WeBank

FISCO BCOS 2.0 智能合约

2019年7月

FISCO 金链盟

目录

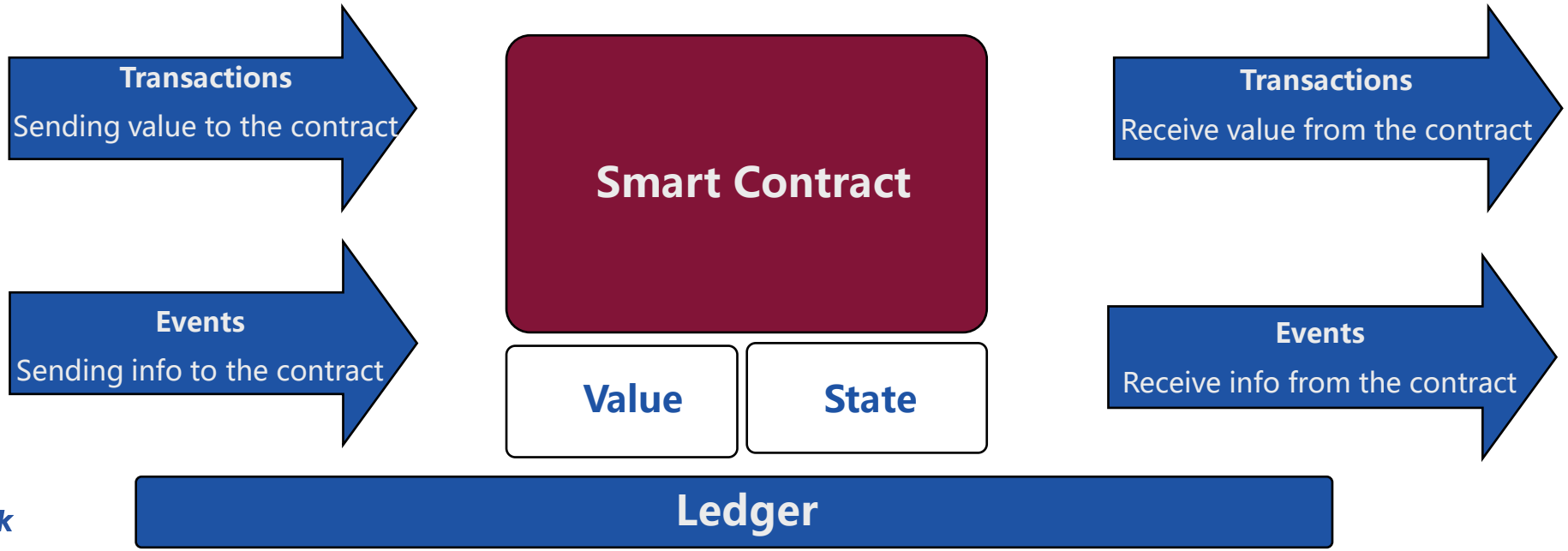
- 基本概念
- EVM原理
- Solidity语言
- 智能合约案例
- 智能合约安全

01

基本概念

什么是智能合约

- 智能合约是在区块链上达成协议的一种方式，智能合约并没有超越传统合约的功能，但它能最小化达成协议所需的信任。
- 智能合约不仅是一个可以自动执行的程序，它自己就是一个区块链参与者。它对接收到的交易进行回应，它可以接收和储存价值，也可以向外发送信息和价值。



智能合约发展史


概念
• 1990，尼克萨博提出智能合约概念
比特币
• 2009，比特币脚本
以太坊
• 2015，以太坊智能合约
BCOS
• 2017，BCOS发布

比特币脚本缺陷

- 比特币的UTXO不一定被某个公钥持有，也可以被基于堆栈的区块链脚本持有，在被脚本持有的情况下，使用者必须提供满足脚本要求的数据才可使用该UTXO。例如，比特币区块链上可以创建要求集齐三把密钥中的两把才能使用的UTXO，即多重签名；也可以对解决某个数学问题的用户发送奖励。
- 比特币脚本存在一些严重的限制：

✓缺少图灵完备性，尽管比特币支持多种运算，但不能支持所有的运算。最主要的确实是循环语句，不支持循环语句的目的是避免交易确认时出现无限循环。理论上程序员可以利用多次的重复if来模拟，但这样导致空间上的低效率。

