Solidity智能合约开发知识

第8.1课：合约间调用

学习目标：掌握Solidity中合约间调用的各种方式、理解接口调用和底层调用的区别、学会安全地进行外部调用、能够在实际项目中正确应用合约间调用技术

预计学习时间：2-3小时

难度等级：中级

目录

---

1. 合约间调用基础概念

1.1 为什么需要合约间调用

在实际的区块链开发中，很少有孤立存在的合约。更常见的情况是，多个合约之间需要相互协作、调用，共同完成复杂的业务逻辑。

合约间调用的必要性：

1. **模块化设计**：
   - 将复杂系统拆分为多个专门的合约
   - 每个合约负责特定功能
   - 提高代码可维护性和可重用性
2. **功能复用**：
   - 使用标准接口（如ERC20、ERC721）实现互操作性
   - 避免重复实现相同功能
   - 利用经过审计的成熟合约
3. **业务复杂性**：
   - DeFi协议需要调用多个外部合约
   - 借贷协议需要价格预言机、代币合约、流动性池
   - 复杂的业务逻辑需要多个合约协作
4. **升级和维护**：
   - 通过代理模式实现合约升级

- 分离数据和逻辑，便于维护
- 支持渐进式功能迭代

实际应用场景：

```
// 场景1：DeFi借贷协议
contract LendingProtocol {
    IERC20 public token;            // 调用代币合约
    IPriceOracle public oracle;     // 调用价格预言机
    ILiquidityPool public pool;     // 调用流动性池

    function borrow(uint256 amount) public {
        // 1. 从价格预言机获取价格
        uint256 price = oracle.getPrice(address(token));

        // 2. 从代币合约转移资金
        token.transferFrom(msg.sender, address(this), amount);

        // 3. 与流动性池交互
        pool.deposit(amount);
    }
}
```

1.2 合约间调用的方式

Solidity提供了多种合约间调用的方式，每种方式都有其特点和适用场景：

1. 接口调用（Interface）：

- 最安全、最规范的方式
- 编译时类型检查
- 代码可读性好
- 适合调用已知接口的合约

2. 底层调用方法：

- `call`：最通用的调用方式，可以发送以太币
- `delegatecall`：在调用者上下文中执行，用于代理模式
- `staticcall`：只读调用，不能修改状态

3. 合约创建：

- `new`关键字：传统创建方式
- `create2`：可预先计算地址的创建方式

1.3 调用上下文的重要性

理解调用上下文是掌握合约间调用的关键。不同的调用方式会在不同的上下文中执行，这直接影响：

- 状态变量的修改位置
- `msg.sender`的值
- 合约余额的归属

- Gas消耗

---

2. 接口调用（Interface）

接口调用是合约间交互最安全、最规范的方式。它提供了类型安全、代码可读性好、Gas效率高等优势。

2.1 什么是接口

在Solidity中，接口（Interface）是一种定义合约必须实现的函数签名的方式。接口本身不包含任何实现代码，只声明函数的名称、参数和返回值。

接口的三个重要特征：

1. **定义函数签名**：接口规定了合约应该提供哪些功能，就像一份合同
2. **不包含实现**：接口只告诉你"有什么"，不告诉你"怎么做"
3. **只声明函数**：接口不包含状态变量，保持极简，专注于函数定义

接口与合约的区别：

```
// 接口：只有函数签名，没有实现
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// 合约：有完整的实现
contract ERC20Token is IERC20 {
    mapping(address => uint256) public balanceOf;

    // 必须实现接口中声明的所有函数
    function transfer(address to, uint256 amount) external returns (bool) {
        // 具体实现...
        return true;
    }

    function balanceOf(address account) external view returns (uint256) {
        // 具体实现...
        return balanceOf[account];
    }
}
```

2.2 接口定义语法

基本语法：

```
interface InterfaceName {
    function functionName(param1, param2) external returns (returnType);
    // 更多函数声明...
}
```

重要规则：

1. **函数必须标记为external**：接口中的函数不能是public、internal或private
2. **没有函数体**：接口中的函数只有签名，没有实现代码
3. **可以继承**：接口可以继承其他接口
4. **不能有状态变量**：接口中不能声明状态变量
5. **不能有构造函数**：接口不能有构造函数

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// ERC20代币接口定义
interface IERC20 {
    // 转账函数：从调用者地址向指定地址转账
    function transfer(address to, uint256 amount) external returns (bool);

    // 授权转账函数：从指定地址向另一个地址转账（需要授权）
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);

    // 查询余额函数：查询指定地址的代币余额
    function balanceOf(address account) external view returns (uint256);

    // 授权函数：授权指定地址使用调用者的代币
    function approve(address spender, uint256 amount) external returns (bool);

    // 查询授权额度：查询owner授权给spender的额度
    function allowance(address owner, address spender)
        external
        view
        returns (uint256);

    // 查询总供应量
    function totalSupply() external view returns (uint256);
}
```

在上面的接口定义中：

- 所有函数都标记为`external`，这是接口的要求
- `view`函数（如`balanceOf`）仍然需要`external`关键字
- 函数只有签名，没有实现代码
- 返回值类型明确声明，便于调用者处理

2.3 接口使用示例

定义了接口后，我们可以在其他合约中使用它来调用外部合约。

基础使用示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 定义ERC20接口
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);
}

// 代币交换合约：使用接口调用外部代币合约
contract TokenSwap {
    // 声明两个代币接口变量
    IERC20 public tokenA;
    IERC20 public tokenB;

    // 定义交换事件，记录每次交换的详细信息
    event Swap(
        address indexed user,
        uint256 amountA,
        uint256 amountB
    );

    // 构造函数：初始化两个代币合约地址
    constructor(address _tokenA, address _tokenB) {
        // 将地址转换为接口类型，这样就可以调用接口中定义的方法
        tokenA = IERC20(_tokenA);
        tokenB = IERC20(_tokenB);
    }

    /**
     * @notice 执行代币交换
     * @param amountA 要交换的tokenA数量
     * @dev 使用接口调用确保类型安全
     */
    function swap(uint256 amountA) external {
        // 步骤1：从用户账户转移tokenA到本合约
        // transferFrom需要用户先调用approve授权本合约使用其代币
        // 如果转账失败，require会回滚整个交易
        require(
            tokenA.transferFrom(msg.sender, address(this), amountA),
            "TokenA transfer failed"
        );

        // 步骤2：计算可交换的tokenB数量（简化示例，1:1兑换）
```

```
        uint256 amountB = amountA;

        // 步骤3：从本合约向用户转移tokenB
        // 如果转账失败，require会回滚整个交易
        require(
            tokenB.transfer(msg.sender, amountB),
            "TokenB transfer failed"
        );

        // 步骤4：触发事件，记录交换信息
        emit Swap(msg.sender, amountA, amountB);
    }

    /**
     * @notice 查询合约持有的代币余额
     * @return balanceA 合约持有的tokenA数量
     * @return balanceB 合约持有的tokenB数量
     */
    function getBalances() external view returns (uint256 balanceA, uint256 balanceB) {
        // 使用接口的view函数查询余额，不消耗Gas
        balanceA = tokenA.balanceOf(address(this));
        balanceB = tokenB.balanceOf(address(this));
    }
}
```

在上面的代码中：

- `TokenSwap`合约通过接口调用外部代币合约
- 使用`IERC20(_tokenA)`将地址转换为接口类型
- 调用`transferFrom`和`transfer`时，编译器会检查参数类型
- 如果传入错误的参数类型，编译时就会报错

2.4 接口调用的优势

使用接口调用有四个明显的优势：

1. 类型安全（Type Safety）：

编译时检查可以减少很多错误。如果你调用的函数不存在，或者参数类型不对，编译器会直接报错，而不是等到运行时才发现问题。

```
contract TypeSafetyExample {
    IERC20 public token;

    function transferTokens(address to, uint256 amount) public {
        // 正确：编译器会检查参数类型
        token.transfer(to, amount);

        // 编译错误：参数类型不匹配
        // token.transfer(to, "100");  // 字符串不能传给uint256参数

        // 编译错误：函数不存在
```

```
        // token.nonExistentFunction();  // 接口中没有这个函数
    }
}
```

