Solidity智能合约开发知识

第7.2课：错误处理和自定义错误

学习目标：掌握Solidity中的三种错误处理机制、理解自定义错误的优势、学会使用try-catch捕获异常、能够在实际项目中正确应用错误处理最佳实践

预计学习时间：2-3小时

难度等级：中级

目录

---

1. 错误处理基础概念

1.1 为什么需要错误处理

在智能合约开发中，错误处理不仅仅是让程序正常运行的技术手段，更是保障合约安全、优化Gas消耗、提升用户体验的关键环节。

错误处理的重要性：

1. **保障合约安全**：
   - 防止非法输入破坏合约状态
   - 防止整数溢出、下溢等运算错误
   - 防止未经授权的操作
   - 确保资金安全
2. **优化Gas消耗**：
   - 尽早检测错误可以避免不必要的计算
   - 自定义错误比字符串错误更节省Gas
   - 合理的错误处理可以减少失败交易的成本
3. **提升用户体验**：
   - 清晰的错误消息帮助用户理解失败原因
   - 避免用户因为不明确的错误而困惑
   - 便于前端应用提供友好的错误提示

4. **便于调试和维护**：
    - 明确的错误信息加速问题定位
    - 结构化的错误类型便于分类处理
    - 降低开发和维护成本

1.2 错误处理的基本原理

交易回滚机制：

当智能合约执行过程中遇到错误时，会触发交易回滚（Transaction Revert）。回滚意味着：

- 所有状态变更都会被撤销
- 合约状态恢复到交易执行前
- 已消耗的Gas不会退还（取决于错误类型）
- 可以返回错误信息给调用者

```
contract RevertExample {
    uint256 public balance = 100;

    function withdraw(uint256 amount) public {
        // 如果余额不足，这里会触发回滚
        require(balance >= amount, "余额不足");

        // 如果上面的require失败，下面的代码不会执行
        balance -= amount;
        // 状态不会被修改
    }
}
```

错误传播：

在Solidity中，错误会沿着调用链向上传播：

- 如果函数A调用函数B，函数B发生错误，错误会传播到函数A
- 除非使用try-catch捕获，否则错误会一直传播到外部调用者
- 整个交易会失败并回滚

```
contract ErrorPropagation {
    function functionA() public {
        functionB();  // 如果functionB失败，functionA也会失败
    }

    function functionB() public pure {
        require(false, "这个错误会传播到functionA");
    }
}
```

1.3 Solidity中的错误处理方式

Solidity提供了三种基本的错误处理机制：

1. **require**：用于验证条件，适合输入检查和状态验证
2. **assert**：用于检查不变量，适合内部一致性验证
3. **revert**：用于自定义错误处理，最灵活

此外，Solidity还支持：

- **自定义错误**（0.8.4+）：结构化的错误类型
- **try-catch**：捕获外部调用的异常

---

2. require/assert/revert详解

2.1 require - 输入验证和条件检查

基本语法：

```
require(condition, "错误消息");
// 或者
require(condition);
```

核心特点：

1. **用于条件验证**：检查输入参数、合约状态等
2. **交易可恢复**：失败时状态回滚
3. **Gas部分返还**：未使用的Gas会退还给调用者
4. **可以带错误消息**：便于调试和用户理解

使用场景：

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

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

    constructor() {
        owner = msg.sender;
        balances[msg.sender] = 1000;
    }

    // 场景1：输入参数验证
    function transfer(address to, uint256 amount) public {
        require(to != address(0), "接收地址不能为零地址");
        require(amount > 0, "转账金额必须大于0");
        require(balances[msg.sender] >= amount, "余额不足");

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

```
    // 场景2：权限检查
    function mint(address to, uint256 amount) public {
        require(msg.sender == owner, "只有所有者可以铸造");
        require(to != address(0), "接收地址不能为零地址");

        balances[to] += amount;
    }

    // 场景3：状态检查
    bool public paused = false;

    function withdraw(uint256 amount) public {
        require(!paused, "合约已暂停");
        require(balances[msg.sender] >= amount, "余额不足");

        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }

    // 场景4：时间检查
    uint256 public lockTime;

    function setLockTime(uint256 duration) public {
        require(msg.sender == owner, "只有所有者可以设置");
        lockTime = block.timestamp + duration;
    }

    function unlock() public {
        require(block.timestamp >= lockTime, "尚未到解锁时间");
        // 解锁操作...
    }
}
```

在上面的代码中：

- `transfer`函数使用require验证输入参数的有效性
- `mint`函数使用require检查调用者权限
- `withdraw`函数使用require检查合约状态
- `unlock`函数使用require检查时间条件

require的工作原理：

```
contract RequireInternals {
    uint256 public value = 100;

    function testRequire(uint256 newValue) public {
        // require内部实际上是这样工作的：
        // if (!condition) {
        //     revert("错误消息");
        // }

        require(newValue <= 200, "值不能超过200");
```

```
        value = newValue;
    }
}
```

当require条件为false时：

1. 交易立即停止执行
2. 所有状态变更回滚
3. 返回错误消息
4. 未使用的Gas退还给调用者

2.2 assert - 不变量检查

基本语法：

```
assert(condition);
```

核心特点：

1. **用于不变量检查**：验证理论上永远为真的条件
2. **交易不可恢复**：失败表示严重bug
3. **消耗全部Gas**：所有Gas都会被消耗，不会退还
4. **没有错误消息**：assert不支持错误消息

使用场景：

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

contract AssertExample {
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;

    constructor() {
        totalSupply = 1000;
        balanceOf[msg.sender] = 1000;
    }

    // 场景1：检查数学运算的正确性
    function transfer(address to, uint256 amount) public {
        require(balanceOf[msg.sender] >= amount, "余额不足");

        uint256 senderBalanceBefore = balanceOf[msg.sender];
        uint256 recipientBalanceBefore = balanceOf[to];

        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;

        // 检查不变量：总和应该保持不变
        assert(
```

```
            balanceOf[msg.sender] + balanceOf[to] ==
            senderBalanceBefore + recipientBalanceBefore
        );
    }

    // 场景2：检查状态一致性
    function mint(address to, uint256 amount) public {
        uint256 oldTotalSupply = totalSupply;
        uint256 oldBalance = balanceOf[to];

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

        // 检查不变量：总供应量变化应该等于余额变化
        assert(totalSupply - oldTotalSupply == balanceOf[to] - oldBalance);
    }

    // 场景3：检查合约状态的内部一致性
    function burn(uint256 amount) public {
        require(balanceOf[msg.sender] >= amount, "余额不足");

        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;

        // 检查不变量：总供应量不应该小于所有余额之和
        // 注意：这只是示例，实际中很难遍历所有地址
        assert(totalSupply >= balanceOf[msg.sender]);
    }
}
```

assert vs require的区别：

```
contract AssertVsRequire {
    uint256 public balance = 100;

    // 使用require：条件可能为假（用户错误）
    function withdrawWithRequire(uint256 amount) public {
        require(balance >= amount, "余额不足");  // 用户可能输入错误金额
        balance -= amount;
    }

    // 使用assert：条件永远应该为真（程序错误）
    function withdrawWithAssert(uint256 amount) public {
        require(balance >= amount, "余额不足");

        uint256 oldBalance = balance;
        balance -= amount;

        // 这个条件理论上永远为真，如果为假说明代码有bug
        assert(balance == oldBalance - amount);
    }
```

```
}
```

何时使用assert：

1. **检查溢出/下溢**（Solidity 0.8.0之前）：

```
contract OverflowCheck {
    function add(uint256 a, uint256 b) public pure returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);  // 检查是否溢出
        return c;
    }
}
```

2. **检查状态一致性**：

```
contract StateConsistency {
    uint256 public total;
    uint256 public partA;
    uint256 public partB;

    function update(uint256 _partA, uint256 _partB) public {
        partA = _partA;
        partB = _partB;
        total = partA + partB;

        // 检查不变量
        assert(total == partA + partB);
    }
}
```

3. **检查合约内部逻辑**：

```
contract InternalLogic {
    enum State { Created, Active, Completed }
    State public state;

    function complete() public {
        require(state == State.Active, "只能完成活跃状态的任务");
        state = State.Completed;

        // 检查状态转换是否正确
        assert(state == State.Completed);
    }
}
```

2.3 revert - 自定义错误处理

基本语法：

```
revert("错误消息");
// 或者使用自定义错误
revert CustomError(param1, param2);
```

核心特点：

1. **灵活的错误处理**：可以在任何位置使用
2. **支持自定义错误**：可以传递结构化的错误信息
3. **交易可恢复**：与require类似，会退还未使用的Gas
4. **更适合复杂逻辑**：在if-else中使用更自然

使用场景：

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

contract RevertExample {
    mapping(address => uint256) public balances;
    mapping(address => bool) public blacklist;

    // 定义自定义错误
    error InsufficientBalance(uint256 available, uint256 required);
    error Blacklisted(address account);
    error InvalidAmount(uint256 amount);

    constructor() {
        balances[msg.sender] = 1000;
    }

    // 场景1：复杂条件判断
    function transfer(address to, uint256 amount) public {
        // 使用revert处理复杂条件
        if (to == address(0)) {
            revert("接收地址不能为零地址");
        }

        if (blacklist[msg.sender]) {
            revert Blacklisted(msg.sender);
        }

        if (blacklist[to]) {
            revert Blacklisted(to);
        }

        if (amount == 0) {
            revert InvalidAmount(amount);
        }

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

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

    // 场景2：多路径错误处理
    function withdraw(uint256 amount, bool emergency) public {
        if (emergency) {
            // 紧急提现，不检查余额
            if (msg.sender != owner) {
                revert("只有所有者可以紧急提现");
            }
            // 紧急提现逻辑...
        } else {
            // 正常提现，检查余额
            if (balances[msg.sender] < amount) {
                revert InsufficientBalance(balances[msg.sender], amount);
            }
            balances[msg.sender] -= amount;
            // 提现逻辑...
        }
    }

    address public owner;

    // 场景3：提前退出函数
    function complexOperation(uint256 value) public {
        // 提前检查，如果不满足条件直接返回
        if (value > 1000) {
            revert("值过大");
        }

        // 执行复杂操作...
        for (uint256 i = 0; i < value; i++) {
            // 某些操作...

            if (/* 某个条件 */ false) {
                revert("操作过程中发生错误");
            }
        }
    }
}
```

revert vs require的选择：

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

    // 使用require：简单的条件检查
    function transferRequire(address to, uint256 amount) public {
        require(to != address(0), "无效接收地址");
```

```
        require(balances[msg.sender] >= amount, "余额不足");

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

    // 使用revert：复杂的条件判断
    function transferRevert(address to, uint256 amount) public {
        if (to == address(0)) {
            revert("无效接收地址");
        }

