合约见解及分析

一：有关链接

1，基本语法：https://docs.soliditylang.org/

2，在线编译、部署、测试和调试：https://remix.ethereum.org/

3，环境编译、部署、测试和调试：https://hardhat.org/

4，游戏形式开发学习： https://cryptozombies.io/zh/

5，合约编写的三方库：https://docs.openzeppelin.com/

6，安全问题汇总：https://swcregistry.io/docs/SWC

二：基础语法（用到的基本语法使用链接都有介绍的）,这面主要说踩坑点

1， uint

无符号 8 到 256 位

坑点分析：越界

解决：如果编译版本 8.0 以上，编译器已经做了越界的处理，越界就报 error，否则要使用 Safemath 进行安全处理。

2， mapping 及数组

坑点分析 1：mapping 类型则不能作为临时变量使用，只能作为某个状态变量的"储存指针"，也就是 mapping 类型必须在编译时进行预先初始化，而不能作为运动时产生的数据。

坑点分析 2：

mapping 是不能遍历的。

坑点分析 3：

delete(将变量设为默认值，但是降低了gas的费用) array[index]

数组的删除：当你去 delete 数组某一元素的时候，数组的的长度没变，只是把删除的元素初始化了。

解决：

```
array[index] = array[array.length - 1];

array.pop();
```

数组长度改变并彻底删掉元素

3， msg.sender, msg.value 等与合约调用相关的变量

坑点分析：和 call，delegatecall 连用攻击风险。

解决：针对 call，delegatecall 的使用，调用者，被调用者及调用环境要搞明白。

4，msg.sender 和 tx.origin

坑点分析：针对 msg.sender 和 tx.origin 获取的 address 问题。

解决：msg.sender 是当前调用合约地址,tx.origin 是最初调用合约地址。

5， external 、public 、internal、private 修饰符

坑点分析：修饰符的攻击风险，例如把私有的变量或者方法设置 public 谁都可以调用。

解决：这些方法和变量考虑清楚进行定义避免出现问题。

6， constant、 view、 pure 修饰词

constant：Solidity v4.17 之前的变量，后续拆分成了 view 和 pure

view：承诺不修改状态。

pure：承诺不读取或修改状态（更为严格）。

例：

```
contract ViewAndPure{

   uint256 value = 10;

   function getValueView() public view returns(uint256){

       return value;
```

```
    }

    //编译会报错

    function getValuePure() public pure returns(uint256){

        return value;//rerunrn 1 就行了。

    }

}
```

坑点分析：理解问题

解决：知道修饰词的含义

坑点分析：如果此函数方法没有用 view 修改词修饰呢是否能读到?

解决：能读到

例：

```
const {

    earlyExitFee,creditBurned

} = await contract.methods

    .calculateEarlyExitFeeLessBurnedCredit('0xd24F43332779947e3A17a5f7D58BB6c4189e8d35',
'0xAc913b113FC02cfE76e793e6C86b6a0c18d32AAE','60000000')

    .call();
```

call 就行。

正常读取 view 的方法不用 call 就行。

7， 数据存储

Evm 是基于字寻址的字数组进行数据存储（2 的 256 次方长度）。

所有的 storage 变量都会存在这个大数组中，每一个都是存储槽。

链接：https://docs.soliditylang.org/en/v0.8.13/internals/layout_in_storage.html

存储类型：

1，值类型：uint,bool， 定长字节数组，等（即当它们用作函数参数或赋值时，它们总是被复制。）

2，引用类型：引用类型是一个复杂类型，占用的空间通常超过 256 位， 拷贝时开销很大。

不定长字节数组，字符串，数组，结构体，针对它，提出了数据位置

数据位置：

存储是以字为单位（字 = 32 字节）方便存取 keccak256 哈希值和椭圆曲线相关运算。

一个卡槽位 256 位（32 字节）。

-memory（一个简单的字节数组，用于临时存储 EVM 代码运行中需要存取得各种数据故在外部调用后会被移除，故 GAS 开销很小）

-storage

(变量是指永久存储在区块链中的变量，由于会永久保存合约状态变量，故 GAS 开销也最大)