2. 代码可读性好（Readability）：

当你看到`IERC20`类型的变量，立刻就知道这是一个符合ERC20标准的代币合约，它支持哪些操作一目了然。

```
contract ReadabilityExample {
    // 看到IERC20类型，就知道这个变量代表一个ERC20代币
    IERC20 public token;

    // 看到IPriceOracle类型，就知道这个变量代表一个价格预言机
    IPriceOracle public oracle;

    // 代码意图非常清晰，不需要查看具体实现就知道能做什么
    function getTokenPrice() public view returns (uint256) {
        return oracle.getPrice(address(token));
    }
}

interface IPriceOracle {
    function getPrice(address token) external view returns (uint256);
}
```

3. Gas效率高（Gas Efficiency）：

接口调用生成的字节码体积小，执行效率高。相比直接使用低级call调用，接口调用的Gas消耗更少。

```
contract GasEfficiencyComparison {
    IERC20 public token;
    address public tokenAddress;

    // 使用接口调用：Gas消耗较低
    function transferWithInterface(address to, uint256 amount) public {
        token.transfer(to, amount);
    }

    // 使用call调用：Gas消耗较高
    function transferWithCall(address to, uint256 amount) public {
        (bool success, ) = tokenAddress.call(
            abi.encodeWithSignature("transfer(address,uint256)", to, amount)
        );
        require(success, "Transfer failed");
    }
}
```

4. 易于测试（Testability）：

在单元测试中，我们可以使用mock合约来模拟接口，而不需要部署真实的合约，这大大简化了测试流程。

```
// 测试用的Mock代币合约
contract MockERC20 is IERC20 {
    mapping(address => uint256) public balanceOf;

    // 实现接口中的所有函数，但逻辑可以简化
    function transfer(address to, uint256 amount) external returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function balanceOf(address account) external view returns (uint256) {
        return balanceOf[account];
    }

    // 其他接口函数的简化实现...
}

// 在测试中使用Mock合约
contract TokenSwapTest {
    function testSwap() public {
        // 部署Mock合约而不是真实合约
        MockERC20 mockTokenA = new MockERC20();
        MockERC20 mockTokenB = new MockERC20();

        // 使用Mock合约测试TokenSwap
        TokenSwap swap = new TokenSwap(
            address(mockTokenA),
            address(mockTokenB)
        );

        // 执行测试...
    }
}
```

2.5 接口继承

接口可以继承其他接口，这样可以组合多个接口的功能。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 基础代币接口
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// 扩展接口：增加了授权功能
interface IERC20Extended is IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
```

```
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);
}

// 使用扩展接口
contract AdvancedTokenSwap {
    // 使用扩展接口，可以调用更多方法
    IERC20Extended public token;

    function swapWithApproval(
        address to,
        uint256 amount
    ) external {
        // 可以使用扩展接口中的方法
        require(
            token.transferFrom(msg.sender, address(this), amount),
            "TransferFrom failed"
        );
        require(
            token.transfer(to, amount),
            "Transfer failed"
        );
    }
}
```

2.6 接口调用的完整示例

以下是一个完整的代币交换合约示例，展示了接口调用的实际应用：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 定义ERC20接口
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
}

// 简单的ERC20代币实现（用于演示）
contract SimpleToken is IERC20 {
    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;
```

```
    string public name;
    string public symbol;
    uint8 public decimals = 18;
    uint256 public totalSupply;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor(string memory _name, string memory _symbol, uint256 _initialSupply) {
        name = _name;
        symbol = _symbol;
        totalSupply = _initialSupply * 10**decimals;
        _balances[msg.sender] = totalSupply;
        emit Transfer(address(0), msg.sender, totalSupply);
    }

    function transfer(address to, uint256 amount) external returns (bool) {
        require(_balances[msg.sender] >= amount, "Insufficient balance");
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        emit Transfer(msg.sender, to, amount);
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool) {
        require(_allowances[from][msg.sender] >= amount, "Insufficient allowance");
        require(_balances[from] >= amount, "Insufficient balance");

        _allowances[from][msg.sender] -= amount;
        _balances[from] -= amount;
        _balances[to] += amount;
        emit Transfer(from, to, amount);
        return true;
    }

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }

    function approve(address spender, uint256 amount) external returns (bool) {
        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }
}

// 代币交换合约（使用接口调用）
```

```
contract TokenSwap {
    // 声明两个代币接口变量
    IERC20 public tokenA;
    IERC20 public tokenB;

    // 交换事件：记录每次交换的详细信息
    event Swap(
        address indexed user,
        address indexed tokenIn,
        address indexed tokenOut,
        uint256 amountIn,
        uint256 amountOut
    );

    // 构造函数：初始化代币合约地址
    constructor(address _tokenA, address _tokenB) {
        // 将地址转换为接口类型
        // 这样编译器会检查这些地址对应的合约是否实现了接口中的函数
        tokenA = IERC20(_tokenA);
        tokenB = IERC20(_tokenB);
    }

    /**
     * @notice 执行代币交换
     * @param amountA 要交换的tokenA数量
     * @dev 用户需要先调用tokenA的approve函数授权本合约使用其代币
     */
    function swap(uint256 amountA) external {
        // 步骤1：检查合约是否有足够的tokenB用于交换
        // 使用接口的view函数查询余额，不消耗Gas
        uint256 contractBalanceB = tokenB.balanceOf(address(this));
        require(contractBalanceB >= amountA, "Insufficient tokenB in contract");

        // 步骤2：从用户账户转移tokenA到本合约
        // transferFrom需要用户先调用tokenA.approve授权本合约
        // 如果转账失败，require会回滚整个交易
        require(
            tokenA.transferFrom(msg.sender, address(this), amountA),
            "TokenA transfer failed"
        );

        // 步骤3：从本合约向用户转移tokenB
        // 简化示例：1:1兑换比例
        uint256 amountB = amountA;
        require(
            tokenB.transfer(msg.sender, amountB),
            "TokenB transfer failed"
        );

        // 步骤4：触发事件，记录交换信息
        // 前端应用可以监听这个事件来更新UI
        emit Swap(msg.sender, address(tokenA), address(tokenB), amountA, amountB);
```

```
    }

    /**
     * @notice 查询合约持有的代币余额
     * @return balanceA 合约持有的tokenA数量
     * @return balanceB 合约持有的tokenB数量
     */
    function getContractBalances()
        external
        view
        returns (uint256 balanceA, uint256 balanceB)
    {
        // 使用接口的view函数查询余额
        // view函数不修改状态，外部调用不消耗Gas
        balanceA = tokenA.balanceOf(address(this));
        balanceB = tokenB.balanceOf(address(this));
    }

    /**
     * @notice 查询用户持有的代币余额
     * @param user 要查询的用户地址
     * @return balanceA 用户的tokenA余额
     * @return balanceB 用户的tokenB余额
     */
    function getUserBalances(address user)
        external
        view
        returns (uint256 balanceA, uint256 balanceB)
    {
        balanceA = tokenA.balanceOf(user);
        balanceB = tokenB.balanceOf(user);
    }
}
```

使用流程：

1. **部署代币合约**：

```
SimpleToken tokenA = new SimpleToken("TokenA", "TKA", 1000000);
SimpleToken tokenB = new SimpleToken("TokenB", "TKB", 1000000);
```

2. **部署交换合约**：

```
TokenSwap swap = new TokenSwap(address(tokenA), address(tokenB));
```

3. **准备交换池**：

```
// 向交换合约转入tokenB作为交换池
tokenB.transfer(address(swap), 100000);
```

4. **用户授权**：

```
// 用户授权交换合约使用其tokenA
tokenA.approve(address(swap), 1000);
```

5. **执行交换**：

```
// 用户执行交换
swap.swap(1000);
```

---

3. 底层调用方法

除了接口调用，Solidity还提供了三种底层调用方法：call、delegatecall和staticcall。这三个方法功能强大但也更危险，需要谨慎使用。

3.1 call方法

call是最通用的底层调用方法，它可以调用任意合约的任意函数，甚至可以发送以太币。

call方法的特点：

1. **可以发送以太币**：这是call相比其他方法的独特优势
2. **在被调用合约的上下文中执行**：被调用的合约会使用它自己的storage、自己的余额
3. **最通用的调用方式**：当你不确定用哪种方法时，call通常是安全的选择

基本语法：

```
(bool success, bytes memory data) = address.call{value: amount}(
    abi.encodeWithSignature("functionName(type1,type2)", arg1, arg2)
);
```

参数说明：

- `address`：要调用的合约地址
- `{value: amount}`：可选，要发送的以太币数量（wei）
- `abi.encodeWithSignature(...)`：编码函数调用数据
- 返回值：`success`表示调用是否成功，`data`是返回的数据

