在合约中，一个合约继承了一个接口，如果合约还有接口未包含的函数A，则用接口创建合约实例的时候，本实例不能调用A函数

验证为：

 interface Idemo {

    function set(uint a) external;

    function get() external view returns(uint);

 }

import "./Idemo.sol";

contract demo is Idemo{

    uint num;

    function set(uint a) external{

        num= a;

    }

    function get() external view returns(uint){

        return num;

    }

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

        return a+b;

    }

}

import "./demo.sol";

contract verify is demo{

    // IERC721 BAYC = IERC721(0xBC4CA0EdA7647A8aB7C2061c2E118A18a936f13D);

    Idemo demo = Idemo(0xd9145CCE52D386f254917e481eB44e9943F39138);

    uint a =1;

    uint b =2;

    uint result;

    function verifying() public  {

        // result = demo.add(a,b);//报错找不到函数

        demo.get();

    }

}

对于数组：constant修饰符不能修饰数组，另外memory修饰的数组可以由new创建，这是动态数组

对于使用oz的upgrade，数组必须在initialize函数里面赋值，直接声明并赋值不得行

对于不生成getter函数的变量，则可以设置为private或者internal，以节约合约字节码

Modifier可以接收参数，并根据此参数做出处理

对于public的静态数组：会生成getter方法，调用时需要传入数组下标

示例： contract Storage {

    uint256[4] public arry;

    constructor() {

        arry = [1,2,3,4];

    }

}

对于这样的arry数组，调用时注册ABI：function arry(uint256) public view returns(uint256)

调用时就直接传入uint256类型的数组下标就行了

//=============================================================================

在解码时：abi.decode不能解析包含函数选择器的部分，只能解析参数部分，即只能解析

abi.encode的编码，若要拿到calldata的前四个字节（函数选择器），则可用bytes4(data)

直接截取。若要拿截掉函数选择器后面的数据并且decode：

    function decodeReplaceOwnerTxData(

        bytes memory data

    ) public pure returns (address oldOwner, address newOwner) {

        bytes memory argsData;

        assembly {

            argsData := add(data, 0x4)

        }

        (oldOwner, newOwner) = abi.decode(argsData, (address, address));

    }

# 内联汇编：

function at(address addr) public view returns (bytes memory code) {

  assembly {

      // 获取代码大小，这需要汇编语言

      let size := extcodesize(addr)

      // 分配输出字节数组 – 这也可以不用汇编语言来实现

      // 通过使用 code = new bytes(size)

      code := mload(0x40)

      // 包括补位在内新的 “memory end”，这里要取反对齐

//即取反对齐就是在后面补0

      mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))

      // 把长度保存到内存中

//这一行将智能合约字节码的长度 size 存储在 code 的开头位置，作为字节数组的第一个 32 字节的槽。

//code 内存区域就包含了两个部分，第一个部分是字节码的长度（32 字节），第二个部分是实际的字节码内容

      mstore(code, size)

      // 实际获取代码，这需要汇编语言

//0：外部合约字节码的起始位置，这里是从开头开始。

//从 add(code, 0x20)位置开始，从0开始复制，复制大小为size

      extcodecopy(addr, add(code, 0x20), 0, size)

  }

}

**code** 被使用为一个动态字节数组，因为返回类型声明为 **bytes memory**，另外mstore(code, size)这样是把大小存入code变量，因为动态字节数组第一个槽存的是数组大小，后面才是存的具体数据

另外动态数组和mapping的存储方式详见官方文档

# 关于事件：

indexed最多是3个。日志的第一部分是topic数组，用于描述事件，长度不能超过4，第一个元素为事件hash，后面才是index的具体值，一般前面填充0.

data 部分的变量不能被直接检索，但可以存储任意大小的数据，因此可以存复杂的东西（>256bit），如果硬是要存在topic，那么就是hash存储。data 部分的变量在存储上消耗的gas相比于 topics 更少

# 数组

-----在数组中，若存在[uint(1),2,3]，则由于uint默认uint256，全部元素就为uint256

这是因为第一个元素指定了，若[1,2,3]这样的数组，如果一个值没有指定type的话，默认就是最小单位的该type

Eg： function f() public pure {

g([uint(1), 2, 3]);

}

function g(uint[3] memory \_data) public pure {

// ...

}

如果没有对传入 g() 函数的数组进行 uint 转换，是会报错的

## Mekerl树踩坑

首先，在用网站[MerkleTree.js example (miguelmota.com)](https://lab.miguelmota.com/merkletreejs/example/)去生成Merkel树的时候要点选中的三个东西

然后，在传入proof的时候，要看清楚leaf是哪个，对应地去传proof，因为在生成树时，你的输入被打乱了

另外，必须在生成树的时候排序，因为openzepplin库里面的函数是这样的：

function \_hashPair(bytes32 a, bytes32 b) private pure returns (bytes32) {

        return a < b ? \_efficientHash(a, b) : \_efficientHash(b, a);

    }

她要判断大小然后计算hash，但用网站生成的计算是不比较的，所以，必须在生成树的时候选择排序按钮

## Pure 和 view

Pure就是纯函数，既不能读取又不能改变链上的变量，view 可以去读取链上变量，但不能改变它。 两者共同点就是不耗gas

在排序地址的时候，若按照地址大小来排序，如：

function sortTokens(address tokenA, address tokenB) internal pure returns (address token0, address token1) {

        require(tokenA != tokenB);

        (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);

        require(token0 != address(0));

}

最后的require其实对于两个地址都做了不为零的判断，因为若是零地址的话，0地址会在第一位，第二位的绝对不是0地址，故只需要判断第一个地址！0即可

===================================================================///==========

## 函数报错排错

遇到报错，但test脚本又可以跑的通，在正式区块链上又跑不通，此时，在区块链浏览器上查看：

点圈起来的：

然后会得到调用函数的所有信息，call，delegatecall（这个一般是调用impl合约，也就是说他是可升级的），staticcall。根据最后一次报错的点一步一步往上捋，然后找到是哪报错。

另外，我自己有脚本：可根据input的内容以及合约ABI恢复出调用的函数是哪个，并且传参是什么。

脚本为：

const { ethers } = require('ethers');

const abi = [ ]

const iface = new ethers.Interface(abi\_linearDecrease);

  const inputData = " "

//   decode

  const parseData = iface.parseTransaction({ data: inputData });

  console.log(parseData);

  console.log("Function Name:", parseData.name);

另外，再自己根据上面的脚本还原出的数据来写etherjs自己调用一下，打印出报错信息以及调用细节

**脚本**：

const { ethers } = require('ethers');

const contractABI = [ ]

// 你的合约地址（在这里替换成你的合约地址）

const contractCDPAddress = ;

// 你的以太坊私钥（用于发送交易）

const privateKey = ;

// 创建以太坊 provider 和 signer

const provider = new ethers.JsonRpcProvider('https://.......');

const signer = new ethers.Wallet(privateKey, provider);

// 通过 ABI 和地址连接到合约

const contract = new ethers.Contract(clipperAddr, abi\_clipper, signer);

// console.log(contract);

const main = async ()=> {

    const res = await contract.functionName("arg");

    console.log(res);

    const multi = res[1] \* res[2];

    console.log(multi);

}

main();

//===============================================//=============================

在写NatSpec 的注释时，要注意函数上方的@param 与传入参数对的上不，若对不上，则hardhat报错**HH600**，还有，注意标签的使用地方，使用错了还是报错**HH600**