Solidity智能合约开发知识

第4.1课：控制流语句

学习目标：掌握条件语句和循环语句的使用、理解循环的Gas成本问题、掌握错误处理机制、学会设计安全的控制流程

预计学习时间：2小时

难度等级：入门进阶

目录

---

1. 控制流概述

1.1 什么是控制流

控制流（Control Flow）是程序执行的顺序和路径。它决定了代码按照什么顺序执行，以及在什么条件下执行。

三种基本控制结构：

顺序结构：

```
// 从上到下依次执行
uint a = 10;
uint b = 20;
uint c = a + b;  // 先执行上面的代码，再执行这行
```

选择结构：

```
// 根据条件选择分支
if (balance > 100) {
    // 条件为true时执行
} else {
    // 条件为false时执行
}
```

循环结构：

```
// 重复执行代码块
for (uint i = 0; i < 10; i++) {
    // 这段代码会执行10次
}
```

1.2 Solidity中的特殊考虑

在传统编程中，控制流主要考虑逻辑正确性。但在Solidity智能合约开发中，还需要特别注意：

Gas成本问题：

- 每个操作都消耗Gas
- 循环会导致Gas成本线性或指数增长
- Gas超过区块限制会导致交易失败

状态修改的原子性：

- 交易要么全部成功，要么全部失败
- 中间状态不会保存
- 错误会导致整个交易回滚

安全性考虑：

- 循环可能被恶意利用（DoS攻击）
- 条件判断需要覆盖所有情况
- 外部调用可能失败或重入

不可变性：

- 代码部署后无法修改
- 逻辑漏洞无法修复
- 必须在开发阶段就保证正确性

---

2. 条件语句

2.1 if语句

if语句是最基本的条件控制语句。

基本语法：

```
if (条件) {
    // 条件为true时执行
}
```

基础示例：

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.0;

contract IfStatement {
    // 简单if语句
    function checkAge(uint age) public pure returns (bool) {
        if (age >= 18) {
            return true;
        }
        return false;
    }

    // 多个条件检查
    function checkEligibility(uint age, uint balance)
        public pure returns (bool)
    {
        if (age >= 18) {
            if (balance >= 1000) {
                return true;
            }
        }
        return false;
    }

    // 使用逻辑运算符
    function checkWithLogic(uint age, uint balance)
        public pure returns (bool)
    {
        if (age >= 18 && balance >= 1000) {
            return true;
        }
        return false;
    }
}
```

条件表达式：

if语句的条件必须是布尔值：

```
contract ConditionalExpressions {
    uint public value = 100;
    address public owner;

    function examples() public view {
        // 正确：比较运算
        if (value > 50) { }
        if (value >= 100) { }
        if (value == 100) { }
        if (value != 0) { }

        // 正确：逻辑运算
        if (value > 50 && value < 150) { }
        if (value == 100 || value == 200) { }
```

```
        if (!(value == 0)) { }

        // 正确：地址比较
        if (msg.sender == owner) { }
        if (owner != address(0)) { }

        // 错误：不是布尔值
        // if (value) { }  // 编译错误!
        // if (owner) { }  // 编译错误!
    }
}
```

2.2 if-else语句

if-else语句提供了两个互斥的执行分支。

```
contract IfElseStatement {
    // 基本if-else
    function checkValue(uint value)
        public pure returns (string memory)
    {
        if (value > 100) {
            return "High";
        } else {
            return "Low";
        }
    }

    // 多重检查
    function checkBalance(uint balance)
        public pure returns (string memory)
    {
        if (balance == 0) {
            return "Empty";
        } else {
            if (balance < 1000) {
                return "Low";
            } else {
                return "Good";
            }
        }
    }
}
```

执行流程：

```
开始
  ↓
检查条件
  ↓
条件为true?
  ├─ 是 → 执行if块 → 结束
  └─ 否 → 执行else块 → 结束
```

特点:

- 二选一的逻辑
- 必定执行其中一个分支
- 适合简单的二分判断

2.3 else if链

else if用于处理多个条件的情况。

```
contract ElseIfChain {
    // 评分系统
    function getGrade(uint score)
        public pure returns (string memory)
    {
        if (score >= 90) {
            return "A";
        } else if (score >= 80) {
            return "B";
        } else if (score >= 70) {
            return "C";
        } else if (score >= 60) {
            return "D";
        } else {
            return "F";
        }
    }

    // 会员等级判断
    function getMemberLevel(uint points)
        public pure returns (string memory)
    {
        if (points >= 10000) {
            return "Diamond";
        } else if (points >= 5000) {
            return "Platinum";
        } else if (points >= 1000) {
            return "Gold";
        } else if (points >= 100) {
            return "Silver";
        } else {
            return "Bronze";
        }
```

```
    }
}
```

执行规则：

条件	结果	是否继续检查
score >= 90	"A"	否
score >= 80	"B"	否
score >= 70	"C"	否
score >= 60	"D"	否
其他	"F"	-

重要提示：

条件从上到下依次检查，第一个满足的条件执行后，后续条件不再检查。

2.4 三元运算符

三元运算符是if-else的简化写法，适合简单的条件赋值。

语法：

```
条件 ? 值1 : 值2
```

如果条件为true，返回值1；否则返回值2。

对比示例：

```
contract TernaryOperator {
    // 使用if-else
    function maxWithIf(uint a, uint b)
        public pure returns (uint)
    {
        if (a > b) {
            return a;
        } else {
            return b;
        }
    }

    // 使用三元运算符（推荐）
    function maxWithTernary(uint a, uint b)
        public pure returns (uint)
    {
        return a > b ? a : b;
    }
```

```
    // 更多示例
    function getStatus(bool isActive)
        public pure returns (string memory)
    {
        return isActive ? "Active" : "Inactive";
    }

    // 嵌套三元运算符（不推荐，难以阅读）
    function complexTernary(uint value)
        public pure returns (string memory)
    {
        return value > 100 ? "High" :
               value > 50 ? "Medium" : "Low";
        // 可读性差，建议用if-else if
    }
}
```

使用建议：

适合使用：

- 简单的条件赋值
- 返回值选择
- 单行逻辑

不适合使用：

- 复杂逻辑
- 多层嵌套
- 需要执行多条语句

---

3. 循环语句

3.1 for循环

for循环是最常用的循环语句，适合已知循环次数的场景。

语法结构：

```
for (初始化; 条件; 更新) {
    // 循环体
}
```

基本示例：

```
contract ForLoop {
    // 基本for循环
    function sum(uint n) public pure returns (uint) {
        uint total = 0;
        for (uint i = 0; i <= n; i++) {
```