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 被调用的目标合约
contract TargetContract {
    uint256 public value;
    address public sender;

    // 接收以太币的函数
```

```
    function setValue(uint256 _value) external payable {
        value = _value;
        sender = msg.sender;
    }

    // 普通函数
    function getValue() external view returns (uint256) {
        return value;
    }
}

// 调用者合约：使用call方法
contract CallerContract {
    // 使用call调用目标合约的函数
    function callSetValue(address target, uint256 newValue) external payable {
        // 使用call调用setValue函数，并发送以太币
        // abi.encodeWithSignature编码函数签名和参数
        (bool success, bytes memory data) = target.call{value: msg.value}(
            abi.encodeWithSignature("setValue(uint256)", newValue)
        );

        // 必须检查返回值，call失败不会自动revert
        require(success, "Call failed");
    }

    // 使用call调用view函数
    function callGetValue(address target) external view returns (uint256) {
        // 调用view函数，不发送以太币
        (bool success, bytes memory returnData) = target.call(
            abi.encodeWithSignature("getValue()")
        );

        require(success, "Call failed");

        // 解码返回值
        // abi.decode用于解码ABI编码的数据
        uint256 value = abi.decode(returnData, (uint256));
        return value;
    }

    // 使用call发送以太币（不调用函数）
    function sendEther(address payable recipient) external payable {
        // 直接向地址发送以太币，不调用任何函数
        (bool success, ) = recipient.call{value: msg.value}("");
        require(success, "Ether transfer failed");
    }
}
```

在上面的代码中：

- `callSetValue`使用call调用目标合约的函数并发送以太币
- `callGetValue`使用call调用view函数并解码返回值

- `sendEther`使用call直接发送以太币

call方法的执行上下文：

```
contract ContextExample {
    uint256 public callerValue;
    address public callerSender;

    function testCall(address target) external {
        // 调用目标合约的setValue函数
        (bool success, ) = target.call(
            abi.encodeWithSignature("setValue(uint256)", 42)
        );
        require(success, "Call failed");

        // 注意：callerValue和callerSender不会被修改
        // 因为setValue是在target合约的上下文中执行的
        // 它修改的是target合约的storage，不是本合约的
    }
}
```

3.2 delegatecall方法

delegatecall是一个特殊的方法，它的关键特点是**在调用者的上下文中执行**。

delegatecall的核心特点：

1. **在调用者的上下文中执行**：
   - 被调用的函数会修改调用者合约的storage
   - 被调用的函数使用的是调用者合约的余额
   - 但代码逻辑来自被调用的合约
2. **msg.sender保持不变**：
   - 在delegatecall中，msg.sender仍然是原始调用者
   - 这对于权限控制非常重要
3. **不能发送以太币**：
   - delegatecall不支持value参数
   - 不能通过delegatecall发送以太币

基本语法：

```
(bool success, bytes memory data) = address.delegatecall(
    abi.encodeWithSignature("functionName(type1,type2)", arg1, arg2)
);
```

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;
```

```
// 逻辑合约：包含业务逻辑
contract LogicContract {
    // 注意：这些变量的存储位置必须与代理合约匹配
    uint256 public value;
    address public owner;

    // 设置值的函数
    function setValue(uint256 _value) external {
        // 这个函数会修改调用者合约的storage
        value = _value;
        owner = msg.sender;  // msg.sender是原始调用者，不是代理合约
    }

    // 获取值的函数
    function getValue() external view returns (uint256) {
        return value;
    }
}

// 代理合约：存储数据，通过delegatecall调用逻辑合约
contract ProxyContract {
    // 存储布局必须与LogicContract完全一致
    address public implementation;  // 逻辑合约地址
    uint256 public value;           // 与LogicContract的value对应
    address public owner;            // 与LogicContract的owner对应

    event Upgraded(address indexed newImplementation);

    constructor(address _implementation) {
        implementation = _implementation;
        owner = msg.sender;
    }

    // fallback函数：将所有调用转发到逻辑合约
    fallback() external payable {
        address impl = implementation;
        require(impl != address(0), "Implementation not set");

        // 使用delegatecall调用逻辑合约
        // 逻辑合约的代码会在本合约的上下文中执行
        (bool success, bytes memory returnData) = impl.delegatecall(msg.data);

        if (!success) {
            // 如果调用失败，回滚
            assembly {
                returndatacopy(0, 0, returndatasize())
                revert(0, returndatasize())
            }
        }

        // 返回数据
        assembly {
```

```
            return(add(returnData, 0x20), mload(returnData))
        }
    }

    // 升级函数：更换逻辑合约
    function upgrade(address newImplementation) external {
        require(msg.sender == owner, "Not owner");
        implementation = newImplementation;
        emit Upgraded(newImplementation);
    }
}
```

delegatecall的执行流程：

```
用户调用 ProxyContract.setValue(100)
    ↓
ProxyContract的fallback函数被触发
    ↓
delegatecall到 LogicContract.setValue(100)
    ↓
LogicContract的代码在ProxyContract的上下文中执行
    ↓
修改的是ProxyContract的value和owner（不是LogicContract的）
    ↓
msg.sender仍然是原始用户（不是ProxyContract）
```

关键理解：

```
contract DelegatecallDemo {
    uint256 public value;
    address public sender;

    function testDelegatecall(address target) external {
        // 调用目标合约的setValue函数
        (bool success, ) = target.delegatecall(
            abi.encodeWithSignature("setValue(uint256)", 88)
        );
        require(success, "Delegatecall failed");

        // 注意：value和sender会被修改！
        // 因为delegatecall在调用者的上下文中执行
        // 目标合约的代码修改的是本合约的storage
    }
}
```

3.3 staticcall方法

staticcall是最安全但功能最受限的调用方法。它的核心特点是**保证不修改状态**。

staticcall的核心特点：

1. **保证不修改状态**：
    - 如果被调用的函数尝试修改状态，调用会直接失败
    - 只能调用view和pure函数
2. **只读查询**：
    - 非常适合调用view和pure函数
    - 提供额外的安全保障
3. **不能发送以太币**：
    - staticcall不支持value参数

基本语法：

```
(bool success, bytes memory data) = address.staticcall(
    abi.encodeWithSignature("functionName()")
);
```

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 目标合约
contract TargetContract {
    uint256 public value = 100;

    // view函数：可以读取状态
    function getValue() external view returns (uint256) {
        return value;
    }

    // 修改状态的函数
    function setValue(uint256 _value) external {
        value = _value;
    }
}

// 调用者合约：使用staticcall
contract StaticcallDemo {
    // 使用staticcall调用view函数（安全）
    function safeGetValue(address target) external view returns (uint256) {
        (bool success, bytes memory returnData) = target.staticcall(
            abi.encodeWithSignature("getValue()")
        );

        require(success, "Staticcall failed");

        // 解码返回值
        uint256 value = abi.decode(returnData, (uint256));
        return value;
    }
```

```
    // 尝试使用staticcall调用修改状态的函数（会失败）
    function unsafeSetValue(address target, uint256 newValue) external {
        // 这个调用会失败，因为setValue会修改状态
        (bool success, ) = target.staticcall(
            abi.encodeWithSignature("setValue(uint256)", newValue)
        );

        // success会是false，因为staticcall不允许修改状态
        require(success, "Staticcall failed: cannot modify state");
    }
}
```

staticcall的安全保障：

```
contract SecurityExample {
    // 使用staticcall确保不会意外修改状态
    function safeQuery(address target) external view returns (uint256) {
        (bool success, bytes memory data) = target.staticcall(
            abi.encodeWithSignature("getValue()")
        );

        require(success, "Query failed");

