Solidity 智能合约开发入门

一、EVM（以太坊虚拟机）基础环境

1.1 EVM 本质与架构

EVM（以太坊虚拟机）是以太坊及其生态中智能合约运行、编译和执行的环境，是一个栈式虚拟机。所有以太坊生态的合约都在 EVM 上运行、编辑和编译。

1.2 EVM 四大执行区（存储区）

合约的性能优化、安全漏洞都与这四个执行区直接相关：

- Stack（栈）

- EVM 采用栈式运行模型
- 指令与操作数按顺序入栈，执行时再出栈计算
- 示例：1+1 的执行过程就是通过入栈与出栈完成

- Storage（链上存储）

- 用于永久保存合约状态数据
- 合约中需要“上链保存”的数据都会写入 storage
- Gas 成本最高的部分，优化核心是减少 storage 读写

- Memory（运行时内存）

- 合约执行过程中的临时内存
- 生命周期仅限于一次函数调用，不上链

- Calldata（调用数据）

- 用户与合约交互时传入的参数区域
- 只读属性，用于函数输入参数

类比理解：

- Storage 类似硬盘（永久存储）
- Memory/Calldata 类似运行时内存（临时存储）

二、Gas 成本与合约优化

2.1 Gas 成本核心

合约优化主要集中在两点：

1. 安全性
2. Gas Fee 成本

2.2 Storage 的 Gas 消耗

- 初始化写入 storage ≈ 20,000 gas
- 修改已有 storage ≈ 5,000 gas（量级）

2.3 优化策略

目标：尽量少存、尽量少改、尽量避免不必要的 storage 访问

三、Solidity 开发环境

3.1 推荐工具：Remix IDE

- 线上合约编辑器，提供私有链与测试链
- 支持编译、部署、debug
- 可查看执行过程、memory、storage 变化

3.2 Solidity 版本管理

- 版本更新频繁（0.9.0 即将推出）
- 不同版本在内存管理和安全机制上存在差异
- 教学基准版本：0.8.20
- ^0.8.20 表示支持 0.8.20 及以上版本编译

重要：阅读协议源码时，必须关注其 Solidity 版本

四、Solidity 数据类型体系

4.1 值类型（Value Types）

直接存值，存储成本相对较低：

- bool
- true / false
- 用于条件判断或返回成功与否
- int / uint（整数类型）
- uint：无符号整数（不带符号位）
- int：有符号整数（带符号位）
- 可指定位宽：uint8 ~ uint256
- 优化技巧：位宽越小，storage 占用越小，gas 越省
- address（地址类型）
- 用于交互目标地址或用户钱包 EOA 地址
- 有类型检查，不满足会回滚报错
- 重要规则：用户地址用 address，不要用 string 存储
- bytes32（定长字节）
- 常用于非地址类的数据存储

4.2 引用类型（Reference Types）

存储的是“引用”，会开辟新的存储空间，gas 成本更高：

- string（字符串）
- 动态数组
- bytes / bytes[]
- mapping（映射）

4.3 Mapping 的特殊作用

常用于账本设计：

- 余额账本
- 授权账本

五、ERC20 代币合约设计

5.1 ERC20 标准概述

ERC20 是以太坊代币的标准接口规范，满足该标准的合约具备通用代币逻辑。

5.2 核心链上存储数据

以下数据均需存储在 storage 中：

- string name：代币名称
- string symbol：代币符号
- uint8 decimals：精度（链上无浮点数）
- uint256 totalSupply：总发行量
- address owner：合约拥有者

5.3 ERC20 账本结构

两本核心账本构成 ERC20 的基础：

- 余额账本

记录每个地址持有的代币数量

- 授权账本

记录 A 地址授权给 B 地址的操作额度

六、构造函数与初始化

Constructor 作用

- 在合约部署时执行
- 用于初始化链上状态
- 接收部署参数（名称、符号、精度等）

注意：纯工具型合约可以不写 constructor

七、函数设计与可见性

7.1 函数可见性修饰符

- external
- 仅能从外部调用
- public
- 内外皆可调用
- internal
- 仅合约内部可调用
- private
- 仅当前合约可调用

