此问卷的8-27题描述了20种代码异味。假设某个智能合约已经存在题目中所描述的代码异味，请判断移除这个代码异味，是否会对智能合约的代码设计、安全性、可读性中的某个方面有一定的帮助。（代码异味是指程序中不好的模式，它可能会造成潜在的漏洞，或者造成代码的可读性下降）

*** 1. 您是否是专业的智能合约开发人员**

- ○ 是
- ○ 否

*** 2. 您是否参加过开源项目**

- ○ 是
- ○ 否

*** 3. 请简述您在智能合约开发的工作职责**

- ○ 测试
- ○ 开发
- ○ 管理
- ○ 其他 _____ *

*** 4. 您从事智能合约相关的工作多少年?**

* 5. 您当前的居住国是?

* 6. 您的最高学历是?

- ○ 高中
- ○ 本科
- ○ 硕士
- ○ 博士
- ○ 其他 _____ *

* 7. 专业技能（请对以下各项进行评分）

	非常同意	同意	中立	不同意	非常不同意	无法回答/无法理解问题
您能够高效地开发智能合约（比如能够在短时间内按需求编写智能合约）	○	○	○	○	○	○
您能够开发高性能的智能合约（智能合约开发过程中考虑过程序的复杂度，如何消耗更少的内存和Gas）	○	○	○	○	○	○
您能够在开发过程过避免设计上的缺陷	○	○	○	○	○	○
您能够重构代码以消除合约中存在的结构缺陷	○	○	○	○	○	○
您能够复用智能合约的代码	○	○	○	○	○	○
您了解多种区块链的不同（如比特币，以太坊等）	○	○	○	○	○	○

* 8. 未检查的外部调用(Unchecked External Call): 以太坊提供了一些地址的底层调用函数，如address.call(), address.send(), address.delegatecall()。然而出于某些原因，这些函数可能会调用失败。当调用失败时，这些函数会返回false，但是不会抛出异常。因此在调用这些函数后，需要对返回值做一个检查。

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 9. 来自外部的DoS攻击(Dos Under External Influence)：当一个异常被合约检测到时，整个transaction将会被回滚。如果导致这个问题的异常没有被修复，则会造成DOS(拒绝访问)攻击。比方说下面这个例子，members是一个数组，保存了要发送的账户的地址，而其中一个地址是攻击者的合约账户。由于transfer对回调函数的2300 gas限制，transfer总会抛出异常，导致这个函数无法被正常执行。

```
for(var i = 0;i < members.length; i++){
    if(this.balance > 0.1 ether)
        members[i].transfer(0.1 ether);
}
```

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 10. Balance恒等逻辑(Strict Balance Equality)：在以太坊中，合约是没有办法拒绝Ether的。哪怕是在fallback函数中直接throw，攻击者也可以用selfdestruct函数强行发送Ether。因此，涉及Balance恒等的逻辑就很可能出问题。如下函数，只有在账户余额等于10eth的时候，才会触发doingSomething()函数。但是攻击者可以调用selfdestruct()函数，发1wei给该合约，使其永远无法等于10eth。

```
function Demo(){
    if(this.balance == 10 eth)
        doingSomething();
}
```

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 11. 整数溢出(Unmatched Type Assignment)：以太坊提供了很多类型的整型变量，如uint8,uint16,uint256。uint8只支持[0,2^8)，当超过这个范围后，将会导致程序出错。比如for(var i = 0;i < member.length; i++)中，i的类型被赋值为了uint8，当数组member的长度超过256后，这个循环将会变为死循环。因为对uint8来说，255+1会变为0.

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 12. tx.origin权限依赖(Transaction State Dependency)：智能合约经常需要检查访问者的权限，比如判断msg.sender是否为管理员地址。但是如果用tx.origin来判断访问者权限，tx.origin得到的就会是启动交易的原始地址，它仍然会是所有者的地址，从而攻击者可以轻松地绕过权限判断。

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 13. 区块信息依赖(Block Info Dependency)：智能合约经常需要创建随机数，而很多合约会使用区块信息来生成随机数，如block.blockhash, block.timestamp。然而，矿工能够在一定程度上操控区块信息，比如当使用block.timestamp获得随机数时，矿工大概能修改900s左右的时间，从而影响随机数生成的结果。

