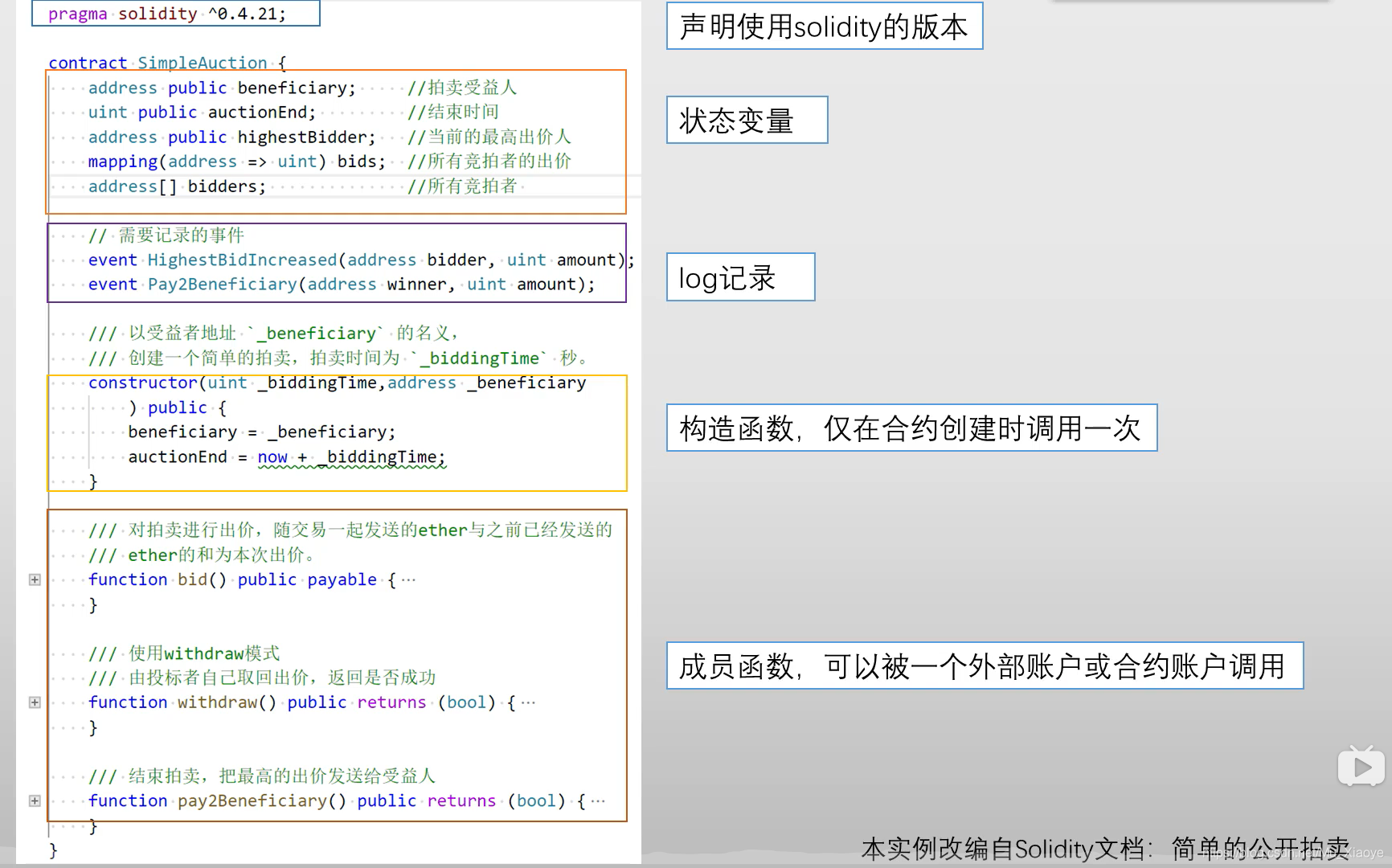
1. 智能合约简介

智能合约：运行在区块链系统上的一段代码，代码逻辑定义了合约内容。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，数据结构为一棵MPT

