Solidity智能合约开发知识

第6.1课：合约继承

学习目标：理解继承的作用和优势、掌握单继承和多重继承、学会使用super关键字、掌握函数重写机制、理解抽象合约和接口的使用

预计学习时间：2.5小时

难度等级：进阶

目录

---

1. 为什么需要继承

1.1 代码复用的问题

在没有继承机制的情况下，开发者会遇到严重的代码复用问题。

问题场景：

假设你要创建三个代币合约：稳定币、治理代币、奖励代币。它们都需要：

- ERC20基本功能（transfer、approve等）
- 权限控制（只有owner可以执行某些操作）
- 暂停功能（紧急情况下暂停转账）

没有继承的做法：

```
// 稳定币合约
contract StableCoin {
    // 复制粘贴ERC20代码
    mapping(address => uint256) public balanceOf;
    function transfer(address to, uint256 amount) public { }

    // 复制粘贴权限控制代码
    address public owner;
    modifier onlyOwner() { }
```

```
    // 复制粘贴暂停功能代码
    bool public paused;
    modifier whenNotPaused() { }
    function pause() public { }
}

// 治理代币合约
contract GovernanceToken {
    // 又复制粘贴一遍所有代码...
    mapping(address => uint256) public balanceOf;
    function transfer(address to, uint256 amount) public { }
    address public owner;
    // ... 完全重复的代码
}

// 奖励代币合约
contract RewardToken {
    // 再次复制粘贴...
}
```

这种做法的问题：

1. **代码冗余**：
    - 三个合约有90%的代码相同
    - 浪费存储空间
    - 增加部署成本
2. **维护困难**：
    - 发现bug需要修改三个地方
    - 容易遗漏
    - 一致性难以保证
3. **升级麻烦**：
    - 添加新功能需要修改所有合约
    - 无法批量更新
    - 测试成本高
4. **容易出错**：
    - 复制粘贴可能出错
    - 某个合约可能用旧版本代码
    - 难以追踪哪个版本最新

真实案例：

某项目有20个代币合约，都是复制粘贴的代码。发现一个安全漏洞后，开发者修改了18个合约，遗漏了2个，最终导致黑客攻击，损失数百万美元。

1.2 继承的解决方案

继承（Inheritance）是面向对象编程的核心特性，它允许一个合约（子合约）继承另一个合约（父合约）的属性和方法。

使用继承的做法：

```
// 基础合约1：ERC20功能
contract BaseERC20 {
    mapping(address => uint256) public balanceOf;

    function transfer(address to, uint256 amount) public virtual returns (bool) {
        require(balanceOf[msg.sender] >= amount);
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }
}

// 基础合约2：权限控制
contract Ownable {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

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

// 基础合约3：暂停功能
contract Pausable {
    bool public paused;

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

    function pause() internal {
        paused = true;
    }
}

// 子合约：继承所有功能
contract StableCoin is BaseERC20, Ownable, Pausable {
    // 自动获得所有父合约的功能
    // 只需要添加特有功能

    function emergencyPause() public onlyOwner {
        pause();
    }
}
```

```
contract GovernanceToken is BaseERC20, Ownable, Pausable {
    // 同样继承所有功能
}

contract RewardToken is BaseERC20, Ownable, Pausable {
    // 同样继承所有功能
}
```

继承的优势：

1. **代码复用**：
   - 公共功能只写一次
   - 多个子合约共享
   - 减少90%以上的重复代码
2. **易于维护**：
   - bug修复只改一处
   - 所有子合约自动受益
   - 保证一致性
3. **模块化设计**：
   - 功能分离清晰
   - 每个合约职责单一
   - 易于理解和测试
4. **灵活扩展**：
   - 子合约可以添加新功能
   - 可以重写父合约函数
   - 组合不同功能模块

1.3 继承的实际应用场景

场景1：代币项目

```
// 基础代币 → 标准代币 → 项目代币
ERC20 → ERC20Burnable → MyToken
```

场景2：权限管理

```
// 基础权限 → 角色管理 → 具体合约
Ownable → AccessControl → ProjectContract
```

场景3：安全功能

```
// 基础合约 → 安全增强 → 最终合约
BaseContract → ReentrancyGuard + Pausable → SecureContract
```

场景4：可升级合约

```
// 存储合约 → 逻辑合约 → 代理合约
Storage → Logic → Proxy
```

---

2. 单继承

2.1 单继承基础语法

单继承是最简单的继承形式，一个子合约只继承一个父合约。

基本语法：

```
contract Parent {
    // 父合约代码
}

contract Child is Parent {
    // 子合约代码
    // 自动继承Parent的所有内容
}
```

