Solidity智能合约开发知识

第9.1课：Gas优化技巧

学习目标：理解Gas成本的构成和计算方式、掌握6种核心的Gas优化技巧、学会使用专业工具来测量和分析Gas消耗、能够在实际项目中灵活应用优化技巧，将合约的Gas成本降低30%到80%

预计学习时间：2-3小时

难度等级：中级

重要提示：Gas优化是智能合约开发中非常重要的技能，直接影响用户体验和合约的竞争力。通过合理的优化，可以将Gas成本降低30%到80%，显著提升合约的经济性。

目录

---

1. Gas成本分析

1.1 Gas费用基础

在智能合约开发中，Gas是执行操作的计算单位。每笔交易都需要消耗Gas，而Gas成本直接影响用户的使用体验和合约的经济性。

Gas费用计算公式：

```
交易费用 = Gas使用量 × Gas价格
```

实际案例：

假设一笔交易使用了21,000个Gas，当时的Gas价格是50 gwei，那么：

- 总费用 = 21,000 × 50 gwei = 1,050,000 gwei = 0.00105 ETH
- 如果ETH价格是3000美元，这笔交易的成本 = 0.00105 × 3000 = 3.15美元

Gas的本质：

我们可以把Gas理解为汽车的汽油：

- 汽车需要汽油才能运行，智能合约需要Gas才能执行
- Gas是计量单位，用户设置愿意支付的Gas价格
- 操作的复杂度决定了Gas的用量

优化目标：

我们的优化目标是降低Gas的用量。Gas价格我们无法控制（由市场决定），但Gas用量是可以通过代码优化来降低的。

1.2 EVM操作成本对比

要优化Gas，我们首先要了解不同操作的成本差异。以下是EVM中常见操作的Gas成本：

基础运算（最便宜）：

操作	Gas成本	说明
ADD（加法）	3	最基础的运算
MUL（乘法）	5	乘法运算
SUB（减法）	3	减法运算
DIV（除法）	5	除法运算

存储操作（最昂贵）：

操作	Gas成本	说明
SSTORE（写入新值）	20,000	写入新的存储槽
SSTORE（修改已有值）	5,000	修改已存在的存储槽
SLOAD（读取存储）	2,100	读取存储槽

其他操作：

操作	Gas成本	说明
CALL（外部调用）	2,600+	调用外部合约
LOG（事件）	375+	发出事件日志
CREATE（创建合约）	32,000+	部署新合约

成本对比：

一次SSTORE（20,000 Gas）的成本相当于：

- 6,666次ADD运算（3 Gas × 6,666 = 20,000 Gas）
- 9.5次SLOAD操作（2,100 Gas × 9.5 ≈ 20,000 Gas）

关键结论：

存储操作是最昂贵的操作，所以优化的重点就是要减少存储操作！

1.3 实际合约Gas消耗分析

让我们通过一个实际的例子来理解Gas消耗的构成：

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

// 简单的转账函数示例
contract TransferExample {
    mapping(address => uint256) public balances;

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

    /**
     * @notice 转账函数
     * @param to 接收者地址
     * @param amount 转账金额
     * @dev 分析这个函数的Gas消耗
     */
    function transfer(address to, uint256 amount) external {
        // 步骤1：读取发送者余额（SLOAD）
        uint256 senderBalance = balances[msg.sender];
        require(senderBalance >= amount, "Insufficient balance");

        // 步骤2：读取接收者余额（SLOAD）
        uint256 receiverBalance = balances[to];

        // 步骤3：更新发送者余额（SSTORE）
        balances[msg.sender] = senderBalance - amount;

        // 步骤4：更新接收者余额（SSTORE）
        balances[to] = receiverBalance + amount;

        // 步骤5：发出事件（LOG）
        emit Transfer(msg.sender, to, amount);
    }
}
```

Gas消耗分析：

1. **两次SLOAD**：读取发送者和接收者的余额
    - 成本：2,100 × 2 = 4,200 Gas
2. **两次SSTORE**：更新两个账户的余额（假设都是修改已有值）
    - 成本：5,000 × 2 = 10,000 Gas
3. **事件LOG**：发出Transfer事件
    - 成本：约1,000 Gas
4. **其他操作**：参数解析、函数调用、require检查等

- 成本：约30,000 Gas

总计：约45,000 Gas

实际成本：

如果Gas价格是100 gwei，ETH价格是3000美元：

- 交易费用 = 45,000 × 100 gwei = 0.0045 ETH
- 美元成本 = 0.0045 × 3000 = 13.5美元

即使是很简单的操作，Gas成本也是不容忽视的。通过优化，我们可以将这个成本降低30%到80%。

---

2. 存储优化

存储优化是Gas优化中最重要、最有效的方法。因为存储操作（SSTORE和SLOAD）是最昂贵的操作，减少存储操作可以带来显著的Gas节省。

2.1 技巧1：存储槽打包

EVM的存储是以32字节（256位）为一个槽来组织的。如果我们不合理安排变量顺序，就会浪费存储槽。

未优化的示例：

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

// 未优化版本：浪费存储槽
contract StoragePackingBad {
    uint8 public a = 1;     // 占用8位，但占用整个槽（256位）
    uint256 public b = 2;   // 占用256位，占用一个槽
    uint8 public c = 3;     // 占用8位，但占用整个槽（256位）
    uint256 public d = 4;   // 占用256位，占用一个槽

    // 总共使用4个存储槽
    // 初始化需要4次SSTORE = 80,000 Gas
}
```

