Solidity智能合约开发知识

第4.2课：特殊类型与全局变量

学习目标：深入理解address类型和转账方法、掌握全局变量的使用、学会枚举类型和状态机模式、理解constant和immutable的Gas优化

预计学习时间：2小时

难度等级：入门进阶

目录

1. address类型深入
2. 三种转账方式
3. 全局变量详解
4. 枚举类型
5. constant和immutable
6. 综合实战案例

---

1. address类型深入

1.1 address类型概述

address是Solidity特有的20字节类型，用于存储以太坊地址（160位）。

地址格式：

```
0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb1
```

地址的组成：

- 以`0x`开头
- 后跟40个十六进制字符（每个字符4位，共160位）
- 总长度：42个字符

地址的来源：

```
contract AddressSources {
    // 1. 用户账户地址（EOA）
    address user = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;

    // 2. 合约地址
    address contractAddr = address(this);

    // 3. 全局变量
    address caller = msg.sender;

    // 4. 计算得出
    address predicted = address(uint160(uint(keccak256(...))));
}
```

1.2 address vs address payable

Solidity提供了两种address类型，它们有重要区别。

基本对比：

特性	address	address payable
存储任何地址	支持	支持
查询余额(.balance)	支持	支持
查询代码(.code)	支持	支持
接收以太币	不支持	支持
transfer方法	不支持	支持
send方法	不支持	支持
call方法	支持	支持

代码示例：

```
contract AddressTypes {
    // 普通address：用于存储地址
    address public owner;
    address public contractAddress;

    // address payable：用于接收以太币
    address payable public recipient;
    address payable public treasury;

    constructor() {
        owner = msg.sender;  // msg.sender是address类型
        contractAddress = address(this);
    }
```

```
    function setRecipient(address _recipient) public {
        // 转换为payable类型
        recipient = payable(_recipient);
    }
}
```

为什么区分两种类型?

安全性考虑:

编译器可以在编译时检查类型，防止给不能接收以太币的地址转账。

```
contract SafetyDemo {
    address normalAddr;
    address payable payableAddr;

    function attemptTransfer() public {
        // 编译错误：address没有transfer方法
        // normalAddr.transfer(1 ether);

        // 正确：只有address payable有transfer方法
        payableAddr.transfer(1 ether);
    }
}
```

类型转换:

```
contract AddressConversion {
    // address → address payable
    function toPayable(address addr) public pure returns (address payable) {
        return payable(addr);
    }

    // address payable → address（自动转换）
    function toNormal(address payable addr) public pure returns (address) {
        return addr;  // 自动转换，无需显式转换
    }

    // 实际使用
    function sendEther(address recipient, uint amount) public {
        address payable payableRecipient = payable(recipient);
        payableRecipient.transfer(amount);
    }
}
```

1.3 address属性和方法

balance - 查询余额

```
```

```
contract BalanceQuery {
    // 查询任何地址的余额
    function getBalance(address account) public view returns (uint) {
        return account.balance;  // 单位：wei
    }

    // 查询合约自己的余额
    function getContractBalance() public view returns (uint) {
        return address(this).balance;
    }

    // 查询调用者余额
    function getMyBalance() public view returns (uint) {
        return msg.sender.balance;
    }

    // 余额比较
    function hasEnoughBalance(address account, uint required)
        public view returns (bool)
    {
        return account.balance >= required;
    }
}
```

重要提示：

- balance的单位是wei
- 1 ether = 10^18 wei
- balance是实时的，随时可能变化

code - 获取代码

```
contract CodeCheck {
    // 检查地址是否是合约
    function isContract(address account) public view returns (bool) {
        return account.code.length > 0;
    }

    // 获取合约字节码
    function getCode(address account) public view returns (bytes memory) {
        return account.code;
    }

    // 获取代码哈希
    function getCodeHash(address account) public view returns (bytes32) {
        return account.codehash;
    }
}
```

应用场景：

```
contract SecurityCheck {
    // 防止合约调用（只允许EOA）
    function onlyEOA() public view {
        require(msg.sender.code.length == 0, "Contracts not allowed");
        // 只有外部账户（EOA）代码长度为0
    }

    // 确保是合约地址
    function onlyContract(address target) public view {
        require(target.code.length > 0, "Not a contract");
    }
}
```

注意事项：

在构造函数中，合约的code.length仍然是0，所以这个检查不是完全可靠的。

---

2. 三种转账方式

2.1 transfer - 最安全的转账

语法：

```
recipient.transfer(amount);
```

特点：

- 失败自动回退（交易失败）
- 固定2300 gas限制
- 最安全的转账方式
- 可能gas不足导致失败

示例：

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

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
        // 如果transfer失败，整个交易回滚
    }

    function sendToOwner(address payable owner) public {
        owner.transfer(1 ether);
    }
}
```

优点：

- 简单易用
- 失败自动回滚，安全
- 不需要检查返回值

缺点：

- 2300 gas限制可能不够
- 接收方如果是复杂合约可能失败
- 无法处理失败情况

2.2 send - 需要检查返回值

语法：

```
bool success = recipient.send(amount);
```

特点：

- 返回bool表示成功/失败
- 固定2300 gas限制
- 失败不会自动回滚
- 容易忘记检查返回值

示例：

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

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;

        bool success = payable(msg.sender).send(amount);
        require(success, "Transfer failed");
        // 必须检查返回值!
    }
}
```