        // 即使target合约有恶意代码，也无法修改本合约的状态
        // staticcall保证了这一点
        return abi.decode(data, (uint256));
    }
}
```

3.4 三种方法的对比

让我们通过一个完整的对比示例来理解三种方法的区别：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 目标合约：用于演示三种调用方法的区别
contract Target {
    uint256 public value;
    address public sender;

    event ValueChanged(uint256 newValue, address caller);

    // 修改状态的函数
    function setValue(uint256 _value) external {
        value = _value;
        sender = msg.sender;
        emit ValueChanged(_value, msg.sender);
    }
```

```
    // 只读函数
    function getValue() external view returns (uint256) {
        return value;
    }
}

// 调用者合约：对比三种调用方法
contract Caller {
    // 本合约的状态变量
    uint256 public value;
    address public sender;

    event CallResult(string method, bool success, uint256 value, address sender);

    /**
     * @notice 使用call方法调用
     * @dev call在目标合约的上下文中执行，修改目标合约的状态
     */
    function testCall(address target, uint256 newValue) external {
        // 记录调用前的状态
        uint256 callerValueBefore = value;
        uint256 targetValueBefore = Target(target).value();

        // 使用call调用目标合约的setValue函数
        (bool success, ) = target.call(
            abi.encodeWithSignature("setValue(uint256)", newValue)
        );
        require(success, "Call failed");

        // 检查状态变化
        uint256 callerValueAfter = value;
        uint256 targetValueAfter = Target(target).value();

        // 结论：call修改了目标合约的状态，没有修改调用者的状态
        emit CallResult(
            "call",
            success,
            targetValueAfter,  // 目标合约的值被修改
            Target(target).sender()  // msg.sender是调用者合约
        );
    }

    /**
     * @notice 使用delegatecall方法调用
     * @dev delegatecall在调用者的上下文中执行，修改调用者的状态
     */
    function testDelegatecall(address target, uint256 newValue) external {
        // 记录调用前的状态
        uint256 callerValueBefore = value;
        uint256 targetValueBefore = Target(target).value();

        // 使用delegatecall调用目标合约的setValue函数
```

```
        (bool success, ) = target.delegatecall(
            abi.encodeWithSignature("setValue(uint256)", newValue)
        );
        require(success, "Delegatecall failed");

        // 检查状态变化
        uint256 callerValueAfter = value;
        uint256 targetValueAfter = Target(target).value();

        // 结论：delegatecall修改了调用者的状态，没有修改目标合约的状态
        emit CallResult(
            "delegatecall",
            success,
            callerValueAfter,  // 调用者的值被修改
            sender  // msg.sender是原始调用者（不是调用者合约）
        );
    }

    /**
     * @notice 使用staticcall方法调用
     * @dev staticcall只能调用view/pure函数，不能修改状态
     */
    function testStaticcall(address target) external view returns (uint256) {
        // 使用staticcall调用view函数
        (bool success, bytes memory returnData) = target.staticcall(
            abi.encodeWithSignature("getValue()")
        );
        require(success, "Staticcall failed");

        // 解码返回值
        uint256 value = abi.decode(returnData, (uint256));

        // 结论：staticcall只读取数据，不修改任何状态
        return value;
    }
}
```

对比表格：

特性	call	delegatecall	staticcall
执行上下文	被调用合约	调用者合约	被调用合约
可发送以太币	是	否	否
可修改状态	是	是	否
msg.sender	调用者合约	原始调用者	调用者合约
适用场景	通用调用	代理模式	只读查询
安全性	中等	低（需谨慎）	高

实际执行结果对比：

假设调用者合约调用`testCall(target, 42)`：

```
调用前：
- Caller.value = 0
- Target.value = 0

调用后（使用call）：
- Caller.value = 0（未改变）
- Target.value = 42（被修改）
- Target.sender = Caller地址
```

假设调用者合约调用`testDelegatecall(target, 88)`：

```
调用前：
- Caller.value = 0
- Target.value = 0

调用后（使用delegatecall）：
- Caller.value = 88（被修改！）
- Target.value = 0（未改变）
- Caller.sender = 原始用户地址（不是Caller地址）
```

3.5 选择正确的调用方法

根据不同的场景，选择合适的调用方法：

使用call的场景：

- 需要发送以太币
- 调用外部合约的普通函数
- 不确定使用哪种方法时的默认选择

使用delegatecall的场景：

- 代理模式（可升级合约）
- 库合约调用
- 需要保持msg.sender不变

使用staticcall的场景：

- 只读查询
- 调用view/pure函数
- 需要额外的安全保障

---

4. 安全的外部调用

外部调用是合约开发中最危险的操作之一。如果处理不当，可能导致重入攻击等严重安全问题。让我们详细学习如何安全地进行外部调用。

4.1 重入攻击防范

重入攻击是智能合约中最常见和最危险的安全漏洞之一。2016年的The DAO攻击就是利用了这个漏洞，导致损失了价值5000万美元的以太币。

重入攻击的原理：

重入攻击发生在合约在执行外部调用之前没有更新状态的情况下。恶意合约可以在接收以太币时再次调用原函数，利用未更新的状态重复提取资金。

不安全的示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 存在重入漏洞的银行合约
contract VulnerableBank {
    mapping(address => uint256) public balances;

    event Deposit(address indexed user, uint256 amount);
    event Withdrawal(address indexed user, uint256 amount);

    // 存款函数
    function deposit() external payable {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    // 危险！存在重入漏洞的提现函数
    function withdraw() external {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        // 问题：先转账，后更新状态
        // 如果msg.sender是恶意合约，它可以在receive函数中再次调用withdraw
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        // 状态更新在外部调用之后，存在重入风险
        balances[msg.sender] = 0;

        emit Withdrawal(msg.sender, amount);
    }

    // 查询合约总余额
    function getContractBalance() external view returns (uint256) {
        return address(this).balance;
    }
}

// 攻击合约：利用重入漏洞
contract Attacker {
```

```
    VulnerableBank public vulnerableBank;
    uint256 public attackCount;

    constructor(address _vulnerableBank) {
        vulnerableBank = VulnerableBank(_vulnerableBank);
    }

    // 接收以太币时触发重入攻击
    receive() external payable {
        // 限制攻击次数，避免Gas耗尽
        if (attackCount < 3 && address(vulnerableBank).balance > 0) {
            attackCount++;
            // 再次调用withdraw，此时balances[msg.sender]还没有被清零
            vulnerableBank.withdraw();
        }
    }

    // 发起攻击
    function attack() external payable {
        require(msg.value >= 1 ether, "Need at least 1 ether");
        attackCount = 0;

        // 步骤1：先存款
        vulnerableBank.deposit{value: msg.value}();

        // 步骤2：发起第一次提现
        vulnerableBank.withdraw();
        // 在receive函数中会触发多次重入调用
    }

    // 查询攻击者余额
    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

攻击流程：

```
1. 攻击者调用attack()，存入1 ether
2. 攻击者调用withdraw()
3. 合约向攻击者转账1 ether
4. 攻击者的receive()函数被触发
5. receive()中再次调用withdraw()
6. 此时balances[攻击者]还是1 ether（因为还没被清零）
7. 合约再次向攻击者转账1 ether
8. 重复步骤4-7，直到攻击次数达到限制
9. 最终攻击者提取了4 ether（1 ether本金 + 3 ether窃取）
```

安全的写法：

遵循"检查-效果-交互"（Checks-Effects-Interactions，CEI）模式：

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 安全的银行合约
contract SecureBank {
    mapping(address => uint256) public balances;

    // 重入锁：防止函数被重入调用
    bool private locked;

    event Deposit(address indexed user, uint256 amount);
    event Withdrawal(address indexed user, uint256 amount);

    // 重入锁修饰符
    modifier noReentrant() {
        require(!locked, "No reentrancy");
        locked = true;  // 设置锁
        _;              // 执行函数
        locked = false; // 释放锁
    }

    // 存款函数
    function deposit() external payable {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    // 安全的提现函数：遵循CEI模式
    function withdraw() external noReentrant {
        // 1. Checks（检查）：验证所有条件
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        // 2. Effects（效果）：先更新状态
        // 这是关键：在外部调用之前更新状态
        balances[msg.sender] = 0;

        // 3. Interactions（交互）：然后进行外部调用
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        emit Withdrawal(msg.sender, amount);
    }

    // 查询合约总余额
    function getContractBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

CEI模式的关键点：

1. **Checks（检查）**：首先验证所有前置条件
2. **Effects（效果）**：然后更新合约状态
3. **Interactions（交互）**：最后进行外部调用

这样即使发生重入，状态已经更新，攻击无法成功。

4.2 防护措施

除了CEI模式，还有其他重要的防护措施：

1. 使用重入锁：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract ReentrancyGuard {
    // 使用布尔变量作为锁
    bool private locked;

    // 重入锁修饰符
    modifier noReentrant() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

    // 在关键函数上使用修饰符
    function withdraw(uint256 amount) external noReentrant {
        // 安全地执行提现逻辑
    }
}
```

2. 限制Gas：

在调用外部合约时，可以限制传递的Gas数量，防止被调用合约执行过于复杂的操作。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract GasLimitExample {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;

        // 限制Gas：最多使用50000 Gas
        // 如果被调用的合约需要更多Gas，调用会失败
        (bool success, ) = msg.sender.call{gas: 50000, value: amount}("");
        require(success, "Transfer failed");
```

```
    }
}
```

