Solidity智能合约开发知识

第3.1课：数组（Arrays）

学习目标：掌握Solidity中定长数组和动态数组的使用、理解数组操作的Gas优化技巧、学会安全地管理数组、避免常见的数组陷阱

预计学习时间：2小时

难度等级：入门进阶

目录

---

1. 数组基础概念

1.1 什么是数组

数组是存储相同类型元素的集合，是智能合约开发中最常用的数据结构之一。数组允许我们在一个变量中存储多个同类型的值。

数组的基本特征：

- **同质性**：数组中所有元素必须是相同类型
- **索引访问**：通过索引（从0开始）访问元素
- **顺序存储**：元素按照添加顺序存储
- **长度属性**：可以通过`length`属性获取数组长度

1.2 数组的分类

Solidity中的数组分为两大类：

定长数组（Fixed-size Array）：

```
uint[5] public fixedArray;  // 长度固定为5
```

特点：

- 长度在声明时确定
- 长度永远不可改变
- 所有元素初始化为默认值（数字类型为0）
- 不能使用push或pop方法

动态数组（Dynamic Array）：

```
uint[] public dynamicArray;  // 长度可变
```

特点：

- 长度可以动态改变
- 可以使用push添加元素
- 可以使用pop删除最后一个元素
- length是可变属性

1.3 数组类型对比

特性	定长数组	动态数组
声明语法	`uint[5]`	`uint[]`
初始化	`[1, 2, 3, 4, 5]`	`[1, 2, 3]`
长度	固定不变	可以改变
push方法	不支持	支持
pop方法	不支持	支持
length属性	常量	可变
Gas成本	相对较低	相对较高
使用场景	固定数量元素	动态数量元素

1.4 数组在区块链中的作用

实际应用场景：

1. **用户列表管理**：

```
address[] public members;  // 存储所有会员地址
```

2. **历史记录追踪**：

```
uint[] public prices;  // 记录价格历史
```

3. **批量数据处理**：

```
uint[] public pendingTransactions;  // 待处理交易列表
```

4. **投票选项**：

```
string[] public candidates;  // 候选人名单
```

---

2. 定长数组

2.1 定长数组的声明和初始化

基本声明：

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

contract FixedArrayExample {
    // 声明定长数组（未初始化，默认值为0）
    uint[5] public fixedArray;

    // 声明并初始化
    uint[3] public numbers = [1, 2, 3];

    // 其他类型的定长数组
    address[10] public addresses;
    bool[4] public flags = [true, false, true, false];
    bytes32[2] public hashes;
}
```

默认值：

未初始化的定长数组元素都有默认值：

- `uint`类型：0
- `bool`类型：false
- `address`类型：0x0000000000000000000000000000000000000000
- `bytes32`类型：0x0000000000000000000000000000000000000000000000000000000000000000

2.2 访问和修改定长数组

```
contract FixedArrayOperations {
    uint[5] public numbers;

    constructor() {
        // 初始化数组
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
```

```
        numbers[4] = 50;
    }

    // 读取元素
    function getElement(uint index) public view returns (uint) {
        require(index < 5, "Index out of bounds");
        return numbers[index];
    }

    // 修改元素
    function setElement(uint index, uint value) public {
        require(index < 5, "Index out of bounds");
        numbers[index] = value;
    }

    // 获取整个数组
    function getAllNumbers() public view returns (uint[5] memory) {
        return numbers;
    }

    // 获取数组长度（始终为5）
    function getLength() public pure returns (uint) {
        uint[5] memory arr;
        return arr.length;  // 返回5
    }
}
```

2.3 定长数组的限制

不能使用的操作：

```
contract FixedArrayLimitations {
    uint[5] public numbers = [1, 2, 3, 4, 5];

    function attemptPush() public {
        // 编译错误：定长数组不支持push
        // numbers.push(6);  // Error!
    }

    function attemptPop() public {
        // 编译错误：定长数组不支持pop
        // numbers.pop();  // Error!
    }

    // 可以使用delete设置为默认值
    function resetElement(uint index) public {
        delete numbers[index];  // 将numbers[index]设为0
    }
}
```

2.4 定长数组的使用场景

适合使用定长数组的场景：

1. **固定数量的配置参数：**

```
contract WeeklySchedule {
    // 一周7天的工作时间（小时）
    uint8[7] public workingHours = [8, 8, 8, 8, 8, 0, 0];
}
```

2. **预定义的选项集合：**

```
contract VotingSystem {
    // 固定的4个选项
    string[4] public options = ["Option A", "Option B", "Option C", "Option D"];
}
```

3. **固定大小的数据记录：**

```
contract GameBoard {
    // 3x3的井字棋棋盘
    uint8[9] public board;  // 0=空, 1=X, 2=O
}
```

---

3. 动态数组

3.1 动态数组的声明和初始化

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

contract DynamicArrayExample {
    // 声明动态数组（空数组）
    uint[] public numbers;

    // 声明并初始化
    address[] public users = [
        0x5B38Da6a701c568545dCfcB03FcB875f56beddC4,
        0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
    ];

    // 其他类型的动态数组
    string[] public names;
    bool[] public flags;
    bytes[] public dataList;

    constructor() {
```

```
        // 在构造函数中添加初始元素
        numbers.push(1);
        numbers.push(2);
        numbers.push(3);
    }
}
```

3.2 动态数组的基本操作

添加元素（push）：

```
contract DynamicArrayPush {
    uint[] public numbers;

    // 添加单个元素
    function addNumber(uint value) public {
        numbers.push(value);
    }

    // 批量添加元素
    function addMultiple(uint[] memory values) public {
        for(uint i = 0; i < values.length; i++) {
            numbers.push(values[i]);
        }
    }

    // push的返回值（Solidity 0.6.0+）
    function pushAndGetLength(uint value) public returns (uint) {
        numbers.push(value);
        return numbers.length;
    }
}
```

删除最后元素（pop）：

```
contract DynamicArrayPop {
    uint[] public numbers = [1, 2, 3, 4, 5];

    // 删除最后一个元素
    function removeLastElement() public {
        require(numbers.length > 0, "Array is empty");
        numbers.pop();
    }

    // 删除多个元素
    function removeMultiple(uint count) public {
        require(count <= numbers.length, "Not enough elements");
        for(uint i = 0; i < count; i++) {
            numbers.pop();
        }
    }
```

```
    // 获取并删除最后元素
    function popAndReturn() public returns (uint) {
        require(numbers.length > 0, "Array is empty");
        uint lastValue = numbers[numbers.length - 1];
        numbers.pop();
        return lastValue;
    }
}
```

获取长度（length）：

```
contract ArrayLength {
    uint[] public data;

    function getLength() public view returns (uint) {
        return data.length;
    }

    function isEmpty() public view returns (bool) {
        return data.length == 0;
    }

    function addAndCheckLength(uint value) public returns (uint) {
        data.push(value);
        return data.length;  // 返回添加后的长度
    }
}
```

3.3 Storage数组 vs Memory数组

Storage数组（状态变量）：

```
contract StorageArray {
    uint[] public storageArray;  // 存储在区块链上

    function addToStorage(uint value) public {
        storageArray.push(value);  // 修改永久保存
    }

    function getStorage() public view returns (uint[] memory) {
        return storageArray;  // 返回storage数组的副本
    }
}
```

