


智能合约安全科普

A Gentle Introduction to Smart Contract Vulnerabilities

王子彦

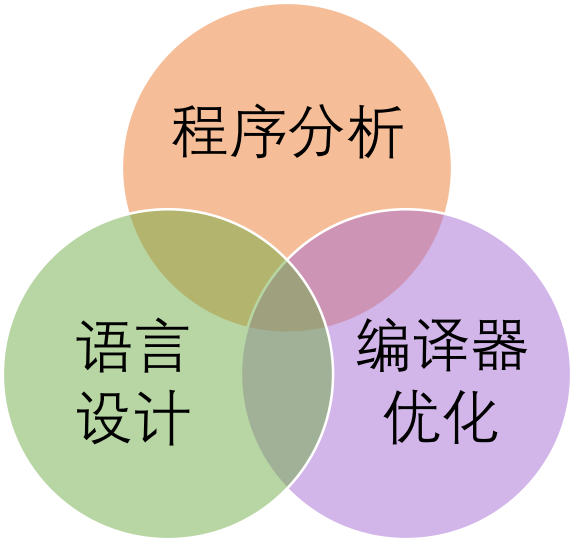
ziyan-wang.github.io

2021/03/30

⚠ 警告 此幻灯片中的代码示例存在安全漏洞，仅供学习，请勿使用 ⚠ WARNING DO NOT USE INSECURE CODE IN THIS SLIDE

王子彦

- 20级专硕
- 陈湘萍老师小组

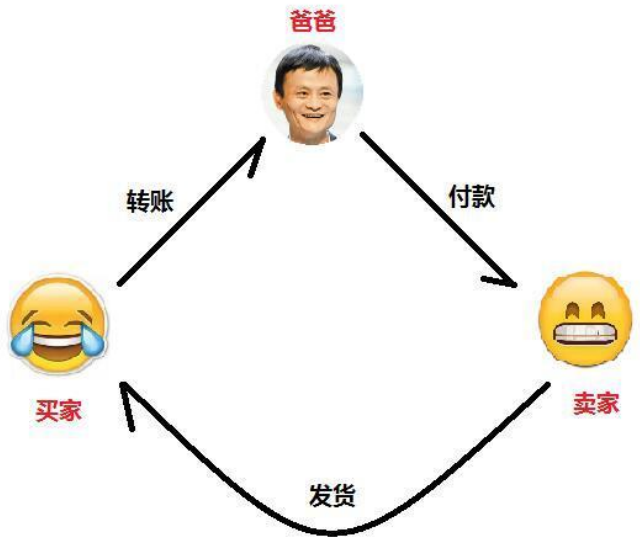


目录

目录

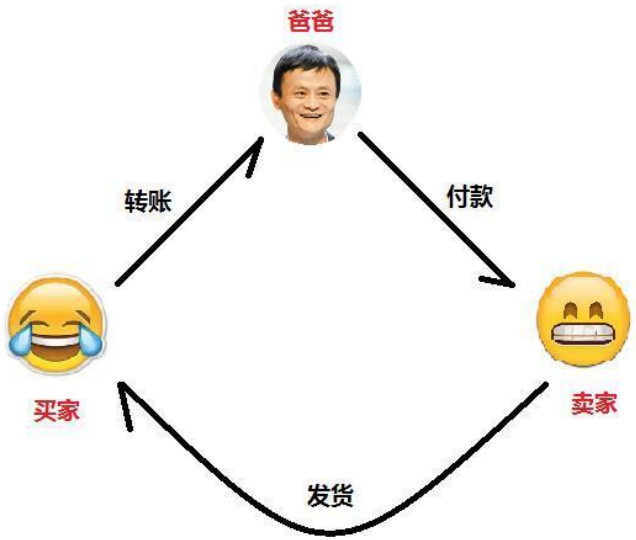
区块链[8][9]

区块链[8][9]

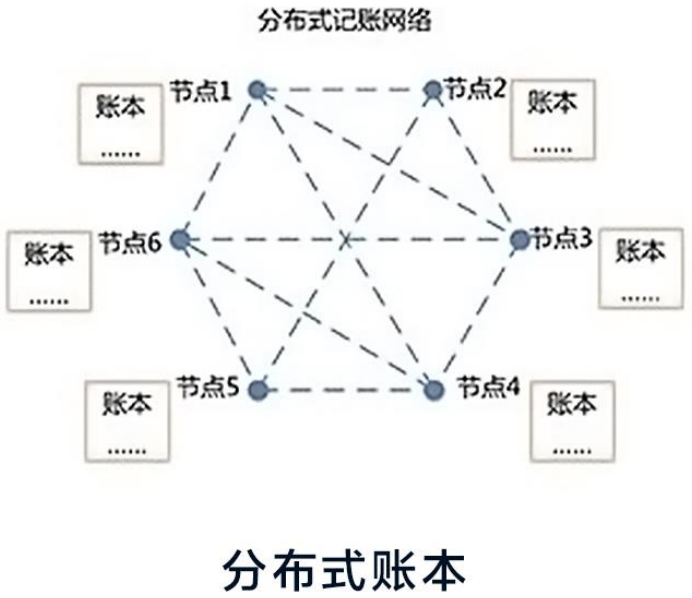


中心化交易

区块链[8][9]



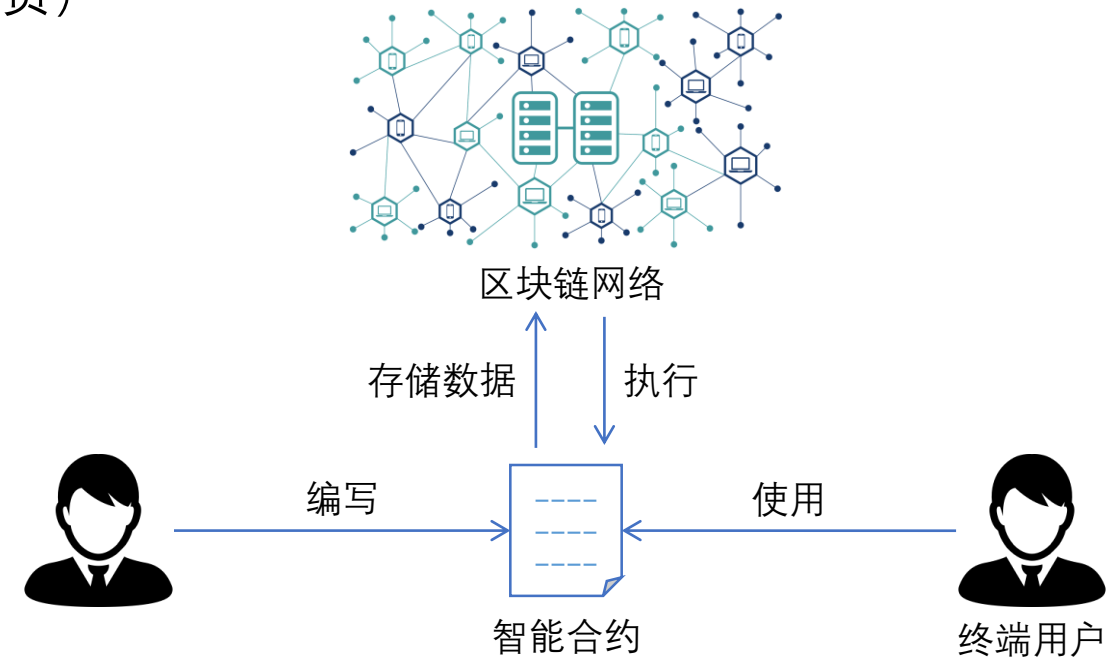
中心化交易



去中心化交易（区块链）

智能合约

- 智能合约是在区块链上执行的代码
- 相同的代码会在区块链网络的所有节点（机器）上几乎同时执行
- 用户在使用智能合约时，需要支付费用；代码消耗的计算资源越多，费用越高（以太坊使用gas作为单位来计费）



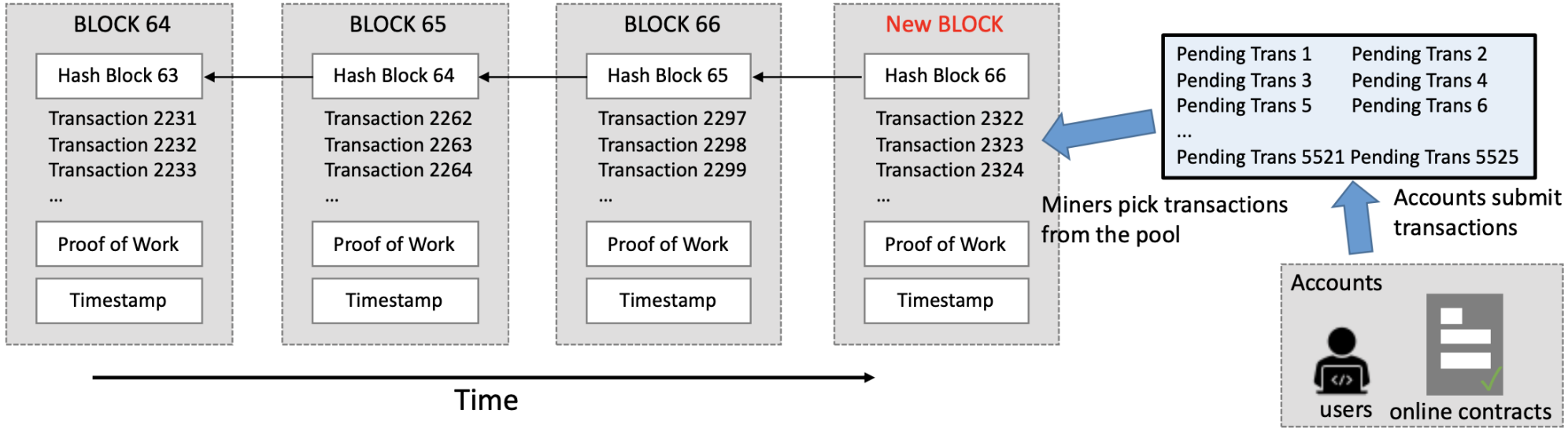
智能合约举例：银行

```
1   contract Bank {
2       mapping(address => uint) balances;   账户余额表
3
4       function deposit() {    存钱函数
5           balances[msg.sender] += msg.value;
6       }
7
8       function withdraw() {    取钱函数
9           uint balance = balances[msg.sender];
10          msg.sender.call.value(balance)();
11          balances[msg.sender] = 0;
12      }
13
14      function() {   fallback函数
15      }
16  }
```

交易打包区块[1]

- 用户调用智能合约的函数时，将产生一笔交易
- 交易会提交到交易池中
- 矿工的机器执行交易，并打包到新区块里
- 矿工可决定要执行哪些交易，以及执行顺序

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function deposit() {
5           balances[msg.sender] += msg.value;
6       }
7
8       function withdraw() {
9           uint balance = balances[msg.sender];
10          msg.sender.call.value(balance)();
11          balances[msg.sender] = 0;
12      }
13
14      function() {
15      }
16  }
```



目录

智能合约的安全漏洞（一）

重入[3]

- 可重入：函数执行在任意时刻被中断，然后操作系统调度执行另外一段代码，这段代码又调用了该函数，不会出错

智能合约的安全漏洞（一）

重入[3]

- 可重入：函数执行在任意时刻被中断，然后操作系统调度执行另外一段代码，这段代码又调用了该函数，不会出错

