Gas优化技巧

Solidity智能合约开发系列课程 – 第9.1课

讲师：[Layer]

GAS OPTIMIZATION

课程内容导航

1. Gas成本分析

Gas费用构成

EVM操作成本

实际案例分析

2. 存储优化

存储槽打包

位域打包

局部存储指针

3. 数据类型优化

整数类型选择

Mapping vs Array

4. 函数优化

可见性修饰符

短路求值

事件替代存储

5. 批量操作

批量处理原理

实现技巧

6. unchecked使用

安全使用场景

注意事项

什么是Gas?

交易费用公式

交易费用 = Gas使用量 × Gas价格

示例计算:

21,000 Gas × 50 gwei = 0.00105 ETH

≈ $3.15（ETH价格$3,000时）

类比说明

核心要点

- Gas = 计量单位
- 用户设置Gas价格
- 操作复杂度决定Gas用量
- 优化目标：降低Gas用量

不同操作的Gas成本

操作类型	Gas成本	说明
基础运算		
ADD（加法）	3	最便宜
MUL（乘法）	5	很便宜
存储操作		
SSTORE（新值）	20,000	最昂贵!
SSTORE（修改/读取）	5,000/2,100	很贵/较贵
其他操作		
CALL（外部调用）	2,600+	中等
LOG（事件）/CREATE（部署）	375+/32,000+	便宜/很贵

SSTORE = 20,000 Gas

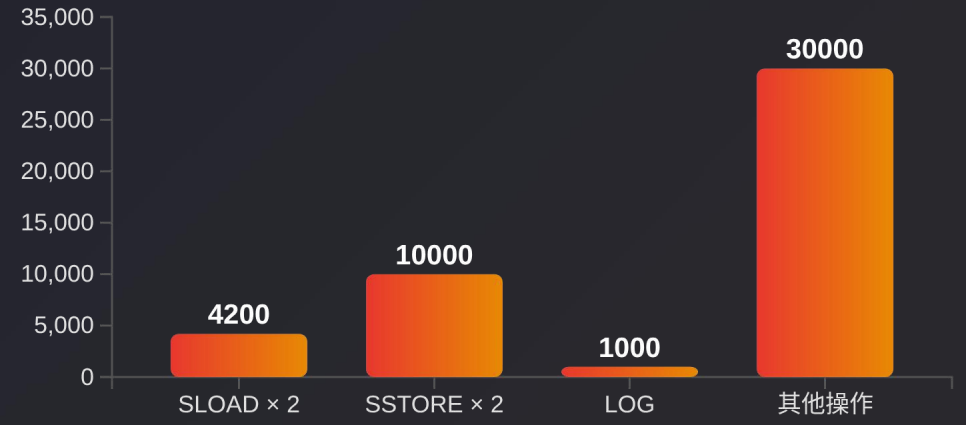
VS

6,666 × ADD = 19,998 Gas

真实案例：简单转账的Gas消耗

```
// ERC20转账函数
function transfer(address to, uint256 amount) public {
balances[msg.sender] -= amount; // SLOAD + SSTORE
balances[to] += amount; // SLOAD + SSTORE
emit Transfer(msg.sender, to, amount);
}
```

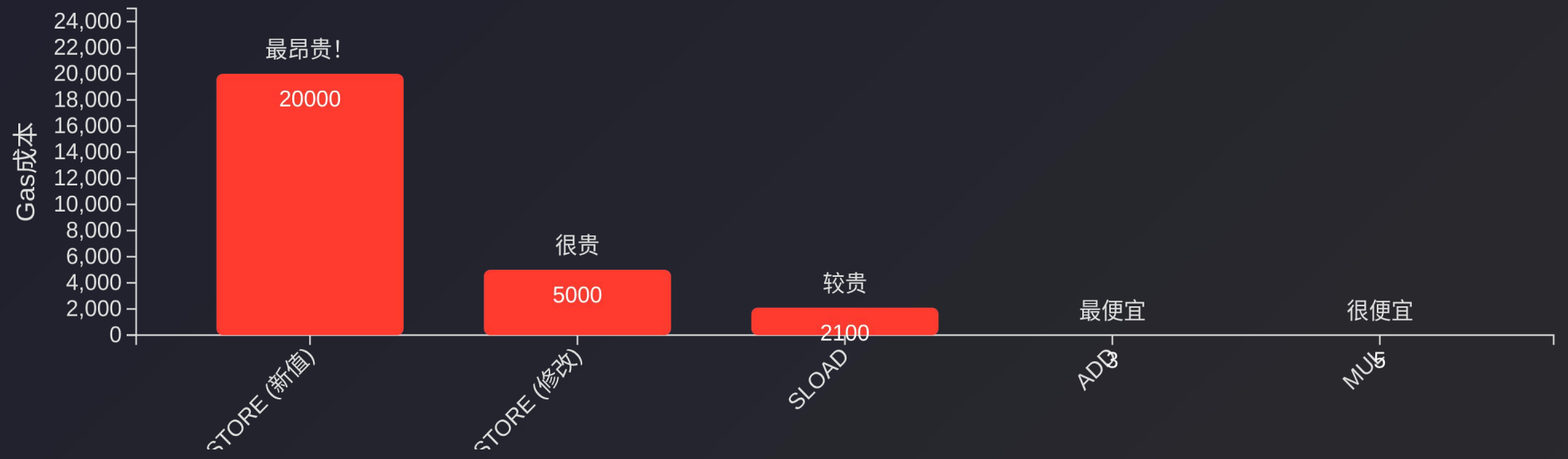
成本分解



SLOAD × 2	4,200 Gas
SSTORE × 2	10,000 Gas
LOG（事件）	1,000 Gas
其他操作	30,000 Gas
总计	≈ 45,000 Gas