-栈（所谓的运行栈，用来保存 EVM 指令的输入和输出数据（每个单元是一个字，实际是 memory 里的一个数据结构，值类型的局部变量是存储在栈上的），几乎免费使用的内存，但有数量限制，最大长度 1024 个元素并包含256位的字（但是栈顶元素调用栈里元素不能超过16元素的距离）超过会报stack too deep）

默认值

-函数的参数和返回值默认是 memory

-局部变量和状态变量默认是 storage（值类型的局部变量存在栈中）

坑点分析：内存问题及运用不好会把值赋错，针对 storage 是引用传递，而 memory 只是值传递。

例 1：修饰符不同最终的结果也是不一样的。

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.23;

contract MemoryTest{

   struct STest{string a;uint b;}

   //状态变量

   STest s;

   function convertStorage(STest storage s) internal{

       STest storage tmp = s;
```

```
        tmp.a = "rudui";
    }
    //值会改变
    function callStorage() public returns ( string memory){
        convertStorage(s);
        return s.a;
    }
    /*************************************************************/
    function memoryToState(STest memory tmp) internal{
        s = tmp;
        tmp.a = "rudui";
    }
    //值不会改变
    function callMemory() public returns(string memory){
        STest memory tmp = STest("memory", 0);
        memoryToState(tmp);
        return s.a;
    }
```

例 2：这样是不能赋值的,参数默认是 memory,局部变量默认是 storage,需要 S tmp 改为 S memory tmp, S s 改为 S memory s。（像动态数组,struct,mapping 这样的复杂数据结构是不能直接在栈里存储的，因为栈只能保存单独的"子"，也就是只能保存实际数据长度小于等于 32 的简单数据类型。所以在智能合约函数中声明动态数组和 struct 时，必须明确指明其位置在 storage 还是 memory 中。

```
)

pragma solidity ^0.4.0;

contract SimpleAssign{
    struct S{string a;uint b;}
    //默认参数是 memory
```

```
    function assign(S s) internal{
    //默认的变量是 storage 的指针
    //Type struct MemoryToLocalVar.S memory is not implicitly convertible to expected type struct MemoryToLocalVar.S storage pointer.
    S tmp = s;
    }
}
```

解决：根据运用情况定义变量。

例：这面以数组为例介绍它的存储位置

数组存储的位置也是按变量顺序，只是里面存的值是数组的长度（存储槽 value 是数组长度）

当数组去 push 一个值的时候这个值应该是存在哪里的呢？

获取位置：

```
function arrLocation(uint256 slot, uint256 index, uint256 elementSize)
    public
    pure
    returns (uint256)
{
    return uint256(keccak256(slot)) + (index * elementSize);
}
```

利用这一方法获取位置，再去调用下面的 rpc 进行查询。
注：每次加完数据后存储的数组长度（也就是存储数据长度的存储槽）都会更新到最新长度。

Rpc-获取存储位置的值：eth_getStorageAt

例：https://rinkeby.infura.io/v3/fb2a09e82a234971ad84203e6f75990e

```
{"jsonrpc":"2.0","method":"eth_getStorageAt","params":
["0xbC98Dd90128d03988B81465dc253E0F1B636792f",
"0x0000000000000000000000000000000000000000000000000000000000000000",
"latest"],"id":1}
```

然后就可以获取到这个位置的存储槽的值。

8， payable 及 Fallback 函数

payable：

用来合约接收及提现 eth

注：

提现 3 中方式：

```
withdrawAddress.transfer(1)

receiveAddress.send(uint256 amount)

receiveAddress.call.valle{}

contract PayableTest{

    address payable withdrawAddress;

    //存入 eth

    function deposit() public payable{

    }

    //设置提现地址

    function setwithdrawAddress(address payable _address) public{

        withdrawAddress = _address;

    }

    //查询当前的余额

    function getBalance() public view returns(uint256){

        return address(this).balance;

    }

    //提现

    function withdraw() external {

        withdrawAddress.transfer(1);
```

```
        //receiveAddress.send(uint256 amount)

        //receiveAddress.call.valle()

    }

}
```

坑点分析：针对 call{value:1}这一调用，因为不限制 gaslimit,若 address 的 callback()有恶意攻击代码，可能带来不可预估的损失，如递归式恶意调用你的程序

```
function test(address payable adr 里没有) public returns (uint){

    (bool sucess, bytes memory s) = adr.call{value:1}(abi.encodeWithSignature("set_num(uint)", 10));

    require(sucess);

    return address(this).balance;