优点：

- 可以处理失败情况
- 有返回值

缺点：

- 容易忘记检查返回值（危险）
- 2300 gas限制
- 不推荐使用

危险示例：

```
contract DangerousSend {
    function badWithdraw(uint amount) public {
        balances[msg.sender] -= amount;

        // 危险：忘记检查返回值
        payable(msg.sender).send(amount);
        // 如果send失败，余额已扣除但ETH没发送
        // 资金永久丢失！
    }
}
```

2.3 call - 最灵活的方式

语法：

```
(bool success, bytes memory data) = recipient.call{value: amount}("");
```

特点：

- 无gas限制（转发所有可用gas）
- 最灵活
- 必须检查返回值
- 有重入攻击风险

示例：

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

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

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

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
        // 必须检查返回值！
    }

    // 指定gas数量
    function withdrawWithGasLimit(uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;

        (bool success, ) = msg.sender.call{value: amount, gas: 10000}("");
        require(success, "Transfer failed");
    }
```

```
}
```

优点：

- 没有gas限制，更灵活
- 可以传递数据
- 可以调用其他函数
- 最推荐的转账方式（配合安全措施）

缺点：

- 必须手动检查返回值
- 有重入攻击风险
- 需要遵循CEI模式

2.4 三种方式对比

方法	gas限制	失败处理	推荐度	使用场景
transfer	2300	自动回滚	中等	简单转账
send	2300	返回false	低	不推荐
call	无限制	返回false	高	配合ReentrancyGuard

当前推荐：

2024年后推荐使用：**call + ReentrancyGuard**

```
contract RecommendedPattern {
    bool private locked;

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

    function withdraw(uint amount) public noReentrant {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // Effects
        balances[msg.sender] -= amount;

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

2.5 转账安全最佳实践

CEI模式（Checks-Effects-Interactions）

这是智能合约安全的黄金法则！

```
function withdraw(uint amount) public {
    // 1. Checks - 检查所有条件
    require(balances[msg.sender] >= amount, "Insufficient balance");
    require(amount > 0, "Amount must be positive");

    // 2. Effects - 更新状态变量
    balances[msg.sender] -= amount;

    // 3. Interactions - 调用外部合约/转账
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

危险示例（重入攻击）：

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

    // 危险：先转账后更新
    function badWithdraw() public {
        uint amount = balances[msg.sender];

        // Interactions（外部调用在前）
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed");

        // Effects（状态更新在后）- 太晚了！
        balances[msg.sender] = 0;
        // 攻击者可以在收到钱后再次调用withdraw
    }
}
```

攻击过程：

```
1. 攻击者调用withdraw
2. 合约向攻击者转账
3. 攻击者合约的fallback被触发
4. 在fallback中再次调用withdraw
5. 因为余额还没清零，可以再次提取
6. 重复步骤2-5，直到合约被掏空
```

安全做法：

```
contract Safe {
```

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

    // 安全：先更新后转账
    function safeWithdraw() public {
        uint amount = balances[msg.sender];
        require(amount > 0, "No balance");

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

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

其他安全措施：

```
// 1. 使用ReentrancyGuard
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureContract is ReentrancyGuard {
    function withdraw() public nonReentrant {
        // 自动防重入
    }
}

// 2. 零地址检查
function sendEther(address payable to, uint amount) public {
    require(to != address(0), "Cannot send to zero address");
    to.transfer(amount);
}

// 3. 余额检查
function sendEther(address payable to, uint amount) public {
    require(address(this).balance >= amount, "Insufficient contract balance");
    to.transfer(amount);
}
```

---

3. 全局变量详解

3.1 全局变量概览

全局变量是Solidity内置的特殊变量，提供区块链、交易、调用的信息。

三大类别：

msg对象 - 消息/调用信息：

- `msg.sender`：调用者地址（最常用）

- `msg.value`：发送的ETH数量（最常用）
- `msg.data`：完整调用数据
- `msg.sig`：函数选择器

block对象 - 区块信息：

- `block.timestamp`：当前区块时间戳（常用）
- `block.number`：当前区块号（常用）
- `block.gaslimit`：区块gas限制
- `block.coinbase`：矿工/验证者地址
- `blockhash(n)`：获取区块哈希

tx对象 - 交易信息：

- `tx.origin`：交易发起者（危险，不要用于权限检查）
- `tx.gasprice`：交易gas价格

其他重要函数：

- `gasleft()`：剩余gas
- `keccak256()`：哈希函数
- `abi.encode()`：编码函数

3.2 msg.sender - 调用者地址

msg.sender是最重要的全局变量，几乎每个合约都会用到。

基本定义：

- 类型：`address`
- 含义：当前函数的直接调用者
- 可以是：外部账户（EOA）或合约地址

核心用途：权限控制

```
contract Ownable {
    address public owner;

    constructor() {
        owner = msg.sender;  // 部署者成为owner
    }

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

    function sensitiveOperation() public onlyOwner {
        // 只有owner可以执行
    }
}
```

其他用途：

```
contract MsgSenderUses {
    mapping(address => uint) public balances;
    mapping(address => bool) public registered;

    // 记录操作者
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // 身份验证
    function register() public {
        require(!registered[msg.sender], "Already registered");
        registered[msg.sender] = true;
    }

    // 转账发送方
    function transfer(address to, uint amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

重要理解：调用链中的msg.sender