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

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

    error InsufficientBalance(uint256 available, uint256 required);
}
```

2.4 三种机制的对比

对比表格：

特性	require	assert	revert
用途	输入验证、条件检查	不变量检查、内部一致性	自定义错误处理
Gas返还	✅ 是	❌ 否（消耗全部）	✅ 是
错误消息	✅ 支持字符串	❌ 不支持	✅ 支持字符串和自定义错误
使用场景	函数入口验证	内部逻辑检查	复杂条件判断
失败影响	交易回滚	交易回滚	交易回滚
典型用例	余额检查、权限验证	数学运算验证	多路径错误处理

完整对比示例：

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

contract ErrorMechanismsComparison {
    uint256 public balance = 1000;
    uint256 public totalSupply = 1000;
    address public owner;
```

```
    error InsufficientBalance(uint256 available, uint256 required);

    constructor() {
        owner = msg.sender;
    }

    // require示例：输入验证
    function withdrawRequire(uint256 amount) public {
        require(amount > 0, "金额必须大于0");           // 输入验证
        require(msg.sender == owner, "只有所有者可以提现"); // 权限检查
        require(balance >= amount, "余额不足");         // 状态检查

        balance -= amount;
    }

    // assert示例：不变量检查
    function transferAssert(address to, uint256 amount) public {
        require(balance >= amount, "余额不足");

        uint256 oldBalance = balance;
        balance -= amount;

        // 检查不变量：新余额应该等于旧余额减去金额
        assert(balance == oldBalance - amount);
    }

    // revert示例：自定义错误
    function withdrawRevert(uint256 amount) public {
        if (amount == 0) {
            revert("金额必须大于0");
        }

        if (msg.sender != owner) {
            revert("只有所有者可以提现");
        }

        if (balance < amount) {
            revert InsufficientBalance(balance, amount);
        }

        balance -= amount;
    }

    // 组合使用示例
    function combinedExample(uint256 amount) public {
        // 1. 使用require进行输入验证
        require(amount > 0, "金额必须大于0");
        require(msg.sender == owner, "只有所有者可以操作");

        // 2. 使用revert处理复杂条件
        if (amount > balance / 2) {
            revert("单次提现不能超过余额的50%");
```

```
        }

        // 3. 执行操作
        uint256 oldBalance = balance;
        balance -= amount;

        // 4. 使用assert检查不变量
        assert(balance == oldBalance - amount);
        assert(balance <= totalSupply);
    }
}
```

选择建议：

1. **优先使用require**：
    - 用于所有需要验证的外部输入
    - 用于检查合约状态是否满足执行条件
    - 用于权限验证
2. **谨慎使用assert**：
    - 只用于检查理论上永远为真的条件
    - 用于开发阶段的调试
    - 用于检查合约内部逻辑的正确性
3. **灵活使用revert**：
    - 用于复杂的条件判断
    - 用于需要传递详细错误信息的场景
    - 结合自定义错误使用

---

3. 自定义错误(0.8.4+)

3.1 自定义错误的定义

自定义错误是Solidity 0.8.4版本引入的重要特性，它允许开发者创建结构化的、可重用的错误类型。

基本语法：

```
// 定义自定义错误
error ErrorName(type1 param1, type2 param2, ...);

// 使用自定义错误
revert ErrorName(value1, value2, ...);
```

简单示例：

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

contract CustomErrorBasics {
    // 定义不带参数的错误
```

```
    error Unauthorized();

    // 定义带参数的错误
    error InsufficientBalance(uint256 available, uint256 required);

    // 定义带多个参数的错误
    error InvalidTransfer(address from, address to, uint256 amount, string reason);

    mapping(address => uint256) public balances;
    address public owner;

    constructor() {
        owner = msg.sender;
        balances[msg.sender] = 1000;
    }

    function transfer(address to, uint256 amount) public {
        // 使用不带参数的错误
        if (msg.sender != owner && amount > 100) {
            revert Unauthorized();
        }

        // 使用带参数的错误
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(balances[msg.sender], amount);
        }

        // 使用带多个参数的错误
        if (to == address(0)) {
            revert InvalidTransfer(msg.sender, to, amount, "接收地址不能为零地址");
        }

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

3.2 自定义错误的优势

1. Gas优化：

自定义错误比字符串错误消耗更少的Gas，这在高频交易场景下能带来显著的成本节省。

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

contract GasOptimization {
    uint256 public balance = 100;

    // 自定义错误
    error InsufficientBalance(uint256 available, uint256 required);
```

```
    // 使用字符串错误（Gas消耗较高）
    function withdrawString(uint256 amount) public {
        require(balance >= amount, "Insufficient balance: available balance is less than required");
        balance -= amount;
    }

    // 使用自定义错误（Gas消耗较低）
    function withdrawCustomError(uint256 amount) public {
        if (balance < amount) {
            revert InsufficientBalance(balance, amount);
        }
        balance -= amount;
    }
}
```

Gas消耗对比（典型场景）：

- 字符串错误：约24,000-28,000 gas（取决于字符串长度）
- 自定义错误：约21,000-23,000 gas
- **节省：约10-20%的Gas**

2. 可重用性：

自定义错误可以在合约中定义一次，然后在多个函数中重复使用，减少代码重复。

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

contract ErrorReusability {
    // 在合约顶部定义所有错误
    error InsufficientBalance(uint256 available, uint256 required);
    error Unauthorized(address caller);
    error InvalidAmount(uint256 amount);
    error InvalidRecipient(address recipient);

    mapping(address => uint256) public balances;
    address public owner;

    constructor() {
        owner = msg.sender;
        balances[msg.sender] = 1000;
    }

    // 在多个函数中重用相同的错误
    function transfer(address to, uint256 amount) public {
        if (to == address(0)) {
            revert InvalidRecipient(to);  // 重用InvalidRecipient
        }
        if (amount == 0) {
            revert InvalidAmount(amount);  // 重用InvalidAmount
```

```
        }
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(balances[msg.sender], amount);  // 重用
InsufficientBalance
        }

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

    function withdraw(uint256 amount) public {
        if (amount == 0) {
            revert InvalidAmount(amount);  // 重用InvalidAmount
        }
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(balances[msg.sender], amount);  // 重用
InsufficientBalance
        }

        balances[msg.sender] -= amount;
    }

    function mint(address to, uint256 amount) public {
        if (msg.sender != owner) {
            revert Unauthorized(msg.sender);  // 重用Unauthorized
        }
        if (to == address(0)) {
            revert InvalidRecipient(to);  // 重用InvalidRecipient
        }

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

3. 可识别性:

自定义错误提供结构化的错误信息，外部合约和前端应用可以根据错误类型进行不同的处理。

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

// 代币合约
contract Token {
    error InsufficientBalance(uint256 available, uint256 required);
    error InsufficientAllowance(uint256 available, uint256 required);
    error InvalidRecipient(address recipient);

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    function transfer(address to, uint256 amount) public returns (bool) {
        if (to == address(0)) revert InvalidRecipient(to);
```

```
        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(balanceOf[msg.sender], amount);
        }

        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function transferFrom(address from, address to, uint256 amount) public returns (bool)
{
        if (to == address(0)) revert InvalidRecipient(to);
        if (balanceOf[from] < amount) {
            revert InsufficientBalance(balanceOf[from], amount);
        }
        if (allowance[from][msg.sender] < amount) {
            revert InsufficientAllowance(allowance[from][msg.sender], amount);
        }

        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        allowance[from][msg.sender] -= amount;
        return true;
    }
}

// 调用者合约可以根据错误类型进行处理
contract TokenCaller {
    Token public token;

    event TransferFailed(string reason);

    constructor(address _token) {
        token = Token(_token);
    }

    function safeTransfer(address to, uint256 amount) public {
        try token.transfer(to, amount) returns (bool success) {
            if (success) {
                // 转账成功
            }
        } catch Error(string memory reason) {
            // 捕获字符串错误
            emit TransferFailed(reason);
        } catch (bytes memory lowLevelData) {
            // 捕获自定义错误（可以解码错误类型和参数）
            // 前端可以根据错误签名判断是哪种错误
            emit TransferFailed("Custom error occurred");
        }
    }
}
```

3.3 自定义错误的最佳实践

1. 使用PascalCase命名：

自定义错误应该使用PascalCase（大驼峰）命名法，每个单词首字母大写。

```
// ✅ 好的命名
error InsufficientBalance(uint256 available, uint256 required);
error Unauthorized(address caller);
error InvalidRecipient(address recipient);
error TransferPaused();
error ExceedsMaxSupply(uint256 current, uint256 max);

// ❌ 不好的命名
error insufficientBalance(uint256 available, uint256 required);  // 应该用PascalCase
error IB(uint256 a, uint256 r);  // 太简短，不清晰
error Error1(uint256 x);  // 没有意义的命名
```

2. 添加相关上下文参数：

为错误添加必要的参数，便于调试和用户理解。

```
contract ContextfulErrors {
    // ✅ 好的错误定义：包含充足的上下文信息
    error InsufficientBalance(
        address account,
        uint256 available,
        uint256 required
    );

    error TransferLimitExceeded(
        address from,
        address to,
        uint256 amount,
        uint256 dailyLimit,
        uint256 usedToday
    );

    error TokenLocked(
        address token,
        uint256 lockedUntil,
        uint256 currentTime
    );

    // ❌ 不好的错误定义：缺少上下文
    error Failed();  // 太笼统
    error Error(uint256 code);  // 使用错误代码不如直接定义明确的错误

    mapping(address => uint256) public balances;
    mapping(address => uint256) public dailyUsed;
    uint256 public constant DAILY_LIMIT = 1000;
```

```
    function transfer(address to, uint256 amount) public {
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balances[msg.sender], amount);
        }

        if (dailyUsed[msg.sender] + amount > DAILY_LIMIT) {
            revert TransferLimitExceeded(
                msg.sender,
                to,
                amount,
                DAILY_LIMIT,
                dailyUsed[msg.sender]
            );
        }

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

3. 组织错误层次结构：

将相关的错误组织在一起，使用命名前缀来分组。

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

contract ErrorHierarchy {
    // 授权相关错误
    error Auth_Unauthorized(address caller);
    error Auth_InsufficientPermission(address caller, bytes32 requiredRole);
    error Auth_AccountLocked(address account, uint256 lockedUntil);

    // 余额相关错误
    error Balance_Insufficient(uint256 available, uint256 required);
    error Balance_ExceedsMaximum(uint256 amount, uint256 maximum);
    error Balance_Frozen(address account);

    // 转账相关错误
    error Transfer_InvalidRecipient(address recipient);
    error Transfer_InvalidAmount(uint256 amount);
    error Transfer_Paused();
    error Transfer_DailyLimitExceeded(uint256 amount, uint256 limit);

    // 时间相关错误
    error Time_TooEarly(uint256 currentTime, uint256 requiredTime);
    error Time_TooLate(uint256 currentTime, uint256 deadline);
    error Time_Expired(uint256 expiryTime);

    mapping(address => uint256) public balances;
```

```
    address public owner;
    bool public paused;

    function transfer(address to, uint256 amount) public {
        if (paused) revert Transfer_Paused();
        if (to == address(0)) revert Transfer_InvalidRecipient(to);
        if (amount == 0) revert Transfer_InvalidAmount(amount);
        if (balances[msg.sender] < amount) {
            revert Balance_Insufficient(balances[msg.sender], amount);
        }

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

4. 避免暴露敏感信息：

错误信息会被记录在区块链上，应避免暴露敏感数据。

```
contract SecureErrors {
    mapping(address => bytes32) private passwordHashes;
    mapping(address => uint256) private balances;

    // ❌ 危险：暴露了密码哈希
    error InvalidPassword(bytes32 providedHash, bytes32 expectedHash);

    // ✅ 安全：只说明密码错误，不暴露哈希值
    error InvalidPassword();

    // ❌ 危险：暴露了内部状态
    error InternalStateError(uint256 secretValue, address adminAddress);

    // ✅ 安全：只说明发生了内部错误
    error InternalStateError();

    function verifyPassword(bytes32 providedHash) public view returns (bool) {
        if (providedHash != passwordHashes[msg.sender]) {
            revert InvalidPassword();  // 不暴露期望的哈希值
        }
        return true;
    }
}
```

5. 文档化错误：

为错误添加NatSpec注释，说明错误的含义和触发条件。

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

```
contract DocumentedErrors {
    /**
     * @notice 当账户余额不足时抛出
     * @param account 余额不足的账户地址
     * @param available 账户当前可用余额
     * @param required 操作所需的最小余额
     */
    error InsufficientBalance(
        address account,
        uint256 available,
        uint256 required
    );

    /**
     * @notice 当调用者没有执行操作的权限时抛出
     * @param caller 尝试执行操作的地址
     * @param requiredRole 执行操作所需的角色标识
     */
    error Unauthorized(address caller, bytes32 requiredRole);

    /**
     * @notice 当转账被暂停时抛出
     * @dev 可以通过unpause()函数恢复转账功能
     */
    error TransferPaused();

