Solidity智能合约开发知识

第3.2课：映射和结构体

学习目标：掌握Mapping的使用和限制、理解Struct的定义和操作、学会Mapping+Struct组合模式、能够设计复杂的数据结构

预计学习时间：2小时

难度等级：入门进阶

目录

---

1. Mapping基础概念

1.1 什么是Mapping

Mapping（映射）是Solidity中最常用的数据结构之一，类似于其他编程语言中的HashMap、Dictionary或关联数组。它提供了一种通过键（key）快速查找值（value）的方式。

基本概念：

- **键值对存储**：每个键对应一个值
- **快速查找**：O(1)时间复杂度
- **哈希存储**：底层使用哈希表实现
- **永久存储**：只能作为storage变量

1.2 Mapping的语法

声明语法：

```
mapping(keyType => valueType) 变量名;
```

支持的键类型：

- 值类型：`uint`, `int`, `address`, `bool`, `bytes1`到`bytes32`, `enum`
- 不支持：引用类型（数组、struct、mapping）

支持的值类型：

- 任何类型都可以作为值，包括：
  - 值类型：`uint`，`bool`，`address`等
  - 引用类型：`string`，`bytes`，`array`，`struct`，`mapping`

1.3 Mapping基本示例

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

contract MappingBasics {
    // 地址到余额的映射
    mapping(address => uint256) public balances;

    // ID到名字的映射
    mapping(uint256 => string) public names;

    // 地址到白名单状态的映射
    mapping(address => bool) public whitelist;

    // 设置余额
    function setBalance(address user, uint256 amount) public {
        balances[user] = amount;
    }

    // 获取余额
    function getBalance(address user) public view returns (uint256) {
        return balances[user];
    }

    // 设置名字（推荐使用 calldata 以节省 Gas）
    function setName(uint256 id, string calldata name) public {
        names[id] = name;
    }

    // 添加到白名单
    function addToWhitelist(address user) public {
        whitelist[user] = true;
    }

    // 检查白名单
    function isWhitelisted(address user) public view returns (bool) {
        return whitelist[user];
    }
}
```

1.4 Mapping的工作原理

底层存储机制：

Mapping并不像数组那样真正存储所有的键值对，而是通过哈希函数计算存储位置。

存储位置计算：

```
存储位置 = keccak256(abi.encode(key, mappingSlot))
```

其中：

- `key`：要查询的键
- `mappingSlot`：mapping在合约storage中的槽位

查找过程：

```
用户查询 balances[0x123...]
    ↓
计算：keccak256(0x123..., mappingSlot)
    ↓
得到storage位置：0xabcd...
    ↓
读取该位置的值
    ↓
返回结果
```

时间复杂度：

- 查找：O(1) 常数时间
- 插入：O(1) 常数时间
- 更新：O(1) 常数时间
- 删除：O(1) 常数时间

这就是为什么mapping如此高效的原因！

---

2. Mapping的特性和限制

2.1 Mapping的五大核心特性

特性1：所有键都"存在"

这是Mapping最重要也最容易混淆的特性。

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

    function testDefault() public view returns (uint256) {
        // 即使从未设置过这个地址的余额
        // 也会返回默认值0，而不是报错
        return balances[address(0x123)];  // 返回：0
    }
}
```

重要理解：

- Mapping中所有可能的键都被认为"存在"
- 未设置的键返回该类型的默认值
- 不会抛出"键不存在"的错误

不同类型的默认值：

```
contract DefaultValues {
    mapping(address => uint256) uintMap;       // 默认: 0
    mapping(address => bool) boolMap;          // 默认: false
    mapping(address => address) addressMap;    // 默认: 0x0000...
    mapping(address => string) stringMap;      // 默认: ""
    mapping(address => int256) intMap;         // 默认: 0

    function showDefaults(address key) public view returns (
        uint256,
        bool,
        address,
        string memory,
        int256
    ) {
        return (
            uintMap[key],      // 0
            boolMap[key],      // false
            addressMap[key],   // 0x0000000000000000000000000000000000000000
            stringMap[key],    // ""
            intMap[key]        // 0
        );
    }
}
```

问题：如何区分"值是0"和"从未设置"?

稍后我们会在Mapping+Struct组合中解决这个问题。

特性2：不存储键列表

Mapping只存储值，不存储键的列表。

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

    // 错误：无法获取所有键
    // function getAllKeys() public view returns (address[] memory) {
    //     // 不可能实现！mapping不存储键列表
    // }

    // 错误：无法获取mapping的大小
    // function getSize() public view returns (uint256) {
    //     // mapping没有.length属性
    // }
}
```

原因：

为了节省存储空间和Gas成本，Solidity的mapping不维护键列表。

特性3：不能遍历

由于不存储键列表，mapping无法被遍历。

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

    // 错误：无法遍历
    // function sumAllBalances() public view returns (uint256) {
    //     uint256 total = 0;
    //     for(address user in balances) {  // 编译错误!
    //         total += balances[user];
    //     }
    //     return total;
    // }
}
```

解决方案：使用Mapping+Array组合（稍后讲解）

特性4：只能用于Storage

Mapping只能作为状态变量，不能在函数内创建。

```
contract StorageOnly {
    // 正确：作为状态变量
    mapping(address => uint256) public balances;

    function test() public {
        // 错误：不能在memory中创建mapping
        // mapping(address => uint256) memory localMap;  // 编译错误!

        // 错误：不能在calldata中使用mapping
        // mapping(address => uint256) calldata dataMap;  // 编译错误!
    }
}
```

为什么?