```
contract A {
    function callB(address b) public {
        B(b).someFunction();
    }
}

contract B {
    function someFunction() public view returns (address) {
        return msg.sender;  // 返回合约A的地址，不是用户地址
    }
}
```

调用链：`用户 → 合约A → 合约B`

在合约B中：

- `msg.sender = 合约A的地址`（直接调用者）
- 不是用户地址

3.3 msg.value - 发送的ETH数量

基本定义：

- 类型：`uint`（单位：wei）
- 含义：随调用发送的以太币数量
- 只在payable函数中有意义

基本用法：

```
contract PaymentContract {
    uint public totalReceived;
    mapping(address => uint) public contributions;

    // 接收支付
    function contribute() public payable {
        require(msg.value > 0, "Must send ETH");

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

    // 精确金额要求
    function buyItem() public payable {
        require(msg.value == 0.1 ether, "Must send exactly 0.1 ETH");
        // 购买逻辑
    }

    // 最小金额要求
    function invest() public payable {
        require(msg.value >= 1 ether, "Minimum 1 ETH");
        // 投资逻辑
    }

    // 范围检查
    function donate() public payable {
        require(msg.value >= 0.01 ether, "Too low");
        require(msg.value <= 10 ether, "Too high");
        // 捐款逻辑
    }
}
```

重要提示：

1. **单位是wei**：`1 ether = 10^18 wei`
2. **只有payable函数可接收**：非payable函数msg.value必须为0
3. **自动增加余额**：msg.value会自动加到合约余额

常见错误：

```
contract CommonMistakes {
    // 错误1：非payable函数尝试接收ETH
    // function deposit() public {
    //     // 如果调用时发送ETH，交易会失败
    // }

    // 错误2：在非payable函数中访问msg.value
    // function getValue() public view returns (uint) {
    //     return msg.value;  // 编译错误!
```

```
    // }

    // 正确：payable函数
    function correctDeposit() public payable {
        // 可以接收ETH
        // 可以访问msg.value
    }
}
```

3.4 block.timestamp - 时间戳

基本定义：

- 类型：`uint`
- 单位：秒（Unix时间戳）
- 含义：当前区块被打包的时间

时间单位：

Solidity提供了方便的时间单位：

```
1 seconds = 1
1 minutes = 60 seconds = 60
1 hours = 60 minutes = 3600
1 days = 24 hours = 86400
1 weeks = 7 days = 604800
```

注意：没有`months`和`years`，因为它们的长度不固定。

使用示例：

```
contract TimeExample {
    uint public deadline;
    uint public startTime;

    constructor(uint durationDays) {
        startTime = block.timestamp;
        deadline = block.timestamp + (durationDays * 1 days);
    }

    // 检查是否过期
    function isExpired() public view returns (bool) {
        return block.timestamp > deadline;
    }

    // 检查是否在有效期内
    function isActive() public view returns (bool) {
        return block.timestamp >= startTime &&
               block.timestamp <= deadline;
    }

    // 剩余时间
```

```
    function timeRemaining() public view returns (uint) {
        if (block.timestamp >= deadline) {
            return 0;
        }
        return deadline - block.timestamp;
    }

    // 已经过时间
    function timePassed() public view returns (uint) {
        return block.timestamp - startTime;
    }
}
```

实战案例：时间锁

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

    // 存款并锁定
    function deposit(uint lockDuration) public payable {
        require(msg.value > 0, "Must deposit ETH");
        require(
            lockDuration >= 1 days && lockDuration <= 365 days,
            "Duration must be 1-365 days"
        );

        balances[msg.sender] += msg.value;
        lockedUntil[msg.sender] = block.timestamp + lockDuration;
    }

    // 提现
    function withdraw() public {
        require(balances[msg.sender] > 0, "No balance");
        require(
            block.timestamp >= lockedUntil[msg.sender],
            "Still locked"
        );

        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;
        lockedUntil[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Transfer failed");
    }

    // 查询剩余锁定时间
    function timeRemaining() public view returns (uint) {
        if (block.timestamp >= lockedUntil[msg.sender]) {
            return 0;
```

```
        }
        return lockedUntil[msg.sender] - block.timestamp;
    }
}
```

安全警告：

矿工可以操纵block.timestamp约15秒范围

适合使用：
- 较长时间间隔（小时、天）
- 时间锁
- 截止日期

不适合使用：
- 关键随机性
- 精确到秒的需求
- 高频交易时间戳

3.5 block.number - 区块号

基本定义：

- 类型：`uint`
- 含义：当前区块在链上的序号
- 以太坊主网：平均12-14秒/块，每天约6400个块

使用示例：

```
contract BlockNumber {
    uint public startBlock;
    uint public endBlock;

    constructor(uint durationInBlocks) {
        startBlock = block.number;
        endBlock = block.number + durationInBlocks;
    }

    // 检查是否在有效期内
    function isActive() public view returns (bool) {
        return block.number >= startBlock &&
               block.number <= endBlock;
    }

    // 剩余区块数
    function blocksRemaining() public view returns (uint) {
        if (block.number >= endBlock) {
            return 0;
        }
        return endBlock - block.number;
    }
```

```
    // 计算经过的区块数
    function blocksPassed() public view returns (uint) {
        return block.number - startBlock;
    }
}
```

