- 差值: ~120,000 gas

美元成本 (12 gwei, ETH $3800):

```
120,000 * 12 * 10^-9 * 3800 = $5.47
```

问题 7: removeLiquidityETHWithPermit 的 permit 失败时回滚到哪一层?

答案: 回滚到 `removeLiquidityETHWithPermit` 的外部调用层

- `permit` 内部 `require` 失败会 revert
- 整个交易回滚，状态不变
- 用户 LP Token 保持不变

调试方法:

1. 使用 Tenderly 或 Foundry 的 trace
2. 检查 `ecrecover` 返回的地址
3. 验证 nonce 是否正确

问题 8: refundETH() 为何用 call{value:amount}("") 而非 transfer?

答案:

1. `transfer` 固定 2300 gas，EIP-1884 后可能不够
2. `call` 转发所有剩余 gas
3. 接收合约可能需要执行复杂的 fallback 逻辑
4. 更好的兼容性

问题 9: 若 Router 被误转 WETH，用户如何无损提取?

答案: 很难无损提取

- Router 没有公开的 `unwrapWETH()` 函数
- 可能的方法:
    1. 通过包含 WETH 的 swap 路径间接取出
    2. 如果有治理机制，可能需要升级合约
- **最佳实践**: 不要向 Router 直接转账

问题 10: UniswapV2 整套合约里哪两处用到 tx.origin?

答案: UniswapV2 核心合约（Router, Library, Pair, Factory）**不使用 tx.origin**

这是有意的安全设计，避免：

- 钓鱼攻击
- 合约调用限制问题

问题 11: 数学证明 getAmountOut 公式满足常数积 k 不变

证明:

设初始状态: `k = x * y`

交换后:

- 新 reserveIn: `x' = x + Δx * 0.997`
- 新 reserveOut: `y' = y - Δy`

由公式:

```
Δy = (Δx * 997 * y) / (x * 1000 + Δx * 997)
```

验证 `x' * y' >= k`:

```
x' * y' = (x + Δx * 0.997) * (y - Δy)
        = (x + Δx * 0.997) * (y - (Δx * 997 * y) / (x * 1000 + Δx * 997))
        = (x + Δx * 0.997) * y * (1 - (Δx * 997) / (x * 1000 + Δx * 997))
        = (x + Δx * 0.997) * y * (x * 1000) / (x * 1000 + Δx * 997)
        = y * (x * 1000 + Δx * 997) * (x * 1000) / (x * 1000 + Δx * 997) / 1000
        = x * y
        = k ✓
```

问题 12: 分母溢出时 Solidity 0.8 行为

答案: Solidity 0.8+ 会自动 revert

- 内置溢出检查
- 无需 SafeMath
- 交易失败，状态回滚

注意: UniswapV2 使用 Solidity 0.6.6，依赖 SafeMath 进行溢出检查。

问题 13: Library 为何不声明为 library 而是 contract?

更正: UniswapV2Library **确实声明为 library**

```
library UniswapV2Library {
```

所有函数都是 `internal`，编译时内联到调用合约。

问题 14: 前端倒序 path 数组但 amountIn 不变，Router 会回滚吗?

答案: 会回滚

回滚点: 取决于具体情况

- 如果 path[0] 与用户持有的代币不匹配，`safeTransferFrom` 失败
- 如果 Pair 不存在，`getReserves` 调用失败
- 如果流动性不足，`getAmountOut` 返回 0，后续 swap 失败

行号: 第 233-235 行 (safeTransferFrom)

问题 15: 若给 Router 加"单区块最大价格滑点"保护，如何实现?

实现思路:

```
// 新增状态变量
mapping(address => mapping(address => uint256)) public lastBlockPrice;
mapping(address => mapping(address => uint256)) public lastBlockNumber;
uint256 public maxSlippagePerBlock = 500; // 5%

// 修改 _swap 函数
function _swap(...) internal virtual {
    for (uint i; i < path.length - 1; i++) {
        // ... 现有代码 ...

        // 新增: 价格滑点检查
        (uint reserveIn, uint reserveOut) = UniswapV2Library.getReserves(factory, input, output);
        uint256 currentPrice = reserveOut * 1e18 / reserveIn;

        if (lastBlockNumber[input][output] == block.number) {
            uint256 priceChange = currentPrice > lastBlockPrice[input][output]
                ? (currentPrice - lastBlockPrice[input][output]) * 10000 / lastBlockPrice[input][output]
                : (lastBlockPrice[input][output] - currentPrice) * 10000 / lastBlockPrice[input][output];
            require(priceChange <= maxSlippagePerBlock, "EXCESSIVE_BLOCK_SLIPPAGE");
        }

        lastBlockPrice[input][output] = currentPrice;
        lastBlockNumber[input][output] = block.number;

        // ... 继续现有 swap 逻辑 ...
    }
}
```

Gas 代价:

- 每跳增加 ~5,000 gas (SLOAD + SSTORE)
- 3 跳交换增加 ~15,000 gas

局限性:

- 仍无法完全防止三明治（攻击者可以分多区块操作）
- 增加了正常用户的 gas 成本

附录: 关键源码行号索引

功能	文件	行号
factory/WETH 声明	Router02.sol	15-16
ensure modifier	Router02.sol	18-21
receive()	Router02.sol	28-30
_addLiquidity	Router02.sol	33-60
addLiquidity	Router02.sol	61-76
addLiquidityETH	Router02.sol	77-100
removeLiquidity	Router02.sol	103-119
removeLiquidityETHWithPermit	Router02.sol	156-169
_swap	Router02.sol	212-223
swapExactTokensForTokens	Router02.sol	224-237
_swapSupportingFeeOnTransferTokens	Router02.sol	321-338
sortTokens	Library.sol	11-15
pairFor	Library.sol	18-26
getReserves	Library.sol	29-33
quote	Library.sol	36-40
getAmountOut	Library.sol	43-50
getAmountIn	Library.sol	53-59
getAmountsOut	Library.sol	62-70
Pair.swap	Pair.sol	159-187
Pair.mint	Pair.sol	110-131
permit	ERC20.sol	81-93