Memory数组（临时变量）：

```
contract MemoryArray {
    // 在memory中创建数组
    function createMemoryArray() public pure returns (uint[] memory) {
```

```
        // 必须指定长度
        uint[] memory arr = new uint[](5);

        // 可以赋值
        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;
        arr[3] = 4;
        arr[4] = 5;

        return arr;
    }

    // Memory数组的限制
    function memoryArrayLimitations() public pure {
        uint[] memory arr = new uint[](5);

        // 不能push（编译错误）
        // arr.push(6);  // Error!

        // 不能pop（编译错误）
        // arr.pop();  // Error!

        // 不能改变长度
        // arr.length = 10;  // Error!
    }
}
```

Storage vs Memory对比：

特性	Storage数组	Memory数组
存储位置	区块链上（永久）	内存中（临时）
创建方式	状态变量声明	`new uint[](n)`
长度	可变（动态数组）	固定（创建时确定）
push/pop	支持	不支持
Gas成本	高（写入区块链）	低（仅计算）
生命周期	永久	函数执行期间

4. 数组基本操作

4.1 访问数组元素

```
contract ArrayAccess {
    uint[] public numbers = [10, 20, 30, 40, 50];
```

```
    // 通过索引访问
    function getElement(uint index) public view returns (uint) {
        require(index < numbers.length, "Index out of bounds");
        return numbers[index];
    }

    // 获取第一个元素
    function getFirst() public view returns (uint) {
        require(numbers.length > 0, "Array is empty");
        return numbers[0];
    }

    // 获取最后一个元素
    function getLast() public view returns (uint) {
        require(numbers.length > 0, "Array is empty");
        return numbers[numbers.length - 1];
    }

    // 安全访问（返回bool表示是否成功）
    function tryGetElement(uint index) public view returns (bool, uint) {
        if (index >= numbers.length) {
            return (false, 0);
        }
        return (true, numbers[index]);
    }
}
```

4.2 修改数组元素

```
contract ArrayModification {
    uint[] public numbers = [1, 2, 3, 4, 5];

    // 修改指定位置的元素
    function updateElement(uint index, uint value) public {
        require(index < numbers.length, "Index out of bounds");
        numbers[index] = value;
    }

    // 递增元素值
    function incrementElement(uint index) public {
        require(index < numbers.length, "Index out of bounds");
        numbers[index] += 1;
    }

    // 批量修改
    function updateMultiple(uint[] memory indices, uint[] memory values) public {
        require(indices.length == values.length, "Length mismatch");

        for(uint i = 0; i < indices.length; i++) {
            require(indices[i] < numbers.length, "Index out of bounds");
```

```
            numbers[indices[i]] = values[i];
        }
    }

    // 重置所有元素为0
    function resetAll() public {
        delete numbers;  // 清空数组，length变为0
    }
}
```

4.3 获取整个数组

```
contract GetWholeArray {
    uint[] public numbers;

    constructor() {
        numbers.push(1);
        numbers.push(2);
        numbers.push(3);
    }

    // 返回整个数组
    function getAllNumbers() public view returns (uint[] memory) {
        return numbers;
    }

    // 返回数组副本并进行处理
    function getDoubledArray() public view returns (uint[] memory) {
        uint[] memory result = new uint[](numbers.length);
        for(uint i = 0; i < numbers.length; i++) {
            result[i] = numbers[i] * 2;
        }
        return result;
    }
}
```

4.4 delete操作的陷阱

delete的行为：

```
contract DeleteTrap {
    uint[] public numbers = [1, 2, 3, 4, 5];

    // delete单个元素
    function deleteElement(uint index) public {
        require(index < numbers.length, "Index out of bounds");
        delete numbers[index];
        // 结果：numbers[index] = 0
        // 重要：length不变！
    }
```

```
    // 演示delete的问题
    function demonstrateDeleteProblem() public {
        // 初始: [1, 2, 3, 4, 5], length = 5
        delete numbers[2];
        // 结果: [1, 2, 0, 4, 5], length = 5
        // 注意: 3变成了0, 但数组长度仍然是5
        // 留下了一个"空洞"
    }

    // delete整个数组
    function deleteArray() public {
        delete numbers;
        // 结果: [], length = 0
        // 清空整个数组
    }

    // 检查空洞
    function hasZeros() public view returns (bool) {
        for(uint i = 0; i < numbers.length; i++) {
            if(numbers[i] == 0) {
                return true;  // 发现空洞
            }
        }
        return false;
    }
}
```

delete操作总结:

操作	效果	length变化	注意事项
`delete arr[i]`	元素重置为0	不变	留下空洞
`delete arr`	清空数组	变为0	完全清空
`arr.pop()`	删除最后元素	减1	真正删除

---

5. 删除数组元素

5.1 删除方法对比

在Solidity中，真正删除数组中间的元素需要特殊处理。有两种主要方法：

方法1：保持顺序（移动元素）

优点：保持元素的原有顺序
缺点：Gas消耗高（需要移动多个元素）

方法2：快速删除（不保序）

优点：Gas消耗低（只需两步操作）
缺点：不保持元素顺序

5.2 方法1：保持顺序删除

```
contract OrderedRemoval {
    uint[] public numbers;

    constructor() {
        numbers = [1, 2, 3, 4, 5];
    }

    // 删除指定索引的元素，保持顺序
    function removeOrdered(uint index) public {
        require(index < numbers.length, "Index out of bounds");

        // 将后面的元素向前移动
        for(uint i = index; i < numbers.length - 1; i++) {
            numbers[i] = numbers[i + 1];
        }

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

    // 示例演示
    function demonstrateOrderedRemoval() public {
        // 初始：[1, 2, 3, 4, 5]
        removeOrdered(1);  // 删除索引1的元素（值为2）
        // 结果：[1, 3, 4, 5]
        // 顺序保持：3、4、5向前移动
    }
}
```

执行过程详解：

```
初始数组：[1, 2, 3, 4, 5]
删除索引1（值为2）：

步骤1：i=1, numbers[1] = numbers[2]  → [1, 3, 3, 4, 5]
步骤2：i=2, numbers[2] = numbers[3]  → [1, 3, 4, 4, 5]
步骤3：i=3, numbers[3] = numbers[4]  → [1, 3, 4, 5, 5]
步骤4：pop()                         → [1, 3, 4, 5]

最终结果：[1, 3, 4, 5]
```

Gas分析：

假设删除索引为`index`，数组长度为`n`：

- 需要移动的元素数量：`n - index - 1`

- 每次赋值约消耗：5,000 gas（storage写入）
- 总Gas消耗：约`5,000 × (n - index - 1) + 5,000`（pop）

5.3 方法2：快速删除（不保序）

```
contract UnorderedRemoval {
    uint[] public numbers;

    constructor() {
        numbers = [1, 2, 3, 4, 5];
    }

    // 快速删除，不保持顺序
    function removeUnordered(uint index) public {
        require(index < numbers.length, "Index out of bounds");

        // 用最后一个元素替换要删除的元素
        numbers[index] = numbers[numbers.length - 1];

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

    // 示例演示
    function demonstrateUnorderedRemoval() public {
        // 初始：[1, 2, 3, 4, 5]
        removeUnordered(1);  // 删除索引1的元素（值为2）
        // 结果：[1, 5, 3, 4]
        // 最后的5移到了索引1的位置
    }
}
```