基于区块的奖励系统：

```
contract BlockRewards {
    uint public constant BLOCKS_PER_DAY = 6400;
    uint public constant REWARD_PER_BLOCK = 10;

    uint public lastRewardBlock;
    mapping(address => uint) public stakes;
    mapping(address => uint) public rewards;

    function stake() public payable {
        require(msg.value > 0, "Must stake");
        stakes[msg.sender] += msg.value;
        lastRewardBlock = block.number;
    }

    function claimRewards() public {
        uint stakeAmount = stakes[msg.sender];
        require(stakeAmount > 0, "No stake");

        uint blocksPassed = block.number - lastRewardBlock;
        uint reward = blocksPassed * REWARD_PER_BLOCK;

        rewards[msg.sender] += reward;
        lastRewardBlock = block.number;
    }
}
```

3.6 tx.origin - 危险，不要用！

基本定义：

- 类型：`address`
- 含义：交易的原始发起者（必定是EOA，不可能是合约）

msg.sender vs tx.origin：

```
调用链：用户 → 合约A → 合约B

在合约B中：
- msg.sender = 合约A（直接调用者）
- tx.origin = 用户（交易发起者）
```

危险案例：钓鱼攻击

```
// 受害合约（有漏洞）
contract Vulnerable {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // 危险：使用tx.origin检查权限
    function transferOwnership(address newOwner) public {
        require(tx.origin == owner, "Not owner");  // 漏洞！
        owner = newOwner;
    }
}

// 攻击合约
contract Attack {
    Vulnerable public victim;
    address public attacker;

    constructor(address _victim) {
        victim = Vulnerable(_victim);
        attacker = msg.sender;
    }

    function attack() public {
        // 转移所有权到攻击者
        victim.transferOwnership(attacker);
    }
}

// 攻击流程：
// 1. 攻击者诱导owner访问恶意网站
// 2. owner点击按钮，调用Attack.attack()
// 3. Attack调用Vulnerable.transferOwnership()
// 4. 在Vulnerable中，tx.origin是owner，检查通过！
// 5. owner被修改为attacker
// 6. 合约被攻击者控制
```

正确做法：

```
contract Safe {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // 安全：使用msg.sender
    function transferOwnership(address newOwner) public {
        require(msg.sender == owner, "Not owner");  // 安全
        owner = newOwner;
    }
}
```

tx.origin的唯一合法用途：

检查调用链中是否包含EOA（很少使用）

```
function isOriginEOA() public view returns (bool) {
    return tx.origin == msg.sender;
    // true：直接由EOA调用
    // false：通过合约调用
}
```

安全原则：

永远不要使用tx.origin进行权限验证！

3.7 其他全局变量

gasleft() - 剩余gas

```
contract GasTracking {
    function expensiveOperation() public view returns (uint gasUsed) {
        uint gasBefore = gasleft();

        // 执行操作
        uint sum = 0;
        for (uint i = 0; i < 100; i++) {
            sum += i;
        }

        gasUsed = gasBefore - gasleft();
        return gasUsed;
    }

    function checkSufficientGas() public view {
        require(gasleft() >= 10000, "Insufficient gas");
        // 继续执行
    }
}
```

keccak256() - 哈希函数

```
contract HashExample {
    // 生成唯一ID
    function generateId(address user, uint nonce)
        public pure returns (bytes32)
    {
        return keccak256(abi.encodePacked(user, nonce));
    }

    // 验证密码
    bytes32 public passwordHash;

    function setPassword(string memory password) public {
        passwordHash = keccak256(bytes(password));
    }

    function checkPassword(string memory password)
        public view returns (bool)
    {
        return keccak256(bytes(password)) == passwordHash;
    }

    // 生成随机数（不够安全，仅示例）
    function randomNumber() public view returns (uint) {
        bytes32 hash = keccak256(
            abi.encodePacked(
                block.timestamp,
                block.difficulty,
                msg.sender
            )
        );
        return uint(hash);
    }
}
```

blockhash() - 区块哈希

```
contract BlockHashExample {
    // 获取最近区块的哈希
    function getRecentBlockHash(uint blockNumber)
        public view returns (bytes32)
    {
        require(
            blockNumber < block.number,
            "Block not yet mined"
        );
        require(
            block.number - blockNumber <= 256,
            "Block too old"
```

```
        );

        return blockhash(blockNumber);
    }

    // 简单随机数（不够安全）
    function simpleRandom() public view returns (uint) {
        bytes32 hash = blockhash(block.number - 1);
        return uint(hash) % 100;  // 0-99的随机数
    }
}
```

限制：

- 只能获取最近256个块的哈希
- 更早的块返回`bytes32(0)`
- 不要用于重要的随机性（推荐Chainlink VRF）

---

4. 枚举类型

4.1 枚举基础

枚举（enum）是用户定义的类型，用于表示一组有限的选项或状态。

定义语法：

```
enum 枚举名 {
    选项1,  // 0
    选项2,  // 1
    选项3   // 2
}
```

基础示例：

```
contract EnumBasics {
    // 定义订单状态枚举
    enum OrderStatus {
        Pending,      // 0
        Paid,         // 1
        Shipped,      // 2
        Delivered,    // 3
        Cancelled     // 4
    }

    // 声明枚举变量
    OrderStatus public status;  // 默认值：Pending (0)

    // 设置枚举值
    function createOrder() public {
        status = OrderStatus.Pending;
```

```
    }

    function payOrder() public {
        require(status == OrderStatus.Pending, "Not pending");
        status = OrderStatus.Paid;
    }

    function shipOrder() public {
        require(status == OrderStatus.Paid, "Not paid");
        status = OrderStatus.Shipped;
    }

    // 检查枚举值
    function isPaid() public view returns (bool) {
        return status == OrderStatus.Paid;
    }
}
```