智能合约编写代码为Solidity，其语法与JavaScript很接近。下图显示了智能合约的代码结构。



address是solidity特有的，后面会详细讲到；mapping是从地址到无符号整数的映射，event 是用来记录日志的，第一个event的参数是拍卖的地址和金额，第二个是获胜者的地址和拍卖金额；solidity 不支持遍历，如果想要遍历元素，自己需要想办法记录一下哈希表有哪些元素，这里是用bidders数组记录的，solidity语言中数组可以是固定长度也可以是动态改变长度的。

构造函数有两种，第一种是像C++构造函数一样，定义一个与contract同名的函数，函数可以有参数但是不能有返回值；新版本更推荐的是用constructor来定义构造函数，这个函数只有合约在创造的时候才被调用一次，构造函数也只有一个。

成员函数里第一个有payable，另外两个函数没有，因为以太坊规定，合约账户要能接受外部转账的话必须标注为payable。

上面是一个网上拍卖的合约，这个例子中的bid函数是用来竞拍出价的，比如说要参与拍卖，要出100个以太币，那么就调用合约中的bid函数。所以拍卖的规则是调用bid函数的时候要把拍卖的出价即100个以太币也发送过去存储在合约里，锁定那里一直到拍卖结束。避免有人凭空出价（实际上没有那么多钱，漫天喊价），所以拍卖的时候要把出的价钱发在智能合约里锁定起来。所以这个bid函数要有能够接受外部转账的能力，所以才标注了payable。

成员函数的withdraw函数就没有payable，withdrew函数的用处是在拍卖结束时，出价最高的人赢得拍卖，其他人没有拍到想要拍到的东西，可以调用withdraw函数，把自己当初的出价，也就是之前bid的时候锁定在智能合约里的以太币，再取出来。withdraw的目的不是真的转账，不需要把钱转给智能合约，而仅仅是通过调用withdraw函数把当初锁定在智能合约的钱取回来，所以没必要用payable。

1. 如何调用智能合约

**2.1 外部账户调用**

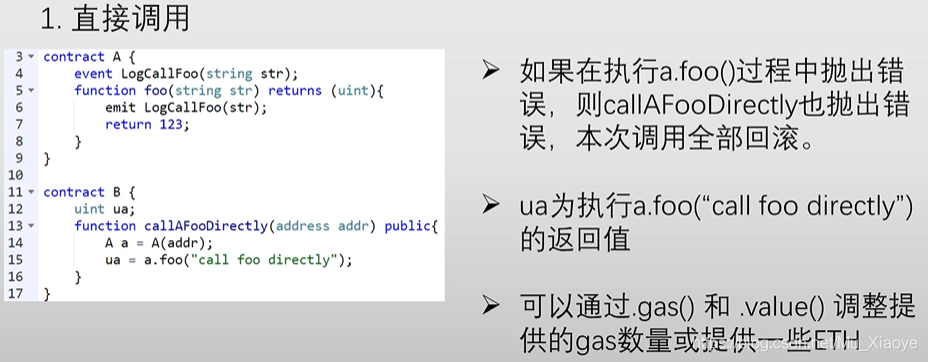


调用智能合约和转账类似，A-B转账，如果B是个普通账户，那么这只是一个普通的转账交易，和BTC的转账交易是一样的，如果B是合约账户的话，那么这个转账其实是发起一次对B的合约的调用，具体调用的是合约中的哪个函数是在数据域(data域)中另外说明的。send address是发起调用的账户地址，To Contract Address是被调用的合约的地址，调用的函数是TXdata里面给出的要调用函数，如果这个函数有参数，那么其参数也在这里的data域里说明的，上面的案例的三个成员函数都是没有参数的，但是有一些成员函数是有参数的。中间一行是调用的参数，Value是发起调用的时候转账花的钱数，这里是0，说明这里只是想调用函数并不想真的转账，所以这里的To contract address函数不需要定义payable。Gas used是这个交易所花的汽油费，gas priced 是单位汽油的价格，gas limit是这比交易愿意支付的最多汽油。

