solidity学习

evm中的两种值类型

基本类型（Value Types）

定义：直接存储值的数据类型，变量之间赋值是复制值本身。

常见类型：

uint, int

bool

address

bytes1 到 bytes32

存储特性：

分配在固定的 storage slot

赋值是值复制（copy by value）

引用类型（Reference Types）

定义：存储的是对数据的“引用”或“指针”，赋值传的是地址。

常见类型：

array（数组）

mapping（映射）

struct（结构体）

存储特性：

变量本身保存的是对实际数据的引用

赋值是引用复制（copy by reference）

需要 keccak256(slot) 来定位实际数据地址（尤其是动态结构）

函数基础语法

// 基本函数结构

function functionName(

参数类型 参数名

)

可见性修饰符

状态可变性修饰符

returns (返回类型)

{

// 函数体

}

## **可见度修饰符**

| **可见性修饰符** | **当前合约** | **其他合约** | **子合约** | **外部账户** |
| --- | --- | --- | --- | --- |
| public | 1 | 1 | 1 | 1 |
| external | 0 | 1 | 0 | 1 |
| private | 1 | 0 | 0 | 0 |
| internal | 1 | 0 | 1 | 0 |

状态可变性修饰符

contract StateModifierDemo {

uint public count = 0;

// pure: 不读取也不修改状态

function add(uint a, uint b) public pure returns (uint) {

return a + b;

}

// view: 只读取状态，不修改状态

function getCount() public view returns (uint) {

return count;

}

// 默认: 可以修改状态

function increment() public {

count += 1;

}

// payable: 可以接收以太币

function deposit() public payable {

// 函数可以接收 ETH

}

}

读取 写入

pure 否 否

view 是 否

default 是 是

payable

payable 是 Solidity 中的一个函数修饰符，用于标识一个函数可以接收以太币（ETH）。当一个函数被标记为 payable 时，它允许在调用该函数时向合约发送以太币。这是合约接收以太币的唯一方式。

接收以太币：只有被标记为 payable 的函数才能接收以太币。如果尝试向一个没有 payable 修饰符的函数发送以太币，交易将会失败。

合约余额：通过 payable 函数接收到的以太币会增加合约的余额。

使用场景：通常用于实现合约的支付功能，比如购买、捐赠等。

contract PayableDemo {

function deposit() public payable {

// 函数可以接收 ETH

}

}

EVM特性

交易流程图

1. 用户发起交易

|

↓

2. 创建新的 EVM 实例

|

↓

3. 加载合约字节码

|

↓

4. 分配 Stack 空间（1024 slots）

|

↓

5. 执行合约代码

|

↓

6. 更新区块链状态（如果需要）

|

↓

1. 销毁 EVM 实例

详细说明

1、用户发起交易

用户签名交易

设定 gas limit 和 gas price

指定目标合约地址和调用数据

2、创建新的 EVM 实例

为每笔交易创建独立的 EVM 环境

初始化执行上下文

准备内存和存储空间

3、加载合约字节码

从区块链状态中读取合约字节码

将字节码加载到 EVM 中

准备执行环境

4、分配 Stack 空间

分配 1024 个 slots

每个 slot 256 位

用于存储临时计算结果

5、执行合约代码

逐条执行操作码

进行状态检查和 gas 计算

处理函数调用和返回值

6、更新区块链状态

写入存储变更

更新账户余额

触发事件日志

7、销毁 EVM 实例

清理内存

释放资源

返回执行结果

注意事项

整个过程是原子性的：要么全部成功，要么全部失败

Gas 限制贯穿整个执行过程

状态变更只在交易成功执行后才会提交

EVM存储结构

1. Stack (栈)

+----------------+

| 基本类型的值 | <- 函数内的局部变量

| 引用的地址 | <- 指向其他存储位置

+----------------+

2. Memory (内存)

+----------------+

| 临时数据 | <- 函数执行期间的临时数据

| 函数参数 | <- memory 类型的参数

| 返回数据 | <- 函数返回值

+----------------+

3. Storage (存储)

+----------------+

| 状态变量 | <- 合约的永久存储数据

| 映射数据 | <- mapping 数据

| 数组数据 | <- storage 数组

+----------------+

EVM Stack (固定大小的栈空间)

+------------------+

| 空闲空间 | <- 1024 个槽位（slots）

| ⬇ | 每个槽位 32 字节（256 位）

+------------------+

| 当前使用空间 | <- 随函数执行压入/弹出

+------------------+

Memory： 存在于 EVM 执行环境中 临时性的，交易执行完就清除 线性寻址（0x00, 0x20, 0x40...）

Storage： 存在于区块链状态中 永久性的，写入区块 使用 slot 和 keccak256 哈希定位

基本类型

在 Solidity 中，直接存储在栈（Stack）中的基本类型包括：

1. 整型（Integer）

contract StackTypes {

function integerTypes() public pure {

// 所有整型都存储在栈中

uint256 a = 1;

uint8 b = 2;

int256 c = -1;

int8 d = -2;

}

}

2. 布尔型（Boolean）

function booleanTypes() public pure {

bool isTrue = true;

bool isFalse = false;

}

3. 地址（Address）

function addressTypes() public pure {

address addr = 0x123...;

address payable payableAddr = payable(0x123...);

}

固定大小字节数组（Fixed-size Bytes）

function bytesTypes() public pure {

bytes1 b1 = 0x12;

bytes32 b32 = 0x123...;

}

枚举（Enum）

enum Status { Active, Inactive }

function enumTypes() public pure {

Status status = Status.Active; // 实际存储为 uint8

}

重要说明：

这些类型在函数（相关文档：数据位置修饰符）调用时：

作为参数传递时是值传递

不需要指定 memory 等位置修饰符

// ✅ 正确：不需要 memory

function example(uint256 num, bool flag, address addr)

public pure {

// ...

}

// ❌ 错误：不能为基本类型指定 memory

function wrong(uint256 memory num) public pure {

// ...

}

大小限制：

function stackLimits() public pure {

// EVM 栈深度限制为 1024

// 每个值占用一个栈槽（32字节）

}

与引用类型对比：

contract TypeComparison {

// 基本类型：直接存储在栈中

uint256 public stackVar = 123;

// 引用类型：存储引用在栈中，数据在其他位置

string public stringVar = "hello"; // 数据在存储中

uint256[] public arrayVar; // 数据在存储中

}

在函数（文档地址：数据位置修饰符）中的使用：

contract StackUsage {

function calculate() public pure returns (uint256) {

// 这些变量都在栈中

uint256 a = 1;

uint256 b = 2;

uint256 c = a + b;

return c;

} // 函数结束时栈变量自动清除

}

注意事项：

栈变量的生命周期仅限于函数执行期间

栈变量不需要手动管理内存

栈操作的 gas 成本较低

需要注意栈深度限制（1024层）

这些类型的特点：

大小固定

值类型（非引用）

操作简单直接

gas 成本低

主要的引用类型：

特点： 在栈中：只存储引用（地址/指针） 实际数据：存储在 Memory 或 Storage 中

数组（Array）

contract ArrayExample {

// Storage 数组（状态变量）

uint[] public storageArray; // 存储在链上

function example() public {

// Memory 数组（局部变量）

uint[] memory memoryArray = new uint[](3);

// 栈中只存储数组的引用（指针）

// 实际数据在 Memory 或 Storage 中

}

}

字符串（String）

contract StringExample {

// Storage 字符串

string public storageStr = "hello"; // 存储在链上

function example() public pure {

// Memory 字符串

string memory memoryStr = "world";

// 栈中存储字符串的引用

// 实际字符串数据在 Memory 或 Storage

}

}

结构体（Struct）

contract StructExample {

struct Person {

string name;

uint age;

}

// Storage 结构体

Person public storagePerson;

function example() public {

// Memory 结构体

Person memory memoryPerson = Person("Alice", 20);

// 栈中存储结构体的引用

// 实际数据在 Memory 或 Storage

}

}

映射（Mapping）

contract MappingExample {

// 只能声明为 Storage

mapping(address => uint) public balances;

function example() public {

// ❌ 不能创建 Memory 映射

// mapping(address => uint) memory memoryMap; // 错误

// 只能在 Storage 中使用

balances[msg.sender] = 100;

}

}

不固定长度的字节数组（bytes）

// 引用类型示例

bytes public dynamicBytes; // 动态字节数组，存储在存储区(storage)

bytes32 public fixedBytes; // 固定长度字节数组，是值类型

function example() public {

// 动态分配内存

dynamicBytes = new bytes(2);

dynamicBytes[0] = 0x12;

dynamicBytes[1] = 0x34;

// 可以改变长度

dynamicBytes.push(0x56);

}

存储位置总结：

Storage（链上存储）

状态变量

永久存储

Gas 成本高 string public storageStr; // 状态变量自动存储在 Storage

Memory（临时内存）

函数参数

函数（相关文档：数据位置修饰符）内的临时变量

函数返回值 function example(string memory param) public { string memory temp = "temp"; }

栈（Stack）

只存储引用（指针）

指向 Memory 或 Storage 的实际数据 // 栈中存储的是引用，指向实际数据 uint[] memory arr = new uint;