4.2 枚举的优势

优势1：代码可读性

```
// 不使用枚举（难理解）
uint public status = 2;

if (status == 2) {
    // 2代表什么？需要查文档
}

// 使用枚举（一目了然）
OrderStatus public status = OrderStatus.Shipped;

if (status == OrderStatus.Shipped) {
    // 清晰明了，状态是"已发货"
}
```

优势2：类型安全

```
contract TypeSafety {
    enum Status { Pending, Active, Completed }
    Status public status;

    function setStatus() public {
        status = Status.Active;      // 正确
        // status = Status.Invalid;  // 编译错误（不存在的值）
        // status = 10;              // 编译错误（类型不匹配）
    }
}
```

优势3：节省Gas

```
contract GasSaving {
    // 使用string：昂贵
    string public statusStr = "Active";  // 存储字符串消耗大量gas

    enum Status { Pending, Active, Completed }
    // 使用enum：便宜
    Status public statusEnum = Status.Active;  // 只存储uint8，非常便宜
}
```

4.3 枚举操作

类型转换：

```
contract EnumConversion {
    enum Status { Pending, Active, Completed }

    function conversions() public pure returns (uint, Status) {
        Status status = Status.Active;

        // 枚举 → 整数
        uint statusValue = uint(status);  // 1

        // 整数 → 枚举
        Status newStatus = Status(2);  // Completed

        return (statusValue, newStatus);
    }

    // 安全转换（检查范围）
    function safeConvert(uint value) public pure returns (Status) {
        require(value <= uint(type(Status).max), "Invalid status value");
        return Status(value);
    }
}
```

获取枚举范围：

```
contract EnumRange {
    enum Status { Pending, Active, Completed }

    function getRange() public pure returns (Status, Status) {
        Status minValue = type(Status).min;  // Pending (0)
        Status maxValue = type(Status).max;  // Completed (2)
        return (minValue, maxValue);
    }
}
```

在映射中使用：

```
contract EnumInMapping {
    enum Role { None, User, Admin, Owner }

    // 地址到角色的映射
    mapping(address => Role) public userRoles;

    // 角色统计
    mapping(Role => uint) public roleCount;

    function assignRole(address user, Role role) public {
        userRoles[user] = role;
        roleCount[role]++;
    }

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

4.4 状态机模式

状态机是枚举最经典的应用场景。

状态转换图：

```
[Fundraising] ──────┐
      │             │
      │ 达到目标     │ 超时未达标
      ↓             ↓
[Successful]    [Failed]
```

完整实现：

```
contract Crowdfunding {
    enum State { Fundraising, Successful, Failed }

    State public currentState = State.Fundraising;
    address public creator;
    uint public goal;
    uint public deadline;
    uint public totalFunded;
    mapping(address => uint) public contributions;

    event StateChanged(State newState);
    event Contribution(address indexed contributor, uint amount);

    modifier inState(State expectedState) {
        require(
            currentState == expectedState,
            "Invalid state for this operation"
        );
```

```
    _;
}

constructor(uint _goal, uint durationDays) {
    creator = msg.sender;
    goal = _goal;
    deadline = block.timestamp + (durationDays * 1 days);
}

// 贡献资金（仅在募资中）
function contribute()
    public
    payable
    inState(State.Fundraising)
{
    require(block.timestamp <= deadline, "Fundraising ended");
    require(msg.value > 0, "Must contribute");

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

    emit Contribution(msg.sender, msg.value);

    // 自动检查是否达到目标
    if (totalFunded >= goal) {
        currentState = State.Successful;
        emit StateChanged(State.Successful);
    }
}

// 检查并更新状态
function checkGoalReached() public inState(State.Fundraising) {
    require(block.timestamp > deadline, "Deadline not passed");

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

    emit StateChanged(currentState);
}

// 创建者提取资金（仅成功时）
function withdrawFunds() public inState(State.Successful) {
    require(msg.sender == creator, "Only creator can withdraw");

    uint amount = address(this).balance;
    (bool sent, ) = creator.call{value: amount}("");
    require(sent, "Transfer failed");
}
```

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

        contributions[msg.sender] = 0;
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Refund failed");
    }
}
```

状态机优势：

1. **状态转换逻辑清晰**：明确哪些操作在哪些状态下可执行
2. **防止无效操作**：错误状态下的操作会被拒绝
3. **代码易于维护**：添加新状态和转换很容易
4. **减少if-else嵌套**：用状态替代复杂的条件判断
5. **增强安全性**：状态检查保证逻辑正确

---

5. constant和immutable

5.1 为什么需要常量

问题场景：

```
contract NoConstant {
    uint public maxSupply = 1000000;  // 存储在storage

    function checkLimit(uint amount) public view returns (bool) {
        return amount <= maxSupply;  // 每次读取storage，消耗2100 gas
    }
}
```

每次访问storage都要消耗gas，但maxSupply永远不变，为什么不优化?

