require(msg.sender == owner);

try {

assert(balance > amount);

Solidity错误处理和自定义错误

contract.myFunction();

revert("Insufficient balance");

第7.2课：从入门到精通

error Unauthorized();

} catch Error(string memory reason) {

Error: 本课程深入讲解Solidity智能合约中的错误处理机制
建议: 确保阅读并理解所有示例代码
Info: 包含代码示例、最佳实践和实际应用场景

讲师：【Layer】

课程大纲

require/assert/revert详解
三种错误处理机制的使用、对比和选择

自定义错误 (0.8.4+)
定义、优势、使用场景和最佳实践

try-catch异常捕获
基本语法、使用场景、注意事项和应用

错误处理最佳实践
10个关键最佳实践及其实现示例

实际应用场景
代币合约、拍卖合约、多签钱包等示例

常见错误与注意事项
常见错误对比与最佳实践建议

require/assert/revert详解

require

用途： 检查输入参数和合约状态

特点： 交易可恢复，gas消耗较低

```
require(balance >= amount, "余额不足");
// 交易可恢复，发送者承担gas费用
```

assert

用途： 检查内部一致性 invariant

特点： 交易不可恢复，gas全部消耗

```
assert(balance == totalSupply);
// 交易不可恢复，发送者承担全部gas费用
```

revert

用途： 自定义错误条件和消息

特点： 交易可恢复，可指定错误消息

```
revert(InsufficientBalance());
// 交易可恢复，可自定义错误类型
```

对比表格

特性	require	assert	revert
交易状态	可恢复	不可恢复	可恢复
gas消耗	较低	全部消耗	中等
错误消息	字符串	无	自定义错误

最佳实践： 优先使用require进行输入验证，assert用于内部一致性检查，revert用于自定义错误条件。

自定义错误 (0.8.4+)

定义

在Solidity 0.8.4版本中引入的特性，允许开发者创建可重用的错误类型，提高代码效率和可读性。

主要优势

Gas优化
相比字符串错误，消耗更少的Gas，提高交易效率

可重用性
一次定义，多处使用，减少代码重复

可识别性
错误类型可被其他合约识别和处理

代码示例

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

// 使用自定义错误
function transfer(address to, uint256 amount) {
  if (balance[msg.sender] < amount)
    revert InsufficientBalance(balance[msg.sender], amount);
}
```

最佳实践

错误命名
使用PascalCase命名错误，清晰表达错误意图

错误参数
添加相关上下文参数，便于错误处理和调试

错误层次
组织相关错误为继承层次结构，提高代码结构化

安全考量
避免在错误中暴露敏感信息，如私钥或机密数据

try–catch异常捕获

语法结构

```
// 基本语法结构
try {
  // 外部调用代码
  externalContract.someFunction();
} catch (Error string reason) {
  // 处理字符串错误
} catch (bytes data) {
  // 处理未知错误
}
```

执行流程

使用场景

- 外部合约调用错误捕获
- ERC20/ERC721代币转账错误处理
- 条件判断前的安全检查

注意事项

- 避免嵌套过深的try–catch结构
- string类型错误比bytes类型更高效
- catch块中的gas消耗需谨慎考虑

实际应用

```
// 代币转账错误处理示例
function transferToken(address to, uint256 amount) public {
  try {
    IERC20(token).transfer(to, amount);
  } catch (Error string reason) {
    // 记录错误原因
    emit TransferFailed(reason);
    revert("Transfer failed");
  } catch (bytes data) {
    // 处理未知错误
    emit UnknownError();
    revert("Unknown error occurred");
  }
}
```

错误处理最佳实践

1. **使用有意义的错误消息**
   提供清晰的错误描述，便于调试和理解
2. **优先使用require**
   在函数开始处进行输入验证
3. **使用assert验证不变量**
   确保合约状态在逻辑上是一致的
4. **自定义错误优于字符串错误**
   节省gas消耗，提高合约效率
5. **合理使用try-catch**
   处理外部调用可能发生的错误

```
// 错误处理最佳实践示例
contract Token {
    error InsufficientBalance(uint available);
    error InvalidRecipient();

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

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

    function transfer(address to, uint amount) public {
        // 1. 输入验证
        require(to != address(0), "无效接收者");

        // 2. 状态检查
        require(balances[msg.sender] >= amount, "余额不足");

        // 3. 更新状态
        balances[msg.sender] -= amount;
        balances[to] += amount;

        // 4. 事件发出
        emit Transfer(msg.sender, to, amount);
    }

    function safeTransfer(address to, uint amount) public {
        // 5. 外部调用使用try-catch
        try IERC20(to).transfer(msg.sender, amount) {
            emit Transfer(msg.sender, to, amount);
        } catch {
            revert("安全转账失败");
        }
    }
}
```

实际应用场景

代币合约

使用require检查余额不足情况，自定义错误提高Gas效率

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

拍卖合约

使用assert验证前置条件，revert处理竞价无效情况

