Solidity合约间调用

第8.1课：从入门到精通

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

讲师【layer】

课程大纲

接口调用

接口定义、使用方式、优势分析

安全的外部调用

重入攻击、Gas限制、最佳实践

call/delegatecall/staticcall

底层调用机制、使用场景、对比分析

实际应用场景

代币交换、多签钱包、代理合约

合约创建：new和create2

创建方式、地址确定性、应用场景

最佳实践与常见错误

安全建议、常见陷阱、注意事项

接口调用（Interface）

什么是接口

- 定义合约必须实现的函数签名
- 不包含函数实现
- 只声明函数，不包含状态变量

接口定义示例

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

接口使用示例

```
contract TokenSwap {
    IERC20 public token;

    function swap(uint256 amount) external {
        require(token.transfer(msg.sender, amount), "Transfer failed");
    }
}
```

接口调用的优势

类型安全
编译时检查，减少错误

代码可读性好
明确合约交互接口

Gas效率高
减少合约大小

易于测试
便于mock和模拟

底层调用方法

call

- ✓ 最通用的调用方法
- ✓ 可以发送以太币
- ✓ 在被调用合约上下文中执行

```
(bool success, bytes memory data) =
address.call{value: amount}(
    abi.encodeWithSignature("function(uint256)", arg)
);
```

delegatecall

- ✓ 在调用者上下文中执行
- ✗ 不能发送以太币
- ✓ 适合库合约和代理模式

```
(bool success, bytes memory data) =
address.delegatecall(
    abi.encodeWithSignature("function(uint256)", arg)
);
```

staticcall

- ✓ 保证不修改状态
- ✓ 只读查询
- ✓ 额外的安全保障

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

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			
可修改状态			
msg.sender	调用者合约	原始调用者	调用者合约

合约创建方式

new关键字

```
Token newToken = new Token(name, supply);
```

特点

- 传统创建方式
- 地址随机生成
- 简单直接

适用场景

- 一般的合约创建
- 不需要预先知道地址

create2

```
//address = keccak256(0xff, sender, salt, bytecode)
Token newToken = new Token{salt: salt}(name, supply);
```

特点

- 地址可预先计算
- 需要salt值
- 适合高级应用

适用场景

- 状态通道
- 反事实实例化

安全的外部调用

重入攻击防范

不安全示例

```
function withdraw() external {
    uint256 amount = balances[msg.sender];
    payable(msg.sender).transfer(amount); // 危险!
    balances[msg.sender] = 0;
}
```

安全示例

```
function withdraw() external {
    uint256 amount = balances[msg.sender];
    balances[msg.sender] = 0; // 先更新状态
    payable(msg.sender).transfer(amount); // 再发送
}
```

防护措施

- 检查-效果-交互模式
- 使用重入锁
- 限制Gas
- 检查返回值

Gas限制

```
(bool success, ) = target.call{gas: 50000}(
    abi.encodeWithSignature("function()")
);
require(success, "Call failed");
```

实际应用场景(一)

代币交换合约

使用接口调用ERC20函数，实现代币的交换和转移

```
require(token.transferFrom(msg.sender, address(this), amount));
require(token.transfer(recipient, amount));
```

通过接口调用确保类型安全和代码可读性

多签钱包

使用call执行外部交易，确保多重签名验证

```
(bool success, ) = target.call{value: amount}(data);
require(success, "Execution failed");
```

通过call方法执行外部交易，灵活且安全

实际应用场景(二)

代理合约

使用delegatecall转发调用，实现合约升级和功能扩展

```
(bool success, ) = implementation.delegatecall(data);
require(success, "Delegatecall failed");
```

通过delegatecall转发调用，保持msg.sender不变

最佳实践与常见错误

最佳实践

优先使用接口调用
类型安全，代码可读性好

使用检查-效果-交互模式
先更新状态，再进行外部调用

始终检查返回值
防止静默失败导致的安全问题

使用重入锁
防止外部函数调用时的重入攻击

常见错误

忘记检查返回值
可能导致交易失败而不自知

忽视重入风险
导致意外的函数调用执行

使用call不检查返回值
绕过Solidity的安全检查

delegatecall存储布局不匹配
导致意外的存储修改

注意事项

理解执行上下文　注意Gas消耗　安全是第一要务

课程总结

接口调用

- 类型安全
- 代码可读性好
- Gas效率高

底层调用

- call - 发送以太币，修改状态
- delegatecall - 委托执行
- staticcall - 只读查询

合约创建

- new - 传统创建，地址随机
- create2 - 地址可预计算

安全调用

- 检查-效果-交互模式
- 重入锁防护和Gas限制

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

实践作业

代币交换合约

★★

实现ERC20代币的交换功能，使用接口调用

- 检查返回值
- 添加事件日志

多签钱包

★★★

实现一个简单的多签名钱包，使用call执行交易

- 实现重入锁
- 应用Gas限制

代理合约

★★★★

使用delegatecall实现代理模式，支持升级功能

- 存储布局匹配
- 实现升级功能

create2工厂

★★★

使用create2创建确定性地址的合约实例

- 地址预计算
- salt值管理

完成这些作业来巩固对合约间调用技术的理解