关键字说明：

- `is`：表示继承关系
- `Child`：子合约（派生合约）
- `Parent`：父合约（基础合约）

简单示例：

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

contract Parent {
    uint256 public value;

    function getValue() public view returns (uint256) {
        return value;
    }

    function setValue(uint256 _value) public {
        value = _value;
    }
}

contract Child is Parent {
    // 自动继承:
    // - value状态变量
    // - getValue()函数
    // - setValue()函数
```

```
    // 添加新功能
    function doubleValue() public view returns (uint256) {
        return value * 2;  // 可以直接访问父合约的value
    }
}
```

子合约获得了什么：

```
// 部署Child合约后，可以调用：
Child child = new Child();

child.value();          // 继承自Parent
child.getValue();       // 继承自Parent
child.setValue(100);    // 继承自Parent
child.doubleValue();    // Child自己的函数
```

2.2 访问权限

子合约对父合约的访问权限取决于可见性修饰符。

父合约可见性	子合约可访问	外部可访问
public	是	是
internal	是	否
private	否	否
external	是（作为外部调用）	是

示例：

```
contract Parent {
    uint256 public publicVar = 1;       // 子合约可访问
    uint256 internal internalVar = 2;   // 子合约可访问
    uint256 private privateVar = 3;     // 子合约不可访问

    function publicFunc() public pure returns (string memory) {
        return "public";
    }

    function internalFunc() internal pure returns (string memory) {
        return "internal";
    }

    function privateFunc() private pure returns (string memory) {
        return "private";
    }
}
```

```
contract Child is Parent {
    function test() public view returns (uint256, uint256) {
        // 可以访问public和internal
        uint256 a = publicVar;
        uint256 b = internalVar;
        // uint256 c = privateVar;  // 编译错误！无法访问private

        publicFunc();    // 可以调用
        internalFunc();  // 可以调用
        // privateFunc();  // 编译错误！无法访问private

        return (a, b);
    }
}
```

重要理解：

- **public**：最开放，子合约和外部都能访问
- **internal**：只有子合约能访问，外部不能
- **private**：最严格，连子合约都不能访问

常见错误：

```
contract Parent {
    uint256 private secretValue = 100;
}

contract Child is Parent {
    function getSecret() public view returns (uint256) {
        // return secretValue;  // 编译错误！
        // private变量子合约无法访问
    }
}
```

解决方案：

如果希望子合约访问，应该使用`internal`而不是`private`。

```
contract Parent {
    uint256 internal protectedValue = 100;  // 改为internal
}

contract Child is Parent {
    function getValue() public view returns (uint256) {
        return protectedValue;  // 可以访问
    }
}
```

2.3 实际示例：代币合约

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

// 父合约：基础代币功能
contract BaseToken {
    string public name;
    string public symbol;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;

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

    constructor(string memory _name, string memory _symbol, uint256 _initialSupply) {
        name = _name;
        symbol = _symbol;
        totalSupply = _initialSupply;
        balanceOf[msg.sender] = _initialSupply;
    }

    function transfer(address to, uint256 amount) public virtual returns (bool) {
        require(to != address(0), "Invalid address");
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");

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

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

// 子合约：扩展功能
contract MyToken is BaseToken {
    uint8 public constant decimals = 18;
    address public owner;

    constructor(
        string memory _name,
        string memory _symbol,
        uint256 _initialSupply
    ) BaseToken(_name, _symbol, _initialSupply) {
        owner = msg.sender;
    }

    // 添加mint功能
    function mint(address to, uint256 amount) public {
        require(msg.sender == owner, "Only owner");
        totalSupply += amount;
        balanceOf[to] += amount;
        emit Transfer(address(0), to, amount);
    }
```

```
    // 添加burn功能
    function burn(uint256 amount) public {
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");
        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;
        emit Transfer(msg.sender, address(0), amount);
    }
}
```

继承关系：

```
BaseToken（父合约）
    ↓ 继承
MyToken（子合约）

MyToken拥有：
├─ BaseToken的功能
│  ├─ name, symbol, totalSupply, balanceOf
│  └─ transfer()
└─ MyToken自己的功能
   ├─ decimals, owner
   └─ mint(), burn()
```

---

3. 多重继承

3.1 多重继承基础

Solidity支持多重继承，一个合约可以同时继承多个父合约。

基本语法：

```
contract Child is Parent1, Parent2, Parent3 {
    // 同时继承多个父合约
}
```

继承顺序：

- 从左到右列出父合约
- 顺序很重要
- 影响super调用和函数解析

基础示例：

```
contract Parent1 {
    function foo() public virtual returns (string memory) {
        return "Parent1";
    }
}
```

```
contract Parent2 {
    function bar() public virtual returns (string memory) {
        return "Parent2";
    }
}

// 多重继承
contract Child is Parent1, Parent2 {
    // 自动获得foo()和bar()

    function test() public view returns (string memory, string memory) {
        return (foo(), bar());
    }
}
```