7.2 状态可变性修饰符

- view

- 只读函数
- 读取链上状态，但不修改
- pure
- 既不读取也不修改链上状态

八、ERC20 核心方法实现
8.1 查询总供应量
8.2 查询余额
8.3 转账 transfer
核心逻辑：
1. 参数校验
   - 数量 ＞0
   - 目标地址非零地址（除非设计销毁逻辑）
   - 转出方余额充足

2. 余额更新
   - 转出方余额减少
   - 转入方余额增加

3. 事件触发

8.4 授权 approve
使用场景：
- 多数 dApp 交互的第一步
- 授权协议合约操作用户代币

实现要点：
1. 修改授权账本
2. 触发 Approval 事件

九、事件（Event）机制
9.1 事件的作用
- 广播状态变更给全网
- 供区块浏览器与链下系统监听
- 链上到链下同步的重要机制

9.2 ERC20 核心事件
- Transfer：转账事件
- Approval：授权事件
- Mint：铸造事件

9.3 indexed 索引
- 最多三个 indexed 参数

- 便于链下检索和过滤

开发实践：DEX 等协议大量依赖事件监听，而非频繁读取 storage

十、Mint 机制与安全设计

安全考虑
- Mint 通常设为 internal
- 防止任何用户随意铸币
- 常见模式：external 函数负责权限校验 + internal 函数负责状态变更

十一、Remix 开发全流程

11.1 编译阶段

- 在虚拟 EVM 中检查错误
- 选择正确的编译器版本

11.2 部署阶段

1. 合约代码 → 字节码
2. 发送到链上部署
3. Constructor 在部署阶段执行

11.3 调用测试

- 只读函数无需 gas
- 写操作需发交易并消耗 gas
- 私有链模式可用于快速 debug

十二、错误处理机制

12.1 require

- 条件校验 + 错误信息
- Gas 消耗相对较高

12.2 revert

- 回滚但不返回字符串
- Gas 优化选择

12.3 assert

- 用于逻辑上"不可能失败"的断言
- 失败会消耗剩余 gas
- 现代开发中较少使用

12.4 自定义错误

- 替代字符串错误信息
- 进一步优化 gas

十三、整数溢出与 unchecked

13.1 Solidity 0.8+ 的默认保护

- 自动进行整数溢出检查
- 增加安全性但消耗少量 gas

13.2 unchecked 使用

- 在确定不会溢出时使用
- 节省少量 gas
- 初学阶段不建议使用

十四、Modifier 修饰符

14.1 基本概念

用于抽象函数的前置条件校验

14.2 使用优势

1. 代码复用：避免重复编写 require
2. 权限统一：确保多函数使用相同权限逻辑
3. Gas 优化：减少部署和执行成本

Q&A

一、在推特等社交平台看到有人故意暴露钱包助记词，导入后发现：

1. 账户有余额但无法转出（提示需要 gas）

2. 转入 gas 后，gas 立即被转走

3. 这是一种什么实现逻辑？是抢跑吗？

1. 合约层权限限制

- 账户类型：暴露的地址可能是合约账户（非普通 EOA 钱包）
- 权限控制：通过 modifier 修饰符限制转账权限
- 表面余额假象：账户显示有余额，但转账方法被 onlyRealOwner 修饰，只有预设的 realOwnerAddress 才能转出资金
- 诱导行为：骗子通过公开助记词诱导用户转入 gas，但用户永远无法转出原始余额

2. 抢跑机器人（Front-running Bot）

- 监听机制：Bot 监听特定地址的所有交易事件（Event）
- 触发条件：一旦检测到该地址有 gas 转入
- 执行策略：
  1. 立即以更高的 gas fee 发起转账交易
  2. 利用矿工优先打包高 gas 交易机制，抢先转走资金
  3. 可配置自动化：转入→转走→再转入（循环诱骗）
- 技术基础：依赖以太坊交易排序机制，非合约逻辑缺陷