45,000 Gas × 100 gwei = 0.0045 ETH ≈ $13.5

为什么存储优化最重要?



1次SSTORE = **6,666次** ADD运算

存储槽打包：合理安排变量顺序

未优化（80,000 Gas）

```
contract Bad {
    uint8 a; // slot 0
    uint256 b; // slot 1
    uint8 c; // slot 2
    uint256 d; // slot 3
}

// 4个存储槽 × 20,000 = 80,000 Gas
```

80,000 Gas

优化后（60,000 Gas）

```
contract Good {
    uint8 a; // slot 0
    uint8 c; // slot 0（共享）
    uint256 b; // slot 1
    uint256 d; // slot 2
}

// 3个存储槽 × 20,000 = 60,000 Gas
```

60,000 Gas

节省：20,000 Gas（25%）

位域打包：一个变量存储多个值

什么是位域打包？

将多个小变量压缩到一个uint256中，减少存储操作次数，节省Gas费用。

Aave的极致优化

```
struct ReserveConfiguration {
// 将20+个参数压缩到一个uint256！
// bit 0–15: LTV
// bit 16–31: 清算阈值
// bit 32–47: 清算奖励
// bit 48–55: 小数位
// bit 56: 是否激活
// ... 更多参数
uint256 data; // 仅1个存储槽！ }
```

效果对比

节省效果

80,000 Gas （80%）

实现要点

- 使用位运算符（&, |, ^, ~）
- 合理规划位域位置和大小
- 添加辅助函数简化访问

局部存储指针：避免重复读取

未优化（重复SLOAD）

```
function bad() external {
    reserves[asset].rate = newRate;
// SLOAD + SSTORE
    reserves[asset].index = newIndex;
// SLOAD + SSTORE
    reserves[asset].timestamp = now;
// SLOAD + SSTORE
}
// 3次SLOAD浪费 = 6,300 Gas
```

浪费 6,300 Gas

优化（使用指针）

```
function good() external {
Reserve storage r = reserves[asset];  // 仅1次SLOAD
    r.rate = newRate;
// 直接SSTORE
    r.index = newIndex;
// 直接SSTORE
    r.timestamp = now;
// 直接SSTORE
}
// 节省 6,300 Gas
```

节省 6,300 Gas

关键点： 频繁访问 → 先赋值给storage指针

选择正确的数据类型

场景	推荐类型	原因
存储（需打包）	uint8/uint16/uint32	节省空间
存储（不打包）	uint256	无需转换
函数参数	uint256	EVM原生支持
内存变量	uint256	避免类型转换
循环计数器	uint256	避免转换开销

推荐

```
//    推荐
function process(uint256 amount) external {
    uint256 result = amount * 2;
}
```

不推荐

```
//    不推荐
function process(uint8 amount) external {
    uint256 result = uint256(amount) * 2; // 额外转换
}
```

Mapping vs Array：如何选择？

Mapping（映射）

- ✓ 访问成本固定（一次SLOAD）
- ✓ 适合随机访问
- ✗ 不能遍历
- ✗ 不能获取长度

💡 适用：用户余额、配置参数

Array（数组）

- ✓ 可以遍历
- ✓ 可以获取长度
- ✗ 访问需要计算索引
- ✗ 遍历成本高（链上危险）

💡 适用：需要遍历的列表

⚠ 永远不要在链上遍历大数组！链下遍历，链上只访问特定元素

函数可见性的Gas差异

成本排序

public	external	internal	private
昂贵	较贵	便宜	最便宜

原因说明

public

需要支持内部和外部调用，参数复制到memory

external

只支持外部调用，参数用calldata（更便宜）

internal

内部调用，无ABI编码开销

private

类似internal，但访问权限更严格

最佳实践

- **外部调用的函数 → 用 external**
  减少参数处理开销
- **内部辅助函数 → 用 internal 或 private**
  避免不必要的public修饰符

短路求值：便宜的检查放前面

未优化（浪费Gas）

浪费 20,000 Gas

```
require(expensiveOracleCall() && amount > 0);

// 先执行20,000 Gas的Oracle调用
// 即使amount=0，也浪费了20,000 Gas
```

优化后（节省Gas）

节省 20,000 Gas

```
require(amount > 0);                // 先检查便宜的
require(expensiveOracleCall());     // 再执行昂贵的

// 如果第一个检查失败，立即返回，节省20,000 Gas
```