继承关系图：

```
        Child
         /\
        /  \
   Parent1  Parent2
```

3.2 C3线性化算法

Solidity使用C3线性化算法（C3 Linearization）确定继承顺序。

基本规则：

1. **从右到左**：`Child is A, B, C`，C最优先，A最后（Solidity中继承列表写法是从基类到派生类）
2. **深度优先**：子合约覆盖父合约
3. **保持单调性**：不能打乱已有的继承关系

简单示例：

```
继承声明：contract C is A, B

继承顺序（解析顺序）：C → B → A

调用链：
C.foo()
  → B.foo() (B在右侧，覆盖A)
    → A.foo()
```

复杂示例：

```
contract GrandParent {
    function identify() public virtual returns (string memory) {
        return "GrandParent";
    }
}
```

```
contract Parent1 is GrandParent {
    function identify() public virtual override returns (string memory) {
        return "Parent1";
    }
}

contract Parent2 is GrandParent {
    function identify() public virtual override returns (string memory) {
        return "Parent2";
    }
}

contract Child is Parent1, Parent2 {
    function identify() public override(Parent1, Parent2) returns (string memory) {
        return "Child";
    }
}
```

继承顺序分析：

```
继承声明：
Child is Parent1, Parent2
Parent1 is GrandParent
Parent2 is GrandParent

C3线性化结果：
Child → Parent2 → Parent1 → GrandParent

原因：
1. Child 继承 Parent1 和 Parent2。
2. 在 Solidity 的继承列表中，Parent2（右侧）比 Parent1（左侧）更具体（Derived），解析时优先于 Parent1。
3. 顺序为：先 Child，再 Parent2，再 Parent1，最后 GrandParent。
```

继承顺序的重要性：

```
继承顺序不同，结果不同：

contract A is B, C { }  // C优先于B（C在右侧，更派生）
contract A is C, B { }  // B优先于C（B在右侧，更派生）

如果B和C有同名函数，右侧的合约优先被调用
```

3.3 实际应用：组合功能模块

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

// 模块1：所有权管理
contract Ownable {
```

```
    address public owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    constructor() {
        owner = msg.sender;
    }

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

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Invalid address");
        address oldOwner = owner;
        owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}

// 模块2：暂停功能
contract Pausable {
    bool public paused;

    event Paused(address account);
    event Unpaused(address account);

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

    modifier whenPaused() {
        require(paused, "Pausable: not paused");
        _;
    }

    function _pause() internal whenNotPaused {
        paused = true;
        emit Paused(msg.sender);
    }

    function _unpause() internal whenPaused {
        paused = false;
        emit Unpaused(msg.sender);
    }
}

// 组合合约：同时拥有两个功能
contract MyToken is Ownable, Pausable {
    mapping(address => uint256) public balanceOf;
```

```
    function transfer(address to, uint256 amount)
        public
        whenNotPaused  // 使用Pausable的修饰符
        returns (bool)
    {
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }

    function pause() public onlyOwner {  // 使用Ownable的修饰符
        _pause();  // 调用Pausable的内部函数
    }

    function unpause() public onlyOwner {
        _unpause();
    }
}
```

模块化设计的优势：

1. **关注点分离**：每个模块只负责一个功能
2. **可组合性**：像积木一样组合功能
3. **可测试性**：每个模块独立测试
4. **可复用性**：模块可以在多个项目中使用

---

4. super关键字

4.1 super的作用

super关键字用于调用父合约的函数，即使子合约已经重写了该函数。

基本概念：

```
contract Parent {
    function greet() public virtual returns (string memory) {
        return "Hello from Parent";
    }
}

contract Child is Parent {
    function greet() public override returns (string memory) {
        // 调用父合约的greet()
        string memory parentGreeting = super.greet();
        return string.concat("Hello from Child, ", parentGreeting);
    }
}
```

调用过程：

```
Child.greet()
    ↓
获取super.greet()的返回值："Hello from Parent"
    ↓
拼接："Hello from Child, Hello from Parent"
    ↓
返回最终结果
```

4.2 单继承中的super

在单继承中，super指向唯一的父合约。

```
contract Counter {
    uint256 public count;

    function increment() public virtual {
        count += 1;
    }
}

contract DoubleCounter is Counter {
    function increment() public override {
        // 先调用父合约的increment（+1）
        super.increment();
        // 再自己加一次（再+1）
        count += 1;
        // 最终效果：每次+2
    }
}
```

使用场景：

场景1：扩展父合约功能

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

    function transfer(address to, uint256 amount) public virtual returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }
}

contract TokenWithFee is BaseToken {
    uint256 public constant FEE = 10;  // 1%
    address public feeCollector;
```

```
    constructor(address _feeCollector) {
        feeCollector = _feeCollector;
    }

    function transfer(address to, uint256 amount) public override returns (bool) {
        uint256 fee = amount * FEE / 1000;
        uint256 amountAfterFee = amount - fee;

        // 收取手续费
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amountAfterFee;
        balanceOf[feeCollector] += fee;

        return true;
    }
}
```