执行过程详解：

```
初始数组：[1, 2, 3, 4, 5]
删除索引1（值为2）：

步骤1：numbers[1] = numbers[4]  → [1, 5, 3, 4, 5]
步骤2：pop()                    → [1, 5, 3, 4]

最终结果：[1, 5, 3, 4]
```

Gas分析：

- 一次赋值：约5,000 gas
- 一次pop：约5,000 gas
- 总Gas消耗：约10,000 gas（常量，不随数组大小变化）

5.4 删除方法选择指南

```
contract RemovalComparison {
    uint[] public orderedArray;
    uint[] public unorderedArray;

    // 初始化两个相同的数组
    function initialize() public {
        delete orderedArray;
        delete unorderedArray;

        for(uint i = 1; i <= 100; i++) {
            orderedArray.push(i);
            unorderedArray.push(i);
        }
    }

    // 保序删除（Gas高）
    function testOrderedRemoval() public {
        require(orderedArray.length > 50, "Not enough elements");

        // 删除中间元素（索引50）
        for(uint i = 50; i < orderedArray.length - 1; i++) {
            orderedArray[i] = orderedArray[i + 1];
        }
        orderedArray.pop();
        // Gas: 约 250,000（需要移动50个元素）
    }

    // 快速删除（Gas低）
    function testUnorderedRemoval() public {
        require(unorderedArray.length > 50, "Not enough elements");

        // 删除中间元素（索引50）
        unorderedArray[50] = unorderedArray[unorderedArray.length - 1];
        unorderedArray.pop();
        // Gas: 约 10,000（固定消耗）
    }
}
```

何时使用哪种方法：

使用场景	推荐方法	原因
需要保持顺序（如排行榜）	保序删除	顺序重要
数组很小（<20个元素）	保序删除	Gas差异小
数组较大且顺序不重要	快速删除	节省Gas
用户列表、ID列表	快速删除	顺序无关紧要
历史记录、时间序列	保序删除	时间顺序重要

6. 数组遍历

6.1 基本遍历方法

```
contract ArrayIteration {
    uint[] public numbers;

    constructor() {
        for(uint i = 1; i <= 10; i++) {
            numbers.push(i);
        }
    }

    // 基本遍历
    function iterateArray() public view returns (uint) {
        uint sum = 0;
        for(uint i = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }
        return sum;
    }

    // 查找元素
    function findElement(uint value) public view returns (bool, uint) {
        for(uint i = 0; i < numbers.length; i++) {
            if(numbers[i] == value) {
                return (true, i);  // 找到，返回索引
            }
        }
        return (false, 0);  // 未找到
    }

    // 过滤元素
    function filterGreaterThan(uint threshold) public view returns (uint[] memory) {
        // 第一次遍历：计数
        uint count = 0;
        for(uint i = 0; i < numbers.length; i++) {
            if(numbers[i] > threshold) {
                count++;
            }
        }

        // 创建结果数组
        uint[] memory result = new uint[](count);

        // 第二次遍历：填充
        uint index = 0;
        for(uint i = 0; i < numbers.length; i++) {
            if(numbers[i] > threshold) {
                result[index] = numbers[i];
                index++;
```

```
            }
        }

        return result;
    }
}
```

6.2 遍历的Gas风险

危险示例：无限增长的数组

```
contract DangerousIteration {
    uint[] public data;

    // 危险：允许无限添加
    function addData(uint value) public {
        data.push(value);
        // 问题：没有限制数组大小
    }

    // 危险：遍历可能gas耗尽
    function sumAll() public view returns (uint) {
        uint total = 0;
        for(uint i = 0; i < data.length; i++) {
            total += data[i];
        }
        return total;
        // 如果data有10,000个元素，这个函数将无法执行！
    }
}
```

问题分析：

假设数组有10,000个元素：

- 每次循环约消耗：5,000 gas（读取storage）
- 总需求：10,000 × 5,000 = 50,000,000 gas
- 以太坊区块gas限制：约30,000,000 gas
- 结果：函数永远无法执行，合约"僵死"

真实案例影响：

多个DeFi项目因为大数组遍历导致：

- 用户无法提取资金
- 合约功能失效
- 需要重新部署合约
- 造成重大损失

6.3 安全遍历实践

```
contract SafeIteration {
    uint[] public data;
    uint public constant MAX_ARRAY_SIZE = 100;

    // 安全实践1：限制数组大小
    function safePush(uint value) public {
        require(data.length < MAX_ARRAY_SIZE, "Array is full");
        data.push(value);
    }

    // 安全实践2：分批处理
    function sumRange(uint start, uint end) public view returns (uint) {
        require(start < end, "Invalid range");
        require(end <= data.length, "End out of bounds");
        require(end - start <= 50, "Range too large");  // 限制单次处理量

        uint total = 0;
        for(uint i = start; i < end; i++) {
            total += data[i];
        }
        return total;
    }

    // 安全实践3：提供分页查询
    function getPage(uint pageNumber, uint pageSize)
        public view returns (uint[] memory)
    {
        require(pageSize <= 20, "Page size too large");

        uint start = pageNumber * pageSize;
        require(start < data.length, "Page out of bounds");

        uint end = start + pageSize;
        if(end > data.length) {
            end = data.length;
        }

        uint[] memory result = new uint[](end - start);
        for(uint i = start; i < end; i++) {
            result[i - start] = data[i];
        }

        return result;
    }
}
```

6.4 遍历优化技巧

```
contract IterationOptimization {
    uint[] public numbers;
```

```
    // 优化前：每次读取length
    function sumBad() public view returns (uint) {
        uint total = 0;
        for(uint i = 0; i < numbers.length; i++) {  // 每次循环读取length
            total += numbers[i];
        }
        return total;
        // Gas: 约 25,000 (100个元素)
    }

    // 优化后：缓存length
    function sumGood() public view returns (uint) {
        uint total = 0;
        uint len = numbers.length;  // 缓存length
        for(uint i = 0; i < len; i++) {
            total += numbers[i];
        }
        return total;
        // Gas: 约 23,000 (100个元素)
        // 节省约 8%
    }

    // 优化：unchecked (谨慎使用)
    function sumOptimized() public view returns (uint) {
        uint total = 0;
        uint len = numbers.length;
        for(uint i = 0; i < len; ) {
            total += numbers[i];
            unchecked { i++; }  // i不会溢出
        }
        return total;
        // 进一步节省gas
    }
}
```

7. 多维数组

7.1 二维数组声明

```
contract MultiDimensionalArrays {
    // 动态二维数组
    uint[][] public matrix;

    // 定长二维数组
    uint[3][4] public fixedMatrix;  // 4行，每行3个元素

    // 混合数组
    uint[][5] public mixedArray;  // 5个动态数组
    uint[3][] public mixedArray2;  // 动态数量的定长数组
}
```

