Solidity智能合约开发知识

第3.3课：函数与修饰符

学习目标：掌握函数的完整结构、理解四种可见性修饰符、掌握状态修饰符的使用、学会创建自定义modifier、能够合理设计函数权限

预计学习时间：2.5小时

难度等级：入门进阶

目录

---

1. 函数基本结构

1.1 完整的函数定义

函数是智能合约的核心组成部分，用于实现合约的各种功能。一个完整的函数包含多个组成部分。

完整语法：

```
function 函数名(参数列表)
    可见性修饰符
    状态修饰符
    自定义修饰符
    returns (返回类型)
{
    // 函数体
}
```

示例：

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

contract FunctionExample {
    uint256 public value;
    address public owner;
```

```
    constructor() {
        owner = msg.sender;
    }

    // 完整的函数定义
    function setValue(uint256 _value)
        public                  // 可见性修饰符
        onlyOwner               // 自定义修饰符
        returns (bool)          // 返回类型
    {
        value = _value;
        return true;
    }

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

1.2 函数的组成部分

必需部分：

1. **function关键字**：声明这是一个函数
2. **函数名**：标识函数的名称
3. **可见性修饰符**：定义谁可以调用（public/external/internal/private）

可选部分：

1. **参数列表**：函数的输入参数（可以为空）
2. **状态修饰符**：定义函数是否修改状态（view/pure/payable）
3. **自定义修饰符**：权限控制和前置检查
4. **返回值**：函数的输出（可以没有返回值）

1.3 基本函数示例

```
contract BasicFunctions {
    uint256 public counter;

    // 最简单的函数（无参数，无返回值）
    function increment() public {
        counter++;
    }

    // 带参数的函数
    function setCounter(uint256 _value) public {
        counter = _value;
    }
```

```
    // 带返回值的函数
    function getCounter() public view returns (uint256) {
        return counter;
    }

    // 带参数和返回值的函数
    function add(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }

    // 多个返回值
    function getValues() public view returns (uint256, uint256) {
        return (counter, counter * 2);
    }

    // 命名返回值
    function calculate(uint256 a, uint256 b)
        public pure
        returns (uint256 sum, uint256 product)
    {
        sum = a + b;
        product = a * b;
        // 命名返回值可以不写return
    }
}
```

1.4 参数和返回值

参数传递：

```
contract ParameterPassing {
    // 值类型参数（传递副本）
    function updateValue(uint256 _value) public pure returns (uint256) {
        _value = _value + 1;  // 修改的是副本
        return _value;
    }

    // 引用类型参数（需要指定存储位置）
    function processArray(uint256[] memory arr) public pure returns (uint256) {
        return arr.length;
    }

    // calldata参数（只读，更省gas）
    function processArrayCalldata(uint256[] calldata arr)
        external pure returns (uint256)
    {
        return arr.length;
    }
}
```

返回值处理：

```
contract ReturnValues {
    // 单个返回值
    function getSingle() public pure returns (uint256) {
        return 42;
    }

    // 多个返回值
    function getMultiple() public pure returns (uint256, bool, string memory) {
        return (42, true, "Hello");
    }

    // 命名返回值
    function getNamedReturns()
        public pure
        returns (uint256 number, bool flag, string memory message)
    {
        number = 42;
        flag = true;
        message = "Hello";
        // 不需要显式return
    }

    // 调用带返回值的函数
    function callFunction() public pure {
        // 接收所有返回值
        (uint256 num, bool f, string memory msg) = getMultiple();

        // 只接收部分返回值
        (uint256 n, , ) = getMultiple();

        // 忽略返回值
        getMultiple();
    }
}
```

2. 可见性修饰符

2.1 可见性修饰符概览

Solidity提供了四种可见性修饰符，用于控制函数的访问权限。

修饰符	外部调用	内部调用	继承合约调用	Gas成本
public	可以	可以	可以	中等
external	可以	不可以	可以	较低
internal	不可以	可以	可以	-
private	不可以	可以	不可以	-

记忆口诀：

- public 最开放，任何人都可以调用
- external 外部强，只能从外部调用
- internal 内部用，本合约和子合约
- private 最保守，只能本合约调用

2.2 Public - 公开函数

public是最常用的可见性修饰符，任何人都可以调用。

```
contract PublicExample {
    uint256 public value;  // public状态变量自动生成getter函数

    // public函数可以被任何人调用
    function publicFunction() public pure returns (string memory) {
        return "This is public";
    }

    // 内部调用示例
    function internalCall() public pure returns (string memory) {
        return publicFunction();  // 可以内部调用
    }
}

// 外部调用示例
contract Caller {
    PublicExample public example;

    constructor(address _addr) {
        example = PublicExample(_addr);
    }

    function callPublic() public view returns (string memory) {
        return example.publicFunction();  // 外部调用public函数
    }
}
```

public的特点：

1. **外部可调用**：任何账户或合约都可以调用
2. **内部可调用**：合约内部可以直接调用
3. **继承可调用**：子合约可以调用和重写
4. **自动生成getter**：public状态变量自动创建getter函数