```
            total += i;
        }
        return total;
    }

    // 数组遍历
    function sumArray(uint[] memory arr)
        public pure returns (uint)
    {
        uint total = 0;
        for (uint i = 0; i < arr.length; i++) {
            total += arr[i];
        }
        return total;
    }

    // 倒序循环
    function countdown(uint n)
        public pure returns (uint[] memory)
    {
        uint[] memory result = new uint[](n);
        for (uint i = n; i > 0; i--) {
            result[n - i] = i;
        }
        return result;
    }

    // 步长为2的循环
    function sumEven(uint n) public pure returns (uint) {
        uint total = 0;
        for (uint i = 0; i <= n; i += 2) {
            total += i;
        }
        return total;
    }
}
```

执行流程：

```
1. 初始化：uint i = 0
     ↓
2. 检查条件：i < n?
     ├─ 否 → 退出循环
     └─ 是 ↓
3. 执行循环体
     ↓
4. 更新：i++
     ↓
5. 返回步骤2
```

for循环的组成部分：

1. **初始化**：`uint i = 0` - 设置循环变量初值
2. **条件**：`i < n` - 每次循环前检查
3. **更新**：`i++` - 每次循环后更新

3.2 while循环

while循环先判断条件，再执行循环体，适合循环次数不确定的场景。

语法结构：

```
while (条件) {
    // 循环体
}
```

示例：

```
contract WhileLoop {
    // 基本while循环
    function countdown(uint start)
        public pure returns (uint)
    {
        uint count = start;
        while (count > 0) {
            count--;
        }
        return count;
    }

    // 查找第一个非零值
    function findNonZero(uint[] memory arr)
        public pure returns (uint)
    {
        uint i = 0;
        while (i < arr.length && arr[i] == 0) {
            i++;
        }
        return i;  // 返回第一个非零值的索引
    }

    // 计算2的幂次
    function powerOfTwo(uint target)
        public pure returns (uint)
    {
        uint result = 1;
        while (result < target) {
            result *= 2;
        }
        return result;
    }
}
```

while循环特点：

- 先判断条件，再执行
- 最少执行0次（条件初始就为false）
- 适合条件驱动的循环

3.3 do-while循环

do-while循环先执行循环体，再判断条件，保证至少执行一次。

语法结构：

```
do {
    // 循环体
} while (条件);
```

示例：

```
contract DoWhileLoop {
    // 基本do-while
    function doWhileDemo(uint n)
        public pure returns (uint)
    {
        uint i = 0;
        uint result = 0;
        do {
            result += i;
            i++;
        } while (i < n);
        return result;
    }

    // 至少执行一次的场景
    function validateInput(uint value)
        public pure returns (bool)
    {
        uint attempts = 0;
        bool valid = false;

        do {
            attempts++;
            valid = (value > 0);
            value = value / 10;
        } while (value > 0 && attempts < 10);

        return valid;
    }
}
```

do-while特点：

- 先执行，再判断
- 至少执行1次
- 使用较少，适合特殊场景

3.4 三种循环的对比

特性	for	while	do-while
语法复杂度	复杂	简单	简单
循环次数	已知	未知	未知
最少执行	0次	0次	1次
适用场景	计数循环	条件循环	至少执行一次
使用频率	最高	中等	较低

选择建议：

1. **优先选择for循环**
    - 循环次数明确
    - 代码结构清晰
    - 不易出错
2. **其次考虑while**
    - 条件驱动
    - 灵活性高
3. **谨慎使用do-while**
    - 特殊场景（至少执行一次）
    - 使用较少

3.5 break和continue

break：立即退出循环

```
contract BreakStatement {
    // 查找目标值
    function findTarget(uint[] memory arr, uint target)
        public pure returns (bool, uint)
    {
        for (uint i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return (true, i);  // 找到就退出
            }
        }
        return (false, 0);
    }

    // 查找第一个满足条件的元素
```

```
    function findFirstGreaterThan(uint[] memory arr, uint threshold)
        public pure returns (uint)
    {
        for (uint i = 0; i < arr.length; i++) {
            if (arr[i] > threshold) {
                return i;  // 找到就退出
            }
        }
        revert("Not found");
    }
}
```

continue：跳过本次循环，继续下一次

```
contract ContinueStatement {
    // 只累加偶数
    function sumEven(uint n)
        public pure returns (uint)
    {
        uint total = 0;
        for (uint i = 0; i <= n; i++) {
            if (i % 2 != 0) {
                continue;  // 跳过奇数
            }
            total += i;
        }
        return total;
    }

    // 跳过零值
    function sumNonZero(uint[] memory arr)
        public pure returns (uint)
    {
        uint total = 0;
        for (uint i = 0; i < arr.length; i++) {
            if (arr[i] == 0) {
                continue;  // 跳过0
            }
            total += arr[i];
        }
        return total;
    }

    // 过滤无效地址
    function countValidAddresses(address[] memory addresses)
        public pure returns (uint)
    {
        uint count = 0;
        for (uint i = 0; i < addresses.length; i++) {
            if (addresses[i] == address(0)) {
                continue;  // 跳过零地址
```

```
            }
            count++;
        }
        return count;
    }
}
```

break vs continue：

关键字	作用	效果	使用场景
break	退出循环	终止整个循环	找到目标值
continue	跳过本次	继续下次循环	过滤特定值

---

4. 循环的Gas成本

4.1 Gas成本问题

循环是Solidity开发中最危险的操作之一，因为Gas成本会随循环次数线性或指数增长。

危险示例：

```
contract DangerousLoop {
    uint[] public data;

    // 危险：无限制的循环
    function sumAll() public view returns (uint) {
        uint total = 0;
        for (uint i = 0; i < data.length; i++) {
            total += data[i];
        }
        return total;
        // 如果data有10,000个元素，这个函数将无法执行！
    }
}
```

Gas消耗分析：

数组大小	循环次数	Gas消耗（估算）	结果
10	10	~5,000	成功
100	100	~50,000	成功
1,000	1,000	~500,000	可能失败
10,000	10,000	~5,000,000	必定失败

区块Gas限制：以太坊每个区块的Gas限制约为30,000,000，单个交易通常限制在10,000,000以内。

4.2 嵌套循环的危险

嵌套循环的Gas消耗呈指数增长。