场景2：添加检查逻辑

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

    function transfer(address to, uint256 amount) public virtual returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }
}

contract SafeTransfer is BasicTransfer {
    function transfer(address to, uint256 amount) public override returns (bool) {
        // 添加额外检查
        require(to != address(0), "Cannot transfer to zero address");
        require(amount > 0, "Amount must be positive");

        // 调用父合约的transfer
        return super.transfer(to, amount);
    }
}
```

4.3 多重继承中的super

在多重继承中，super不是指向单个父合约，而是按照继承顺序调用下一个合约。

关键理解：

```
contract A {
    function foo() public virtual returns (string memory) {
        return "A";
    }
}
```

```
contract B is A {
    function foo() public virtual override returns (string memory) {
        return string.concat("B->", super.foo());
    }
}

contract C is A {
    function foo() public virtual override returns (string memory) {
        return string.concat("C->", super.foo());
    }
}

contract D is B, C {
    function foo() public override(B, C) returns (string memory) {
        return string.concat("D->", super.foo());
    }
}
```

调用链分析：

1）、Solidity 从右到左处理继承列表。
2）、C 在继承列表中更靠右，所以在 D 之后首先出现。
3）、B 在 C 之后。
4）、A 是共同的基类，放在最后。

```
D.foo() 执行
↓
super.foo() → 指向 C（线性化顺序的下一个）
↓
C.foo() 执行
↓
C 中的 super.foo() → 指向 B（不是 A！）
↓
B.foo() 执行
↓
B 中的 super.foo() → 指向 A
↓
A.foo() 返回 "A"
↓
B.foo() 返回 "B->A"
↓
C.foo() 返回 "C->B->A"
↓
D.foo() 返回 "D->C->B->A"
```

重要提示：
super 的含义：在多重继承中，super 不是指"直接父合约"，而是指C3 线性化顺序中的下一个合约。
在 D 的上下文中，C 虽然直接继承自 A，但 C 的 super 会指向 B
这是因为在整个继承链中，C 的下一个是 B，而不是直接跳到 A

---

5. 构造函数继承

5.1 构造函数执行顺序

构造函数总是按照继承顺序执行，从父到子。

执行规则：

1. **父合约优先**：所有父合约构造函数先执行
2. **按继承顺序**：从左到右
3. **最后是子合约**：子合约构造函数最后执行

示例：

```
contract A {
    uint256 public valueA;

    constructor() {
        valueA = 1;
        // 第1个执行
    }
}

contract B {
    uint256 public valueB;

    constructor() {
        valueB = 2;
        // 第2个执行
    }
}

contract C is A, B {
    uint256 public valueC;

    constructor() {
        valueC = 3;
        // 第3个执行
    }
}
```

执行顺序：

```
部署C合约
    ↓
1. A的构造函数执行（valueA = 1）
    ↓
2. B的构造函数执行（valueB = 2）
    ↓
3. C的构造函数执行（valueC = 3）
    ↓
完成
```

验证执行顺序：

```
contract A {
    event Log(string message);

    constructor() {
        emit Log("A constructor");
    }
}

contract B {
    event Log(string message);

    constructor() {
        emit Log("B constructor");
    }
}

contract C is A, B {
    event Log(string message);

    constructor() {
        emit Log("C constructor");
    }
}

// 部署C后，查看事件日志：
// 1. "A constructor"
// 2. "B constructor"
// 3. "C constructor"
```

5.2 构造函数参数传递

当父合约的构造函数需要参数时，有两种传递方式。

方式1：在继承声明时传递（固定值）

```
contract Parent {
    uint256 public value;
```

```
    constructor(uint256 _value) {
        value = _value;
    }
}

// 在继承声明时传递固定值
contract Child is Parent(100) {
    constructor() {
        // Parent构造函数接收100
    }
}
```

特点：

- 适合固定值
- 代码简洁
- 不够灵活

方式2：在子构造函数中传递（动态值）

```
contract Parent {
    uint256 public value;

    constructor(uint256 _value) {
        value = _value;
    }
}

// 在子构造函数中传递动态值
contract Child is Parent {
    constructor(uint256 _value) Parent(_value) {
        // 通过参数传递给Parent
    }
}
```

特点：

- 更灵活
- 可以传递动态值
- 推荐使用

综合示例：

```
contract Base {
    string public name;
    uint256 public version;

    constructor(string memory _name, uint256 _version) {
        name = _name;
        version = _version;
    }
}
```

```
contract Extended is Base {
    address public creator;

    constructor(
        string memory _name,
        uint256 _version
    ) Base(_name, _version) {
        creator = msg.sender;
    }
}
```

5.3 多重继承的构造函数

多个父合约都需要参数时，必须全部初始化。