使用场景：

- 对外提供的接口函数
- 用户需要调用的功能
- 需要被继承和重写的函数
- 最常用的可见性修饰符

2.3 External - 外部函数

external函数只能从外部调用，在某些情况下比public更省gas。

```
contract ExternalExample {
    // external函数只能从外部调用
    function externalFunction() external pure returns (string memory) {
        return "This is external";
    }

    // 错误：不能在内部直接调用external函数
    // function internalCall() public view returns (string memory) {
    //     return externalFunction();  // 编译错误!
    // }

    // 正确：使用this调用（但这实际上是外部调用，消耗更多gas）
    function callExternal() public view returns (string memory) {
        return this.externalFunction();  // 可以，但不推荐
    }

    // external函数处理大数组更省gas
    function processLargeArray(uint256[] calldata data)
        external pure returns (uint256)
    {
        uint256 sum = 0;
        for(uint256 i = 0; i < data.length; i++) {
            sum += data[i];
        }
        return sum;
    }
}
```

external的优势：

1. **Gas优化**：可以直接从calldata读取参数，不需要复制到memory
2. **适合大数组**：处理大数组或字符串时更高效
3. **明确语义**：清楚表明这是一个外部接口

public vs external Gas对比：

```
contract GasComparison {
    // public函数：参数会从calldata复制到memory
    function publicProcess(uint256[] memory data)
        public pure returns (uint256)
    {
        return data.length;
    }
    // Gas: ~3,000 (100个元素)

    // external函数：直接从calldata读取
    function externalProcess(uint256[] calldata data)
```

```
        external pure returns (uint256)
    {
        return data.length;
    }
    // Gas: ~1,000 (100个元素)
    // 节省: ~66%
}
```

何时使用external:

- 只给外部调用的函数
- 参数包含大数组或长字符串
- 追求gas优化
- 接口定义

2.4 Internal - 内部函数

internal函数只能在合约内部和继承合约中调用。

```
contract InternalExample {
    uint256 private value;

    // internal函数：内部辅助函数
    function _setValue(uint256 _value) internal {
        require(_value > 0, "Value must be positive");
        value = _value;
    }

    // public函数调用internal函数
    function setValue(uint256 _value) public {
        _setValue(_value);  // 内部调用
    }

    // internal辅助函数
    function _calculateFee(uint256 amount) internal pure returns (uint256) {
        return amount * 3 / 100;  // 3% fee
    }

    function processWithFee(uint256 amount) public pure returns (uint256) {
        uint256 fee = _calculateFee(amount);
        return amount - fee;
    }
}

// 继承合约可以调用internal函数
contract InheritedContract is InternalExample {
    function useInternal(uint256 _value) public {
        _setValue(_value);  // 子合约可以调用父合约的internal函数
    }
}
```

internal的特点：

1. **内部可调用**：本合约可以调用
2. **继承可调用**：子合约可以调用和重写
3. **外部不可调用**：外部账户和合约不能调用

使用场景：

- 内部辅助函数
- 可复用的逻辑
- 给子合约继承使用的函数
- 实现细节封装

命名规范：

通常internal函数以下划线开头，如`_setValue`、`_calculateFee`，这是一种常见的命名约定。

2.5 Private - 私有函数

private是最严格的可见性，只能在当前合约内部调用。

```
contract PrivateExample {
    uint256 private secretValue;

    // private函数：只能本合约调用
    function _updateSecret(uint256 _value) private {
        secretValue = _value;
    }

    // public函数调用private函数
    function setSecret(uint256 _value) public {
        _updateSecret(_value);
    }

    // private计算函数
    function _complexCalculation(uint256 a, uint256 b)
        private pure returns (uint256)
    {
        return (a * b) + (a ** 2) - (b ** 2);
    }

    function calculate(uint256 a, uint256 b) public pure returns (uint256) {
        return _complexCalculation(a, b);
    }
}

// 继承合约不能调用private函数
contract InheritedPrivate is PrivateExample {
    function tryCallPrivate() public {
        // _updateSecret(100);  // 编译错误！无法调用父合约的private函数
    }
}
```

重要警告：private不等于隐私

```
contract PrivacyWarning {
    uint256 private secretNumber = 12345;

    // 即使是private，数据仍然在区块链上公开！
    // 任何人都可以通过读取storage来查看
}
```

如何读取private变量（使用Web3.js）：

```
// 任何人都可以读取storage
const value = await web3.eth.getStorageAt(contractAddress, 0);
console.log(value);  // 可以看到"private"的secretNumber
```

private的特点：

1. **只能本合约调用**：子合约也不能调用
2. **最严格的访问控制**
3. **不代表数据隐私**：区块链上所有数据都是公开的

使用场景：

- 纯内部逻辑
- 不希望子合约访问的函数
- 实现细节的封装

2.6 可见性选择指南

决策流程：

```
外部用户需要调用这个函数吗?
├─ 是 → 参数包含大数组或长字符串吗?
│        ├─ 是 → external (省gas)
│        └─ 否 → public
└─ 否 → 子合约需要访问吗?
         ├─ 是 → internal
         └─ 否 → private
```