```
1   int a = 1;
2
3   int f()
4   {
5       a += 1;
6       return a;
7   }
8
9   int g()
10  {
11      return f();
12  }
```

智能合约的安全漏洞（一）

重入[3]

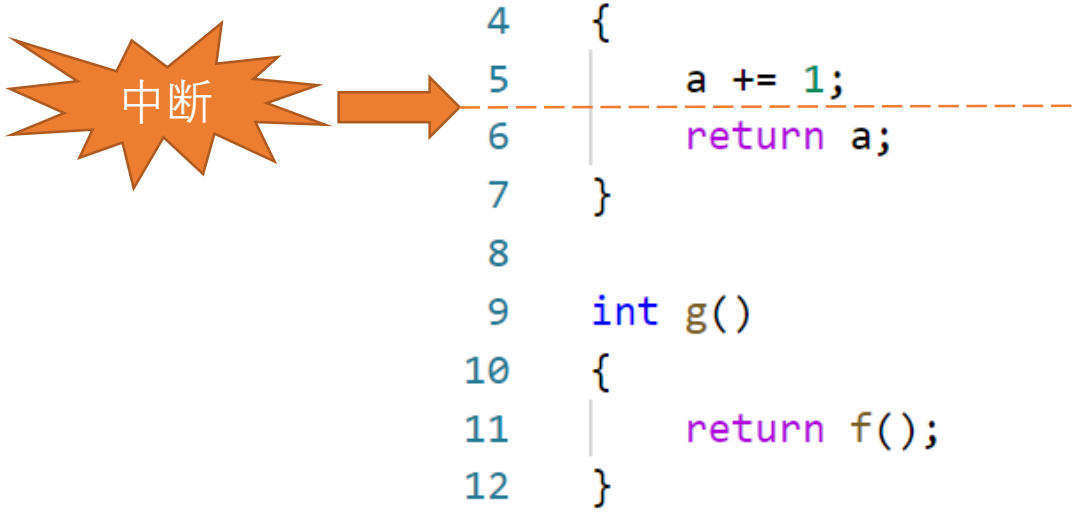
- 可重入：函数执行在任意时刻被中断，然后操作系统调度执行另外一段代码，这段代码又调用了该函数，不会出错

```
1   int a = 1;
2
3   int f()
4   {
5       a += 1;
6       return a;
7   }
8
9   int g()
10  {
11      return f();
12  }
```

智能合约的安全漏洞（一）

重入[3]

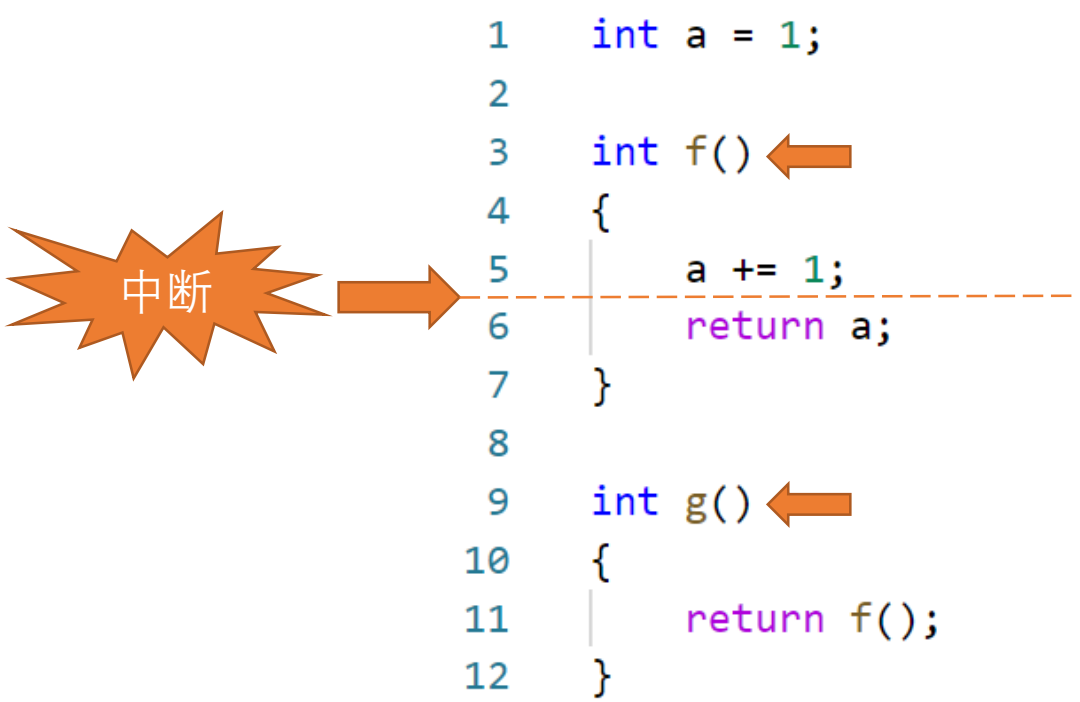
- 可重入：函数执行在任意时刻被中断，然后操作系统调度执行另外一段代码，这段代码又调用了该函数，不会出错



智能合约的安全漏洞（一）

重入[3]

- 可重入：函数执行在任意时刻被中断，然后操作系统调度执行另外一段代码，这段代码又调用了该函数，不会出错



智能合约的安全漏洞（一）

重入

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A

智能合约的安全漏洞（一）

重入

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           msg.sender.call.value(balance)();
7           balances[msg.sender] = 0;
8       }
9   }
10
11  contract Attacker {
12      Bank bank;
13      function attack() { bank.withdraw(); }
14      function() { bank.withdraw(); }
15  }
```

智能合约的安全漏洞（一）

重入

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A

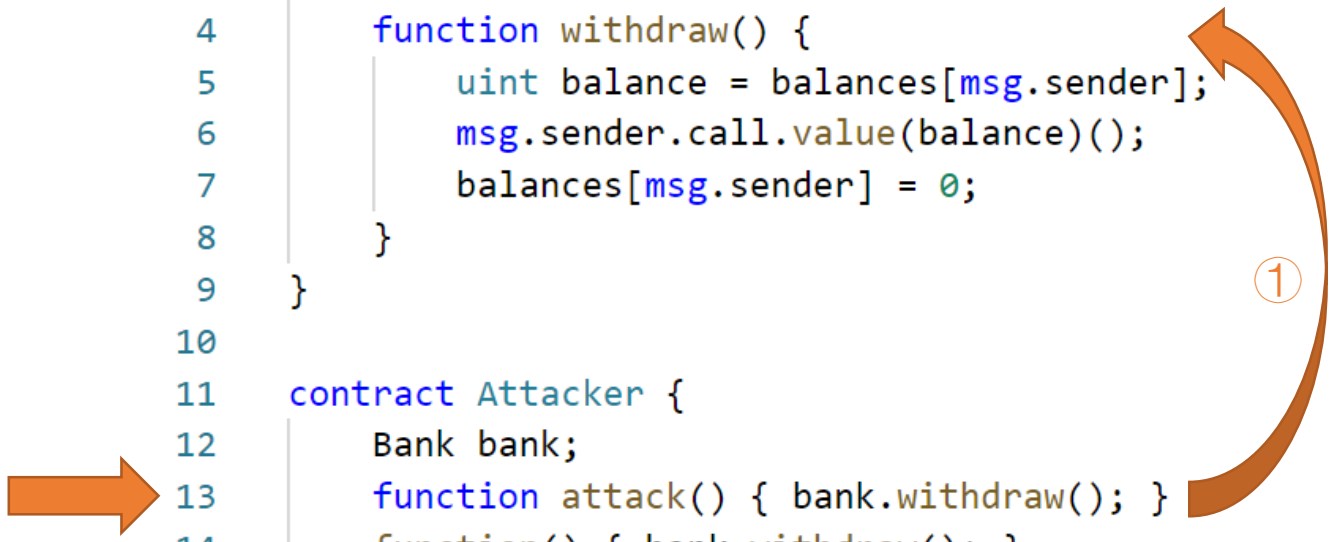
```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           msg.sender.call.value(balance)();
7           balances[msg.sender] = 0;
8       }
9   }
10
11  contract Attacker {
12      Bank bank;
13      function attack() { bank.withdraw(); }
14      function() { bank.withdraw(); }
15  }
```

智能合约的安全漏洞（一）

重入

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A

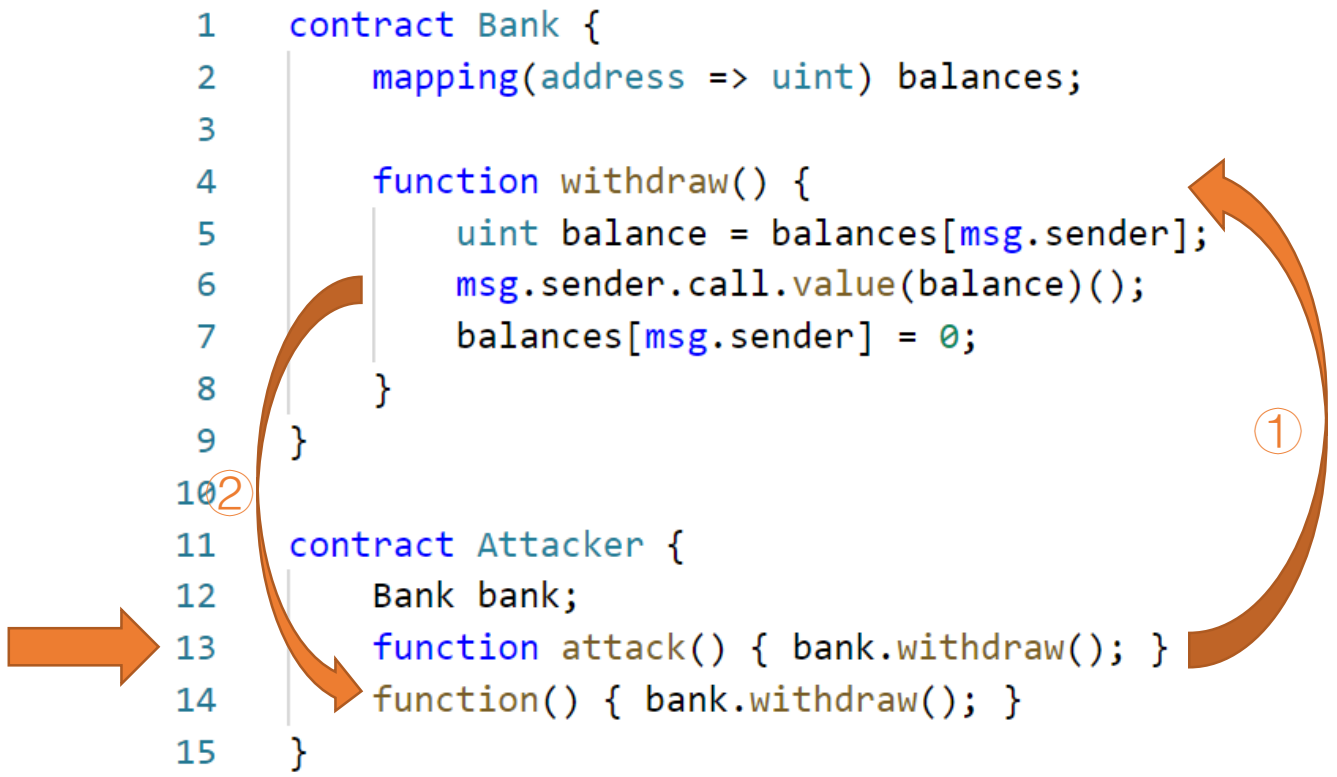
```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           msg.sender.call.value(balance)();
7           balances[msg.sender] = 0;
8       }
9   }
10
11  contract Attacker {
12      Bank bank;
13      function attack() { bank.withdraw(); }
14      function() { bank.withdraw(); }
15  }
```