}
```

回调合约：

```
fallback() external payable {

     a = a +  1;

}
```

解决：尽量使用前两种出金。

Fallback：

回退函数是合约里的特殊函数，没有名字，不能有参数，没有返回值。

调用：

1，向合约地址发送金额为 0 的交易会调用（这面指 eth）

2，向合约请求不存在的方法会调用

用途：

1，空投：用户转 0 就行，只有手续费。

调用示例：合约不存在此函数

```
uint public a;

fallback() external {

     a = a +  1;

}

//合约不存在此函数

function sendCall() public {

    (bool success,) = address(this).call(abi.encodeWithSignature("specificSend()"));

    //(bool success,) = address(this).delegatecall(abi.encodeWithSignature("specificSend()"));

    if(success){

    }

}
```

9， event 事件

坑点分析：无用事件。

解决：和业务方面及自己的需求方面定义好事件。

10， 合约的重载和继承

坑点分析：暂无

11， 合约间调用之接口的定义

坑点分析：部分合约协议方法的修改。

例如 ERC20 代码的转账方法，有的 Erc20 的转账是有返回值的，有的 Erc20 转账是无返回值，这样你按照新有返回的 Erc20 协议去定义接口并直接利用此 interface 去进行合约间调用，无返回值的 Erc20 的转账则会调用失败。

解决：写适配有无返回值的类。

```
library SafeERC20 {
```

```
    using SafeMath for uint256;
    using Address for address;
    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }
    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to,
value));
    }
    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        if(token.allowance(address(this),spender) < value){
            _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender,
value));
        }
    }
    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender,
newAllowance));
    }
    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(value, "SafeERC20:
decreased allowance below zero");
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender,
newAllowance));
    }
    function _callOptionalReturn(IERC20 token, bytes memory data) private {
```

```
        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed");
        if (returndata.length > 0) { // Return data is optional
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}
```

12， 合约库 library（避免重复造轮子)

```
坑点分析：
1，没有状态变量
2，不能够继承或被继承
3，不能接收以太币
```

三：高级语法

1， 合约间调用

（1）， interface 调用

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;
interface Erc20ReserveInterface{
    function getExternalErc20ReserveAddresses() external view returns(address);
}
contract InterfaceTest {
    function test(address _address) public{
       address returnAddress = Erc20ReserveInterface(_address).getExternalErc20ReserveAddresses();
```

```
    }

}
```

（2），签名方式调用：call,delegatecall

函数的设计目的是为了使用存储在另一个合约的库代码。

二者执行代码的上下文环境的不同，当使用 call 调用其它合约的函数时，代码是在被调用的合约的环境里执行，对应的，使用 delegatecall 进行函数调用时代码则是在正在调用函数的合约的环境里执行。

针对 call，delegatecall 的环境一定要搞清楚，负责就会出现问题。

坑点分析 1：msg.sender 问题：

解决：address(test).delegatecall（） msg.sender 是当前合约的地址

address(test).call（）msg.sender 是被调用的合约地址

坑点分析 2：(bool success,) =
address(this).delegatecall(abi.encodeWithSignature("specificSend()"))

就算调用参数及方法不对，也不会执行合约异常，只是 success 和 false。

解决：这面需要注意下。

```
 // SPDX-License-Identifier: GPL-3.0


pragma solidity >=0.8.0;
contract Test {
    uint256 public value = 5;
    function specificSend() public{
        if(address(this) == msg.sender){
            value = 10;
        }
    }
}
contract CallTest {
    function sendCall(address test) public {
```

```
        (bool success,) = address(test).call(abi.encodeWithSignature("specificSend()"));

        //(bool success,) =
address(test).delegatecall(abi.encodeWithSignature("specificSend()"));

        if(success){

        }

    }

}
```

坑点分析 3：赋值问题

我们知道使用 delegatecall 时代码执行的上下文是当前的合约，这代表使用的存储也是当前合约，当然这里指的是 storage 存储，然而我们要执行的是在目标合约那里的 opcode，当我们的操作涉及到了 storage 变量时，其对应的访存指令其实是硬编码在我们的操作指令当中的，而 EVM 中访问 storage 存储的依据就是这些变量的存储位