**2.2 合约账户调用**

合约账户之间也可以进行调用。其调用方式如下：

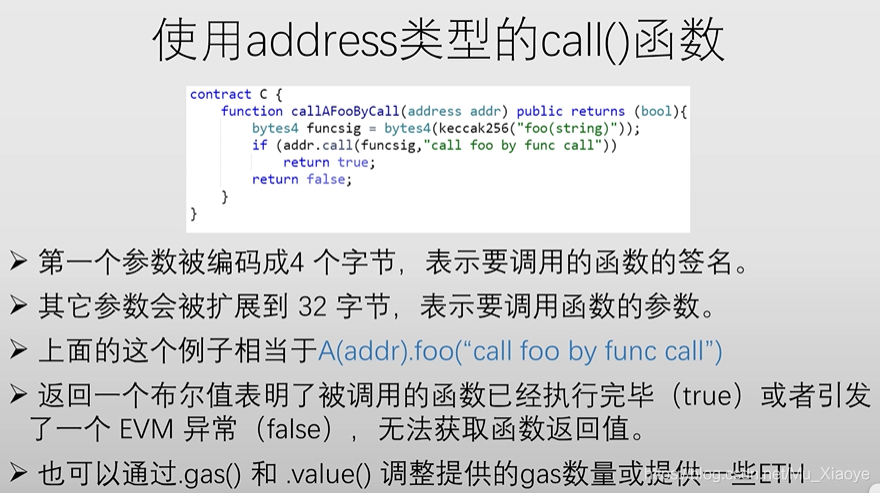
1. 直接调用



合约A和合约B，A合约就只是写成Log（日志），event是定义一个事件，叫LogCallFoo，emit 来调用这个事件，emit的作用就是写一个log，对于程序运行没有影响；B合约参数是一个地址，就是A合约的地址，然后这个语句把这个地址转换成A这个合约的一个实例，然后调用foo这个函数。

以太坊中规定一个交易只有外部账户才能发起，合约账户不能自己主动发起一个交易。这个例子当中实际上是需要一个外部账户调用了合约B当中的函数CallAFooDirectly，然后这个函数再调用A合约中的foo函数。

1. Address调用



第一个参数是函数的signature，后面跟的是调用参数。这种方法和上面的方法的区别：一是对错误处理的不同，上面的方法调用的时候，如果调用的合约在执行过程中出现错误，会导致发起这个调用的合约跟着一起回滚，上面的例子当中如果合约A出现异常，B也会跟着出现异常；addr.call()这种方法，如果在调用过程中被调用的合约产生异常，call函数会返回False，表明这个调用时失败的，但是发起调用的这个函数并不会抛出异常，而是可以继续执行。