- Mapping的存储结构依赖于哈希计算
- Memory和calldata不支持这种存储方式
- Mapping必须在区块链上永久存储

特性5：不能作为参数或返回值

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

    // 错误：不能作为函数参数
    // function processMapping(
    //     mapping(address => uint256) storage map  // 编译错误!
    // ) public {
    //     // ...
    // }

    // 错误：不能作为返回值
    // function getMapping() public view
    //     returns (mapping(address => uint256) storage)  // 编译错误!
    // {
    //     return balances;
    // }
}
```

原因：

- Mapping的大小不确定
- 无法复制整个mapping
- 传递mapping的成本无法预估

2.2 Mapping的限制总结

操作	是否支持	说明
赋值	支持	`map[key] = value`
查询	支持	`value = map[key]`
删除单个值	支持	`delete map[key]`
删除整个mapping	不支持	无法清空整个mapping
遍历	不支持	没有键列表
获取长度	不支持	没有.length属性
作为参数	不支持	不能传递给函数
作为返回值	不支持	不能返回
Memory中使用	不支持	只能storage

2.3 delete操作

虽然不能删除整个mapping，但可以删除单个键的值。

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

    function setBalance(address user, uint256 amount) public {
        balances[user] = amount;
    }

    // 删除单个键的值
    function deleteBalance(address user) public {
        delete balances[user];
        // 将balances[user]重置为默认值0
    }

    function demonstrateDelete() public {
        address user = address(0x123);

        // 设置值
        balances[user] = 1000;
        // balances[user] = 1000

        // 删除
        delete balances[user];
        // balances[user] = 0 (回到默认值)
    }
}
```

delete的效果:

- 将指定键的值重置为默认值
- 不是真正"删除"，而是"重置"
- 可以获得部分Gas退款

---

3. 嵌套Mapping

3.1 嵌套Mapping的概念

嵌套Mapping是指mapping的值本身也是一个mapping。

语法：

```
mapping(keyType1 => mapping(keyType2 => valueType)) 变量名;
```

理解方式：

- 外层mapping：第一级键值对
- 内层mapping：第二级键值对
- 类似于二维表或矩阵

3.2 嵌套Mapping示例

```
contract NestedMapping {
    // 用户地址 → 代币地址 → 余额数量
    mapping(address => mapping(address => uint256)) public tokenBalances;

    // 设置代币余额
    function setTokenBalance(
        address user,
        address token,
        uint256 amount
    ) public {
        tokenBalances[user][token] = amount;
    }

    // 获取代币余额
    function getTokenBalance(
        address user,
        address token
    ) public view returns (uint256) {
        return tokenBalances[user][token];
    }

    // 转账代币
    function transferToken(
        address token,
        address to,
        uint256 amount
    ) public {
```

```
        require(tokenBalances[msg.sender][token] >= amount, "Insufficient balance");

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

3.3 ERC20授权机制

嵌套Mapping最经典的应用是ERC20代币的授权机制。

```
contract ERC20Authorization {
    // 余额：地址 → 余额
    mapping(address => uint256) public balances;

    // 授权：所有者 → 被授权者 → 授权数量
    mapping(address => mapping(address => uint256)) public allowance;

    // 授权
    function approve(address spender, uint256 amount) public {
        allowance[msg.sender][spender] = amount;
        // msg.sender授权spender可以花费amount数量的代币
    }

    // 查询授权额度
    function getAllowance(
        address owner,
        address spender
    ) public view returns (uint256) {
        return allowance[owner][spender];
    }

    // 代他人转账
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public {
        // 检查授权额度
        require(allowance[from][msg.sender] >= amount, "Allowance exceeded");
        require(balances[from] >= amount, "Insufficient balance");

        // 扣除授权额度
        allowance[from][msg.sender] -= amount;

        // 转账
        balances[from] -= amount;
        balances[to] += amount;
    }
}
```

ERC20授权流程：

```
步骤1：Alice授权Bob可以花费100个代币
alice.approve(bob, 100)
    ↓
allowance[alice][bob] = 100

步骤2：Bob代Alice转账50个代币给Carol
bob.transferFrom(alice, carol, 50)
    ↓
检查：allowance[alice][bob] >= 50 ✓
    ↓
扣除授权：allowance[alice][bob] = 50
    ↓
转账：balances[alice] -= 50, balances[carol] += 50

结果：
- Alice余额减少50
- Carol余额增加50
- Bob剩余授权额度50
```

应用场景：

- DeFi协议（如Uniswap）代用户交易代币
- 交易所代用户充值
- 支付网关代用户支付

3.4 多层嵌套Mapping

理论上可以无限嵌套，但实际很少使用三层以上。

```
contract MultiLevelMapping {
    // 三层嵌套：用户 → 游戏 → 关卡 → 分数
    mapping(address => mapping(uint256 => mapping(uint256 => uint256))) public gameScores;

    function setScore(
        uint256 gameId,
        uint256 level,
        uint256 score
    ) public {
        gameScores[msg.sender][gameId][level] = score;
    }

    function getScore(
        address player,
        uint256 gameId,
        uint256 level
    ) public view returns (uint256) {
        return gameScores[player][gameId][level];
    }
}
```

注意：

- 嵌套层数越多，代码可读性越差
- 通常不超过2-3层
- 考虑使用struct替代深层嵌套

---

4. Mapping与Array组合

4.1 为什么需要组合使用

Mapping的问题：不能遍历

Array的问题：查找效率低（O(n)）

解决方案：组合使用，发挥各自优势

数据结构	优势	劣势
Mapping	O(1)查找	不能遍历
Array	可以遍历	O(n)查找
Mapping+Array	O(1)查找 + 可遍历	需要维护一致性

4.2 基本组合模式