```
contract NestedLoopDanger {
    // 危险：O(n²)复杂度
    function multiplicationTable(uint n)
        public pure returns (uint[][] memory)
    {
        uint[][] memory table = new uint[][](n);

        for (uint i = 0; i < n; i++) {          // 外循环
            table[i] = new uint[](n);
            for (uint j = 0; j < n; j++) {      // 内循环
                table[i][j] = (i + 1) * (j + 1);
            }
        }

        return table;
    }
}
```

Gas消耗对比：

输入大小(n)	循环总次数	Gas消耗（估算）	状态
n=10	100	~50,000	安全
n=50	2,500	~500,000	警告
n=100	10,000	~2,000,000	危险
n=200	40,000	~8,000,000	极危险

三大危害：

1. **Gas耗尽**：交易失败，但已消耗的Gas不退还
2. **资金锁定**：如果提款函数有循环，用户可能无法提款
3. **DoS攻击**：恶意用户可以故意让合约无法使用

4.3 循环安全实践

方案1：限制循环次数

```
contract SafeLoop {
    uint public constant MAX_ARRAY_SIZE = 100;

    // 限制输入大小
    function safeSum(uint[] memory data)
```

```
        public pure returns (uint)
    {
        require(data.length <= MAX_ARRAY_SIZE, "Array too large");

        uint total = 0;
        for (uint i = 0; i < data.length; i++) {
            total += data[i];
        }
        return total;
    }
}
```

方案2：使用mapping代替循环

```
contract UseMappingInstead {
    // 不好：需要循环查找
    address[] public users;

    function getUserBalance(address user) public view returns (uint) {
        // O(n) 复杂度，很慢
        for (uint i = 0; i < users.length; i++) {
            if (users[i] == user) {
                return i;
            }
        }
        return 0;
    }

    // 好：使用mapping，O(1)查询
    mapping(address => uint) public balances;

    function getBalance(address user) public view returns (uint) {
        return balances[user];  // 直接访问，快速
    }
}
```

方案3：分批处理

```
contract BatchProcessing {
    uint[] public data;
    uint public constant BATCH_SIZE = 50;

    // 分批处理大数组
    function processBatch(uint startIndex, uint batchSize)
        public
    {
        require(batchSize <= BATCH_SIZE, "Batch too large");

        uint endIndex = startIndex + batchSize;
        require(endIndex <= data.length, "Out of bounds");
```

```
        for (uint i = startIndex; i < endIndex; i++) {
            // 每次处理50个，分多次交易完成
            data[i] = data[i] * 2;
        }
    }
}
```

方案4：链下计算，链上存储

```
contract OffchainCalculation {
    mapping(address => uint) public rewards;

    // 前端计算好结果，合约只存储
    function setRewards(
        address[] calldata users,
        uint[] calldata amounts
    ) external {
        require(users.length == amounts.length, "Length mismatch");
        require(users.length <= 100, "Too many users");

        // 只是简单的赋值，不做复杂计算
        for (uint i = 0; i < users.length; i++) {
            rewards[users[i]] = amounts[i];
        }
    }
}
```

4.4 Gas优化效果对比

方法	Gas消耗	适用场景	推荐度
无限制循环	极高	危险，避免	不推荐
限制循环次数	中等	小数据集(<100)	推荐
mapping查询	恒定(低)	单个查询	强烈推荐
分批处理	分散	大数据集	推荐
链下计算	几乎为0	复杂计算	强烈推荐

实际案例对比：

处理1000个用户的积分：

方案A：循环遍历

```
function updateAll() public {
    for (uint i = 0; i < users.length; i++) {
        scores[users[i]] += 10;
    }
}
// Gas: ~2,000,000 (可能失败)
```

方案B：mapping直接更新

```
function updateUser(address user) public {
    scores[user] += 10;
}
// Gas: ~25,000 (每次)
// 用户自己调用，分散Gas成本
```

---

5. 错误处理

5.1 require - 输入验证

require是最常用的错误处理机制，用于验证外部输入和前置条件。

语法：

```
require(条件, "错误消息");
```

条件为false时：

- 交易立即回滚
- 显示错误消息
- 返还剩余Gas
- 所有状态改变撤销

基本示例：

```
contract RequireExample {
    mapping(address => uint) public balances;

    function transfer(address to, uint amount) public {
        // 检查1：地址有效性
        require(to != address(0), "Cannot transfer to zero address");

        // 检查2：金额有效性
        require(amount > 0, "Amount must be positive");

        // 检查3：余额充足
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // 所有检查通过，执行转账
        balances[msg.sender] -= amount;
```

```
        balances[to] += amount;
    }
}
```

典型使用场景：

```
contract RequireUseCases {
    address public owner;
    bool public paused;

    constructor() {
        owner = msg.sender;
    }

    // 场景1：权限检查
    function ownerOnly() public view {
        require(msg.sender == owner, "Not the owner");
        // 操作...
    }

    // 场景2：状态检查
    function whenNotPaused() public view {
        require(!paused, "Contract is paused");
        // 操作...
    }

    // 场景3：参数验证
    function setAge(uint age) public pure {
        require(age > 0 && age < 150, "Invalid age");
        // 设置年龄...
    }

    // 场景4：余额检查
    function withdraw(uint amount) public view {
        require(address(this).balance >= amount, "Insufficient contract balance");
        // 提款...
    }

    // 场景5：时间条件
    function afterDeadline(uint deadline) public view {
        require(block.timestamp >= deadline, "Too early");
        // 操作...
    }
}
```

5.2 assert - 内部检查

assert用于检查不应该失败的条件，主要用于检测代码bug和不变量。

语法：

```
assert(条件);
```

条件为false时：

- 交易回滚
- 表示代码有bug
- 返还剩余Gas（Solidity 0.8.0+）
- 不支持错误消息

示例：

```
contract AssertExample {
    mapping(address => uint) public balances;
    uint public totalSupply;

    function transfer(address to, uint amount) public {
        // require：验证外部输入
        require(to != address(0), "Invalid address");
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // 记录转账前，两个账户的余额总和
        uint balanceSumBefore = balances[msg.sender] + balances[to];

        // 执行转账
        balances[msg.sender] -= amount;
        balances[to] += amount;

        // assert：检查不变量
        // 逻辑上：转账后，两个账户的余额总和必须与转账前完全一致
        assert(balances[msg.sender] + balances[to] == balanceSumBefore);
        // 如果这个检查失败，说明余额计算逻辑出现了极其严重的 Bug（如溢出或赋值错误）
    }

    function mint(address to, uint amount) public {
        require(to != address(0), "Invalid address");

        uint supplyBefore = totalSupply;

        balances[to] += amount;
        totalSupply += amount;