注意事项：

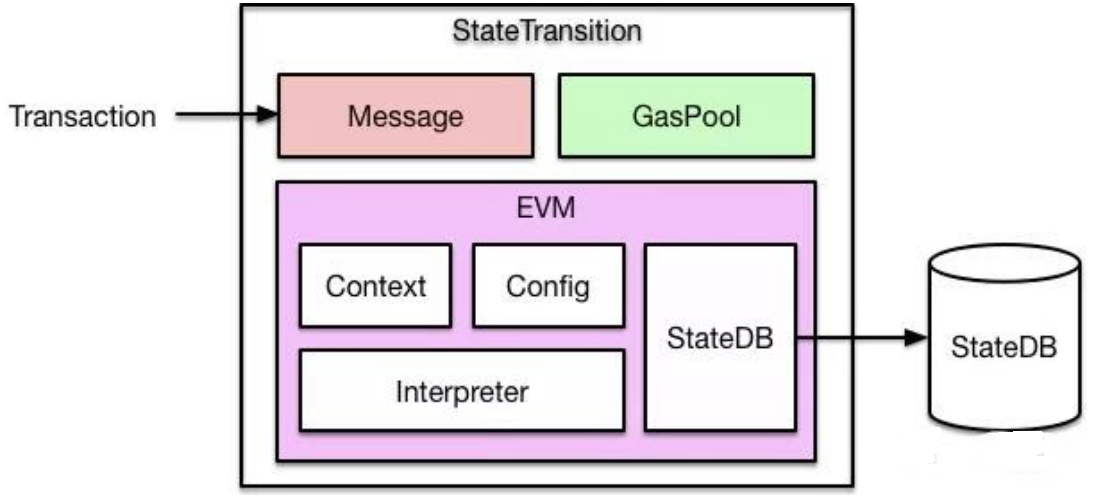
引用类型必须指定存储位置（memory/storage）

Mapping 只能用于 Storage

状态变量默认是 Storage

函数参数默认是 Memory

Storage 到 Memory 会创建副本



数据类型详解

移位运算结果的正负取决于操作符左边的数。x << y 和 x \* (2\*\*y) 是相等的，x >> y 和 x / (2\*y) 是相等的。

防止整型溢出问题，一个方法是对加法运算的结果进行判断，防止出现异常值，例如：

function add(uint256 a, uint256 b) internal pure returns (uint256){ uint256 c = a + b;

require(c >= a); // 做溢出判断，加法的结果肯定比任何一个元素大。

return c;

}

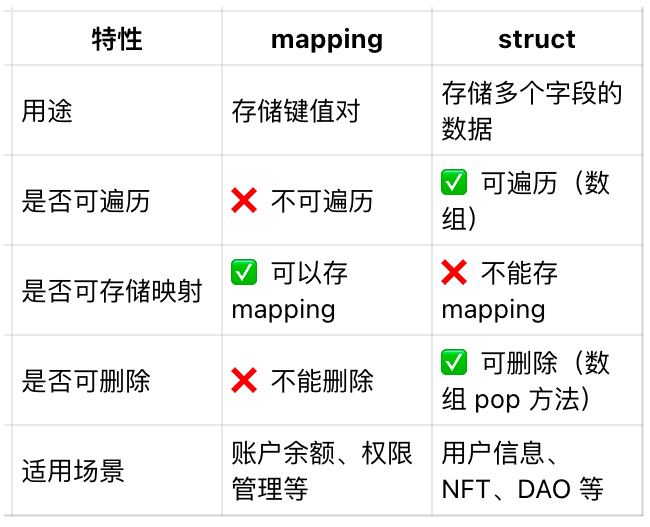
映射的特定

映射的特点

默认值： 未初始化的映射键会返回 ValueType 的默认值（例如 uint256 默认 0）。

不可遍历： Solidity 不支持遍历 mapping，只能通过 key 访问特定 value。

可修改但不可删除： 可以修改 mapping 中的值，但不能删除整个 mapping。

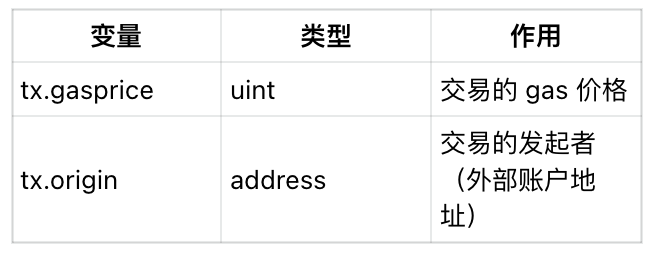


solidity全局变量

Solidity 提供了一些内置的 全局变量（Global Variables），用于访问区块链状态、交易信息等数据。这些变量不需要额外声明，可以在智能合约中直接调用。

### **2.1 区块信息（Block Information）**

[](https://private-user-images.githubusercontent.com/105115833/446413196-0dc032f5-7217-47bd-b7d6-855c256a79a4.png?jwt=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJnaXRodWIuY29tIiwiYXVkIjoicmF3LmdpdGh1YnVzZXJjb250ZW50LmNvbSIsImtleSI6ImtleTUiLCJleHAiOjE3NTY1MjI5OTksIm5iZiI6MTc1NjUyMjY5OSwicGF0aCI6Ii8xMDUxMTU4MzMvNDQ2NDEzMTk2LTBkYzAzMmY1LTcyMTctNDdiZC1iN2Q2LTg1NWMyNTZhNzlhNC5wbmc_WC1BbXotQWxnb3JpdGhtPUFXUzQtSE1BQy1TSEEyNTYmWC1BbXotQ3JlZGVudGlhbD1BS0lBVkNPRFlMU0E1M1BRSzRaQSUyRjIwMjUwODMwJTJGdXMtZWFzdC0xJTJGczMlMkZhd3M0X3JlcXVlc3QmWC1BbXotRGF0ZT0yMDI1MDgzMFQwMjU4MTlaJlgtQW16LUV4cGlyZXM9MzAwJlgtQW16LVNpZ25hdHVyZT1iMmU2MGYwMTRhZmNjOTYyZWFjNzk3ZjA2NGQ1ODVkNmIwZDhhMjI4NTIxMGM5MDMyY2MzOWIyZmZkOTU1NWIwJlgtQW16LVNpZ25lZEhlYWRlcnM9aG9zdCJ9.T2W0QD_E3wVEm5mDeW3Q-IbzQojDav9wc76asvxjd64)**2.2 交易信息（Transaction Information）**



### **2.3 消息信息（Message Information）**

### 

### **2.4 以太坊环境信息（Ethereum Environment Information）**

### 

this 关键字用于引用当前合约的实例。它可以用于：

获取当前合约的地址。

在合约内部调用自身的函数（但会创建新的交易）。

访问当前合约的 balance。 示例：使用 this 关键字

contract ThisExample {

function getContractAddress() public view returns (address) {

return address(this);

}

function getContractBalance() public view returns (uint) {

return address(this).balance;

}

}

注意事项：

this 关键字调用合约自身的函数时，会触发新的交易，因此会消耗额外的 gas。

this.balance 获取的是合约的 ETH 余额，而 msg.sender.balance 可以用于获取调用者的余额。



总结

Solidity 提供了许多 全局变量，可以访问 区块链状态、交易信息、消息信息等。

this 关键字可用于引用当前合约地址、余额或调用自身函数（会消耗额外 gas）。

数组、哈希、加密函数 也是 Solidity 重要的全局功能。

在开发合约时，应 正确使用全局变量，避免 安全隐患（如 tx.origin）。

revert的工作机制

当前函数执行被中止；

所有状态更改（包括调用之前的更改）都会被回滚；

未使用的 gas 会被退还给调用者；

如果有错误信息，会被返回给调用者（前端或调用合约可以读取这个信息）。



try-catch语句

Solidity 从 0.6.0 版本开始引入了 try-catch 语句，用于处理外部函数调用和合约创建中的错误。这是 Solidity 中错误处理的重要机制之一。

适用场景

try-catch 只能用于以下两种情况：

外部函数调用（使用 address 调用或合约实例调用）

合约创建（使用 new 关键字）

错误类型详解

Error(string memory reason)

对应 revert("description") 或 require(false, "description")

可以获取错误描述字符串

2. Panic(uint errorCode)

对应 Solidity 的 "panic" 错误（类似断言失败）

常见错误码：

0x01: 断言失败

0x11: 算术运算溢出

0x12: 除以零

0x21: 无效的数组索引

0x31: 分配过多内存

0x32: 调用未初始化的内部函数类型变量

默认的 catch 块

捕获所有其他类型的错误

包含原始错误数据（bytes 类型）

自定义修饰符

自定义修饰符（Modifier）用于在函数执行前或执行后添加额外的检查或逻辑。它可以复用代码，并且常用于权限控制或状态检查。

语法

modifier 修饰符名 {

// 前置逻辑

\_; // 执行函数体

// 后置逻辑

}

接口的特点

接口只能声明函数，不能实现函数。

接口中的函数必须标记为 external。

接口不能定义状态变量或构造函数。

接口可以继承其他接口。

OpenZeppelin 是一个广泛使用的 Solidity 库，提供了许多经过审计的智能合约模板，如 ERC20、ERC721、Ownable 等。使用 OpenZeppelin 可以大大减少开发时间并提高合约的安全性。

1. 合约收款方式

payable修饰符

function funcName() public payable() {

}

🔹 receive() 函数

✅ 用途

当合约收到纯 ETH 转账（例如 address(this).transfer() 或 address(this).send()）且没有调用数据（data为空）时，会调用 receive() 函数。

✅ 语法

receive() external payable { // 收款逻辑 }

external：只能被外部调用。

payable：允许接收 ETH。

不能有参数，也不能返回值。

每个合约只能有一个 receive() 函数。

✅ 使用场景

contract MyContract {

event Received(address sender, uint amount);

receive() external payable {

emit Received(msg.sender, msg.value);