```
contract MappingArrayCombo {
    // 主数据存储
    mapping(address => uint256) public balances;

    // 键列表（用于遍历）
    address[] public userList;

    // 存在性检查（用于去重）
    mapping(address => bool) public isUser;

    // 添加用户
    function addUser(address user, uint256 balance) public {
        require(!isUser[user], "User already exists");

        // 设置余额
        balances[user] = balance;

        // 添加到列表
        userList.push(user);

        // 标记为已存在
        isUser[user] = true;
    }
```

```
    // 快速查找（O(1)）
    function getBalance(address user) public view returns (uint256) {
        return balances[user];
    }

    // 检查用户是否存在（O(1)）
    function checkUser(address user) public view returns (bool) {
        return isUser[user];
    }

    // 遍历所有用户
    function getAllUsers() public view returns (address[] memory) {
        return userList;
    }

    // 获取用户数量
    function getUserCount() public view returns (uint256) {
        return userList.length;
    }

    // 批量查询余额
    function getAllBalances() public view returns (uint256[] memory) {
        uint256[] memory allBalances = new uint256[](userList.length);

        for(uint256 i = 0; i < userList.length; i++) {
            allBalances[i] = balances[userList[i]];
        }

        return allBalances;
    }
}
```

4.3 组合模式架构图

添加用户流程：

```
addUser(0x123, 1000)
        |
        +------------------+------------------+
        |                  |                  |
        v                  v                  v
balances[0x123]     userList.push()      isUser[0x123]
    = 1000             (0x123)              = true
        |                  |                  |
        +------------------+------------------+
                           |
                      添加完成
```

查询流程：

快速查找余额：`balances[0x123]` → O(1)

```
检查用户存在：isUser[0x123] → O(1)
遍历所有用户：for(userList) → O(n)
```

4.4 删除操作的优化

删除元素时需要同时维护mapping和array的一致性。

```
contract DeleteOptimization {
    mapping(address => uint256) public balances;
    address[] public userList;
    mapping(address => bool) public isUser;
    mapping(address => uint256) public userIndex;  // 记录索引

    // 添加用户
    function addUser(address user, uint256 balance) public {
        require(!isUser[user], "User already exists");

        balances[user] = balance;
        userIndex[user] = userList.length;  // 记录索引
        userList.push(user);
        isUser[user] = true;
    }

    // 删除用户（快速删除，不保序）
    function removeUser(address user) public {
        require(isUser[user], "User does not exist");

        uint256 index = userIndex[user];
        uint256 lastIndex = userList.length - 1;

        // 如果不是最后一个元素，用最后一个元素替换
        if(index != lastIndex) {
            address lastUser = userList[lastIndex];
            userList[index] = lastUser;
            userIndex[lastUser] = index;
        }

        // 删除最后一个元素
        userList.pop();

        // 清理mapping
        delete balances[user];
        delete isUser[user];
        delete userIndex[user];
    }
}
```

删除过程示例：

```
初始状态：
userList = [0xA, 0xB, 0xC, 0xD]
```

```
userIndex[0xA] = 0
userIndex[0xB] = 1
userIndex[0xC] = 2
userIndex[0xD] = 3

删除 0xB:

步骤1: 找到索引
index = userIndex[0xB] = 1

步骤2: 用最后元素替换
userList[1] = userList[3] = 0xD
userIndex[0xD] = 1

步骤3: 删除最后元素
userList.pop()

步骤4: 清理mapping
delete balances[0xB]
delete isUser[0xB]
delete userIndex[0xB]

结果:
userList = [0xA, 0xD, 0xC]
```

---

5. Struct结构体

5.1 什么是Struct

Struct（结构体）是一种自定义的复合数据类型，允许将多个相关的变量组织在一起。

作用:

- 组织相关数据
- 提高代码可读性
- 创建复杂的数据模型
- 实现面向对象的数据封装

5.2 Struct定义

基本语法:

```
struct 结构体名 {
    类型1 字段1;
    类型2 字段2;
    ...
}
```

示例:

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

contract StructBasics {
    // 定义用户结构体
    struct User {
        string name;
        uint256 age;
        address wallet;
        bool isActive;
    }

    // 定义书籍结构体
    struct Book {
        string title;
        string author;
        uint256 price;
        bool available;
    }

    // 定义提案结构体
    struct Proposal {
        string description;
        uint256 voteCount;
        uint256 deadline;
        bool executed;
    }
}
```

5.3 Struct定义位置

位置1：合约内部

```
contract MyContract {
    struct User {
        string name;
        uint256 age;
    }

    User public admin;
}
```

位置2：合约外部（全局）

```
// 全局定义，多个合约可以使用
struct User {
    string name;
    uint256 age;
}

contract ContractA {
    User public userA;
}

contract ContractB {
    User public userB;
}
```

位置3：库文件中

```
library Types {
    struct User {
        string name;
        uint256 age;
        address wallet;
    }
}

contract MyContract {
    Types.User public admin;
}
```