选择建议：

```
contract VisibilityChoice {
    // 对外接口：用户需要调用 → public或external
    function deposit() public payable {
        // 对外服务
    }

    // 大参数：用external省gas
    function processBatch(uint256[] calldata items) external {
        // 处理大数组
```

```
    }

    // 内部辅助：给本合约和子合约用 → internal
    function _validate(address user) internal view returns (bool) {
        // 内部验证逻辑
    }

    // 私有逻辑：只给本合约用 → private
    function _calculateSecret(uint256 seed) private pure returns (uint256) {
        // 私有计算
    }
}
```

最佳实践：

1. **默认使用最严格的可见性**：从private开始，需要时再放宽
2. **对外接口明确**：public或external清楚表明意图
3. **大数组用external**：显著节省gas
4. **内部函数用下划线**：`_functionName`作为命名约定

---

3. 状态修饰符

3.1 状态修饰符概览

状态修饰符定义了函数与区块链状态的交互方式。

修饰符	读取状态	修改状态	接收ETH	Gas消耗（外部调用）
默认	可以	可以	不可以	正常消耗
view	可以	不可以	不可以	0（不改变状态）
pure	不可以	不可以	不可以	0（不改变状态）
payable	可以	可以	可以	正常消耗

选择建议：

- 需要修改状态？→ 默认或payable
- 只读取状态？→ view
- 不读不写？→ pure
- 需要接收ETH？→ payable

重要提示：能用pure就pure，能用view就view！

3.2 View - 只读函数

view函数承诺不修改状态，只读取数据。

```
contract ViewExample {
```

```
    uint256 public counter = 0;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // view函数：读取状态变量
    function getCounter() public view returns (uint256) {
        return counter;  // 可以读取状态
    }

    // view函数：可以读取多个状态变量
    function getInfo() public view returns (uint256, address) {
        return (counter, owner);
    }

    // view函数：可以读取全局变量
    function getBlockInfo() public view returns (uint256, address) {
        return (block.timestamp, msg.sender);
    }

    // view函数：可以进行计算
    function calculateDouble() public view returns (uint256) {
        return counter * 2;  // 读取并计算
    }

    // view函数：可以调用其他view函数
    function complexView() public view returns (uint256) {
        uint256 c = getCounter();
        return c * 2;
    }
}
```

view函数可以做什么：

1. 读取状态变量
2. 读取全局变量（msg、block、tx等）
3. 调用其他view或pure函数
4. 进行计算和逻辑判断

view函数不能做什么：

```
contract ViewRestrictions {
    uint256 public value;

    function badView1() public view {
        // value = 100;  // 错误：不能修改状态变量
    }

    function badView2() public view {
        // selfdestruct(payable(msg.sender));  // 错误：不能销毁合约
```

```
    }

    function badView3() public view {
        // emit SomeEvent();  // 错误：不能触发事件
    }
}
```

Gas消耗：

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

    // 外部直接调用view函数：不消耗gas
    function getValue() public view returns (uint256) {
        return value;
    }

    // 但在交易中调用view函数：消耗gas
    function modifyAndView() public returns (uint256) {
        value = 200;  // 修改状态，消耗gas
        return getValue();  // 内部调用view，也消耗gas
    }
}
```

重要提示：

- 外部直接调用view函数（只读查询）：不消耗gas
- 在交易中调用view函数：消耗gas
- view承诺不修改状态，但编译器会检查

3.3 Pure - 纯函数

pure函数既不读取也不修改状态，只使用参数和局部变量。

```
contract PureExample {
    // pure函数：只使用参数
    function add(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }

    // pure函数：数学计算
    function calculate(uint256 x) public pure returns (uint256) {
        uint256 result = x * x + 2 * x + 1;
        return result;
    }

    // pure函数：字符串处理
    function concatenate(string memory a, string memory b)
        public pure returns (string memory)
    {
        return string(abi.encodePacked(a, b));
```

```
    }

    // pure函数：可以调用其他pure函数
    function complexPure(uint256 a, uint256 b)
        public pure returns (uint256)
    {
        uint256 sum = add(a, b);
        return sum * 2;
    }

    // pure函数：数组处理
    function arraySum(uint256[] memory arr)
        public pure returns (uint256)
    {
        uint256 sum = 0;
        for(uint256 i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        return sum;
    }
}
```

pure函数可以做什么：

1. 使用函数参数
2. 使用局部变量
3. 调用其他pure函数
4. 进行纯计算

pure函数不能做什么：

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

    function badPure1() public pure returns (uint256) {
        // return value;  // 错误：不能读取状态变量
    }

    function badPure2() public pure returns (uint256) {
        // return block.timestamp;  // 错误：不能读取全局变量
    }

    function badPure3() public pure returns (address) {
        // return msg.sender;  // 错误：不能读取msg
    }

    function badPure4() public pure returns (uint256) {
        // return address(this).balance;  // 错误：不能读取余额
    }
}
```

pure函数的类比：

pure函数就像数学函数：

```
f(x) = x + 1
```

- 输入确定，输出确定
- 没有副作用
- 不依赖外部状态

使用场景：

```
contract PureUseCases {
    // 工具函数
    function min(uint256 a, uint256 b) public pure returns (uint256) {
        return a < b ? a : b;
    }

    function max(uint256 a, uint256 b) public pure returns (uint256) {
        return a > b ? a : b;
    }

    // 验证函数
    function isValidAddress(address addr) public pure returns (bool) {
        return addr != address(0);
    }

    // 计算函数
    function calculateFee(uint256 amount, uint256 feePercent)
        public pure returns (uint256)
    {
        return amount * feePercent / 100;
    }
}
```