问题分析：

- `uint8`只需要8位，但EVM的存储槽是256位
- 如果`uint8`变量之间被`uint256`隔开，它们无法共享存储槽
- 每个`uint8`都会占用一个完整的存储槽，造成巨大浪费

优化后的示例：

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

// 优化版本：合理打包变量
contract StoragePackingGood {
    uint8 public a = 1;     // 占用8位
```

```
    uint8 public c = 3;     // 占用8位，与a共享一个槽
    uint256 public b = 2;   // 占用256位，占用一个槽
    uint256 public d = 4;   // 占用256位，占用一个槽

    // 总共使用3个存储槽（a和c共享一个槽）
    // 初始化需要3次SSTORE = 60,000 Gas
    // 节省了20,000 Gas，节省比例25%
}
```

优化效果：

- 未优化版本：4个存储槽，80,000 Gas
- 优化版本：3个存储槽，60,000 Gas
- 节省：20,000 Gas（25%）

关键原则：

把小类型的变量（uint8、uint16、uint32、bool等）放在一起，让它们共享存储槽。

打包规则：

- 一个存储槽可以存储：
  - 1个uint256
  - 2个uint128
  - 4个uint64
  - 8个uint32
  - 16个uint16
  - 32个uint8
  - 32个bool

注意事项：

- 变量必须连续声明才能打包
- 如果中间有uint256等大类型，会打断打包
- 结构体内部的变量可以自动打包

2.2 技巧2：位域打包

位域打包是一种更极致的优化方式，核心思想是在一个变量中存储多个值，使用位运算来访问和修改。

实际案例：Aave协议

Aave是一个知名的DeFi借贷协议，他们在存储优化方面做了非常极致的优化。他们的`ReserveConfiguration`结构体需要存储20多个配置参数，包括：

- LTV（贷款价值比）
- 清算阈值
- 清算奖励
- 小数位
- 是否激活
- 等等

传统方式（未优化）：

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

// 传统方式：每个参数一个变量
contract ReserveConfigBad {
    uint256 public ltv;                // 贷款价值比
    uint256 public liquidationThreshold;  // 清算阈值
    uint256 public liquidationBonus;       // 清算奖励
    uint8 public decimals;             // 小数位
    bool public isActive;              // 是否激活

    // 至少需要5个存储槽
    // 初始化需要5次SSTORE = 100,000 Gas
}
```

Aave的优化方式：

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

// Aave的优化方式：所有参数压缩到一个uint256
contract ReserveConfigGood {
    // 将所有配置参数压缩到一个uint256中
    // 使用不同的位段来存储不同的参数
    uint256 public data;

    // 位段定义
    uint256 private constant LTV_MASK = 0xFFFF;                    // 第0-15位：LTV
    uint256 private constant LIQUIDATION_THRESHOLD_MASK = 0xFFFF0000;  // 第16-31位：清算阈
值
    uint256 private constant LIQUIDATION_BONUS_MASK = 0xFFFF00000000;  // 第32-47位：清算奖
励
    uint256 private constant DECIMALS_MASK = 0xFF000000000000;     // 第48-55位：小数位
    uint256 private constant IS_ACTIVE_MASK = 0x100000000000000;   // 第56位：是否激活

    /**
     * @notice 设置LTV
     * @param ltv 贷款价值比 (0-65535)
     */
    function setLTV(uint256 ltv) external {
        // 清除旧的LTV值
        data = data & ~LTV_MASK;
        // 设置新的LTV值
        data = data | (ltv & LTV_MASK);
    }

    /**
     * @notice 获取LTV
     */
    function getLTV() external view returns (uint256) {
        return data & LTV_MASK;
```

```
    }

    /**
     * @notice 设置清算阈值
     * @param threshold 清算阈值 (0-65535)
     */
    function setLiquidationThreshold(uint256 threshold) external {
        data = data & ~LIQUIDATION_THRESHOLD_MASK;
        data = data | ((threshold << 16) & LIQUIDATION_THRESHOLD_MASK);
    }

    /**
     * @notice 获取清算阈值
     */
    function getLiquidationThreshold() external view returns (uint256) {
        return (data & LIQUIDATION_THRESHOLD_MASK) >> 16;
    }

    // 其他参数的设置和获取函数类似...