推荐做法：

- 简单项目：合约内部定义
- 复杂项目：库文件或接口中定义
- 多合约共享：全局定义或库文件

5.4 创建Struct实例

有三种方式创建struct实例：

方式1：逐个赋值

```
contract CreateStruct1 {
    struct User {
        string name;
        uint256 age;
        address wallet;
        bool isActive;
    }

    User public admin;
```

```
    function createUser1() public {
        admin.name = "Alice";
        admin.age = 25;
        admin.wallet = msg.sender;
        admin.isActive = true;
    }
}
```

特点：

- 灵活，可以只设置部分字段
- 代码较长
- 适合部分更新

方式2：构造器语法

```
contract CreateStruct2 {
    struct User {
        string name;
        uint256 age;
        address wallet;
        bool isActive;
    }

    User public admin;

    function createUser2() public {
        admin = User("Bob", 30, msg.sender, true);
    }
}
```

特点：

- 简洁
- 必须按照定义顺序传参
- 容易出错（顺序错误）

方式3：键值对（推荐）

```
contract CreateStruct3 {
    struct User {
        string name;
        uint256 age;
        address wallet;
        bool isActive;
    }

    User public admin;

    function createUser3() public {
        admin = User({
```

```
            name: "Charlie",
            age: 35,
            wallet: msg.sender,
            isActive: true
        });
    }
}
```

特点：

- 最清晰
- 不需要记住字段顺序
- 可读性最好
- 推荐使用

5.5 Struct的存储位置

Storage中的Struct

```
contract StorageStruct {
    struct User {
        string name;
        uint256 age;
    }

    // 状态变量
    User public admin;

    // 数组
    User[] public users;

    // Mapping值
    mapping(address => User) public userMap;

    function updateAdmin() public {
        // Storage引用
        User storage user = admin;
        user.name = "New Admin";  // 直接修改storage
    }
}
```

Memory中的Struct

```
contract MemoryStruct {
    struct User {
        string name;
        uint256 age;
    }

    User[] public users;
```

```
    function createMemoryUser() public pure returns (User memory) {
        // 在memory中创建
        User memory user = User({
            name: "Temp User",
            age: 20
        });

        return user;
    }

    function processUser() public view {
        // 从storage复制到memory
        User memory user = users[0];
        user.age = 30;  // 修改memory副本，不影响storage
    }
}
```

Calldata中的Struct

```
contract CalldataStruct {
    struct User {
        string name;
        uint256 age;
    }

    // 外部函数参数用calldata（只读）
    function processUser(User calldata user) external pure returns (string memory) {
        // user.age = 30;  // 编译错误：calldata是只读的
        return user.name;
    }
}
```

存储位置对比：

位置	可修改	Gas成本	使用场景
Storage	是	高	永久保存
Memory	是	中	临时处理
Calldata	否	低	外部参数

5.6 访问和修改Struct

```
contract StructOperations {
    struct User {
        string name;
        uint256 age;
        address wallet;
```

```
        bool isActive;
    }

    User[] public users;

    // 添加用户（引用类型参数如果不修改，推荐使用 calldata）
    function addUser(string calldata name, uint256 age) public {
        users.push(User({
            name: name,
            age: age,
            wallet: msg.sender,
            isActive: true
        }));
    }

    // 访问字段
    function getUserAge(uint256 index) public view returns (uint256) {
        require(index < users.length, "Index out of bounds");
        return users[index].age;
    }

    // 修改字段
    function updateUserAge(uint256 index, uint256 newAge) public {
        require(index < users.length, "Index out of bounds");
        users[index].age = newAge;
    }

    // Storage引用修改
    function deactivateUser(uint256 index) public {
        require(index < users.length, "Index out of bounds");

        User storage user = users[index];
        user.isActive = false;
    }

    // 整体替换
    function replaceUser(uint256 index, User memory newUser) public {
        require(index < users.length, "Index out of bounds");
        users[index] = newUser;
    }

    // 获取完整用户信息
    function getUser(uint256 index) public view returns (User memory) {
        require(index < users.length, "Index out of bounds");
        return users[index];
    }
}
```

6. Mapping与Struct组合

6.1 为什么组合使用

Mapping的优势：

- O(1)快速查找
- 按键索引

Struct的优势：

- 组织复杂数据
- 提高可读性

组合优势：

- 快速查找复杂数据结构
- 代码更清晰
- 功能更强大

这是Solidity开发中最常用的模式！

6.2 基本组合模式

```
contract MappingStructBasic {
    // 定义用户信息结构体
    struct UserInfo {
        string name;
        uint256 balance;
        uint256 registeredAt;
        bool exists;  // 重要：标记用户是否真实存在
    }

    // Mapping存储用户信息
    mapping(address => UserInfo) public users;

    // 注册用户（使用 calldata 优化 Gas）
    function register(string calldata name) public {
        require(!users[msg.sender].exists, "Already registered");

        users[msg.sender] = UserInfo({
            name: name,
            balance: 0,
            registeredAt: block.timestamp,
            exists: true
        });
    }

    // 查询用户信息
    function getUserInfo(address user) public view returns (UserInfo memory) {
        require(users[user].exists, "User not found");
        return users[user];
    }

    // 检查用户是否存在
```

```
    function isRegistered(address user) public view returns (bool) {
        return users[user].exists;
    }

    // 存款
    function deposit() public payable {
        require(users[msg.sender].exists, "Not registered");
        users[msg.sender].balance += msg.value;
    }
}
```

6.3 exists字段的重要性

问题场景：

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

// 查询某个地址
balances[0x123...]  // 返回: 0

// 问题：这个0代表什么?
// 1. 用户余额确实是0?
// 2. 用户从未注册?
// 无法区分!
```

解决方案：添加exists字段

```
struct UserInfo {
    uint256 balance;
    bool exists;  // 明确标记是否存在
}

mapping(address => UserInfo) public users;