```
contract A {
    uint256 public valueA;
    constructor(uint256 _a) {
        valueA = _a;
    }
}

contract B {
    uint256 public valueB;
    constructor(uint256 _b) {
        valueB = _b;
    }
}

contract C is A, B {
    uint256 public valueC;

    // 方式1：混合传递
    constructor(uint256 _a, uint256 _c)
        A(_a)         // 动态传递给A
        B(200)        // 固定值传递给B
    {
        valueC = _c;
    }

    // 方式2：全部动态传递（推荐）
    constructor(uint256 _a, uint256 _b, uint256 _c)
        A(_a)
        B(_b)
    {
        valueC = _c;
    }
}
```

执行顺序保持不变：

无论如何传递参数，执行顺序始终是：
`A() → B() → C()`

---

6. 函数重写

6.1 virtual和override关键字

函数重写（Function Overriding）允许子合约修改父合约函数的行为。

基本规则：

1. **父合约**：函数必须标记为`virtual`
2. **子合约**：重写函数必须标记为`override`
3. **两者必须配对**：缺一不可

基础示例：

```
contract Parent {
    // virtual：表示这个函数可以被重写
    function getValue() public virtual returns (uint256) {
        return 100;
    }
}

contract Child is Parent {
    // override：表示重写父合约的函数
    function getValue() public override returns (uint256) {
        return 200;  // 修改返回值
    }
}
```

调用结果：

```
Parent parent = new Parent();
parent.getValue();  // 返回：100

Child child = new Child();
child.getValue();   // 返回：200（已重写）
```

6.2 函数签名必须匹配

重写函数必须与父合约函数签名完全一致。

必须相同的部分：

- 函数名
- 参数类型
- 参数顺序
- 返回类型

可以不同的部分：

- 可见性（可以更开放，不能更严格）
- 状态修饰符（可以更严格，不能更宽松）

示例：

```
contract Parent {
    function foo(uint256 a) public virtual returns (uint256) {
        return a;
    }
}

contract Child is Parent {
    // 正确：签名完全相同
    function foo(uint256 a) public override returns (uint256) {
        return a * 2;
    }

    // 错误：参数不同
    // function foo(uint256 a, uint256 b) public override returns (uint256) {
    //     return a + b;
    // }

    // 错误：返回类型不同
    // function foo(uint256 a) public override returns (string memory) {
    //     return "hello";
    // }
}
```

可见性规则：

```
contract Parent {
    function foo() internal virtual returns (uint256) {
        return 1;
    }
}

contract Child is Parent {
    // 正确：internal → public（更开放）
    function foo() public override returns (uint256) {
        return 2;
    }

    // 错误：internal → private（更严格）
    // function foo() private override returns (uint256) {
    //     return 2;
    // }
}
```

6.3 多重继承中的override

当多个父合约有同名函数时，必须明确指定重写哪些。

```
contract A {
    function foo() public virtual returns (string memory) {
        return "A";
    }
}

contract B {
    function foo() public virtual returns (string memory) {
        return "B";
    }
}

contract C is A, B {
    // 必须明确指定：override(A, B)
    function foo() public override(A, B) returns (string memory) {
        return "C";
    }

    // 错误：不明确
    // function foo() public override returns (string memory) {
    //     return "C";
    // }
}
```

语法规则：

```
// 单继承：简单override
function foo() public override returns (string memory) { }

// 多重继承：明确指定
function foo() public override(Parent1, Parent2) returns (string memory) { }

// 如果继续被继承，还要加virtual
function foo() public virtual override(Parent1, Parent2) returns (string memory) { }
```

6.4 使用super在重写中调用父函数

```
contract Logger {
    event Log(string message);

    function log(string memory message) public virtual {
        emit Log(message);
    }
}

contract TimestampLogger is Logger {
    function log(string memory message) public override {
        // 先调用父合约的log
```

```
        super.log(message);

        // 再添加时间戳日志
        emit Log(
            string.concat(
                "Timestamp: ",
                uint2str(block.timestamp)
            )
        );
    }

    function uint2str(uint _i) internal pure returns (string memory) {
        if (_i == 0) return "0";
        uint j = _i;
        uint len;
        while (j != 0) {
            len++;
            j /= 10;
        }
        bytes memory bstr = new bytes(len);
        uint k = len;
        while (_i != 0) {
            k = k-1;
            uint8 temp = (48 + uint8(_i - _i / 10 * 10));
            bytes1 b1 = bytes1(temp);
            bstr[k] = b1;
            _i /= 10;
        }
        return string(bstr);
    }
}
```

7. 抽象合约

7.1 什么是抽象合约

抽象合约（Abstract Contract）是包含至少一个未实现函数的合约。

定义语法：