        // 检查不变量：新的总供应 = 旧的 + 增发的
        assert(totalSupply == supplyBefore + amount);
    }
}
```

require vs assert对比：

特性	require	assert
用途	外部验证	内部检查
失败原因	用户错误/外部条件	代码bug
错误消息	支持	不支持
Gas返还	是	是（0.8.0+）
使用频率	非常高（90%）	很低（10%）

使用原则：

- **require**：验证用户输入、检查外部条件
- **assert**：检查代码逻辑、验证不变量

大多数情况下使用require，只在需要检查不变量时使用assert。

5.3 revert - 灵活的错误处理

revert提供了更灵活的错误处理方式。

基础用法：

```
contract RevertExample {
    mapping(address => uint) public balances;

    function complexCheck(address to, uint amount) public {
        // 方式1：带字符串消息
        if (to == address(0)) {
            revert("Invalid address");
        }

        if (amount == 0) {
            revert("Amount cannot be zero");
        }

        if (balances[msg.sender] < amount) {
            revert("Insufficient balance");
        }

        // 执行转账
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

自定义错误（Solidity 0.8.4+）：

```
contract CustomErrors {
    // 定义自定义错误
```

```
    error InsufficientBalance(uint requested, uint available);
    error InvalidAddress(address addr);
    error AmountTooLow(uint amount, uint minimum);
    error Unauthorized(address caller);

    mapping(address => uint) public balances;
    address public owner;
    uint public constant MIN_AMOUNT = 100;

    constructor() {
        owner = msg.sender;
    }

    function transfer(address to, uint amount) public {
        // 使用自定义错误
        if (to == address(0)) {
            revert InvalidAddress(to);
        }

        if (amount < MIN_AMOUNT) {
            revert AmountTooLow(amount, MIN_AMOUNT);
        }

        if (balances[msg.sender] < amount) {
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });
        }

        balances[msg.sender] -= amount;
        balances[to] += amount;
    }

    function adminFunction() public {
        if (msg.sender != owner) {
            revert Unauthorized(msg.sender);
        }
        // 管理员操作
    }
}
```

自定义错误的优势：

特性	字符串错误	自定义错误
Gas成本	高	低（节省约50%）
可带参数	否	是
类型安全	否	是
易于解析	难	易
前端集成	复杂	简单

Gas成本对比：

```
contract GasComparison {
    // 字符串错误：~24,000 gas
    function stringError() public pure {
        require(false, "This is an error message");
    }

    error CustomError();

    // 自定义错误：~12,000 gas
    function customError() public pure {
        revert CustomError();
    }
    // 节省：约50%
}
```

5.4 错误处理对比

特性	require	assert	revert
用途	输入验证	内部检查	灵活控制
条件判断	需要	需要	不需要
错误消息	支持	不支持	支持
自定义错误	支持	不支持	支持
Gas返还	是	是	是
使用频率	很高	低	中等

选择指南：

```
需要错误处理?
    ↓
简单条件判断?
    ├─ 是 → require
    └─ 否 → 复杂逻辑?
            ├─ 是 → revert
            └─ 否 → 检查不变量?
                    └─ 是 → assert
```

---

6. 状态机模式

6.1 什么是状态机

状态机（State Machine）是一种管理复杂状态转换的设计模式。

概念:

- 合约在任何时刻都处于某个特定状态
- 只能从当前状态转换到特定的下一个状态
- 某些操作只能在特定状态下执行

6.2 状态机实现

```
contract StateMachine {
    // 定义状态
    enum State {
        Preparing,    // 准备中
        Active,       // 进行中
        Checking,     // 检查中
        Success,      // 成功
        Failed,       // 失败
        Cancelled     // 已取消
    }

    State public currentState;

    // 状态检查modifier
    modifier inState(State expected) {
        require(currentState == expected, "Invalid state for this operation");
        _;
    }

    constructor() {
        currentState = State.Preparing;
    }

    // 只能在Preparing状态执行
    function start() public inState(State.Preparing) {
        currentState = State.Active;
    }
```

```
    // 只能在Active状态执行
    function contribute() public payable inState(State.Active) {
        // 贡献资金
    }

    // 只能在Active状态执行
    function check() public inState(State.Active) {
        currentState = State.Checking;
    }

    // 状态转换
    function finalize() public inState(State.Checking) {
        if (address(this).balance >= 100 ether) {
            currentState = State.Success;
        } else {
            currentState = State.Failed;
        }
    }

    // 紧急取消
    function cancel() public {
        require(
            currentState == State.Preparing || currentState == State.Active,
            "Cannot cancel at this stage"
        );
        currentState = State.Cancelled;
    }
}
```

状态转换图：

```
[Preparing]
    │
    │ start()
    ↓
[Active] ──────────┐
    │              │
    │ check()      │ contribute()
    ↓              │
[Checking] ────────┘
    │
    │ finalize()
    ├──────────┬──────────┐
    ↓          ↓          ↓
[Success] [Failed] [Cancelled]
```

6.3 众筹合约示例

```
contract Crowdfunding {
```

```
enum State { Fundraising, Success, Failed, PaidOut }

State public currentState = State.Fundraising;

address public creator;
uint public goal;
uint public deadline;
uint public totalFunded;

mapping(address => uint) public contributions;

constructor(uint _goal, uint _duration) {
    creator = msg.sender;
    goal = _goal;
    deadline = block.timestamp + _duration;
}

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

// 贡献资金（只在Fundraising状态）
function contribute()
    public payable inState(State.Fundraising)
{
    require(block.timestamp < deadline, "Campaign ended");
    require(msg.value > 0, "Must send ETH");

    contributions[msg.sender] += msg.value;
    totalFunded += msg.value;
}

// 检查目标（deadline后调用）
function checkGoalReached() public inState(State.Fundraising) {
    require(block.timestamp >= deadline, "Campaign not ended yet");

    if (totalFunded >= goal) {
        currentState = State.Success;
    } else {
        currentState = State.Failed;
    }
}

// 创建者提取资金（成功后）
function payout() public inState(State.Success) {
    require(msg.sender == creator, "Only creator can payout");

    currentState = State.PaidOut;
    payable(creator).transfer(address(this).balance);
}
```

```
    // 退款（失败后）
    function refund() public inState(State.Failed) {
        uint amount = contributions[msg.sender];
        require(amount > 0, "No contribution to refund");

        contributions[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

状态机的优势：

1. **状态转换清晰**：明确哪些操作在哪些状态下可以执行
2. **减少if-else嵌套**：用状态替代复杂的条件判断
3. **易于维护**：添加新状态和转换很容易
4. **防止无效操作**：在错误状态下的操作会被拒绝
5. **代码可读性高**：状态名称清楚表达合约状态

---

7. 最佳实践

7.1 错误处理最佳实践

原则1：尽早检查，尽早失败