    /**
     * @notice 当操作在时间锁定期内执行时抛出
     * @param currentTime 当前区块时间戳
     * @param unlockTime 解锁时间戳
     */
    error TimeLocked(uint256 currentTime, uint256 unlockTime);

    mapping(address => uint256) public balances;

    function transfer(address to, uint256 amount) public {
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balances[msg.sender], amount);
        }

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

3.4 自定义错误的完整示例

以下是一个完整的代币合约示例，展示了自定义错误的综合应用：

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

```
/**
 * @title TokenWithCustomErrors
 * @notice 使用自定义错误的ERC20代币合约示例
 */
contract TokenWithCustomErrors {
    // ============ 自定义错误定义 ============

    /// @notice 余额不足
    error InsufficientBalance(address account, uint256 available, uint256 required);

    /// @notice 授权额度不足
    error InsufficientAllowance(address owner, address spender, uint256 available, uint256
required);

    /// @notice 无效的接收地址
    error InvalidRecipient(address recipient);

    /// @notice 无效的金额
    error InvalidAmount(uint256 amount);

    /// @notice 未授权的操作
    error Unauthorized(address caller);

    /// @notice 转账功能已暂停
    error TransferPaused();

    /// @notice 超过最大供应量
    error ExceedsMaxSupply(uint256 requested, uint256 maxSupply);

    /// @notice 数组长度不匹配
    error ArrayLengthMismatch(uint256 length1, uint256 length2);

    // ============ 状态变量 ============

    string public name = "CustomError Token";
    string public symbol = "CET";
    uint8 public decimals = 18;
    uint256 public totalSupply;
    uint256 public constant MAX_SUPPLY = 1000000 * 10**18;

    address public owner;
    bool public paused;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    // ============ 事件 ============

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

```
    event Unpause();

    // ============ 修饰符 ============

    modifier onlyOwner() {
        if (msg.sender != owner) revert Unauthorized(msg.sender);
        _;
    }

    modifier whenNotPaused() {
        if (paused) revert TransferPaused();
        _;
    }

    // ============ 构造函数 ============

    constructor(uint256 _initialSupply) {
        owner = msg.sender;
        _mint(msg.sender, _initialSupply);
    }

    // ============ 公共函数 ============

    /**
     * @notice 转账代币
     * @param to 接收地址
     * @param amount 转账金额
     */
    function transfer(address to, uint256 amount)
        public
        whenNotPaused
        returns (bool)
    {
        if (to == address(0)) revert InvalidRecipient(to);
        if (amount == 0) revert InvalidAmount(amount);
        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balanceOf[msg.sender], amount);
        }

        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;

        emit Transfer(msg.sender, to, amount);
        return true;
    }

    /**
     * @notice 授权第三方使用代币
     * @param spender 被授权地址
     * @param amount 授权金额
     */
    function approve(address spender, uint256 amount) public returns (bool) {
```

```
        if (spender == address(0)) revert InvalidRecipient(spender);

        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    /**
     * @notice 从授权额度中转账
     * @param from 发送方地址
     * @param to 接收方地址
     * @param amount 转账金额
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public whenNotPaused returns (bool) {
        if (to == address(0)) revert InvalidRecipient(to);
        if (amount == 0) revert InvalidAmount(amount);

        if (balanceOf[from] < amount) {
            revert InsufficientBalance(from, balanceOf[from], amount);
        }

        if (allowance[from][msg.sender] < amount) {
            revert InsufficientAllowance(
                from,
                msg.sender,
                allowance[from][msg.sender],
                amount
            );
        }

        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        allowance[from][msg.sender] -= amount;

        emit Transfer(from, to, amount);
        return true;
    }

    /**
     * @notice 批量转账
     * @param recipients 接收地址数组
     * @param amounts 转账金额数组
     */
    function batchTransfer(
        address[] memory recipients,
        uint256[] memory amounts
    ) public whenNotPaused returns (bool) {
        if (recipients.length != amounts.length) {
```

```
            revert ArrayLengthMismatch(recipients.length, amounts.length);
        }

        uint256 totalAmount = 0;
        for (uint256 i = 0; i < amounts.length; i++) {
            totalAmount += amounts[i];
        }

        if (balanceOf[msg.sender] < totalAmount) {
            revert InsufficientBalance(msg.sender, balanceOf[msg.sender], totalAmount);
        }

        for (uint256 i = 0; i < recipients.length; i++) {
            if (recipients[i] == address(0)) {
                revert InvalidRecipient(recipients[i]);
            }

            balanceOf[msg.sender] -= amounts[i];
            balanceOf[recipients[i]] += amounts[i];
            emit Transfer(msg.sender, recipients[i], amounts[i]);
        }

        return true;
    }

    /**
     * @notice 铸造新代币（仅所有者）
     * @param to 接收地址
     * @param amount 铸造金额
     */
    function mint(address to, uint256 amount) public onlyOwner {
        if (to == address(0)) revert InvalidRecipient(to);
        if (totalSupply + amount > MAX_SUPPLY) {
            revert ExceedsMaxSupply(totalSupply + amount, MAX_SUPPLY);
        }

        _mint(to, amount);
    }

    /**
     * @notice 销毁代币
     * @param amount 销毁金额
     */
    function burn(uint256 amount) public {
        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balanceOf[msg.sender], amount);
        }

        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;

        emit Transfer(msg.sender, address(0), amount);
```

```
    }

    /**
     * @notice 暂停转账（仅所有者）
     */
    function pause() public onlyOwner {
        paused = true;
        emit Pause();
    }

    /**
     * @notice 恢复转账（仅所有者）
     */
    function unpause() public onlyOwner {
        paused = false;
        emit Unpause();
    }

    // ============ 内部函数 ============

    function _mint(address to, uint256 amount) private {
        totalSupply += amount;
        balanceOf[to] += amount;
        emit Transfer(address(0), to, amount);
    }
}
```

---

4. try-catch异常捕获

4.1 try-catch基础

try-catch是Solidity中用于捕获外部合约调用异常的机制。它让我们能够优雅地处理外部调用可能出现的各种错误情况。

基本语法：

```
try externalContract.someFunction() returns (returnType returnValue) {
    // 成功时执行的代码
} catch Error(string memory reason) {
    // 捕获require/revert的字符串错误
} catch Panic(uint errorCode) {
    // 捕获assert失败和内部错误
} catch (bytes memory lowLevelData) {
    // 捕获其他所有错误（包括自定义错误）
}
```

重要限制：

1. **只能捕获外部调用**：try-catch只能用于外部合约调用，不能用于当前合约的内部函数
2. **必须是外部调用**：被调用的函数必须标记为`external`或`public`

3. **不能捕获内部错误**：当前合约内部的错误会直接传播，不会被catch捕获

4.2 基础示例

以下是一个完整的try-catch使用示例：

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

// 外部合约：可能失败的计算器
contract Calculator {
    error DivisionByZero();

    function divide(uint256 a, uint256 b) external pure returns (uint256) {
        if (b == 0) revert DivisionByZero();
        return a / b;
    }

    function riskyOperation(uint256 value) external pure returns (uint256) {
        require(value < 100, "Value too high");
        return value * 2;
    }
}

// 调用者合约：使用try-catch处理异常
contract CalculatorCaller {
    Calculator public calculator;

    event OperationSuccess(uint256 result);
    event OperationFailed(string reason);
    event UnknownError();

    constructor(address _calculator) {
        calculator = Calculator(_calculator);
    }

    // 基础try-catch示例
    function safeDivide(uint256 a, uint256 b) public returns (uint256) {
        try calculator.divide(a, b) returns (uint256 result) {
            // 成功时执行
            emit OperationSuccess(result);
            return result;
        } catch Error(string memory reason) {
            // 捕获字符串错误
            emit OperationFailed(reason);
            return 0;
        } catch (bytes memory lowLevelData) {
            // 捕获自定义错误和其他错误
            emit UnknownError();
            return 0;
        }
    }
```

```
    // 处理require错误
    function safeRiskyOperation(uint256 value) public returns (uint256) {
        try calculator.riskyOperation(value) returns (uint256 result) {
            emit OperationSuccess(result);
            return result;
        } catch Error(string memory reason) {
            // 捕获"Value too high"错误
            emit OperationFailed(reason);
            return 0;
        } catch {
            // 简化的catch，捕获所有其他错误
            emit UnknownError();
            return 0;
        }
    }
}
```

4.3 catch子句类型

Solidity提供了三种类型的catch子句：

1. catch Error(string memory reason)：

捕获使用require或revert抛出的字符串错误。

```
contract StringErrorCatch {
    interface IExternal {
        function doSomething() external;
    }

    IExternal public externalContract;

    event ErrorCaught(string reason);

    function callExternal() public {
        try externalContract.doSomething() {
            // 成功
        } catch Error(string memory reason) {
            // 捕获字符串错误
            // 例如：require(false, "这是错误消息")
            emit ErrorCaught(reason);
        }
    }
}
```

2. catch Panic(uint errorCode)：

捕获Panic错误，这些错误通常由assert失败或运行时错误（如除以零、数组越界等）引起。

```
contract PanicCatch {
```

```
    interface IExternal {
        function riskyCalculation(uint256 a, uint256 b) external returns (uint256);
    }

    IExternal public externalContract;

    event PanicCaught(uint256 errorCode);

    function callExternal(uint256 a, uint256 b) public {
        try externalContract.riskyCalculation(a, b) returns (uint256 result) {
            // 成功
        } catch Panic(uint errorCode) {
            // 捕获Panic错误
            // errorCode可能的值:
            // 0x01: assert失败
            // 0x11: 算术运算溢出/下溢
            // 0x12: 除以零或模零
            // 0x21: 枚举转换错误
            // 0x22: 访问存储字节数组错误
            // 0x31: 对空数组调用.pop()
            // 0x32: 数组越界
            // 0x41: 分配过多内存
            // 0x51: 调用零值internal function
            emit PanicCaught(errorCode);
        }
    }
}
```

3. catch (bytes memory lowLevelData):

捕获所有其他类型的错误，包括自定义错误、没有错误消息的revert等。

```
contract LowLevelCatch {
    interface IExternal {
        function doSomething() external;
    }

    IExternal public externalContract;

    event LowLevelErrorCaught(bytes data);

    function callExternal() public {
        try externalContract.doSomething() {
            // 成功
        } catch (bytes memory lowLevelData) {
            // 捕获所有其他错误
            // 包括自定义错误
            // 可以解析lowLevelData获取错误详情
            emit LowLevelErrorCaught(lowLevelData);
        }
    }
}
```

4.4 完整的try-catch示例

以下是一个综合示例，展示了如何处理不同类型的错误：

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

// 外部ERC20代币合约接口
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// 模拟的代币合约
contract MockToken {
    mapping(address => uint256) public balanceOf;
    bool public paused = false;

    error TransferPaused();
    error InsufficientBalance(uint256 available, uint256 required);

    constructor() {
        balanceOf[msg.sender] = 1000;
    }

    function transfer(address to, uint256 amount) external returns (bool) {
        if (paused) revert TransferPaused();

        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(balanceOf[msg.sender], amount);
        }

        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function pause() external {
        paused = true;
    }

    function riskyCalculation(uint256 a, uint256 b) external pure returns (uint256) {
        return a / b;  // 如果b=0，会触发Panic
    }
}

// 代币处理合约
contract TokenHandler {
    IERC20 public token;
```

```
    event TransferSuccess(address indexed to, uint256 amount);
    event TransferFailedString(address indexed to, uint256 amount, string reason);
    event TransferFailedPanic(address indexed to, uint256 amount, uint256 errorCode);
    event TransferFailedCustom(address indexed to, uint256 amount, bytes data);

    constructor(address _token) {
        token = IERC20(_token);
    }

    /**
     * @notice 安全转账：捕获所有类型的错误
     */
    function safeTransfer(address to, uint256 amount) public {
        try token.transfer(to, amount) returns (bool success) {
            if (success) {
                emit TransferSuccess(to, amount);
            } else {
                emit TransferFailedString(to, amount, "Transfer returned false");
            }
        } catch Error(string memory reason) {
            // 捕获字符串错误（require/revert with string）
            emit TransferFailedString(to, amount, reason);
        } catch Panic(uint errorCode) {
            // 捕获Panic错误（assert/运行时错误）
            emit TransferFailedPanic(to, amount, errorCode);
        } catch (bytes memory lowLevelData) {
            // 捕获自定义错误和其他错误
            emit TransferFailedCustom(to, amount, lowLevelData);
        }
    }