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 14. 重入攻击(Re-entrancy)：重入攻击是以太坊的著名攻击，下面的合约是一个重入攻击的例子。Attacker合约在reentrancy函数中调用了Victim合约的withDraw函数。正常情况下，withDraw函数会给Attacker合约发送ether后，清零账户。然后这两个过程是分开执行的。当withDraw函数给Attacker发钱时，会调用Attacker的回调函数，回调函数会再一次执行该过程，直至Victim账户余额被盗光。

```
contract Victim {
    mapping(address => uint) public userBalannce;
    ...
    function withDraw(){
        uint amount = userBalannce[msg.sender];
        if(amount > 0){
            msg.sender.call.value(amount)();
            userBalannce[msg.sender] = 0;
        }
    }
    ...
}
contract Attacker{
    ...
    function() payable{
        Victim(msg.sender).withDraw();
    }
    function reentrancy(address addr){
        Victim(addr).withDraw();}
    ...
}
```

- ○ 非常重要
- ○ 重要
- ○ 中立
- ○ 不重要
- ○ 非常不重要
- ○ 我没理解(请简述原因) _____ *

* 15. 地址硬编码(Hard Code Address)：由于智能合约一经部署就无法修改的特性，直接把地址写入到智能合约中可能会出现一些问题。第一个问题是地址不符合规定，比如地址写错，或者不符合EIP55规范。第二个问题就是调用了selfdestruct的合约。比方说下面的函数，如果addr的地址被直接写入在合约中，那么当addr对应的合约调用selfdestruct函数后，下述函数将失去作用，因为将Ether发送到一个调用selfdestruct的合约，Ether会永远丢失。

```
function withDraw(address addr, uint amount){
    addr.call.value(amount);
}
```

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 16. 循环调用(Nest Call)：以太坊的CALL指令消耗的gas量非常高。比方说一个非0值的transfer调用的CALL指令会消耗9000gas。 如果在一个循环中大量使用CALL指令，很容易被外部攻击，引发out of gas错误。比方下述函数，member数量如果可以无限制增加，则有可能被外部攻击，引起out of gas error。

```
for(uint i = 0; i < member.length; i++){
    member[i].send(1 wei);
}
```

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 17. 使用不推荐的API(Deprecated APIs)：EVM的部分指令会由于硬分叉而被弃用或者被修改，同时Solidity的文档中经常也会弃用某些写法和API。比如说目前版本的Solidity建议用revert替代throw，用selfdestruct替代suicide，并且不建议使用CALLCODE。为了防止未来代码复用时出现无法预见的问题以及提高代码可读性，尽可能避免使用这些不被推荐的API。

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 18. 未固定的编译版本(Unfixed Compiler Version)：Solidity的版本迭代非常快，不同的版本之间可能会存在一定的差异。为了将来能够更好的复用代码，请在合约中指明唯一可使用的代码版本。比如pragma solidity 0.4.25;说明该合约只支持0.4.25，而pragma solidity ^0.4.25;这种写法则不被推荐，因为它支持0.4.25及以上，5.0以下的所有版本

- ◯ 非常重要
- ◯ 重要
- ◯ 中立
- ◯ 不重要
- ◯ 非常不重要
- ◯ 我没理解(请简述原因) _____ *

* 19. 令人困惑的数据类型(Misleading Data Location)： 传统编程语言，如Java，C中，定义在函数内部的变量都是局部变量，他们的生命周期通常随着函数的结束而结束。但是在Solidity中，struct，mapping，arrays这些数据类型默认都是storage变量，而且EVM是不支持动态分配storage变量的。因此在Solidity函数内部定义这三种类型时，却不指定memory类型，会使程序出现bug。
下面这个合约创建了局部变量tmp，它默认的类型是storage，但是EVM无法动态分配空间给它，因此它就被绑定在slot 0的storage变量上，也就是variable。每调用一次reAssignArray()函数，variable变量就会+1，因此这种写法会造成程序的bug

```
contract Demo{
    uint variable;
    uint[] investList;
    function reAssignArray(){
        uint[] tmp;
        tmp.push(0);
        investList = tmp;
    }
}
```

- ◯ 非常重要
- ◯ 重要
- ◯ 中立
- ◯ 不重要
- ◯ 非常不重要
- ◯ 我没理解(请简述原因) _____ *

* 20. 无用的变量(Unused Statement)：虽然在函数内部增加一些无用的变量并不会影响到程序的执行，但是会对代码可读性造成一定的影响。因此，请尽可能地移除合约中的无用变量。比如下面的合约中，newValue就是一个完全无用的变量，移除它可以增强代码可读性。

```
contract Demo{
    bool state = false;
    function changeState(bool newState, uint value){
        uint newValue = value;
        state = newState;
    }
}
```

