谈谈区块链（03）hyperledger入门

最好的项目是fabric这个项目，

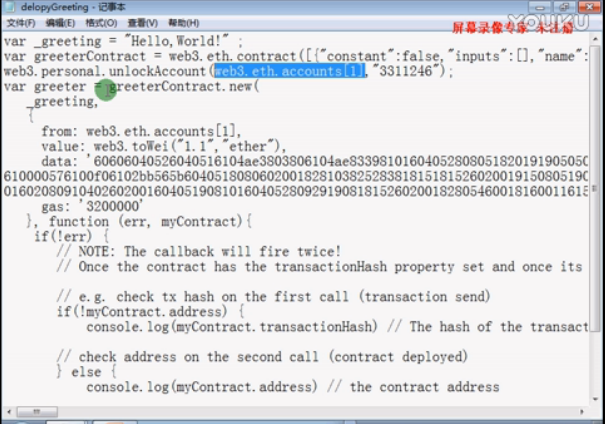


谈谈区块链（09）：以太坊智能合约入门

智能合约的执行：

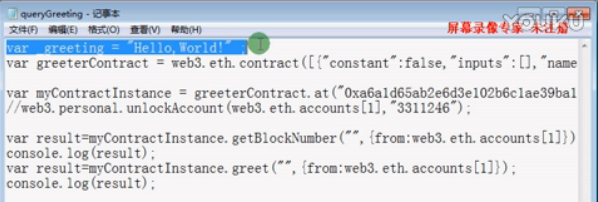
官方的例子：

var \_greeting = /\* var of type string here \*/ ; var helloworldContract = web3.eth.contract([{"constant":false,"inputs":[],"name":"kill","outputs":[],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"\_newgreeting","type":"string"}],"name":"setGreeting","outputs":[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"greet","outputs":[{"name":"","type":"string"}],"payable":false,"type":"function"},{"inputs":[{"name":"\_greeting","type":"string"}],"type":"constructor"}]); var helloworld = helloworldContract.new( \_greeting, { from: web3.eth.accounts[0], data: '', gas: 4700000 }, function (e, contract){ console.log(e, contract); if (typeof contract.address !== 'undefined') { console.log('Contract mined! address: ' + contract.address + ' transactionHash: ' + contract.transactionHash); } })

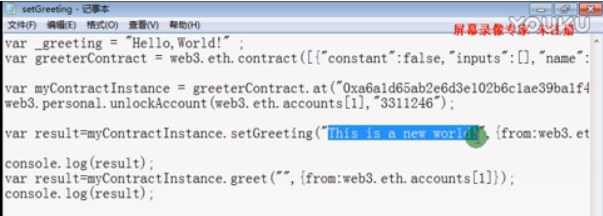




访问智能合约：



设置智能合约：





调用的时候其中第一个是参数！！！！！

1.函数声明中的payable用来允许像这样feed.info.value(10).gas(800)接受参数

2.函数可以被标记为external,public,internal,private四种，其中默认是public。对于状态变量(state variables),不能是external的，其默认值是internal

3.可以在创建合约中，转发代币

4.事件(Event)。 可选的功能。合约中的Deposit（充值）和Send（发送）事件是会被记录在以太坊虚拟机日志中的数据。它们实际上没有任何作用，但是用事件(Event)把交易记录进日志是好的做法。

5.call.（模拟evm的执行） 我们使用call来检查变量的值，例如conference.quota.call().then(...，还可以通过传参数，例如call(0), 来获取mapping在index 0处的元素。Solidity的文档说这是一种特殊的“消息调用”

因为 1.不会为矿工记录和 2.不需要从钱包账户/地址发起（因此它没有被账户持有者私钥做签名）。另一方面，交易/事务(Transaction)会被矿工记录，必须来自于一个账户（也就是有签名），会被

记录到区块链上。对合约中数据做的任何修改都是交易。仅仅是检查一个变量的值则不是。因此在读取变量时不要忘记加上call()！否则会发生奇怪的事情。（此外如果在读取变量是遇到问题别忘记检

查它是否是public。）call()也能用于调用不是交易的函数。如果一个函数本来是交易，但你却用call()来调用，则不会在区块链上产生交易。

6.Gas

7.gaslimit：block里面所有交易的累加值不能超过

比特币是通过size或者时间来限制交易数

以太坊是通过gaslimit

以太坊上的每笔交易都会被收取一定数量的gas，gas的目的是限制执行交易所需的工作量，同时为执行支付费用。当EVM执行交易时，gas将按照特定规则被逐渐消耗。

8.以太坊的TPS怎么控制？？？

9.transcation and messagecall？？

gas price（gas价格，以太币计）是由交易创建者设置的，发送账户需要预付的交易费用 = gas price \* gas amount。 如果执行结束还有gas剩余，这些gas将被返还给发送账户。

