



智能合约设计模式

Solidity智能合约开发系列课程 – 第9.2课

 讲师: [Layer]

6个核心设计模式



访问控制模式

权限管理，确保系统安全和完整性，控制谁可以执行敏感操作



提现模式

安全转账，防止重入攻击，确保资金安全转移



状态机模式

生命周期管理，通过定义状态和转换规则管理合约行为



代理模式

合约升级，通过分离存储和逻辑实现合约的可升级性



工厂模式

批量部署，统一管理和创建多个相同类型但配置不同的合约实例



紧急停止模式

风险控制，在紧急情况下暂停合约功能，保护资产安全

访问控制模式：必要性与实现

⚠️ 没有访问控制的问题

- 🚫 任何人都可以修改合约配置
- 💰 任何人都可以铸造代币
- 💵 任何人都可以提取资金

后果：灾难性的安全问题!

🛡️ 有访问控制的好处

- ✅ 只有授权者能执行敏感操作
- 👤 不同角色有不同权限
- ↔️ 可以转移或撤销权限

安全、灵活、可审计

🔒 Ownable模式实现

```
contract Ownable {
    address public owner;
    constructor() {
        owner = msg.sender;
    }
    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }
}
```

👥 RBAC实现

```
import "@openzeppelin/contracts/access/AccessControl.sol";
contract MyContract is AccessControl {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    function mint(address to, uint256 amount)
        public onlyRole(MINTER_ROLE)
    {
    }
}
```

提现模式：风险与安全方案

⚠️ 危险的提现方式

```
function withdraw() public {
    uint256 amount = balances[msg.sender];
    msg.sender.call{value: amount}(""); // 先转账
    balances[msg.sender] = 0;           // 后更新状态
}
```

三大核心风险：

🔁 重入攻击 (Reentrancy)：恶意合约在receive中再次调用withdraw

💰 Gas不足导致转账失败：transfer/send仅提供2300 Gas

🔒 合约余额被锁死：某一笔失败影响其他用户

🕒 著名案例

2016年The DAO攻击，损失5000万美元，根本原因就是重入漏洞

🛡️ 安全解决方案

Pull Over Push模式

```
contract PullPayment {
    mapping(address => uint256)
    public pendingWithdrawals;

    function withdraw() public {
        uint256 amount =
        pendingWithdrawals[msg.sender];
        require(amount > 0, "No
        funds");

        // 先更新状态

        pendingWithdrawals[msg.sender]
        = 0;

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

为什么安全？

🛡️ 有效防止重入攻击

CEI原则

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

    function withdraw() public {
        // 1. Checks: 检查条件
        uint256 amount =
        balances[msg.sender];
        require(amount > 0, "No
        balance");

        // 2. Effects: 更新状态
        balances[msg.sender] = 0;

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

👤 转账失败不影响其他用户

状态机模式：生命周期管理

什么是状态机？

状态机模式通过定义有限状态和状态转换规则，管理智能合约的生命周期和行为。

应用场景



众筹项目

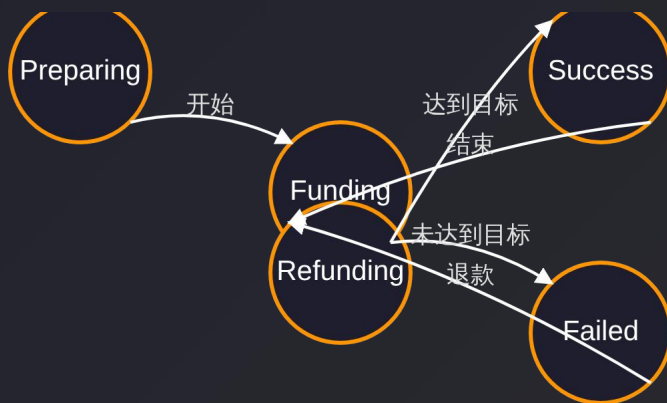


拍卖系统



游戏合约

ICO状态流转



代码实现

```
contract ICO {
  enum State { Preparing, Funding, Success, Failed }
  State public state = State.Preparing;

  modifier inState(State expected) {
    require(state == expected, "Invalid state");
    %; }

  function contribute() public payable
    inState(State.Funding) { // 投资 }

  function finalize() public inState(State.Funding) { if
    (totalRaised >= goal) { state = State.Success; }
    else { state = State.Failed; } }
}
```

代理模式：合约升级方案

🧩 模式原理

分离合约的数据存储和业务逻辑，实现可升级性

🏗️ 架构图

用户



代理合约(Proxy)



逻辑合约

🔑 delegatecall特性

- 在Proxy的上下文中执行
- 使用Proxy的storage
- msg.sender保持不变

</> Proxy合约实现