    /**
     * @notice 批量安全转账：单个失败不影响其他
     */
    function batchSafeTransfer(
        address[] memory recipients,
        uint256[] memory amounts
    ) public {
        require(recipients.length == amounts.length, "Array length mismatch");

        for (uint256 i = 0; i < recipients.length; i++) {
            try token.transfer(recipients[i], amounts[i]) returns (bool success) {
                if (success) {
                    emit TransferSuccess(recipients[i], amounts[i]);
                }
            } catch Error(string memory reason) {
                emit TransferFailedString(recipients[i], amounts[i], reason);
                // 继续处理下一个，不中断整个批次
            } catch Panic(uint errorCode) {
                emit TransferFailedPanic(recipients[i], amounts[i], errorCode);
                // 继续处理下一个
            } catch (bytes memory lowLevelData) {
```

```
                emit TransferFailedCustom(recipients[i], amounts[i], lowLevelData);
                // 继续处理下一个
            }
        }
    }

    /**
     * @notice 条件转账：先检查余额再转账
     */
    function transferIfSufficient(address to, uint256 amount) public returns (bool) {
        // 先检查余额
        try token.balanceOf(address(this)) returns (uint256 balance) {
            if (balance < amount) {
                emit TransferFailedString(to, amount, "Insufficient contract balance");
                return false;
            }

            // 余额充足，尝试转账
            try token.transfer(to, amount) returns (bool success) {
                if (success) {
                    emit TransferSuccess(to, amount);
                    return true;
                } else {
                    emit TransferFailedString(to, amount, "Transfer returned false");
                    return false;
                }
            } catch Error(string memory reason) {
                emit TransferFailedString(to, amount, reason);
                return false;
            } catch {
                emit TransferFailedCustom(to, amount, "");
                return false;
            }
        } catch {
            emit TransferFailedString(to, amount, "Balance check failed");
            return false;
        }
    }
}
```

4.5 try-catch的使用场景

场景1：ERC20代币转账

处理代币转账可能出现的各种异常情况。

```
contract TokenTransferHandler {
    IERC20 public token;

    event TransferAttempted(address to, uint256 amount, bool success);
```

```
    function safeTransferToken(address to, uint256 amount) public {
        try token.transfer(to, amount) returns (bool success) {
            emit TransferAttempted(to, amount, success);
        } catch {
            // 转账失败，记录但不revert
            emit TransferAttempted(to, amount, false);
        }
    }
}
```

场景2：多合约交互

在复杂的DeFi协议中，需要调用多个外部合约。

```
contract DeFiProtocol {
    interface ILendingPool {
        function deposit(address asset, uint256 amount) external;
    }

    interface ISwapRouter {
        function swap(address tokenIn, address tokenOut, uint256 amountIn) external
returns (uint256);
    }

    ILendingPool public lendingPool;
    ISwapRouter public swapRouter;

    event StepFailed(string step, string reason);

    function complexOperation(
        address tokenIn,
        address tokenOut,
        uint256 amount
    ) public {
        // 步骤1：交换代币
        try swapRouter.swap(tokenIn, tokenOut, amount) returns (uint256 amountOut) {
            // 步骤2：存入借贷池
            try lendingPool.deposit(tokenOut, amountOut) {
                // 全部成功
            } catch Error(string memory reason) {
                emit StepFailed("deposit", reason);
                // 回滚或执行补救措施
            }
        } catch Error(string memory reason) {
            emit StepFailed("swap", reason);
            // 处理交换失败
        }
    }
}
```

场景3：合约升级和迁移

在合约升级过程中安全地调用新旧合约。

```
contract ContractMigration {
    address public oldContract;
    address public newContract;

    interface IOldContract {
        function getData(uint256 id) external view returns (bytes memory);
    }

    interface INewContract {
        function setData(uint256 id, bytes memory data) external;
    }

    event MigrationSuccess(uint256 id);
    event MigrationFailed(uint256 id, string reason);

    function migrateData(uint256[] memory ids) public {
        for (uint256 i = 0; i < ids.length; i++) {
            uint256 id = ids[i];

            // 从旧合约读取数据
            try IOldContract(oldContract).getData(id) returns (bytes memory data) {
                // 写入新合约
                try INewContract(newContract).setData(id, data) {
                    emit MigrationSuccess(id);
                } catch Error(string memory reason) {
                    emit MigrationFailed(id, reason);
                }
            } catch Error(string memory reason) {
                emit MigrationFailed(id, reason);
            }
        }
    }
}
```

4.6 try-catch注意事项

1. 避免嵌套过深

过深的嵌套会让代码难以理解和维护。

```
// ❌ 不好：嵌套过深
function badNestedTryCatch() public {
    try external1.call1() {
        try external2.call2() {
            try external3.call3() {
                // 太多层级
            } catch {
                // ...
            }
```

```
        } catch {
            // ...
        }
    } catch {
        // ...
    }
}

// ✅ 好：将逻辑拆分到不同函数
function goodSeparatedCalls() public {
    if (!tryCall1()) return;
    if (!tryCall2()) return;
    tryCall3();
}

function tryCall1() private returns (bool) {
    try external1.call1() {
        return true;
    } catch {
        return false;
    }
}
```

2. 注意Gas消耗

catch块中的代码也会消耗Gas，需要保持简单。

```
contract GasAwareTryCatch {
    // ❌ 不好：catch块中有复杂逻辑
    function badCatch() public {
        try externalCall() {
            // ...
        } catch {
            // 复杂的循环和计算
            for (uint256 i = 0; i < 1000; i++) {
                // 大量Gas消耗
            }
        }
    }

    // ✅ 好：catch块保持简单
    function goodCatch() public {
        try externalCall() {
            // ...
        } catch {
            // 只记录错误或设置标志
            emit ErrorOccurred();
        }
    }

    event ErrorOccurred();
    function externalCall() public {}
```

```
}
```

3. 处理返回值

正确处理try块中的返回值。

```
contract ReturnValueHandling {
    interface IExternal {
        function getValue() external returns (uint256);
    }

    IExternal public externalContract;

    // ✅ 正确处理返回值
    function handleReturnValue() public returns (uint256) {
        try externalContract.getValue() returns (uint256 value) {
            // 使用返回值
            return value * 2;
        } catch {
            // 返回默认值
            return 0;
        }
    }
}
```

---

5. 错误处理最佳实践

良好的错误处理不仅能让合约更安全,还能提升用户体验和降低Gas成本。以下是10个关键的最佳实践。

5.1 使用有意义的错误消息

错误消息应该清晰地说明问题所在,帮助开发者调试和用户理解。

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

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

    // ❌ 不好：错误消息不明确
    function badTransfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "Error");  // 太笼统
        require(to != address(0), "Invalid");              // 不够具体
        require(amount > 0, "Bad amount");                 // 不专业

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

    // ✅ 好：清晰、具体的错误消息
```

```
    function goodTransfer(address to, uint256 amount) public {
        require(
            balances[msg.sender] >= amount,
            "余额不足：您的余额少于转账金额"
        );
        require(
            to != address(0),
            "无效接收地址：接收地址不能为零地址"
        );
        require(
            amount > 0,
            "无效金额：转账金额必须大于0"
        );

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

    // ✅ 更好：使用自定义错误提供结构化信息
    error InsufficientBalance(address account, uint256 available, uint256 required);
    error InvalidRecipient(address recipient);
    error InvalidAmount(uint256 amount);

    function bestTransfer(address to, uint256 amount) public {
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balances[msg.sender], amount);
        }
        if (to == address(0)) {
            revert InvalidRecipient(to);
        }
        if (amount == 0) {
            revert InvalidAmount(amount);
        }

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

5.2 优先使用require进行输入验证

require应该放在函数开始处,尽早检测错误,避免不必要的计算。

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

    // ❌ 不好：在执行操作后才检查
    function badWithdraw(uint256 amount) public {
        balances[msg.sender] -= amount;  // 可能导致整数下溢
```

```
        require(balances[msg.sender] >= 0, "余额不足");  // 检查太晚
    }

    // ✅ 好：在函数开始处检查所有条件
    function goodWithdraw(uint256 amount) public {
        // 1. 先检查所有输入和状态
        require(amount > 0, "金额必须大于0");
        require(balances[msg.sender] >= amount, "余额不足");

        // 2. 然后执行操作
        balances[msg.sender] -= amount;
        totalTransferred += amount;

        // 3. 最后触发事件
        emit Withdrawal(msg.sender, amount);
    }

    event Withdrawal(address indexed account, uint256 amount);
}
```

5.3 使用assert验证不变量

assert应该只用于检查理论上永远不应该失败的条件。

```
contract InvariantChecks {
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;

    // 不变量：所有余额之和应该等于总供应量
    function transfer(address to, uint256 amount) public {
        require(balanceOf[msg.sender] >= amount, "余额不足");
        require(to != address(0), "无效接收地址");

        // 记录操作前的状态
        uint256 totalBefore = balanceOf[msg.sender] + balanceOf[to];

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

        // 使用assert检查不变量
        // 转账前后的总和应该保持不变
        assert(balanceOf[msg.sender] + balanceOf[to] == totalBefore);
    }

    function mint(address to, uint256 amount) public {
        uint256 oldTotalSupply = totalSupply;
        uint256 oldBalance = balanceOf[to];

        totalSupply += amount;
        balanceOf[to] += amount;
```

```
        // 检查不变量：总供应量的增加应该等于余额的增加
        assert(totalSupply - oldTotalSupply == balanceOf[to] - oldBalance);
    }
}
```

5.4 自定义错误优于字符串错误

在Solidity 0.8.4+版本中,应该优先使用自定义错误。

```
contract CustomVsString {
    uint256 public balance = 1000;

    // 字符串错误示例
    function withdrawString(uint256 amount) public {
        require(balance >= amount, "Insufficient balance: your balance is less than the
requested amount");
        // 长字符串消耗更多Gas
        balance -= amount;
    }

    // 自定义错误示例
    error InsufficientBalance(uint256 available, uint256 required);

    function withdrawCustom(uint256 amount) public {
        if (balance < amount) {
            revert InsufficientBalance(balance, amount);
        }
        // 更少的Gas消耗
        balance -= amount;
    }
}
```

Gas消耗对比（失败时）：

- 字符串错误：~24,000 gas
- 自定义错误：~21,000 gas
- **节省：~12.5%**

5.5 合理使用try-catch

try-catch应该只用于外部调用,不要过度使用。

```
contract TryCatchUsage {
    interface IExternal {
        function riskyOperation() external returns (bool);
    }

    IExternal public externalContract;

    // ✅ 正确：用于外部调用
```

```
    function callExternal() public {
        try externalContract.riskyOperation() returns (bool success) {
            if (success) {
                // 处理成功情况
            }
        } catch {
            // 处理失败情况
        }
    }

    // ❌ 错误：不能用于内部函数
    function internalFunction() internal {
        // 内部函数的错误会直接传播,无法catch
    }

    // ❌ 不好：不需要try-catch的情况
    function unnecessaryTryCatch(uint256 value) public {
        // 如果知道操作一定会成功,不需要try-catch
        if (value > 0) {
            // 执行操作
        }
    }
}
```

5.6 遵循Checks-Effects-Interactions模式

按照检查、效果、交互的顺序组织代码,这是防止重入攻击的最佳实践。

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

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

    event Withdrawal(address indexed account, uint256 amount);

    // ✅ 正确：遵循CEI模式
    function withdraw(uint256 amount) public {
        // 1. Checks（检查）：验证所有条件
        require(amount > 0, "金额必须大于0");
        require(balances[msg.sender] >= amount, "余额不足");