3.4 Payable - 可支付函数

payable函数可以接收ETH。

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

    // payable函数：可以接收ETH
    function deposit() public payable {
        require(msg.value > 0, "Must send ETH");
        balances[msg.sender] += msg.value;
        totalReceived += msg.value;
    }
```

```
    // payable函数：带参数
    function depositWithMessage(string memory message) public payable {
        require(msg.value > 0, "Must send ETH");
        balances[msg.sender] += msg.value;
        // 可以使用message参数
    }

    // 查询合约余额
    function getContractBalance() public view returns (uint256) {
        return address(this).balance;
    }

    // 提取ETH
    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }

    // 接收ETH的特殊函数
    receive() external payable {
        // 直接转账ETH到合约时调用
        balances[msg.sender] += msg.value;
    }

    fallback() external payable {
        // 调用不存在的函数时调用
        balances[msg.sender] += msg.value;
    }
}
```

没有payable会发生什么：

```
contract NoPayable {
    // 没有payable修饰符
    function normalFunction() public {
        // 如果调用时发送ETH，交易会失败并回退
    }

    // 错误：尝试访问msg.value但没有payable
    function badFunction() public {
        // uint256 amount = msg.value;  // 编译错误!
    }
}
```

msg.value的使用：

```
contract MsgValueExample {
    event Received(address sender, uint256 amount);

    function checkValue() public payable {
```

```
        if(msg.value > 1 ether) {
            emit Received(msg.sender, msg.value);
        } else {
            revert("Must send at least 1 ETH");
        }
    }

    function getExactAmount() public payable {
        require(msg.value == 0.5 ether, "Must send exactly 0.5 ETH");
        // 处理逻辑
    }
}
```

receive和fallback：

```
contract SpecialPayable {
    event Received(address sender, uint256 amount);
    event Fallback(address sender, uint256 amount, bytes data);

    // receive：接收纯ETH转账（无data）
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }

    // fallback：调用不存在的函数或带data的转账
    fallback() external payable {
        emit Fallback(msg.sender, msg.value, msg.data);
    }
}
```

执行流程：

```
发送ETH到合约
    ↓
msg.data是空的吗?
    ├─ 是 → 有receive()吗?
    │       ├─ 是 → 调用receive()
    │       └─ 否 → 调用fallback()
    └─ 否 → 调用对应函数或fallback()
```

---

4. 自定义Modifier

4.1 什么是Modifier

Modifier（修饰符）是函数执行前的检查点，用于权限控制、状态检查和参数验证。

基本语法：

```
modifier 修饰符名称(参数) {
    require(条件, "错误信息");
    _;  // 下划线表示函数体的位置
}
```

下划线（_）的作用：

下划线是占位符，表示被修饰函数的函数体将在这个位置执行。

```
contract ModifierBasics {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // 定义modifier
    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;  // 函数体会插入到这里
    }

    // 使用modifier
    function restrictedFunction() public onlyOwner {
        // 只有owner可以执行
    }
}
```

4.2 Modifier执行流程

执行顺序：

```
调用函数
    ↓
执行modifier检查
    ↓
条件满足?
    ├─ 是 → 执行函数体（_的位置）
    └─ 否 → 回退交易
```

实际执行等价：

```
// 使用modifier的函数
function setValue(uint256 _value) public onlyOwner {
    value = _value;
}

// 等价于
function setValue(uint256 _value) public {
    require(msg.sender == owner, "Not the owner");  // modifier的内容
    value = _value;  // 原函数体
}
```

4.3 常用Modifier模式

模式1：权限控制

```
contract AccessControl {
    address public owner;
    mapping(address => bool) public admins;

    constructor() {
        owner = msg.sender;
    }

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

    // 只有admin
    modifier onlyAdmin() {
        require(admins[msg.sender], "Not an admin");
        _;
    }

    // owner或admin
    modifier onlyAuthorized() {
        require(
            msg.sender == owner || admins[msg.sender],
            "Not authorized"
        );
        _;
    }

    function addAdmin(address admin) public onlyOwner {
        admins[admin] = true;
    }

    function removeAdmin(address admin) public onlyOwner {
        admins[admin] = false;
    }
```

```
    function adminFunction() public onlyAdmin {
        // 只有admin可以调用
    }
}
```

模式2：状态检查

```
contract StateCheck {
    bool public paused = false;
    bool public initialized = false;

    // 未暂停检查
    modifier whenNotPaused() {
        require(!paused, "Contract is paused");
        _;
    }

    // 已暂停检查
    modifier whenPaused() {
        require(paused, "Contract is not paused");
        _;
    }

    // 初始化检查
    modifier whenInitialized() {
        require(initialized, "Not initialized");
        _;
    }

    function normalOperation() public whenNotPaused whenInitialized {
        // 正常操作
    }

    function emergencyStop() public whenNotPaused {
        paused = true;
    }

    function resume() public whenPaused {
        paused = false;
    }
}
```

模式3：参数验证