1. 代理调用



主要区别是delegatecall()不需要切换到被调用的合约的环境中去执行，而是在当前的合约中执行就可以了，比如就用当前的账户余额存储之类的。

1. fallback()函数

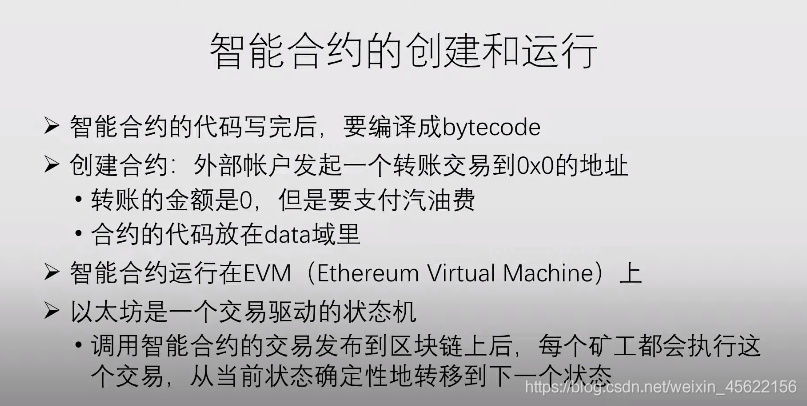


无参数无返回值，无函数名，fallback关键字并没有出现在函数名里面。调用合约的时候，A调用B合约，要在转账交易的data域说明调用的是合约B中的哪个函数，如果A给B转了一笔钱，没有说明调用的是哪个函数，也就是data域是空的，这个时候缺省的就是调用这个fallback函数，这也是为什么叫fallback函数，因为没有别的函数可以调用了，就只能调用他。还有一种情况是你要调用的函数不存在，在你的data域里你说你要调用这个函数，实际合约当中没有这个函数，也是调用fallback函数，这也是为什么这个函数没有参数也没有返回值。fallback函数也可能需要标注payable关键词，就如果fallback函数需要有接受转账的能力的话是需要写payable，一般情况都是写成payable，如果合约账户没有任何函数标志为payable，包括fallback函数也没有，那么这个合约没有任何能力可以接受外部的转账。如果有人往合约里转钱就会引发异常。

转账金额可以为0，是给收款人的，但是汽油费是要给矿工的，不给的话矿工不会把交易打包到区块链上的。

1. 智能合约的创建

智能合约的创建是由某一个外部的账户发起一笔转账交易，转给0X0地址，然后把要发布的合约代码放到data域里面。Java Virtual Machine是为了增强可一致性，EVM也是类似的思想，通过加一层虚拟机，对智能合约的运行提供一致性的平台，所以EVM又叫world wide compute，EVM的寻址空间是非常大的，是256位的，像如之前讲的uint和signed int就是256位的，普通计算机是64位的。



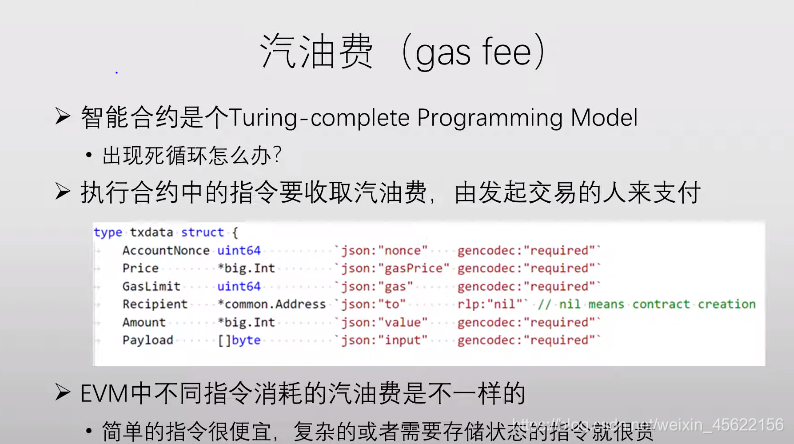
1. 汽油费

比特币和以太坊两种区块链模型的设计理念是有很大差别的，比特币的设计理念是简单，脚本语言的功能很有限，不支持循环。而以太坊是要提供一个图灵完备的编程模型。很多功能在比特币系统上实现不了或者比较困难，在以太坊中实现起来却是非常容易。当然这样也会带来一些问题，比如说出现死循环怎么办，当一个全节点收到一个对智能合约的调用，怎么知道这个调用执行起来会不会导致死循环，有什么解法吗？

答案是没有办法，因为这是一个典型的停机问题(halting problem). 停机问题是不可解的，需要注意一下这个问题不是NPC的（Non-deterministic Polynomial的问题，即多项式复杂程度的非确定性问题），NPC的问题是可解的，只不过没有多项式时间的解法，很多NPC问题有很多自然的指数时间的解法，比如哈密尔顿回路问题，把所有可能性枚举一遍，n个顶点的排列是n!，把每个组合检查一下是不是构成一个合法的回路，就知道它有没有哈密尔顿回路，哈密尔顿回路是可解的，只不过解的复杂度是指数级的。停机问题已经从理论上证明不存在这样的算法能够对任意给定的输入程序判断出这个程序是否会停机，这是不可解的。