}

}

fallback() 函数

✅ 用途

当调用合约函数时，找不到对应函数签名

或者调用时带有数据，但合约中没有 receive() 函数可调用

会触发 fallback() 函数。

✅ 语法（两种）

1. 允许收款：

fallback() external payable { // fallback 收款逻辑 }

2. 不收款，仅响应错误调用：

fallback() external { // fallback 非 payable，不能接收 ETH }

``

✅ 使用场景

contract MyContract {

event FallbackCalled(address sender, uint amount, bytes data);

fallback() external payable {

emit FallbackCalled(msg.sender, msg.value, msg.data);

}

}

📊 receive vs fallback 对比总结

特性 receive() fallback()

是否能接收 ETH 是（必须是 payable） 可选（payable 或不写）

是否接收 data 否（data 必须为空） 是（data 非空或无函数匹配）

是否必须存在 否（可选） 否（可选）

常见触发条件 纯 ETH 转账，无数据 错误调用或带 data 转账

实战建议：

如果你只是想接收纯 ETH，可以只写 receive() payable。

如果你想对任何未知调用做处理（比如 proxy、日志记录），就用 fallback()。

如果两者都写了，Solidity 会优先调用 receive()，只在 data 不为空时才会调用 fallback()。

📥 2. 查看合约收到的余额

address(this).balance

返回当前合约地址的 ETH 余额（单位为 wei）

💸 3. 合约向外转账的三种方式

转给 外部账户、合约账户

✅ address.transfer

payable(msg.sender).transfer(1 ether);

固定 2300 gas，失败自动 revert

✅ address.send

bool success = payable(msg.sender).send(1 ether);

require(success, "Send failed");

同样只提供 2300 gas，但需要手动检查返回值

✅ call（推荐）

(bool success, ) = payable(msg.sender).call{value: 1 ether}("");

require(success, "Call failed");

可调 gas，兼容新版本，官方推荐方式

🛡️ 4. 安全建议

使用 call 替代 transfer / send，避免 Out of Gas 错误

使用 ReentrancyGuard 防止重入攻击

避免在 receive() 中执行复杂逻辑

重入攻击

当合约调用外部地址（如 call 转账）时，如果该地址是一个合约，它可以在未完成前一次调用前，反复调用回原合约的函数，造成重复提现等安全问题。

攻击演示：易受攻击的合约

// VulnerableVault.sol

contract VulnerableVault {

mapping(address => uint) public balances;

function deposit() external payable {

balances[msg.sender] += msg.value;

}

function withdraw() external {

require(balances[msg.sender] > 0, "No balance");

// 发送 ETH（外部调用，容易被攻击者重入）

(bool success, ) = msg.sender.call{value: balances[msg.sender]}("");

require(success, "Transfer failed");

// 更新余额（放在调用后，导致漏洞）

balances[msg.sender] = 0;

}

}

攻击者合约：

// Attacker.sol

contract Attacker {

VulnerableVault public target;

constructor(address \_target) {

target = VulnerableVault(\_target);

}

// 回调函数，趁机再次提取

receive() external payable {

if (address(target).balance > 1 ether) {

target.withdraw();

}

}

function attack() external payable {

require(msg.value >= 1 ether, "Need 1 ETH");

target.deposit{value: 1 ether}();

target.withdraw();

}

}

流程说明

用户向 VulnerableVault 合约 deposit() 存入 1 ETH

攻击者调用 withdraw()，触发合约转账 call

攻击者合约在 receive() 中再次调用 withdraw()

因为合约尚未更新 balances，攻击者可多次提取

合约余额被掏空，攻击成功

如何防止重入攻击？

使用“检查-效果-交互”模式：

function withdraw() external {

uint amount = balances[msg.sender];

require(amount > 0, "No balance");

// 先更新状态

balances[msg.sender] = 0;

// 再转账（外部调用）

(bool success, ) = msg.sender.call{value: amount}("");

require(success, "Transfer failed");

}

使用 ReentrancyGuard（OpenZeppelin 提供）

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureVault is ReentrancyGuard {

mapping(address => uint) public balances;

function deposit() external payable {

balances[msg.sender] += msg.value;

}

function withdraw() external nonReentrant {

uint amount = balances[msg.sender];

require(amount > 0, "No balance");

balances[msg.sender] = 0;

(bool success, ) = msg.sender.call{value: amount}("");

require(success, "Transfer failed");

}

}

防御措施 说明

状态更新在前 防止多次调用利用旧状态

使用 ReentrancyGuard 简洁防御，适合大多数场景

限制外部合约调用 检查 tx.origin 或设白名单

什么是event？

event 是 Solidity 提供的一种日志机制，可以在链上记录特定行为（比如转账、投票、交易等）。它们被写入到交易日志（transaction logs）中，可以被客户端或监听程序（如 DApp）读取和响应。

特性 描述

用途 记录关键事件（转账、投票等）

存储位置 区块链的交易日志（不计入存储 gas）

查询效率 indexed 参数可提升筛选效率

前端 可监听这些事件进行 UI 更新或通知

以太坊虚拟机EVM

1. EVM 概述

EVM 是什么？

以太坊虚拟机（EVM）是一个运行智能合约的去中心化计算环境，负责处理智能合约的部署和执行。它是以太坊协议的一部分，并且独立于外部系统。

EVM 的特点：

独立沙盒环境：智能合约在 EVM 中执行，没有网络、文件系统或其他进程访问。

图灵完备的状态机：EVM 是一个“准”图灵完备的状态机，但受限于 Gas 消耗，以防止无限循环执行的问题。

2. EVM 的数据结构

基于栈的架构：EVM 使用栈来存储数据，最大支持 1024 个栈元素，每个元素是 256 位的字（byte），所有计算都在栈上完成。

三类存储结构：

程序代码存储区（ROM）：不可变，存储智能合约的字节码。

内存（Memory）：可变，临时存储执行期间的数据，随调用结束而清除。

存储（Storage）：持久化存储，每个智能合约都有一个唯一的存储区，存储合约状态。

3. 账户类型

外部账户（EOA）：由用户控制，通过私钥签名交易。

合约账户：存储智能合约代码，由合约代码控制其行为。

地址空间共享：外部账户和合约账户共享相同的地址空间。外部账户通过公钥生成地址，合约账户通过 nonce 和创建者的地址生成地址。

4. 交易与合约调用

交易的定义：交易是一个账户与另一个账户之间的信息交换，可以包含二进制数据（payload）或以太币（Ether）。

交易执行的过程：

如果目标账户为合约，EVM 会执行合约代码，并将输入数据作为参数传递。

如果目标账户不存在，交易将创建一个新合约，将代码部署到该合约地址。

合约调用与消息传递：EVM 内部使用消息调用来实现合约间的通信，类似于外部交易。每次调用会携带 Gas 限制，以避免 Gas 消耗殆尽导致调用失败。

5. Gas 机制

Gas 的作用：

防止计算资源滥用，确保交易的执行不会陷入无限循环。

交易发起者需支付 Gas，执行时消耗的 Gas 根据指令复杂度确定。

Gas 价格和限制：

发起交易时设定 Gas 价格和 Gas 上限。若 Gas 消耗超出设定的上限，交易会触发 out-of-gas 异常，导致状态回滚。

Gas 费用计算：每个指令有对应的 Gas 消耗，复杂度越高，消耗越大。

6. 存储、内存与栈

存储（Storage）：

每个合约都有一个持久化的存储区，是 256 位的 key-value 键值对。

存储操作成本高，读取和写入的 Gas 消耗较大。

内存（Memory）：

在每次调用中重新初始化的可变区域，用于临时存储。

按字节地址访问，内存的扩展需要支付额外的 Gas，扩展成本随内存增长而增加。

栈（Stack）：

EVM 使用栈来管理数据，最多容纳 1024 个 256 位的字。所有运算都在栈上进行，栈顶元素可以交换或复制。

7. EVM 的指令集

基本操作：

EVM 支持 256 位的算术、位运算、逻辑运算、条件跳转等基本指令。

控制流：合约可以执行条件跳转（JUMP）或非条件跳转，EVM 会校验跳转目标的合法性。

区块属性访问：

EVM 指令集允许访问当前区块的属性，如区块号和时间戳，这些信息常用于智能合约逻辑中。

8. 消息调用与合约之间的交互

消息调用：智能合约之间的交互是通过消息调用完成的，调用时可以携带数据和以太币。

深度限制：EVM 限制了消息调用的深度为 1024 层，以避免递归调用陷入无限循环。

调用合约的数据传递：调用数据存储在 calldata 中，返回数据存储在主存中。

9. 代理调用与库的使用

代理调用（delegatecall）：允许在调用者的上下文中执行被调用合约的代码，msg.sender 和 msg.value 不会改变。这种方式允许动态加载外部合约的代码，常用于合约升级和库的实现。