✓缺少价值控制，比特币脚本无法为UTXO提供精确控制，由于UTXO是不可分割的，比特币脚本只能决定一次授予整个UTXO或否。唯一的方法是非常低效地生成许多不同面值的UTXO，并挑出正确的UTXO来使用。

✓缺少状态，UTXO只能是已花费或者未花费状态，比特币脚本无法访问本UTXO以外的状态，这使得实现诸如期权合约、多阶段合约或者二元状态的合约变得非常困难，诸如限额的应用是不可能实现的。

以太坊智能合约

- 以太坊智能合约借鉴了比特币脚本，对它的应用范围进行了扩展。如果说比特币是利用区块链技术的专用计算器，那么以太坊就是利用区块链技术的通用计算机。简单地讲，以太坊 = 区块链 + 智能合约。
- 与比特币脚本相比，以太坊智能合约最大的不同点是：它可以支持图灵完备的开发语言，其允许开发者在智能合约内执行任意功能，对资产进行精确状态，访问整个区块链的状态。
- 以太坊智能合约支持的应用包括：

  ✓差价合约

  ✓代币系统（ERC20）

  ✓储蓄钱包

  ✓保险

  ✓多重签名合约

交易执行原理

- 本质上，交易是从一个账户发往另一个账户的消息
- 如果交易发送的目的地是外部账户，那么这个交易的目的就是**转账**
- 如果交易发送的目的地是合约账户，那么这个交易的目的就是**调用智能合约**。收到这类交易后，底层会取出目的地合约账户的代码（code），以交易的数据（data）做为智能合约调用的参数，初始化一个EVM并执行
- 如果交易发送的目的地是空地址，那么这个交易的目的就是**创建智能合约**。收到这类交易后，底层会根据发送者的地址和发送者的nonce，计算哈希值，作为一个新的合约账户的地址。同时，EVM会执行交易的数据（data），将执行的结果，做为合约代码保存到新合约账户的代码中

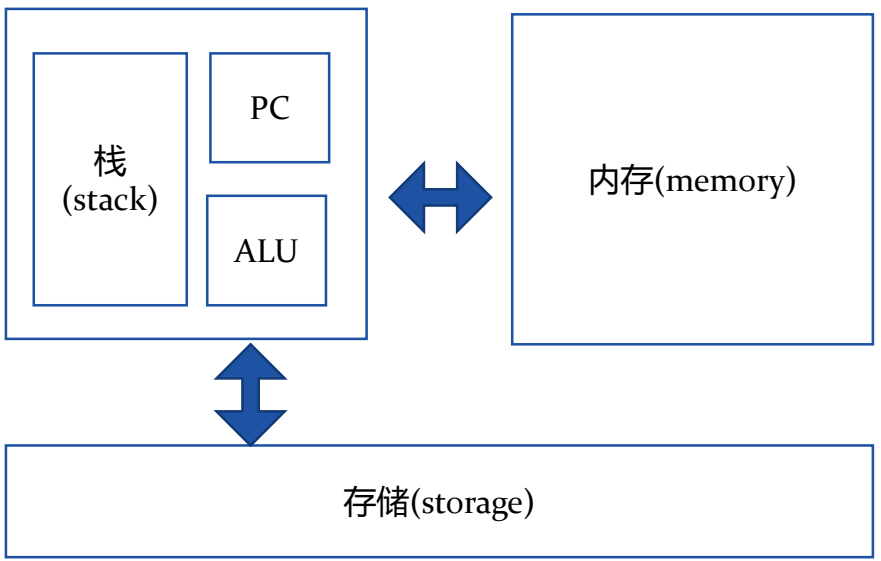
02

EVM原理

EVM架构

EVM的核心是大小为256bit的寄存器，称为PC，PC总是指向某条EVM指令，从EVM启动开始，直到EVM停机，ALU一直不停地执行PC指向的指令，再更新PC指向下一条指令

- **栈**中每个元素的大小为256bit，栈的最大深度为1024



- **内存**是临时存储设备，在ALU执行时，用来存放智能合约和智能合约处理的数据。
- 内存是一个线性的整数数组，每个整数有唯一的地址（数组索引），大小为256bit。