3. 检查返回值：

永远要检查外部调用的返回值，不检查返回值可能导致交易失败却不自知。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract ReturnValueCheck {
    // 危险：不检查返回值
    function badTransfer(address to, uint256 amount) external {
        (bool success, ) = to.call{value: amount}("");
        // 如果success是false，代码继续执行，可能导致状态不一致
    }

    // 正确：检查返回值
    function goodTransfer(address to, uint256 amount) external {
        (bool success, ) = to.call{value: amount}("");
        require(success, "Transfer failed");
        // 如果失败，整个交易会回滚
    }
}
```

4. 使用OpenZeppelin的ReentrancyGuard：

OpenZeppelin提供了经过审计的重入锁实现，可以直接使用：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureContract is ReentrancyGuard {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) external nonReentrant {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

4.3 Gas限制的最佳实践

关于Gas限制，需要注意以下几点：

1. 设置合理的Gas限制：

```
contract GasLimitBestPractice {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;

        // 根据实际测试确定合适的Gas限制
        // 太低了会导致正常操作失败
        // 太高了会增加被攻击的风险
        (bool success, ) = msg.sender.call{gas: 50000, value: amount}("");
        require(success, "Transfer failed");
    }
}
```

2. 避免Gas限制过低：

Gas限制不能太低，否则正常的操作也无法完成。通常建议根据实际测试来确定合适的Gas限制值。

3. 考虑使用transfer或send：

对于简单的以太币转账，可以使用`transfer`或`send`，它们有固定的Gas限制（2300 Gas），更安全：

```
contract TransferExample {
    mapping(address => uint256) public balances;

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;

        // transfer有固定的2300 Gas限制，更安全
        payable(msg.sender).transfer(amount);
    }
}
```

---

5. 合约创建方式

在合约间调用的场景中，有时我们需要动态创建新的合约。Solidity提供了两种创建合约的方式：new关键字和create2。

5.1 new关键字

new是传统的创建方式，使用起来非常简单直接。

new关键字的特点：

1. **地址由创建者和nonce决定**：
    - 新合约地址 = f(创建者地址, nonce)

    - nonce是创建者的交易计数
    - 地址是随机的，无法提前知道
2. **立即部署**：
    - 创建后，合约会立即部署到链上
    - 返回新合约的地址
3. **简单易用**：
    - 语法简单，一行代码即可创建

基本语法：

```
ContractType newContract = new ContractType(arg1, arg2, ...);
```

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 简单的计数器合约
contract Counter {
    uint256 public count;
    address public owner;

    constructor(address _owner) {
        owner = _owner;
        count = 0;
    }

    function increment() external {
        require(msg.sender == owner, "Not owner");
        count++;
    }
}

// 工厂合约：使用new创建新合约
contract CounterFactory {
    // 记录所有创建的计数器地址
    address[] public counters;

    event CounterCreated(address indexed counterAddress, address owner);

    /**
     * @notice 使用new创建新的计数器合约
     * @return 新创建的计数器合约地址
     */
    function createCounter() external returns (address) {
        // 使用new关键字创建新合约
        // 构造函数参数是msg.sender（调用者地址）
        Counter newCounter = new Counter(msg.sender);

        // 获取新合约的地址
```

```
        address counterAddress = address(newCounter);

        // 记录新合约地址
        counters.push(counterAddress);

        // 触发事件
        emit CounterCreated(counterAddress, msg.sender);

        return counterAddress;
    }

    /**
     * @notice 查询所有创建的计数器数量
     */
    function getCounterCount() external view returns (uint256) {
        return counters.length;
    }

    /**
     * @notice 查询指定索引的计数器地址
     */
    function getCounter(uint256 index) external view returns (address) {
        require(index < counters.length, "Index out of range");
        return counters[index];
    }
}
```

使用示例：

```
// 部署工厂合约
CounterFactory factory = new CounterFactory();

// 创建第一个计数器
address counter1 = factory.createCounter();
// counter1的地址是随机的，无法提前知道

// 创建第二个计数器
address counter2 = factory.createCounter();
// counter2的地址也是随机的，与counter1不同
```

适用场景：

- 一般的合约创建需求
- 不需要预先知道合约地址的场景
- 简单的工厂模式
- 每次用户请求就创建一个新的合约实例

5.2 create2

create2是一个更高级的创建方式，它最大的特点是**地址可预先计算**。

create2的核心特点：

1. **地址可预先计算**：
   - 在合约部署之前，就可以计算出它将来会被部署到哪个地址
   - 地址计算公式：`address = keccak256(0xff, sender, salt, bytecode)`
2. **通过salt控制地址**：
   - salt是一个你提供的随机数（bytes32类型）
   - 通过改变salt，可以控制生成不同的地址
   - 相同的salt、sender和bytecode会产生相同的地址
3. **确定性部署**：
   - 可以在多个链上部署相同地址的合约
   - 便于跨链交互

基本语法：

```
ContractType newContract = new ContractType{salt: saltValue}(arg1, arg2, ...);
```

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 简单的计数器合约
contract Counter {
    uint256 public count;
    address public owner;

    constructor(address _owner) {
        owner = _owner;
        count = 0;
    }

    function increment() external {
        require(msg.sender == owner, "Not owner");
        count++;
    }
}

// 工厂合约：使用create2创建确定性地址的合约
contract CounterFactory {
    event CounterCreated(address indexed counterAddress, bytes32 salt);

    /**
     * @notice 使用new创建（地址不可预测）
     */
    function createWithNew() external returns (address) {
        Counter counter = new Counter(msg.sender);
        return address(counter);
    }
```

```
    /**
     * @notice 使用create2创建（地址可预测）
     * @param salt 用于计算地址的盐值
     * @return 新创建的计数器合约地址
     */
    function createWithCreate2(bytes32 salt) external returns (address) {
        // 使用create2创建，指定salt值
        Counter counter = new Counter{salt: salt}(msg.sender);

        address counterAddress = address(counter);
        emit CounterCreated(counterAddress, salt);

        return counterAddress;
    }

    /**
     * @notice 预计算create2地址
     * @param salt 盐值
     * @param deployer 部署者地址（通常是本合约地址）
     * @return 预计算的合约地址
     */
    function computeAddress(bytes32 salt, address deployer)
        external
        view
        returns (address)
    {
        // 获取合约的创建字节码
        // type(Counter).creationCode 获取Counter合约的字节码
        // abi.encode(msg.sender) 编码构造函数参数
        bytes memory bytecode = abi.encodePacked(
            type(Counter).creationCode,
            abi.encode(msg.sender)
        );

        // 计算create2地址
        // 公式：keccak256(0xff + deployer + salt + keccak256(bytecode))
        bytes32 hash = keccak256(
            abi.encodePacked(
                bytes1(0xff),
                deployer,  // 工厂合约地址
                salt,      // 盐值
                keccak256(bytecode)  // 字节码的哈希
            )
        );

        // 将哈希转换为地址（取后20字节）
        return address(uint160(uint256(hash)));
    }
}
```

地址计算公式详解：

```
create2地址 = keccak256(
    0xff +                    // 固定前缀
    factory地址 +             // 创建者地址
    salt +                    // 盐值（32字节）
    keccak256(bytecode)       // 合约字节码的哈希
)
```

使用示例：

```
// 部署工厂合约
CounterFactory factory = new CounterFactory();

// 步骤1：选择一个salt值
bytes32 salt = 0x0000000000000000000000000000000000000000000000000000000000000001;

// 步骤2：预计算地址（在部署前就能知道地址）
address predictedAddress = factory.computeAddress(salt, address(factory));

// 步骤3：使用create2创建合约
address actualAddress = factory.createWithCreate2(salt);