合约调用的四种方式

CALL：最常用的调用方式，修改被调用者的存储。

CALLCODE：修改调用者的存储，而不是被调用者的存储（不再推荐使用）。

DELEGATECALL：修复了 CALLCODE 的问题，调用者保持其上下文环境，msg.sender 不变。

STATICCALL：用于调用 view 或 pure 类型的函数，不允许修改状态。

10. 日志与事件

日志（Logs）：EVM 支持日志记录，日志可以从区块链外部查询，但合约内部无法访问。日志用于智能合约的事件通知。

事件：合约通过 emit 关键字触发事件，事件会被记录在区块链的日志中，用于通知外部监听者。

11. 合约创建与销毁

合约创建：通过特殊指令创建新的合约，执行合约代码后，将结果作为合约的代码存储在区块链上。

合约销毁（selfdestruct）：合约通过 selfdestruct 操作将其代码和存储从区块链上移除，同时将剩余的以太币发送到指定的地址。

常见面试问题：

1. EVM 与传统虚拟机的区别是什么？

回答：EVM 是为去中心化区块链环境设计的虚拟机，不依赖外部系统，独立处理合约执行，防止恶意操作导致无限循环。

2. 如何在 EVM 中优化 Gas 消耗？

回答：通过减少存储写操作、批量操作、合理的内存扩展和避免复杂的循环来优化 Gas 消耗。

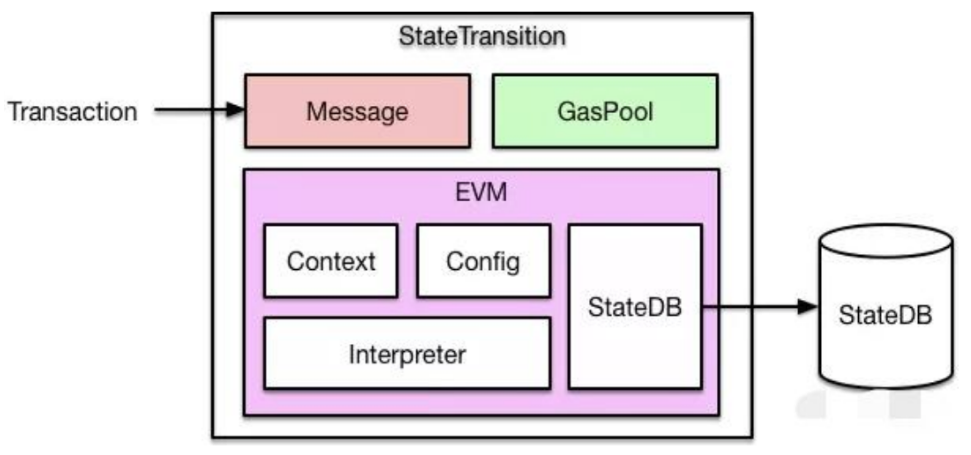
3. EVM 如何处理合约之间的调用？

回答：EVM 通过消息调用在合约之间进行交互，每个调用都会分配独立的内存空间和 Gas 限制。

EVM工作原理

EVM 解释执行流程

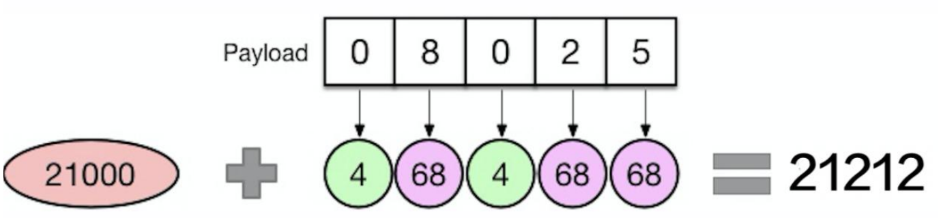
以下是以太坊虚拟机业务流程，如图所示：



输入一笔交易，内部会转换成一个 Message 对象，传入 EVM 执行。如果是一笔普通转账交易，那么直接修改 StateDB 中对应的账户余额即可。如果是智能合约的创建或者调用，则通过 EVM 中的解释器加载和执行字节码，执行过程中可能会查询或者修改 StateDB。

\*\*固定油费（Intrinsic Gas） \*\*

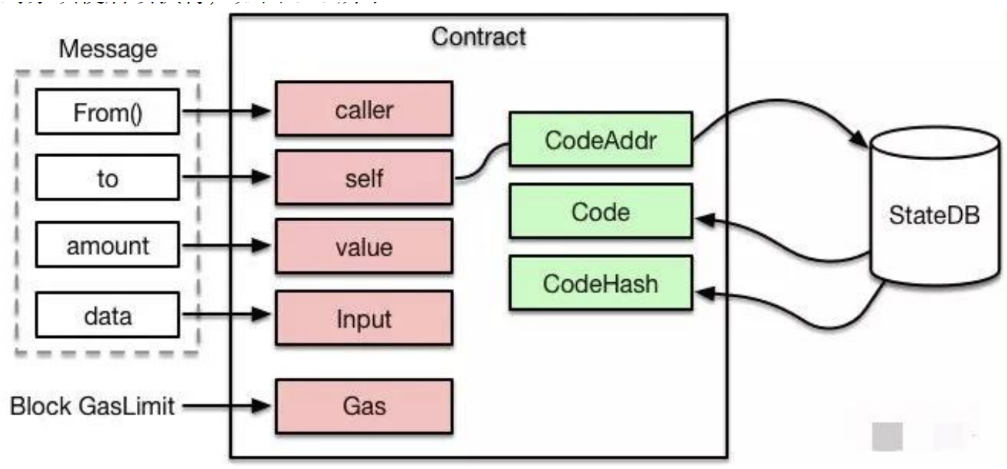
每笔交易过来，不管什么情况先需要收取一笔固定油费，计算方法如图所示：如果交易不带额外数据（Payload），比如普通转账，那么需要收取 21000 的油费。



如果交易携带额外数据，那么这部分数据也是需要收费的，具体来说是按字节收费：字节为 0 的收 4 块，字节不为 0 收 68 块，所以很多做合约优化操作目的就是减少数据中不为 0 的字节数量，从而降低油费 gas 消耗。

\*\*生成 Contract 对象 \*\*

交易会被转换成一个 Message 对象传入 EVM，而 EVM 则会根据 Message 生成一个 Contract 对象以便后续执行，如图所示：



可以看到，Contract 中会根据合约地址，从 StateDB 中加载对应的代码，后面就可以送入解释器执行了。另外，执行合约能够消耗的油费有一个上限，就是节点配置的每个区块能够容纳的 GasLimit

送入解释器执行

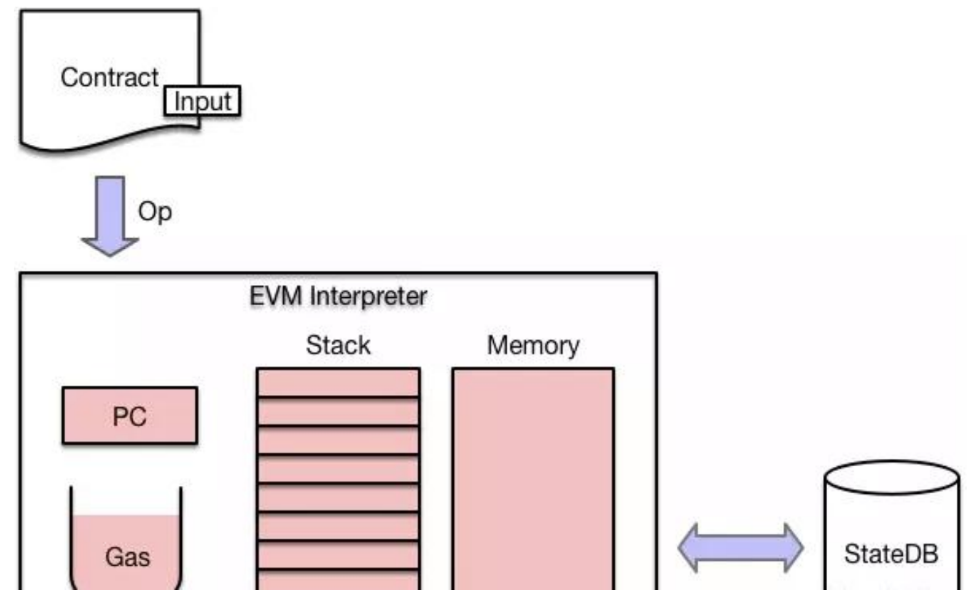
代码跟输入都有了，就可以送入解释器执行了。EVM 是基于栈的虚拟机，解释器中需要操作四大组件，如图所示：

PC：类似于 CPU 中的 PC 寄存器，指向当前执行的指令

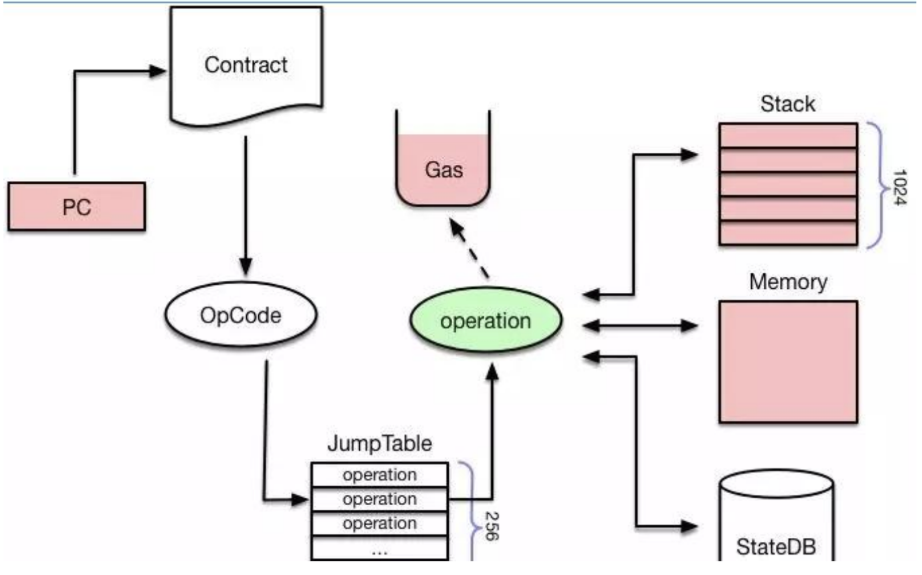
Stack：执行堆栈，位宽为 256 bits，最大深度为 1024

Memory：内存空间

Gas：油费池，耗光邮费则交易执行失败



详细解释执行流程，如图所示：



EVM 的每条指令称为一个 OpCode，占用一个字节，所以指令集最多不超过 256，比如图所示，该图示就是一个示例（PUSH1=0x60, MSTORE=0x52）：