```
contract ParameterValidation {
    // 地址验证
    modifier validAddress(address _addr) {
        require(_addr != address(0), "Invalid address");
        _;
    }
```

```
    // 金额验证
    modifier minValue(uint256 _minValue) {
        require(msg.value >= _minValue, "Insufficient value");
        _;
    }

    // 范围验证
    modifier inRange(uint256 _value, uint256 _min, uint256 _max) {
        require(_value >= _min && _value <= _max, "Out of range");
        _;
    }

    function transfer(address to, uint256 amount)
        public
        validAddress(to)
    {
        // 转账逻辑
    }

    function deposit() public payable minValue(0.1 ether) {
        // 至少0.1 ETH
    }

    function setValue(uint256 value) public inRange(value, 1, 100) {
        // value必须在1-100之间
    }
}
```

模式4：时间锁

```
contract TimeLock {
    uint256 public lockTime;

    modifier afterTime(uint256 _time) {
        require(block.timestamp >= _time, "Too early");
        _;
    }

    modifier beforeTime(uint256 _time) {
        require(block.timestamp < _time, "Too late");
        _;
    }

    constructor() {
        lockTime = block.timestamp + 1 days;
    }

    function executeAfterLock() public afterTime(lockTime) {
        // 锁定期后才能执行
    }
```

```
    function executeBeforeLock() public beforeTime(lockTime) {
        // 锁定期前才能执行
    }
}
```

模式5：重入保护

```
contract ReentrancyGuard {
    bool private locked = false;

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

    function withdraw(uint256 amount) public noReentrant {
        // 防止重入攻击
        // 提取资金逻辑
    }
}
```

4.4 组合多个Modifier

一个函数可以使用多个modifier。

```
contract MultipleModifiers {
    address public owner;
    bool public paused = false;
    uint256 public value;

    constructor() {
        owner = msg.sender;
    }

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

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

    modifier validValue(uint256 _value) {
        require(_value > 0, "Invalid value");
        _;
```

```
    }

    // 组合三个modifier
    function criticalFunction(uint256 _value)
        public
        onlyOwner
        whenNotPaused
        validValue(_value)
    {
        value = _value;
    }
}
```

执行顺序：

```
调用 criticalFunction(100)
    ↓
1. 检查 onlyOwner
    ↓
2. 检查 whenNotPaused
    ↓
3. 检查 validValue(100)
    ↓
4. 都通过，执行函数体
    ↓
完成
```

任何一个检查失败，交易立即回退。

4.5 带返回值的Modifier

Modifier可以在函数执行前后进行操作。

```
contract ModifierWithLogic {
    uint256 public counter = 0;

    // modifier在函数前后执行
    modifier countCalls() {
        counter++;  // 函数执行前
        _;          // 执行函数体
        counter++;  // 函数执行后
    }

    function doSomething() public countCalls {
        // 每次调用，counter增加2
    }

    // modifier可以修改返回值（不推荐）
    modifier addOne() {
        _;
        // 注意：不能直接修改返回值
```

```
    }
}
```

---

5. 函数重载

5.1 什么是函数重载

函数重载（Function Overloading）允许同名函数有不同的参数。

```
contract FunctionOverloading {
    // 版本1：两个参数
    function transfer(address to, uint256 amount) public {
        // 简单转账
    }

    // 版本2：三个参数（重载）
    function transfer(
        address to,
        uint256 amount,
        string memory memo
    ) public {
        // 带备注的转账
    }

    // 版本3：不同类型（重载）
    function getValue() public pure returns (uint256) {
        return 42;
    }

    function getValue(uint256 multiplier) public pure returns (uint256) {
        return 42 * multiplier;
    }
}
```

5.2 重载规则

可以重载的情况：

1. **参数数量不同**
2. **参数类型不同**
3. **参数顺序不同**

```
contract OverloadingRules {
    // 参数数量不同
    function process(uint256 a) public pure returns (uint256) {
        return a;
    }

    function process(uint256 a, uint256 b) public pure returns (uint256) {
```

```
        return a + b;
    }

    // 参数类型不同
    function getValue(uint256 id) public pure returns (uint256) {
        return id;
    }

    function getValue(address user) public pure returns (address) {
        return user;
    }

    // 参数顺序不同
    function swap(uint256 a, address b) public pure {}
    function swap(address a, uint256 b) public pure {}
}
```

不能重载的情况：

```
contract CannotOverload {
    // 错误：只有返回值不同不能重载
    // function test() public pure returns (uint256) {
    //     return 1;
    // }

    // function test() public pure returns (bool) {  // 编译错误！
    //     return true;
    // }

    // 错误：只有modifier不同不能重载
    // function test() public pure {}
    // function test() public view {}  // 编译错误！
}
```

5.3 调用重载函数

Solidity会根据参数自动匹配正确的函数。

```
contract CallOverloaded {
    function transfer(address to, uint256 amount) public {
        // 版本1
    }

    function transfer(
        address to,
        uint256 amount,
        string memory memo
    ) public {
        // 版本2
    }
```

```
    function testCalls() public {
        address addr = address(0x123);

        // 调用版本1
        transfer(addr, 100);