// 步骤4：验证地址是否匹配
require(predictedAddress == actualAddress, "Address mismatch");
```

适用场景：

1. **状态通道**：
    - 链下计算好合约地址
    - 用户可以先向地址转账
    - 需要时再实际部署合约
2. **确定性部署**：
    - 多链部署相同地址的合约
    - 便于跨链交互
    - 简化地址管理
3. **Uniswap V2的应用**：
    - 每个交易对的地址可以通过公式计算
    - 用户可以在Pair未创建时就知道地址
    - Router可以直接计算目标地址，无需查询

create2 vs new对比：

特性	new	create2
地址可预测性	不可预测	完全可预测
地址计算	由nonce决定	由salt决定
适用场景	一般创建	高级应用
Gas消耗	较低	稍高
灵活性	高	中等

---

6. 实际应用场景

学完理论知识，让我们看看这些技术在实际项目中是如何应用的。通过实际案例，我们可以更好地理解合约间调用的价值和应用方式。

6.1 代币交换合约

代币交换是DeFi中最常见的应用场景。在代币交换合约中，我们需要调用ERC20合约来实现代币的转移。

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// ERC20接口定义
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
}

// 代币交换合约：使用接口调用实现代币交换
contract TokenSwap {
    // 声明两个代币接口变量
    IERC20 public tokenA;
    IERC20 public tokenB;

    // 交换比例（简化示例，使用固定比例）
    uint256 public exchangeRate = 1; // 1:1兑换

    // 事件：记录每次交换的详细信息
    event Swap(
        address indexed user,
        address indexed tokenIn,
```

```
        address indexed tokenOut,
        uint256 amountIn,
        uint256 amountOut
    );

    // 构造函数：初始化代币合约地址
    constructor(address _tokenA, address _tokenB) {
        // 将地址转换为接口类型，确保类型安全
        tokenA = IERC20(_tokenA);
        tokenB = IERC20(_tokenB);
    }

    /**
     * @notice 执行代币交换
     * @param amountA 要交换的tokenA数量
     * @dev 用户需要先调用tokenA的approve函数授权本合约
     */
    function swap(uint256 amountA) external {
        // 步骤1：检查合约是否有足够的tokenB用于交换
        // 使用接口的view函数查询余额，不消耗Gas
        uint256 contractBalanceB = tokenB.balanceOf(address(this));
        uint256 amountB = amountA * exchangeRate;
        require(contractBalanceB >= amountB, "Insufficient tokenB in contract");

        // 步骤2：从用户账户转移tokenA到本合约
        // transferFrom需要用户先调用tokenA.approve授权本合约
        // 使用接口调用，编译器会检查参数类型
        require(
            tokenA.transferFrom(msg.sender, address(this), amountA),
            "TokenA transfer failed"
        );

        // 步骤3：从本合约向用户转移tokenB
        // 使用接口调用，确保类型安全
        require(
            tokenB.transfer(msg.sender, amountB),
            "TokenB transfer failed"
        );

        // 步骤4：触发事件，记录交换信息
        // 前端应用可以监听这个事件来更新UI
        emit Swap(msg.sender, address(tokenA), address(tokenB), amountA, amountB);
    }

    /**
     * @notice 查询合约持有的代币余额
     */
    function getContractBalances()
        external
        view
        returns (uint256 balanceA, uint256 balanceB)
    {
```

```
        // 使用接口的view函数查询余额
        balanceA = tokenA.balanceOf(address(this));
        balanceB = tokenB.balanceOf(address(this));
    }
}
```

关键点：

- 通过接口调用确保了类型安全和代码可读性
- 清楚地知道token是一个IERC20合约，支持哪些操作一目了然
- 如果代币合约不符合ERC20标准，编译时就会报错

6.2 多签钱包

多签钱包是另一个常见的应用场景。它需要多个签名者确认后才能执行交易，使用call执行外部交易以实现灵活性。

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 多签钱包合约：使用call执行外部交易
contract MultiSigWallet {
    // 自定义错误
    error NotOwner();
    error TxNotExists();
    error TxAlreadyExecuted();
    error TxAlreadyConfirmed();
    error InsufficientConfirmations();
    error ExecutionFailed();

    // 所有者列表
    address[] public owners;
    mapping(address => bool) public isOwner;

    // 所需确认数
    uint256 public required;

    // 交易结构体
    struct Transaction {
        address to;         // 目标地址
        uint256 value;      // 发送的以太币数量
        bytes data;         // 调用数据
        bool executed;      // 是否已执行
        uint256 confirmations; // 确认数
    }

    // 交易列表
    Transaction[] public transactions;

    // 确认映射：交易ID => 所有者地址 => 是否已确认
```

```
    mapping(uint256 => mapping(address => bool)) public confirmations;

    // 事件
    event Deposit(address indexed sender, uint256 amount);
    event Submit(uint256 indexed txId);
    event Confirm(address indexed owner, uint256 indexed txId);
    event Execute(uint256 indexed txId);
    event ExecutionFailure(uint256 indexed txId);

    // 修饰符：只有所有者可以调用
    modifier onlyOwner() {
        if (!isOwner[msg.sender]) revert NotOwner();
        _;
    }

    // 修饰符：交易必须存在
    modifier txExists(uint256 _txId) {
        if (_txId >= transactions.length) revert TxNotExists();
        _;
    }

    // 修饰符：交易未执行
    modifier notExecuted(uint256 _txId) {
        if (transactions[_txId].executed) revert TxAlreadyExecuted();
        _;
    }

    // 构造函数：初始化所有者和所需确认数
    constructor(address[] memory _owners, uint256 _required) {
        require(_owners.length > 0, "Owners required");
        require(
            _required > 0 && _required <= _owners.length,
            "Invalid required"
        );

        for (uint256 i = 0; i < _owners.length; i++) {
            address owner = _owners[i];
            require(owner != address(0), "Invalid owner");
            require(!isOwner[owner], "Duplicate owner");

            isOwner[owner] = true;
            owners.push(owner);
        }

        required = _required;
    }

    // 接收以太币
    receive() external payable {
        emit Deposit(msg.sender, msg.value);
    }
```

```
    /**
     * @notice 提交交易
     * @param _to 目标地址
     * @param _value 发送的以太币数量
     * @param _data 调用数据
     * @return 交易ID
     */
    function submit(
        address _to,
        uint256 _value,
        bytes memory _data
    ) external onlyOwner returns (uint256) {
        uint256 txId = transactions.length;

        transactions.push(Transaction({
            to: _to,
            value: _value,
            data: _data,
            executed: false,
            confirmations: 0
        }));

        emit Submit(txId);
        return txId;
    }

    /**
     * @notice 确认交易
     * @param _txId 交易ID
     */
    function confirm(uint256 _txId)
        external
        onlyOwner
        txExists(_txId)
        notExecuted(_txId)
    {
        if (confirmations[_txId][msg.sender]) revert TxAlreadyConfirmed();

        confirmations[_txId][msg.sender] = true;
        transactions[_txId].confirmations += 1;

        emit Confirm(msg.sender, _txId);
    }

    /**
     * @notice 执行交易（使用call执行外部调用）
     * @param _txId 交易ID
     * @dev 使用call方法实现灵活性，可以调用任意合约的任意函数
     */
    function execute(uint256 _txId)
        external
        onlyOwner
```

```
        txExists(_txId)
        notExecuted(_txId)
    {
        Transaction storage transaction = transactions[_txId];

        // 检查确认数是否足够
        if (transaction.confirmations < required) {
            revert InsufficientConfirmations();
        }

        // 标记为已执行（防止重入）
        transaction.executed = true;

        // 使用call执行外部交易
        // call方法提供了灵活性，可以调用任意合约的任意函数
        (bool success, ) = transaction.to.call{value: transaction.value}(
            transaction.data
        );