首先 PC 会从合约代码中读取一个 OpCode，然后从一个 JumpTable 中检索出对应的 operation，也就是与其相关联的函数集合。接下来会计算该操作需要消耗的油费，如果油费耗光则执行失败，返回 ErrOutOfGas 错误。如果油费充足，则调用 execute()执行该指令，根据指令类型的不同，会分别对 Stack、Memory 或者 StateDB 进行读写操作。

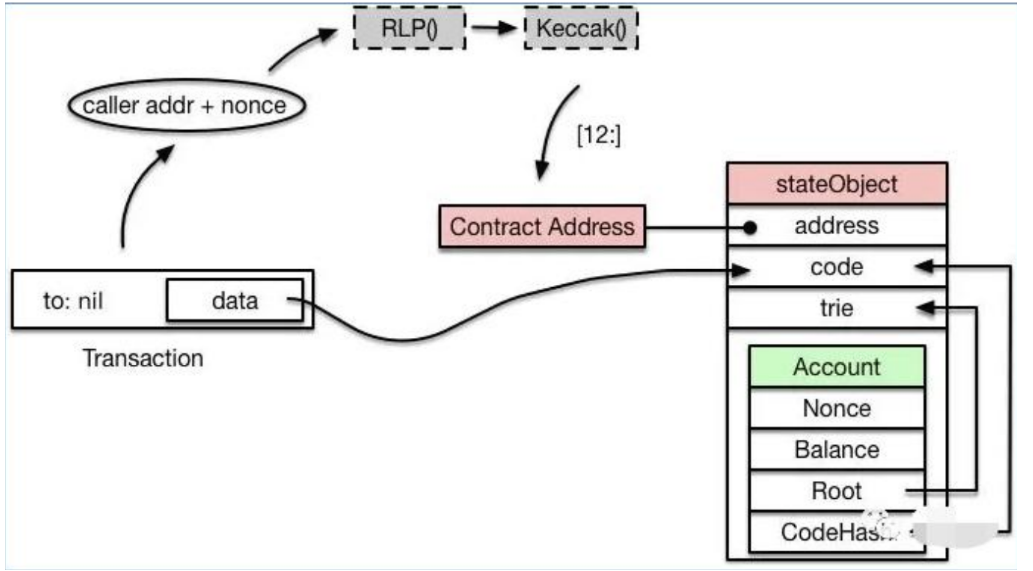
创建合约流程

上面的内容讲述了合约解释执行流程，那么创建合约的流程是怎么样的呢？

如果某一笔交易的 to 地址为 nil，则表明该交易是用于创建智能合约的。

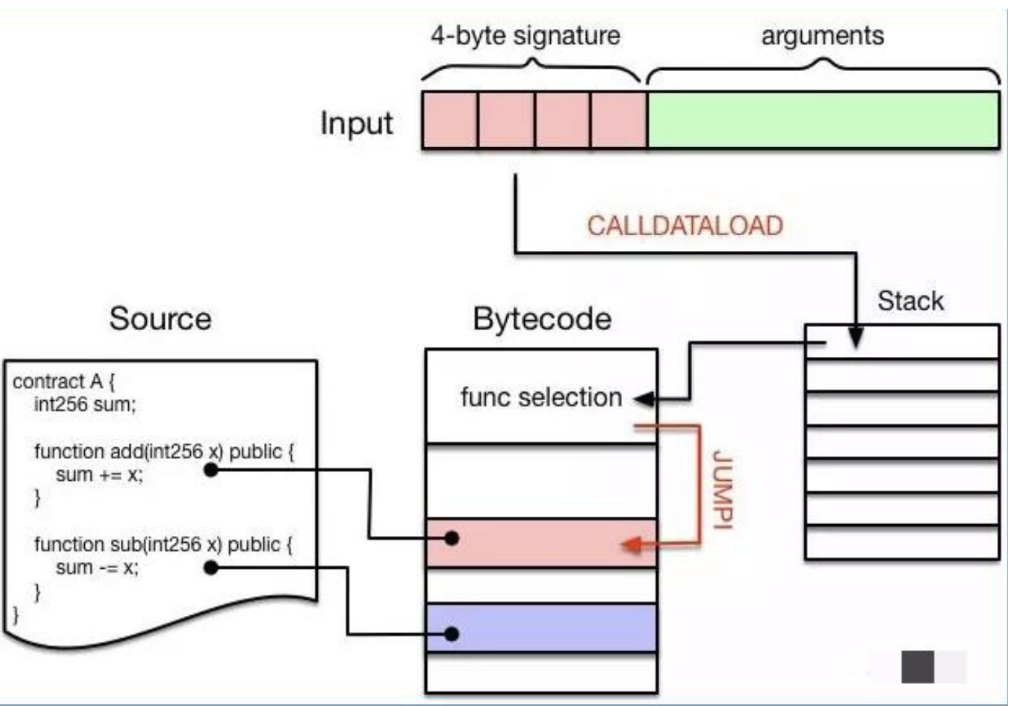
首先需要创建合约地址，采用下面的计算公式：Keccak(RLP(call\_addr, nonce))[:12]。也就是说，对交易发起人的地址和 nonce 进行 RLP 编码，再算出 Keccak 哈希值，取后 20 个字节作为该合约的地址。

下一步就是根据合约地址创建对应的 stateObject，然后存储交易中包含的合约代码。该合约的所有状态变化会存储在一个 storage trie 中，最终以 Key-Value 的形式存储到 StateDB 中。代码一经存储则无法改变，而 storage trie 中的内容则是可以通过调用合约进行修改的，比如通过 SSTORE 指令。创建流程如图所示：



调用合约流程

那么，EVM 又是怎么知道交易想调用的是合约里的哪个函数呢？前面提到跟合约代码一起送到解释器里的还有一个 Input，而这个 Input 数据是由交易提供的。调用流程如图所示：



Input 数据通常分为两个部分：

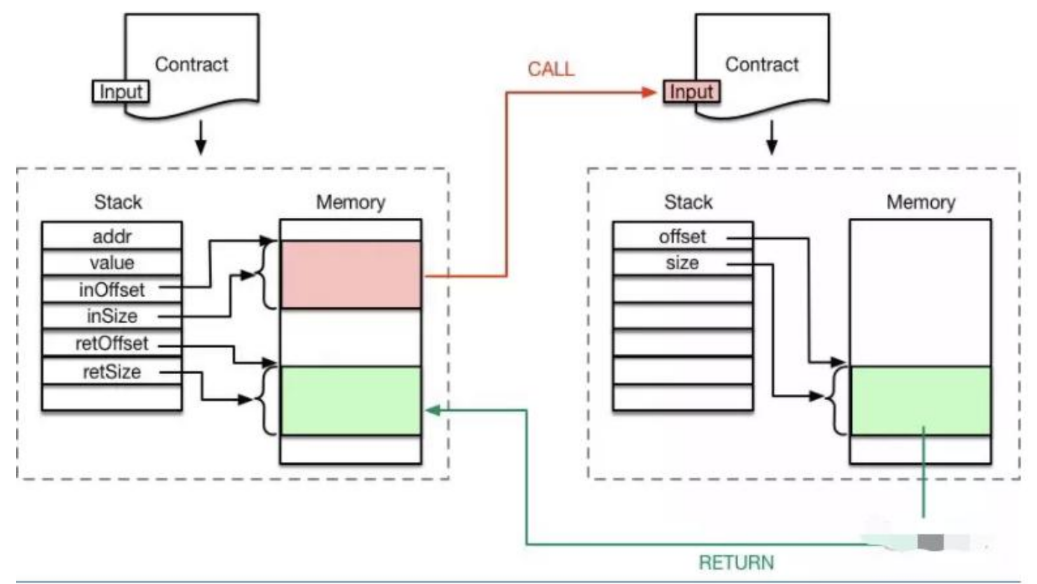
前面 4 个字节被称为“4-byte signature”，是某个函数签名的 Keccak 哈希值的前 4 个字节，作为该函数的唯一标识。

后面跟的就是调用该函数需要提供的参数了，长度不定。 举个例子：在部署完 A 合约后，调用 add(1)对应的 Input 数据是：

0x87db03b70000000000000000000000000000000000000000000000000000000000000001

合约调用合约

以最简单的 CALL 为例，调用流程如图所示：



可以看到，调用者把调用参数存储在内存中，然后执行 CALL 指令。

CALL 指令执行时会创建新的 Contract 对象，并以内存中的调用参数作为其 Input。解释器会为新合约的执行创建新的 Stack 和 Memory，从而不会破环原合约的执行环境。新合约执行完成后，通过 RETURN 指令把执行结果写入之前指定的内存地址，然后原合约继续向后执行。