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

        // 3. Interactions（交互）：外部调用和事件
        emit Withdrawal(msg.sender, amount);
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "转账失败");
    }
```

```
    // ❌ 危险：外部调用在状态更新之前
    function badWithdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "余额不足");

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

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

5.7 提供错误恢复机制

在可能的情况下,提供错误恢复或回退方案。

```
contract ErrorRecovery {
    address public primaryOracle;
    address public fallbackOracle;

    interface IOracle {
        function getPrice() external view returns (uint256);
    }

    // ✅ 好：有回退方案
    function getPriceWithFallback() public view returns (uint256) {
        // 首先尝试主要预言机
        try IOracle(primaryOracle).getPrice() returns (uint256 price) {
            return price;
        } catch {
            // 如果主要预言机失败,尝试备用预言机
            try IOracle(fallbackOracle).getPrice() returns (uint256 price) {
                return price;
            } catch {
                // 都失败了,返回默认值或revert
                revert("无法获取价格");
            }
        }
    }

    // 批量操作中的部分失败处理
    mapping(address => uint256) public balances;

    event TransferSuccess(address to, uint256 amount);
    event TransferFailed(address to, uint256 amount, string reason);

    function batchTransfer(
        address[] memory recipients,
        uint256[] memory amounts
    ) public returns (uint256 successCount) {
```

```
        require(recipients.length == amounts.length, "数组长度不匹配");

        for (uint256 i = 0; i < recipients.length; i++) {
            if (recipients[i] == address(0)) {
                emit TransferFailed(recipients[i], amounts[i], "无效地址");
                continue;  // 跳过这个,继续处理其他
            }

            if (balances[msg.sender] < amounts[i]) {
                emit TransferFailed(recipients[i], amounts[i], "余额不足");
                continue;
            }

            balances[msg.sender] -= amounts[i];
            balances[recipients[i]] += amounts[i];
            emit TransferSuccess(recipients[i], amounts[i]);
            successCount++;
        }

        return successCount;
    }
}
```

5.8 避免错误消息过长

过长的错误消息会增加Gas消耗。

```
contract MessageLength {
    uint256 public value;

    // ❌ 不好：错误消息过长
    function badSet(uint256 newValue) public {
        require(
            newValue < 100,
            "The value you provided is too large. The maximum allowed value is 99. Please provide a smaller value and try again. For more information, please refer to the documentation."
        );
        value = newValue;
    }

    // ✅ 好：简洁但清晰的错误消息
    function goodSet(uint256 newValue) public {
        require(newValue < 100, "值必须小于100");
        value = newValue;
    }

    // ✅ 更好：使用自定义错误
    error ValueTooLarge(uint256 provided, uint256 maximum);

    function bestSet(uint256 newValue) public {
```

```
        if (newValue >= 100) {
            revert ValueTooLarge(newValue, 99);
        }
        value = newValue;
    }
}
```

5.9 文档化错误条件

使用NatSpec注释说明函数可能抛出的错误。

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

contract DocumentedErrors {
    error InsufficientBalance(uint256 available, uint256 required);
    error InvalidRecipient(address recipient);
    error TransferPaused();

    mapping(address => uint256) public balances;
    bool public paused;

    /**
     * @notice 转账代币
     * @param to 接收方地址
     * @param amount 转账金额
     * @dev 抛出以下错误：
     *      - TransferPaused: 如果转账功能被暂停
     *      - InvalidRecipient: 如果接收方地址为零地址
     *      - InsufficientBalance: 如果发送方余额不足
     */
    function transfer(address to, uint256 amount) public {
        if (paused) revert TransferPaused();
        if (to == address(0)) revert InvalidRecipient(to);
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(balances[msg.sender], amount);
        }

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

5.10 测试错误处理

确保测试覆盖所有错误情况。

```
// 测试合约示例（使用Hardhat或Truffle）
contract ErrorHandlingTest {
    TokenWithErrors public token;
```

```
    function setUp() public {
        token = new TokenWithErrors();
    }

    // 测试正常情况
    function testTransferSuccess() public {
        token.transfer(address(0x1), 100);
        // 验证余额变化
    }

    // 测试错误情况
    function testTransferInsufficientBalance() public {
        // 期望revert
        try token.transfer(address(0x1), 10000) {
            revert("应该失败但成功了");
        } catch Error(string memory reason) {
            // 验证错误消息
            require(
                keccak256(bytes(reason)) == keccak256(bytes("余额不足")),
                "错误消息不正确"
            );
        }
    }
}

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

    constructor() {
        balances[msg.sender] = 1000;
    }

    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "余额不足");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

5.11 完整的最佳实践示例

以下是一个综合应用所有最佳实践的完整示例：

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

/**
 * @title BestPracticeToken
 * @notice 展示错误处理最佳实践的代币合约
 */
```

```
contract BestPracticeToken {
    // ============ 自定义错误 ============

    /// @notice 余额不足
    error InsufficientBalance(address account, uint256 available, uint256 required);

    /// @notice 无效的接收地址
    error InvalidRecipient(address recipient);

    /// @notice 无效的金额
    error InvalidAmount(uint256 amount);

    /// @notice 未授权的操作
    error Unauthorized(address caller);

    /// @notice 转账已暂停
    error TransferPaused();

    // ============ 状态变量 ============

    mapping(address => uint256) public balanceOf;
    address public owner;
    bool public paused;

    // ============ 事件 ============

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Pause();
    event Unpause();

    // ============ 修饰符 ============

    modifier onlyOwner() {
        if (msg.sender != owner) revert Unauthorized(msg.sender);
        _;
    }

    modifier whenNotPaused() {
        if (paused) revert TransferPaused();
        _;
    }

    // ============ 构造函数 ============

    constructor() {
        owner = msg.sender;
        balanceOf[msg.sender] = 1000000;
    }

    // ============ 公共函数 ============

    /**
```

```
     * @notice 转账代币
     * @param to 接收方地址
     * @param amount 转账金额
     * @dev 可能抛出的错误：
     *      - TransferPaused: 转账被暂停
     *      - InvalidRecipient: 接收地址无效
     *      - InvalidAmount: 金额无效
     *      - InsufficientBalance: 余额不足
     */
    function transfer(address to, uint256 amount)
        public
        whenNotPaused
        returns (bool)
    {
        // 1. Checks: 输入验证（使用require或自定义错误）
        if (to == address(0)) revert InvalidRecipient(to);
        if (amount == 0) revert InvalidAmount(amount);
        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balanceOf[msg.sender], amount);
        }

        // 2. Effects: 状态更新
        uint256 senderBalanceBefore = balanceOf[msg.sender];
        uint256 recipientBalanceBefore = balanceOf[to];

        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;

        // 使用assert检查不变量
        assert(
            balanceOf[msg.sender] + balanceOf[to] ==
            senderBalanceBefore + recipientBalanceBefore
        );

        // 3. Interactions: 触发事件
        emit Transfer(msg.sender, to, amount);

        return true;
    }

    /**
     * @notice 暂停转账（仅所有者）
     */
    function pause() public onlyOwner {
        paused = true;
        emit Pause();
    }

    /**
     * @notice 恢复转账（仅所有者）
     */
    function unpause() public onlyOwner {
```

```
        paused = false;
        emit Unpause();
    }
}
```

---

6. 实际应用场景

现在让我们通过一些实际的应用场景,看看错误处理在真实项目中的应用。

6.1 代币合约

代币合约是最常见的智能合约类型,需要严格的错误处理来保障资金安全。

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

contract TokenContract {
    // 自定义错误
    error InsufficientBalance(address account, uint256 available, uint256 required);
    error InsufficientAllowance(address owner, address spender, uint256 available, uint256
required);
    error InvalidRecipient(address recipient);
    error InvalidAmount(uint256 amount);

    string public name = "MyToken";
    string public symbol = "MTK";
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

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

    constructor(uint256 _initialSupply) {
        totalSupply = _initialSupply;
        balanceOf[msg.sender] = _initialSupply;
    }

    /**
     * @notice 转账代币
     */
    function transfer(address to, uint256 amount) public returns (bool) {
        // 输入验证
        if (to == address(0)) revert InvalidRecipient(to);
        if (amount == 0) revert InvalidAmount(amount);

        // 状态检查
        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balanceOf[msg.sender], amount);
```

```
        }

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

        emit Transfer(msg.sender, to, amount);
        return true;
    }

    /**
     * @notice 授权第三方使用代币
     */
    function approve(address spender, uint256 amount) public returns (bool) {
        if (spender == address(0)) revert InvalidRecipient(spender);

        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    /**
     * @notice 从授权额度中转账
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public returns (bool) {
        // 输入验证
        if (to == address(0)) revert InvalidRecipient(to);
        if (amount == 0) revert InvalidAmount(amount);

        // 余额检查
        if (balanceOf[from] < amount) {
            revert InsufficientBalance(from, balanceOf[from], amount);
        }

        // 授权检查
        if (allowance[from][msg.sender] < amount) {
            revert InsufficientAllowance(
                from,
                msg.sender,
                allowance[from][msg.sender],
                amount
            );
        }

        // 执行转账
        balanceOf[from] -= amount;
        balanceOf[to] += amount;
        allowance[from][msg.sender] -= amount;
```

```
        emit Transfer(from, to, amount);
        return true;
    }
}
```

6.2 拍卖合约

拍卖合约需要处理复杂的业务逻辑和时间限制,错误处理至关重要。

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

contract AuctionContract {
    // 自定义错误
    error BidTooLow(uint256 currentBid, uint256 newBid);
    error AuctionEnded();
    error AuctionNotEnded();
    error NotHighestBidder();
    error WithdrawalFailed();
    error AlreadyWithdrawn();

    address public owner;
    uint256 public auctionEnd;
    uint256 public highestBid;
    address public highestBidder;

    mapping(address => uint256) public pendingReturns;
    mapping(address => bool) public hasWithdrawn;

    event NewBid(address indexed bidder, uint256 amount);
    event AuctionEnded(address winner, uint256 amount);
    event Withdrawal(address indexed bidder, uint256 amount);

    constructor(uint256 _duration) {
        owner = msg.sender;
        auctionEnd = block.timestamp + _duration;
    }

    /**
     * @notice 出价函数
     */
    function bid() public payable {
        // 检查拍卖是否还在进行
        if (block.timestamp >= auctionEnd) {
            revert AuctionEnded();
        }

        // 检查出价是否高于当前最高价
        if (msg.value <= highestBid) {
            revert BidTooLow(highestBid, msg.value);
```

```
        }

        // 使用assert检查时间不变量
        assert(auctionEnd > block.timestamp);

        // 如果有之前的最高出价者,记录待退款
        if (highestBidder != address(0)) {
            pendingReturns[highestBidder] += highestBid;
        }

        // 更新最高出价
        highestBidder = msg.sender;
        highestBid = msg.value;

        emit NewBid(msg.sender, msg.value);
    }

    /**
     * @notice 提取未中标的出价
     */
    function withdraw() public returns (bool) {
        // 检查是否有待退款
        uint256 amount = pendingReturns[msg.sender];
        require(amount > 0, "没有待退款");

        // 检查是否已经提取过
        if (hasWithdrawn[msg.sender]) {
            revert AlreadyWithdrawn();
        }

        // 先更新状态,防止重入攻击
        pendingReturns[msg.sender] = 0;
        hasWithdrawn[msg.sender] = true;

        // 使用try-catch处理转账
        (bool success, ) = msg.sender.call{value: amount}("");
        if (!success) {
            // 如果转账失败,恢复状态
            pendingReturns[msg.sender] = amount;
            hasWithdrawn[msg.sender] = false;
            revert WithdrawalFailed();
        }

        emit Withdrawal(msg.sender, amount);
        return true;
    }

    /**
     * @notice 结束拍卖（仅所有者）
     */
    function endAuction() public {
        require(msg.sender == owner, "只有所有者可以结束拍卖");
```

```
        if (block.timestamp < auctionEnd) {
            revert AuctionNotEnded();
        }

        emit AuctionEnded(highestBidder, highestBid);

        // 转账给所有者
        (bool success, ) = owner.call{value: highestBid}("");
        require(success, "转账失败");
    }
}
```