解决方案：使用constant或immutable

5.2 constant - 编译时常量

定义：

constant必须在声明时赋值，值在编译时确定，运行时不能改变。

语法：

```
类型 public constant 变量名 = 值;
```

示例：

```
contract ConstantExample {
    // 常量定义
    uint public constant MAX_SUPPLY = 1000000;
    uint public constant MIN_AMOUNT = 100;
    address public constant ZERO_ADDRESS = address(0);
    string public constant NAME = "My Token";

    // 计算型常量
    uint public constant DECIMALS = 18;
    uint public constant MAX_SUPPLY_WITH_DECIMALS = MAX_SUPPLY * 10**DECIMALS;

    // 使用常量
    function checkAmount(uint amount) public pure returns (bool) {
        return amount >= MIN_AMOUNT && amount <= MAX_SUPPLY;
        // 直接使用常量值，无需读取storage，gas = 0
    }
}
```

constant的特点：

1. **编译时确定**：值必须是常量表达式
2. **内联替换**：编译器将常量直接替换到代码中
3. **不占storage**：不消耗storage槽位
4. **访问cost = 0**：相当于直接使用数字
5. **不可修改**：运行时绝对不能改变

可以和不可以：

```
contract ConstantRules {
    // 可以：字面值
    uint public constant NUMBER = 100;

    // 可以：计算表达式
    uint public constant RESULT = 50 * 2;

    // 可以：使用其他常量
    uint public constant BASE = 10;
    uint public constant DERIVED = BASE * 10;

    // 不可以：运行时值
    // uint public constant TIME = block.timestamp;  // 编译错误
    // uint public constant SENDER = uint160(msg.sender);  // 编译错误

    // 不可以：在构造函数中赋值
    // uint public constant VALUE;
    // constructor() {
    //     VALUE = 100;  // 编译错误
    // }
}
```

5.3 immutable - 部署时常量

定义：

immutable可以在构造函数中赋值，一旦部署后就不能改变。

语法：

```
类型 public immutable 变量名;
```

示例：

```
contract ImmutableExample {
    // immutable变量
    address public immutable OWNER;
    address public immutable FACTORY;
    uint public immutable DEPLOYED_AT;
    uint public immutable INITIAL_SUPPLY;

    // 在构造函数中赋值
    constructor(address factory, uint supply) {
        OWNER = msg.sender;
        FACTORY = factory;
        DEPLOYED_AT = block.timestamp;
        INITIAL_SUPPLY = supply;
    }

    // 使用immutable
    function checkOwner() public view returns (bool) {
        return msg.sender == OWNER;
        // 访问immutable比storage便宜，约200 gas
    }
}
```

immutable的特点：

1. **部署时确定**：在构造函数中赋值
2. **运行时不可变**：部署后不能修改
3. **不占storage**：存储在合约代码中
4. **访问便宜**：约200 gas（比storage的2100便宜）
5. **可用运行时值**：可以使用msg.sender、block.timestamp等

5.4 三种变量类型对比

特性	storage变量	constant	immutable
赋值时机	任何时候	编译时	构造函数
可修改性	可修改	不可修改	不可修改
存储位置	Storage	代码（内联）	代码
访问成本	~2100 gas	0 gas	~200 gas
可用运行时值	是	否	是
典型用途	动态数据	固定常量	部署时确定的值

使用场景对比：

```
contract ComparisonExample {
    // Storage：运行时可变
    uint public totalSupply;  // 可以mint和burn
    mapping(address => uint) public balances;  // 会频繁变化
    address public owner;  // 可以转移所有权

    // Constant：编译时确定，永不改变
    uint public constant MAX_SUPPLY = 1000000;
    uint public constant DECIMALS = 18;
    string public constant NAME = "My Token";

    // Immutable：部署时确定，之后不变
    address public immutable CREATOR;
    address public immutable FACTORY;
    uint public immutable CREATED_AT;

    constructor(address factory) {
        CREATOR = msg.sender;
        FACTORY = factory;
        CREATED_AT = block.timestamp;
        owner = msg.sender;
    }
}
```

5.5 Gas优化效果

优化示例：

```
contract GasOptimization {
    // 未优化：storage
    uint public feeRate = 300;  // 每次访问：~2100 gas

    function calculateFee1(uint amount) public view returns (uint) {
        return amount * feeRate / 10000;
    }
```

```
    // Gas: ~2500

    // 优化：constant
    uint public constant FEE_RATE = 300;  // 访问：0 gas

    function calculateFee2(uint amount) public pure returns (uint) {
        return amount * FEE_RATE / 10000;
    }
    // Gas: ~400
    // 节省：~84%
}
```

实际项目中的应用：

```
contract TokenWithOptimization {
    // Constant：永远不变的值
    string public constant NAME = "My Token";
    string public constant SYMBOL = "MTK";
    uint8 public constant DECIMALS = 18;
    uint public constant MAX_SUPPLY = 1000000 * 10**DECIMALS;
    uint public constant TRANSFER_FEE = 100;  // 1%

    // Immutable：部署时确定
    address public immutable OWNER;
    address public immutable FACTORY;
    uint public immutable DEPLOYMENT_TIME;

    // Storage：会变化的值
    uint public totalSupply;
    mapping(address => uint) public balances;
    bool public paused;

    constructor(address factory) {
        OWNER = msg.sender;
        FACTORY = factory;
        DEPLOYMENT_TIME = block.timestamp;
    }

    function transfer(address to, uint amount) public returns (bool) {
        // 使用constant：0 gas
        uint fee = amount * TRANSFER_FEE / 10000;

        // 检查immutable：~200 gas
        require(msg.sender != OWNER || !paused, "Transfers paused");

        // 操作storage：正常gas
        balances[msg.sender] -= amount;
        balances[to] += amount - fee;

        return true;
    }
```

```
}
```