```
contract Proxy {
    address public implementation;
    address public admin;

    function upgrade(address newImpl) external {
        require(msg.sender == admin);
        implementation = newImpl;
    }

    fallback() external payable {
        address impl = implementation;
        assembly {
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)
            return datacopy(0, 0, returndatasize())
        }
        switch result
        case 0 { revert(0, returndatasize()) }
        default { return(0, returndatasize()) }
    }
}
```

⚠️ 关键注意事项

- ✓ 存储布局必须兼容：代理和逻辑合约的存储变量布局必须兼容

工厂模式：批量部署优化

为什么需要工厂？

需要统一管理、批量创建相同类型但具有不同配置的合约实例

应用场景

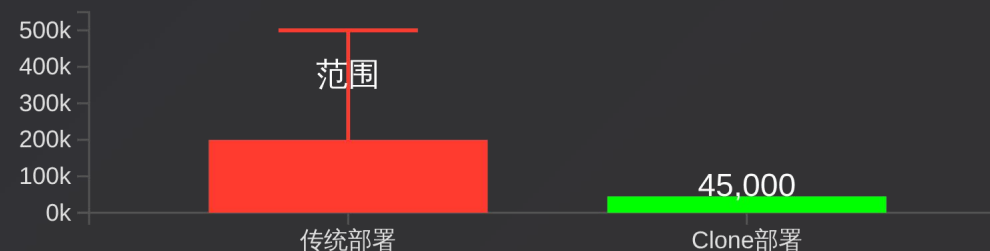


Uniswap为每个交易对创建Pair合约 (如ETH/USDT)



NFT市场为每个系列创建独立合约

Gas成本对比



传统部署 Clone部署

节省Gas: 80-90%

工厂模式优势

- 提高部署效率
- 简化合约管理
- 统一部署参数
- 降低总体Gas成本

紧急停止模式：风险控制

⚡ Circuit Breaker (断路器) 模式

提供一种在紧急情况下暂停合约操作的机制，防止进一步执行和潜在损失。

紧急情况场景

- 🛡️ 发现安全漏洞
- 💀 遭受攻击
- ⚠️ 预言机数据异常

最佳实践

- 👥 结合多签钱包
- 🕒 设置时间锁
- 📝 记录暂停原因

</> OpenZeppelin Pausable 实现

```
import "@openzeppelin/contracts/security/Pausable.sol";
contract MyToken is Pausable, Ownable {
    function transfer(address to, uint256 amount) public whenNotPaused
        // 暂停时无法调用
    {
        // 转账逻辑
    }
    function emergencyWithdraw() public whenPaused // 仅暂停时可调用
    {
        // 紧急提现逻辑
    }
    function pause() public onlyOwner {
        _pause();
    }
    function unpause() public onlyOwner {
        _unpause();
    }
}
```






💡 关键点

Circuit Breaker 是保护合约资产的重要机制，但应谨慎使用

设计模式对比与选择指南





模式	主要用途	适用场景	复杂度	Gas成本
访问控制	权限管理	几乎所有合约	★★	⚡ 低
提现模式	安全转账	涉及资金	★★★★	⚡⚡ 中
状态机	生命周期	众筹、拍卖	★★	⚡ 低
代理模式	合约升级	需要升级	★★★★★	⚡⚡⚡ 高
工厂模式	批量部署	多实例	★★★★	⚡⚡ 中
紧急停止	风险控制	金融合约	★★	⚡ 低

选择指南:

-  基础合约 (必备)
→ 访问控制 + 紧急停止
-  涉及资金 (必须)
→ 提现模式 + CEI原则
-  有生命周期
→ 状态机模式
-  需要升级
→ 代理模式 (谨慎)
-  批量部署
→ 工厂模式





模式组合应用案例

DeFi借贷协议

-  访问控制 (多签管理员)
-  紧急停止 (快速应对风险)
-  提现模式 (安全取款)
-  代理模式 (协议升级)




示例: Compound、AAVE

NFT交易市场

-  访问控制 (平台管理)
-  工厂模式 (创建集合)
-  状态机 (拍卖流程)
-  提现模式 (资金结算)

示例: OpenSea、Blur

DAO治理系统

-  访问控制 (投票权管理)
-  状态机 (提案流程)
-  代理模式 (DAO升级)
-  紧急停止 (治理暂停)

示例: Compound Governance

最佳实践与学习资源

✓ DO (推荐)

- ✓ 使用OpenZeppelin标准实现
- ✓ 从简单开始，逐步增加复杂度
- ✓ 组合多个模式解决复杂问题
- ✓ 充分测试所有场景
- ✓ 进行专业安全审计

✗ DON'T (避免)

- ✗ 不要重复造轮子
- ✗ 不要为了模式而模式
- ✗ 不要忽视Gas优化
- ✗ 不要单点故障

🎓 学习资源

官方文档

📖 Solidity官方文档

📖 OpenZeppelin Contracts

设计模式资源

🧩 Solidity Patterns

优秀项目源码

💎 Uniswap、Compound、AAVE

核心要点回顾与实践建议

🧩 6个核心模式

- 🔒 访问控制 → Ownable/RBAC
- 💰 提现模式 → Pull Payment + CEI
- 🔄 状态机 → 生命周期管理
- ↔ 代理模式 → 合约升级
- 🏭 工厂模式 → 批量部署
- ⚡ 紧急停止 → Circuit Breaker

★ 核心原则

- 🛡 安全第一：使用标准实现，充分测试
- 🪄 简洁优先：不过度设计，保持简单
- 📦 组合使用：多个模式配合解决问题
- 📖 持续学习：研究优秀项目，关注安全

📌 实践建议

- 🔧 在测试网充分测试
- 📄 阅读OpenZeppelin源码
- 👥 参与代码审查
- ⚠ 学习真实漏洞案例