6.3 多签钱包

多签钱包需要处理多个签名者的协调和外部调用的异常。

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

contract MultiSigWallet {
    // 自定义错误
    error NotOwner();
    error TxNotExists(uint256 txId);
    error TxAlreadyExecuted(uint256 txId);
    error TxAlreadyConfirmed(uint256 txId);
    error TxNotConfirmed(uint256 txId);
    error CannotExecute(uint256 txId);
    error ExecutionFailed(uint256 txId);

    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;

    event Deposit(address indexed sender, uint256 amount);
    event Submit(uint256 indexed txId);
    event Confirm(address indexed owner, uint256 indexed txId);
    event Revoke(address indexed owner, uint256 indexed txId);
    event Execute(uint256 indexed txId);
    event ExecutionFailure(uint256 indexed txId, string reason);
```

```
    modifier onlyOwner() {
        if (!isOwner[msg.sender]) revert NotOwner();
        _;
    }

    modifier txExists(uint256 _txId) {
        if (_txId >= transactions.length) revert TxNotExists(_txId);
        _;
    }

    modifier notExecuted(uint256 _txId) {
        if (transactions[_txId].executed) revert TxAlreadyExecuted(_txId);
        _;
    }

    modifier notConfirmed(uint256 _txId) {
        if (confirmations[_txId][msg.sender]) revert TxAlreadyConfirmed(_txId);
        _;
    }

    constructor(address[] memory _owners, uint256 _required) {
        require(_owners.length > 0, "所有者不能为空");
        require(
            _required > 0 && _required <= _owners.length,
            "无效的所需确认数"
        );

        for (uint256 i = 0; i < _owners.length; i++) {
            address owner = _owners[i];
            require(owner != address(0), "无效的所有者地址");
            require(!isOwner[owner], "所有者重复");

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

        required = _required;
    }

    receive() external payable {
        emit Deposit(msg.sender, msg.value);
    }

    /**
     * @notice 提交交易
     */
    function submit(
        address _to,
        uint256 _value,
        bytes memory _data
    ) public 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
     */
    function confirm(uint256 _txId)
        public
        onlyOwner
        txExists(_txId)
        notExecuted(_txId)
        notConfirmed(_txId)
    {
        confirmations[_txId][msg.sender] = true;
        transactions[_txId].confirmations += 1;

        emit Confirm(msg.sender, _txId);
    }

    /**
     * @notice 执行交易（使用try-catch处理外部调用）
     */
    function execute(uint256 _txId)
        public
        onlyOwner
        txExists(_txId)
        notExecuted(_txId)
    {
        Transaction storage transaction = transactions[_txId];

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

        transaction.executed = true;

        // 使用try-catch处理外部调用
        (bool success, bytes memory returnData) = 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;

            // 提取失败原因
            string memory reason;
            if (returnData.length > 0) {
                assembly {
                    reason := mload(add(returnData, 0x20))
                }
            } else {
                reason = "Unknown error";
            }

            emit ExecutionFailure(_txId, reason);
            revert ExecutionFailed(_txId);
        }
    }

    /**
     * @notice 撤销确认
     */
    function revoke(uint256 _txId)
        public
        onlyOwner
        txExists(_txId)
        notExecuted(_txId)
    {
        if (!confirmations[_txId][msg.sender]) {
            revert TxNotConfirmed(_txId);
        }

        confirmations[_txId][msg.sender] = false;
        transactions[_txId].confirmations -= 1;

        emit Revoke(msg.sender, _txId);
    }
}
```

6.4 DeFi借贷协议

DeFi协议需要处理复杂的金融逻辑和多个外部合约调用。

```
// 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);
}

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

contract LendingProtocol {
    // 自定义错误
    error InsufficientCollateral(uint256 provided, uint256 required);
    error LoanNotFound(uint256 loanId);
    error Unauthorized();
    error TokenTransferFailed();
    error PriceOracleFailed();
    error CollateralRatioTooLow();

    struct Loan {
        address borrower;
        address collateralToken;
        address borrowToken;
        uint256 collateralAmount;
        uint256 borrowAmount;
        bool active;
    }

    mapping(uint256 => Loan) public loans;
    uint256 public loanCount;

    IPriceOracle public priceOracle;
    uint256 public constant COLLATERAL_RATIO = 150; // 150%

    event LoanCreated(uint256 indexed loanId, address indexed borrower);
    event LoanRepaid(uint256 indexed loanId);
    event CollateralLiquidated(uint256 indexed loanId);

    constructor(address _priceOracle) {
        priceOracle = IPriceOracle(_priceOracle);
    }

    /**
     * @notice 创建贷款
     */
    function createLoan(
        address collateralToken,
        address borrowToken,
        uint256 collateralAmount,
        uint256 borrowAmount
    ) public returns (uint256) {
        // 获取价格（使用try-catch处理预言机调用）
        uint256 collateralPrice;
```

```
        uint256 borrowPrice;

        try priceOracle.getPrice(collateralToken) returns (uint256 price) {
            collateralPrice = price;
        } catch {
            revert PriceOracleFailed();
        }

        try priceOracle.getPrice(borrowToken) returns (uint256 price) {
            borrowPrice = price;
        } catch {
            revert PriceOracleFailed();
        }

        // 计算抵押价值
        uint256 collateralValue = collateralAmount * collateralPrice;
        uint256 borrowValue = borrowAmount * borrowPrice;
        uint256 requiredCollateral = (borrowValue * COLLATERAL_RATIO) / 100;

        // 检查抵押率
        if (collateralValue < requiredCollateral) {
            revert InsufficientCollateral(collateralValue, requiredCollateral);
        }

        // 转移抵押物（使用try-catch）
        try IERC20(collateralToken).transferFrom(
            msg.sender,
            address(this),
            collateralAmount
        ) returns (bool success) {
            if (!success) revert TokenTransferFailed();
        } catch {
            revert TokenTransferFailed();
        }

        // 创建贷款
        uint256 loanId = loanCount++;
        loans[loanId] = Loan({
            borrower: msg.sender,
            collateralToken: collateralToken,
            borrowToken: borrowToken,
            collateralAmount: collateralAmount,
            borrowAmount: borrowAmount,
            active: true
        });

        // 发放借款
        try IERC20(borrowToken).transfer(msg.sender, borrowAmount) returns (bool success)
{
            if (!success) {
                // 如果借款发放失败,退还抵押物
                IERC20(collateralToken).transfer(msg.sender, collateralAmount);
```

```
                revert TokenTransferFailed();
            }
        } catch {
            // 如果借款发放失败,退还抵押物
            IERC20(collateralToken).transfer(msg.sender, collateralAmount);
            revert TokenTransferFailed();
        }

        emit LoanCreated(loanId, msg.sender);
        return loanId;
    }

    /**
     * @notice 偿还贷款
     */
    function repayLoan(uint256 loanId) public {
        Loan storage loan = loans[loanId];

        // 检查贷款是否存在且活跃
        if (!loan.active) revert LoanNotFound(loanId);
        if (loan.borrower != msg.sender) revert Unauthorized();

        // 收回借款
        try IERC20(loan.borrowToken).transferFrom(
            msg.sender,
            address(this),
            loan.borrowAmount
        ) returns (bool success) {
            if (!success) revert TokenTransferFailed();
        } catch {
            revert TokenTransferFailed();
        }

        // 返还抵押物
        try IERC20(loan.collateralToken).transfer(
            loan.borrower,
            loan.collateralAmount
        ) returns (bool success) {
            if (!success) {
                // 如果返还失败,退回借款
                IERC20(loan.borrowToken).transfer(msg.sender, loan.borrowAmount);
                revert TokenTransferFailed();
            }
        } catch {
            // 如果返还失败,退回借款
            IERC20(loan.borrowToken).transfer(msg.sender, loan.borrowAmount);
            revert TokenTransferFailed();
        }

        loan.active = false;
        emit LoanRepaid(loanId);
    }
```

```
}
```

7. 常见错误与注意事项

在使用错误处理机制时,开发者经常会遇到一些陷阱。了解这些常见错误可以帮助你避免类似问题。

7.1 忘记检查条件

这是最常见也是最危险的错误之一。

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

    // ❌ 危险：没有检查余额就执行转账
    function badTransfer(address to, uint256 amount) public {
        // 直接执行转移,没有检查余额
        balances[msg.sender] -= amount;  // 可能导致整数下溢
        balances[to] += amount;
    }

    // ✅ 正确：先检查余额
    function goodTransfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "余额不足");
        require(to != address(0), "无效地址");

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

实际案例:
某些早期的ERC20代币合约由于没有充分检查余额,导致用户可以转账超过自己余额的代币,造成严重的安全问题。

7.2 使用assert而非require

在应该使用require的地方错误地使用assert。

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

    // ❌ 错误：用assert检查用户输入
    function badWithdraw(uint256 amount) public {
        assert(balances[msg.sender] >= amount);  // 错误：消耗全部Gas

        balances[msg.sender] -= amount;
    }

    // ✅ 正确：用require检查用户输入
    function goodWithdraw(uint256 amount) public {
```

```
        require(balances[msg.sender] >= amount, "余额不足");  // 正确：只消耗部分Gas

        balances[msg.sender] -= amount;
    }

    // ✅ 正确：用assert检查不变量
    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "余额不足");

        uint256 totalBefore = balances[msg.sender] + balances[to];

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

        // 检查不变量：总和应该保持不变
        assert(balances[msg.sender] + balances[to] == totalBefore);
    }
}
```

Gas对比：

- require失败：退还未使用的Gas
- assert失败：消耗全部Gas（可能是用户余额的全部）

7.3 错误消息不明确

不清晰的错误消息会让调试变得困难。

```
contract UnclearErrors {
    mapping(address => uint256) public balances;
    uint256 public maxTransfer = 1000;

    // ❌ 不好：错误消息不明确
    function badTransfer(address to, uint256 amount) public {
        require(to != address(0));  // 没有消息
        require(amount > 0);  // 没有消息
        require(balances[msg.sender] >= amount, "Error");  // 太笼统
        require(amount <= maxTransfer, "Failed");  // 不够具体

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

    // ✅ 好：清晰的错误消息
    function goodTransfer(address to, uint256 amount) public {
        require(to != address(0), "接收地址不能为零地址");
        require(amount > 0, "转账金额必须大于0");
        require(balances[msg.sender] >= amount, "余额不足");
        require(amount <= maxTransfer, "转账金额超过限制");

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

```
    }

    // ✅ 更好：使用自定义错误
    error InvalidRecipient(address recipient);
    error InvalidAmount(uint256 amount);
    error InsufficientBalance(uint256 available, uint256 required);
    error ExceedsMaxTransfer(uint256 amount, uint256 maximum);

    function bestTransfer(address to, uint256 amount) public {
        if (to == address(0)) revert InvalidRecipient(to);
        if (amount == 0) revert InvalidAmount(amount);
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(balances[msg.sender], amount);
        }
        if (amount > maxTransfer) {
            revert ExceedsMaxTransfer(amount, maxTransfer);
        }

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

7.4 不恰当的try-catch使用

try-catch有特定的使用场景,不能滥用。

```
contract ImproperTryCatch {
    // ❌ 错误：try-catch不能用于当前合约的内部函数
    function badInternalCall() public {
        // try this.internalFunction() {  // 编译错误
        //     // ...
        // } catch {
        //     // ...
        // }
    }

    function internalFunction() internal pure returns (uint256) {
        return 42;
    }

    // ❌ 错误：try-catch不能用于public函数的内部调用
    function badPublicCall() public {
        // try this.publicFunction() {  // 虽然能编译,但会失败
        //     // ...
        // } catch {
        //     // ...
        // }
    }

    function publicFunction() public pure returns (uint256) {
```

```
        return 42;
    }

    // ✅ 正确：try-catch用于外部合约调用
    interface IExternal {
        function doSomething() external returns (bool);
    }

    IExternal public externalContract;

    function goodExternalCall() public {
        try externalContract.doSomething() returns (bool success) {
            // 处理成功
        } catch {
            // 处理失败
        }
    }
}
```