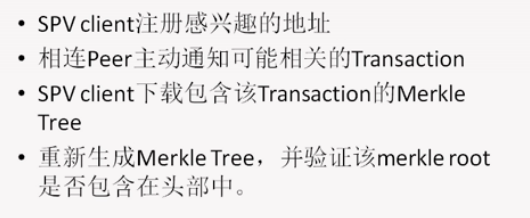
无论执行到什么位置，一旦gas被耗尽（比如降为负值），将会触发一个out-of-gas异常。当前调用帧所做的所有状态修改都将被回滚

1. 用solidity编写的智能合约可以用js脚本直接来执行

谈谈区块链（11）：以太坊web3.js详解

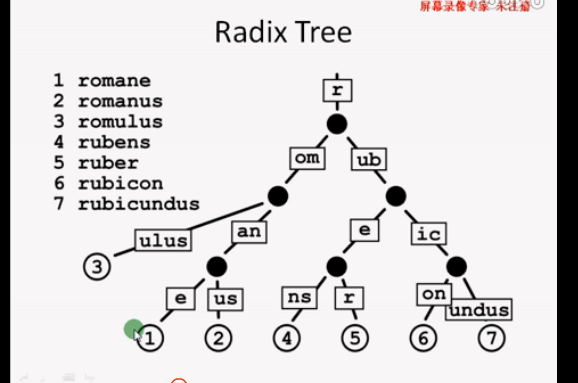
谈谈区块链（12）：Merkle树和Merkle\_Patricia树

比特币默克尔树最大的好处应该就是SPV证明方式。

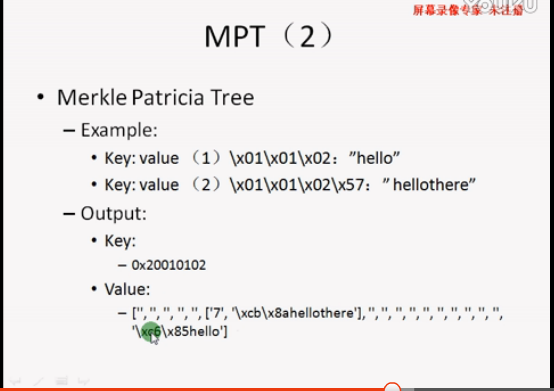


对于以太坊：

基数树：便于查找和索引

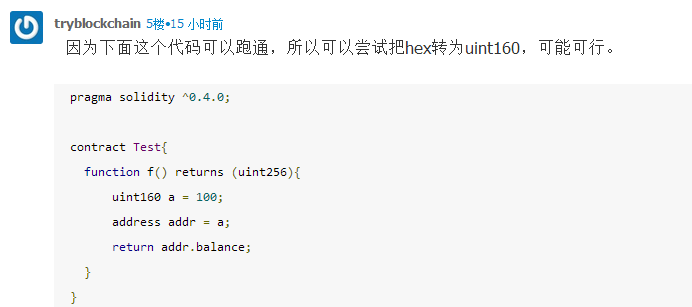


RLP的编码方式：





谈谈区块链(20)



地址强制转化关系

向合约账户和普通账户发送以太币的区别？

pragma solidity ^0.4.0;正则表达式，只接受大于等于这个版本的

执行到throw的地方，执行会终止，gas会全部用完

特殊变量：

block.blockhash(uint blockNumber) returns (bytes32)，指定区块的哈希值，只支持最近256个区块，且不包含当前区块。