智能合约的安全漏洞（一）

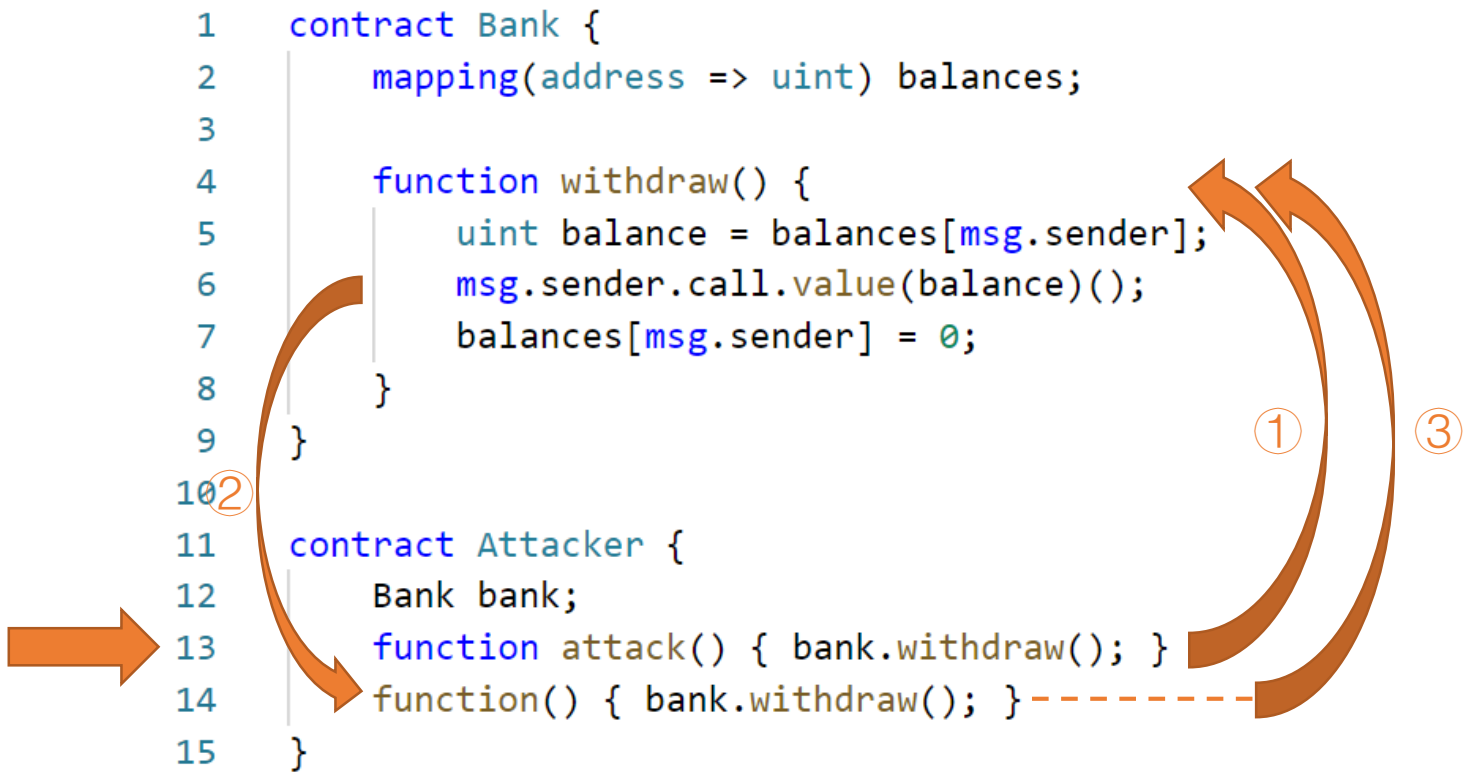
重入

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A



重入

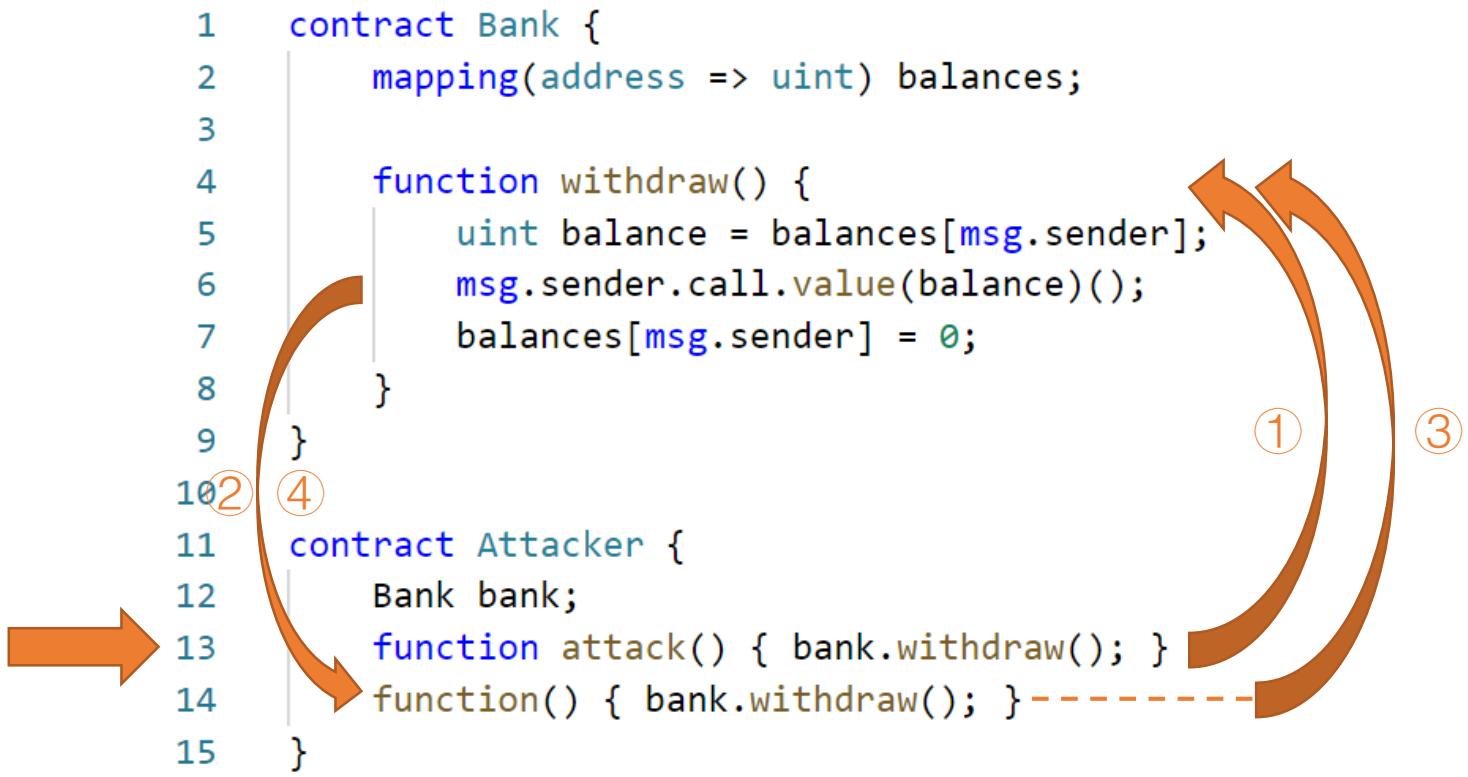
- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A



智能合约的安全漏洞（一）

重入

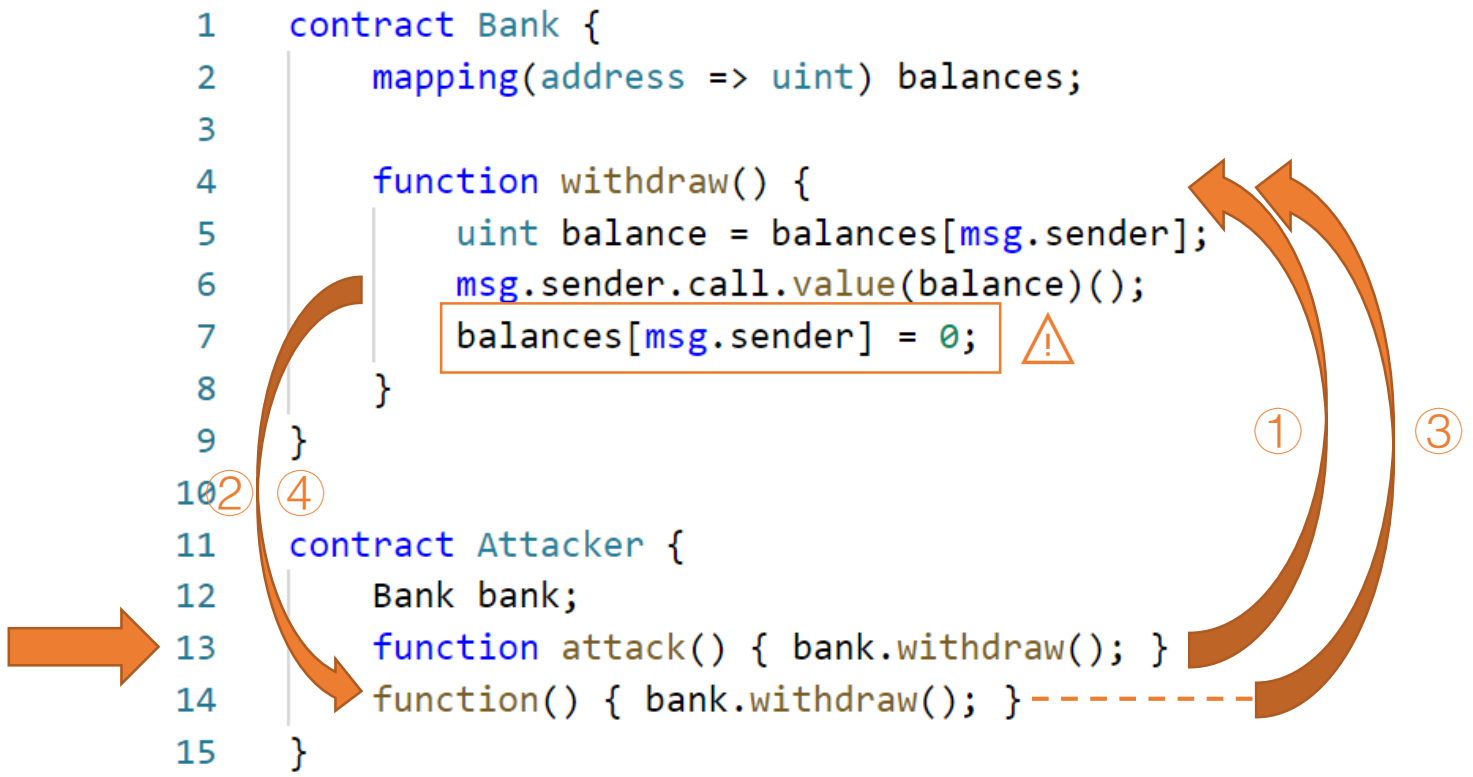
- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A