**那么以太坊中是如何应对这个问题的呢？**

以太坊把这个问题推给发起交易的账户，以太坊引入了汽油费机制，你发起一个对智能合约的调用需要支付相应的汽油费。



交易的数据结构：

AccountNonce是交易的序号，用于防止前面说到的replay attack(重放攻击);price是单位汽油的价格，Gaslimit是这个交易愿意支付的最大汽油量，两者相乘之后就是这个交易可能消耗的最大汽油费;recipient是收款人的地址；amount的转账金额；payload就是之前说的data域，用于存放调用的是合约中哪一个函数以及函数的参数取值是什么。

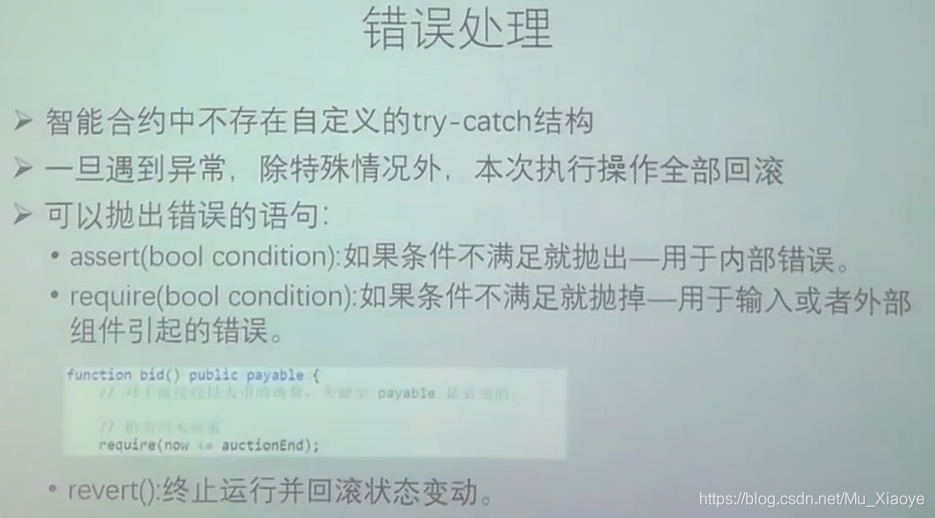
当一个全节点收到一个对智能合约的调用的时候，先按照这个调用给出的gas limit算出可能花掉的最大汽油费，然后一次性把汽油费从发起调用的账户中扣掉，然后再根据实际执行情况算出实际花了多少汽油费，汽油费不够会引起回滚(但是已经消耗的汽油费不会回滚)。

简单指令例如加减法消耗的汽油费比较少，复杂的指令消耗的比较多，比如说取哈希，这个运算虽然一条指令就可以完成，但是汽油费就比较贵。除了计算量之外，需要存储状态的指令消耗的汽油费也是比较大的。相比之下，如果只是为了读取公共数据，那些指令是可以免费的。

当遇到死循环时，汽油费会不断消耗，当消耗至不足以执行下一次循环时就会出错，不再继续执行该智能合约。

1. 智能合约的错误处理

以太坊中交易具有原子性，要么全执行，要么全不执行，不会只执行一部分(包含智能合约)。需要注意的是，在执行过程中产生错误导致回滚，已经消耗掉的汽油费是不会退回的。从而有效防止了恶意节点对全节点进行恶意调用。



一些语言像java用户可以自己定义出现错误怎么办，solidity没有try-catch结构，不可以。

**出现错误的情况：**

1. 错误情况一：之前所说的汽油费，如果这个交易执行完之后没有达到当初的gaslimit，那么多余的汽油费会被退回到这个账户里；相反的，如果执行到一半，gaslimit用完了，合约的执行要退回到开始执行之前的状态，而且这个时候已经消耗的汽油费是不退的。为什么这么设计呢？防止一些恶意的节点发动denial service attack，发动一个计算量很大的合约然后不停地调用这个合约，每次调用的时候给的汽油费都不够，反正最后汽油费都会退回来，对恶意节点来说没什么损失，但是对矿工来说白白浪费了很多资源。
2. 错误情况二：assert语句和require语句，这里两个语句都是用来判断判断某种条件，如果条件不满足的话就会导致抛出异常。assert语句一般来说是用来判断某种内部条件，和c语言中的类似；reuire语句判断某种外部条件，比如说判断函数的输入是否符合要求，下图所给的例子是bid函数里，判断当前时间now是否小于等于拍卖结束时间，如果符合条件，继续执行，不符合，即拍卖时间已经结束了，这个时候就会抛出异常。
3. 错误情况三：revert语句无条件抛出异常，如果执行到revert语句，那么他自动的就会导致回滚，早期版本用的是throw语句，新版本的solidity建议改为revert语句。