- **存储**：账户拥有持久的存储空间。
- 存储是key-value结构，key和value均为256bit的二进制串，在不借助外力的情况下，存储是不可遍历的，智能合约只能读写本账户的存储，无法读写其它账户的存储

EVM详解

- EVM是供智能合约运行的沙盒环境，与外部环境**完全**隔离
- EVM无法访问网络、文件系统其它进程，智能合约在EVM内只允许进行有限的操作
- EVM为哈佛架构的虚拟机，指令、数据和栈完全分离
- EVM执行流程：

1. 调用方传入调用智能合约的参数到区块链底层：address、data和value
2. 区块链底层到存储中查找address对应的合约账户，读出合约账户的code数据
3. 区块链底层初始化一个新EVM，将data和value放入memory，并根据code启动EVM
4. EVM解析data的参数，开始执行
5. EVM执行完成后，返回值放入memory
6. 区块链底层从memory获取返回值，传回给调用方

交易

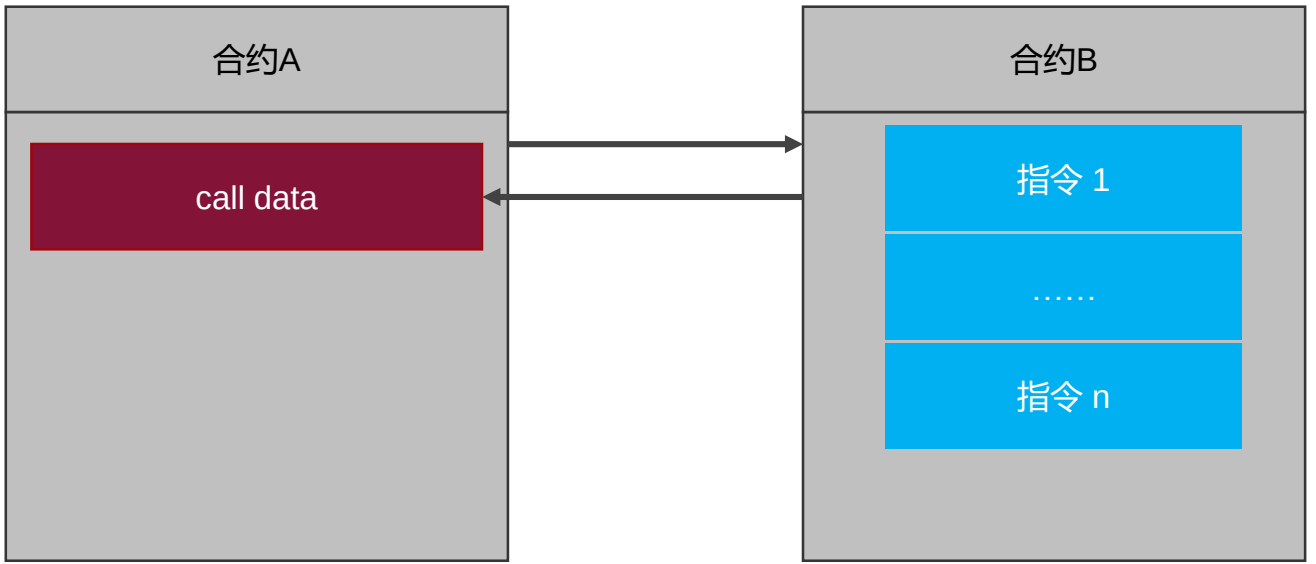
- 交易可以看作是一个账户向另一个账户发送的消息；
- 消息包含一个称为payload的二进制数据和以太；
- 如果接受账户是一个合约，payload表示输入；
- 如果目标地址为空，则创建一个新合约，payload表示EVM bytecode