        if (success) {
            emit Execute(_txId);
        } else {
            // 执行失败，恢复状态
            transaction.executed = false;
            emit ExecutionFailure(_txId);
            revert ExecutionFailed();
        }
    }
}
```

关键点：

- 通过call方法实现了灵活性，可以调用任意合约的任意函数
- 无法提前知道目标合约的接口，所以使用call而不是接口调用
- 严格检查返回值，确保交易执行成功
- 在实际的多签钱包实现中，通常还会结合Gas限制、重入锁等安全措施

6.3 代理合约

代理合约是合约升级的核心技术，它使用delegatecall来实现合约逻辑的升级。

完整示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 逻辑合约 V1：初始版本
contract ImplementationV1 {
    // 注意：存储布局必须与代理合约匹配
    uint256 public value;
    address public owner;
```

```
    /**
     * @notice 设置值
     * @param _value 要设置的值
     */
    function setValue(uint256 _value) external {
        // 这个函数会修改调用者合约（代理合约）的storage
        value = _value;
        // msg.sender是原始调用者，不是代理合约
        owner = msg.sender;
    }

    /**
     * @notice 获取值
     */
    function getValue() external view returns (uint256) {
        return value;
    }
}

// 逻辑合约 V2：升级版本（值翻倍）
contract ImplementationV2 {
    // 存储布局必须与V1和代理合约完全一致
    uint256 public value;
    address public owner;

    /**
     * @notice 设置值（新逻辑：值翻倍）
     * @param _value 要设置的值
     */
    function setValue(uint256 _value) external {
        // 新逻辑：值翻倍
        value = _value * 2;
        owner = msg.sender;
    }

    /**
     * @notice 获取值
     */
    function getValue() external view returns (uint256) {
        return value;
    }

    /**
     * @notice 新增功能：重置值
     * @dev V1没有这个函数，升级后可以使用
     */
    function reset() external {
        value = 0;
    }
}

// 代理合约：存储数据，通过delegatecall调用逻辑合约
```

```
contract Proxy {
    // 存储布局必须与逻辑合约完全一致
    address public implementation; // 逻辑合约地址
    uint256 public value;          // 与逻辑合约的value对应
    address public owner;           // 与逻辑合约的owner对应

    event Upgraded(address indexed newImplementation);

    // 构造函数：初始化逻辑合约地址
    constructor(address _implementation) {
        implementation = _implementation;
        owner = msg.sender;
    }

    /**
     * @notice 升级函数：更换逻辑合约
     * @param newImplementation 新的逻辑合约地址
     */
    function upgrade(address newImplementation) external {
        require(msg.sender == owner, "Not owner");
        implementation = newImplementation;
        emit Upgraded(newImplementation);
    }

    /**
     * @notice fallback函数：将所有调用转发到逻辑合约
     * @dev 使用delegatecall调用逻辑合约，逻辑合约的代码在代理合约的上下文中执行
     */
    fallback() external payable {
        address impl = implementation;
        require(impl != address(0), "Implementation not set");

        // 使用delegatecall调用逻辑合约
        // 逻辑合约的代码会在本合约（代理合约）的上下文中执行
        // 这意味着修改的是代理合约的storage，而不是逻辑合约的
        (bool success, bytes memory returnData) = impl.delegatecall(msg.data);

        if (!success) {
            // 如果调用失败，回滚
            assembly {
                returndatacopy(0, 0, returndatasize())
                revert(0, returndatasize())
            }
        }

        // 返回数据
        assembly {
            return(add(returnData, 0x20), mload(returnData))
        }
    }

    // 接收以太币
```

```
    receive() external payable {}
}
```

工作原理：

```
1. 用户调用 Proxy.setValue(50)
   ↓
2. Proxy的fallback函数被触发（因为Proxy没有setValue函数）
   ↓
3. fallback函数使用delegatecall调用 Implementation.setValue(50)
   ↓
4. Implementation的代码在Proxy的上下文中执行
   ↓
5. 修改的是Proxy的value（不是Implementation的）
   ↓
6. msg.sender仍然是原始用户（不是Proxy）
```

升级流程：

```
V1时期：
- Proxy.value = 0
- 调用setValue(50) → Proxy.value = 50（V1逻辑：直接赋值）

升级到V2：
- upgrade(V2地址) → 逻辑切换，但Proxy.value保持50

V2时期：
- 调用setValue(50) → Proxy.value = 100（V2逻辑：50*2=100）
- 调用reset() → Proxy.value = 0（V2新功能）
```

关键点：

- 通过delegatecall实现了调用，并且保持了msg.sender不变
- 在逻辑合约中，msg.sender仍然是原始的调用者，而不是代理合约
- 这对于权限控制非常重要
- 存储布局必须兼容，如果逻辑合约的存储布局发生变化，可能导致数据损坏

---

7. 最佳实践与常见错误

在实际开发中，遵循最佳实践可以避免很多问题。同时，了解常见错误可以帮助我们少踩坑。

7.1 最佳实践

1. 优先使用接口调用：

接口调用类型安全，代码可读性好，应该作为首选方案。只有在需要更高灵活性时才考虑使用底层调用方法。

```
// 推荐：使用接口调用
interface IERC20 {
```

```
    function transfer(address to, uint256 amount) external returns (bool);
}

contract GoodExample {
    IERC20 public token;

    function transferTokens(address to, uint256 amount) external {
        require(token.transfer(to, amount), "Transfer failed");
    }
}

// 不推荐：直接使用call（除非必要）
contract BadExample {
    address public token;

    function transferTokens(address to, uint256 amount) external {
        (bool success, ) = token.call(
            abi.encodeWithSignature("transfer(address,uint256)", to, amount)
        );
        require(success, "Transfer failed");
    }
}
```

2. 使用检查-效果-交互模式：

这是防止重入攻击的金科玉律。先更新状态，再进行外部调用。

```
// 正确：遵循CEI模式
function withdraw(uint256 amount) external {
    // 1. Checks：检查条件
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // 2. Effects：更新状态
    balances[msg.sender] -= amount;

    // 3. Interactions：外部调用
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}

// 错误：外部调用在状态更新之前
function badWithdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // 危险：先进行外部调用
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");

    // 状态更新在外部调用之后（重入风险）
    balances[msg.sender] -= amount;
}
```

3. 始终检查返回值：

无论是call、delegatecall还是staticcall，都要检查返回的success值。

```
// 正确：检查返回值
function callExternal(address target) external {
    (bool success, bytes memory data) = target.call(
        abi.encodeWithSignature("someFunction()")
    );
    require(success, "Call failed");
    // 处理返回数据...
}

// 错误：忽略返回值
function badCallExternal(address target) external {
    (bool success, ) = target.call(
        abi.encodeWithSignature("someFunction()")
    );
    // 如果success是false，代码继续执行，可能导致状态不一致
}
```

4. 使用重入锁：

对于涉及资金转移的关键函数，建议使用重入锁提供额外的保护。

```
// 推荐：使用重入锁
contract SecureContract {
    bool private locked;

    modifier noReentrant() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

    function withdraw(uint256 amount) external noReentrant {
        // 安全地执行提现逻辑
    }
}
```

5. 理解执行上下文：

一定要清楚call、delegatecall、staticcall的区别，选择正确的调用方法。

```
// call：在被调用合约的上下文中执行
function useCall(address target) external {
    target.call(...);  // 修改target的状态
}

// delegatecall：在调用者的上下文中执行
```

```
function useDelegatecall(address target) external {
    target.delegatecall(...);  // 修改本合约的状态
}

// staticcall：只读，不能修改状态
function useStaticcall(address target) external view {
    target.staticcall(...);  // 只读取数据
}
```

7.2 常见错误

1. 忘记检查返回值：

这是最常见的错误之一。很多开发者调用外部合约后，没有检查success值，导致即使调用失败也继续执行。

```
// 错误：忘记检查返回值
function badTransfer(address to, uint256 amount) external {
    (bool success, ) = to.call{value: amount}("");
    // 如果success是false，代码继续执行
    // 可能导致状态不一致
}

// 正确：检查返回值
function goodTransfer(address to, uint256 amount) external {
    (bool success, ) = to.call{value: amount}("");
    require(success, "Transfer failed");
}
```

2. 忽视重入风险：

不少开发者认为自己的合约很简单，不会有重入问题。但实际上，只要有外部调用，就存在重入风险。

```
// 危险：存在重入风险
function vulnerableWithdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // 先转账，后更新状态
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");

    balances[msg.sender] -= amount;  // 更新太晚
}

// 安全：遵循CEI模式
function safeWithdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount, "Insufficient balance");

    // 先更新状态
    balances[msg.sender] -= amount;

    // 后转账
```

```
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

3. 使用call不检查返回值：

call方法即使失败也不会抛出异常，只会返回false。如果不检查返回值，失败的调用会被忽略。

```
// 错误：不检查返回值
function badCall(address target) external {
    target.call(...);  // 如果失败，不会抛出异常
    // 代码继续执行，可能导致问题
}

// 正确：检查返回值
function goodCall(address target) external {
    (bool success, ) = target.call(...);
    require(success, "Call failed");
}
```

4. delegatecall存储布局不匹配：

使用代理模式时，如果逻辑合约的存储布局与代理合约不一致，会导致数据混乱。

```
// 代理合约
contract Proxy {
    address public implementation;  // slot 0
    uint256 public value;           // slot 1
}

// 错误：存储布局不匹配
contract BadImplementation {
    uint256 public value;           // slot 0（错误！）
    address public implementation;  // slot 1（错误！）
    // 存储布局与代理合约不一致，会导致数据错乱
}