7.5 忽略函数返回值

某些函数返回bool值表示成功或失败,不检查返回值可能导致问题。

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

contract ReturnValueHandling {
    IERC20 public token;

    // ❌ 危险：忽略返回值
    function badTransfer(address to, uint256 amount) public {
        token.transfer(to, amount);  // 如果失败了呢?
        // 代码继续执行,好像转账成功了
    }

    // ✅ 正确：检查返回值
    function goodTransfer(address to, uint256 amount) public {
        bool success = token.transfer(to, amount);
        require(success, "代币转账失败");
    }

    // ✅ 更好：使用try-catch
    function bestTransfer(address to, uint256 amount) public {
        try token.transfer(to, amount) returns (bool success) {
            require(success, "代币转账返回false");
        } catch {
            revert("代币转账失败");
        }
    }
}
```

7.6 重入攻击漏洞

不正确的错误处理顺序可能导致重入攻击。

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

    // ❌ 危险：外部调用在状态更新之前
    function vulnerableWithdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "余额不足");

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

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

    // ✅ 安全：遵循CEI模式
    function safeWithdraw(uint256 amount) public {
        // 1. Checks
        require(balances[msg.sender] >= amount, "余额不足");

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

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

7.7 Gas限制问题

过长的错误消息或复杂的catch块可能导致Gas不足。

```
contract GasLimitIssues {
    // ❌ 不好：错误消息过长
    function badError() public pure {
        require(
            false,
            "This is an extremely long error message that contains way too much
information and will consume a lot of gas. You should avoid writing such long error
messages because they increase the transaction cost significantly and provide diminishing
returns in terms of usefulness."
        );
    }

    // ✅ 好：简洁的错误消息
    function goodError() public pure {
```

```
        require(false, "操作失败");
    }

    // ✅ 更好：使用自定义错误
    error OperationFailed(uint256 code);

    function bestError() public pure {
        revert OperationFailed(1);
    }

    // ❌ 不好：catch块中有复杂逻辑
    interface IExternal {
        function doSomething() external;
    }

    IExternal public externalContract;

    function badCatch() public {
        try externalContract.doSomething() {
            // ...
        } catch {
            // 复杂的循环可能超出Gas限制
            for (uint256 i = 0; i < 1000; i++) {
                // 大量计算...
            }
        }
    }

    // ✅ 好：catch块保持简单
    event ErrorOccurred();

    function goodCatch() public {
        try externalContract.doSomething() {
            // ...
        } catch {
            emit ErrorOccurred();
            // 简单的错误处理
        }
    }
}
```

7.8 注意事项总结

1. 始终检查输入：

- 验证所有外部输入
- 检查地址是否为零地址
- 检查数值是否在有效范围内
- 检查数组长度是否匹配

2. 选择正确的错误机制：

- require：用于输入验证和条件检查
- assert：用于不变量检查
- revert：用于复杂的错误处理
- 自定义错误：优先使用以节省Gas

3. 遵循Checks-Effects-Interactions模式：

- Checks：检查所有条件
- Effects：更新状态
- Interactions：进行外部调用

4. 使用try-catch处理外部调用：

- 只用于外部合约调用
- 不要嵌套过深
- 保持catch块简单

5. 提供清晰的错误信息：

- 使用描述性的错误消息
- 避免过长的消息
- 考虑使用自定义错误

6. 测试所有错误路径：

- 测试正常情况
- 测试所有错误情况
- 测试边界条件

---

8. Gas消耗对比分析

理解不同错误处理机制的Gas消耗,有助于做出更优的设计选择。

8.1 错误机制Gas对比

以下是不同错误处理方式的Gas消耗对比：

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

contract GasComparison {
    uint256 public value = 100;

    // 自定义错误定义
    error ValueTooHigh(uint256 current, uint256 maximum);
    error ValueTooLow(uint256 current, uint256 minimum);

    // 1. require + 短字符串
    function testRequireShortString(uint256 newValue) public {
        require(newValue <= 200, "Too high");
        value = newValue;
    }
```

```
    // 2. require + 长字符串
    function testRequireLongString(uint256 newValue) public {
        require(
            newValue <= 200,
            "The value you provided is too high and exceeds the maximum allowed"
        );
        value = newValue;
    }

    // 3. require + 自定义错误
    function testRequireCustomError(uint256 newValue) public {
        if (newValue > 200) revert ValueTooHigh(newValue, 200);
        value = newValue;
    }

    // 4. assert
    function testAssert(uint256 newValue) public {
        value = newValue;
        assert(value <= 1000);  // 如果失败会消耗所有Gas
    }

    // 5. revert + 字符串
    function testRevertString(uint256 newValue) public {
        if (newValue > 200) revert("Value too high");
        value = newValue;
    }

    // 6. revert + 自定义错误
    function testRevertCustomError(uint256 newValue) public {
        if (newValue > 200) revert ValueTooHigh(newValue, 200);
        value = newValue;
    }
}
```

Gas消耗数据（失败时）：

错误机制	Gas消耗	节省比例
require + 长字符串	~24,500 gas	基准
require + 短字符串	~23,800 gas	2.9%
revert + 字符串	~23,900 gas	2.4%
require + 自定义错误	~21,200 gas	13.5%
revert + 自定义错误	~21,300 gas	13.1%
assert（消耗全部）	全部Gas	-

关键发现：

1. **自定义错误比字符串错误节省约13-15%的Gas**
2. 字符串越长,Gas消耗越高
3. assert失败时消耗全部Gas,应该避免用于输入验证
4. require和revert配合自定义错误的Gas消耗相近

8.2 成功执行时的Gas对比

当条件满足,函数成功执行时的Gas消耗:

```
contract SuccessGasComparison {
    uint256 public value;

    error ValueOutOfRange(uint256 value);

    // require版本
    function setWithRequire(uint256 newValue) public {
        require(newValue < 100, "Value too high");
        value = newValue;
    }

    // 自定义错误版本
    function setWithCustomError(uint256 newValue) public {
        if (newValue >= 100) revert ValueOutOfRange(newValue);
        value = newValue;
    }

    // 无检查版本（不推荐）
    function setWithoutCheck(uint256 newValue) public {
        value = newValue;
    }
}
```

Gas消耗数据（成功时）：

版本	Gas消耗	差异
无检查	~43,400 gas	基准
require + 字符串	~43,650 gas	+0.6%
自定义错误	~43,580 gas	+0.4%

结论:

- 成功执行时,不同错误处理方式的Gas差异很小（<1%）
- 主要差异体现在失败时的Gas消耗
- 错误检查的额外成本是值得的（安全性换来的代价很小）

8.3 实际应用中的Gas优化建议

1. 使用自定义错误替代字符串

```
contract GasOptimizedToken {
    // ✅ 推荐：使用自定义错误
    error InsufficientBalance(uint256 available, uint256 required);

    mapping(address => uint256) public balanceOf;

    function transfer(address to, uint256 amount) public {
        if (balanceOf[msg.sender] < amount) {
            revert InsufficientBalance(balanceOf[msg.sender], amount);
        }
        // 每次失败节省约3,000 gas
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
    }
}
```

2. 合并相似的检查

```
contract CheckOptimization {
    // ❌ 不够优化：多个独立检查
    function badTransfer(address to, uint256 amount) public {
        require(to != address(0), "Invalid recipient");
        require(amount > 0, "Invalid amount");
        require(amount <= 1000, "Amount too high");
        // 每个require都有基础开销
    }

    // ✅ 更优：合并检查
    function goodTransfer(address to, uint256 amount) public {
        require(
            to != address(0) && amount > 0 && amount <= 1000,
            "Invalid parameters"
        );
        // 单个require,减少基础开销
    }

    // ✅ 最优：使用自定义错误分别处理
    error InvalidRecipient();
    error InvalidAmount();

    function bestTransfer(address to, uint256 amount) public {
        if (to == address(0)) revert InvalidRecipient();
        if (amount == 0 || amount > 1000) revert InvalidAmount();
        // 清晰且节省Gas
    }
}
```

3. 早期退出以节省Gas

```
contract EarlyExit {
```

```
    mapping(address => uint256) public balances;
    mapping(address => bool) public blacklist;

    // ✅ 好：先检查简单条件
    function transfer(address to, uint256 amount) public {
        // 1. 先检查最简单的条件（消耗最少Gas）
        if (to == address(0)) revert("Invalid recipient");
        if (amount == 0) revert("Invalid amount");

        // 2. 再检查状态读取相关的条件（消耗中等Gas）
        if (blacklist[msg.sender]) revert("Sender blacklisted");
        if (blacklist[to]) revert("Recipient blacklisted");

        // 3. 最后检查最复杂的条件（消耗最多Gas）
        if (balances[msg.sender] < amount) revert("Insufficient balance");

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

8.4 批量操作的错误处理

在批量操作中,错误处理策略会显著影响Gas消耗。

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

    error TransferFailed(uint256 index, address recipient);

    event TransferSuccess(address indexed to, uint256 amount);
    event TransferSkipped(address indexed to, uint256 amount, string reason);

    // 策略1：一个失败全部失败
    function batchTransferStrictMode(
        address[] memory recipients,
        uint256[] memory amounts
    ) public {
        require(recipients.length == amounts.length, "Length mismatch");

        // 一次性检查总金额
        uint256 totalAmount = 0;
        for (uint256 i = 0; i < amounts.length; i++) {
            totalAmount += amounts[i];
        }
        require(balances[msg.sender] >= totalAmount, "Insufficient balance");

        // 执行所有转账
        for (uint256 i = 0; i < recipients.length; i++) {
            balances[msg.sender] -= amounts[i];
```

```
            balances[recipients[i]] += amounts[i];
            emit TransferSuccess(recipients[i], amounts[i]);
        }
    }

    // 策略2：单个失败不影响其他
    function batchTransferLenientMode(
        address[] memory recipients,
        uint256[] memory amounts
    ) public returns (uint256 successCount) {
        require(recipients.length == amounts.length, "Length mismatch");

        for (uint256 i = 0; i < recipients.length; i++) {
            // 检查单个转账
            if (recipients[i] == address(0)) {
                emit TransferSkipped(recipients[i], amounts[i], "Invalid recipient");
                continue;
            }

            if (balances[msg.sender] < amounts[i]) {
                emit TransferSkipped(recipients[i], amounts[i], "Insufficient balance");
                continue;
            }

            // 执行转账
            balances[msg.sender] -= amounts[i];
            balances[recipients[i]] += amounts[i];
            emit TransferSuccess(recipients[i], amounts[i]);
            successCount++;
        }

        return successCount;
    }
}
```

Gas分析：

- **严格模式**：Gas更低（一次性检查）,但一个失败全部回滚
- **宽松模式**：Gas稍高（多次检查）,但部分成功更灵活

根据业务需求选择合适的策略。

---

9. 实践练习

通过实践练习来巩固错误处理的知识。

9.1 练习1：基础错误处理（⭐⭐难度）

任务：创建一个简单的银行合约,实现存款、取款功能,并使用适当的错误处理。

要求：

1. 使用require进行输入验证
2. 使用自定义错误提供详细的错误信息
3. 确保在余额不足时正确抛出异常

参考答案:

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

contract SimpleBank {
    // 自定义错误
    error InsufficientBalance(address account, uint256 available, uint256 required);
    error InvalidAmount(uint256 amount);
    error WithdrawalFailed();

    mapping(address => uint256) public balances;

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

    /**
     * @notice 存款
     */
    function deposit() public payable {
        require(msg.value > 0, "存款金额必须大于0");