block.coinbase (address) 当前矿工的地址。

block.difficulty (uint)当前块的难度。

block.gaslimit (uint)当前块的gas限制。

block.number (uint)当前区块的块号。

block.timestamp (uint)当前块的时间。

msg.data (bytes)完整的调用数据（calldata）。

msg.gas (uint)当前还剩的gas。

msg.sender (address)调用发起人的地址。

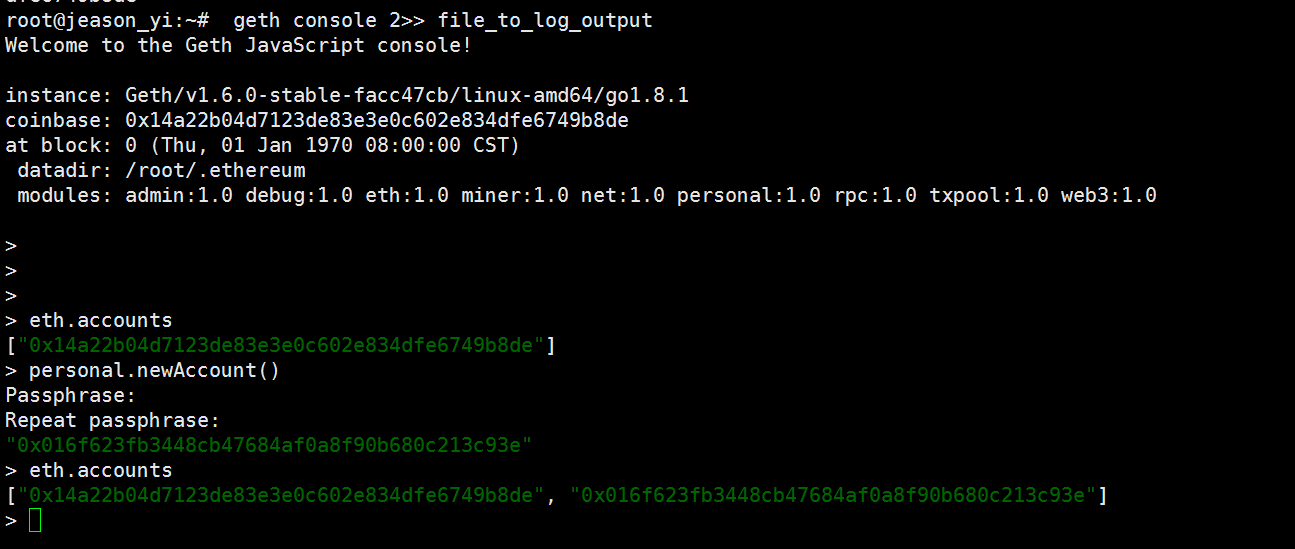
msg.sig (bytes4)调用数据的前四个字节。

msg.value (uint)这个消息所附带的钱，单位为wei。

now (uint)当前块的时间戳，等同于block.timestamp

tx.gasprice (uint) 交易的gas价格。

tx.origin (address)交易的发送者（完整的调用链）



谈谈区块链（24）记录

注：本文以Solidity作为智能合约的编程语言。

先看下面两段代码：

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(!addr.call.value(20 ether)()){

throw;

}

以及：

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(!addr.send(20 ether)){

throw;

}

这两段代码都是向0x6c8f…的合约地址发送20个ether，第二段代码没有漏洞，而第一段代码却存在严重的安全漏洞。为什么？

我们先来看一下addr.call.value()()（注意：是两个括号，第一个括号是对要转移多少以太币的赋值，第二个括号是方法的调用）和addr.send()的区别。两者都是向某个地址发送以太币，都是一个新的message call，不同的是这两个调用的gaslimit不一样。send()给予0的gas（相当于call.gas(0).value()())，而call.value()()给予全部（当前剩余）的gas。

注：对于需要调用fallback函数又没有给予任何gas的情况，EVM将自动把gas调整为不超过2300。

看下面的代码：

function withdrawBalance() {

amountToWithdraw = userBalances[msg.sender];

if（amountToWithdraw > 0){

if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }

userBalances[msg.sender] = 0;

}

}

这是一段给用户取款的代码，让用户一次性从你的智能合约里取回存款。例如，你的合约账户共有1000个ether，而某用户存有10个ether。因为该代码存有严重的递归调用漏洞，该用户可轻松地将你账户里的1000个ether全部提走。

首先，该段代码使用了addr.call.value()()来发送ether，而不是send()，给fallback函数的调用提供了足够多的gas。你只要将fallback函数写成如下的方式便可取走所有的ether：

function () {

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(COUNT<100){

addr.call("withdrawBalance");

COUNT++;

}

}

在这段fallback代码中，当计数器小于100时，递归调用withdrawBalance函数。在这种情况下，

msg.sender.call.value(amountToWithdraw)()

将被调用100次，从而取走100\*10 ether。

所以在写智能合约时，需要考虑到它可能被递归调用，在这个case里，我们可以这样调整代码以防止递归调用而出现的问题：

function withdrawBalance() {

amountToWithdraw = userBalances[msg.sender];

userBalances[msg.sender] = 0;

if（amountToWithdraw > 0){

if (!(msg.sender.call.value(amountToWithdraw)())) {

userBalances[msg.sender] = amountToWithdraw;

throw;

}

}

}

看下面这段代码：

function sendether(){

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

addr.send(20 ether);

//you think the send should return true

var thesendok = true;

//do something regarding send returns ok

...

}

并且对方的fallback函数定义为：

function(){

//do nothing

}

你认为你的代码肯定是安全的，因为对方已经明确定义了fallback方法。但是你错了，攻击者只需要制造出1023个嵌套调用，然后再调用sendether()，就可以让add.send(20 ether)失败，而其它执行成功。代码如下：

function hack(){

var count = 0;

while(count < 1023){

this.hack();//this keyword makes it a message call

count++;

}

if(count==1023){

thecallingaddr.call("sendether");

}

}

所以为了解决深度限制的问题，正确的写法应该是在每次涉及到call depth增加的地方都检查调用返回是否正确，如下：

function sendether(){

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(!addr.send(20 ether)){

throw; //somebody hacks me

}

//you think the send should return true

var thesendok = true;

//do something regarding send returns ok

...