安全建议
- 绝不导入来历不明的助记词/私钥
- 测试小额：如需测试，先转入极小金额验证
- 查看合约：通过区块浏览器检查地址是否为合约账户

二、modifier 是什么？具体操作逻辑是怎样的？

1. 基本定义
- 功能定位：Solidity 中的函数修饰符，用于封装前置条件检查逻辑
- 核心价值：代码复用、权限统一、Gas 优化

2. 与传统 require 对比

方式	代码示例	缺点
每个函数单独 require	每个函数都写 `require(msg.sender == owner, "...")`	代码重复、部署 Gas 高、维护困难
使用 modifier	函数后加 `onlyOwner` 修饰符	一次定义，多处复用，Gas 更优

3. 扩展应用场景
- 重入保护：nonReentrant 修饰符防止重入攻击
- 时间锁：onlyAfter(block.timestamp)限制执行时间
- 角色权限：onlyAdmin、onlyMinter 等多角色控制
- 参数校验：validAmount(amount)统一校验参数有效性

4. Gas 优化原理
- 部署时：modifier 逻辑只编译一次，减少合约字节码大小
- 执行时：跳转到统一校验代码，比分散 require 更高效
- 开发时：减少代码冗余，降低出错概率

三、
问题 1：发币是否需要多个合约？
- 每个 ERC20 代币都是独立合约：每个代币项目都需要单独部署自己的 ERC20 合约
- 链上数据不可变性：
  - 部署时设定的参数（名称、符号、精度等）永久记录在链上
  - 除非合约内预设了修改方法，否则这些参数永远无法更改
  - 示例：部署时设置 name="LXDAO"，该代币将永远叫这个名字
- 实际意义：要发行自己的代币，必须编写并部署独立的 ERC20 合约

问题 2：EVM 是否存在并发问题？
- 无传统并发：EVM 是单线程顺序执行，不存在多线程并发问题

- 但有重入风险：
  - 防护措施：
  1. 检查-生效-交互模式：先更新状态，再执行外部调用
  2. 重入锁：使用 modifier 实现

四、

问题 1：新代币标准是否兼容 ERC20？

标准独立性

- ERC-721：NFT 标准，每个 token 唯一
- ERC-1155：多代币标准，可同时包含同质化和非同质化代币
- 各自独立：每个标准是独立的接口规范，不存在“兼容”关系

为什么需要标准？

1. 钱包支持：钱包应用需要统一的接口来显示余额、转账
2. 交易所集成：交易所需要标准方法进行充提币
3. DApp 交互：DeFi 协议需要标准接口进行代币操作

非标准代币的问题

- 如果代币 A 的转账方法叫 sendToken，代币 B 叫 transferFunds
- 每个 DApp 都需要为每个非标代币编写特殊逻辑
- 钱包无法统一显示和管理

问题 2：标准能否升级？接口是否可变？

接口稳定性原则

- 接口不变：ERC20 标准的方法签名和参数不可更改
- 实现可变：合约内部逻辑可以自由实现和优化

为什么接口不能变？

1. 向后兼容：已有数千个 ERC20 代币和数百个集成应用
2. 生态依赖：钱包、交易所、DApp 都依赖固定接口
3. 升级成本：改变标准会导致整个生态需要更新

如何扩展功能？

- 新标准：创建新的 ERC 标准（如 ERC-777、ERC-1363）
- 封装模式：在标准接口基础上添加额外功能
- 代理模式：通过可升级合约添加新方法

五、除了标准，还有哪些“轮子”可用？

1. OpenZeppelin 合约库

- 定位：经过审计的标准合约实现库
- 优势：
  - 免去重复编写标准逻辑
  - 经过多次安全审计

- 持续维护更新

2. 可升级合约模式

问题背景

- 传统合约部署后无法修改
- 发现 bug 或需要升级时，必须部署新合约并迁移用户

升级流程

1. 部署新的 Logic 合约（V2）
2. 更新 Proxy 中的逻辑合约地址指向 V2
3. 用户继续使用原 Proxy 地址，自动获得新功能

3. 钻石架构（EIP-2535）

- 多逻辑合约：一个代理可路由到多个实现合约
- 模块化升级：不同功能模块独立升级
- Gas 优化：避免重复部署存储合约