```
abstract contract 合约名 {
    // 至少一个未实现的函数
}
```

基本示例：

```
abstract contract Animal {
    // 抽象函数：只有声明，没有实现
    function makeSound() public virtual returns (string memory);
```

```
    // 普通函数：可以有实现
    function sleep() public pure returns (string memory) {
        return "Zzz...";
    }

    // 可以有状态变量
    uint256 public age;
}

// 实现抽象合约
contract Dog is Animal {
    // 必须实现makeSound
    function makeSound() public pure override returns (string memory) {
        return "Woof!";
    }
}

contract Cat is Animal {
    function makeSound() public pure override returns (string memory) {
        return "Meow!";
    }
}
```

抽象合约的特点：

1. **不能直接部署**：必须被继承
2. **可以有实现**：部分函数可以有实现
3. **可以有状态变量**：可以定义状态变量
4. **可以有构造函数**：可以有构造函数
5. **强制实现**：子合约必须实现所有抽象函数

7.2 抽象合约的使用场景

场景1：定义基础框架

```
abstract contract BaseToken {
    string public name;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;

    constructor(string memory _name, uint256 _supply) {
        name = _name;
        totalSupply = _supply;
    }

    // 抽象函数：每个代币的转账逻辑可能不同
    function transfer(address to, uint256 amount)
        public virtual returns (bool);

    // 普通函数：查询余额逻辑相同
```

```
    function getBalance(address account) public view returns (uint256) {
        return balanceOf[account];
    }
}

// 标准代币：简单转账
contract StandardToken is BaseToken {
    constructor(string memory _name, uint256 _supply)
        BaseToken(_name, _supply)
    { }

    function transfer(address to, uint256 amount)
        public override returns (bool)
    {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
        return true;
    }
}

// 带手续费代币：转账扣手续费
contract FeeToken is BaseToken {
    uint256 public constant FEE = 100;  // 1%

    constructor(string memory _name, uint256 _supply)
        BaseToken(_name, _supply)
    { }

    function transfer(address to, uint256 amount)
        public override returns (bool)
    {
        uint256 fee = amount * FEE / 10000;
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount - fee;
        balanceOf[address(this)] += fee;
        return true;
    }
}
```

场景2：强制实现特定功能

```
abstract contract Mintable {
    // 强制子合约实现mint功能
    function mint(address to, uint256 amount) public virtual;
}

abstract contract Burnable {
    // 强制子合约实现burn功能
    function burn(uint256 amount) public virtual;
}
```

```
contract FullToken is Mintable, Burnable {
    mapping(address => uint256) public balanceOf;

    // 必须实现mint
    function mint(address to, uint256 amount) public override {
        balanceOf[to] += amount;
    }

    // 必须实现burn
    function burn(uint256 amount) public override {
        balanceOf[msg.sender] -= amount;
    }
}
```

7.3 抽象合约 vs 普通合约

特性	普通合约	抽象合约
关键字	contract	abstract contract
可部署	是	否
未实现函数	不允许	允许
状态变量	允许	允许
构造函数	允许	允许
使用场景	完整实现	定义规范

---

8. 接口

8.1 什么是接口

接口（Interface）是纯粹的接口定义，只声明函数签名，不包含任何实现。

定义语法：

```
interface 接口名 {
    // 只有函数声明
}
```

基本示例：

```
interface ICounter {
    // 所有函数必须是external
    function getCount() external view returns (uint256);
    function increment() external;
    function decrement() external;
```

```
    // 可以定义事件
    event CountChanged(uint256 newCount);
}

// 实现接口
contract Counter is ICounter {
    uint256 private count;

    event CountChanged(uint256 newCount);

    function getCount() external view override returns (uint256) {
        return count;
    }

    function increment() external override {
        count++;
        emit CountChanged(count);
    }

    function decrement() external override {
        count--;
        emit CountChanged(count);
    }
}
```

8.2 接口的特点和限制

接口的特点：

1. **使用interface关键字**
2. **不能有实现**：所有函数都是声明
3. **不能有状态变量**：不能定义storage变量
4. **不能有构造函数**
5. **所有函数必须external**
6. **可以继承其他接口**
7. **可以定义事件**

接口vs合约对比：

特性	合约	抽象合约	接口
关键字	contract	abstract contract	interface
函数实现	必须全部实现	可以部分实现	不能有实现
函数可见性	任意	任意	必须external
状态变量	允许	允许	不允许
构造函数	允许	允许	不允许
可部署	是	否	否

8.3 ERC20接口标准

ERC20是最经典的接口定义示例。

```
interface IERC20 {
    // 查询函数
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function allowance(address owner, address spender)
        external view returns (uint256);

    // 操作函数
    function transfer(address to, uint256 amount) external returns (bool);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external returns (bool);