        // 调用版本2
        transfer(addr, 100, "Payment");
    }
}
```

5.4 重载的实际应用

```
contract PracticalOverloading {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event TransferWithMemo(
        address indexed from,
        address indexed to,
        uint256 value,
        string memo
    );

    mapping(address => uint256) public balances;

    // 简单转账
    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
        emit Transfer(msg.sender, to, amount);
    }

    // 带备注的转账
    function transfer(
        address to,
        uint256 amount,
        string memory memo
    ) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
        emit TransferWithMemo(msg.sender, to, amount, memo);
    }

    // 批量转账
    function transfer(address[] memory recipients, uint256[] memory amounts) public {
        require(recipients.length == amounts.length, "Length mismatch");

        for(uint256 i = 0; i < recipients.length; i++) {
            require(balances[msg.sender] >= amounts[i], "Insufficient balance");
            balances[msg.sender] -= amounts[i];
```

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

---

6. 最佳实践

6.1 可见性选择原则

原则1：默认使用最严格的可见性

```
contract VisibilityPrinciple {
    // 从最严格开始
    function _helperFunction() private pure returns (uint256) {
        return 42;
    }

    // 需要子合约访问时改为internal
    function _internalHelper() internal pure returns (uint256) {
        return 42;
    }

    // 需要对外提供时才用public/external
    function publicInterface() public pure returns (uint256) {
        return _helperFunction();
    }
}
```

原则2：大参数用external

```
contract LargeParameters {
    // 大数组：用external + calldata
    function processBatch(uint256[] calldata items) external {
        // 处理大数组
    }

    // 小参数：用public也可以
    function processSmall(uint256 a, uint256 b) public {
        // 处理小数据
    }
}
```

6.2 状态修饰符原则

原则1：能用pure就pure，能用view就view

```
contract StatePrinciple {
```

```
    uint256 public value = 100;

    // 纯计算：用pure
    function add(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }

    // 只读取：用view
    function getValue() public view returns (uint256) {
        return value;
    }

    // 需要修改：不加修饰符
    function setValue(uint256 _value) public {
        value = _value;
    }
}
```

原则2：明确标注函数是否修改状态

```
contract ClearIntent {
    uint256 public counter;

    // 清楚表明这是查询函数
    function getCounter() public view returns (uint256) {
        return counter;
    }

    // 清楚表明这会修改状态
    function incrementCounter() public {
        counter++;
    }
}
```

6.3 Modifier原则

原则1：权限控制用modifier

```
contract ModifierPrinciple {
    address public owner;

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

    // 好：使用modifier
    function restrictedFunction() public onlyOwner {
        // 逻辑
    }
```

```
    // 不好：手动检查
    function badFunction() public {
        require(msg.sender == owner, "Not owner");  // 不推荐
        // 逻辑
    }
}
```

原则2：modifier名称要清晰

```
contract ClearModifiers {
    // 好的命名
    modifier onlyOwner() { _; }
    modifier whenNotPaused() { _; }
    modifier validAddress(address addr) { _; }

    // 不好的命名
    modifier check() { _; }  // 太笼统
    modifier m1() { _; }     // 无意义
}
```

原则3：检查失败要有明确错误信息

```
contract ClearErrors {
    address public owner;

    // 好：明确的错误信息
    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    // 不好：无错误信息
    modifier badModifier() {
        require(msg.sender == owner);  // 不推荐
        _;
    }
}
```

6.4 安全原则

原则1：private不等于隐私

```
contract PrivacyWarning {
    // 即使是private，数据仍然公开
    uint256 private secretNumber = 12345;

    // 不要存储真正的隐私数据在区块链上
    // 任何人都可以读取storage
}
```

原则2：谨慎使用external

```
contract ExternalSafety {
    // external函数容易受到攻击
    // 确保有足够的验证
    function externalFunction(uint256 value) external {
        require(value > 0, "Invalid value");
        require(msg.sender != address(0), "Invalid sender");
        // 更多验证...
    }
}
```

原则3：组合modifier要考虑顺序

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

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

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

    // 先检查权限，再检查状态（更合理）
    function goodOrder() public onlyOwner whenNotPaused {
        // 逻辑
    }
}
```

6.5 命名规范

```
contract NamingConventions {
    // 内部函数：下划线开头
    function _internalHelper() internal {}

    // 私有函数：下划线开头
    function _privateHelper() private {}

    // Public/External：正常命名
    function publicFunction() public {}
    function externalFunction() external {}

    // Modifier：描述性命名
    modifier onlyOwner() { _; }
    modifier whenNotPaused() { _; }
```

```
    modifier validAddress(address addr) { _; }
}
```

---

7. 实战练习

练习1：三角色权限管理系统

需求：

创建一个完整的权限管理系统：

1. 定义三种角色：Owner、Admin、User
2. 实现角色分配和检查
3. 不同角色有不同权限
4. Owner可以添加Admin
5. Admin可以添加User
6. 所有人可以查询角色

参考代码框架：

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

contract RoleManagement {
    // TODO: 定义角色枚举
    enum Role { None, User, Admin, Owner }

    // TODO: 存储用户角色
    mapping(address => Role) public roles;

    address public owner;

    constructor() {
        owner = msg.sender;
        roles[msg.sender] = Role.Owner;
    }