便宜检查在前 → 昂贵检查在后

事件替代存储：节省97%成本

Storage（存储历史记录）

60,000 Gas

Event（事件日志）

1,500 Gas

节省 58,500 Gas (97%)

使用Storage场景

- 链上需要读取的数据
- 影响合约逻辑的数据

使用Event场景

- 仅记录/查询的历史数据
- 前端显示的数据

不推荐：使用Storage

```
Transaction[] public history; // 60,000 Gas per record
```

推荐：使用Event

```
event TransactionExecuted(address user, uint256amount);
emit TransactionExecuted(msg.sender, amount); // 1,500 Gas
```

批量操作：减少交易次数

成本对比

分3次转账

交易1	51,000 Gas
交易2	51,000 Gas
交易3	51,000 Gas

总计: 153,000 Gas

VS

批量转账

基础Gas	21,000 Gas
批量操作	90,000 Gas

总计: 111,000 Gas

节省: 42,000 Gas（27%）

实现要点

设置批量数量上限

每次批量处理最多100笔交易，避免Gas消耗过高

使用calldata参数

而非memory，减少Gas消耗约20%

先验证总额，再执行

避免部分失败情况，提高交易可靠性

最佳实践

- 批量操作可显著降低Gas成本
- 合理设置批量大小，平衡成本和效率
- 适用于：批量转账、批量更新等场景

unchecked：跳过溢出检查

Solidity 0.8.x的成本

每次运算都有溢出检查：

- 加法：+20 Gas
- 减法：+20 Gas
- 乘法：+30 Gas
- 循环：+2,000+ Gas

安全使用场景

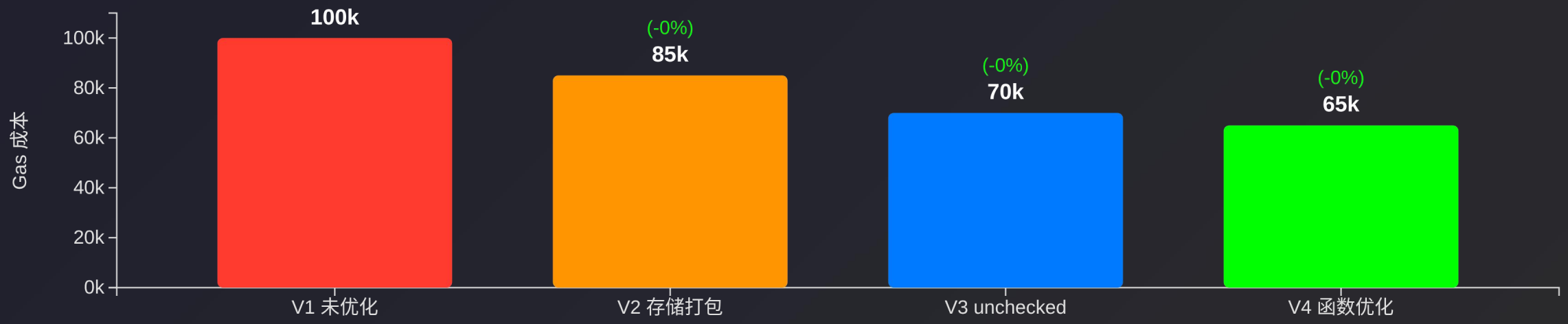
```
// 循环计数器
for (uint256 i = 0; i < length;) {
process(data[i]);
unchecked { i++; }
}
```

```
// 已经require检查
require(balance >= amount);
unchecked { balance -= amount; }
```

危险用法

```
// 可能溢出！
unchecked { balance += randomAmount; }
```

实战：ERC20代币优化



V1: $15.00 → V4: $9.75 每笔交易节省 $5.25

变量打包　unchecked循环　external函数　局部指针

Gas优化工具箱

推荐工具

Hardhat Gas Reporter
查看每个函数的Gas消耗

Foundry Gas Snapshots
对比前后版本差异

Tenderly
在线模拟和分析

EVM.codes
查询操作码成本

优化检查清单

- [x] 变量是否打包?
- [x] 是否使用局部指针?
- [x] 函数是否用external?
- [x] 是否使用constant/immutable?
- [x] 循环是否用unchecked?
- [x] 是否提供批量操作?
- [x] 事件替代不必要的存储?

Gas优化核心要点

六大优化技巧

- 存储优化 → 打包变量，使用指针
- 数据类型 → 选对类型，避免转换
- 函数优化 → 用external，短路求值
- 批量操作 → 减少交易，降低基础成本
- unchecked → 安全场景跳过检查
- 事件日志 → 替代不必要的存储

四大优化原则

- 测量优先：用工具找瓶颈
- 安全第一：不牺牲安全性
- 20/80法则：优化热点函数
- 用户至上：降低用户成本

智能合约安全最佳实践

- 重入攻击防护
- 整数溢出保护
- 访问控制机制

本节课你将学到

- 理解Gas成本的构成和计算方式
- 掌握6种核心Gas优化技巧
- 学会使用工具测量和分析Gas消耗
- 能够将合约Gas成本降低30-80%
- 在实际项目中应用优化技巧