智能合约的安全漏洞（一）

重入

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A



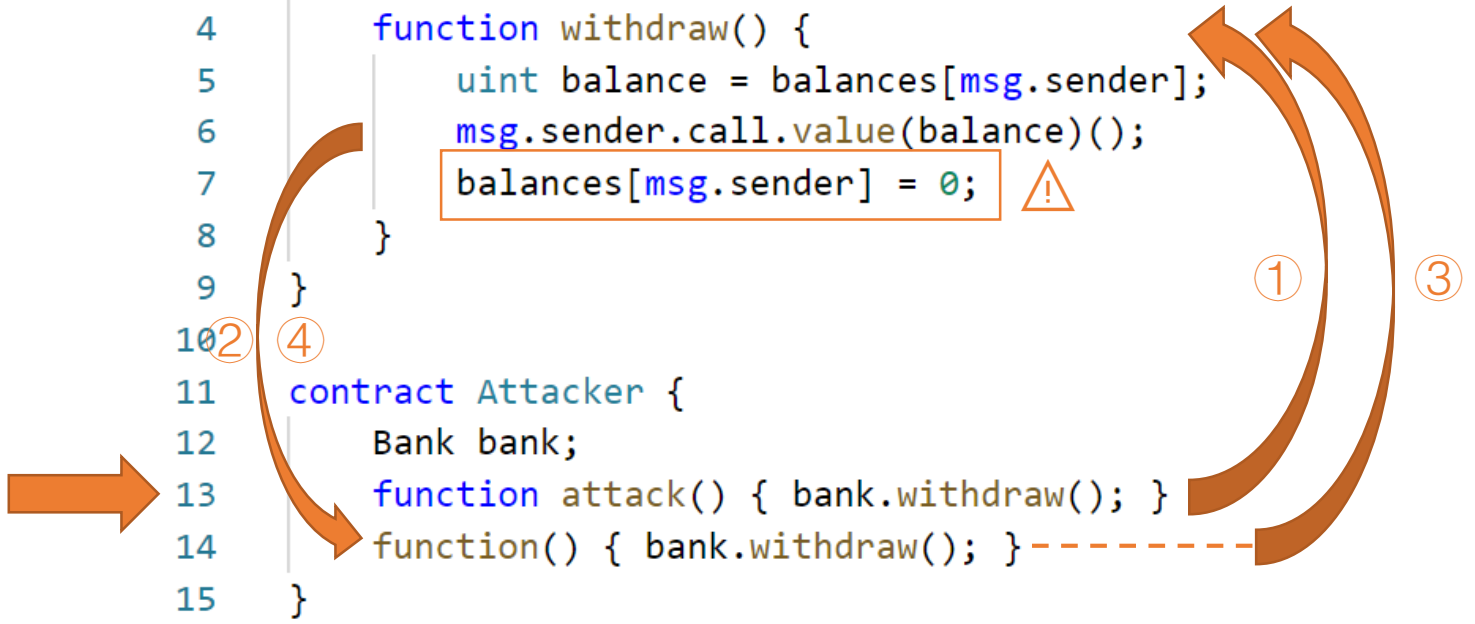
智能合约的安全漏洞（一）

重入

通过观察可发现：
传统程序的重入 -> 计算逻辑错误😐
智能合约的重入 -> 偷钱😱

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           msg.sender.call.value(balance)();
7           balances[msg.sender] = 0;
8       }
9   }
10
11  contract Attacker {
12      Bank bank;
13      function attack() { bank.withdraw(); }
14      function() { bank.withdraw(); }
15  }
```



智能合约的安全漏洞（一）

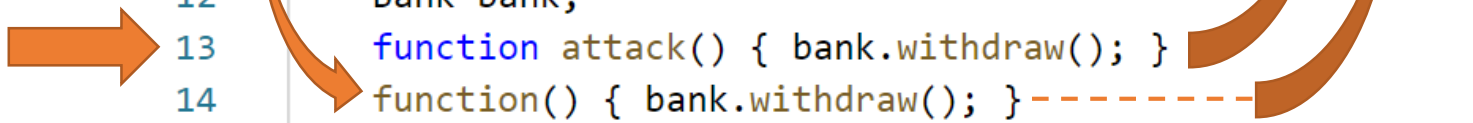
重入

通过观察可发现：
传统程序的重入 -> 计算逻辑错误😐
智能合约的重入 -> 偷钱😱

- 智能合约中的重入：A合约调用B合约时，会将控制权转移给B，B可以反过来调用A

在移交控制权之前，
应先完成对状态的修改

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           msg.sender.call.value(balance)();
7           balances[msg.sender] = 0;
8       }
9   }
10
11  contract Attacker {
12      Bank bank;
13      function attack() { bank.withdraw(); }
14      function() { bank.withdraw(); }
15  }
```



智能合约的安全漏洞（一）

重入

- 应对措施一：先做完所有的内部工作，再调用外部函数

智能合约的安全漏洞（一）

重入

- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

智能合约的安全漏洞（一）

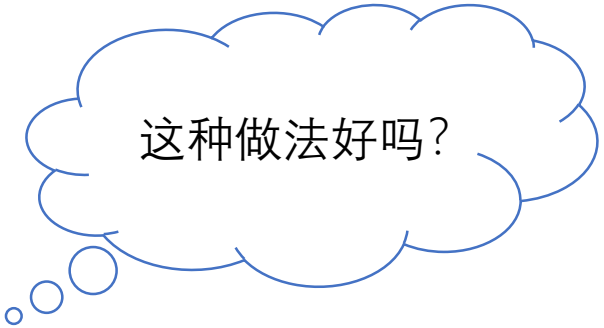
重入

- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

智能合约的安全漏洞（一）

重入

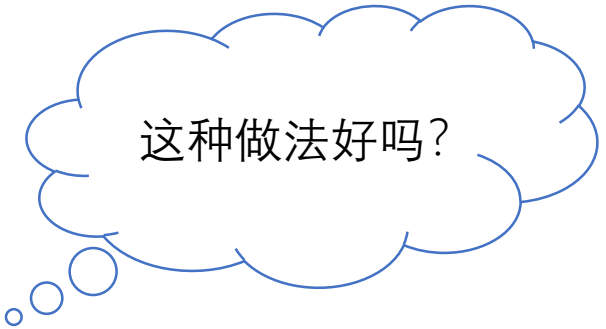


- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

智能合约的安全漏洞（一）

重入



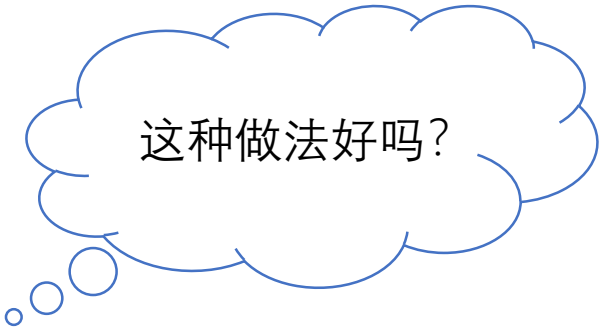
- 应对措施一：先做完所有的内部工作，再调用外部函数

更复杂的情况 → 需求：实现一个只能调用一次的取钱函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

智能合约的安全漏洞（一）

重入



- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

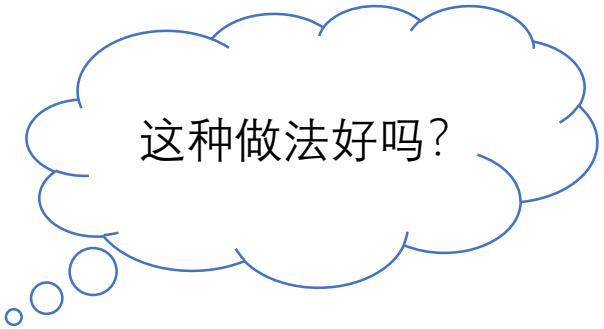
更复杂的情况

需求：实现一个只能调用一次的取钱函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool done = false;
4
5       function withdraw() {
6           uint balance = balances[msg.sender];
7           balances[msg.sender] = 0;
8           (bool success, ) = msg.sender.call.value(balance)();
9           require(success);
10      }
11      function withdrawOnce() {
12          if (done) { return; }
13          withdraw();
14          done = true;
15      }
16  }
```

智能合约的安全漏洞（一）

重入



- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

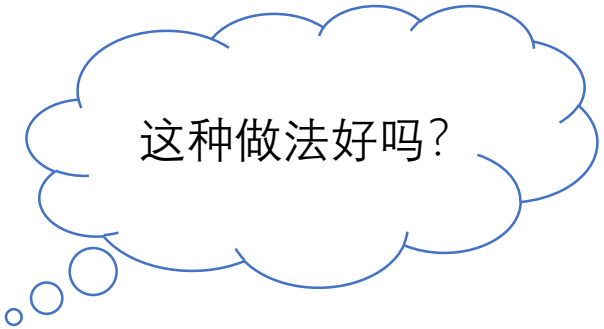
更复杂的情况

需求：实现一个只能调用一次的取钱函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool done = false;
4
5       function withdraw() {
6           uint balance = balances[msg.sender];
7           balances[msg.sender] = 0;
8           (bool success, ) = msg.sender.call.value(balance)();
9           require(success);
10      }
11      function withdrawOnce() {
12          if (done) { return; }
13          withdraw();
14          done = true;
15      }
16  }
```

智能合约的安全漏洞（一）

重入



- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

更复杂的情况

需求：实现一个只能调用一次的取钱函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool done = false;
4
5       function withdraw() {
6           uint balance = balances[msg.sender];
7           balances[msg.sender] = 0;
8           (bool success, ) = msg.sender.call.value(balance)();
9           require(success);
10      }
11      function withdrawOnce() {
12          if (done) { return; }
13          withdraw();
14          done = true;
15      }
16  }
```

也需要将该函数视为不安全的“外部函数”

智能合约的安全漏洞（一）

重入

这种做法好吗？

存在风险：多层函数调用造成外部调用被隐藏

- 应对措施一：先做完所有的内部工作，再调用外部函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

更复杂的情况

需求：实现一个只能调用一次的取钱函数

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool done = false;
4
5       function withdraw() {
6           uint balance = balances[msg.sender];
7           balances[msg.sender] = 0;
8           (bool success, ) = msg.sender.call.value(balance)();
9           require(success);
10      }
11      function withdrawOnce() {
12          if (done) { return; }
13          withdraw();
14          done = true;
15      }
16  }
```

也需要将该函数视为不安全的“外部函数”

智能合约的安全漏洞（一）

重入[4]

- 应对措施二：互斥锁

智能合约的安全漏洞（一）

重入[4]

- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

智能合约的安全漏洞（一）

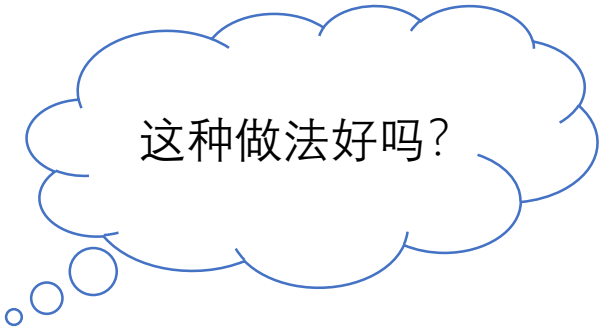
重入[4]

- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

智能合约的安全漏洞（一）

重入[4]

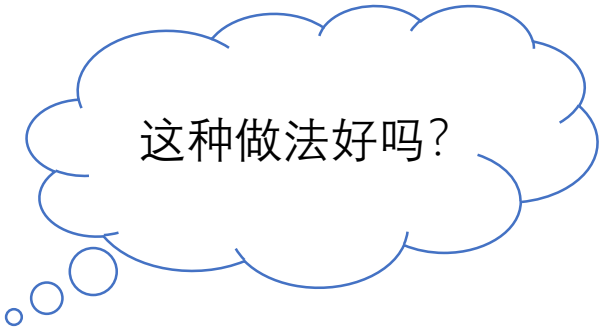


- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

智能合约的安全漏洞（一）

重入[4]



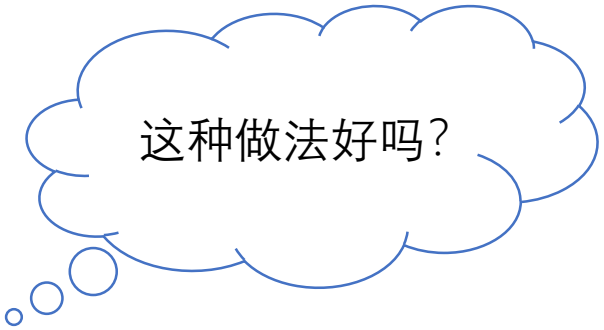
- 应对措施二：互斥锁

更复杂的情况 → 完成某个操作，需要跨合约交互

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

智能合约的安全漏洞（一）

重入[4]



- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

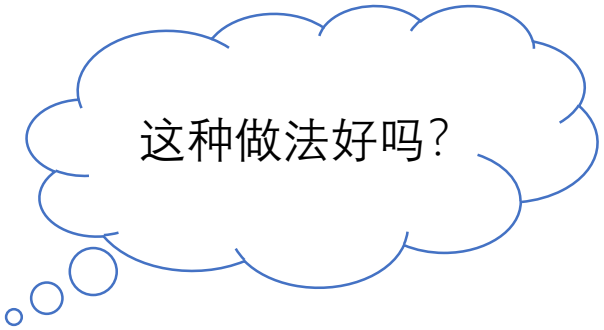
更复杂的情况

完成某个操作，需要跨合约交互

```
1   contract StateHolder {
2       uint private n;
3       address private lockHolder;
4
5       function getLock() {
6           require(lockHolder == address(0));
7           lockHolder = msg.sender;
8       }
9
10      function releaseLock() {
11          require(msg.sender == lockHolder);
12          lockHolder = address(0);
13      }
14
15      function set(uint newState) {
16          require(msg.sender == lockHolder);
17          n = newState;
18      }
19  }
```

智能合约的安全漏洞（一）

重入[4]



- 应对措施二：互斥锁

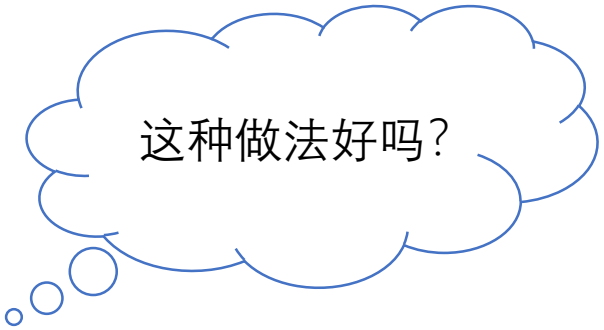
更复杂的情况 → 完成某个操作，需要跨合约交互

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

```
1   contract StateHolder {
2       uint private n;
3       address private lockHolder;
4
5       function getLock() {
6           require(lockHolder == address(0));
7           lockHolder = msg.sender;
8       }
9
10      function releaseLock() {
11          require(msg.sender == lockHolder);
12          lockHolder = address(0);
13      }
14
15      function set(uint newState) {
16          require(msg.sender == lockHolder);
17          n = newState;
18      }
19  }
```

智能合约的安全漏洞（一）

重入[4]



- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

更复杂的情况 → 完成某个操作，需要跨合约交互

```
1   contract StateHolder {
2       uint private n;
3       address private lockHolder;
4
5       function getLock() {                    ①
6           require(lockHolder == address(0));
7           lockHolder = msg.sender;
8       }
9
10      function releaseLock() {
11          require(msg.sender == lockHolder);
12          lockHolder = address(0);
13      }
14
15      function set(uint newState) {           ②
16          require(msg.sender == lockHolder);
17          n = newState;
18      }
19  }
```

智能合约的安全漏洞（一）

重入[4]

这种做法好吗？

- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

更复杂的情况

完成某个操作，需要跨合约交互

```
1   contract StateHolder {
2       uint private n;
3       address private lockHolder;
4
5       function getLock() {
6           require(lockHolder == address(0));
7           lockHolder = msg.sender;
8       }
9
10      function releaseLock() {
11          require(msg.sender == lockHolder);
12          lockHolder = address(0);
13      }
14
15      function set(uint newState) {
16          require(msg.sender == lockHolder);
17          n = newState;
18      }
19  }
```

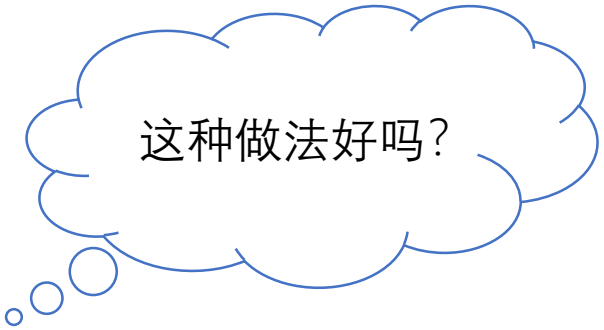
①

③

②

智能合约的安全漏洞（一）

重入[4]



- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

更复杂的情况 → 完成某个操作，需要跨合约交互

```
1   contract StateHolder {
2       uint private n;
3       address private lockHolder;
4
5       function getLock() {                    ①
6           require(lockHolder == address(0));
7           lockHolder = msg.sender;
8       }
9
10      function releaseLock() {                ③ ⚠
11          require(msg.sender == lockHolder);
12          lockHolder = address(0);
13      }
14
15      function set(uint newState) {           ②
16          require(msg.sender == lockHolder);
17          n = newState;
18      }
19  }
```

智能合约的安全漏洞（一）

重入[4]

这种做法好吗？

存在风险：
1．攻击者可以一直不释放锁
2．设计不当可导致死锁/活锁

- 应对措施二：互斥锁

```
1   contract Bank {
2       mapping(address => uint) balances;
3       bool locked;
4
5       function withdraw() {
6           require(locked == false);
7           locked = true;
8           uint balance = balances[msg.sender];
9           (bool success, ) = msg.sender.call.value(balance)();
10          if (success) {
11              balances[msg.sender] = 0;
12          }
13          locked = false;
14      }
15  }
```

更复杂的情况

完成某个操作，需要跨合约交互

```
1   contract StateHolder {
2       uint private n;
3       address private lockHolder;
4
5       function getLock() {
6           require(lockHolder == address(0));
7           lockHolder = msg.sender;
8       }
9
10      function releaseLock() {
11          require(msg.sender == lockHolder);
12          lockHolder = address(0);
13      }
14
15      function set(uint newState) {
16          require(msg.sender == lockHolder);
17          n = newState;
18      }
19  }
```

智能合约的安全漏洞（二）

交易顺序依赖

- 非法抢先交易：在传统金融市场中，股票交易是人工进行的，客户告诉工作人员他要做的交易后，后者可以抢先交易，谋取利润

智能合约的安全漏洞（二）

交易顺序依赖

- 非法抢先交易：在传统金融市场中，股票交易是人工进行的，客户告诉工作人员他要做的交易后，后者可以抢先交易，谋取利润

```
contract PuzzleRewarder {
    address owner;
    uint reward;
    bool solved;

    function setReward(uint amount) {
        if (msg.sender == owner) {
            reward = amount;
        }
    }

    function claimReward(uint answer) {
        if (solved == false && isRight(answer)) {
            msg.sender.transfer(reward);
            solved = true;
        }
    }
}
```

智能合约的安全漏洞（二）

交易顺序依赖

- 非法抢先交易：在传统金融市场中，股票交易是人工进行的，客户告诉工作人员他要做的交易后，后者可以抢先交易，谋取利润

```
1   contract PuzzleRewarder {
2       address owner;
3       uint reward;
4       bool solved;
5
6       function setReward(uint amount) {
7           if (msg.sender == owner) {
8               reward = amount;
9           }
10      }
11
12      function claimReward(uint answer) {
13          if (solved == false && isRight(answer)) {
14              msg.sender.transfer(reward);
15              solved = true;
16          }
17      }
18  }
```

智能合约的安全漏洞（二）

交易顺序依赖

- 非法抢先交易：在传统金融市场中，股票交易是人工进行的，客户告诉工作人员他要做的交易后，后者可以抢先交易，谋取利润

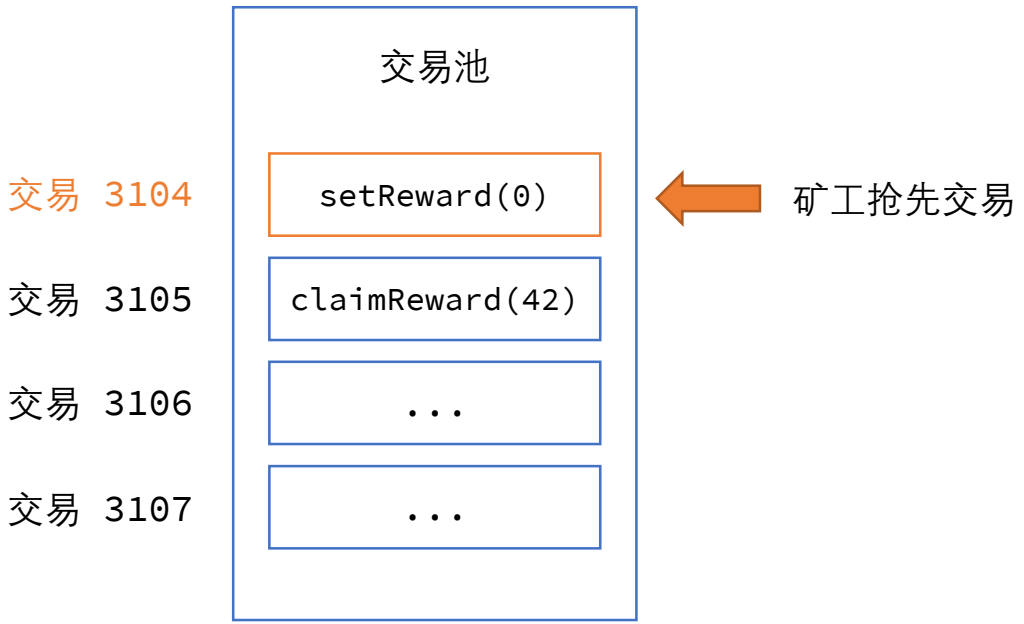
```
1   contract PuzzleRewarder {
2       address owner;
3       uint reward;
4       bool solved;
5
6       function setReward(uint amount) {
7           if (msg.sender == owner) {
8               reward = amount;
9           }
10      }
11
12      function claimReward(uint answer) {
13          if (solved == false && isRight(answer)) {
14              msg.sender.transfer(reward);
15              solved = true;
16          }
17      }
18  }
```

智能合约的安全漏洞（二）

交易顺序依赖

- 非法抢先交易：在传统金融市场中，股票交易是人工进行的，客户告诉工作人员他要做的交易后，后者可以抢先交易，谋取利润

```
1   contract PuzzleRewarder {
2       address owner;
3       uint reward;
4       bool solved;
5
6       function setReward(uint amount) {
7           if (msg.sender == owner) {
8               reward = amount;
9           }
10      }
11
12      function claimReward(uint answer) {
13          if (solved == false && isRight(answer)) {
14              msg.sender.transfer(reward);
15              solved = true;
16          }
17      }
18  }
```

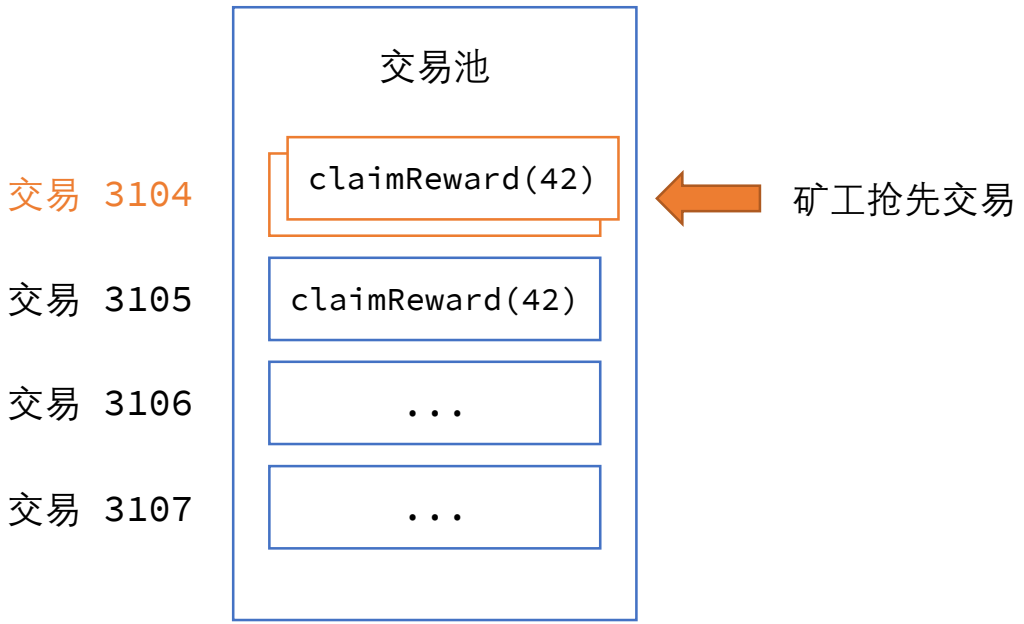


智能合约的安全漏洞（二）

交易顺序依赖

- 非法抢先交易：在传统金融市场中，股票交易是人工进行的，客户告诉工作人员他要做的交易后，后者可以抢先交易，谋取利润

```
1   contract PuzzleRewarder {
2       address owner;
3       uint reward;
4       bool solved;
5
6       function setReward(uint amount) {
7           if (msg.sender == owner) {
8               reward = amount;
9           }
10      }
11
12      function claimReward(uint answer) {
13          if (solved == false && isRight(answer)) {
14              msg.sender.transfer(reward);
15              solved = true;
16          }
17      }
18  }
```