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

实现ERC20接口：

```
contract MyToken is IERC20 {
    string public name = "My Token";
    string public symbol = "MTK";
    uint8 public decimals = 18;

    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    constructor(uint256 initialSupply) {
```

```
        _totalSupply = initialSupply;
        _balances[msg.sender] = initialSupply;
    }

    function totalSupply() external view override returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) external view override returns (uint256) {
        return _balances[account];
    }

    function allowance(address owner, address spender)
        external view override returns (uint256)
    {
        return _allowances[owner][spender];
    }

    function transfer(address to, uint256 amount)
        external override returns (bool)
    {
        require(to != address(0), "Invalid address");
        require(_balances[msg.sender] >= amount, "Insufficient balance");

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

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

    function approve(address spender, uint256 amount)
        external override returns (bool)
    {
        require(spender != address(0), "Invalid spender");

        _allowances[msg.sender][spender] = amount;

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

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external override returns (bool) {
        require(from != address(0), "From zero");
        require(to != address(0), "To zero");
        require(_balances[from] >= amount, "Insufficient balance");
        require(_allowances[from][msg.sender] >= amount, "Insufficient allowance");
```

```
        _balances[from] -= amount;
        _balances[to] += amount;
        _allowances[from][msg.sender] -= amount;

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

8.4 接口用于合约交互

接口最重要的应用是合约间交互。

场景：合约A调用合约B

```
// 定义接口
interface IToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// 使用接口与其他合约交互
contract Exchanger {
    function swapTokens(address tokenAddress, address recipient, uint256 amount)
        public
    {
        // 通过接口与代币合约交互
        IToken token = IToken(tokenAddress);

        // 检查余额
        uint256 balance = token.balanceOf(address(this));
        require(balance >= amount, "Insufficient balance");

        // 执行转账
        bool success = token.transfer(recipient, amount);
        require(success, "Transfer failed");
    }
}
```

接口的优势：

1. **解耦**：不需要知道合约的完整代码
2. **标准化**：统一的接口规范
3. **互操作性**：不同合约可以互相调用
4. **节省gas**：不需要导入完整合约代码

---

9. 实战练习

练习1：实现完整的权限管理系统

需求：

创建一个模块化的权限管理系统：

1. Ownable合约：单一所有者管理
2. Pausable合约：暂停功能
3. MyContract：组合两个功能

参考答案：

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

contract Ownable {
    address public owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

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

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Invalid address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}

contract Pausable {
    bool public paused;

    event Paused(address account);
    event Unpaused(address account);

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

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

    function _pause() internal whenNotPaused {
```

```
        paused = true;
        emit Paused(msg.sender);
    }

    function _unpause() internal whenPaused {
        paused = false;
        emit Unpaused(msg.sender);
    }
}

contract MyContract is Ownable, Pausable {
    uint256 public value;

    function setValue(uint256 _value) public onlyOwner whenNotPaused {
        value = _value;
    }

    function pause() public onlyOwner {
        _pause();
    }

    function unpause() public onlyOwner {
        _unpause();
    }
}
```

练习2：实现动物抽象合约

需求：

1. 创建Animal抽象合约，定义makeSound抽象函数
2. 创建Dog和Cat子合约实现makeSound
3. 添加共同的eat函数

参考答案：

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

abstract contract Animal {
    string public species;

    constructor(string memory _species) {
        species = _species;
    }

    // 抽象函数：子合约必须实现
    function makeSound() public virtual returns (string memory);

    // 普通函数：所有动物共用
    function eat() public pure returns (string memory) {
        return "Eating...";
```

```
    }

    function sleep() public pure returns (string memory) {
        return "Sleeping...";
    }
}

contract Dog is Animal {
    constructor() Animal("Dog") { }

    function makeSound() public pure override returns (string memory) {
        return "Woof! Woof!";
    }
}

contract Cat is Animal {
    constructor() Animal("Cat") { }

    function makeSound() public pure override returns (string memory) {
        return "Meow! Meow!";
    }
}
```

练习3：使用OpenZeppelin创建代币

需求：

使用OpenZeppelin库创建一个完整的代币合约：

1. 继承ERC20
2. 继承Ownable
3. 添加mint功能

参考答案：

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

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract MyToken is ERC20, Ownable {
    constructor(uint256 initialSupply) ERC20("My Token", "MTK") Ownable(msg.sender) {
        _mint(msg.sender, initialSupply);
    }

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
}
```

10. 常见问题解答

Q1：什么时候使用单继承，什么时候使用多重继承?

答：根据功能模块数量选择。

单继承：

- 简单的扩展
- 线性关系
- 一个基础合约

多重继承：

- 组合多个功能模块
- 需要多个基础功能
- 模块化设计

Q2：private和internal在继承中有什么区别?

答：子合约的访问权限不同。

- **private**：子合约不能访问
- **internal**：子合约可以访问

推荐：如果希望子合约访问，使用internal。

Q3：super在多重继承中指向哪个合约?