解决：这种较复杂的上下文环境下涉及到 storage 变量时可能造成的变量覆盖，对于这种漏洞感觉如有需要还是避免直接使用 delegatecall 来进行调用，应该使用 library 来实现代码的复用，

```
pragma solidity =0.8.0;

contract calltest {

    address public c;

    address public b;

    function test() public returns (address a){

        a=address(this);

        b=a;



    }

}

contract compare {

    address public b;

    address public c;

    function withdelegatecall(address _address) public{

        address(_address).delegatecall(abi.encodeWithSignature("test()"));

    }

}
```

2，合约升级

作为合约升级可以在地址不变的情况下对合约进行修改，更改智能合约以修复他们发现的错误，

同时也暴露了中心化问题（owner 可以作恶）。

如被攻击可以找到漏洞代码，及时对合约进行升级，避免造成更大损失。

1,主-从模式（其实就是接口模式）

利用 Interface 进行合约的升级

例：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;
contract Erc20Reserve{
    function getExternalErc20ReserveAddresses() external view returns(address){
        reture 0x000;
    }
}
interface Erc20ReserveInterface{
    function getExternalErc20ReserveAddresses() external view returns(address);
}
contract InterfaceTest {
    address erc20Reserve;
    function test() public{
        address returnAddress =
Erc20ReserveInterface(erc20Reserve).getExternalErc20ReserveAddresses();
    }
    function setErc20Reserve(address _address) external onlyOwner{
        erc20Reserve =  _address
    }
}
```

如上：erc20Reserve 是可以 owner 设置的,这样 owner 可以在保证方法和参数不变的情况下修改内部逻辑再次设置 erc20Reserve 地址改变逻辑。

2,代理模式升级

例如 openzeppelin 的 ProxyFactory

https://docs.openzeppelin.com/learn/upgrading-smart-contracts#upgrading-a-contract-via-plugins

可以按照文档自己练一下。

例：0x5818428C703d9Be22dE5fE8A3FcEAc770210d384 eth 测试网

升级如何运作：

1，您编写的合约，称为包含逻辑的实现合约。

2，一个 ProxyAdmin 作为代理的管理员。

3,实现合约的代理，这是您实际与之交互的合约。

坑点：

1,升级合约的构造函数需要去掉,用 initialize 这一方法进行替换（此方法只保证调用一次），这是对某些潜在攻击的缓解。

2,由于技术限制，当您将合同升级到新版本时，您无法更改该合同的存储布局。

这意味着，如果您已经在合约中声明了一个状态变量，您就不能删除它、更改它的类型或在它之前声明另一个变量

幸运的是，这个限制只影响状态变量。您可以根据需要更改合约的功能和事件。

error：New variables should be placed after all existing inherited variables

所以要放在变量之后。

```
// contracts/Box.sol

contract Box {

    uint256 private _value;



    // We can safely add a new variable after the ones we had declared
```

```
    address private _owner;


    // ...
}
```

实现：

1,生成 Box 地址

```
const Box = await ethers.getContractFactory('Box');
console.log('Deploying Box...');
const box = await upgrades.deployProxy(Box, [42], { initializer: 'store' });
await box.deployed();
console.log('Box deployed to:', box.address);
```

2,修改 Box 得到 BoxV2 进行合约的升级

```
const BoxV2 = await ethers.getContractFactory('BoxV2');
console.log('Upgrading Box...');
await upgrades.upgradeProxy('Box 地址', BoxV2);
console.log('Box upgraded');
```

3，升级成功

3，合约克隆

EIP1167，根据最小代理合约：https://eips.ethereum.org/EIPS/eip-1167

https://github.com/optionality/clone-factory

目的：廉价部署（部署克隆的低 gas）等

原理：为了以不可变的方式简单且廉价地克隆合约功能，该标准指定了一个最小的字节码实现，它将所有调用委托给一个已知的固定地址。

例：

1，先部署 MetaCoinClonable 合约

2，再部署 MetaCoinCloneFactory 合约

3，MetaCoinCloneFactory 执行 setLibraryAddress 方法

4，MetaCoinCloneFactory 再执行 createMetaCoin 生成克隆地址。

5，MetaCoinCloneFactory 再执行 getMetaCoins()方法拿到克隆后的地址。

6，进行验证。