    // 只需要1个存储槽
    // 初始化只需要1次SSTORE = 20,000 Gas
    // 相比传统方式，节省了80,000 Gas，节省比例80%!
}
```

优化效果：

- 传统方式：5个存储槽，100,000 Gas
- 位域打包：1个存储槽，20,000 Gas
- 节省：80,000 Gas（80%）

优缺点分析：

优点：

- 大幅节省Gas成本
- 适合存储大量配置参数

缺点：

- 代码复杂度增加
- 需要使用位运算
- 可读性降低

适用场景：

- DeFi协议中的配置参数
- 需要存储大量小数值的场景
- Gas成本敏感的应用

2.3 技巧3：局部存储指针

局部存储指针可以避免重复读取存储，显著减少SLOAD操作。

未优化的示例：

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

contract LocalStoragePointerBad {
    struct Reserve {
        uint256 rate;       // 利率
        uint256 index;      // 索引
        uint256 timestamp;  // 时间戳
    }

    mapping(address => Reserve) public reserves;

    /**
     * @notice 更新储备信息（未优化版本）
     * @param asset 资产地址
     * @param newRate 新利率
     * @param newIndex 新索引
     * @dev 每次访问reserves[asset]都会触发SLOAD
     */
    function updateReserve(
        address asset,
        uint256 newRate,
        uint256 newIndex
    ) external {
        // 第一次SLOAD：读取reserves[asset]
        reserves[asset].rate = newRate;

        // 第二次SLOAD：再次读取reserves[asset]
        reserves[asset].index = newIndex;

        // 第三次SLOAD：再次读取reserves[asset]
        reserves[asset].timestamp = block.timestamp;

        // 总共执行3次SLOAD = 6,300 Gas
    }
}
```

问题分析：

每次访问`reserves[asset]`都会触发一次SLOAD操作。如果我们需要更新同一个结构体的多个字段，就会重复读取存储，浪费Gas。

优化后的示例：

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

contract LocalStoragePointerGood {
    struct Reserve {
```

```
        uint256 rate;
        uint256 index;
        uint256 timestamp;
    }

    mapping(address => Reserve) public reserves;

    /**
     * @notice 更新储备信息（优化版本）
     * @param asset 资产地址
     * @param newRate 新利率
     * @param newIndex 新索引
     * @dev 使用局部存储指针，只读取一次存储
     */
    function updateReserve(
        address asset,
        uint256 newRate,
        uint256 newIndex
    ) external {
        // 创建storage类型的局部变量
        // 这只需要一次SLOAD
        Reserve storage r = reserves[asset];

        // 通过局部变量访问，直接SSTORE，不需要重复SLOAD
        r.rate = newRate;
        r.index = newIndex;
        r.timestamp = block.timestamp;

        // 总共只执行1次SLOAD = 2,100 Gas
        // 节省了4,200 Gas
    }
}
```

优化效果：

- 未优化版本：3次SLOAD，6,300 Gas
- 优化版本：1次SLOAD，2,100 Gas
- 节省：4,200 Gas（66.7%）

关键要点：

1. **使用storage类型**：局部变量必须是`storage`类型，而不是`memory`类型
   - `storage`：指向存储的引用，不复制数据
   - `memory`：复制数据到内存，不会节省Gas
2. **适用场景**：
   - 需要频繁访问同一个存储变量
   - 需要更新结构体的多个字段
   - 需要更新mapping中的多个值
3. **语法示例**：

```
// 正确：使用storage
Reserve storage r = reserves[asset];

// 错误：使用memory（不会节省Gas）
Reserve memory r = reserves[asset];
```

---

3. 数据类型优化

数据类型的选择看起来是个小问题，但选错了类型会带来额外的Gas消耗。我们需要根据不同的场景选择合适的类型。

3.1 类型选择原则

存储变量：

- 如果需要打包：使用uint8、uint16、uint32等小类型，可以节省空间
- 如果不需要打包：直接使用uint256，因为EVM原生支持uint256，不需要类型转换

函数参数和内存变量：

- 应该使用uint256，因为EVM的栈和内存都是按256位来操作的
- 使用uint256不需要转换，反而用小类型需要额外的转换操作

循环计数器：

- 应该使用uint256，避免类型转换的开销

推荐做法：

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

contract TypeOptimizationGood {
    // 存储变量：如果需要打包，使用小类型
    uint8 public smallValue;  // 可以与其他小类型打包

    // 存储变量：如果不需要打包，使用uint256
    uint256 public largeValue;

    /**
     * @notice 函数参数使用uint256（推荐）
     * @param amount 金额
     * @dev 参数用uint256，内部计算也用uint256，不需要转换
     */
    function processAmount(uint256 amount) external {
        // 内部计算也用uint256，不需要任何转换
        uint256 result = amount * 2;
        largeValue = result;
    }
}
```

不推荐做法：

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

contract TypeOptimizationBad {
    uint256 public value;