// 现在可以明确区分
function checkUser(address user) public view returns (string memory) {
    if(!users[user].exists) {
        return "User not registered";
    } else if(users[user].balance == 0) {
        return "User registered, balance is 0";
    } else {
        return "User registered with balance";
    }
}
```

exists字段的作用：

1. **区分默认值和设置值**：明确用户是否真实存在
2. **逻辑清晰**：避免误判
3. **安全性**：防止对不存在用户的操作

4. **最佳实践**：几乎所有项目都使用这个模式

6.4 完整组合模式

结合Mapping、Struct和Array，实现功能完整的数据管理。

```
contract CompletePattern {
    // 用户信息结构体
    struct UserInfo {
        string name;
        string email;
        uint256 balance;
        uint256 registeredAt;
        bool exists;
    }

    // 主数据存储
    mapping(address => UserInfo) public users;

    // 地址列表（用于遍历）
    address[] public userAddresses;

    // 用户计数器
    uint256 public userCount;

    // 最大用户限制
    uint256 public constant MAX_USERS = 1000;

    // 事件
    event UserRegistered(address indexed user, string name);
    event UserUpdated(address indexed user, string name);
    event Deposit(address indexed user, uint256 amount);

    // 注册用户
    function register(string memory name, string memory email) public {
        require(!users[msg.sender].exists, "Already registered");
        require(userCount < MAX_USERS, "Max users reached");
        require(bytes(name).length > 0, "Name required");

        users[msg.sender] = UserInfo({
            name: name,
            email: email,
            balance: 0,
            registeredAt: block.timestamp,
            exists: true
        });

        userAddresses.push(msg.sender);
        userCount++;

        emit UserRegistered(msg.sender, name);
    }
```

```
// 更新个人资料
function updateProfile(string memory name, string memory email) public {
    require(users[msg.sender].exists, "Not registered");

    users[msg.sender].name = name;
    users[msg.sender].email = email;

    emit UserUpdated(msg.sender, name);
}

// 存款
function deposit() public payable {
    require(users[msg.sender].exists, "Not registered");
    require(msg.value > 0, "Must send ETH");

    users[msg.sender].balance += msg.value;

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

// 查询用户信息
function getUserInfo(address user) public view returns (UserInfo memory) {
    require(users[user].exists, "User not found");
    return users[user];
}

// 检查用户是否注册
function isRegistered(address user) public view returns (bool) {
    return users[user].exists;
}

// 获取所有用户地址
function getAllUsers() public view returns (address[] memory) {
    return userAddresses;
}

// 分批查询用户
function getUsersByRange(
    uint256 start,
    uint256 end
) public view returns (address[] memory) {
    require(start < end, "Invalid range");
    require(end <= userAddresses.length, "End out of bounds");

    uint256 length = end - start;
    address[] memory result = new address[](length);

    for(uint256 i = 0; i < length; i++) {
        result[i] = userAddresses[start + i];
    }
```

```
        return result;
    }

    // 批量查询用户信息
    function getUserInfoBatch(
        address[] memory addresses
    ) public view returns (UserInfo[] memory) {
        UserInfo[] memory result = new UserInfo[](addresses.length);

        for(uint256 i = 0; i < addresses.length; i++) {
            result[i] = users[addresses[i]];
        }

        return result;
    }
}
```

6.5 Struct中包含Mapping

Struct中可以包含mapping，但有严格限制。

```
contract StructWithMapping {
    // 提案结构体
    struct Proposal {
        string description;
        uint256 voteCount;
        uint256 deadline;
        bool executed;
        mapping(address => bool) voters;  // struct内的mapping
    }

    // 存储提案
    mapping(uint256 => Proposal) public proposals;
    uint256 public proposalCount;

    // 创建提案
    function createProposal(
        string memory description,
        uint256 duration
    ) public returns (uint256) {
        uint256 proposalId = proposalCount++;

        Proposal storage p = proposals[proposalId];
        p.description = description;
        p.voteCount = 0;
        p.deadline = block.timestamp + duration;
        p.executed = false;

        return proposalId;
    }
```

```
    // 投票
    function vote(uint256 proposalId) public {
        Proposal storage p = proposals[proposalId];

        require(block.timestamp < p.deadline, "Voting ended");
        require(!p.voters[msg.sender], "Already voted");

        p.voters[msg.sender] = true;
        p.voteCount++;
    }

    // 检查是否已投票
    function hasVoted(
        uint256 proposalId,
        address voter
    ) public view returns (bool) {
        return proposals[proposalId].voters[voter];
    }
}
```

包含mapping的struct的限制：

1. **只能在storage中**：不能在memory或calldata
2. **不能作为参数**：不能传递给函数
3. **不能作为返回值**：不能返回
4. **不能在数组中**：不能创建包含mapping的struct数组

```
contract MappingStructLimitations {
    struct ProposalWithMapping {
        string description;
        mapping(address => bool) voters;
    }

    // 正确：storage中使用
    mapping(uint256 => ProposalWithMapping) public proposals;

    // 错误：不能作为参数
    // function process(ProposalWithMapping memory p) public {
    //     // 编译错误!
    // }

    // 错误：不能作为返回值
    // function getProposal(uint256 id)
    //     public view returns (ProposalWithMapping memory) {
    //     // 编译错误!
    // }

    // 错误：不能在数组中
    // ProposalWithMapping[] public proposalArray;  // 编译错误!
}
```

7. 常见设计模式

7.1 模式1：用户管理系统