重要：数组声明的顺序

在Solidity中，多维数组的声明顺序与其他语言相反：

```
contract ArrayOrderConfusion {
    // uint[3][4] 表示什么?
    uint[3][4] public arr;

    // 正确理解:
    // 这是4个长度为3的数组
    // 不是3行4列的矩阵!

    function demonstrateOrder() public {
        // arr[0] 是一个长度为3的数组
        // arr[1] 是一个长度为3的数组
        // arr[2] 是一个长度为3的数组
        // arr[3] 是一个长度为3的数组

        // 总共4个数组，每个数组有3个元素
    }
}
```

记忆技巧：从右向左读

```
uint[3][4]
      ↑  ↑
      |  |
      |  └─ 4个
      └──── 长度为3的数组
```

7.2 二维数组操作

```
contract TwoDimensionalArray {
    uint[][] public matrix;

    // 添加一行
    function addRow(uint[] memory row) public {
```

```
        matrix.push(row);
    }

    // 初始化矩阵
    function initializeMatrix() public {
        delete matrix;  // 清空

        // 添加3行
        uint[] memory row1 = new uint[](3);
        row1[0] = 1; row1[1] = 2; row1[2] = 3;
        matrix.push(row1);

        uint[] memory row2 = new uint[](3);
        row2[0] = 4; row2[1] = 5; row2[2] = 6;
        matrix.push(row2);

        uint[] memory row3 = new uint[](3);
        row3[0] = 7; row3[1] = 8; row3[2] = 9;
        matrix.push(row3);

        // 结果:
        // [1, 2, 3]
        // [4, 5, 6]
        // [7, 8, 9]
    }

    // 访问元素
    function getElement(uint row, uint col) public view returns (uint) {
        require(row < matrix.length, "Row out of bounds");
        require(col < matrix[row].length, "Column out of bounds");
        return matrix[row][col];
    }

    // 修改元素
    function setElement(uint row, uint col, uint value) public {
        require(row < matrix.length, "Row out of bounds");
        require(col < matrix[row].length, "Column out of bounds");
        matrix[row][col] = value;
    }

    // 获取矩阵维度
    function getDimensions() public view returns (uint rows, uint cols) {
        rows = matrix.length;
        if(rows > 0) {
            cols = matrix[0].length;
        } else {
            cols = 0;
        }
    }
}
```

7.3 二维数组遍历

```
contract MatrixOperations {
    uint[][] public matrix;

    constructor() {
        // 初始化3x3矩阵
        for(uint i = 0; i < 3; i++) {
            uint[] memory row = new uint[](3);
            for(uint j = 0; j < 3; j++) {
                row[j] = i * 3 + j + 1;
            }
            matrix.push(row);
        }
    }

    // 计算矩阵所有元素之和
    function sumMatrix() public view returns (uint) {
        uint total = 0;
        uint rows = matrix.length;

        for(uint i = 0; i < rows; i++) {
            uint cols = matrix[i].length;
            for(uint j = 0; j < cols; j++) {
                total += matrix[i][j];
            }
        }
        return total;
    }

    // 查找元素位置
    function findElement(uint value) public view returns (bool, uint, uint) {
        for(uint i = 0; i < matrix.length; i++) {
            for(uint j = 0; j < matrix[i].length; j++) {
                if(matrix[i][j] == value) {
                    return (true, i, j);  // 找到，返回行列
                }
            }
        }
        return (false, 0, 0);  // 未找到
    }

    // 获取指定行
    function getRow(uint row) public view returns (uint[] memory) {
        require(row < matrix.length, "Row out of bounds");
        return matrix[row];
    }
}
```

7.4 三维及更高维数组

```
contract HighDimensionalArrays {
    // 三维数组
    uint[][][] public cube;

    // 添加一个2D平面
    function addPlane(uint[][] memory plane) public {
        cube.push(plane);
    }

    // 访问三维元素
    function getElement3D(uint x, uint y, uint z) public view returns (uint) {
        require(x < cube.length, "X out of bounds");
        require(y < cube[x].length, "Y out of bounds");
        require(z < cube[x][y].length, "Z out of bounds");
        return cube[x][y][z];
    }

    // 警告：高维数组非常复杂，通常应避免使用
    // 考虑使用mapping代替
}
```

高维数组的问题：

1. **Gas消耗极高**：每增加一维，Gas成本指数增长
2. **代码复杂**：难以理解和维护
3. **遍历困难**：多层嵌套循环容易出错
4. **替代方案**：使用mapping组合或结构体

更好的替代方案：

```
contract BetterAlternative {
    // 使用mapping替代三维数组
    mapping(uint => mapping(uint => mapping(uint => uint))) public betterCube;

    function setValue(uint x, uint y, uint z, uint value) public {
        betterCube[x][y][z] = value;
    }

    function getValue(uint x, uint y, uint z) public view returns (uint) {
        return betterCube[x][y][z];
    }

    // 优势：
    // - Gas消耗更低
    // - 代码更清晰
    // - 不需要初始化
    // - 支持稀疏数据
}
```

---

8. Gas优化技巧

8.1 优化技巧1：缓存数组长度

问题：每次读取`array.length`都会访问storage，消耗200 gas。

```
contract LengthCaching {
    uint[] public data;

    // 未优化：每次循环读取length
    function sumUnoptimized() public view returns (uint) {
        uint total = 0;
        for(uint i = 0; i < data.length; i++) {  // 每次读取200 gas
            total += data[i];
        }
        return total;
    }
    // 100个元素约消耗：25,000 gas

    // 优化：缓存length
    function sumOptimized() public view returns (uint) {
        uint total = 0;
        uint len = data.length;  // 只读取一次
        for(uint i = 0; i < len; i++) {
            total += data[i];
        }
        return total;
    }
    // 100个元素约消耗：23,000 gas
    // 节省：2,000 gas (8%)
}
```

节省计算：

- 数组长度：n
- 未优化：n × 200 gas（读取length）
- 优化后：1 × 200 gas
- 节省：(n - 1) × 200 gas

8.2 优化技巧2：限制数组最大长度

```
contract ArraySizeLimit {
    uint[] public data;
    uint public constant MAX_ARRAY_SIZE = 100;

    // 限制数组大小
    function safePush(uint value) public {
        require(data.length < MAX_ARRAY_SIZE, "Array is full");
        data.push(value);
    }
```

```
    // 批量添加也要检查
    function safePushMultiple(uint[] memory values) public {
        require(
            data.length + values.length <= MAX_ARRAY_SIZE,
            "Would exceed max size"
        );

        for(uint i = 0; i < values.length; i++) {
            data.push(values[i]);
        }
    }
}
```

为什么限制大小?