    /**
     * @notice 函数参数使用uint8（不推荐）
     * @param amount 金额
     * @dev 参数用uint8，在函数内部计算时需要转换为uint256
     */
    function processAmount(uint8 amount) external {
        // 需要将uint8转换为uint256，有转换成本
        uint256 result = uint256(amount) * 2;
        value = result;
    }
}
```

关键原则：

除非是为了存储打包，否则都用uint256。

3.2 Mapping vs Array

在数据结构的选择上，我们经常会纠结：到底是用Mapping还是Array?

Mapping的特点：

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

contract MappingExample {
    // Mapping：访问成本固定，只需要一次SLOAD
    mapping(address => uint256) public balances;

    /**
     * @notice 使用Mapping存储余额
     * @dev 访问成本固定，适合随机访问
     */
    function getBalance(address user) external view returns (uint256) {
        // 只需要一次SLOAD，成本固定
        return balances[user];
    }
}
```

优点：

- 访问成本固定（一次SLOAD）
- 适合随机访问

- 适合存储用户余额、配置参数等需要快速查找的数据

缺点：

- 不能遍历
- 不能获取长度

Array的特点：

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

contract ArrayExample {
    // Array：可以遍历，可以获取长度
    address[] public users;

    /**
     * @notice 遍历数组（危险！）
     * @dev 遍历成本线性增长，数组越大成本越高
     */
    function getAllUsers() external view returns (address[] memory) {
        // 如果数组很大，Gas消耗会非常高
        return users;
    }
}
```

优点：

- 可以遍历
- 可以获取长度
- 适合存储需要遍历的列表

缺点：

- 访问时需要计算索引
- 遍历的成本很高，线性增长

重要警告：

永远不要在链上遍历大数组！

原因：

遍历数组的Gas成本是线性增长的，数组越大，成本越高。如果数组太大，可能会超出区块的Gas限制（约15,000,000 Gas），导致交易失败。

错误示例：

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

contract BadArrayUsage {
    address[] public users;
```

```
    /**
     * @notice 危险：遍历大数组
     * @dev 如果users数组很大，这个函数会消耗大量Gas，甚至可能失败
     */
    function processAllUsers() external {
        // 危险：如果数组有1000个元素，需要循环1000次
        // 每次循环都有SLOAD和SSTORE，成本非常高
        for (uint256 i = 0; i < users.length; i++) {
            // 处理每个用户...
        }
    }
}
```

正确做法：

链下遍历，链上只访问特定元素。

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

contract GoodArrayUsage {
    address[] public users;
    mapping(address => bool) public isUser;

    /**
     * @notice 添加用户
     * @param user 用户地址
     */
    function addUser(address user) external {
        require(!isUser[user], "User already exists");
        users.push(user);
        isUser[user] = true;
    }

    /**
     * @notice 处理特定用户（推荐）
     * @param index 用户索引
     * @dev 只访问特定元素，成本固定
     */
    function processUser(uint256 index) external {
        require(index < users.length, "Index out of range");
        address user = users[index];
        // 处理这个用户...
    }

    /**
     * @notice 获取用户数量
     * @dev 前端可以用这个来遍历
     */
    function getUserCount() external view returns (uint256) {
        return users.length;
    }
```

```
}
```

最佳实践：

1. 前端先遍历获取所有数据（通过事件或链下查询）
2. 用户选择要操作的元素
3. 合约只处理那个特定元素

---

4. 函数优化

函数优化包括可见性修饰符的选择、短路求值的应用，以及用事件替代存储等技巧。

4.1 函数可见性优化

不同的可见性修饰符会影响Gas成本。从成本上看，从高到低依次是：public、external、internal、private。

成本对比：

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

contract FunctionVisibility {
    uint256 public value;

    /**
     * @notice public函数（成本最高）
     * @dev 需要同时支持内部调用和外部调用，参数需要复制到memory
     */
    function setValuePublic(uint256 _value) public {
        value = _value;
    }

    /**
     * @notice external函数（推荐用于外部调用）
     * @dev 只支持外部调用，参数直接用calldata，比memory便宜
     */
    function setValueExternal(uint256 _value) external {
        value = _value;
    }

    /**
     * @notice internal函数（用于内部调用）
     * @dev 内部调用，不需要ABI编码，成本较低
     */
    function setValueInternal(uint256 _value) internal {
        value = _value;
    }

    /**
     * @notice private函数（用于私有函数）
```

```
     * @dev 类似internal，但访问权限更严格
     */
    function setValuePrivate(uint256 _value) private {
        value = _value;
    }
}
```

为什么会有差异：

1. **public函数**：
   - 需要同时支持内部调用和外部调用
   - 参数需要复制到memory
   - 开销最大
2. **external函数**：
   - 只支持外部调用
   - 参数直接用calldata
   - calldata比memory便宜，所以external比public便宜
3. **internal函数**：
   - 内部调用，不需要ABI编码
   - 成本较低
4. **private函数**：
   - 类似internal，但访问权限更严格
   - 成本相近

最佳实践：

- 如果函数是给外部调用的，就用external
- 内部辅助函数，用internal或private
- 避免不必要的public

很多人习惯性地把所有函数都写成public，这是不好的习惯。如果函数确定只会被外部调用，就应该用external。

4.2 短路求值优化

短路求值是一个简单但经常被忽视的优化技巧。核心思想是：把便宜的检查放在前面。

未优化的示例：

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

contract ShortCircuitBad {
    /**
     * @notice 未优化：昂贵的检查在前
     * @dev 即使amount是0，expensiveOracleCall也会执行，浪费Gas
     */
    function processPayment(uint256 amount) external {
        // 问题：即使amount是0，这个昂贵的Oracle调用也会执行
        require(expensiveOracleCall(), "Oracle check failed");
        require(amount > 0, "Amount must be greater than 0");
```