```
contract UserManagement {
    struct User {
        string name;
        uint256 balance;
        bool exists;
    }

    mapping(address => User) public users;
    address[] public userList;

    function register(string memory name) public {
        require(!users[msg.sender].exists, "Already registered");

        users[msg.sender] = User(name, 0, true);
        userList.push(msg.sender);
    }
}
```

使用场景：任何需要管理用户的应用

7.2 模式2：ID自增系统

```
contract IDSystem {
    struct Item {
        string name;
        address owner;
        uint256 createdAt;
    }

    mapping(uint256 => Item) public items;
    uint256 public itemCount;

    function createItem(string memory name) public returns (uint256) {
        uint256 itemId = itemCount++;

        items[itemId] = Item({
            name: name,
            owner: msg.sender,
            createdAt: block.timestamp
        });

        return itemId;
    }
}
```

使用场景：NFT、订单系统、票务系统

7.3 模式3：双向映射

```
contract BidirectionalMapping {
    mapping(address => uint256) public addressToId;
    mapping(uint256 => address) public idToAddress;
    uint256 public nextId;

    function register() public returns (uint256) {
        require(addressToId[msg.sender] == 0, "Already registered");

        uint256 id = ++nextId;
        addressToId[msg.sender] = id;
        idToAddress[id] = msg.sender;

        return id;
    }

    function getUserByAddress(address user) public view returns (uint256) {
        return addressToId[user];
    }

    function getAddressById(uint256 id) public view returns (address) {
        return idToAddress[id];
    }
}
```

使用场景：需要双向查询的系统

7.4 模式4：一对多关系

```
contract OneToMany {
    // 用户 → 关注列表
    mapping(address => address[]) public following;

    // 用户 → 粉丝列表
    mapping(address => address[]) public followers;

    // 快速检查是否关注
    mapping(address => mapping(address => bool)) public isFollowing;

    function follow(address user) public {
        require(!isFollowing[msg.sender][user], "Already following");
        require(msg.sender != user, "Cannot follow yourself");

        following[msg.sender].push(user);
        followers[user].push(msg.sender);
        isFollowing[msg.sender][user] = true;
    }

    function getFollowing(address user) public view returns (address[] memory) {
```

```
        return following[user];
    }

    function getFollowers(address user) public view returns (address[] memory) {
        return followers[user];
    }
}
```

使用场景：社交网络、关注系统

7.5 模式5：计数器模式

```
contract CounterPattern {
    struct TokenInfo {
        string name;
        uint256 supply;
    }

    mapping(address => uint256) public balances;
    mapping(uint256 => TokenInfo) public tokens;

    uint256 public totalSupply;
    uint256 public tokenCount;

    function mint(address to, uint256 amount) public {
        balances[to] += amount;
        totalSupply += amount;  // 保持总量一致
    }

    function createToken(string memory name, uint256 supply) public {
        tokens[tokenCount++] = TokenInfo(name, supply);
    }
}
```

使用场景：代币系统、供应量管理

---

8. 实战练习

练习1：完整用户管理系统

需求：

创建一个完整的用户管理系统，实现以下功能：

1. 用户注册（包含name、email）
2. 更新个人资料
3. 存款功能（payable）
4. 查询用户信息
5. 获取所有用户列表

6. 分批查询用户
7. 限制最多1000个用户

参考代码框架:

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

contract UserManagementSystem {
    // TODO: 定义User结构体
    struct User {
        // name, email, balance, registeredAt, exists
    }

    // TODO: 定义数据存储
    mapping(address => User) public users;
    address[] public userAddresses;
    uint256 public userCount;
    uint256 public constant MAX_USERS = 1000;

    // TODO: 实现注册功能
    function register(string memory name, string memory email) public {
        // 检查是否已注册
        // 检查是否达到上限
        // 创建用户
        // 添加到列表
        // 更新计数
    }

    // TODO: 实现其他功能...
}
```

完整参考答案:

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

contract UserManagementSystem {
    struct User {
        string name;
        string email;
        uint256 balance;
        uint256 registeredAt;
        bool exists;
    }

    mapping(address => User) public users;
    address[] public userAddresses;
    uint256 public userCount;
    uint256 public constant MAX_USERS = 1000;
```

```
    event UserRegistered(address indexed user, string name);
    event UserUpdated(address indexed user);
    event Deposit(address indexed user, uint256 amount);

    function register(string memory name, string memory email) public {
        require(!users[msg.sender].exists, "Already registered");
        require(userCount < MAX_USERS, "Max users reached");
        require(bytes(name).length > 0, "Name required");
        require(bytes(email).length > 0, "Email required");

        users[msg.sender] = User({
            name: name,
            email: email,
            balance: 0,
            registeredAt: block.timestamp,
            exists: true
        });

        userAddresses.push(msg.sender);
        userCount++;

        emit UserRegistered(msg.sender, name);
    }

    function updateProfile(string memory name, string memory email) public {
        require(users[msg.sender].exists, "Not registered");

        users[msg.sender].name = name;
        users[msg.sender].email = email;

        emit UserUpdated(msg.sender);
    }

    function deposit() public payable {
        require(users[msg.sender].exists, "Not registered");
        require(msg.value > 0, "Must send ETH");

        users[msg.sender].balance += msg.value;