答：指向继承链中的下一个合约，不是直接父合约。

按照C3线性化算法确定的顺序调用。

Q4：什么时候用抽象合约，什么时候用接口?

答：根据需求选择。

抽象合约：

- 需要部分实现
- 有状态变量
- 有构造函数
- 定义基础框架

接口：

- 纯接口定义
- 标准规范（ERC20、ERC721）
- 合约间交互
- 不需要实现

Q5：必须重写父合约的所有virtual函数吗?

答：不是必须的。

- virtual只是表示"可以"重写
- 子合约可以选择重写或不重写
- 只有abstract函数必须实现

Q6：多重继承时如何避免冲突?

答：遵循几个原则。

1. **明确指定override**：`override(A, B)`
2. **调用super链**：让所有父合约都执行
3. **合理设计继承顺序**：最通用的在最右边
4. **避免菱形继承**：尽量简化继承关系

Q7：继承会增加Gas成本吗?

答：部署时会增加，执行时不会。

部署成本：

- 继承的代码会包含在子合约中
- 字节码更大，部署成本更高

执行成本：

- 执行时与普通函数相同
- 没有额外开销
- 可能因为代码复用反而更优化

---

11. 知识点总结

继承基础

单继承：

```
contract Child is Parent { }
```

多重继承：

```
contract Child is Parent1, Parent2 { }
```

访问权限：

- public：子合约可访问
- internal：子合约可访问
- private：子合约不可访问

super关键字

作用：

- 调用父合约函数
- 按继承链顺序调用
- 不是指向直接父合约

使用场景：

- 扩展父合约功能
- 调用链
- 多重继承中的函数调用

构造函数继承

执行顺序：

- 父合约优先
- 从左到右
- 最后是子合约

参数传递：

```
// 方式1：固定值
contract Child is Parent(100) { }

// 方式2：动态值（推荐）
contract Child is Parent {
    constructor(uint v) Parent(v) { }
}
```

函数重写

关键字：

- virtual：可以被重写
- override：重写父合约函数

规则：

- 签名必须相同
- 多重继承需明确指定：`override(A, B)`
- 可见性可以更开放

抽象合约

特点：

- abstract关键字
- 可以有未实现函数
- 不能部署
- 可以有状态变量

使用场景：

- 定义基础框架
- 部分实现
- 强制子合约实现特定功能

接口

特点：

- interface关键字
- 所有函数external
- 不能有实现
- 不能有状态变量

使用场景：

- 标准规范（ERC20、ERC721）
- 合约间交互
- 纯接口定义

---

12. 学习检查清单

完成本课后，你应该能够：

继承基础：

- [ ] 理解继承的作用和优势
- [ ] 会使用单继承语法
- [ ] 会使用多重继承语法
- [ ] 理解继承的访问权限

super关键字：

- [ ] 理解super的作用
- [ ] 会在单继承中使用super
- [ ] 理解多重继承中的super调用链
- [ ] 会使用super扩展父合约功能

构造函数：

- [ ] 理解构造函数执行顺序
- [ ] 会传递构造函数参数
- [ ] 会处理多重继承的构造函数

函数重写：

- [ ] 理解virtual和override的配对使用
- [ ] 会重写父合约函数
- [ ] 会在多重继承中明确指定override

- [ ] 理解函数签名匹配规则

抽象合约和接口：

- [ ] 会定义和使用抽象合约
- [ ] 会定义和使用接口
- [ ] 理解两者的区别
- [ ] 会实现ERC20接口

---

13. 下一步学习

完成本课后，建议：

1. **实践所有示例代码**：在Remix中测试继承关系
2. **研究OpenZeppelin**：学习专业的合约继承设计
3. **实现完整项目**：创建带继承的代币合约
4. **准备学习第6.2课**：库合约

下节课预告：第6.2课 - 库合约Library

我们将学习：

- 库合约的定义和特性
- using for语法
- 内部库和外部库
- OpenZeppelin库的使用
- 库合约的最佳实践

---

14. 扩展资源

官方文档：

- Solidity继承：https://docs.soliditylang.org/en/latest/contracts.html#inheritance
- 抽象合约：https://docs.soliditylang.org/en/latest/contracts.html#abstract-contracts
- 接口：https://docs.soliditylang.org/en/latest/contracts.html#interfaces

学习资源：

- Solidity by Example - Inheritance：https://solidity-by-example.org/inheritance/
- OpenZeppelin Contracts：https://github.com/OpenZeppelin/openzeppelin-contracts

实战项目：
研究OpenZeppelin的继承设计：

- ERC20.sol
- Ownable.sol
- Pausable.sol
- ReentrancyGuard.sol

工具推荐：

- Remix IDE：测试继承关系
- Hardhat：专业开发框架
- OpenZeppelin Contracts Wizard：自动生成合约