```
        // 处理支付...
    }

    /**
     * @notice 模拟昂贵的Oracle调用
     * @dev 这个函数会消耗大量Gas
     */
    function expensiveOracleCall() internal view returns (bool) {
        // 模拟外部调用，消耗约20,000 Gas
        return true;
    }
}
```

问题分析：

如果`amount`是0，第一个require会失败，但`expensiveOracleCall()`已经执行了，浪费了20,000个Gas。

优化后的示例：

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

contract ShortCircuitGood {
    /**
     * @notice 优化：便宜的检查在前
     * @dev 先检查便宜的amount > 0，再检查昂贵的Oracle调用
     */
    function processPayment(uint256 amount) external {
        // 先检查便宜的amount > 0
        require(amount > 0, "Amount must be greater than 0");

        // 如果第一个检查失败，函数立即返回，不会执行第二个检查
        require(expensiveOracleCall(), "Oracle check failed");

        // 处理支付...
    }

    function expensiveOracleCall() internal view returns (bool) {
        return true;
    }
}
```

优化效果：

- 如果`amount`是0，函数立即返回，不会执行昂贵的Oracle调用
- 节省了20,000 Gas

关键原则：

便宜的检查在前，昂贵的检查在后。

实际应用：

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

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

    /**
     * @notice 优化的检查顺序
     * @param to 接收者地址
     * @param amount 转账金额
     */
    function transfer(address to, uint256 amount) external {
        // 1. 便宜的检查：金额大于0
        require(amount > 0, "Amount must be greater than 0");

        // 2. 便宜的检查：地址不为零地址
        require(to != address(0), "Invalid recipient");

        // 3. 便宜的检查：余额足够
        require(balances[msg.sender] >= amount, "Insufficient balance");

        // 4. 昂贵的检查：外部合约调用（如果有）
        // require(externalContract.check(), "External check failed");

        // 执行转账...
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

4.3 事件 vs 存储

这是一个非常重要的优化技巧：用事件替代存储。

成本对比：

方式	Gas成本	说明
Storage存储	60,000+	每条记录需要SSTORE
Event事件	1,500+	每条记录只需要LOG

节省比例：约97%！

使用Storage的场景：

如果数据需要在链上读取，或者影响合约逻辑，那必须用Storage。

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

contract StorageExample {
    // 必须用Storage：余额影响合约逻辑
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 amount) external {
        // 余额必须在链上存储，因为其他函数需要读取和修改
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

使用Event的场景：

如果数据只是用来记录或查询的历史数据，或者是前端需要显示的数据，或者是审计日志，那就应该用Event。

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

contract EventExample {
    // 使用事件记录历史交易
    event TransactionExecuted(
        address indexed from,
        address indexed to,
        uint256 amount,
        uint256 timestamp
    );

    /**
     * @notice 使用事件记录交易（推荐）
     * @param to 接收者
     * @param amount 金额
     * @dev 前端可以通过监听事件来获取所有历史交易
     */
    function executeTransaction(address to, uint256 amount) external {
        // 执行交易逻辑...

        // 使用事件记录，只需要1,500 Gas
        emit TransactionExecuted(msg.sender, to, amount, block.timestamp);
    }
}
```

对比示例：

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

contract EventVsStorage {
    // 方式1：使用数组存储历史（昂贵）
```

```
    struct Transaction {
        address from;
        address to;
        uint256 amount;
        uint256 timestamp;
    }
    Transaction[] public transactions;

    // 方式2：使用事件记录历史（便宜）
    event TransactionExecuted(
        address indexed from,
        address indexed to,
        uint256 amount,
        uint256 timestamp
    );

    /**
     * @notice 使用存储记录（不推荐用于历史数据）
     * @dev 每条记录需要60,000 Gas
     */
    function recordWithStorage(address to, uint256 amount) external {
        transactions.push(Transaction({
            from: msg.sender,
            to: to,
            amount: amount,
            timestamp: block.timestamp
        }));
    }

    /**
     * @notice 使用事件记录（推荐用于历史数据）
     * @dev 每条记录只需要1,500 Gas，节省97%
     */
    function recordWithEvent(address to, uint256 amount) external {
        emit TransactionExecuted(msg.sender, to, amount, block.timestamp);
    }
}
```