EVM指令集

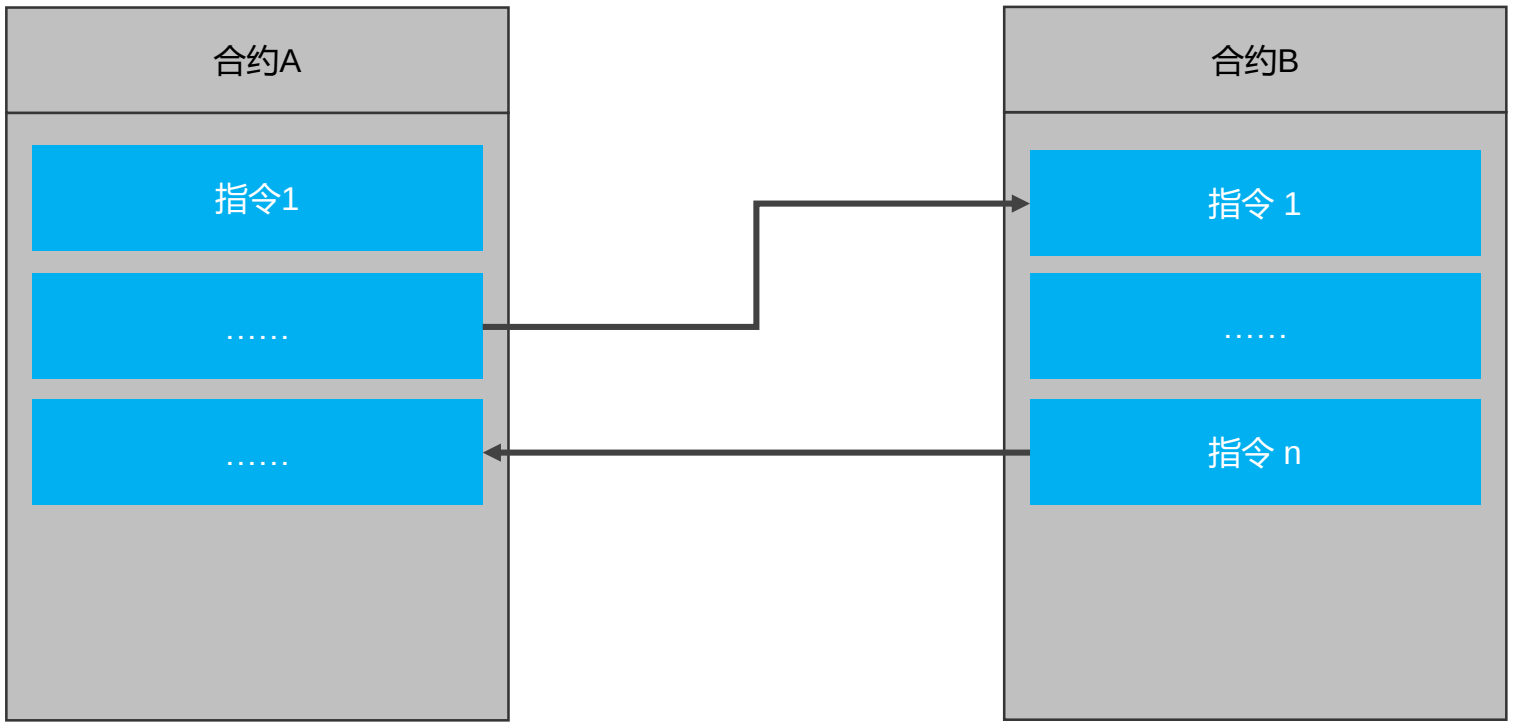
指令分类	指令列表
栈操作	PUSH1..PUSH32 DUP1..DUP16 SWAP1..SWAP16 POP
运算	ADD MUL SUB DIV SDIV MOD SMOD ADDMOD MULMOD EXP SIGNEXTEND LT GT SLT SGT EQ ISZERO AND OR XOR NOT BYTE SHA3
内存	MLOAD MSTORE
存储	SLOAD SSTORE
条件跳转	JUMP JUMPI PC JUMPDEST JUMPIF JUMPV JUMPSUB JUMPSUBV JUMPTO JUMPC RETURNSUB
日志	LOG0..LOG5
调用	ETHCALL MSIZE CREATE CALL CALLCODE RETURN
GAS	GASPRICE GAS
环境	ADDRESS BALANCE ORIGIN CALLER CALLVALUE CALLDATALOAD CALLDATASIZE CALLDATACOPY CODESIZE CODECOPY EXTCODESIZE EXTCODECOPY BLOCKHASH COINBASE TIMESTAMP NUMBER DIFFICULY GASLIMIT
机器控制	STOP SUICIDE