```
pragma solidity ^0.5.0;
contract MetaCoinClonable {
    mapping (address => uint) balances;
    function initialize(address metaCoinOwner, uint256 initialBalance) public {
        balances[metaCoinOwner] = initialBalance;
    }
    function sendCoin(address receiver, uint amount) public returns(bool sufficient) {
        if (balances[msg.sender] < amount) return false;
            balances[msg.sender] -= amount;
            balances[receiver] += amount;
            return true;
    }
    function getBalance(address addr) view public returns(uint) {
        return balances[addr];
    }
}

// https://github.com/optionality/clone-factory/blob/master/contracts/CloneFactory.sol
contract CloneFactory {
    function createClone(address target) internal returns (address result) {
        bytes20 targetBytes = bytes20(target);
        assembly {
            let clone := mload(0x40)
            mstore(clone, 0x3d602d80600a3d3981f3363d3d373d3d3d363d730000000000000000000000000)
```

```
            mstore(add(clone, 0x14), targetBytes)
            mstore(add(clone, 0x28),
0x5af43d82803e903d91602b57fd5bf3000000000000000000000000000000000000)
            result := create(0, clone, 0x37)
        }
    }
}
//以后就可以一直用这个合约部署一样功能的合约。
contract MetaCoinCloneFactory is CloneFactory{
    MetaCoinClonable[] public metaCoinAddresses;
    event MetaCoinCreated(MetaCoinClonable metaCoin);
    address public libraryAddress;
    address public metaCoinOwner;

    function setLibraryAddress(address _libraryAddress) external{
        libraryAddress = _libraryAddress;
    }
    function createMetaCoin(address _metaCoinOwner, uint256 initialBalance) external {
        MetaCoinClonable metaCoin = MetaCoinClonable(
            createClone(libraryAddress)
        );
        metaCoin.initialize(_metaCoinOwner, initialBalance);

        metaCoinAddresses.push(metaCoin);
        emit MetaCoinCreated(metaCoin);
    }
    function getMetaCoins() external view returns (MetaCoinClonable[] memory) {
        return metaCoinAddresses;
    }
}
```

四：优化问题

1，gas 优化：

（1），数值类型：uint

1，排序：合约的状态变量以紧凑的方式存储在存储中（只针对值类型）
，每个槽32字节
uint128,uint128，uint256。比uint128，uint256，uint128 省存储（前2槽后3槽）

2，如果不排序

uint256，uint256，uint256 比uint128,uint256,uint128

省gas（在读取的时候，由于evm一次运行32个字
节，所以说读值的时候要将大小从32字节减少到
所需字节）

3，初始值（因为存储槽默认就是0）

uint256 value 比 uint256 = 0 更省gas

（2）， external 、public

当可以使用 external 时，不要使用 public
external 的效率更高，因为 Solidity 不需要允许两个入口点。
external 比 public 更节省 gas。
对于 public 函数，每次调用时 Solidity 会将参数 copy 到内存中；而调用 external 函数，则可以直接读取 calldata。内存分配在 EVM 中是非常昂贵的，而读 calldata 则相对廉价很多。

五：测试

都是利用 hardhat 进行测试

1，单合约测试

```
const { expect } = require("chai");

const { ethers } = require("hardhat");

describe("Greeter", function () {

    it("Should return the new greeting once it's changed", async function () {

    const Greeter = await ethers.getContractFactory("Greeter");
```

```
    const greeter = await Greeter.deploy("Hello, world!");

    await greeter.deployed();

    expect(await greeter.greet()).to.equal("Hello, world!");

    const setGreetingTx = await greeter.setGreeting("Hola, mundo!");

    // wait until the transaction is mined

    await setGreetingTx.wait();

    expect(await greeter.greet()).to.equal("Hola, mundo!");

    });

});
```

运行：

```
$ npx hardhat test


Contract: Greeter

    ✓ Should return the new greeting once it's changed (762ms)

1 passing (762ms)
```

2，多关联合约测试

主要模拟智能合约依赖项进行测试

文档：https://ethereum-waffle.readthedocs.io/en/latest/mock-contract.html

主要用到 ethereum-waffle 的 mock 功能，进行其他合约的 mock,进行合约多功能的测试。

例：