```
contract EarlyCheck {
    mapping(address => uint) public balances;

    // 好的做法：所有检查放在开头
    function goodExample(address to, uint amount) public {
        // 所有验证在前面
        require(to != address(0), "Invalid address");
        require(amount > 0, "Invalid amount");
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // 执行业务逻辑
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }

    // 不好的做法：检查和业务逻辑混在一起
    function badExample(address to, uint amount) public {
        balances[msg.sender] -= amount;  // 还没检查就修改状态
        require(to != address(0), "Invalid address");  // 检查太晚
        balances[to] += amount;
    }
}
```

原则2：清晰的错误消息

```
contract ClearErrorMessages {
    // 不好：模糊的错误消息
    function badError(uint x) public pure {
        require(x > 0, "Error");  // 什么错误?
        require(x < 100, "Bad");  // 为什么bad?
    }

    // 好：清晰的错误消息
    function goodError(uint x) public pure {
        require(x > 0, "Value must be positive");
        require(x < 100, "Value exceeds maximum limit of 100");
    }
}
```

原则3：检查顺序优化

便宜的检查放前面，可以节省Gas。

```
contract CheckOrder {
    mapping(address => uint) public balances;
    mapping(address => bool) public whitelist;

    // 优化：便宜的检查在前
    function optimizedOrder(address to, uint amount) public view {
        require(amount > 0, "Invalid amount");              // 最便宜
        require(to != address(0), "Invalid address");       // 便宜
        require(balances[msg.sender] >= amount, "Low balance");  // 中等
        require(whitelist[to], "Not whitelisted");          // 稍贵
        // 如果前面的检查失败，后面的检查就不需要执行了
    }
}
```

原则4：使用自定义错误

```
contract CustomErrorPractice {
    error InsufficientBalance(uint requested, uint available);
    error InvalidRecipient(address recipient);
    error AmountBelowMinimum(uint amount, uint minimum);

    mapping(address => uint) public balances;
    uint public constant MIN_AMOUNT = 100;

    function transfer(address to, uint amount) public {
        if (to == address(0)) {
            revert InvalidRecipient(to);
        }

        if (amount < MIN_AMOUNT) {
            revert AmountBelowMinimum(amount, MIN_AMOUNT);
        }
```

```
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });
        }

        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

原则5：不要忽略返回值

```
contract CheckReturnValues {
    // 危险：忽略返回值
    function dangerousCall(address token, address to, uint amount) public {
        // 如果transfer失败，这里不会知道
        // token.transfer(to, amount);  // 危险!
    }

    // 安全：检查返回值
    function safeCall(address token, address to, uint amount) public {
        (bool success, ) = token.call(
            abi.encodeWithSignature("transfer(address,uint256)", to, amount)
        );
        require(success, "Transfer failed");
    }
}
```

原则6：合理使用modifier

```
contract ModifierForChecks {
    address public owner;
    bool public paused;

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

    modifier whenNotPaused() {
        require(!paused, "Paused");
        _;
    }

    // 使用modifier简化代码
    function adminFunction() public onlyOwner whenNotPaused {
        // 不需要在函数内重复写require
        // 代码更清晰
```

```
    }
}
```

7.2 循环使用最佳实践

实践1：优先考虑mapping

```
contract PreferMapping {
    // 不好：使用数组需要循环
    address[] public users;

    function findUser(address user) public view returns (bool) {
        for (uint i = 0; i < users.length; i++) {
            if (users[i] == user) {
                return true;
            }
        }
        return false;
    }
    // Gas：O(n)，随数组增长

    // 好：使用mapping，O(1)查询
    mapping(address => bool) public isUser;

    function checkUser(address user) public view returns (bool) {
        return isUser[user];
    }
    // Gas：O(1)，恒定成本
}
```

实践2：必须循环时严格限制

```
contract LimitLoops {
    uint[] public data;
    uint public constant MAX_ARRAY_SIZE = 100;

    function safePush(uint value) public {
        require(data.length < MAX_ARRAY_SIZE, "Array is full");
        data.push(value);
    }

    function safeProcess() public view returns (uint) {
        uint total = 0;
        uint len = data.length;  // 缓存length

        for (uint i = 0; i < len; i++) {
            total += data[i];
        }
        return total;
    }
}
```

实践3：避免嵌套循环

```
contract AvoidNesting {
    // 不好：嵌套循环
    function badPattern(uint[][] memory matrix)
        public pure returns (uint)
    {
        uint total = 0;
        for (uint i = 0; i < matrix.length; i++) {
            for (uint j = 0; j < matrix[i].length; j++) {
                total += matrix[i][j];  // O(n²)
            }
        }
        return total;
    }

    // 好：使用mapping或其他数据结构避免嵌套
    mapping(bytes32 => uint) public values;

    function goodPattern(uint x, uint y) public view returns (uint) {
        bytes32 key = keccak256(abi.encode(x, y));
        return values[key];  // O(1)
    }
}
```

实践4：分批处理

```
contract BatchProcessing {
    uint[] public data;
    uint public constant BATCH_SIZE = 50;

    function processBatch(uint startIndex) public {
        uint endIndex = startIndex + BATCH_SIZE;
        if (endIndex > data.length) {
            endIndex = data.length;
        }

        for (uint i = startIndex; i < endIndex; i++) {
            data[i] = data[i] * 2;
        }
    }

    function getTotalBatches() public view returns (uint) {
        return (data.length + BATCH_SIZE - 1) / BATCH_SIZE;
    }
}
```

7.3 安全编程原则

Checks-Effects-Interactions模式

这是智能合约开发中最重要的安全模式。

```
contract CEIPattern {
    mapping(address => uint) public balances;

    // 正确：遵循CEI模式
    function withdraw(uint amount) public {
        // 1. Checks - 检查
        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：先外部调用，后更新状态（重入攻击风险）
    function dangerousWithdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

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

        // 后更新状态（可能被重入攻击）
        balances[msg.sender] -= amount;
    }
}
```

为什么CEI模式重要?