Message Call

- 和调用合约类似，一个合约可以通过消息调用的方式调用另一个合约；
- 消息调用包含payload，以太，gas等
- 在合约A通过消息调用调用合约B时，A会留出一个存储区域calldata给B；
- B执行结束后，把数据存储在A的calldata内



Delegate Call



03

Solidity 语言

Solidity

- Solidity是运行在EVM上，面向合约的高级编程语言，它的语法受C++、Python和JavaScript的影响
- Solidity使用静态类型，支持继承、库和用户自定义类型等特性

参考：http://solidity.readthedocs.io/en/v0.4.24/

Solidity开发环境

- Remix编辑器
  - ✓Solidity的在线编辑器
  - ✓支持语法高亮、代码补全和代码管理等功能
  - ✓支持编译Solidity代码
  - https://remix.ethereum.org/

合约长什么样?

```
pragma solidity ^0.4.16;

/* this is my first contract */
import “./another.sol” as another;

contract HelloWorld {

}
```

版本号0.4.16声明不兼容0.4.16之前的版本；^符号表明不接受大版本（0.5.0以及后续）编译器编译。

合约必须以contract开头

EVM变量类型

- Solidity的变量，根据其声明位置、修饰符的不同，分三种类型
  - calldata：calldata保存在栈中，保存public函数的参数，该类数据无需手工声明，由solidity自动分配，由于EVM最多访问深度为16的栈元素，因此calldata最多支持16个变量
  - memory：memory变量位于EVM的memory区，存放函数的临时局部数据，函数执行完后即销毁，函数内临时变量和私有函数的参数均为memory变量
  - storage：storage变量保存在State存储中，声明在合约内、函数外的变量均为storage变量

```
pragma solidity ^0.4.16;

/* this is my first contract */
import “./another.sol” as another;

contract HelloWorld {

}
```

变量寻址

- calldata变量存储于栈中，EVM的运算指令直接从栈中取数计算
- memory变量存储于内存，通过EVM的MLOAD、MSTORE指令操作
- 存储变量保存在State存储中
- 存储器变量在State中按Key-Value存储，存储规则如下：
  - 变量Hash Key：按合约hash+变量序号计算hash
  - 数组Hash Key：变量Hash Key对应的Value，存储数组大小，变量Hash Key+下标值的Hash Key对应的Value，存储元素值
  - Mapping Hash Key：变量Hash Key+Mapping Key Hash的Hash Key对应的Value，存储元素值

Solidity数据类型

- 简单数据类型
  - bool：bool类似C/C++中的bool，长度为256bit
  - int/uint：solidity的基本数字类型，长度为256bit
  - address：solidity的地址类型，用来表示账户地址
- 定长数据类型
  - bytes1, bytes2, bytes3 … bytes32：不同长度的byte类型，从1字节到32字节
  - 定长数据类型有通用的length()方法，获取数据长度
- 变长数据类型
  - bytes：变长byte串，存储原始字节数据
  - string：变长string串，存储utf-8格式数据

Address类型

- Address存储外部账户地址和合约账户地址
- Address类型包含下列方法：
  - balance()：获取当前账户的余额
  - send()：执行转账，失败返回false
  - transfer()：执行转账，失败throw
  - call()：以外部函数调用的形式调用某个address的函数
  - callcode()：以内部函数调用的形式调用某个address的函数
  - delegatecall()：以内部函数调用的形式调用某个address的函数，设置msg.sender和msg.value

结构体

- 结构体只能在内部函数调用时使用，无法在外部函数调用时使用
- 结构体无法保存到storage变量中

```
pragma solidity ^0.4.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

数组

- 声明在函数内的memory类型定长数组，需要使用new[]()来申请memory空间

```
pragma solidity ^0.4.16;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        a[6] = 8;
    }
}
```

- 声明为函数参数的定长数组