关键原则：

- 如果数据需要在链上读取或影响逻辑：用Storage
- 如果数据只是历史记录或前端显示：用Event

前端可以通过监听事件来获取所有历史数据，完全不需要存储在链上。

---

5. 批量操作

批量操作是另一个重要的优化方向。核心思想是减少交易次数，从而减少基础Gas费用。

5.1 批量操作的优势

每笔交易都需要支付基础Gas费用（约21,000 Gas）。如果我们能把多个操作合并到一笔交易中，就可以节省基础费用。

成本对比：

假设我们要给3个地址转账，每次转账的执行费用是30,000 Gas：

方式1：分3次转账：

- 交易1：21,000（基础费用）+ 30,000（执行费用）= 51,000 Gas
- 交易2：21,000（基础费用）+ 30,000（执行费用）= 51,000 Gas
- 交易3：21,000（基础费用）+ 30,000（执行费用）= 51,000 Gas
- **总计**：153,000 Gas

方式2：批量转账：

- 交易1：21,000（基础费用）+ 90,000（执行费用，3次转账）= 111,000 Gas
- **总计**：111,000 Gas

节省：42,000 Gas（27.5%）

转账的数量越多，节省的比例越高。

5.2 批量操作实现

批量转账示例：

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

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

    /**
     * @notice 单个转账
     * @param to 接收者地址
     * @param amount 转账金额
     */
    function transfer(address to, uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }

    /**
     * @notice 批量转账（优化版本）
     * @param recipients 接收者地址数组
     * @param amounts 转账金额数组
     * @dev 一次交易处理多个转账，节省基础Gas费用
     */
    function batchTransfer(
        address[] calldata recipients,
        uint256[] calldata amounts
```

```
    ) external {
        // 检查数组长度是否匹配
        require(recipients.length == amounts.length, "Length mismatch");

        // 设置批量数量上限，防止Gas超出区块限制
        require(recipients.length <= 100, "Too many recipients");

        // 先计算总额（使用unchecked优化循环）
        uint256 totalAmount = 0;
        for (uint256 i = 0; i < amounts.length;) {
            totalAmount += amounts[i];
            unchecked { i++; }
        }

        // 检查余额是否足够（先验证总额，避免执行到一半失败）
        require(balances[msg.sender] >= totalAmount, "Insufficient balance");

        // 执行批量转账
        balances[msg.sender] -= totalAmount;
        for (uint256 i = 0; i < recipients.length;) {
            balances[recipients[i]] += amounts[i];
            unchecked { i++; }
        }
    }
}
```

实现要点：

1. **设置批量数量上限**：
   - 防止Gas超出区块限制
   - 通常设置为100个左右
2. **参数使用calldata**：
   - `calldata`比`memory`便宜
   - 对于数组参数，应该使用`calldata`
3. **先验证总额**：
   - 在执行批量操作前，先计算总额并验证
   - 避免执行到一半失败，浪费Gas
4. **使用unchecked优化循环**：
   - 循环计数器确定不会溢出，可以使用unchecked

其他批量操作场景：

批量操作在实际项目中非常常见：

1. **空投代币**：给多个地址空投代币
2. **批量支付**：给多个地址支付工资或奖励
3. **批量更新配置**：更新多个配置参数
4. **批量审批**：一次性审批多个地址

---

6. unchecked使用

从Solidity 0.8.0开始，所有的算术运算都默认有溢出检查。这大大提高了安全性，但也带来了额外的Gas成本。

6.1 溢出检查的成本

每次算术运算的溢出检查成本：

- 加法：额外20 Gas
- 减法：额外20 Gas
- 乘法：额外30 Gas

如果有100次循环，就额外需要2,000多个Gas。在Gas成本敏感的场景下，这是不能忽视的。

6.2 安全使用unchecked

我们可以在确保安全的情况下，使用`unchecked`跳过溢出检查。

场景1：循环计数器：

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

contract UncheckedLoop {
    uint256[] public data;

    /**
     * @notice 未优化：循环计数器有溢出检查
     * @dev 每次循环的i++都有溢出检查，浪费Gas
     */
    function sumWithCheck() external view returns (uint256) {
        uint256 total = 0;
        uint256 length = data.length;
        for (uint256 i = 0; i < length; i++) {
            total += data[i];
        }
        return total;
    }

    /**
     * @notice 优化：使用unchecked跳过循环计数器的溢出检查
     * @dev i从0开始递增，只要length是合理的值，i就不可能溢出
     */
    function sumWithoutCheck() external view returns (uint256) {
        uint256 total = 0;
        uint256 length = data.length;
        for (uint256 i = 0; i < length;) {
            total += data[i];
            unchecked { i++; }
        }
        return total;
    }
```

```
}
```

场景2：已检查的运算：

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

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

    /**
     * @notice 优化：在已检查的情况下使用unchecked
     * @param amount 转账金额
     * @dev 已经用require确保balance >= amount，减法不可能下溢
     */
    function withdraw(uint256 amount) external {
        uint256 balance = balances[msg.sender];

        // 先检查余额是否足够
        require(balance >= amount, "Insufficient balance");

        // 因为已经检查过，减法不可能下溢，可以使用unchecked
        unchecked {
            balances[msg.sender] = balance - amount;
        }
    }
}
```

6.3 危险的反例

错误示例：

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

contract UncheckedDanger {
    uint256 public balance;

    /**
     * @notice 危险：没有检查就使用unchecked
     * @param amount 增加的金额
     * @dev 这可能导致溢出，非常危险！
     */
    function addBalance(uint256 amount) external {
        // 危险：没有检查就使用unchecked
        // 如果balance + amount超过uint256的最大值，会发生溢出
        unchecked {
            balance += amount;
        }
    }
}
```