1. **防止Gas耗尽**：确保遍历操作可以完成
2. **可预测成本**：用户知道最大Gas消耗
3. **避免DoS攻击**：防止恶意用户填满数组
4. **合约可用性**：确保合约长期可用

推荐的最大长度：

操作复杂度	推荐最大长度
简单读取	≤ 1,000
简单计算	≤ 500
复杂计算	≤ 100
安全保守	≤ 100

8.3 优化技巧3：分批处理

```
contract BatchProcessing {
    uint[] public data;

    // 分批求和
    function sumRange(uint start, uint end) public view returns (uint) {
        require(start < end, "Invalid range");
        require(end <= data.length, "End out of bounds");

        uint total = 0;
        for(uint i = start; i < end; i++) {
            total += data[i];
        }
        return total;
    }

    // 分批删除
```

```
    function deleteRange(uint start, uint count) public {
        require(start + count <= data.length, "Range out of bounds");

        for(uint i = 0; i < count; i++) {
            // 删除start位置count次
            // 每次删除后，start位置的元素都会变化
            removeOrdered(start);
        }
    }

    function removeOrdered(uint index) private {
        require(index < data.length, "Index out of bounds");
        for(uint i = index; i < data.length - 1; i++) {
            data[i] = data[i + 1];
        }
        data.pop();
    }
}
```

使用示例：

```
// 假设有1000个元素的数组
// 不要一次处理全部
// sumRange(0, 1000);  // 可能gas耗尽

// 分批处理
sumRange(0, 100);    // 处理前100个
sumRange(100, 200);  // 处理接下来100个
sumRange(200, 300);  // 继续...
// 可以分多次交易完成
```

8.4 优化技巧4：使用calldata

对于外部函数的数组参数，使用calldata替代memory：

```
contract CalldataOptimization {
    uint[] public stored;

    // 未优化：使用memory
    function processMemory(uint[] memory arr) public {
        for(uint i = 0; i < arr.length; i++) {
            stored.push(arr[i]);
        }
    }
    // Gas: 约 150,000 (100个元素)

    // 优化：使用calldata
    function processCalldata(uint[] calldata arr) external {
        for(uint i = 0; i < arr.length; i++) {
            stored.push(arr[i]);
        }
```

```
    }
    // Gas: 约 120,000 (100个元素)
    // 节省: 30,000 gas (20%)
}
```

calldata vs memory:

特性	memory	calldata
存储位置	内存（临时）	调用数据区
可修改性	可修改	只读
Gas成本	需要复制数据	无需复制
使用场景	内部函数、需要修改	外部函数、只读
函数类型	public/external	仅external

8.5 优化技巧5：避免循环中的storage写入

重要说明：这个优化技巧有严格的适用场景。对于更新数组中的部分元素，直接在循环中写storage通常已经是最优解。只有在特定场景下，memory优化才有效。

场景1：批量更新数组元素（部分更新 - 不适用优化）

以下示例展示了为什么对于部分更新，memory优化反而更差：

```
contract BatchUpdateOptimization {
    uint[] public scores;

    // 初始化函数：创建测试数据（用于演示）
    function initialize() external {
        // 创建10个初始分数: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90
        for(uint i = 0; i < 10; i++) {
            scores.push(i * 10);
        }
    }

    // 方式1：循环中直接写storage（推荐）
    function updateScoresBad(uint[] calldata indices, uint[] calldata newScores) external
{
        require(indices.length == newScores.length, "Arrays length mismatch");
        require(scores.length > 0, "Scores array is empty");

        for(uint i = 0; i < indices.length; i++) {
            require(indices[i] < scores.length, "Index out of bounds");
            scores[indices[i]] = newScores[i];  // 每次循环都写storage
        }
    }
    // Gas: 约 42,024 (更新3个元素，10个元素的数组)
    // 优势：只写入需要更新的元素，不读取其他元素
```

```
    // 方式2：先在memory中处理，最后一次性写入（不推荐）
    function updateScoresGood(uint[] calldata indices, uint[] calldata newScores) external
{
        require(indices.length == newScores.length, "Arrays length mismatch");
        require(scores.length > 0, "Scores array is empty");

        uint len = indices.length;

        // 检查所有索引是否有效
        for(uint i = 0; i < len; i++) {
            require(indices[i] < scores.length, "Index out of bounds");
        }

        // 将storage数组复制到memory（读取所有元素，很贵！）
        uint[] memory tempScores = new uint[](scores.length);
        for(uint i = 0; i < scores.length; i++) {
            tempScores[i] = scores[i];  // 读取所有元素到memory
        }

        // 在memory中更新（不写storage）
        for(uint i = 0; i < len; i++) {
            tempScores[indices[i]] = newScores[i];
        }

        // 一次性写回storage（写入整个数组，很贵！）
        scores = tempScores;
    }
    // Gas：约 54,482（更新3个元素，10个元素的数组）
    // 更差：多了 12,458 gas (29.6%)，不推荐！
}
```

测试结果分析（更新3个元素，数组长度为10）：

方式	Gas消耗	相对updateScoresBad	说明
updateScoresBad	42,024	基准	直接写storage，只更新需要的元素
updateScoresGood	54,482	+29.6%	更差！多了12,458 gas

为什么"优化"版本更差?

1. **读取整个数组到memory**：
    - 需要读取所有10个元素（包括不需要更新的7个）
    - Storage读取成本：约2,100 gas/次
    - 10次读取 = 约21,000 gas
2. **创建memory数组**：
    - 分配内存需要gas开销
    - 数组越大，开销越大
3. **写入整个数组到storage**：

    - 需要写入所有10个元素（包括未修改的7个）
    - Storage写入成本：约20,000 gas/次
    - 10次写入 = 约200,000 gas
4. **直接写storage的优势**：
    - 只写入需要更新的3个元素
    - 不读取其他元素
    - 总成本：3次写入 ≈ 60,000 gas（加上其他开销）

结论：

对于**更新数组中的部分元素**，直接在循环中写storage已经是最优解。只有在以下情况下，memory优化才可能有效：

1. **需要更新大部分或全部元素**（>80%）
2. **数组很小**（<5个元素）
3. **需要复杂的计算**，在memory中计算后再写回

最佳实践：对于部分更新，直接写storage；对于全量替换，考虑使用memory优化。

测试步骤：

1. **部署合约后，先调用 `initialize()`**：
    - 这会创建10个初始分数：`[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]`
2. **然后调用 `updateScoresBad`**：
    - `indices`：`[1,2,3]`
    - `newScores`：`[100,200,300]`
    - 这会更新索引1、2、3的分数为100、200、300
3. **验证结果**：
    - 调用 `scores` 查看数组，应该看到：`[0, 100, 200, 300, 40, 50, 60, 70, 80, 90]`

常见错误：

- ❌ **数组为空**：如果没调用 `initialize()`，`scores` 数组是空的，访问任何索引都会revert
- ❌ **索引越界**：如果索引 >= `scores.length`，会revert
- ✅ **正确做法**：先调用 `initialize()` 初始化数据，再调用更新函数

场景2：全量替换数组（适用优化）

当需要更新数组中的大部分或全部元素时，memory优化可能有效：

```
contract FullArrayUpdate {
    uint[] public data;

    // 未优化：循环中逐个更新
    function updateAllBad(uint[] calldata newData) external {
        require(newData.length == data.length, "Length mismatch");

        for(uint i = 0; i < data.length; i++) {
            data[i] = newData[i];  // 每次循环都写storage
        }
    }
```

```
    // Gas：取决于数组长度
    // 问题：每个元素都要写storage

    // 优化：一次性替换整个数组
    function updateAllGood(uint[] calldata newData) external {
        require(newData.length == data.length, "Length mismatch");

        // 复制到memory
        uint[] memory temp = new uint[](newData.length);
        for(uint i = 0; i < newData.length; i++) {
            temp[i] = newData[i];
        }

        // 一次性替换
        data = temp;
    }
    // Gas：可能更优（取决于数组大小和编译器优化）
    // 注意：需要实际测试验证
}
```