```
function bid(uint256 amount) public {
  require(amount > currentBid, "出价必须高于当前最高价");
  assert(auctionEnd > block.timestamp);
  currentBid = amount;
}
```

多签钱包

结合try-catch处理不同执行结果，确保交易安全性

```
function executeTransaction() public {
  try myContract.call(data) {
    // 成功处理
  } catch {
    // 失败处理
  }
}
```

关键要点

- 根据场景选择合适的错误处理机制
- 优先使用自定义错误提高Gas效率
- 结合场景需求设计错误处理流程

常见错误与注意事项

常见错误

1. 忘记检查条件

```
function transfer(address to, uint amount) {
    balance[msg.sender] -= amount;
    balance[to] += amount;
    // 错误：忘记检查余额是否足够
}
```

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

2. 使用assert而非require

```
function withdraw(uint amount) {
    assert(balance >= amount); // 错误：
assert不会释放gas
    balance -= amount;
}
```

```
function withdraw(uint amount) {
    require(balance >= amount, "余额不足
"); // 正确：require会释放gas
    balance -= amount;
}
```

3. 错误消息不明确

```
function approve(address spender, uint
amount) {
    require(amount > 0); // 错误：没有明确
的错误消息
    allowed[msg.sender][spender] =
amount;
}
```

```
function approve(address spender, uint
amount) {
    require(amount > 0, "金额必须大于0");
    allowed[msg.sender][spender] =
amount;
}
```

注意事项

使用自定义错误

自定义错误(0.8.4+)比字符串错误更节省gas，提高合约效率

错误处理与gas

错误触发时，require/revert会释放gas，assert不会

try-catch使用

外部调用时使用try-catch捕获异常，避免合约执行中断

错误消息长度

错误消息应简洁明了，避免过长导致gas超出限制

Remix效果

Remix环境模拟

```
pragma solidity ^0.8.4;
contract ErrorHandlingDemo {
error InsufficientBalance();
error Unauthorized();
function testRequire(uint256 amount) public {
require(amount <= 100, "Amount too high");
// ...
}
function testCustomError(uint256 amount) public {
if (amount > 100) revert InsufficientBalance();
// ...
}
}
```

控制台输出

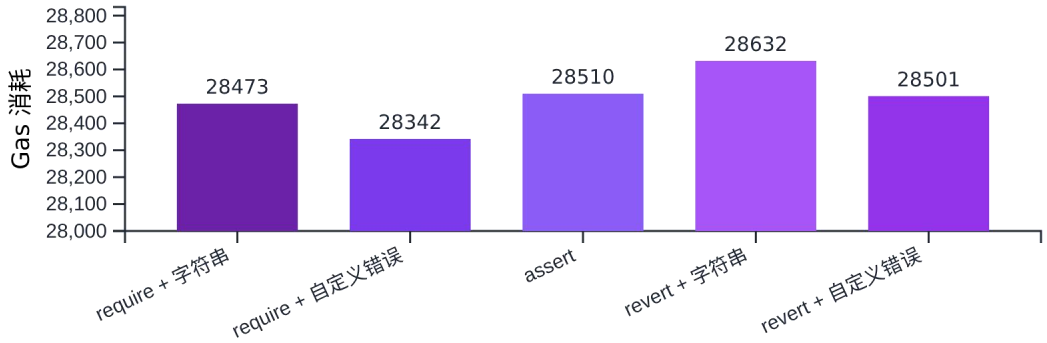
Error: Amount too high (gas: 28473)

Error: VM Exception while processing transaction:

at InsufficientBalance()

at testCustomError() (gas: 28342)

Gas消耗对比



要点

- 在Remix中部署合约并调用不同函数测试错误处理
- 比较require、assert、revert和自定义错误的gas消耗差异
- 使用try-catch捕获并处理不同类型的错误
- 观察不同错误类型对应的错误消息格式

课程总结与实践作业

核心知识点回顾

- require/assert/revert三种错误处理机制的选择与使用
- 自定义错误(0.8.4+)提高gas效率和代码可读性
- try-catch异常捕获的语法与应用场景
- 错误处理最佳实践提升合约安全性

实践作业

作业1：代币合约错误处理

实现一个简单的代币合约，包含适当的错误处理，确保在余额不足、权限不足等情况下正确抛出异常。

作业2：拍卖合约异常捕获

创建一个拍卖合约，使用try-catch捕获可能出现的异常，并实现合理的错误处理逻辑。

作业3：自定义错误优化

将之前作业中的字符串错误消息替换为自定义错误，比较并解释gas消耗的差异。

作业4：多合约错误处理

设计一个包含多个合约的系统，展示不同合约之间的错误传递和处理机制。