命名规范：

```
// 常量通常用大写 + 下划线
uint public constant MAX_SUPPLY = 1000000;
uint public constant MIN_AMOUNT = 100;
address public constant ZERO_ADDRESS = address(0);

// immutable也可以用大写
address public immutable OWNER;
uint public immutable CREATED_AT;
```

---

6. 综合实战案例

6.1 支付商店合约

综合运用msg.sender、msg.value、address payable。

```
contract SimpleShop {
    address public immutable OWNER;
    uint public constant ITEM_PRICE = 0.1 ether;

    mapping(address => uint) public purchases;

    event ItemPurchased(address indexed buyer, uint quantity, uint totalPaid);
    event Withdrawal(address indexed owner, uint amount);

    constructor() {
        OWNER = msg.sender;
    }

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

    // 购买商品
    function buyItem(uint quantity) public payable {
        require(quantity > 0, "Invalid quantity");

        uint totalCost = ITEM_PRICE * quantity;
        require(msg.value == totalCost, "Incorrect payment");

        purchases[msg.sender] += quantity;

        emit ItemPurchased(msg.sender, quantity, msg.value);
    }
```

```
    // 查询购买数量
    function getPurchases(address buyer) public view returns (uint) {
        return purchases[buyer];
    }

    // 提现（仅owner）
    function withdraw() public onlyOwner {
        uint balance = address(this).balance;
        require(balance > 0, "No balance to withdraw");

        (bool sent, ) = OWNER.call{value: balance}("");
        require(sent, "Transfer failed");

        emit Withdrawal(OWNER, balance);
    }

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

6.2 完整众筹合约

综合运用枚举、时间戳、msg.value等所有知识点。

```
contract AdvancedCrowdfunding {
    enum State { Fundraising, Successful, Failed, PaidOut }

    State public currentState = State.Fundraising;

    address public immutable CREATOR;
    uint public immutable GOAL;
    uint public immutable DEADLINE;
    uint public immutable MINIMUM_CONTRIBUTION = 0.01 ether;

    uint public totalFunded;
    uint public contributorCount;

    mapping(address => uint) public contributions;
    address[] public contributors;

    event StateChanged(State oldState, State newState, uint timestamp);
    event Contribution(address indexed contributor, uint amount, uint totalFunded);
    event FundsWithdrawn(address indexed creator, uint amount);
    event Refunded(address indexed contributor, uint amount);

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

```
    modifier onlyCreator() {
        require(msg.sender == CREATOR, "Only creator");
        _;
    }

    constructor(uint goalAmount, uint durationDays) {
        require(goalAmount > 0, "Goal must be positive");
        require(durationDays >= 1 && durationDays <= 90, "Duration: 1-90 days");

        CREATOR = msg.sender;
        GOAL = goalAmount;
        DEADLINE = block.timestamp + (durationDays * 1 days);
    }

    // 贡献资金
    function contribute() public payable inState(State.Fundraising) {
        require(block.timestamp <= DEADLINE, "Fundraising ended");
        require(msg.value >= MINIMUM_CONTRIBUTION, "Below minimum");

        // 新贡献者
        if (contributions[msg.sender] == 0) {
            contributors.push(msg.sender);
            contributorCount++;
        }

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

        emit Contribution(msg.sender, msg.value, totalFunded);

        // 达到目标自动成功
        if (totalFunded >= GOAL) {
            State oldState = currentState;
            currentState = State.Successful;
            emit StateChanged(oldState, State.Successful, block.timestamp);
        }
    }

    // 检查并更新状态
    function checkGoalReached() public inState(State.Fundraising) {
        require(block.timestamp > DEADLINE, "Still active");

        State oldState = currentState;
        State newState;

        if (totalFunded >= GOAL) {
            newState = State.Successful;
        } else {
            newState = State.Failed;
        }
```

```
        currentState = newState;
        emit StateChanged(oldState, newState, block.timestamp);
    }

    // 创建者提取资金
    function withdrawFunds() public onlyCreator inState(State.Successful) {
        currentState = State.PaidOut;

        uint amount = address(this).balance;
        (bool sent, ) = CREATOR.call{value: amount}("");
        require(sent, "Transfer failed");

        emit FundsWithdrawn(CREATOR, amount);
    }

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

        contributions[msg.sender] = 0;
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Refund failed");

        emit Refunded(msg.sender, amount);
    }

    // 查询函数
    function getInfo() public view returns (
        State state,
        uint goal,
        uint funded,
        uint deadline,
        uint timeRemaining,
        uint contributors
    ) {
        uint remaining = 0;
        if (block.timestamp < DEADLINE) {
            remaining = DEADLINE - block.timestamp;
        }

        return (
            currentState,
            GOAL,
            totalFunded,
            DEADLINE,
            remaining,
            contributorCount
        );
    }

    function getProgress() public view returns (uint percentage) {
```

```
        return (totalFunded * 100) / GOAL;
    }

    function isActive() public view returns (bool) {
        return currentState == State.Fundraising &&
               block.timestamp <= DEADLINE;
    }
}
```

---

7. 常见问题解答

Q1：为什么要区分address和address payable?

答：为了类型安全。

编译器可以检查：

- 防止向普通address转账（会失败）
- 明确哪些地址可以接收ETH
- 减少运行时错误

Q2：transfer、send、call应该用哪个?

答：2024年推荐使用call。

推荐：

```
(bool sent, ) = recipient.call{value: amount}("");
require(sent, "Failed");
```

理由：

- transfer和send有2300 gas限制，可能不够用
- call更灵活，可以转发所有gas
- 配合ReentrancyGuard使用很安全

避免：

- send：容易忘记检查返回值

Q3：tx.origin为什么危险?