场景3：需要复杂计算的批量更新（适用优化）

当更新需要复杂计算时，在memory中计算后再写入可能更优：

```
contract ComplexCalculation {
    uint[] public results;
    uint public multiplier;

    // 未优化：循环中读取storage、计算、写storage
    function processBad(uint[] calldata inputs) external {
        for(uint i = 0; i < inputs.length; i++) {
            // 每次循环都要读取multiplier（storage读取）
            results.push(inputs[i] * multiplier);  // 读storage + 计算 + 写storage
        }
    }

    // 优化：缓存storage变量，在memory中计算
    function processGood(uint[] calldata inputs) external {
        uint mult = multiplier;  // 只读取一次storage
        uint len = inputs.length;
        uint[] memory temp = new uint[](len);

        // 在memory中计算
        for(uint i = 0; i < len; i++) {
            temp[i] = inputs[i] * mult;  // 只读memory，不读storage
        }

        // 批量写入
        for(uint i = 0; i < len; i++) {
            results.push(temp[i]);
        }
    }
```

```
    // 优势：减少了storage读取次数
}
```

优化原理和适用场景：

1. **Storage写入成本高**：
   - 每次storage写入需要约20,000 gas
   - 在循环中写入会累积大量gas消耗
2. **Memory操作成本低**：
   - Memory读写只需约3-10 gas
   - 在memory中完成计算后再写入storage更高效
3. **优化有效的场景**：
   - ✅ **全量替换数组**：需要更新所有或大部分元素
   - ✅ **复杂计算**：需要读取多个storage变量进行计算
   - ✅ **缓存storage变量**：减少重复读取storage
   - ✅ **小数组**（<5个元素）：复制成本低
4. **优化无效的场景**：
   - ❌ **部分更新**：只更新少量元素（如3/10），memory优化反而更差
   - ❌ **向数组追加元素**：push操作已优化，无需此技巧
   - ❌ **大数组部分更新**：复制整个数组的成本 > 直接更新的成本

关键原则：

1. **部分更新**：直接在循环中写storage
2. **全量替换**：考虑使用memory优化
3. **复杂计算**：缓存storage变量，在memory中计算
4. **实际测试**：优化效果因场景而异，需要实际测试验证

总结：这个优化技巧不是万能的，需要根据具体场景判断。对于部分更新，直接写storage通常已经是最优解。

8.6 Gas优化效果对比

```
contract OptimizationComparison {
    uint[] public data;

    // 初始化测试数据
    function initialize(uint count) public {
        delete data;
        for(uint i = 0; i < count; i++) {
            data.push(i);
        }
    }

    // 级别1：完全未优化
    function level1_NoOptimization() public view returns (uint) {
        uint total = 0;
        for(uint i = 0; i < data.length; i++) {  // 每次读length
            total += data[i];
        }
```

```
        return total;
    }
    // 100个元素：约25,000 gas

    // 级别2：缓存length
    function level2_CacheLength() public view returns (uint) {
        uint total = 0;
        uint len = data.length;  // 缓存
        for(uint i = 0; i < len; i++) {
            total += data[i];
        }
        return total;
    }
    // 100个元素：约23,000 gas（节省8%）

    // 级别3：缓存length + unchecked
    function level3_CacheAndUnchecked() public view returns (uint) {
        uint total = 0;
        uint len = data.length;
        for(uint i = 0; i < len; ) {
            total += data[i];
            unchecked { i++; }
        }
        return total;
    }
    // 100个元素：约21,000 gas（节省16%）
}
```

优化效果总结：

优化级别	Gas消耗	节省比例
未优化	25,000	-
缓存length	23,000	8%
+unchecked	21,000	16%

---

9. 数组vs映射

9.1 何时使用数组

数组的优势：

1. 可以遍历所有元素
2. 保持元素顺序
3. 可以获取所有数据
4. 支持索引访问

适合使用数组的场景：

```
contract ArrayUseCases {
    // 场景1：需要遍历的小集合
    address[] public members;  // 会员列表（<100人）

    // 场景2：需要保持顺序
    uint[] public priceHistory;  // 价格历史记录

    // 场景3：需要返回所有数据
    string[] public announcements;  // 公告列表

    // 场景4：固定大小的集合
    uint[7] public weeklyData;  // 一周的数据
}
```

9.2 何时使用映射

映射的优势：

1. 恒定时间（O(1)）查找
2. 不受数量限制
3. Gas消耗稳定
4. 适合大数据集

适合使用映射的场景：

```
contract MappingUseCases {
    // 场景1：大量数据的查找
    mapping(address => uint) public balances;  // 用户余额

    // 场景2：不需要遍历
    mapping(bytes32 => bool) public usedNonces;  // 已使用的nonce

    // 场景3：数据量不确定
    mapping(address => bool) public isWhitelisted;  // 白名单

    // 场景4：只需键值查询
    mapping(uint => address) public tokenOwners;  // NFT所有者
}
```

9.3 数组与映射的对比

特性	数组	映射
遍历	可以	不可以
顺序	保持	无序
查找速度	O(n)	O(1)
获取所有数据	可以	不可以
大小限制	有（gas限制）	无
Gas成本（查找）	随大小增加	恒定
Gas成本（遍历）	线性增长	不支持
默认值	无	有
删除元素	复杂	简单

9.4 组合使用：最佳实践

最强大的模式是**数组+映射组合**：

```
contract ArrayPlusMappingPattern {
    // 组合模式：数组+映射
    address[] public userList;  // 可遍历
    mapping(address => bool) public isUser;  // 快速查找
    mapping(address => uint) public userIndex;  // 快速定位

    uint public constant MAX_USERS = 1000;

    // 添加用户
    function addUser(address user) public {
        require(!isUser[user], "User already exists");
        require(userList.length < MAX_USERS, "User list is full");

        userList.push(user);
        isUser[user] = true;
        userIndex[user] = userList.length - 1;
    }

    // 快速检查（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;
    }
```

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

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

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

        userList.pop();
        delete isUser[user];
        delete userIndex[user];
    }

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

组合模式的优势：

1. **O(1)查找**：通过mapping快速检查存在性
2. **可遍历**：通过数组遍历所有元素
3. **快速删除**：通过userIndex快速定位并删除
4. **数据一致性**：三个数据结构保持同步

---

10. 实践练习

练习1：安全数组管理合约

任务要求：

创建一个完整的数组管理合约，实现以下功能：

1. 限制最大长度为100
2. 实现安全的添加功能（safePush）
3. 实现两种删除方法（保序和快速）
4. 实现分批求和功能（sumRange）
5. 实现查找功能（返回元素索引）
6. 实现获取所有元素功能