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

    function getUserInfo(address user) public view returns (User memory) {
        require(users[user].exists, "User not found");
        return users[user];
    }

    function getAllUsers() public view returns (address[] memory) {
        return userAddresses;
    }

    function getUsersByRange(
```

```
        uint256 start,
        uint256 end
    ) public view returns (address[] memory) {
        require(start < end, "Invalid range");
        require(end <= userAddresses.length, "End out of bounds");

        uint256 length = end - start;
        address[] memory result = new address[](length);

        for(uint256 i = 0; i < length; i++) {
            result[i] = userAddresses[start + i];
        }

        return result;
    }

    function isRegistered(address user) public view returns (bool) {
        return users[user].exists;
    }
}
```

练习2：投票系统

需求：

创建一个提案投票系统：

1. 定义Proposal结构体（包含voters的mapping）
2. 支持创建提案
3. 支持投票（每人只能投一次）
4. 查询提案信息
5. 获取获胜提案

参考答案：

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

contract VotingSystem {
    struct Proposal {
        string description;
        uint256 voteCount;
        uint256 deadline;
        bool executed;
        mapping(address => bool) voters;
    }

    mapping(uint256 => Proposal) public proposals;
    uint256 public proposalCount;

    event ProposalCreated(uint256 indexed proposalId, string description);
    event Voted(uint256 indexed proposalId, address indexed voter);
```

```
    function createProposal(
        string memory description,
        uint256 duration
    ) public returns (uint256) {
        require(bytes(description).length > 0, "Description required");
        require(duration > 0, "Duration must be positive");

        uint256 proposalId = proposalCount++;

        Proposal storage p = proposals[proposalId];
        p.description = description;
        p.voteCount = 0;
        p.deadline = block.timestamp + duration;
        p.executed = false;

        emit ProposalCreated(proposalId, description);

        return proposalId;
    }

    function vote(uint256 proposalId) public {
        require(proposalId < proposalCount, "Proposal does not exist");

        Proposal storage p = proposals[proposalId];

        require(block.timestamp < p.deadline, "Voting has ended");
        require(!p.voters[msg.sender], "Already voted");

        p.voters[msg.sender] = true;
        p.voteCount++;

        emit Voted(proposalId, msg.sender);
    }

    function hasVoted(
        uint256 proposalId,
        address voter
    ) public view returns (bool) {
        require(proposalId < proposalCount, "Proposal does not exist");
        return proposals[proposalId].voters[voter];
    }

    function getProposalInfo(uint256 proposalId) public view returns (
        string memory description,
        uint256 voteCount,
        uint256 deadline,
        bool executed
    ) {
        require(proposalId < proposalCount, "Proposal does not exist");

        Proposal storage p = proposals[proposalId];
```

```
        return (p.description, p.voteCount, p.deadline, p.executed);
    }

    function getWinningProposal() public view returns (uint256 winningProposalId) {
        uint256 maxVotes = 0;

        for(uint256 i = 0; i < proposalCount; i++) {
            if(proposals[i].voteCount > maxVotes) {
                maxVotes = proposals[i].voteCount;
                winningProposalId = i;
            }
        }

        return winningProposalId;
    }
}
```

练习3：NFT市场

挑战任务：

创建一个简单的NFT市场合约：

1. 定义NFT结构体（id、owner、price、forSale）
2. 铸造NFT功能
3. 上架/下架功能
4. 购买功能
5. 查询所有在售NFT

提示：

- 使用ID自增模式
- 使用mapping存储NFT
- 使用array追踪在售列表

---

9. 常见问题解答

Q1：为什么mapping不能遍历?

答：Mapping的底层实现决定了它不能遍历。

技术原因：

1. **不存储键列表**：Mapping只存储值，不存储键
2. **哈希存储**：通过哈希函数计算存储位置
3. **无限键空间**：理论上所有可能的键都"存在"
4. **Gas成本**：如果要遍历，成本无法预估

解决方案：使用Mapping+Array组合模式

Q2：如何实现可遍历的mapping?

答：使用Mapping+Array组合模式。

```
mapping(address => uint256) public data;  // 存储数据
address[] public keys;                    // 存储键列表
mapping(address => bool) public exists;   // 快速检查

function add(address key, uint256 value) public {
    if(!exists[key]) {
        keys.push(key);
        exists[key] = true;
    }
    data[key] = value;
}

function iterate() public view returns (uint256[] memory) {
    uint256[] memory values = new uint256[](keys.length);
    for(uint256 i = 0; i < keys.length; i++) {
        values[i] = data[keys[i]];
    }
    return values;
}
```

Q3：struct中包含mapping有什么限制?

答：包含mapping的struct有严格限制：

1. **只能在storage中使用**：不能在memory或calldata
2. **不能作为函数参数**：无法传递
3. **不能作为返回值**：无法返回
4. **不能在数组中**：不能创建数组

原因：Mapping的存储特性决定了它只能在永久存储中。

Q4：mapping + struct组合的优势是什么?

答：这是最强大的数据结构组合。

优势：

1. **快速查找**：O(1)时间复杂度
2. **组织复杂数据**：struct组织多个相关字段
3. **代码清晰**：语义明确
4. **功能强大**：几乎可以建模任何数据关系

示例对比：

```
// 不使用struct（混乱）
mapping(address => string) names;
mapping(address => uint256) balances;
mapping(address => bool) active;
mapping(address => uint256) registeredAt;
```

```
// 使用struct（清晰）
struct User {
    string name;
    uint256 balance;
    bool active;
    uint256 registeredAt;
}
mapping(address => User) users;
```

Q5：什么时候使用嵌套mapping?