\*\*合约的四种调用方式 \*\*

在中大型的项目中，操作者不可能在一个智能合约中实现所有的功能，而且这样也不利于分工合作。一般情况下，操作者会把代码按功能划分到不同的库或者合约中，然后提供接口互相调用。

在 Solidity 中，如果只是为了代码复用，可以把公共代码抽出来，部署到一个 library 中，后面就可以像调用 C 库、Java 库一样使用了。但是 library 中不允许定义任何 storage 类型的变量，这就意味着 library 不能修改合约的状态。如果需要修改合约状态，操作者需要部署一个新的合约，这就涉及到合约调用合约的情况。

合约调用合约有下面 4 种方式：

CALL

CALLCODE

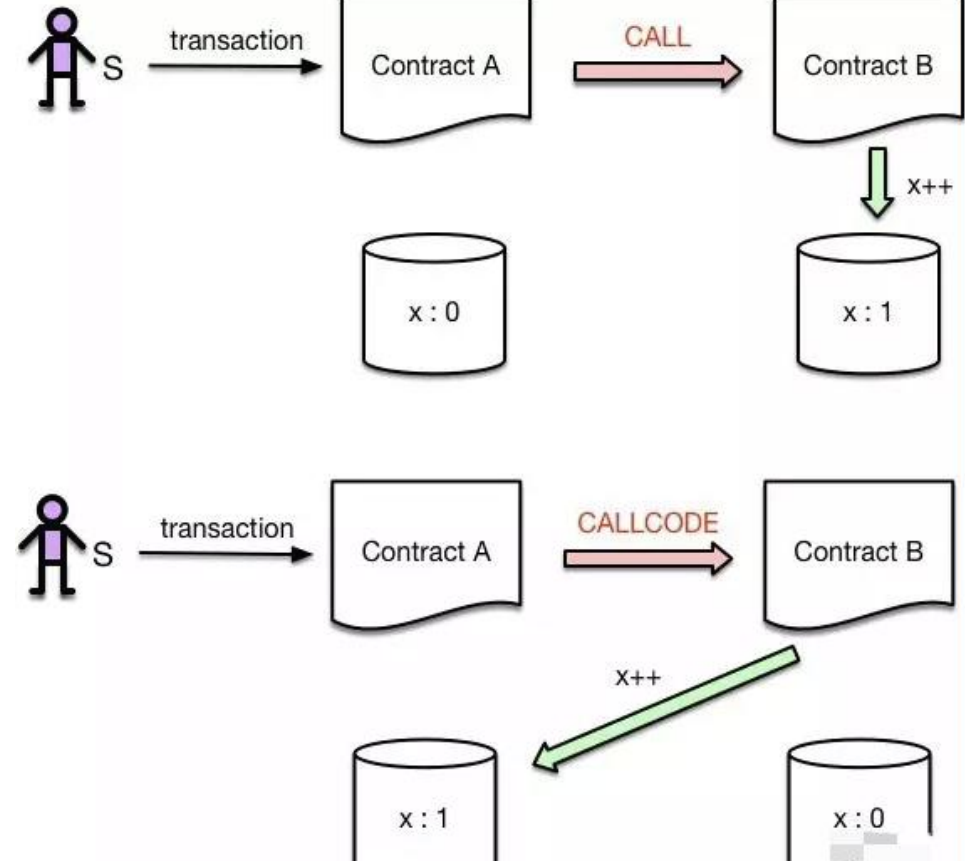
DELEGATECALL

STATICCALL

\*\*CALL vs. CALLCODE \*\*

CALL 和 CALLCODE 的区别在于：代码执行的上下文环境不同。

具体来说，CALL 修改的是被调用者的 storage，而 CALLCODE 修改的是调用者的 storage。二者的对比如图所示：

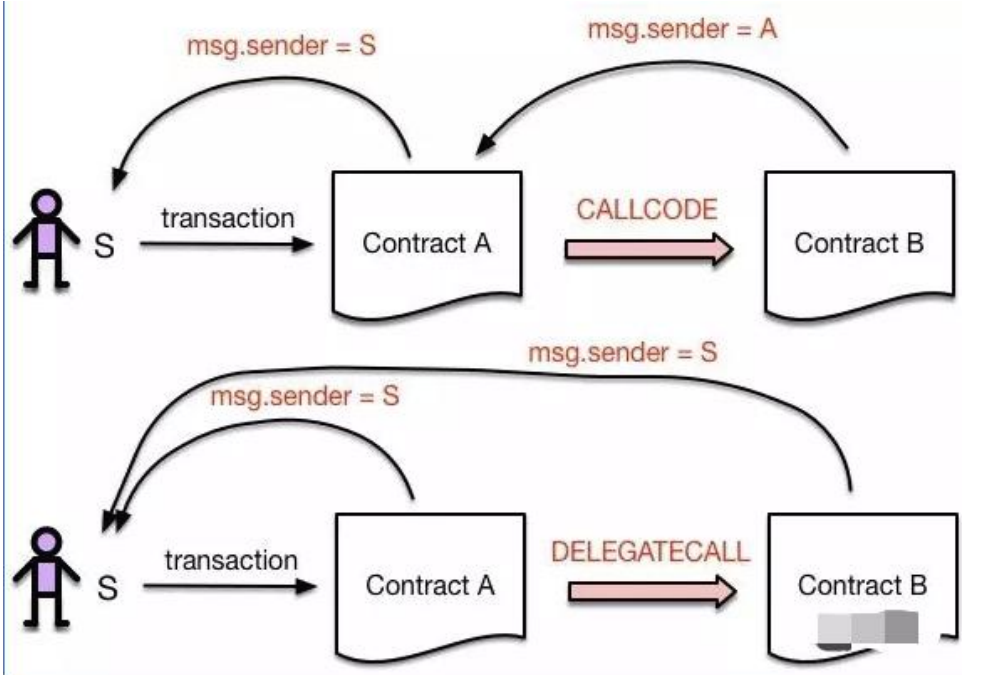


\*\*CALLCODE vs. DELEGATECALL \*\*

实际上，可以认为 DELEGATECALL 是 CALLCODE 的一个 bugfix 版本，官方已经不建议使用 CALLCODE 了。

CALLCODE 和 DELEGATECALL 的区别在于：msg.sender 不同。

具体来说，DELEGATECALL 会一直使用原始调用者的地址，而 CALLCODE 不会。二者的对比如图所示：



\*\*STATICCALL \*\*

STATICCALL\*\* \*\*放在这里似乎有滥竽充数之嫌，因为目前 Solidity 中并没有一个 lowlevel API 可以直接调用它，仅仅是计划将来在编译器层面把调用 view 和 pure 类型的函数编译成 STATICCALL 指令。

view 类型的函数表明其不能修改状态变量，而 pure 类型的函数则更加严格，连读取状态变量都不允许。

目前是在编译阶段来检查这一点的，如果不符合规定则会出现编译错误。如果将来换成 STATICCALL 指令，就可以完全在运行时阶段来保证这一点了，操作者可能会看到一个执行失败的交易。

Gas

gas 是以太坊协议中的计量单位，用它来计量在以太坊区块链上执行具体操作所要花费的计算量和存储资源。与比特币协议中仅仅以交易数据的字节大小来计算交易费不同，以太坊协议中计算交易费时需要计量由交易和智能合约代码执行所引发的所有计算步骤。

由交易和合约代码触发的每个操作都会消耗固定数量的 gas，以下是黄皮书中的几个例子：

相加两个数值需要消耗 3 gas。

计算 Keccak-256 哈希需要消耗 30 gas，每 256 位输入数据额外消耗 6 gas。

发送一个交易需要消耗 21000 gas。

gas 是以太坊中极其重要的组成部分，它扮演了双重角色：作为以太坊中浮动的价格和矿工奖励之间的缓冲，以及对抗拒绝服务攻击的防范措施。为了防止网络中意外的或者恶意的无限循环以及其他形式的计算浪费，每个交易的创建者都需要设定一个限制来表明他们愿意为交易执行所付出的计算量。gas 系统借此来降低攻击者们发送“垃圾”交易的意愿，因为他们必须为他们所消耗的计算量、带宽和存储资源付出代价。

执行阶段的 gas 计量

当需要 EVM 进行计算来完成交易时，第一个 EVM 实例将获得与交易中所指定的 gas 上限相同数量的可用 gas。每个操作码的执行都会消耗 gas，于是这个 EVM 的 gas 供给就会在程序执行过程中相应减少。在每个操作之前，EVM 会检查是否还有足够的 gas 来支付操作的执行。如果已经没有足够的 gas，则执行会被中止，交易也会被撤销。

如果 EVM 中的执行成功结束，没有出现 gas 不足的情况，那么执行中实际消耗的 gas 费用就会被作为交易费支付给矿工。交易费将会基于交易中指定的 gas 价格折算为以太币：

矿工费=实际消耗的 gas×gas 价格

由交易提供的剩余 gas 将返还给发送者，同样将会基于交易中指定的 gas 价格折算为以太币：

剩余 gas=gas 上限-实际 gas 消耗

返还的以太币=剩余 gas×gas 价格

如果一个交易在执行过程中“gas 不足”，操作会被立即中止，并产生“out of gas”异常。

交易也会被撤销，所有对状态的修改都会回滚。

但尽管交易没有成功，发送者仍然需要支付交易费，因为矿工们已经为到发生错误时点的操作付出了计算量，他们理应从这些付出中获得报酬。

gas 计量原则

EVM 中的不同操作所花费的 gas 是被仔细选择确定的，以此来在攻击中保护以太坊区块链。操作中包含了越多计算量的操作码其 gas 消耗会越高。比如，执行 SHA3 函数的代价（消耗 30 gas）大概是 ADD 操作（消耗 3 gas）的 10 倍。更重要的是，某些操作，比如 EXP，需要基于操作数的大小额外支付 gas。使用 EVM 的内存和向合约的链上存储保存数据同样是要消耗 gas 的。