- ◯ 非常重要
- ◯ 重要
- ◯ 中立
- ◯ 不重要
- ◯ 非常不重要
- ◯ 我没理解(请简述原因) _____ *

* 21. 不符合标准的ERC20合约(Unmatched ERC-20 Standard)：ERC20合约是一种普遍存在在以太坊上的智能合约，但是目前很多合约不符合规范。比方说ERC20标准定义了9个标准函数和2个Event，然而某些合约会漏掉部分函数，或者修改函数的返回值类型

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 22. 函数缺少返回值(Missing Return Statement)：某些智能合约的函数定义了返回值，却在函数结尾没有返回。对于这种函数，EVM虚拟机会自动添加一个默认的返回值。比如下面函数定义了bool返回值，但是在实现时没有添加返回值。EVM会自动默认返回值类型为false。虽然不影响功能使用，但是当外部程序调用这个函数值时，外部程序会永远收到false的返回值，这有可能会造成外部程序逻辑判断错误

```
function test(address addr) returns(bool){
    addr.transfer(0.1 ether);
}
```

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 23. 缺少中断器(Missing Interrupter)：由于区块链的特性，智能合约一旦部署到区块链上就无法修改，而bug是很难避免的。为了应对未来的紧急情况，在代码中增加中断器可以减少损失。比如在合约中增加selfdestruct函数，当出现bug时，可以销毁合约，并且取回所有Ether，待bug修复后，重新部署修复的合约到区块链上。

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 24. 缺少提示器(Missing Reminder)：智能合约被部署到区块链后，就是一串Bytecode，没有源码信息，外部程序通过ABI与合约交互。添加一些提示器，如抛出一个事件(event)，可以让外部程序更好地与合约进行交互，以减少不必要的错误。

- 非常重要
- 重要
- 中立
- 不重要
- 非常不重要
- 我没理解(请简述原因) _____ *

* 25. 贪婪合约(Greedy Contract): 一个合约账户的余额是由合约的代码控制的。如果一个合约没有转账功能，那么它内部的余额将被永远锁在函数中。我们认为一个合约有收钱的功能，却没有转账的功能则为贪婪合约。贪婪合约必须添加转账功能。

- ◯ 非常重要
- ◯ 重要
- ◯ 中立
- ◯ 不重要
- ◯ 非常不重要
- ◯ 我没理解(请简述原因) _____ *

* 26. 高Gas消耗的函数类型(High Gas Consumption Function Type): 对于public类型的函数，EVM会把函数的参数拷贝到memory中处理。对于external函数，EVM则是把参数放在stack中处理。EVM对stack的操作所需要的gas远远小于操作memory消耗的gas。因此如果一个函数能够被写成external，请不要用public。下面两个函数的唯一区别是public和external。然而highGas函数消耗了496gas，而lowGas则只需要261gas.

```
function highGas(uint[20] a) public returns (uint){
    return a[10]*2;
}
function lowGas(uint[20] a) external returns (uint){
    return a[10]*2;
}
```

- ◯ 非常重要
- ◯ 重要
- ◯ 中立
- ◯ 不重要
- ◯ 非常不重要
- ◯ 我没理解(请简述原因) _____ *

* 27. 高Gas消耗的变量类型(High Gas Consumption Data Type): bytes是以太坊的一个可变长度类型，byte[]的功能和bytes相同，但是bytes使用的gas小于byte[]。因为EVM每次读取32个bytes，byte[]永远占用32 bytes的整数倍，而bytes能更紧密地存储数据。因此用bytes来替换byte[]能节约gas消耗。

- ◯ 非常重要
- ◯ 重要
- ◯ 中立
- ◯ 不重要
- ◯ 非常不重要
- ◯ 我没理解(请简述原因) _____ *

28. 选答：如有选择“非常有用”、“有用”的选项， 请选择1-2种代码异味，简述原因，格式为: 异味名称: 原因 （如果没有该选项的答案，则填无）

29. 选答：如有选择“不是很有用”、“完全没用”的选项， 请选择1-2种代码异味，简述原因，格式为，异味名称: 原因（如果没有该选项的答案，则填无）

30. 选答：如有选择“有时有用”的选项， 请选择1-2种代码异味，简述原因，格式为: 异味名称: 原因 （如果没有该选项的答案，则填无）

31. 如果参与抽奖，请输入邮箱，我们会随机抽取两位参与者，赠送50美元的亚马逊代金券