重入攻击示例：

```
// 受害合约（有漏洞）
contract Vulnerable {
    mapping(address => uint) public balances;

    function withdraw() public {
        uint amount = balances[msg.sender];

        // 危险：先转账
        msg.sender.call{value: amount}("");

        // 后更新余额
        balances[msg.sender] = 0;
    }
}
```

```
// 攻击合约
contract Attacker {
    Vulnerable public victim;

    fallback() external payable {
        // 在收到钱后，再次调用withdraw
        // 因为余额还没更新，可以重复提取
        if (address(victim).balance > 0) {
            victim.withdraw();  // 重入攻击!
        }
    }
}
```

正确做法：

```
contract SafeContract {
    mapping(address => uint) public balances;

    function withdraw() public {
        uint amount = balances[msg.sender];
        require(amount > 0, "No balance");

        // 先更新状态
        balances[msg.sender] = 0;

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

---

8. 实战练习

练习1：投票系统

需求：

创建一个完整的投票系统：

1. 支持创建多个提案
2. 每个提案有截止时间
3. 只有owner可以创建提案
4. 每个地址只能投一次票
5. 可以查询投票结果
6. 可以获取获胜提案

参考代码框架：

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

```
contract VotingSystem {
    struct Proposal {
        string description;
        uint voteCount;
        uint deadline;
        bool exists;
    }

    address public owner;
    uint public proposalCount;

    mapping(uint => Proposal) public proposals;
    mapping(uint => mapping(address => bool)) public hasVoted;

    constructor() {
        owner = msg.sender;
    }

    // TODO: 实现创建提案
    function createProposal(string memory description, uint durationDays)
        public
    {
        // 检查权限
        // 验证参数
        // 创建提案
    }

    // TODO: 实现投票
    function vote(uint proposalId) public {
        // 检查提案存在
        // 检查是否已投票
        // 检查是否已截止
        // 执行投票
    }

    // TODO: 获取获胜提案
    function getWinner() public view returns (uint) {
        // 遍历所有提案
        // 找出票数最多的
    }
}
```

完整参考答案:

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

contract VotingSystem {
    struct Proposal {
        string description;
```

```
        uint voteCount;
        uint deadline;
        bool exists;
    }

    address public owner;
    uint public proposalCount;

    mapping(uint => Proposal) public proposals;
    mapping(uint => mapping(address => bool)) public hasVoted;

    event ProposalCreated(uint indexed proposalId, string description, uint deadline);
    event Voted(uint indexed proposalId, address indexed voter);

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call");
        _;
    }

    function createProposal(string memory description, uint durationDays)
        public onlyOwner
    {
        require(bytes(description).length > 0, "Empty description");
        require(durationDays >= 1 && durationDays <= 30, "Invalid duration");

        uint proposalId = proposalCount++;
        uint deadline = block.timestamp + (durationDays * 1 days);

        proposals[proposalId] = Proposal({
            description: description,
            voteCount: 0,
            deadline: deadline,
            exists: true
        });

        emit ProposalCreated(proposalId, description, deadline);
    }

    function vote(uint proposalId) public {
        require(proposals[proposalId].exists, "Proposal does not exist");
        require(block.timestamp <= proposals[proposalId].deadline, "Voting ended");
        require(!hasVoted[proposalId][msg.sender], "Already voted");

        hasVoted[proposalId][msg.sender] = true;
        proposals[proposalId].voteCount++;

        emit Voted(proposalId, msg.sender);
    }
```

```
    function getWinner() public view returns (uint winningProposalId) {
        uint maxVotes = 0;

        for (uint i = 0; i < proposalCount; i++) {
            if (proposals[i].voteCount > maxVotes) {
                maxVotes = proposals[i].voteCount;
                winningProposalId = i;
            }
        }

        return winningProposalId;
    }

    function getProposalInfo(uint proposalId)
        public view
        returns (
            string memory description,
            uint voteCount,
            uint deadline,
            bool hasEnded
        )
    {
        require(proposals[proposalId].exists, "Proposal does not exist");

        Proposal memory p = proposals[proposalId];
        return (
            p.description,
            p.voteCount,
            p.deadline,
            block.timestamp > p.deadline
        );
    }
}
```

练习2：安全批量转账

需求：

实现一个安全的批量转账功能：

1. 验证数组长度一致
2. 限制批量大小
3. 预先检查总金额
4. 验证所有地址
5. 保证原子性

完整参考答案：

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

```
contract SafeBatchTransfer {
    mapping(address => uint) public balances;
    uint public constant MAX_BATCH_SIZE = 50;

    event Transfer(address indexed from, address indexed to, uint amount);
    event BatchTransfer(address indexed from, uint count, uint totalAmount);

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function batchTransfer(
        address[] memory recipients,
        uint[] memory amounts
    ) public {
        // 1. 检查数组长度相等
        require(
            recipients.length == amounts.length,
            "Length mismatch"
        );

        // 2. 限制批量大小
        require(
            recipients.length <= MAX_BATCH_SIZE,
            "Batch too large"
        );

        // 3. 预先计算总金额
        uint totalAmount = 0;
        for (uint i = 0; i < amounts.length; i++) {
            totalAmount += amounts[i];
        }

        // 4. 检查余额充足
        require(
            balances[msg.sender] >= totalAmount,
            "Insufficient balance"
        );

        // 5. 验证所有地址和金额
        for (uint i = 0; i < recipients.length; i++) {
            require(recipients[i] != address(0), "Invalid address");
            require(amounts[i] > 0, "Invalid amount");
        }

        // 6. 执行转账（所有检查都通过后）
        for (uint i = 0; i < recipients.length; i++) {
            balances[msg.sender] -= amounts[i];
            balances[recipients[i]] += amounts[i];

            emit Transfer(msg.sender, recipients[i], amounts[i]);
        }
```

```
        emit BatchTransfer(msg.sender, recipients.length, totalAmount);
    }

    function getBalance(address user) public view returns (uint) {
        return balances[user];
    }
}
```

练习3：状态机众筹合约

挑战任务：

使用状态机模式创建一个众筹合约：

1. 定义4个状态：Fundraising, Success, Failed, PaidOut
2. 实现状态转换逻辑
3. 不同状态下允许不同操作
4. 使用modifier控制状态

提示：

- 使用enum定义状态
- 使用inState modifier
- 遵循CEI模式

---

9. 常见陷阱

陷阱1：无限循环

```
contract InfiniteLoopTrap {
    // 危险：忘记更新循环变量
    function badLoop() public pure {
        uint i = 0;
        while (i < 10) {
            // 忘记 i++
            // 无限循环！永远无法完成
        }
    }

    // 正确：记得更新
    function goodLoop() public pure {
        uint i = 0;
        while (i < 10) {
            // 处理逻辑
            i++;  // 不要忘记
        }
    }
}
```