关键原则：

只在确保安全的情况下使用unchecked：

1. 循环计数器（确定不会溢出）
2. 已经用require检查过的情况
3. 数学上不可能溢出的情况

不要为了省Gas牺牲安全性！

---

7. 综合优化案例

现在让我们通过一个综合案例来看看这些技巧的实际效果。我们以ERC20代币合约为例，逐步应用各种优化技巧。

7.1 未优化版本（V1）

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

// V1：未优化版本
contract ERC20V1 {
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    uint256 public totalSupply;
    string public name;
    string public symbol;
    uint8 public decimals;

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

    constructor(string memory _name, string memory _symbol) {
        name = _name;
        symbol = _symbol;
        decimals = 18;
    }

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

// Gas消耗：约100,000 Gas
```

7.2 应用存储打包（V2）

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

// V2：应用存储打包
contract ERC20V2 {
    // 将小类型变量放在一起，共享存储槽
    string public name;
    string public symbol;
    uint8 public decimals;  // 可以与bool等小类型打包

    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

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

    constructor(string memory _name, string memory _symbol) {
        name = _name;
        symbol = _symbol;
        decimals = 18;
    }

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

// Gas消耗：约85,000 Gas（节省15%）
```

7.3 应用unchecked和external（V3）

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

// V3：应用unchecked和external
contract ERC20V3 {
    string public name;
    string public symbol;
    uint8 public decimals;

    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
```

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

    constructor(string memory _name, string memory _symbol) {
        name = _name;
        symbol = _symbol;
        decimals = 18;
    }

    // 使用external而不是public
    function transfer(address to, uint256 amount) external returns (bool) {
        uint256 senderBalance = balanceOf[msg.sender];
        require(senderBalance >= amount, "Insufficient balance");

        // 使用unchecked，因为已经检查过
        unchecked {
            balanceOf[msg.sender] = senderBalance - amount;
        }
        balanceOf[to] += amount;
        emit Transfer(msg.sender, to, amount);
        return true;
    }
}

// Gas消耗：约70,000 Gas（累计节省30%）
```

7.4 应用局部存储指针（V4）

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

// V4：应用局部存储指针
contract ERC20V4 {
    string public name;
    string public symbol;
    uint8 public decimals;

    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

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

    constructor(string memory _name, string memory _symbol) {
        name = _name;
        symbol = _symbol;
        decimals = 18;
    }
```

```
    function transfer(address to, uint256 amount) external returns (bool) {
        // 使用局部存储指针
        uint256 senderBalance = balanceOf[msg.sender];
        require(senderBalance >= amount, "Insufficient balance");

        unchecked {
            balanceOf[msg.sender] = senderBalance - amount;
        }

        // 使用局部存储指针访问接收者余额
        uint256 receiverBalance = balanceOf[to];
        balanceOf[to] = receiverBalance + amount;

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

// Gas消耗：约65,000 Gas（累计节省35%）
```

7.5 优化效果总结

版本	Gas消耗	节省比例	应用技巧
V1	100,000	-	未优化
V2	85,000	15%	存储打包
V3	70,000	30%	+ unchecked + external
V4	65,000	35%	+ 局部存储指针

实际成本对比：

假设Gas价格是50 gwei，ETH价格是3000美元：

- V1版本：100,000 × 50 gwei = 0.005 ETH = 15美元
- V4版本：65,000 × 50 gwei = 0.00325 ETH = 9.75美元
- **每笔交易节省**：5.25美元

应用的关键技巧：

1. 变量打包：将小类型变量放在一起
2. unchecked循环：在安全的场景下跳过溢出检查
3. external函数：使用external而不是public
4. 局部存储指针：避免重复读取存储

这些技巧组合使用，效果非常显著！

---

8. 工具与检查清单

工欲善其事，必先利其器。让我们了解一些Gas优化的工具和检查清单。

8.1 Gas优化工具

1. Hardhat Gas Reporter：

Hardhat Gas Reporter是一个Hardhat插件，可以在测试时自动生成每个函数的Gas消耗报告。

```
// hardhat.config.js
require("hardhat-gas-reporter");

module.exports = {
  gasReporter: {
    enabled: true,
    currency: "USD",
    gasPrice: 50
  }
};
```

2. Foundry Gas Snapshots：

Foundry是一个新的开发框架，它的Gas快照功能可以对比前后版本的Gas差异。