```
pragma solidity ^0.4.16;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) public pure {
        // ...
    }
}
```

数组特性

- 定长数组和变长数组无法互相转换
- memory类型的定长数组，一旦声明，无法改变长度
- 函数内部不支持二维数组
- 声明为storage的数组，有两个额外的成员函数：
  - length：获取当前数组长度
  - push：往数组尾部追加数据
- 如果函数的返回值被声明为变长数组，如bytes或string，这个函数返回的变长数组无法在Solidity内部获取，只能由外部获取（web3js、web3sdk等）

Mapping

- Mapping声明为 mapping(_KeyType => _ValueType)，Key支持所有简单类型和string，Value支持所有类型，包括mapping
- Mapping可视为一个hashtable
- 获取不存在的键值，当类型是int/uint时，取得0，当类型是string时，取得空string
- Mapping仅能声明为storage变量
- Mapping不能遍历，但可以通过与数组结合实现遍历

时间与日期

- Solidity支持使用字符标识时间单位，如：
  - ✓1 == 1 seconds
  - ✓1 minutes == 60 seconds
  - ✓1 hours == 60 minutes
  - ✓1 days == 24 hours
  - ✓1 weeks == 7 days
- 通过now，可以获取当前区块的时间戳

```
function f(uint start, uint daysAfter) public {
    if (now >= start + daysAfter * 1 days) {
      // ...
    }
}
```

环境变量和方法

- blockhash(uint blockNumber) returns (bytes32)：获取指定块hash
- block.coinbase (address)：获取出块者
- block.difficulty (uint)：当前块难度（POW相关，已废弃）
- block.gaslimit (uint)：当前gas限制（eth相关，已废弃）
- block.number (uint)：当前块高
- block.timestamp (uint)：当前块时间戳，单位毫秒
- gasleft() returns (uint256)：当前剩余gas（eth相关，已废弃）
- msg.data (bytes)：交易数据
- msg.sender (address)：消息的上一级发送者（外部账号或合约账号的地址）
- msg.sig (bytes4)：调用函数的签名
- msg.value (uint)：转账的币值（FISCO-BCOS已废弃）
- now (uint)：当前块时间戳，单位毫秒，等同于block.timestamp
- tx.gasprice (uint)：当前交易的gasprice（FISCO-BCOS已废弃）
- tx.origin (address)：消息的原始发送者（外部账号）

函数

- 函数声明

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    function bid() public payable { // Function
        // ...
    }
}
```

- 函数修饰符
  - ✓ public：标识函数为公有，可以被外部访问
  - ✓ constant：标识函数为只读，调用该函数时，允许使用call()，不需要发交易
  - ✓ payable：标识函数可以收发balance，与eth有关，已废弃

控制结构

- 大部分控制结构语法类似Javascript语法的控制结构，支持if、else、while、do、for、break、continue、return和三元运算符？：

```
function foo() {
  // To be called by a vulnerable contract with a withdraw function.
  // This will double withdraw.

  vulnerableContract v;
  uint times;
  if (times == 0 && attackModeIsOn) {
    times = 1;
    v.withdraw();

   } else { times = 0; }
}
```

异常处理

- 智能合约出现异常时，EVM将被中止，已执行的改动会被回滚，下列几类操作均会导致EVM的异常：
  - ✓ 访问空Address
  - ✓ 外部函数调用时，调用不存在的函数
  - ✓ 外部函数调用时，传入了错误的参数列表
  - ✓ 除（模）0错误
- 智能合约方法内调用throw()，可以主动制造异常，放弃本次交易，丢弃所有改动

```
function withdrawBalance() {
  if ( withdrawMutex[msg.sender] == true) { throw; }
  withdrawMutex[msg.sender] = true;
  amountToWithdraw = userBalances[msg.sender];
  if (amountToWithdraw > 0) {
    if (!(msg.sender.send(amountToWithdraw))) { throw; }
  }
  userBalances[msg.sender] = 0;
  withdrawMutex[msg.sender] = false;
}
```

函数修饰器

- 函数修饰器可以实现类似AOP的切面编程功能，在函数执行前后进行检查

```
pragma solidity ^0.4.22;

contract priced {
    mapping (address => bool) registeredAddresses;
    uint price;

    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }

    function register() public costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}
```

事件（Event）

- 事件声明类似函数，包括名称和参数
- 事件可以在函数内调用，调用事件后，产生的事件标识和事件参数会保存在Receipt中
- 已产生的事件只能由外部读取，无法由合约调用

```
pragma solidity ^0.4.21;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

    function bid() public payable {
        // ...
        HighestBidIncreased(msg.sender, msg.value); // Triggering event
    }
}
```