答：当需要建模二维或多维关系时使用。

典型场景：

1. **授权关系**：`mapping(address => mapping(address => uint256)) allowance`
2. **多代币余额**：`mapping(address => mapping(address => uint256)) tokenBalances`
3. **关注关系**：`mapping(address => mapping(address => bool)) isFollowing`
4. **游戏数据**：`mapping(address => mapping(uint256 => Score)) gameScores`

Q6：如何删除mapping中的元素?

答：使用`delete`关键字重置为默认值。

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

function deleteBalance(address user) public {
    delete balances[user];  // 重置为0
}
```

注意：

- delete不是真正删除，而是重置为默认值
- 可以获得部分Gas退款
- 如果配合array使用，需要同时维护array

Q7：exists字段为什么重要?

答：exists字段用于区分"默认值"和"设置的值"。

问题场景：

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

balances[user]  // 返回0
// 无法区分：
// 1. 用户余额确实是0
// 2. 用户从未注册
```

解决方案：

```
struct User {
    uint256 balance;
    bool exists;
}

mapping(address => User) users;

// 现在可以区分
if(!users[user].exists) {
    // 用户不存在
} else if(users[user].balance == 0) {
    // 用户存在，余额为0
}
```

---

10. 知识点总结

Mapping特性总结

核心特性：

- 键值对存储结构
- O(1)时间复杂度
- 基于哈希表实现
- 只能用于storage

五大特性：

1. 所有键都"存在"（返回默认值）
2. 不存储键列表
3. 不能遍历
4. 只能用于storage
5. 不能作为参数/返回值

操作支持：

- 支持：赋值、查询、delete单个值
- 不支持：遍历、获取长度、delete整个mapping

Struct特性总结

定义和使用：

- 自定义复合数据类型
- 组织相关数据
- 提高代码可读性

三种创建方式：

1. 逐个赋值（灵活）
2. 构造器语法（简洁）
3. 键值对语法（推荐）

存储位置：

- Storage：永久存储
- Memory：临时存储
- Calldata：只读参数

特殊限制：

- 包含mapping的struct只能在storage
- 不能作为参数和返回值（如果包含mapping）

组合模式总结

Mapping + Struct：

- 快速查找复杂数据
- 代码清晰
- 添加exists字段标记存在

Mapping + Array：

- 实现可遍历的mapping
- O(1)查找 + 遍历能力
- 需要维护一致性

Mapping + Struct + Array：

- 最完整的模式
- 快速查找 + 复杂数据 + 遍历
- 几乎所有项目都使用

关键设计原则：

1. 使用struct组织数据
2. 使用mapping快速查找
3. 使用array实现遍历
4. 添加exists标记
5. 添加计数器

---

11. 学习检查清单

完成本课后，你应该能够：

Mapping基础：

- [ ] 理解mapping的工作原理
- [ ] 会使用mapping进行键值存储
- [ ] 理解mapping的五大特性
- [ ] 知道mapping的限制

嵌套Mapping：

- [ ] 会使用嵌套mapping
- [ ] 理解ERC20授权机制
- [ ] 会设计多维数据关系

Mapping与Array组合：

- [ ] 理解为什么需要组合使用
- [ ] 会实现可遍历的mapping
- [ ] 会维护数据一致性
- [ ] 会实现快速删除

Struct：

- [ ] 会定义struct
- [ ] 会使用三种创建方式
- [ ] 理解storage/memory/calldata的区别
- [ ] 会访问和修改struct字段

Mapping与Struct组合：

- [ ] 理解exists字段的重要性
- [ ] 会实现完整组合模式
- [ ] 会设计复杂数据结构
- [ ] 理解包含mapping的struct的限制

设计模式：

- [ ] 掌握用户管理模式
- [ ] 掌握ID自增模式
- [ ] 掌握双向映射模式
- [ ] 掌握一对多关系模式
- [ ] 掌握计数器模式

---

12. 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. **分析真实项目**：研究OpenZeppelin、Uniswap等项目的数据结构
4. **准备学习第3.3课**：函数与修饰符

下节课预告：第3.3课 - 函数与修饰符

我们将学习：

- 函数可见性（public/private/internal/external）

- 状态修饰符（view/pure/payable）
- 自定义modifier
- 函数重载
- 返回值处理

---

13. 扩展资源

官方文档：

- Solidity Mapping文档：https://docs.soliditylang.org/en/latest/types.html#mapping-types
- Solidity Struct文档：https://docs.soliditylang.org/en/latest/types.html#structs

学习资源：

- Solidity by Example - Mapping：https://solidity-by-example.org/mapping/
- Solidity by Example - Struct：https://solidity-by-example.org/structs/

实战项目：
研究开源项目中的数据结构设计：

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

设计模式：

- Solidity Patterns：https://fravoll.github.io/solidity-patterns/
- Smart Contract Best Practices

工具推荐：

- Remix IDE：在线开发环境
- Hardhat：专业开发框架
- Etherscan：查看已部署合约

课程结束

恭喜你完成第3.2课！Mapping和Struct是智能合约开发中最核心的数据结构，掌握它们的组合使用是成为优秀Solidity开发者的关键。

核心要点回顾：

- Mapping：快速查找，不能遍历
- Struct：组织数据，提高可读性
- 组合模式：Mapping+Struct+Array = 最强大的数据结构
- exists字段：明确标记存在，避免混淆

记住：几乎所有专业的智能合约都使用Mapping+Struct+Array组合模式。掌握这个模式 = 掌握数据设计的精髓！

继续加油，下节课见！