陷阱2：循环中修改数组

```
contract ArrayModificationTrap {
    uint[] public array;

    // 危险：边遍历边修改长度
    function dangerousDelete() public {
        for (uint i = 0; i < array.length; i++) {
            if (array[i] == 0) {
                // 删除元素会改变长度
                delete array[i];  // 只是设为0，不改变length
            }
        }
    }

    // 危险：边遍历边pop
    function veryDangerous() public {
        for (uint i = 0; i < array.length; i++) {
            array.pop();  // 改变length，可能跳过元素
        }
    }

    // 正确：从后向前删除
    function safeDelete() public {
        for (uint i = array.length; i > 0; i--) {
            if (array[i - 1] == 0) {
                // 从后向前删除安全
                array.pop();
            }
        }
    }
}
```

陷阱3：整数溢出

```
contract OverflowTrap {
    // Solidity 0.8.0之前：危险
    function oldVersion() public pure returns (uint8) {
        uint8 x = 255;
        // x++;  // 溢出变成0（0.8.0之前）
        return x;
    }

    // Solidity 0.8.0+：自动检查
    function newVersion() public pure returns (uint8) {
        uint8 x = 255;
        // x++;  // 交易回滚
        return x;
    }
```

```
    // 使用unchecked需谨慎
    function withUnchecked() public pure returns (uint) {
        uint x = 0;
        unchecked {
            x--;  // 不会回滚，变成最大值
        }
        return x;
    }
}
```

陷阱4：block.timestamp操纵

```
contract TimestampTrap {
    // 不好：用于关键随机性
    function badRandom() public view returns (uint) {
        // 矿工可以在一定范围内操纵timestamp
        return block.timestamp % 100;
    }

    // 可以：用于时间检查
    function goodTimeCheck(uint deadline) public view returns (bool) {
        return block.timestamp >= deadline;
    }
}
```

陷阱5：深层嵌套

```
contract NestingTrap {
    // 不好：深层嵌套，难以理解
    function badNesting(uint a, uint b, uint c)
        public pure returns (string memory)
    {
        if (a > 0) {
            if (b > 0) {
                if (c > 0) {
                    if (a > b) {
                        if (b > c) {
                            return "Complex result";
                        }
                    }
                }
            }
        }
        return "Default";
    }

    // 好：使用early return
    function goodPattern(uint a, uint b, uint c)
        public pure returns (string memory)
    {
```

```
        if (a == 0) return "a is zero";
        if (b == 0) return "b is zero";
        if (c == 0) return "c is zero";
        if (a <= b) return "a not greater than b";
        if (b <= c) return "b not greater than c";

        return "Complex result";
    }
}
```

---

10. Gas优化技巧

技巧1：短路求值

逻辑运算符支持短路求值，可以节省Gas。

```
contract ShortCircuit {
    mapping(address => uint) public balances;

    // 利用短路求值优化
    function optimizedCheck(uint amount) public view returns (bool) {
        // 便宜的检查在前，如果失败就不执行后面的
        if (amount > 0 && balances[msg.sender] >= amount) {
            return true;
        }
        return false;
    }

    // 顺序很重要
    function checkOrder(address user, uint amount)
        public view returns (bool)
    {
        // 好：便宜的检查在前
        return amount > 0 && user != address(0) && balances[user] >= amount;

        // 不好：昂贵的检查在前
        // return balances[user] >= amount && amount > 0 && user != address(0);
    }
}
```

技巧2：缓存storage变量

```
contract CacheStorage {
    uint[] public data;

    // 未优化：重复读取storage
    function unoptimized() public view returns (uint) {
        uint total = 0;
        for (uint i = 0; i < data.length; i++) {  // 每次读取length
```

```
            total += data[i];
        }
        return total;
    }
    // Gas: ~25,000 (100个元素)

    // 优化：缓存length
    function optimized() public view returns (uint) {
        uint total = 0;
        uint len = data.length;  // 只读一次
        for (uint i = 0; i < len; i++) {
            total += data[i];
        }
        return total;
    }
    // Gas: ~23,000 (100个元素)
    // 节省: ~8%
}
```

技巧3：使用unchecked（谨慎）

```
contract UncheckedOptimization {
    uint[] public data;

    // 未优化：检查溢出
    function normalLoop() public view returns (uint) {
        uint total = 0;
        for (uint i = 0; i < data.length; i++) {
            total += data[i];
        }
        return total;
    }

    // 优化：确定不溢出时使用unchecked
    function optimizedLoop() public view returns (uint) {
        uint total = 0;
        uint len = data.length;

        for (uint i = 0; i < len; ) {
            total += data[i];
            unchecked {
                i++;  // i不可能溢出
            }
        }
        return total;
    }
    // 进一步节省gas
}
```

警告：只在确定不会溢出时使用unchecked！

技巧4：优化循环变量类型

```
contract LoopVariableType {
    // 不推荐：uint8需要额外转换
    function withUint8() public pure {
        for (uint8 i = 0; i < 10; i++) {
            // uint8需要额外的类型转换操作
        }
    }

    // 推荐：uint256是EVM原生类型
    function withUint256() public pure {
        for (uint256 i = 0; i < 10; i++) {
            // 直接使用，无额外成本
        }
    }
}
```

技巧5：批量操作合并

```
contract BatchOperations {
    mapping(address => uint) public scores;

    // 不好：多次交易
    function updateOne(address user, uint score) public {
        scores[user] = score;
    }
    // 需要调用n次，n笔交易费用

    // 好：一次交易完成
    function updateBatch(
        address[] calldata users,
        uint[] calldata scoreList
    ) external {
        require(users.length == scoreList.length, "Length mismatch");
        require(users.length <= 50, "Batch too large");

        for (uint i = 0; i < users.length; i++) {
            scores[users[i]] = scoreList[i];
        }
    }
    // 只需1笔交易费用
}
```

Gas节省对比：

优化方法	Gas节省	风险
缓存storage	~2,000/次	低
短路求值	~500/次	低
unchecked	~100/次	高
批量操作	~20,000/次	中
链下计算	大量	低

---

11. 常见问题解答

Q1：什么时候用if-else，什么时候用三元运算符?

答：根据复杂度和可读性选择。

使用三元运算符：

- 简单的条件赋值
- 单行逻辑
- 返回值选择

使用if-else：

- 复杂逻辑
- 多条语句
- 嵌套条件

Q2：为什么要避免循环?

答：循环在智能合约中有三大问题。

1. **Gas成本高**：每次循环都消耗Gas
2. **可能失败**：大循环可能超过Gas限制
3. **安全风险**：可能被恶意利用（DoS攻击）

解决方案：

- 优先使用mapping（O(1)查询）
- 必须循环时严格限制次数
- 考虑分批处理
- 链下计算，链上存储

Q3：require、assert、revert的区别?