Constant函数

- Solidity中，函数分为constant和非constant两类，constant函数不修改任何状态，调用时无需发送交易，非constant函数会修改状态，调用时需要发送交易
- 以下行为均算作修改状态
  - 写入storage变量
  - 调用event
  - 调用其它合约
  - 收发ether（FISCO-BCOS已废弃）
  - 调用了非constant的函数
- 在声明为constant的函数中修改了状态，其做的任何修改在交易执行完成后均会被回退

函数重载

- 同名函数可以有不同的入参

```
pragma solidity ^0.4.16;

contract A {
    function f(uint _in) public returns (uint out) {
        out = 1;
    }

    function f(uint _in, bytes32 _key) public returns (uint out) {
        out = 2;
    }
}
```

内部函数调用

- 合约内函数相互调用时，称为内部函数调用，此时函数的参数和返回值均由memory传递，支持传入和传出变长数组

```
pragma solidity ^0.4.16;

contract C {
    function g(uint a) public pure returns (uint ret) { return f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

外部函数调用

- 一个合约外调另一个合约的public方法时，称为外部函数调用
- 外部函数调用时，主调EVM会构造一个全新的被调EVM，并将调用参数拷贝至被调EVM的memory，由被调EVM执行被调的合约方法，执行完成后的返回值，被调EVM将其拷贝到主调EVM的memory
- 外部函数调用时，不支持变长数组的返回值
- 注意，外部函数调用时，EVM不检查被调合约是否存在、是否合法或是否符合预期，只要提供了ABI和参数，即可调用

```
pragma solidity ^0.4.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) public { feed = InfoFeed(addr); }
    function callFeed() public { feed.info.value(10).gas(800)(); }
}
```

密码学函数

- addmod(uint x, uint y, uint k) returns (uint)：计算(x + y) % k
- mulmod(uint x, uint y, uint k) returns (uint)：计算(x * y) % k
- keccak256(...) returns (bytes32)：计算SHA-3-256(Keccak-256)
- sha256(...) returns (bytes32)：计算SHA-256
- sha3(...) returns (bytes32)：等同于keccak256
- ripemd160(...) returns (bytes20)：计算RIPEMD-160 hash，例如账号地址
- ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)：从EC签名还原公钥

创建合约

- 创建智能合约，有两种方式：
  - 发送创建合约交易

```
pragma solidity ^0.4.0;
```

  - 在智能合约内使用new()创建合约

```
contract D {
    uint x;
    function D(uint a) public payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
    }
}
```

继承

- Solidity支持多继承，实现多态
- Solidity的所有函数都是虚函数
- 一个继承了多个合约的合约，在EVM内部视为一个合约，该合约整合了多个父类合约的代码

```
pragma solidity ^0.4.11;

contract A {
    uint public a;

    function A(uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    function B() public {}
}
```

抽象合约

- 抽象合约内可以只声明方法，没有具体实现
- 抽象合约多用于抽象父类、声明ABI

```
pragma solidity ^0.4.0;

contract Feline {
    function utterance() public returns (bytes32);
}

contract Cat is Feline {
    function utterance() public returns (bytes32) { return "miaow"; }
}
```

04

智能合约案例

Storage合约

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public constant returns
(uint) {
        return storedData;
    }
}
```