    // TODO: 定义modifier
    modifier onlyOwner() {
        // 检查是否为Owner
    }

    modifier onlyAdmin() {
        // 检查是否为Admin或Owner
    }

    // TODO: 实现功能函数
    function addAdmin(address user) public onlyOwner {
        // Owner添加Admin
    }
```

```
    function addUser(address user) public onlyAdmin {
        // Admin添加User
    }

    function getRole(address user) public view returns (Role) {
        // 查询角色
    }
}
```

完整参考答案:

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

contract RoleManagement {
    enum Role { None, User, Admin, Owner }

    mapping(address => Role) public roles;
    address public owner;

    event RoleAssigned(address indexed user, Role role);
    event RoleRevoked(address indexed user);

    constructor() {
        owner = msg.sender;
        roles[msg.sender] = Role.Owner;
        emit RoleAssigned(msg.sender, Role.Owner);
    }

    modifier onlyOwner() {
        require(roles[msg.sender] == Role.Owner, "Only owner can call");
        _;
    }

    modifier onlyAdmin() {
        require(
            roles[msg.sender] == Role.Admin || roles[msg.sender] == Role.Owner,
            "Only admin or owner can call"
        );
        _;
    }

    modifier onlyUser() {
        require(roles[msg.sender] != Role.None, "Must have a role");
        _;
    }

    function addAdmin(address user) public onlyOwner {
        require(user != address(0), "Invalid address");
        require(roles[user] != Role.Owner, "Cannot change owner role");
```

```
        roles[user] = Role.Admin;
        emit RoleAssigned(user, Role.Admin);
    }

    function addUser(address user) public onlyAdmin {
        require(user != address(0), "Invalid address");
        require(roles[user] == Role.None, "User already has a role");
        roles[user] = Role.User;
        emit RoleAssigned(user, Role.User);
    }

    function revokeRole(address user) public onlyOwner {
        require(user != owner, "Cannot revoke owner role");
        delete roles[user];
        emit RoleRevoked(user);
    }

    function getRole(address user) public view returns (Role) {
        return roles[user];
    }

    function hasRole(address user, Role role) public view returns (bool) {
        return roles[user] == role;
    }
}
```

练习2：支付合约

需求：

创建一个完整的支付合约：

1. 支持存款（deposit）
2. 支持提款（withdraw）
3. 支持紧急停止（pause）
4. Owner可以暂停/恢复合约
5. 查询余额
6. 限制最小存款金额

参考答案：

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

contract PaymentContract {
    address public owner;
    bool public paused = false;
    mapping(address => uint256) public balances;

    uint256 public constant MIN_DEPOSIT = 0.01 ether;

    event Deposit(address indexed user, uint256 amount);
```

```
    event Withdrawal(address indexed user, uint256 amount);
    event Paused(address indexed by);
    event Unpaused(address indexed by);

    constructor() {
        owner = msg.sender;
    }

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

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

    modifier whenPaused() {
        require(paused, "Contract is not paused");
        _;
    }

    modifier minValue(uint256 minAmount) {
        require(msg.value >= minAmount, "Insufficient value");
        _;
    }

    function deposit() public payable whenNotPaused minValue(MIN_DEPOSIT) {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    function withdraw(uint256 amount) public whenNotPaused {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
        emit Withdrawal(msg.sender, amount);
    }

    function getBalance() public view returns (uint256) {
        return balances[msg.sender];
    }

    function getContractBalance() public view returns (uint256) {
        return address(this).balance;
    }

    function pause() public onlyOwner whenNotPaused {
        paused = true;
        emit Paused(msg.sender);
    }
```

```
    function unpause() public onlyOwner whenPaused {
        paused = false;
        emit Unpaused(msg.sender);
    }

    receive() external payable {
        deposit();
    }
}
```

练习3：多签名钱包（挑战）

需求：

创建一个简单的多签名钱包：

1. 需要多个签名者确认才能执行交易
2. 提交交易提案
3. 签名者确认提案
4. 达到阈值后执行

提示：

- 使用struct存储提案
- 使用mapping记录确认
- 使用modifier检查权限

---

8. 常见问题解答

Q1：public和external的区别是什么?

答：两者都可以从外部调用，但有重要区别。

主要区别：

1. **内部调用**：
    - public：可以内部调用
    - external：不能直接内部调用（需要用`this.function()`）
2. **Gas成本**：
    - public：参数从calldata复制到memory
    - external：直接从calldata读取，更省gas
3. **参数类型**：
    - public：参数必须是memory
    - external：参数可以是calldata

何时用external：

- 只给外部调用的函数
- 参数包含大数组或长字符串

- 追求gas优化

Q2：view和pure的区别是什么?

答：两者都不修改状态，但读取权限不同。

对比：

特性	view	pure
读取状态变量	可以	不可以
读取全局变量	可以	不可以
修改状态	不可以	不可以
使用参数	可以	可以

使用场景：

- view：查询状态、获取数据
- pure：纯计算、工具函数

Q3：多个modifier的执行顺序是怎样的?

答：从上到下依次执行，任何一个失败就回退。