参考代码框架：

```
// SPDX-License-Identifier: MIT
```

```solidity
pragma solidity ^0.8.0;

contract SafeArrayManager {
    uint[] public data;
    uint public constant MAX_SIZE = 100;

    // TODO: 实现以下功能

    // 1. 安全添加
    function safePush(uint value) public {
        // 检查大小限制
        // 添加元素
    }

    // 2. 保序删除
    function removeOrdered(uint index) public {
        // 检查索引
        // 移动元素
        // pop最后元素
    }

    // 3. 快速删除
    function removeUnordered(uint index) public {
        // 检查索引
        // 替换为最后元素
        // pop
    }

    // 4. 分批求和
    function sumRange(uint start, uint end) public view returns (uint) {
        // 检查范围
        // 计算总和
    }

    // 5. 查找元素
    function findElement(uint value) public view returns (bool, uint) {
        // 遍历查找
        // 返回是否找到和索引
    }

    // 6. 获取所有元素
    function getAll() public view returns (uint[] memory) {
        // 返回整个数组
    }
}
```

完整参考答案:

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

```
contract SafeArrayManager {
    uint[] public data;
    uint public constant MAX_SIZE = 100;

    event ElementAdded(uint value, uint index);
    event ElementRemoved(uint index, uint value);

    // 1. 安全添加
    function safePush(uint value) public {
        require(data.length < MAX_SIZE, "Array is full");
        data.push(value);
        emit ElementAdded(value, data.length - 1);
    }

    // 2. 保序删除
    function removeOrdered(uint index) public {
        require(index < data.length, "Index out of bounds");

        uint removedValue = data[index];

        for(uint i = index; i < data.length - 1; i++) {
            data[i] = data[i + 1];
        }
        data.pop();

        emit ElementRemoved(index, removedValue);
    }

    // 3. 快速删除
    function removeUnordered(uint index) public {
        require(index < data.length, "Index out of bounds");

        uint removedValue = data[index];

        data[index] = data[data.length - 1];
        data.pop();

        emit ElementRemoved(index, removedValue);
    }

    // 4. 分批求和
    function sumRange(uint start, uint end) public view returns (uint) {
        require(start < end, "Invalid range");
        require(end <= data.length, "End out of bounds");

        uint total = 0;
        for(uint i = start; i < end; i++) {
            total += data[i];
        }
        return total;
    }
```

```
    // 5. 查找元素
    function findElement(uint value) public view returns (bool found, uint index) {
        uint len = data.length;
        for(uint i = 0; i < len; i++) {
            if(data[i] == value) {
                return (true, i);
            }
        }
        return (false, 0);
    }

    // 6. 获取所有元素
    function getAll() public view returns (uint[] memory) {
        return data;
    }

    // 辅助功能
    function getLength() public view returns (uint) {
        return data.length;
    }

    function isEmpty() public view returns (bool) {
        return data.length == 0;
    }

    function isFull() public view returns (bool) {
        return data.length >= MAX_SIZE;
    }
}
```

练习2：Gas优化挑战

任务：优化以下函数，至少节省15% Gas。

原始代码（未优化）：

```
contract UnoptimizedCode {
    uint[] public data;

    function process(uint[] memory values) public {
        for(uint i = 0; i < values.length; i++) {
            if(values[i] > 10) {
                data.push(values[i]);
            }
        }
    }
}
```

优化提示：

- 使用calldata替代memory

- 缓存数组长度
- 考虑减少storage写入

参考答案：

```
contract OptimizedCode {
    uint[] public data;

    function process(uint[] calldata values) external {
        uint len = values.length;

        // 使用临时 memory 数组收集符合条件的值
        uint[] memory temp = new uint[](len);
        uint count = 0;

        // 一次遍历完成收集
        for(uint i = 0; i < len; i++) {
            if(values[i] > 10) {
                temp[count] = values[i];
                count++;
            }
        }

        // 批量 push（连续操作更省 gas）
        for(uint i = 0; i < count; i++) {
            data.push(temp[i]);
        }
    }
}
```

进一步优化：

```
contract OptimizedCode {
    uint[] public data;

    function process(uint[] calldata values) external {
        uint len = values.length;
        uint currentLen = data.length;
        uint count = 0;

        // 第一次遍历：计算符合条件的数量
        for(uint i = 0; i < len; i++) {
            if(values[i] > 10) {
                count++;
            }
        }

        // 预先扩展数组（一次性写入长度）
        if(count > 0) {
            uint newLen = currentLen + count;
            assembly {
```

```
                // 直接扩展数组长度，避免多次 push
                sstore(add(data.slot, 0), newLen)
            }

            // 第二次遍历：直接赋值到已分配的位置
            uint index = currentLen;
            for(uint i = 0; i < len; i++) {
                if(values[i] > 10) {
                    data[index] = values[i];  // 直接赋值，比 push 省 gas
                    index++;
                }
            }
        }
    }
}
```

练习3：实战项目 - 简单待办事项列表

需求分析：

创建一个去中心化的待办事项管理合约：

- 每个用户有自己的待办列表
- 可以添加、完成、删除待办
- 可以查看所有待办和已完成的待办
- 限制每个用户最多100个待办事项

完整实现：

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

contract TodoList {
    struct Todo {
        string task;
        bool completed;
        uint256 timestamp;
    }

    // 每个用户的待办列表
    mapping(address => Todo[]) private userTodos;
    uint public constant MAX_TODOS = 100;

    event TodoAdded(address indexed user, uint index, string task);
    event TodoCompleted(address indexed user, uint index);
    event TodoDeleted(address indexed user, uint index);

    // 添加待办
    function addTodo(string memory task) public {
        require(bytes(task).length > 0, "Task cannot be empty");
        require(bytes(task).length <= 200, "Task too long");
        require(userTodos[msg.sender].length < MAX_TODOS, "Todo list is full");
```

```
        userTodos[msg.sender].push(Todo({
            task: task,
            completed: false,
            timestamp: block.timestamp
        }));

        emit TodoAdded(msg.sender, userTodos[msg.sender].length - 1, task);
    }

    // 标记为完成
    function completeTodo(uint index) public {
        require(index < userTodos[msg.sender].length, "Index out of bounds");
        require(!userTodos[msg.sender][index].completed, "Already completed");

        userTodos[msg.sender][index].completed = true;
        emit TodoCompleted(msg.sender, index);
    }

    // 删除待办（快速删除，不保序）
    function deleteTodo(uint index) public {
        require(index < userTodos[msg.sender].length, "Index out of bounds");

        uint lastIndex = userTodos[msg.sender].length - 1;

        if(index != lastIndex) {
            userTodos[msg.sender][index] = userTodos[msg.sender][lastIndex];
        }

        userTodos[msg.sender].pop();
        emit TodoDeleted(msg.sender, index);
    }

    // 获取所有待办
    function getAllTodos() public view returns (Todo[] memory) {
        return userTodos[msg.sender];
    }

    // 获取待办数量
    function getTodoCount() public view returns (uint) {
        return userTodos[msg.sender].length;
    }

    // 获取未完成的待办
    function getPendingTodos() public view returns (Todo[] memory) {
        Todo[] memory allTodos = userTodos[msg.sender];
        uint pendingCount = 0;

        // 计算未完成数量
        for(uint i = 0; i < allTodos.length; i++) {
            if(!allTodos[i].completed) {
                pendingCount++;
```

```
            }
        }

        // 创建结果数组
        Todo[] memory pending = new Todo[](pendingCount);
        uint index = 0;