智能合约的安全漏洞（二）

交易顺序依赖

- 应对措施：先提交哈希，再提交原值

智能合约的安全漏洞（二）

交易顺序依赖

- 应对措施：先提交哈希，再提交原值

```
1   contract PuzzleRewarder {
2       mapping(address => bytes) hashes;
3
4       function commit(bytes hashValue) {
5           hashes[msg.sender] == hashValue;
6       }
7
8       function reveal(uint answer) {
9           byte hashValue = hash(answer, msg.sender);
10          if (hashes[msg.sender] == hashValue && isRight(answer)) {
11              msg.sender.transfer(reward);
12          }
13      }
14  }
```

智能合约的安全漏洞（二）

交易顺序依赖

- 应对措施：先提交哈希，再提交原值

```
1   contract PuzzleRewarder {
2       mapping(address => bytes) hashes;
3
4       function commit(bytes hashValue) {
5           hashes[msg.sender] == hashValue;
6       }
7
8       function reveal(uint answer) {
9           byte hashValue = hash(answer, msg.sender);
10          if (hashes[msg.sender] == hashValue && isRight(answer)) {
11              msg.sender.transfer(reward);
12          }
13      }
14  }
```

智能合约的安全漏洞（二）

交易顺序依赖

- 应对措施：先提交哈希，再提交原值

```
1   contract PuzzleRewarder {
2       mapping(address => bytes) hashes;
3
4       function commit(bytes hashValue) {
5           hashes[msg.sender] == hashValue;
6       }
7
8       function reveal(uint answer) {
9           byte hashValue = hash(answer, msg.sender);
10          if (hashes[msg.sender] == hashValue && isRight(answer)) {
11              msg.sender.transfer(reward);
12          }
13      }
14  }
```

智能合约的安全漏洞（三）

系统属性依赖[1]

- 矿工可以操纵时间戳等系统属性

智能合约的安全漏洞（三）

系统属性依赖[1]

- 矿工可以操纵时间戳等系统属性

```
1   contract Gamble {
2       function randomReward(uint bet) {
3           uint t = bet * block.timestamp;
4           if (t > 1000) {
5               msg.sender.send.value(reward)();
6           }
7       }
8   }
```

智能合约的安全漏洞（三）

系统属性依赖[1]

- 矿工可以操纵时间戳等系统属性

```
1   contract Gamble {
2       function randomReward(uint bet) {
3           uint t = bet * block.timestamp;
4           if (t > 1000) {
5               msg.sender.send.value(reward)();
6           }
7       }
8   }
```

智能合约的安全漏洞（三）

系统属性依赖[1]

- 矿工可以操纵时间戳等系统属性

```
1  contract Gamble {
2      function randomReward(uint bet) {
3          uint t = bet * block.timestamp;
4          if (t > 1000) {
5              msg.sender.send.value(reward)();
6          }
7      }
8  }
```

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	GasPrice	The transaction gas price.
1	TxOrigin	The transaction origin account.
2	Coinbase	The the beneficiary address of the block.
3	BlockNum	The block number.
4	Timestamp	The block timestamp.
5	GasLimit	The block gas limit.
6	Difficulty	The block difficulty.

智能合约的安全漏洞（四）

无界批量操作[2]

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest() { // 每人获得5%利息
9           for (uint i = 0; i < accounts.length; i += 1) {
10              accounts[i].balance = accounts[i].balance * 105 / 100;
11          }
12      }
13  }
```

智能合约的安全漏洞（四）

无界批量操作[2]

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest() { // 每人获得5%利息
9           for (uint i = 0; i < accounts.length; i += 1) {
10              accounts[i].balance = accounts[i].balance * 105 / 100;
11          }
12      }
13  }
```

智能合约的安全漏洞（四）

无界批量操作[2]

- 代码中有循环，循环的次数取决于外部输入，且次数无上限

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest() { // 每人获得5%利息
9           for (uint i = 0; i < accounts.length; i += 1) {
10              accounts[i].balance = accounts[i].balance * 105 / 100;
11          }
12      }
13  }
```

智能合约的安全漏洞（四）

无界批量操作[2]

- 代码中有循环，循环的次数取决于外部输入，且次数无上限
- 执行消耗的gas可能超过区块允许的上限，导致DoS（拒绝服务）

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest() { // 每人获得5%利息
9           for (uint i = 0; i < accounts.length; i += 1) {
10              accounts[i].balance = accounts[i].balance * 105 / 100;
11          }
12      }
13  }
```

智能合约的安全漏洞（四）

无界批量操作[2]

- 代码中有循环，循环的次数取决于外部输入，且次数无上限
- 执行消耗的gas可能超过区块允许的上限，导致DoS（拒绝服务）

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest() { // 每人获得5%利息
9           for (uint i = 0; i < accounts.length; i += 1) {
10              accounts[i].balance = accounts[i].balance * 105 / 100;
11          }
12      }
13  }
```

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	
ADDMOD/MULMOD	8	
POP	2	Stack operation
PUSH/DUP/SWAP	3	
JUMP	8	Unconditional jump
CALLDATALOAD	3	Get input data of current environment
MLOAD/MSTORE	3	Memory operation
SLOAD	200	Storage operation
SSTORE	5,000/20,000	
BALANCE	400	Get balance of an account
SHA3	30	Compute Keccak-256 hash

智能合约的安全漏洞（四）

无界批量操作[2]

- 应对措施一：暂停-继续

智能合约的安全漏洞（四）

无界批量操作[2]

- 应对措施一：暂停-继续

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];
    uint nextAccount;

    function applyInterest() {  // 每人获得5%利息
        for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
            accounts[i].balance = accounts[i].balance * 105 / 100;
        }
        nextAccount = i < accounts.length ? i : 0;
    }
}
```

智能合约的安全漏洞（四）

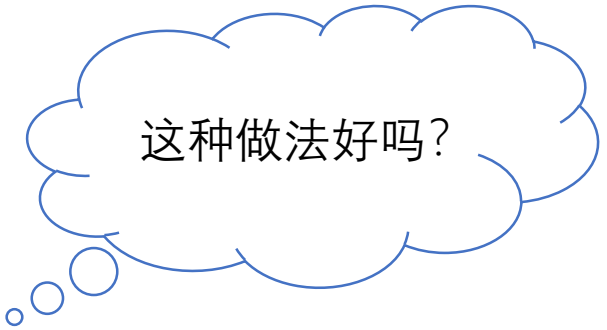
无界批量操作[2]

- 应对措施一：暂停-继续

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7       uint nextAccount;
8
9       function applyInterest() {  // 每人获得5%利息
10          for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
11              accounts[i].balance = accounts[i].balance * 105 / 100;
12          }
13          nextAccount = i < accounts.length ? i : 0;
14      }
15  }
```

智能合约的安全漏洞（四）

无界批量操作[2]

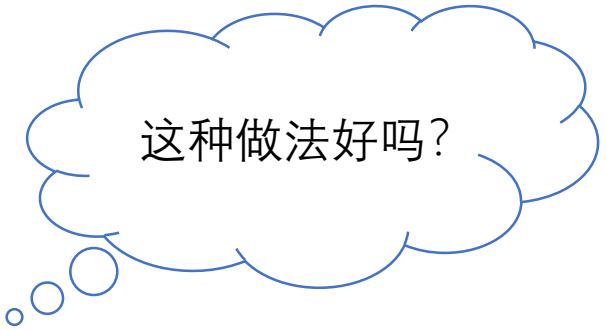


- 应对措施一：暂停-继续

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7       uint nextAccount;
8
9       function applyInterest() {  // 每人获得5%利息
10          for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
11              accounts[i].balance = accounts[i].balance * 105 / 100;
12          }
13          nextAccount = i < accounts.length ? i : 0;
14      }
15  }
```

智能合约的安全漏洞（四）

无界批量操作[2]



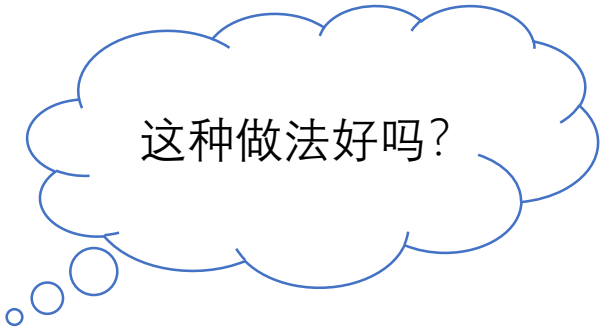
- 应对措施一：暂停-继续

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	
ADDMOD/MULMOD	8	
POP	2	Stack operation
PUSH/DUP/SWAP	3	
JUMP	8	Unconditional jump
CALLDATALOAD	3	Get input data of current environment
MLOAD/MSTORE	3	Memory operation
SLOAD	200	Storage operation
SSTORE	5,000/20,000	
BALANCE	400	Get balance of an account
SHA3	30	Compute Keccak-256 hash

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7       uint nextAccount;
8
9       function applyInterest() {  // 每人获得5%利息
10          for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
11              accounts[i].balance = accounts[i].balance * 105 / 100;
12          }
13          nextAccount = i < accounts.length ? i : 0;
14      }
15  }
```

智能合约的安全漏洞（四）

无界批量操作[2]



存在风险：每个操作消耗的gas并非固定值，未来可能会改变

- 应对措施一：暂停-继续

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7       uint nextAccount;
8
9       function applyInterest() {  // 每人获得5%利息
10          for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
11              accounts[i].balance = accounts[i].balance * 105 / 100;
12          }
13          nextAccount = i < accounts.length ? i : 0;
14      }
15  }
```

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	
ADDMOD/MULMOD	8	
POP	2	Stack operation
PUSH/DUP/SWAP	3	
JUMP	8	Unconditional jump
CALLDATALOAD	3	Get input data of current environment
MLOAD/MSTORE	3	Memory operation
SLOAD	200	Storage operation
SSTORE	5,000/20,000	
BALANCE	400	Get balance of an account
SHA3	30	Compute Keccak-256 hash

智能合约的安全漏洞（四）

无界批量操作

- 应对措施二：分页

智能合约的安全漏洞（四）

无界批量操作

- 应对措施二：分页

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];

    function applyInterest(uint pageSize, uint pageIndex) {  // 每人获得5%利息
        uint begin = pageIndex * pageSize;
        uint end = begin + pageSize;
        for (uint i = begin; i < end; i += 1) {
            accounts[i].balance = accounts[i].balance * 105 / 100;
        }
    }
}
```

智能合约的安全漏洞（四）

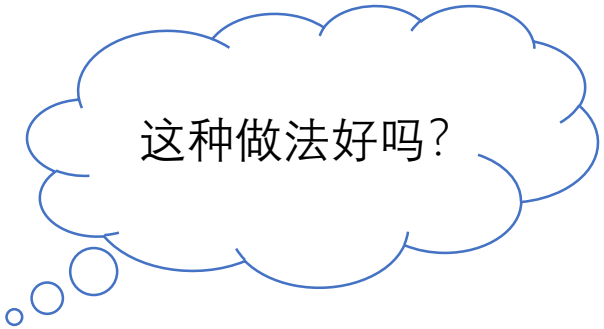
无界批量操作

- 应对措施二：分页

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest(uint pageSize, uint pageIndex) {  // 每人获得5%利息
9           uint begin = pageIndex * pageSize;
10          uint end = begin + pageSize;
11          for (uint i = begin; i < end; i += 1) {
12              accounts[i].balance = accounts[i].balance * 105 / 100;
13          }
14      }
15  }
```