```
function test()
    public
    modifier1  // 先执行
    modifier2  // 再执行
    modifier3  // 最后执行
{
    // 都通过后执行函数体
}
```

Q4：为什么private变量不是隐私的?

答：区块链上所有数据都是公开的。

原因：

- 所有storage数据都存储在区块链上
- 任何人都可以读取storage
- private只是访问控制，不是加密

如何保护隐私：

- 不在链上存储隐私数据
- 使用链下存储
- 使用零知识证明等加密技术

Q5：什么时候使用modifier，什么时候直接用require?

答：根据复用性和可读性决定。

使用modifier：

- 需要在多个函数中复用
- 权限控制
- 状态检查
- 提高代码可读性

直接用require：

- 只在一个函数中使用
- 函数特定的验证
- 简单检查

Q6：receive和fallback的区别?

答：两者都可以接收ETH，但触发条件不同。

receive：

- 接收纯ETH转账（msg.data为空）
- 必须是external payable
- 更节省gas

fallback：

- 调用不存在的函数
- 或带data的ETH转账
- 可以不是payable

优先级：

```
发送ETH → msg.data为空?
    ├─ 是 → 有receive? → 调用receive
    │              └─ 无 → 调用fallback
    └─ 否 → 调用fallback
```

Q7：函数重载有什么限制?

答：只能通过参数区分，不能通过返回值区分。

可以重载：

- 参数数量不同
- 参数类型不同
- 参数顺序不同

不能重载：

- 只有返回值不同

- 只有modifier不同
- 只有可见性不同

---

9. 知识点总结

函数基本结构

完整语法：

```
function 函数名(参数)
    可见性修饰符
    状态修饰符
    自定义修饰符
    returns (返回类型)
{
    // 函数体
}
```

可见性修饰符（4种）

修饰符	描述	使用场景
public	任何人都可以调用	对外接口
external	只能从外部调用，省gas	外部专用、大参数
internal	内部和继承合约可调用	内部逻辑、可继承
private	只能本合约调用	私有逻辑

状态修饰符（3种）

修饰符	读取状态	修改状态	接收ETH
view	可以	不可以	不可以
pure	不可以	不可以	不可以
payable	可以	可以	可以

自定义Modifier

作用：

- 权限控制
- 状态检查
- 参数验证
- 可组合使用

语法：

```
modifier 名称(参数) {
    require(条件, "错误");
    _;
}
```

函数重载

规则：

- 同名不同参数可以重载
- 参数类型、数量、顺序不同即可
- 只有返回值不同不能重载

---

10. 学习检查清单

完成本课后，你应该能够：

函数基础：

- [ ] 理解函数的完整结构
- [ ] 会定义带参数和返回值的函数
- [ ] 会处理多个返回值
- [ ] 会使用命名返回值

可见性修饰符：

- [ ] 理解四种可见性的区别
- [ ] 知道何时使用public/external/internal/private
- [ ] 理解public和external的gas差异
- [ ] 知道private不等于隐私

状态修饰符：

- [ ] 理解view/pure/payable的区别
- [ ] 会正确使用view和pure
- [ ] 会处理payable函数和msg.value
- [ ] 理解receive和fallback

自定义Modifier：

- [ ] 会创建自定义modifier
- [ ] 理解modifier的执行流程
- [ ] 会组合多个modifier
- [ ] 掌握常用modifier模式

函数重载：

- [ ] 理解函数重载的规则
- [ ] 会实现函数重载
- [ ] 知道重载的限制

最佳实践：

- [ ] 知道如何选择可见性
- [ ] 理解何时用modifier
- [ ] 掌握命名规范
- [ ] 理解安全原则

---

11. 下一步学习

完成本课后，建议：

1. **实践所有示例代码**：在Remix中部署和测试
2. **完成练习题**：巩固知识点
3. **研究真实项目**：分析OpenZeppelin、Uniswap等项目的函数设计
4. **准备学习第4.1课**：控制流语句

下节课预告：第4.1课 - 控制流语句

我们将学习：

- if-else条件语句
- for/while/do-while循环
- break和continue
- require/assert/revert错误处理

---

12. 扩展资源

官方文档：

- Solidity函数文档：https://docs.soliditylang.org/en/latest/contracts.html#functions
- 可见性和Getter：https://docs.soliditylang.org/en/latest/contracts.html#visibility-and-getters

学习资源：

- Solidity by Example - Functions：https://solidity-by-example.org/function/
- Solidity by Example - View and Pure：https://solidity-by-example.org/view-and-pure-functions/

实战项目：
研究开源项目中的函数设计：

- OpenZeppelin Contracts：标准合约库
- Uniswap V2/V3：DEX协议
- Compound：借贷协议

安全资源：

- Smart Contract Best Practices
- Common Attack Vectors

课程结束

恭喜你完成第3.3课！函数和修饰符是智能合约的核心组成部分，掌握它们的正确使用方式是成为优秀Solidity开发者的关键。

核心要点回顾：

- 可见性修饰符关系到安全和gas优化
- external比public更省gas（大参数）
- private不等于数据隐私
- view和pure查询不消耗gas
- payable才能接收ETH
- modifier是权限控制的最佳方式

记住：函数设计体现了合约的安全性和效率。始终从最严格的可见性开始，能用pure就pure，能用view就view！

继续加油，下节课见！