答：容易受到钓鱼攻击。

危险场景：

```
用户 → 恶意合约 → 你的合约

在你的合约中：
tx.origin = 用户（通过检查！）
msg.sender = 恶意合约

如果用tx.origin检查权限，攻击者可以代表用户操作
```

安全原则：永远使用msg.sender做权限检查！

Q4：block.timestamp可以用于随机数吗?

答：不能用于重要的随机性。

问题：

- 矿工可以操纵时间戳约15秒
- 可预测性
- 不够随机

适合用途：

- 时间锁（小时、天级别）
- 截止日期
- 时间间隔检查

真正的随机数方案：

- Chainlink VRF
- 提交-揭示方案
- 预言机

Q5：constant和immutable的区别?

答：赋值时机不同。

constant：

- 编译时赋值
- 必须是常量表达式
- 访问cost = 0

immutable：

- 构造函数中赋值
- 可以用运行时值
- 访问cost ≈ 200 gas

选择建议：

- 纯常量用constant
- 部署时确定的值用immutable
- 运行时可变的用storage

Q6：枚举的底层类型是什么?

答：uint8

```
enum Status { A, B, C }  // 实际上是uint8

Status s = Status.B;
uint value = uint(s);  // 1
```

特点：

- 从0开始编号
- 最多256个值（uint8的范围）
- 存储消耗很小

Q7：如何选择使用枚举还是uint?

答：根据语义和安全性需求。

使用枚举：

- 有限的状态集合
- 需要类型安全
- 提高可读性
- 防止无效值

使用uint：

- 数值有意义
- 需要数学运算
- 范围不固定

---

8. 知识点总结

address类型

两种类型：

- address：基础类型，不能接收ETH
- address payable：可以接收ETH，有transfer/send方法

属性和方法：

- `.balance`：查询余额（wei）
- `.code`：获取字节码
- `.codehash`：代码哈希
- `.transfer()`：转账（payable专用）
- `.send()`：转账+返回值（payable专用）
- `.call()`：底层调用

类型转换：

- `payable(addr)`：转为address payable
- 自动转换：address payable → address

转账方式

方法	特点	推荐度
transfer	自动回滚，2300 gas	中
send	需检查返回值，2300 gas	低
call	无限制，灵活	高

安全原则：遵循CEI模式！

全局变量

msg对象：

- `msg.sender`：调用者（最常用）
- `msg.value`：发送的ETH（payable函数）
- `msg.data`：调用数据
- `msg.sig`：函数选择器

block对象：

- `block.timestamp`：时间戳（常用）
- `block.number`：区块号（常用）
- `blockhash()`：区块哈希

危险变量：

- `tx.origin`：永远不要用于权限检查！

枚举类型

定义和使用：

```
enum State { A, B, C }
State public state = State.A;
```

优势：

- 代码可读性高
- 类型安全
- 节省gas

应用：

- 状态机模式
- 有限选项集合
- 角色管理

constant和immutable

constant：

- 编译时常量
- 访问cost = 0 gas
- 用于固定值

immutable：

- 部署时常量
- 访问cost ≈ 200 gas
- 用于部署时确定的值

优化效果：节省约84-95%的gas

---

9. 学习检查清单

完成本课后，你应该能够：

address类型：

- [ ] 理解address和address payable的区别
- [ ] 会使用address的属性和方法
- [ ] 会进行地址类型转换
- [ ] 会检查地址是否是合约

转账方式：

- [ ] 理解三种转账方式的区别
- [ ] 知道何时使用哪种方式
- [ ] 理解CEI模式的重要性
- [ ] 会防范重入攻击

全局变量：

- [ ] 掌握msg.sender的用法
- [ ] 掌握msg.value的用法
- [ ] 理解block.timestamp的限制
- [ ] 知道tx.origin的危险性

枚举类型：

- [ ] 会定义和使用枚举
- [ ] 理解枚举的优势
- [ ] 会实现状态机模式
- [ ] 会进行枚举类型转换

constant和immutable：

- [ ] 理解两者的区别
- [ ] 知道何时使用constant
- [ ] 知道何时使用immutable
- [ ] 理解gas优化效果

---

10. 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. **完成综合项目**：实现完整的众筹合约
3. **研究真实项目**：分析OpenZeppelin、Uniswap的代码
4. **准备学习第6课**：合约继承

下节课预告：第6.1课 - 合约继承

我们将学习：

- 单继承和多重继承
- super关键字的使用
- 构造函数继承
- 函数重写（override）
- 抽象合约和接口
- 继承中的最佳实践

---

11. 扩展资源

官方文档：

- Solidity地址类型：https://docs.soliditylang.org/en/latest/types.html#address
- 全局变量：https://docs.soliditylang.org/en/latest/units-and-global-variables.html
- 枚举：https://docs.soliditylang.org/en/latest/types.html#enums

学习资源：

- Solidity by Example - Payable：https://solidity-by-example.org/payable/
- Solidity by Example - Enum：https://solidity-by-example.org/enum/

安全资源：

- Smart Contract Best Practices
- Reentrancy Attack Prevention
- OpenZeppelin Security Guidelines

工具推荐：

- Remix IDE：在线开发环境
- Etherscan：区块链浏览器

- OpenZeppelin Contracts：安全合约库