真实世界中的资源消耗必须与 gas 消耗相匹配的重要性在 2016 年获得了印证。当时，一个攻击者发现并利用了 gas 消耗与实际资源消耗的不匹配，他生成了很多计算量极其巨大的交易，并使以太坊主网堵塞到几乎停止运转。这个问题最终通过一个硬分叉（代号“Tangerine Whistle”）调整了 gas 消耗才得以解决。

gas 消耗和 gas 价格

gas 消耗是对 EVM 所进行的计算和使用的存储的一个计量，与此相应，gas 本身也有一个用以太币来计量的价格。在发送一个交易的时候，发送者需要指定一个他们愿意为每单位 gas 支付的 gas 价格（以以太币为单位），来使市场可以决定这些以太币价格与计算操作的消耗（以 gas 为单位）之间的关系：

交易费=总 gas 消耗 ×gas 价格（用以太币作为单位）

以太坊网络中的矿工们在构造新区块的时候可以从处于等待状态的交易中来选取那些 gas 价格较高的交易。为交易提供较高的 gas 价格可以使矿工们更容易选择它们，从而使它们更快地得到确认。

在实际操作的时候，交易的发送者需要指定一个 gas 上限，这个上限应该高于或等于他们希望被使用的 gas 数量。如果 gas 上限高于实际消耗的 gas 数量，发送者会获得剩余数量的返还，因为矿工们应该仅从他们实际付出的工作中获得报酬。

清晰地理解 gas 消耗和 gas 价格的区别非常重要，这里再次说明：

gas 消耗是执行特定操作所需要的单位 gas 的数量。

gas 价格是在发送交易到以太坊网络时指定的希望为每单位 gas 所支付的以太币数量。尽管 gas 是有价格的，但它既不能“持有”也不能“花费”。gas 仅在 EVM 内作为对计算工作量的计量而存在。向发送者收取的交易费是用以太币来计算的，它首先会被换算为供 EVM 计量的 gas，而后会以以太币为单位结算为交易费来支付给矿工们。

负的 gas 消耗

以太坊鼓励删除使用过的存储变量，这将会计入在合约执行过程中返还的对 gas 使用的消耗。

EVM 中有两个操作带有负的 gas 消耗：

删除一个合约（SELFDESTRUCT）会返还 24000 gas。

将一个非零值的存储地址设置为 0（SSTORE[x]=0）会返还 15000 gas。为了避免通过返还机制牟利，交易的最大 gas 返还数量被设定为交易中总共消耗的 gas 数量的一半（向下舍入）。

区块的 gas 限制

区块的 gas 限制指的是一个区块中的所有交易总共能消耗的最大 gas 数量，它也限定了一个区块中能包含多少交易。

例如，假定有 5 个交易，它们的 gas 上限分别为 30000、30000、40000、50000 和 50000。

如果区块的 gas 限制是 180000，那么任意四个交易都可以包含到一个区块中，第 5 个交易则需要等待后续区块。就像先前讨论过的那样，矿工们决定哪些交易会被包含到一个新区块中。不同的矿工大概会选择不同的组合，这主要是因为他们从网络中接收到这些交易的顺序是不同的。

如果一个矿工尝试包含一个 gas 消耗超过区块 gas 限制的交易，那么这个区块将会被网络拒绝。大多数以太坊客户端会停止处理这个交易，并给出一个类似“transaction exceeds block gas limit”的警告。目前以太坊主网的区块 gas 限制是 800 万，这意味着每个区块大概可以包含 380 个基础交易（每个交易消耗 21000 gas）。

区块的 gas 限制是多少

区块的 gas 限制是由网络中的矿工们共同决定的。希望在以太坊网络中挖矿的人可以使用一个挖矿程序，比如 Ethminer，它将会连接到 geth 或者 Parity 客户端。以太坊协议有一个内置的机制允许矿工们对后续区块的 gas 限制进行投票以改变其容量。某个矿工可以提议对区块的 gas 限制做增减最多 1/1024（0.0976%）的调整。投票的结果将会决定一个基于当时网络需求的可变的区块大小。这个机制与默认的挖矿策略是绑定的。在默认的挖矿策略中，矿工们会对 gas 限制进行投票，这个限制至少为 470 万，但实际上是以最近的平均区块 gas 消耗（基于 1024 个区块计算的指数移动平均值）的 150% 为目标的一个数值。

为什么区块链相关操作一定是异步的？  
因为这些操作通常要：

与区块链节点通讯

等待网络确认（像转账、部署合约）

查询远程链上的状态

这些都是“耗时的任务”，不能卡住主线程，只能异步处理。

概念 举例 说明

同步 console.log() 马上执行完毕

异步 await ethers.getSigners() 网络请求、需要时间

不加 await 得到 Promise 无法直接拿到值

async 函数里才能用 await async function main() 必须加上 async

可升级合约原理

一、代理的必要性与作用

在区块链世界中，合约一旦部署便不可更改，这是其不可变性的特性。然而，在实际应用里，修复 bug、打补丁以及处理安全漏洞等需求无法忽视。代理模式的出现解决了这一矛盾，让合约升级成为可能，甚至成为了行业内事实上的标准。

代理的工作机制

（一）相关术语

代理合约：作为代理存在的合约，负责将所有调用委托给它所代理的合约，同时也是存储层，负责存储状态变量。

实现合约：即需要进行升级或修补的合约，是代理合约所代理的对象，也被称作逻辑层。

工作流程

代理合约会将实现合约的地址作为状态变量进行存储。用户不会直接向实现合约发送调用，而是通过代理合约来间接调用。代理合约使用 delegatecall 函数将调用委托给实现合约，并将实现合约返回的数据回传给调用者，若出现错误则进行回退。

delegatecall. 自己的上下文，逻辑合约的函数

User ----------> Proxy -----------> Implementation

(storage layer) (logic layer)

升级机制

合约升级的本质是通过授权改变代理合约中存储的实现合约地址变量，使其指向新部署的、升级后的实现合约。这样，代理合约就会将后续的调用委托给新的实现合约，而旧合约依然会存在于区块链上。

upgrade call

Admin -----------> Proxy -----> Implementation\_v1

|

--------> Implementation\_v2

代理合约面临的问题及解决方案

（一）代理合约与实现合约的存储碰撞

若直接在代理合约中声明 address implementation，会导致与实现合约的存储发生冲突，即实现合约中的多个变量在存储槽中重叠。

|Proxy |Implementation |

|------------------------|---------------|

|address implementation |address var1 | <- 碰撞!

| |mapping var2 |

| |uint256 var3 |

| |... |

解决方案是选择一个伪随机槽，将 implementation 的地址写入该槽。根据 EIP-1967，可通过以下方式计算槽位置：

bytes32 private constant implementationPosition = bytes32(uint256(

keccak256('eip1967.proxy.implementation ')) - 1

));

每次访问或修改实现合约地址时，都会对该槽进行读写操作。

（二）不同实现合约间的存储碰撞

由于代理合约是存储层，升级到新的实现合约时，若添加新的状态变量，必须将其附加到存储布局中，新合约应扩展而非修改存储布局，否则会发生碰撞。

错误示例 ❌

|ImplementationV1 |ImplementationV2|

|-----------------|----------------|

|address foo |address baz | <- 碰撞!

|mapping bar |address foo |

| |mapping bar |

| |... |

正确示例 ✅

|ImplementationV1 |ImplementationV2|

|-----------------|----------------|

|address foo |address foo |

|mapping bar |mapping bar |

| |address baz | <- 扩展

| |... |

初始化构造函数代码问题

因为代理合约是存储层，初始化逻辑应在代理内部运行。但不能代理调用实现合约的构造函数，因为构造函数代码仅在部署时运行一次，不属于运行时字节码。

解决办法是将构造函数代码转移到实现合约的 initializer 函数中，该函数需确保只被调用一次。

import "@openzeppelin/contracts - upgradeable/proxy/utils/Initializable.sol ";

contract MyContract is Initializable {

// `initializer` modifier makes sure it runs only once

function initialize(

address arg1,

uint256 arg2,

bytes memory arg3

) public payable initializer {

// "constructor "code...

}

}

代理合约与实现合约的函数冲突

代理合约也需要有自己的函数，如 upgradeTo(address impl) 用于决定是否修改实现合约地址。若实现合约有同名函数，就需要一种机制来决定是否将调用委托给实现合约。

OpenZeppelin 采用的方式是通过代理合约的管理员或所有者地址判断。若管理员（msg.sender == admin）发起调用，代理合约将自行执行该函数；否则，将调用委托给实现合约。

msg.sender ->| proxy `owner` 调用 | 其他人调用

----------------|------------------------------------

`owner()` | proxy.owner() | erc20.owner()

`upgradeTo(..)` | proxy.upgradeTo(..) | reverts

`transfer(..)` | reverts | erc20.transfer(..)