        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    /**
     * @notice 取款
     */
    function withdraw(uint256 amount) public {
        // 输入验证
        if (amount == 0) revert InvalidAmount(amount);

        // 余额检查
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(msg.sender, balances[msg.sender], amount);
        }

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

        // 转账
        (bool success, ) = msg.sender.call{value: amount}("");
        if (!success) {
            // 如果转账失败,恢复余额
            balances[msg.sender] += amount;
            revert WithdrawalFailed();
        }
```

```
        emit Withdrawal(msg.sender, amount);
    }

    /**
     * @notice 查询余额
     */
    function getBalance() public view returns (uint256) {
        return balances[msg.sender];
    }
}
```

9.2 练习2：try-catch实践（⭐⭐⭐难度）

任务：创建一个代币交换合约,使用try-catch处理外部ERC20代币转账可能出现的异常。

要求：

1. 使用try-catch捕获代币转账错误
2. 区分不同类型的错误（字符串错误、Panic、自定义错误）
3. 实现失败回滚机制

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);
}

contract TokenSwap {
    event SwapSuccess(address indexed user, address tokenIn, address tokenOut, uint256
amountIn, uint256 amountOut);
    event SwapFailed(address indexed user, string reason);
    event SwapFailedPanic(address indexed user, uint256 errorCode);
    event SwapFailedCustom(address indexed user, bytes errorData);

    /**
     * @notice 交换代币
     */
    function swap(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 amountOut
    ) public returns (bool) {
        // 步骤1：从用户转入tokenIn
        try IERC20(tokenIn).transferFrom(msg.sender, address(this), amountIn) returns
(bool success) {
```

```
            if (!success) {
                emit SwapFailed(msg.sender, "TransferFrom returned false");
                return false;
            }

            // 步骤2：向用户转出tokenOut
            try IERC20(tokenOut).transfer(msg.sender, amountOut) returns (bool success2) {
                if (!success2) {
                    // tokenOut转账失败,退还tokenIn
                    IERC20(tokenIn).transfer(msg.sender, amountIn);
                    emit SwapFailed(msg.sender, "Transfer returned false");
                    return false;
                }

                emit SwapSuccess(msg.sender, tokenIn, tokenOut, amountIn, amountOut);
                return true;

            } catch Error(string memory reason) {
                // tokenOut转账失败,退还tokenIn
                IERC20(tokenIn).transfer(msg.sender, amountIn);
                emit SwapFailed(msg.sender, reason);
                return false;
            } catch Panic(uint errorCode) {
                // tokenOut转账Panic,退还tokenIn
                IERC20(tokenIn).transfer(msg.sender, amountIn);
                emit SwapFailedPanic(msg.sender, errorCode);
                return false;
            } catch (bytes memory errorData) {
                // tokenOut转账其他错误,退还tokenIn
                IERC20(tokenIn).transfer(msg.sender, amountIn);
                emit SwapFailedCustom(msg.sender, errorData);
                return false;
            }

        } catch Error(string memory reason) {
            emit SwapFailed(msg.sender, reason);
            return false;
        } catch Panic(uint errorCode) {
            emit SwapFailedPanic(msg.sender, errorCode);
            return false;
        } catch (bytes memory errorData) {
            emit SwapFailedCustom(msg.sender, errorData);
            return false;
        }
    }
}
```

9.3 练习3：自定义错误优化（⭐⭐⭐难度）

任务：将练习1中的字符串错误全部替换为自定义错误,比较Gas消耗差异。

要求：

1. 定义完整的自定义错误类型
2. 替换所有字符串错误
3. 在Remix中测试并比较Gas消耗

参考答案：

见练习1的答案（已经使用自定义错误）

Gas对比测试步骤：

1. 创建两个版本的合约：一个使用字符串错误,一个使用自定义错误
2. 在Remix中部署两个合约
3. 分别调用会失败的函数（如取款超过余额）
4. 在控制台查看Gas消耗
5. 记录并比较差异

9.4 练习4：完整的DApp错误处理（⭐⭐⭐⭐难度）

任务：设计一个众筹合约,实现完整的错误处理系统。

要求：

1. 使用require进行输入验证
2. 使用自定义错误提供详细信息
3. 使用try-catch处理退款过程中的异常
4. 遵循Checks-Effects-Interactions模式
5. 处理所有可能的边界情况

参考答案：

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

contract Crowdfunding {
    // ============ 自定义错误 ============

    error CampaignNotActive();
    error CampaignAlreadyEnded();
    error GoalNotReached(uint256 current, uint256 goal);
    error GoalAlreadyReached();
    error InvalidAmount(uint256 amount);
    error Unauthorized(address caller);
    error RefundFailed(address contributor);
    error WithdrawalFailed();
    error AlreadyRefunded(address contributor);

    // ============ 状态变量 ============

    address public owner;
    uint256 public goal;
    uint256 public deadline;
    uint256 public totalRaised;
    bool public ended;
```

```
    bool public goalReached;

    mapping(address => uint256) public contributions;
    mapping(address => bool) public refunded;

    // ============ 事件 ============

    event Contribution(address indexed contributor, uint256 amount);
    event GoalReached(uint256 totalAmount);
    event Withdrawal(address indexed owner, uint256 amount);
    event Refund(address indexed contributor, uint256 amount);
    event RefundFailed(address indexed contributor, uint256 amount, string reason);

    // ============ 修饰符 ============

    modifier onlyOwner() {
        if (msg.sender != owner) revert Unauthorized(msg.sender);
        _;
    }

    modifier campaignActive() {
        if (ended) revert CampaignAlreadyEnded();
        if (block.timestamp >= deadline) revert CampaignNotActive();
        _;
    }

    // ============ 构造函数 ============

    constructor(uint256 _goal, uint256 _duration) {
        require(_goal > 0, "目标金额必须大于0");
        require(_duration > 0, "持续时间必须大于0");

        owner = msg.sender;
        goal = _goal;
        deadline = block.timestamp + _duration;
    }

    // ============ 公共函数 ============

    /**
     * @notice 贡献资金
     */
    function contribute() public payable campaignActive {
        // 1. Checks: 输入验证
        if (msg.value == 0) revert InvalidAmount(msg.value);

        // 2. Effects: 更新状态
        contributions[msg.sender] += msg.value;
        totalRaised += msg.value;

        // 检查是否达到目标
        if (!goalReached && totalRaised >= goal) {
```

```
            goalReached = true;
            emit GoalReached(totalRaised);
        }

        // 3. Interactions: 触发事件
        emit Contribution(msg.sender, msg.value);
    }

    /**
     * @notice 结束众筹
     */
    function endCampaign() public {
        // 检查时间
        require(block.timestamp >= deadline, "众筹尚未结束");
        require(!ended, "众筹已经结束");

        ended = true;

        if (totalRaised >= goal) {
            goalReached = true;
        }
    }

    /**
     * @notice 提取资金（仅所有者，目标达成后）
     */
    function withdraw() public onlyOwner {
        // 1. Checks
        require(ended, "众筹尚未结束");
        if (!goalReached) revert GoalNotReached(totalRaised, goal);

        uint256 amount = address(this).balance;
        require(amount > 0, "没有可提取的资金");

        // 2. Effects: 清空余额（防止重入）
        // 注意：这里简化处理，实际应该更细致

        // 3. Interactions: 转账
        (bool success, ) = owner.call{value: amount}("");
        if (!success) revert WithdrawalFailed();

        emit Withdrawal(owner, amount);
    }

    /**
     * @notice 退款（目标未达成时）
     */
    function refund() public {
        // 1. Checks
        require(ended, "众筹尚未结束");
        if (goalReached) revert GoalAlreadyReached();
```

```
        uint256 amount = contributions[msg.sender];
        require(amount > 0, "没有贡献");

        if (refunded[msg.sender]) revert AlreadyRefunded(msg.sender);

        // 2. Effects: 更新状态
        contributions[msg.sender] = 0;
        refunded[msg.sender] = true;

        // 3. Interactions: 退款
        (bool success, ) = msg.sender.call{value: amount}("");
        if (!success) {
            // 如果退款失败，恢复状态
            contributions[msg.sender] = amount;
            refunded[msg.sender] = false;
            revert RefundFailed(msg.sender);
        }

        emit Refund(msg.sender, amount);
    }

    /**
     * @notice 批量退款（仅所有者，用于紧急情况）
     */
    function batchRefund(address[] memory contributors) public onlyOwner {
        require(ended, "众筹尚未结束");
        require(!goalReached, "目标已达成，不能退款");

        for (uint256 i = 0; i < contributors.length; i++) {
            address contributor = contributors[i];
            uint256 amount = contributions[contributor];

            if (amount == 0 || refunded[contributor]) {
                continue;  // 跳过已退款或没有贡献的地址
            }

            // 更新状态
            contributions[contributor] = 0;
            refunded[contributor] = true;

            // 尝试退款，使用try-catch处理异常
            (bool success, ) = contributor.call{value: amount}("");

            if (success) {
                emit Refund(contributor, amount);
            } else {
                // 退款失败，恢复状态并记录
                contributions[contributor] = amount;
                refunded[contributor] = false;
                emit RefundFailed(contributor, amount, "Transfer failed");
            }
        }
```

```
    }

    /**
     * @notice 查询剩余时间
     */
    function timeLeft() public view returns (uint256) {
        if (block.timestamp >= deadline) {
            return 0;
        }
        return deadline - block.timestamp;
    }
}
```

---

10. 学习检查清单

完成本课后，你应该能够：

基础概念：

- [ ] 理解错误处理的重要性
- [ ] 知道交易回滚的工作原理
- [ ] 理解错误传播机制

三种错误机制：

- [ ] 理解require的用途和特点
- [ ] 理解assert的用途和特点
- [ ] 理解revert的用途和特点
- [ ] 能够根据场景选择合适的错误机制
- [ ] 知道三种机制的Gas消耗差异

自定义错误：

- [ ] 会定义自定义错误
- [ ] 理解自定义错误的三大优势
- [ ] 能够为错误添加适当的参数
- [ ] 知道如何组织错误层次结构
- [ ] 理解自定义错误的Gas优势

try-catch：

- [ ] 理解try-catch的语法结构
- [ ] 知道try-catch只能用于外部调用
- [ ] 会使用不同的catch子句
- [ ] 能够在实际项目中正确使用try-catch
- [ ] 理解try-catch的使用限制

最佳实践:

- [ ] 会使用有意义的错误消息
- [ ] 优先使用require进行输入验证
- [ ] 会使用assert验证不变量
- [ ] 优先使用自定义错误
- [ ] 合理使用try-catch
- [ ] 遵循Checks-Effects-Interactions模式
- [ ] 会提供错误恢复机制
- [ ] 会避免错误消息过长
- [ ] 会文档化错误条件
- [ ] 会测试错误处理

实践能力:

- [ ] 能够在代币合约中实现错误处理
- [ ] 能够在复杂业务逻辑中使用错误处理
- [ ] 能够使用try-catch处理外部调用
- [ ] 能够优化Gas消耗

常见错误:

- [ ] 知道如何避免忘记检查条件
- [ ] 知道何时不应使用assert
- [ ] 会编写清晰的错误消息
- [ ] 理解try-catch的正确使用场景
- [ ] 会处理函数返回值
- [ ] 能够防止重入攻击

---

11. 总结

错误处理是智能合约开发中的核心技能。通过本课的学习,你应该已经掌握了:

1. **三种错误处理机制**:
   - require用于输入验证
   - assert用于不变量检查
   - revert用于灵活的错误处理
2. **自定义错误的优势**:
   - Gas优化（节省约13-15%）
   - 代码可重用性
   - 更好的错误识别能力
3. **try-catch的应用**:

    - 只用于外部合约调用
    - 区分不同类型的错误
    - 实现优雅的错误处理
4. **最佳实践**：
    - 遵循CEI模式
    - 使用自定义错误
    - 提供清晰的错误信息
    - 测试所有错误路径