}

了解了智能合约里面的这三个基本的漏洞，下面我们来看看在著名的The DAO事件中，黑客是怎么利用它们来窃取以太币，造成以太坊发展史上影响最为惨重的一次事件。

谈谈区块链(36)

1. internal function call VS message call (两种调用方式) （其实就是不跨栈）

Stack based on evm

This关键字是不能用于构造函数的

内部调用(Internal Function Calls)

外部函数调用(External Function Calls)

Message call最多可以嵌套1024个

字段memory数组不能修改长度

1. Address.call



只有方法是payable的，才可以用info.value(10).gas(800)()这样的（间接发送交易）

其它合约的函数必须使用外部的方式调用。对于一个外部调用，所有函数的参数必须要拷贝到内存中。当从其它合约调用函数时，可以指定传递的wei为单位的数额和gas。

pragma solidity ^0.4.0;

contract InfoFeed {

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

}

contract Consumer {

InfoFeed feed;

function setFeed(address addr) returns (address) {

feed = InfoFeed(addr);

return addr;

}

function callFeed() returns (uint) {

return feed.info.value(10).gas(800)();

}

}

InfoFeed.info()函数声明中的payable用来允许像这样feed.info.value(10).gas(800)接受参数。

**深入解析调用合约的三种方法**

当获取合约实例之后（比如 testInstance），在geth console中可以通过三种方法调用合约方法（比如testFunc）

testInstance.testFunc.sendTransaction();

testInstance.testFunc();

testInstance.testFunc.call();

本文将讲解这三种调用方法的区别

testInstance.testFunc.sendTransaction(); 会创建一个交易，调用之后会返回一个交易hash值，它会广播到网络，等待矿工打包, 它会消耗gas。

testInstance.testFunc.call(); 它完全是一个本地调用，不会向区块链网络广播任何东西，它的返回值完全取决于 testFunc 方法的代码，不会消耗gas

testInstance.testFunc(); 它会比较特殊，由于有constant标识的方法不会修改状态变量，所以它不会被编译器执行。所以，如果testFunc() 有constant标识，它并不会被编译器执行，web3.js会执行call()的本地操作。相反如果没有constant标识，会执行sendTransaction()操作。

来验证一下

写个合约，代码如下

pragma solidity ^0.4.2;

contract Test {

uint public testMem;

function testFunc1() returns (string resMes){

testMem++;

resMes = "try to modify testMem,but has no constant label";

}

function testFunc2() constant returns (string resMes){

testMem--;

resMes = "try to modify testMem and has constant label";

}}

将合约部署到私有链，并获取合约实例testInstance 调用testFunc1

> testInstance.testFunc1({from:eth.accounts[0]})

I0117 19:38:21.348763 internal/ethapi/api.go:1047] Tx(0x157d429be29953ea451ea95cf468a3a67c4a86e9b49d1b6b97cc15c579a27003) to: 0xc9bc867a613381f35b4430a6cb712eff8bb50310"0x157d429be29953ea451ea95cf468a3a67c4a86e9b49d1b6b97cc15c579a27

可见，确实创建了一笔交易，开启挖矿，等待打包…再查看下

> testInstance.testMem()1> eth.getTransaction('0x157d429be29953ea451ea95cf468a3a67c4a86e9b49d1b6b97cc15c579a27003'){

blockHash: "0x9fa0c7d071e1d2e772de3f6f326595b9d9159b0056213416018c75f2d5c04ad2",

blockNumber: 118,

from: "0xcb1f9cd557b5dd81955a4df89e9b4c8a33023c12",

gas: 90000,

gasPrice: 20000000000,

hash: "0x157d429be29953ea451ea95cf468a3a67c4a86e9b49d1b6b97cc15c579a27003",

input: "0x561f5f89",

nonce: 45,

r: "0xb77558d48ab4efcaa24309b1003a7c4efabfdda26c3844c6aa18c58c6a08181a",

s: "0x57fe6dde27fa7cfd2fb422e2adc9465125750b3271127b68bb65d496d99be531",

to: "0xc9bc867a613381f35b4430a6cb712eff8bb50310",

transactionIndex: 0,

v: "0x1c",

value: 0}

可见，它确实是一笔交易，修改了合约的状态变量，并且有90000的gas消耗。 再来试下testFunc2

> testInstance.testFunc2({from:eth.accounts[0]})"try to modify testMem and has constant label"

确实只是在本地执行，并没有创建交易，所以更不会修改合约的状态变量。

谈谈区块链（37）记录

核心：Zcash零知识证明原理

谈谈区块链（35）记录

Rootstock是bitcoin的一个侧链