```
//利用 deployMockContract 进行依赖合约的 mock。
{deployMockContract} from '@ethereum-waffle/mock-contract'

import {deployMockContract} from '@ethereum-waffle/mock-contract';

...

const mockContract = await deployMockContract(wallet, contractAbi);

//进行合约的 mock 数据

await mockContract.mock.balanceOf.returns(utils.parseEther('999999'));
```

运行：

```
$ npx hardhat test

Contract: Greeter
    ✓ Should return the new greeting once it's changed (762ms)

1 passing (762ms)
```

六：安全问题（典型）

1，call、delegatecall 滥用

```
 // SPDX-License-Identifier: GPL-3.0

 pragma solidity >=0.8.0;

//二者执行代码的上下文环境的不同，当使用 call 调用其它合约的函数时，代码是在被调用的合约的环境里执行，对应的，使用 delegatecall 进行函数调用时代码则是在调用函数的合约的环境里执行,

contract CallTest {

    uint256 public value = 5;

    function sendNormal() public {

        specificSend();

    }

    //call 会修改 msg.sender 为调用者本身 所以说 msg.sender 不是调用方，已经是被调合约了，已经是当前合约了

    //delegatecall 恰恰相反还是调用合约

    function sendCall() public {

        (bool success,) = address(this).call(abi.encodeWithSignature("specificSend()"));

        //(bool success,) =
address(this).delegatecall(abi.encodeWithSignature("specificSend()"));

        if(success){

        }

    }

    function sender(bytes memory data) public{
```

```
        (bool success,) = address(this).call(data);
        if(success){
        }
    }
    function specificSend() public{
        if(address(this) == msg.sender){
         value = 10;
        }
    }
}
```

坑点分析：

1，msg.sender 处理不好（针对 call、delegatecall 的使用）,有可能会让调用方执行方法绕过权限进行攻击。

2，address(this).call(data) 直接执行字节也是同上,调用利用拼好的字节执行方法对合约进行攻击。

解决：逻辑清晰,在运用 call 及 delegatecall 的时候要考虑完善。

2， uint 越界攻击

坑点分析:

1，如果编译器还是用老版本并且没有做 SafeMath 处理,则会遇到越界攻击，导致合约内部计算出错。

解决：高编译器 8.0 以上或者 SafeMath 处理。

3， 时间戳操纵（block.timestamp 或 blockhash 攻击问题）

15s 规则， 如果一个块的时间戳是超过未来 15s 的，目前 geth 与 parity 实现版本都会拒绝，所以想用这个变量的，需要权衡是否在 15 内完成。

来源：https://consensys.net/blog/developers/solidity-best-practices-for-smart-contract-security/

坑点分析：

不要依赖 block.timestamp 或 blockhash 作为随机性的来源，除非你知道自己在做什么。

时间戳和区块哈希都会在一定程度上受到矿工的影响。例如，采矿社区中的不良行为者可以在未来的时间戳上运行赌场支付功能（当前时间是否能被15整除，旷工就可以判断自己是否要打包）

当前区块的时间戳必须严格大于上一个区块的时间戳，但唯一的保证是它将位于规范链中两个连续区块的时间戳之间（目前是15s）。

例如：随机数问题给了节点可操作性。

解决：

除非知道你做什么,对于节点修改时间损失问题在可控范围内。

3，未初始化存储指针

像动态数组,struct,mapping 这样的复杂数据结构是不能直接在栈里存储的，因为栈只能保存单独的"子"，也就是只能保存实际数据长度小于等于 32 的简单数据类型。所以在智能合约函数中声明动态数组和 struct 时，必须明确指明其位置在 storage 还是 memory 中。

```
pragma solidity =0.4.23;

contract testContract{

    bytes32 public name;

    address public mappedAddress;

    struct Person {

        bytes32 name;

        address mappedAddress;

    }

    function test(bytes32 _name , address  _mappedAddress) public{

        Person person;

        person.name = _name;

        person.mappedAddress = _mappedAddress;

    }

}
```

坑点分析：在智能合约函数中声明了临时的动态数组或者 struct，而没有指定位置，且没有进行初始化，那么这些变量默认成为存储指针。

解决：

高版本编译器已经可以检测这种问题了。

增加 memory 修饰就行了。