智能合约的安全漏洞（四）

无界批量操作

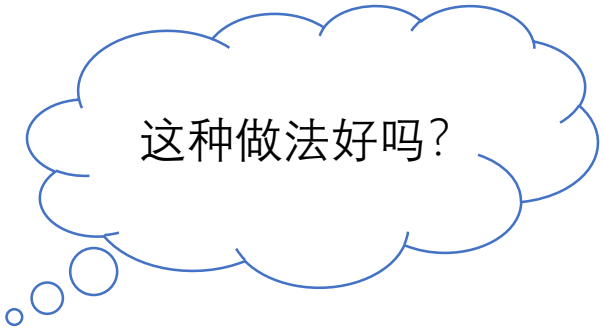


- 应对措施二：分页

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest(uint pageSize, uint pageIndex) {  // 每人获得5%利息
9           uint begin = pageIndex * pageSize;
10          uint end = begin + pageSize;
11          for (uint i = begin; i < end; i += 1) {
12              accounts[i].balance = accounts[i].balance * 105 / 100;
13          }
14      }
15  }
```

智能合约的安全漏洞（四）

无界批量操作



存在风险：无法确保原子性

- 应对措施二：分页

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest(uint pageSize, uint pageIndex) {  // 每人获得5%利息
9           uint begin = pageIndex * pageSize;
10          uint end = begin + pageSize;
11          for (uint i = begin; i < end; i += 1) {
12              accounts[i].balance = accounts[i].balance * 105 / 100;
13          }
14      }
15  }
```

智能合约的安全漏洞（四）

无界批量操作

- 应对措施三：使用“拉”模式代替“推”模式
- 让合约的用户单独调用函数，而不是由合约的管理者统一处理

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function applyMyOwnInterest() { // 获得自己的5%利息
5           balances[msg.sender] = balances[msg.sender] * 105 / 100;
6       }
7   }
```

目录

这些安全漏洞的根本原因？

- 人们没有意识到智能合约与传统程序的差别，依然在用写传统语言代码的方式来写智能合约

这些安全漏洞的根本原因？ [1]

- 环境的不确定性

这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为
  - 不确定的系统属性（链上属性）
  - 不确定的交易排序
  - 不确定的Gas消耗

这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为　　重入
  - 不确定的系统属性（链上属性）
  - 不确定的交易排序
  - 不确定的Gas消耗

这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为　　重入
  - 不确定的系统属性（链上属性）　系统属性依赖
  - 不确定的交易排序
  - 不确定的Gas消耗

这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为　　重入
  - 不确定的系统属性（链上属性）　　系统属性依赖
  - 不确定的交易排序
  - 不确定的Gas消耗

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	`GasPrice`	The transaction gas price.
1	`TxOrigin`	The transaction origin account.
2	`Coinbase`	The the beneficiary address of the block.
3	`BlockNum`	The block number.
4	`Timestamp`	The block timestamp.
5	`GasLimit`	The block gas limit.
6	`Difficulty`	The block difficulty.

这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为 重入
  - 不确定的系统属性（链上属性） 系统属性依赖
  - 不确定的交易排序 交易顺序依赖
  - 不确定的Gas消耗

Table 3. The system properties accessible by a contract during runtime.

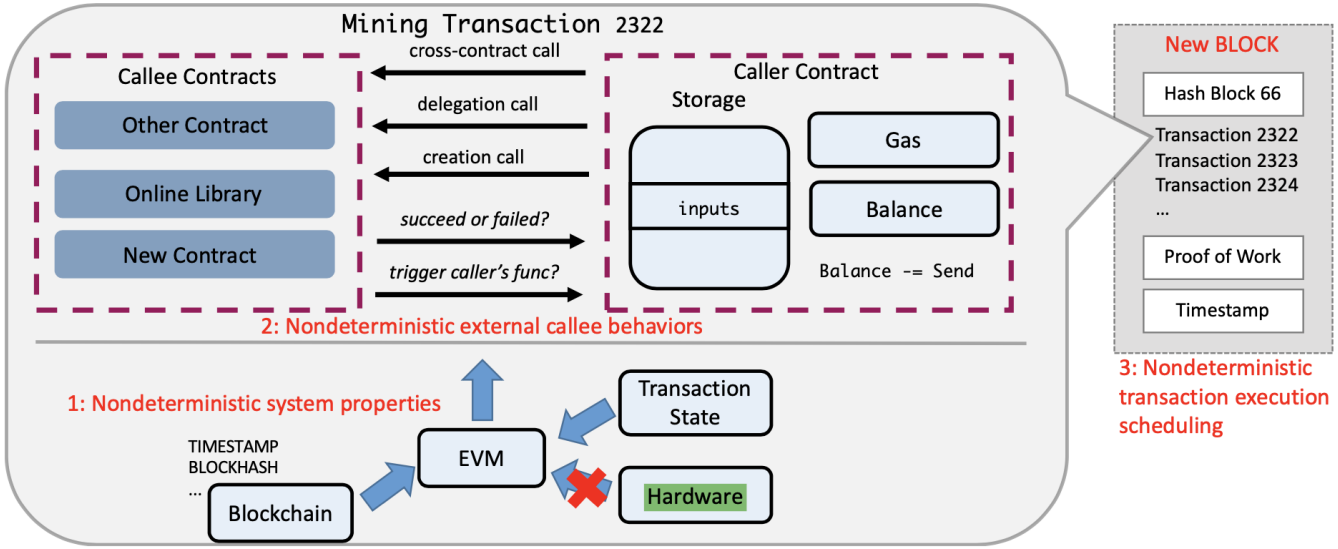
Index	Notation	Explanation
0	GasPrice	The transaction gas price.
1	TxOrigin	The transaction origin account.
2	Coinbase	The the beneficiary address of the block.
3	BlockNum	The block number.
4	Timestamp	The block timestamp.
5	GasLimit	The block gas limit.
6	Difficulty	The block difficulty.

这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为 重入
  - 不确定的系统属性（链上属性） 系统属性依赖
  - 不确定的交易排序 交易顺序依赖
  - 不确定的Gas消耗

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	GasPrice	The transaction gas price.
1	TxOrigin	The transaction origin account.
2	Coinbase	The the beneficiary address of the block.
3	BlockNum	The block number.
4	Timestamp	The block timestamp.
5	GasLimit	The block gas limit.
6	Difficulty	The block difficulty.

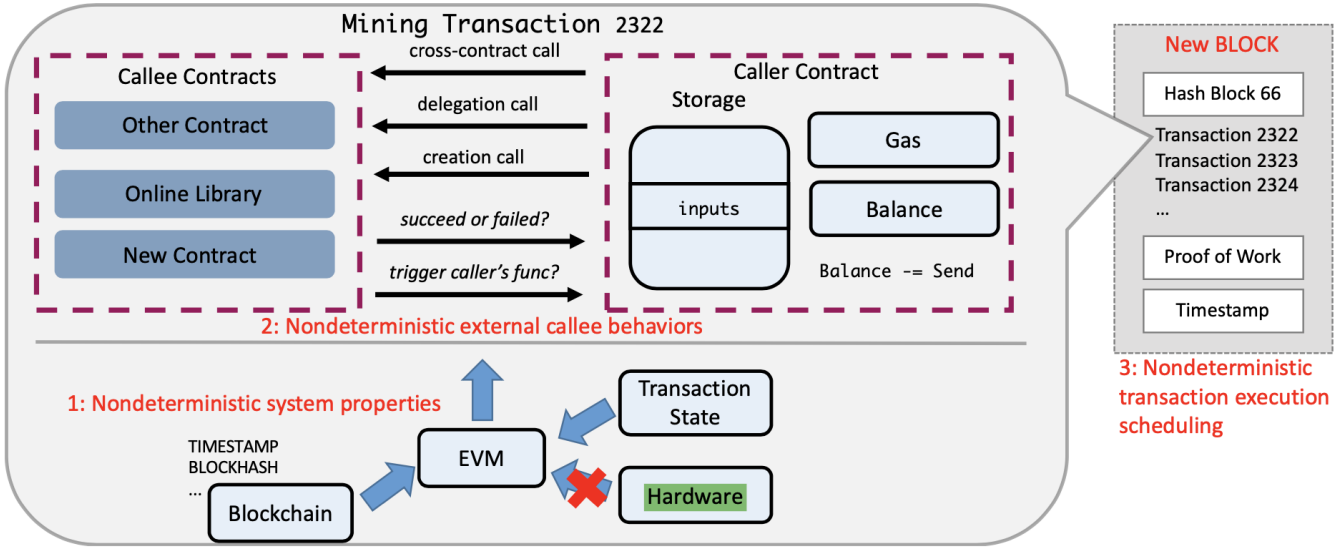


这些安全漏洞的根本原因？ [1]

- 环境的不确定性
  - 不确定的外部被调用者行为 重入
  - 不确定的系统属性（链上属性） 系统属性依赖
  - 不确定的交易排序 交易顺序依赖
  - 不确定的Gas消耗 无界批量操作

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	GasPrice	The transaction gas price.
1	TxOrigin	The transaction origin account.
2	Coinbase	The the beneficiary address of the block.
3	BlockNum	The block number.
4	Timestamp	The block timestamp.
5	GasLimit	The block gas limit.
6	Difficulty	The block difficulty.



这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() reentrant {
5           uint balance = balances[msg.sender];
6           (bool success, ) = msg.sender.call.value(balance)();
7           if (success) {
8               balances[msg.sender] = 0;
9           }
10      }
11  }
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() reentrant {
5           uint balance = balances[msg.sender];
6           (bool success, ) = msg.sender.call.value(balance)();
7           if (success) {
8               balances[msg.sender] = 0;
9           }
10      }
11  }
```

声明为reentrant表示函数可重入，否则（默认）不可重入

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() reentrant {
5           uint balance = balances[msg.sender];
6           (bool success, ) = msg.sender.call.value(balance)();
7           if (success) {
8               balances[msg.sender] = 0;
9           }
10      }
11  }
```

声明为reentrant表示函数可重入，否则（默认）不可重入

这些安全漏洞的根本原因？ [1]

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口

这些安全漏洞的根本原因？ [1]

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	GasPrice	The transaction gas price.
1	TxOrigin	The transaction origin account.
2	Coinbase	The the beneficiary address of the block.
3	BlockNum	The block number.
4	Timestamp	The block timestamp.
5	GasLimit	The block gas limit.
6	Difficulty	The block difficulty.

这些安全漏洞的根本原因？ [1]

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	GasPrice	The transaction gas price.
1	TxOrigin	The transaction origin account.
2	Coinbase	The the beneficiary address of the block.
3	BlockNum	The block number.
4	Timestamp	The block timestamp.
5	GasLimit	The block gas limit.
6	Difficulty	The block difficulty.

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件

```
contract PuzzleRewarder {
    address owner;
    uint reward;
    bool solved;

    function setReward(uint amount) {
        if (msg.sender == owner) {
            reward = amount;
        }
    }

    function claimReward(uint answer) {
        if (solved == false && isRight(answer)) {
            msg.sender.transfer(reward);
            solved = true;
        }
    }
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件

```
contract PuzzleRewarder {
    address owner;
    uint reward;
    bool solved;

    function setReward(uint amount) {
        if (msg.sender == owner) {
            reward = amount;
        }
    }

    function claimReward(uint answer) {
        if (solved == false && isRight(answer)) {
            msg.sender.transfer(reward);
            solved = true;
        }
    }
}
```

```
contract User {
    PuzzleRewarder puzzleRewarder;
    function solve() {
        puzzleRewarder.claimReward(42, condition = (reward > 100));
    }
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件