答：三者用途不同。

require（最常用）：

- 验证用户输入

- 检查外部条件
- 支持错误消息

assert（很少用）：

- 检查代码逻辑
- 验证不变量
- 不支持错误消息

revert（中等使用）：

- 复杂条件判断
- 支持自定义错误
- 更灵活

Q4：如何防止重入攻击?

答：遵循CEI模式和使用ReentrancyGuard。

CEI模式（推荐）：

```
function withdraw(uint amount) public {
    // 1. Checks
    require(balances[msg.sender] >= amount);

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

    // 3. Interactions（后外部调用）
    payable(msg.sender).transfer(amount);
}
```

ReentrancyGuard：

```
bool private locked;

modifier noReentrant() {
    require(!locked, "Reentrant call");
    locked = true;
    _;
    locked = false;
}

function withdraw(uint amount) public noReentrant {
    // 函数逻辑
}
```

Q5：状态机模式的优势是什么?

答：状态机让复杂的状态管理变得简单。

优势：

1. 状态转换清晰明确
2. 减少if-else嵌套
3. 防止无效操作
4. 易于理解和维护
5. 安全性更高

应用场景：

- 众筹合约（筹款中→成功/失败）
- 拍卖合约（进行中→结束→已支付）
- 游戏合约（准备→进行→结束）
- 订单系统（创建→支付→发货→完成）

Q6：循环中可以修改数组吗?

答：非常危险，容易出错。

问题：

- 边遍历边修改长度可能跳过元素
- 索引可能越界
- 逻辑容易出错

解决方案：

- 从后向前删除
- 标记删除，循环后统一处理
- 使用新数组存储结果

Q7：for循环的三个部分可以省略吗?

答：可以省略，但要小心。

```
// 省略初始化
uint i = 0;
for (; i < 10; i++) { }

// 省略更新
for (uint i = 0; i < 10; ) {
    // 手动更新
    i++;
}

// 省略所有（无限循环）
for (;;) {
    // 需要用break退出
    if (condition) break;
}
```

---

12. 知识点总结

条件语句

if语句：

- 基本条件判断
- 条件必须是布尔值
- 可以嵌套使用

if-else：

- 二选一逻辑
- 必定执行一个分支

else if链：

- 多条件判断
- 从上到下检查
- 第一个满足就执行

三元运算符：

- 简化写法
- 适合简单赋值
- 避免多层嵌套

循环语句

for循环：

- 已知循环次数
- 最常用
- 三个组成部分

while循环：

- 条件驱动
- 先判断后执行
- 最少0次

do-while循环：

- 先执行后判断
- 至少执行1次
- 使用较少

break和continue：

- break：退出循环
- continue：跳过本次

Gas成本控制

关键原则：

1. 能不循环就不循环

2. 必须循环就限制次数
3. 链下计算，链上存储
4. 使用mapping代替数组遍历

优化技巧：

- 限制数组大小
- 缓存storage变量
- 分批处理
- 短路求值
- 使用unchecked（谨慎）

错误处理

三种方式：

- require：输入验证（最常用）
- assert：内部检查（少用）
- revert：灵活控制（中等）

最佳实践：

- 尽早检查，尽早失败
- 清晰的错误消息
- 使用自定义错误节省Gas
- 检查顺序影响成本

安全原则

CEI模式：

1. Checks - 检查条件
2. Effects - 更新状态
3. Interactions - 外部调用

其他原则：

- 防御性编程
- 避免重入攻击
- 状态机模式
- 原子性保证

---

13. 学习检查清单

完成本课后，你应该能够：

条件语句：

- [ ] 会使用if/if-else/else if
- [ ] 会使用三元运算符
- [ ] 理解条件判断的顺序

- [ ] 会组合逻辑运算符

循环语句：

- [ ] 会使用for/while/do-while
- [ ] 理解三种循环的区别
- [ ] 会使用break和continue
- [ ] 理解循环的执行流程

Gas成本：

- [ ] 理解循环的Gas问题
- [ ] 知道如何限制循环次数
- [ ] 会使用mapping替代循环
- [ ] 理解分批处理的思路
- [ ] 知道嵌套循环的危险

错误处理：

- [ ] 会使用require验证输入
- [ ] 理解assert的使用场景
- [ ] 会使用revert和自定义错误
- [ ] 理解三者的区别
- [ ] 会写清晰的错误消息

状态机：

- [ ] 理解状态机的概念
- [ ] 会使用enum定义状态
- [ ] 会实现状态转换
- [ ] 会用modifier控制状态

安全实践：

- [ ] 理解CEI模式
- [ ] 知道如何防止重入
- [ ] 会避免常见陷阱
- [ ] 掌握安全编程原则

---

14. 下一步学习

完成本课后，建议：

1. **实践所有示例代码**：在Remix中部署和测试
2. **完成练习题**：巩固知识点
3. **研究真实项目**：分析开源合约的控制流设计

4. **准备学习第4.2课**：特殊类型与全局变量

下节课预告：第4.2课 - 特殊类型与全局变量

我们将学习：

- address类型深入
- address vs address payable
- 转账方法对比
- 全局变量详解（msg/block/tx）
- enum枚举类型进阶
- constant和immutable的Gas优化

---

15. 扩展资源

官方文档：

- Solidity控制流：https://docs.soliditylang.org/en/latest/control-structures.html
- 错误处理：https://docs.soliditylang.org/en/latest/control-structures.html#error-handling

学习资源：

- Solidity by Example - If-Else：https://solidity-by-example.org/if-else/
- Solidity by Example - Loop：https://solidity-by-example.org/loop/
- Solidity by Example - Error：https://solidity-by-example.org/error/

安全资源：

- Smart Contract Weakness Classification
- Consensys Best Practices
- OpenZeppelin Security Guidelines

Gas优化：

- Gas Optimization Tips
- EVM Opcodes Gas Costs

实战项目：
研究开源项目的控制流设计：

- OpenZeppelin Contracts
- Uniswap V2/V3
- Compound Protocol

课程结束

恭喜你完成第4.1课！控制流是程序逻辑的基础，在智能合约中还需要特别注意Gas成本和安全性。

核心要点回顾：

- 条件语句实现逻辑分支
- 循环语句要严格控制
- 能不循环就不循环，必须循环就限制次数

- require是你最好的朋友
- 错误处理是安全的第一道防线
- 状态机让复杂逻辑变清晰
- 遵循CEI模式防止重入攻击

记住这句话：在区块链上，每一行代码都关系到真金白银。安全永远是第一位的。

继续加油，下节课见！