storage变量，该变量位于EVM的storage区

”set”函数，该函数有一个参数，对此函数的调用，需要发交易执行

”get”函数，该函数声明为constant，不会更改智能合约的状态，不需要发交易执行

Asset合约

```
pragma solidity ^0.4.21;

contract Asset {
    address public minter;
    mapping (address => uint) public balances;

    event Sent(address from, address to, uint amount);

    function Coin() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount)
public {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount)
public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

只有minter才有权限发行资产

声明了key为address、value为uint的mapping，存储各个账户的余额

通知账户变动的事件，由send()调用

铸币

转账接口

05

智能合约安全

The DAO事件回顾

14
个事件

- 2016年4月30日，The DAO上线开始为期28天的全球众筹
- 2016年5月10日，10天时间融得以太币的价值已达到3400万美元
- 2016年5月15日，众筹金额超过1亿美元
- 2016年5月28日，众筹结束，融得超过1150万个以太币，相当于超过1.5亿美元价值，成为全球历史上最大金额众筹项目
- 2016年6月9日，以太坊开发人员Peter Vessenes指出The DAO存在递归调用漏洞
- 2016年6月14日，修复方案被提交，等到The DAO成员的审核
- 2016年6月16日，递归调用问题再次被提及
- 2016年6月17日，黑客发起针对The DAO智能合约多个漏洞的攻击，其中也包含了递归调用漏洞，并向一个匿名地址转移了3600万个以太币，几乎占据了The DAO众筹总量1150个的三分之一
- 2016年6月18日，开放交易验证后，社区号召大家通过发送大量垃圾交易阻塞交易验证的形式减缓黑客的继续偷盗；同时白帽通过使用与黑客同样的方法将剩余2/3未被盗取资金转移到安全账户
- 2016年6月24日，以太坊社区提交了软分叉提案
- 2016年6月28日，Felix Lange指出软分叉提案存在DoS攻击风险
- 2016年6月30日，以太坊创始人Vitalik Buterin提出硬分叉设想
- 2016年7月15日，具体硬分叉方案公布，建立退币合约
- 2016年7月21日，超过85%的算力支持硬分叉，以太坊硬分叉成功

举例：递归漏洞

```
function splitDAO(…) … {
        …
        //Burn DAO Tokens
        Transfer(msg.sender, 0, balances[msg.sender]);
        withdrawRewardFor(msg.sender); // be nice, and get his rewards
        totalSupply -= balances[msg.sender];
        balances[msg.sender] = 0;
        paidOut[msg.sender] = 0;
        return true;
    }
```

```
function withdrawRewardFor(address _account) noEther internal returns (bool _success) {
        if ((balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply < paidOut[_account])
            throw;

        uint reward =
            (balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply - paidOut[_account];
        if (!rewardAccount.payOut(_account, reward))
            throw;
        paidOut[_account] += reward;
        return true;
    }
```

```
function payOut(address _recipient, uint _amount) returns (bool) {
        if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
            throw;
        if (_recipient.call.value(_amount)()) {
            PayOut(_recipient, _amount);
            return true;
        } else {
            return false;
        }
    }
```

递归调用

举例：权限漏洞

- 合约的关键接口应有权限判断，否则会被恶意利用
- Parity multi-sig钱包中曾出现过权限漏洞，导致价值3200万美元的ETH被盗

```
pragma solidity ^0.4.15;

contract Unprotected{
    address private owner;

    modifier onlyowner {
        require(msg.sender==owner);
        _;
    }

    function Unprotected()
        public
    {
        owner = msg.sender;
    }

    // This function should be protected
    function changeOwner(address _newOwner)
        public
    {
       owner = _newOwner;
    }

    function changeOwner_fixed(address _newOwner)
        public
        onlyowner
    {
       owner = _newOwner;
    }
}
```

举例：整数溢出漏洞

- 无符号的整数运算中，运算可能溢出导致取得预期外的值
- 在整数运算前先做判断，避免溢出

```
pragma solidity ^0.4.15;

contract Overflow {
    uint private sellerBalance=0;

    function add(uint value) returns (bool){
        sellerBalance += value; // possible overflow

        // possible auditor assert
        // assert(sellerBalance >= value);
    }

    function safe_add(uint value) returns (bool){
        require(value + sellerBalance >= sellerBalance);
        sellerBalance += value;
    }
}
```

举例：时间戳篡改

- 区块的时间戳来自矿工，并不一定是现实世界的时间戳
- 矿工可以为了特定目的生成符合要求的恶意时间戳
- 良好的智能合约应避免依赖特定的时间戳

```
pragma solidity ^0.4.15;

function play() public {
require(now > 1521763200 && neverPlayed == true);
 neverPlayed = false;
 msg.sender.transfer(1500 ether);
}
```

微众银行/金链盟区块链开源产品：Please Star

- **WeIdentity：实体身份标识 x 可信数据交换**
  - https://github.com/WeBankFinTech/WeIdentity
- **WeEvent：基于区块链的事件驱动架构**
  - https://github.com/WeBankFinTech/WeEvent
- **WeBASE：区块链中间件平台**
  - https://github.com/WeBankFinTech/WeBASE
- **FISCO-BCOS：**
  - https://github.com/FISCO-BCOS/FISCO-BCOS

WeBank

谢谢！