```
1   contract PuzzleRewarder {
2       address owner;
3       uint reward;
4       bool solved;
5
6       function setReward(uint amount) {
7           if (msg.sender == owner) {
8               reward = amount;
9           }
10      }
11
12      function claimReward(uint answer) {
13          if (solved == false && isRight(answer)) {
14              msg.sender.transfer(reward);
15              solved = true;
16          }
17      }
18  }
```

```
contract User {
    PuzzleRewarder puzzleRewarder;
    function solve() {
        puzzleRewarder.claimReward(42, condition = (reward > 100));
    }
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件

```
1   contract PuzzleRewarder {
2       address owner;
3       uint reward;
4       bool solved;
5
6       function setReward(uint amount) {
7           if (msg.sender == owner) {
8               reward = amount;
9           }
10      }
11
12      function claimReward(uint answer) {
13          if (solved == false && isRight(answer)) {
14              msg.sender.transfer(reward);
15              solved = true;
16          }
17      }
18  }
```

```
contract User {
    PuzzleRewarder puzzleRewarder;
    function solve() {
        puzzleRewarder.claimReward(42, condition = (reward > 100));
    }
}
```

满足该条件，调用才会成功

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];

    function applyInterest() autopaging(for account in accounts) { // 每人获得5%利息
        account.balance = account.balance * 105 / 100;
    }
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

```
1   contract Bank {
2       struct Account {
3           address userAddress;
4           uint balance;
5       }
6       Account accounts[];
7
8       function applyInterest() autopaging(for account in accounts) { // 每人获得5%利息
9           account.balance = account.balance * 105 / 100;
10      }
11  }
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];

    function applyInterest() autopaging(for account in accounts) { // 每人获得5%利息
        account.balance = account.balance * 105 / 100;
    }
}
```

```
for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
    accounts[i].balance = accounts[i].balance * 105 / 100;
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];

    function applyInterest() autopaging(for account in accounts) { // 每人获得5%利息
        account.balance = account.balance * 105 / 100;
    }
}
```

```
for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
    accounts[i].balance = accounts[i].balance * 105 / 100;
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];

    function applyInterest() autopaging(for account in accounts) { // 每人获得5%利息
        account.balance = account.balance * 105 / 100;
    }
}
```

```
for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
    accounts[i].balance = accounts[i].balance * 105 / 100;
}
```

这些安全漏洞的根本原因？

- 以太坊Solidity语言设计缺陷
  - 有机会在设计的初期从编程语言上扼杀这些漏洞
    - 重入：提供语言级别的关键字，声明函数是否可重入
    - 系统属性依赖：不提供系统属性的访问接口
    - 交易顺序依赖：为函数调用增加限定条件
    - 无界批量操作：
      1. 在语言级别上自动分页
      2. 完全移除gas的可观察性

> “也许你会说，只有当人滥用这个特性的时候，才会导致问题。然而语言设计的问题往往就在于，一旦你允许某种奇葩的用法，就一定会有人自作聪明去用。因为你无法确信别人是否会那样做，所以你随时都得提高警惕，而不能放松下心情来。”
>
> ——王垠[6]

```
contract Bank {
    struct Account {
        address userAddress;
        uint balance;
    }
    Account accounts[];

    function applyInterest() autopaging(for account in accounts) { // 每人获得5%利息
        account.balance = account.balance * 105 / 100;
    }
}
```

```
for (uint i = nextAccount; i < accounts.length && msg.gas > 100000; i += 1) {
    accounts[i].balance = accounts[i].balance * 105 / 100;
}
```

这些安全漏洞的根本原因？

- 以太坊设计者没有对语言和工具进行良好的抽象，暴露了过多的底层细节，增加开发者和用户的负担
  - 应减少底层函数调用接口

这些安全漏洞的根本原因？

- 以太坊设计者没有对语言和工具进行良好的抽象，暴露了过多的底层细节，增加开发者和用户的负担
  - 应减少底层函数调用接口

```
contract Bank {
    mapping(address => uint) balances;

    function withdraw() {
        uint balance = balances[msg.sender];
        balances[msg.sender] = 0;
        (bool success, ) = msg.sender.call.value(balance)();
        require(success);
    }
}

contract Attacker {
    Bank bank;
    function attack() { bank.withdraw(); }
    function() { bank.withdraw(); }
}
```

这些安全漏洞的根本原因？

- 以太坊设计者没有对语言和工具进行良好的抽象，暴露了过多的底层细节，增加开发者和用户的负担
  - 应减少底层函数调用接口

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

call是一个底层函数，不会抛出异常，
需要手动检查是否成功

这些安全漏洞的根本原因？

- 以太坊设计者没有对语言和工具进行良好的抽象，暴露了过多的底层细节，增加开发者和用户的负担
  - 应减少底层函数调用接口

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

call是一个底层函数，不会抛出异常，需要手动检查是否成功

更安全的设计

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           try {
8               msg.sender.call.value(balance)();
9           } catch(error) {
10              revert();
11          }
12      }
13  }
```

这些安全漏洞的根本原因？

- 以太坊设计者没有对语言和工具进行良好的抽象，暴露了过多的底层细节，增加开发者和用户的负担
  - 应减少底层函数调用接口

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           (bool success, ) = msg.sender.call.value(balance)();
8           require(success);
9       }
10  }
11
12  contract Attacker {
13      Bank bank;
14      function attack() { bank.withdraw(); }
15      function() { bank.withdraw(); }
16  }
```

call是一个底层函数，不会抛出异常，需要手动检查是否成功

更安全的设计

```
1   contract Bank {
2       mapping(address => uint) balances;
3
4       function withdraw() {
5           uint balance = balances[msg.sender];
6           balances[msg.sender] = 0;
7           try {
8               msg.sender.call.value(balance)();
9           } catch(error) {
10              revert();
11          }
12      }
13  }
```

所有可能抛出异常的语句都需要使用try-catch处理异常

目录

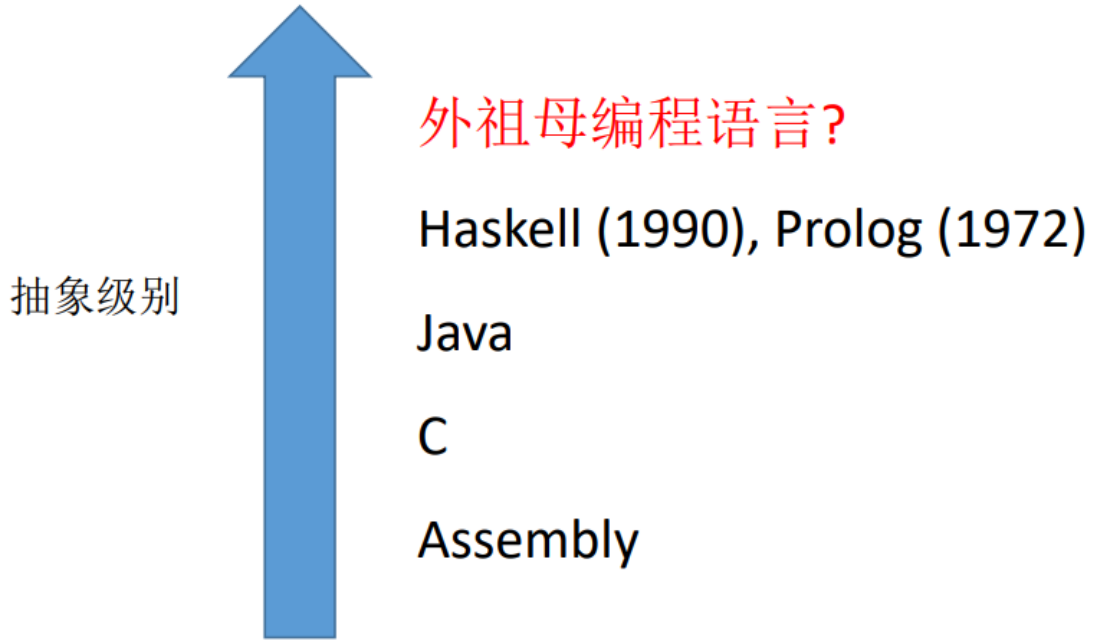
总结[7]

总结[7]

> *Abstraction is what you need.*
>
> *— Ziyan Wang (2021)*

总结[7]

- 程序设计语言的发展历史就是提高抽象级别



> *Abstraction is what you need.*
>
> *— Ziyan Wang (2021)*

来源：北京大学熊英飞软件分析技术课程

总结[7]

- 程序设计语言的发展历史就是提高抽象级别

抽象级别

外祖母编程语言？

Haskell (1990), Prolog (1972)

Java

C

Assembly

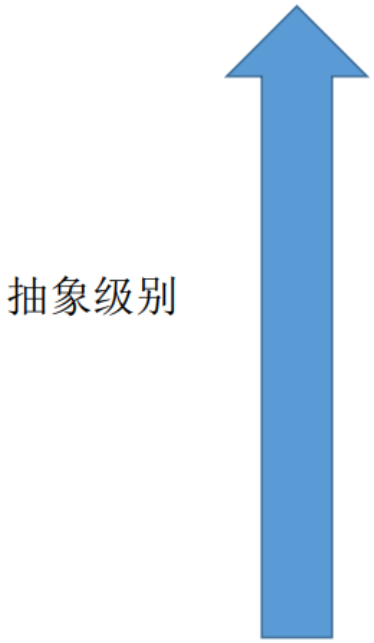
> *Abstraction is what you need.*
>
> *— Ziyan Wang (2021)*

```
less code, less bug
less known, less bug
```

来源：北京大学熊英飞软件分析技术课程

总结[7]

- 程序设计语言的发展历史就是提高抽象级别



外祖母编程语言？

Haskell (1990), Prolog (1972)

Java

C

Assembly

> *Abstraction is what you need.*
>
> *— Ziyan Wang (2021)*

```
less code, less bug
less known, less bug
```

抽象的坏处：性能降低

来源：北京大学熊英飞软件分析技术课程

总结

- 智能合约没有必要使用图灵完备的编程语言
- “美元，人民币，黄金…它们有合约的功能吗？没有。为什么数字货币一定要捆绑这种功能呢？我觉得这违反了模块化设计的原则：一个事物只做一点事，把它做到最好。数字货币就应该像货币一样，能够实现转账交换的简单功能就可以了。合约应该是另外独立的系统，不应该跟货币捆绑在一起。

  那合约怎么办呢？交给律师和会计去办，或者使用另外独立的系统。你有没有想过，为什么世界上的法律系统不是程序控制自动执行的呢？为什么我们需要律师和法官，而不只是机器人？为什么有些国家的法庭还需要有陪审团，而不光是按照法律条款判案？这不只是历史遗留问题。你需要理解法律的本质属性才会明白，完全不通过人来进行的机械化执法是不可行的。智能合约就是要把人完全从这个系统里剔除出去，那是会出问题的。”

  —王垠[10]

参考资料

[1] Wang et al, Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts

[2] Grech et al, MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts

[3] https://zh.wikipedia.org/wiki/%E5%8F%AF%E9%87%8D%E5%85%A5

[4] https://consensys.github.io/smart-contract-best-practices/known_attacks

[5] https://swcregistry.io/

[6] https://www.yinwang.org/blog-cn/2016/09/18/rust

[7] https://xiongyingfei.github.io/SA/2020/main.htm

[8] https://www.zhihu.com/question/37290469/answer/107612456

[9] https://www.zhihu.com/question/37290469/answer/1562163408

[10] http://www.yinwang.org/blog-cn/2018/02/22/smart-contract