```
# 生成Gas快照
forge snapshot

# 对比Gas差异
forge snapshot --diff
```

3. Tenderly：

Tenderly是一个在线的模拟和分析工具，可以可视化地分析交易的Gas消耗，非常适合调试复杂的交易。

4. EVM.codes：

EVM.codes是一个操作码参考网站，可以查询每个操作码的Gas成本，对于深入优化非常有用。

8.2 优化检查清单

在优化合约时，可以对照这个清单逐项检查：

存储优化：

- [ ] 变量是否打包？小类型变量是否放在一起？
- [ ] 是否使用了局部存储指针？避免重复读取存储？
- [ ] 是否用事件替代了不必要的存储？

数据类型优化：

- [ ] 函数参数是否使用uint256？
- [ ] 循环计数器是否使用uint256？

- [ ] 是否避免了不必要的类型转换?

函数优化:

- [ ] 外部函数是否使用external而不是public?
- [ ] 内部函数是否使用internal或private?
- [ ] 检查顺序是否优化? 便宜的检查在前?

其他优化:

- [ ] 是否使用了constant和immutable?
- [ ] 循环是否使用unchecked?
- [ ] 是否提供了批量操作?
- [ ] 是否避免了链上遍历大数组?

这个清单可以帮助我们系统地审查合约，避免遗漏优化点。

---

9. 实践练习

理论学习之后，实践是巩固知识的最好方式。以下是不同难度的练习题目。

9.1 练习1：存储打包优化（二星难度）

任务：优化以下合约的存储布局，减少存储槽的使用。

要求:

1. 分析当前合约使用的存储槽数量
2. 重新排列变量顺序，实现存储打包
3. 对比优化前后的Gas消耗

初始代码:

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

contract StoragePackingExercise {
    uint256 public value1;
    bool public flag1;
    uint256 public value2;
    uint8 public count;
    uint256 public value3;
    bool public flag2;

    // 当前使用6个存储槽
}
```

9.2 练习2：函数优化（二星难度）

任务：优化以下合约的函数可见性和检查顺序。

要求：

1. 将public函数改为external（如果适用）
2. 优化检查顺序，便宜的检查在前
3. 使用局部存储指针优化存储访问

初始代码：

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

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

    function transfer(address to, uint256 amount) public {
        require(expensiveCheck(), "Expensive check failed");
        require(amount > 0, "Amount must be greater than 0");
        require(to != address(0), "Invalid recipient");

        balances[msg.sender] -= amount;
        balances[to] += amount;
    }

    function expensiveCheck() internal view returns (bool) {
        // 模拟昂贵的检查
        return true;
    }
}
```

9.3 练习3：批量操作实现（三星难度）

任务：实现一个批量转账函数。

要求：

1. 支持批量转账给多个地址
2. 设置合理的批量数量上限
3. 先验证总额，再执行批量操作
4. 使用unchecked优化循环

9.4 练习4：综合优化（四星难度）

任务：对一个完整的ERC20代币合约进行综合优化。

要求：

1. 应用所有学到的优化技巧
2. 使用工具测量Gas消耗
3. 对比优化前后的效果
4. 编写优化报告

---

10. 学习检查清单

完成本课后，你应该能够：

Gas成本分析：

- [ ] 理解Gas费用的计算公式
- [ ] 知道不同操作的Gas成本差异
- [ ] 能够分析合约的Gas消耗

存储优化：

- [ ] 理解存储槽打包的原理
- [ ] 会使用位域打包优化配置参数
- [ ] 会使用局部存储指针减少SLOAD

数据类型优化：

- [ ] 知道何时使用小类型，何时使用uint256
- [ ] 理解Mapping和Array的适用场景
- [ ] 知道避免链上遍历大数组

函数优化：

- [ ] 理解不同可见性修饰符的Gas差异
- [ ] 会使用短路求值优化检查顺序
- [ ] 知道用事件替代不必要的存储

批量操作和unchecked：

- [ ] 会实现批量操作函数
- [ ] 知道安全使用unchecked的场景
- [ ] 理解批量操作的优势

工具使用：

- [ ] 会使用Gas测量工具
- [ ] 会使用优化检查清单
- [ ] 能够系统化地优化合约

---

11. 总结

Gas优化是智能合约开发中非常重要的技能。通过本课的学习，你应该已经掌握了：

1. **存储优化**：
    - 存储槽打包：合理安排变量顺序
    - 位域打包：极致优化配置参数
    - 局部存储指针：减少重复读取

2. **数据类型优化**：
    - 选择合适的类型
    - 理解Mapping和Array的适用场景
    - 避免链上遍历大数组
3. **函数优化**：
    - 合理使用external
    - 短路求值优化
    - 用事件替代存储
4. **批量操作**：
    - 减少交易次数
    - 降低基础Gas费用
5. **unchecked使用**：
    - 在安全的场景下跳过溢出检查
    - 不要为了省Gas牺牲安全性

关键原则：

1. **测量优先**：不要盲目优化，要用工具找到真正的瓶颈
2. **安全第一**：不要为了省Gas牺牲安全性，这是底线
3. **20/80法则**：把精力放在优化热点函数上
4. **用户至上**：优化的最终目的是降低用户的使用成本

通过合理的优化，我们可以将合约的Gas成本降低30%到80%，显著提升合约的经济性和用户体验。记住：优化是一个持续的过程，需要在实际项目中不断实践和改进。