透明代理和UUPS代理

具体参考04透明代理.md 在透明代理模式（EIP-1967）中，升级逻辑位于代理合约内，即升级操作由代理合约处理。需调用 upgradeTo(address newImpl) 函数来升级到新的实现合约。不过，由于升级逻辑在代理合约中，部署此类代理的成本较高。

同时，透明代理需要管理机制来决定是委托调用实现合约的功能，还是执行代理合约本身的功能。以 Box 合约为例：

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.13;

contract Box {

uint256 private \_value;

function store(uint256 value) public { /\*..\*/ }

function retrieve() public view returns (uint256) { /\*..\*/ }

}

contract BoxProxy {

function \_delegate(address implementation) internal virtual { /\*..\*/ }

function getImplementationAddress() public view returns (address) { /\*..\*/ }

fallback() external { /\*..\*/ }

// Upgrade logic in Proxy contract

function upgradeTo(address newImpl) external {

// Changes stored address of implementation of contract

// at its slot in storage

}

}

UUPS代理

与透明代理不同，UUPS 代理的升级逻辑由实现合约本身处理，实现合约不仅包含业务逻辑，还包含升级逻辑。

可通过让实现合约继承包含升级逻辑的通用标准接口（如 OpenZeppelin 的 UUPSUpgradeable 接口）来使其符合 UUPS 标准。强烈建议继承该接口，因为若新版本实现合约不包含升级逻辑，将导致升级机制被锁定。

透明代理模式详解

代理模式简介

在以太坊的智能合约世界中，合约一旦部署便具有不可修改性。这种特性虽然为合约提供了强大的安全保障，但在实际应用场景里，当我们需要修复合约中的 bug 或者对其功能进行升级时，就会面临挑战。为了解决这一问题，代理模式应运而生。代理模式的核心思想是将合约的逻辑与数据存储分离开来，从而实现合约的可升级性。

透明代理（Transparent Proxy）是代理模式中最为流行的一种，由 OpenZeppelin 提出并在行业内得到了广泛的应用。

透明代理的核心问题：函数选择器冲突

代理模式在实际应用中面临的核心挑战是函数选择器冲突问题。当代理合约和逻辑合约中存在名称和参数都相同的函数时，就会产生冲突。例如，若逻辑合约和代理合约都包含 upgradeTo(address) 函数，那么在调用时，系统将无法明确应该执行哪个函数。

透明代理模式通过区分管理员和普通用户的调用，巧妙地解决了这一问题：

对管理员的调用：直接在代理合约中执行。

对普通用户的调用：转发到逻辑合约执行。

透明代理使用流程

部署逻辑合约 V1：首先部署第一个版本的逻辑合约。

部署透明代理合约：部署透明代理合约，并将其指向 V1 的地址。

交互阶段：调用代理合约的地址，但使用 V1 的 ABI 进行交互。

升级准备：当需要进行升级时，部署 V2 逻辑合约。

合约升级：管理员调用代理合约的 upgradeTo 函数，将其指向 V2 地址。

继续交互：继续使用代理合约地址，但使用 V2 的 ABI 进行交互。

最佳实践与安全建议

状态变量布局：在升级合约时，不要修改、删除或重排现有状态变量，以确保存储布局的兼容性。

管理员权限分离：考虑使用多签钱包或 DAO 作为管理员，以增强权限管理的安全性。

初始化函数：确保初始化函数有适当的访问控制，防止未授权的初始化操作。

升级后验证：每次升级后，都要验证新功能是否正常工作，确保升级的稳定性。

存储冲突规避：使用特定命名模式或结构体避免存储冲突，保证合约的正常运行。

UUPS代理

设计优势

这种设计带来了以下显著优势：

轻量化：代理合约中不包含复杂的升级逻辑。

灵活性：开发者可以完全定制升级权限和流程。

标准化：基于 ERC - 1967 和 ERC - 1822，兼容性好，便于集成。

关键组成部分

代理合约（Proxy Contract）

职责

存储当前实现合约的地址。

使用 delegatecall 将所有调用转发到实现合约。

存储规范

遵循 ERC - 1967 标准，使用固定的存储槽 IMPLEMENTATION\_SLOT 保存实现合约地址。

轻量设计

代理合约不包含任何升级逻辑，仅负责转发调用。

实现合约（Implementation Contract）

职责

包含具体的业务逻辑和升级逻辑。

提供对新实现合约的兼容性检查。

接口规范

实现 ERC - 1822 标准中的 proxiableUUID 函数，用于验证实现合约的兼容性。

通常基于 OpenZeppelin 的 UUPSUpgradeable 模块进行开发。

升级管理

权限控制

实现合约通过 \_authorizeUpgrade 函数限制升级权限。

升级执行

升级操作由实现合约的 upgradeTo 或 upgradeToAndCall 函数执行。

安全机制

内置上下文校验，确保只有代理合约可以调用升级逻辑。

校验新实现合约的兼容性，避免意外升级。

UUPS代理的工作原理

UUPS 代理的工作流程分为两部分：调用转发和合约升级。

调用转发

1、用户向代理合约发送请求。

2、代理合约通过 delegatecall 将请求转发到当前实现合约。

3、实现合约处理逻辑，并使用代理合约的存储。

合约升级

升级过程通过实现合约中的 upgradeTo 或 upgradeToAndCall 完成：

1、上下文校验：检查调用者是否为代理合约，防止直接调用实现合约中的升级逻辑。

2、权限验证：调用 \_authorizeUpgrade 检查升级权限。

3、兼容性检查：验证新实现合约的 proxiableUUID 是否与 ERC - 1967 标准兼容。

4、存储更新：将代理合约中的实现地址更新为新合约地址。

5、初始化（可选）：在升级完成后，通过 upgradeToAndCall 执行初始化逻辑。

UUPS 与 Transparent Proxy 的对比



UUPS 代理的安全注意事项

权限控制

确保 \_authorizeUpgrade 实现了严格的权限验证（如 onlyOwner）。

避免任意用户触发升级逻辑。

存储布局

新旧实现合约的存储布局必须完全兼容，否则会导致数据混乱。

兼容性检查

新实现合约必须实现 proxiableUUID 并返回正确的存储槽值。

初始化

升级后需要初始化新的状态变量时，务必通过 upgradeToAndCall 调用初始化函数。

UUPS（Universal Upgradeable Proxy Standard） 和 透明代理（Transparent Proxy） 的主要区别确实集中在升级权限管理和函数调用路由逻辑上，而普通业务逻辑的代理执行（即用户与合约的交互）在两者中确实是基本相同的。以下是关键点的对比：

1. 代理执行逻辑的相同点

业务逻辑调用流程：

无论是 UUPS 还是透明代理，当用户调用逻辑合约（Implementation）的函数时，代理合约（Proxy）都会通过 delegatecall 将请求转发到逻辑合约执行。业务逻辑的执行完全一致，用户无需关心底层是哪种代理模式。

存储结构：

两者都使用代理合约的存储，逻辑合约仅提供代码执行。

2. 核心区别：升级权限与路由逻辑



3. 关键细节

UUPS

升级函数（如 upgradeTo(address)）必须由逻辑合约实现，并通过 delegatecall 触发。\_authorizeUpgrade(address)

升级权限通常通过逻辑合约的访问控制（如 OpenZeppelin 的 Ownable）管理。

风险点：如果逻辑合约未包含升级函数或升级函数存在漏洞，代理将无法再升级。

透明代理

代理合约直接处理升级（如通过 upgradeTo(address)），但仅允许管理员（Admin）调用。

用户与管理员调用通过 msg.sender 自动路由：

如果是管理员，调用代理合约的升级函数。

如果是普通用户，转发到逻辑合约。

风险点：管理员地址可能被误操作（如直接调用逻辑合约的初始化函数导致漏洞）。

StateDB 是区块链系统中的一个核心概念，主要用于存储和管理系统的状态数据。下面是对它的详细解释：

什么是 StateDB？

StateDB状态数据库是区块链系统中用于存储每个账户（或合约）的最新状态的组件。它记录了包括账户余额、合约存储变量、nonce 等所有与当前区块状态相关的数据。

它解决了什么问题？

在区块链中，每个新区块的生成都伴随着系统状态的更新（比如账户余额变化、合约变量修改等），因此必须有一个机制来追踪和存储这些变化。这就是 StateDB 的作用：

保存所有账户和合约的当前状态。

提供查询和更新状态的能力。

支持区块的回滚与状态快照，便于处理链的重组（reorg）或回退。

tateDB 通常包含哪些内容？

以以太坊为例，StateDB 中包含：

项目 说明

地址（Address） 每个用户或合约的唯一标识

余额（Balance） 账户中持有的 ETH 数量

Nonce 账户已发送交易的数量，用于防止重放攻击

存储（Storage） 合约中的变量（键值对），通过 Merkle Patricia Tree 组织

代码（Code） 智能合约的字节码

这些数据通过一种叫做 Merkle Patricia Trie 的数据结构存储，从而实现高效的查找、验证与同步。

数据结构与存储形式

在如以太坊这样的系统中，StateDB 不是直接存在于普通的数据库中，而是被组织为：

账户状态树（State Trie）：每个账户为一个节点

每个合约的存储又构成一个独立的 trie（合约存储 Trie）

整个状态树的根哈希（stateRoot）会被写入区块头中，确保状态不可篡改且可验证。

特性 说明

核心作用 存储全网状态，包括账户、合约等

数据结构 多级 Merkle Patricia Trie

与区块的关系 每个区块都有一个对应的状态根哈希（stateRoot）

使用场景 状态读取、状态更新、验证历史状态