// 正确：存储布局匹配
contract GoodImplementation {
    address public implementation;  // slot 0（匹配）
    uint256 public value;           // slot 1（匹配）
    // 存储布局与代理合约一致
}
```

7.3 注意事项总结

1. 理解执行上下文：

- call在被调用合约执行
- delegatecall在调用者合约执行
- staticcall只读，不能修改状态

2. 注意Gas消耗：

- 外部调用会消耗额外的Gas
- 特别是跨合约调用
- 在设计时要考虑Gas优化

3. 安全是第一要务：

- 无论如何优化，都不能牺牲安全性
- 宁可多花一些Gas做安全检查
- 也不要留下安全隐患

在区块链世界，安全永远是第一位的。一旦出现安全问题，损失往往是不可逆的。

---

8. 实践练习

理论学习之后，实践是巩固知识的最好方式。以下是不同难度的练习题目。

8.1 练习1：代币交换合约（二星难度）

任务：实现一个简单的ERC20代币交换功能，使用接口调用。

要求：

1. 使用接口调用确保类型安全
2. 检查返回值，确保转账成功
3. 添加事件日志，记录每次交换
4. 实现1:1的交换比例

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

contract TokenSwap {
    IERC20 public tokenA;
    IERC20 public tokenB;

    event Swap(
        address indexed user,
        uint256 amountA,
        uint256 amountB
```

```
        );

        constructor(address _tokenA, address _tokenB) {
            tokenA = IERC20(_tokenA);
            tokenB = IERC20(_tokenB);
        }

        function swap(uint256 amountA) external {
            require(
                tokenA.transferFrom(msg.sender, address(this), amountA),
                "Transfer A failed"
            );

            uint256 amountB = amountA; // 1:1兑换
            require(
                tokenB.transfer(msg.sender, amountB),
                "Transfer B failed"
            );

            emit Swap(msg.sender, amountA, amountB);
        }
    }
```

8.2 练习2：多签钱包（三星难度）

任务：实现一个简单的多签名钱包，使用call执行交易。

要求：

1. 实现重入锁保护
2. 应用Gas限制，防止恶意调用
3. 支持多个所有者
4. 需要达到指定确认数才能执行

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract MultiSigWallet {
    address[] public owners;
    mapping(address => bool) public isOwner;
    uint256 public required;

    struct Transaction {
        address to;
        uint256 value;
        bytes data;
        bool executed;
        uint256 confirmations;
    }
```

```
    Transaction[] public transactions;
    mapping(uint256 => mapping(address => bool)) public confirmations;

    bool private locked;

    modifier noReentrant() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

    modifier onlyOwner() {
        require(isOwner[msg.sender], "Not owner");
        _;
    }

    constructor(address[] memory _owners, uint256 _required) {
        require(_owners.length > 0, "Owners required");
        require(_required > 0 && _required <= _owners.length, "Invalid required");

        for (uint256 i = 0; i < _owners.length; i++) {
            isOwner[_owners[i]] = true;
            owners.push(_owners[i]);
        }
        required = _required;
    }

    function submit(address _to, uint256 _value, bytes memory _data)
        external
        onlyOwner
        returns (uint256)
    {
        uint256 txId = transactions.length;
        transactions.push(Transaction({
            to: _to,
            value: _value,
            data: _data,
            executed: false,
            confirmations: 0
        }));
        return txId;
    }

    function confirm(uint256 _txId) external onlyOwner {
        require(!confirmations[_txId][msg.sender], "Already confirmed");
        confirmations[_txId][msg.sender] = true;
        transactions[_txId].confirmations += 1;
    }

    function execute(uint256 _txId) external onlyOwner noReentrant {
        Transaction storage tx = transactions[_txId];
```

```
        require(!tx.executed, "Already executed");
        require(tx.confirmations >= required, "Insufficient confirmations");

        tx.executed = true;

        // 使用Gas限制
        (bool success, ) = tx.to.call{gas: 50000, value: tx.value}(tx.data);
        require(success, "Execution failed");
    }
}
```

8.3 练习3：代理合约（四星难度）

任务：使用delegatecall实现一个代理模式，支持升级功能。

要求：

1. 存储布局要匹配，确保升级时数据不会损坏
2. 实现升级功能，可以切换不同的逻辑合约
3. 使用fallback函数转发调用

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract Implementation {
    address public implementation;  // 必须与Proxy的存储布局匹配
    uint256 public value;
    address public owner;

    function setValue(uint256 _value) external {
        value = _value;
        owner = msg.sender;
    }

    function getValue() external view returns (uint256) {
        return value;
    }
}

contract Proxy {
    address public implementation;
    uint256 public value;
    address public owner;

    constructor(address _implementation) {
        implementation = _implementation;
        owner = msg.sender;
    }

    function upgrade(address newImplementation) external {
```

```
        require(msg.sender == owner, "Not owner");
        implementation = newImplementation;
    }

    fallback() external payable {
        address impl = implementation;
        require(impl != address(0), "Implementation not set");

        (bool success, bytes memory returnData) = impl.delegatecall(msg.data);

        if (!success) {
            assembly {
                returndatacopy(0, 0, returndatasize())
                revert(0, returndatasize())
            }
        }

        assembly {
            return(add(returnData, 0x20), mload(returnData))
        }
    }

    receive() external payable {}
}
```

8.4 练习4：create2工厂（三星难度）

任务：使用create2创建确定性地址的合约实例。

要求：

1. 实现地址预计算，在部署前就能知道合约地址
2. 完成salt值管理，确保不同的实例有不同的地址
3. 验证预计算的地址与实际部署的地址一致

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract Counter {
    uint256 public count;
    address public owner;

    constructor(address _owner) {
        owner = _owner;
        count = 0;
    }

    function increment() external {
        require(msg.sender == owner, "Not owner");
        count++;
```

```
    }
}

contract CounterFactory {
    event CounterCreated(address indexed counterAddress, bytes32 salt);

    function createWithCreate2(bytes32 salt) external returns (address) {
        Counter counter = new Counter{salt: salt}(msg.sender);
        address counterAddress = address(counter);
        emit CounterCreated(counterAddress, salt);
        return counterAddress;
    }

    function computeAddress(bytes32 salt, address deployer)
        external
        view
        returns (address)
    {
        bytes memory bytecode = abi.encodePacked(
            type(Counter).creationCode,
            abi.encode(msg.sender)
        );

        bytes32 hash = keccak256(
            abi.encodePacked(
                bytes1(0xff),
                deployer,
                salt,
                keccak256(bytecode)
            )
        );

        return address(uint160(uint256(hash)));
    }
}
```

---

9. 学习检查清单

完成本课后，你应该能够：

基础概念：

- [ ] 理解为什么需要合约间调用
- [ ] 知道合约间调用的各种方式
- [ ] 理解调用上下文的重要性

接口调用：

- [ ] 会定义接口
- [ ] 理解接口调用的优势

- [ ] 能够在实际项目中使用接口调用

底层调用方法：

- [ ] 理解call、delegatecall、staticcall的区别
- [ ] 知道何时使用哪种调用方法
- [ ] 理解执行上下文的影响

安全的外部调用：

- [ ] 理解重入攻击的原理
- [ ] 会使用CEI模式防止重入攻击
- [ ] 会使用重入锁
- [ ] 会检查返回值
- [ ] 会设置Gas限制

合约创建：

- [ ] 会使用new关键字创建合约
- [ ] 会使用create2创建确定性地址的合约
- [ ] 理解create2的地址计算公式

实际应用：

- [ ] 能够在代币交换中使用接口调用
- [ ] 能够在多签钱包中使用call
- [ ] 能够实现代理模式

最佳实践：

- [ ] 优先使用接口调用
- [ ] 遵循CEI模式
- [ ] 始终检查返回值
- [ ] 理解执行上下文

---

10. 总结

合约间调用是智能合约开发的核心技能。通过本课的学习，你应该已经掌握了：

1. **接口调用**：
    - 最安全、最规范的调用方式
    - 类型安全、代码可读性好、Gas效率高
    - 应该作为首选方案
2. **底层调用方法**：
    - **call**：最通用，可以发送以太币
    - **delegatecall**：在调用者上下文中执行，用于代理模式

    - **staticcall**：只读查询，最安全
3. **安全的外部调用**：
    - 遵循CEI模式防止重入攻击
    - 使用重入锁提供额外保护
    - 始终检查返回值
    - 合理设置Gas限制
4. **合约创建**：
    - **new**：传统创建方式，地址不可预测
    - **create2**：地址可预先计算，适合高级应用
5. **关键要点**：
    - 优先使用接口调用
    - 始终考虑安全性
    - 理解执行上下文