        // 填充结果
        for(uint i = 0; i < allTodos.length; i++) {
            if(!allTodos[i].completed) {
                pending[index] = allTodos[i];
                index++;
            }
        }

        return pending;
    }

    // 获取已完成的待办
    function getCompletedTodos() public view returns (Todo[] memory) {
        Todo[] memory allTodos = userTodos[msg.sender];
        uint completedCount = 0;

        for(uint i = 0; i < allTodos.length; i++) {
            if(allTodos[i].completed) {
                completedCount++;
            }
        }

        Todo[] memory completed = new Todo[](completedCount);
        uint index = 0;

        for(uint i = 0; i < allTodos.length; i++) {
            if(allTodos[i].completed) {
                completed[index] = allTodos[i];
                index++;
            }
        }

        return completed;
    }
}
```

11. 常见问题解答

Q1：为什么delete arr[i]不改变数组长度?

答：`delete`操作只是将元素重置为默认值（数字类型为0），而不是真正删除。这是Solidity的设计决定，目的是：

1. **保持索引一致性**：其他元素的索引不会改变
2. **避免昂贵的操作**：不需要移动后续所有元素
3. **明确的语义**：delete只是重置，不是删除

如果要真正删除元素，需要使用我们讲解的两种删除方法。

Q2：uint[3][4]到底表示什么?

答：这是Solidity中最容易混淆的地方。`uint[3][4]`表示：

- 4个数组
- 每个数组长度为3

从右向左读更容易理解：`[4]`表示4个，`[3]`表示长度为3的数组。

如果你想要3行4列的矩阵，应该声明为`uint[4][3]`。

Q3：为什么要限制数组最大长度?

答：限制数组大小是关键的安全措施：

1. **防止Gas耗尽**：大数组遍历可能超过区块gas限制
2. **保证可用性**：确保所有函数都能正常执行
3. **防止DoS攻击**：恶意用户不能通过填满数组使合约失效
4. **用户体验**：用户可以预估交易成本

不限制数组大小是智能合约开发的常见错误，可能导致资金被永久锁定。

Q4：memory数组为什么不能push?

答：Memory数组在创建时必须指定固定大小，因为：

1. **内存分配**：Memory是栈式分配，创建时就分配固定空间
2. **性能考虑**：动态扩展memory数组会很低效
3. **设计简化**：简化EVM实现

只有storage中的动态数组才支持push/pop操作。

Q5：什么时候用数组，什么时候用mapping?

答：简单判断规则：

使用数组：

- 需要遍历所有元素
- 需要保持顺序
- 元素数量少（<100）
- 需要返回所有数据

使用mapping：

- 只需按键查找
- 数据量大或不确定
- 不需要遍历

- 查找频繁

最佳实践：组合使用（数组+mapping）

Q6：如何安全地遍历大数组?

答：几种安全策略：

1. **限制数组大小**：设置MAX_SIZE常量
2. **分批处理**：使用start和end参数
3. **缓存length**：避免重复读取
4. **避免在遍历中写storage**：先在memory中处理
5. **考虑用mapping替代**：如果不需要遍历

Q7：数组的gas成本如何计算?

答：主要成本来源：

- **push操作**：约20,000-40,000 gas（首次写入更贵）
- **pop操作**：约5,000 gas
- **读取元素**：约200-2,100 gas（storage读取）
- **遍历**：约5,000 gas × 元素数量

优化技巧：

- 缓存length：节省约200 gas/次
- 使用calldata：节省20-30%
- 批量操作：减少交易数

---

12. 知识点总结

数组类型

定长数组：

- 语法：`uint[5]`
- 长度固定
- 不能push/pop
- Gas成本较低

动态数组：

- 语法：`uint[]`
- 长度可变
- 支持push/pop
- Gas成本较高

关键操作

基本操作：

- `push(value)`：添加元素
- `pop()`：删除最后元素
- `length`：获取长度
- `delete arr[i]`：重置为0，不改变length

删除元素：

- 保序删除：移动元素，保持顺序，Gas高
- 快速删除：替换+pop，不保序，Gas低

Storage vs Memory

Storage数组：

- 永久存储
- 支持push/pop
- Gas成本高

Memory数组：

- 临时存储
- 必须定长
- 不支持push/pop
- Gas成本低

Gas优化

1. **缓存length**：避免重复读取storage
2. **限制大小**：防止gas耗尽
3. **分批处理**：可控的gas消耗
4. **使用calldata**：外部函数参数优化
5. **避免循环写storage**：先在memory处理

安全实践

1. **始终检查索引**：防止越界
2. **限制数组大小**：设置MAX_SIZE
3. **小心遍历**：大数组可能gas耗尽
4. **组合使用mapping**：快速查找+可遍历
5. **分批处理大数组**：避免单次消耗过多gas

---

13. 学习检查清单

完成本课后，你应该能够：

数组基础：

- [ ] 理解定长数组和动态数组的区别
- [ ] 会声明和初始化不同类型的数组
- [ ] 理解storage和memory数组的差异

- [ ] 掌握数组的基本操作（push/pop/length）

数组操作：

- [ ] 会访问和修改数组元素
- [ ] 理解delete操作的陷阱
- [ ] 掌握两种删除元素的方法
- [ ] 会遍历数组并进行计算

多维数组：

- [ ] 理解多维数组的声明顺序
- [ ] 会操作二维数组
- [ ] 知道何时避免使用高维数组

Gas优化：

- [ ] 会缓存数组长度
- [ ] 会限制数组最大长度
- [ ] 会使用calldata优化
- [ ] 会分批处理大数组
- [ ] 理解为什么要避免循环写storage

安全实践：

- [ ] 会检查数组越界
- [ ] 理解大数组遍历的风险
- [ ] 会选择数组还是mapping
- [ ] 会组合使用数组和mapping

---

14. 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. **对比Gas消耗**：亲自测试优化效果
4. **准备学习第3.2课**：映射和结构体

下节课预告：第3.2课 - 映射和结构体

我们将学习：

- mapping的使用和限制
- struct的定义和操作
- mapping和struct的组合使用
- 复杂数据结构的设计模式

---

15. 扩展资源

官方文档：

- Solidity数组文档：https://docs.soliditylang.org/en/latest/types.html#arrays
- Solidity内存布局：https://docs.soliditylang.org/en/latest/internals/layout_in_memory.html

学习资源：

- Solidity by Example - Arrays：https://solidity-by-example.org/array/
- OpenZeppelin Contracts：研究专业合约中的数组使用

Gas优化：

- Gas Optimization Tips：研究专业的gas优化技巧
- Ethereum Gas Tracker：了解当前gas价格

安全资源：

- Smart Contract Weakness Classification：了解数组相关的安全问题
- Consensys Best Practices：学习数组使用的最佳实践

实战项目：
研究开源项目中的数组使用：

- Uniswap：流动性池管理
- Compound：市场列表管理
- ERC721：NFT索引管理

课程结束

恭喜你完成第3.1课！数组是智能合约开发中的重要数据结构，正确使用数组并进行Gas优化是成为优秀Solidity开发者的必备技能。

记住关键点：

- 限制数组大小至关重要
- 缓存length节省gas
- 大数组遍历很危险
- 组合使用数组和mapping最强大

继续练习，下节课见！