**智能合约的嵌套调用**

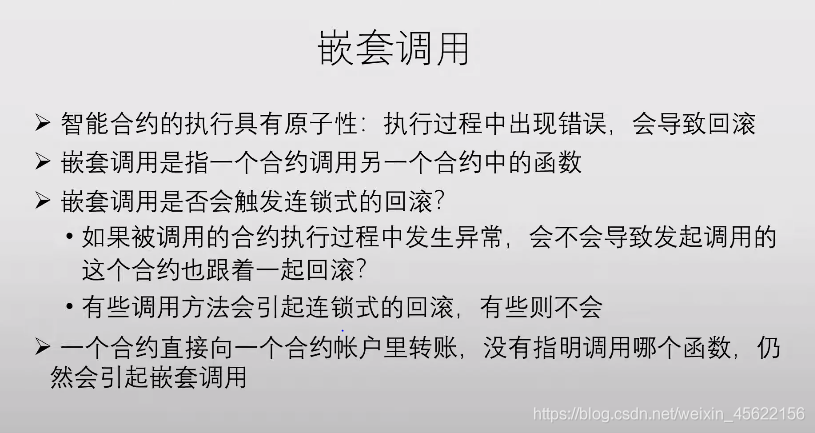
Q1： 前面说智能合约调用出现错误会导致回滚，那么如果是嵌套调用，一个智能合约调用另外一个智能合约，被调用的智能合约出现错误是不是会导致发起调用的智能合约也跟着一起回滚呢？叫做连锁式回滚。

A1: 不一定，这个取决于调用智能合约的方式，如果这个智能合约是直接调用的，那么它会触发连锁式的回滚，整个交易都会回滚。如果是用call()这种方式调用，他就不会引起连锁式回滚，只会使当前的调用失败，返回一个False的返回值

Q2：有些情况下，从表面上看，你并没有调用任何函数，比如说单纯的账户转账，但是如果这是以个合约账户的话，转账的本身就有可能触发对函数的调用，为什么呢？

A2: 因为有fallback函数，这就是一种嵌套调用，一个合约往另外一个合约里转账，就可能调用这个合约里的fallback函数。

（给合约转账，合约里没有fallback函数，也没有说明调用哪个函数，call本身就会返回false，但是不会引起连锁式回滚。）

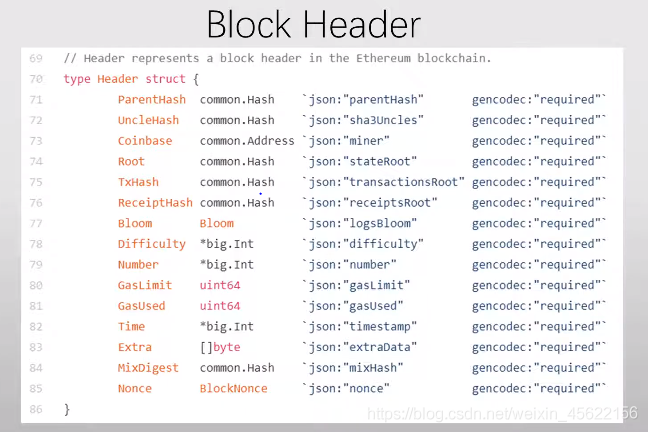


1. 数据结构

**6.1 blockheader数据结构回顾**

比特币中规定每个区块不能超过1M，是写在协议里不能更改的，比特币的交易是比较简单的，基本上可以用交易的字节数来衡量出这个交易消耗的资源有多少，但是以太坊这么规定是不行的，因为智能合约的逻辑很复杂，有的交易从字节上看可能很小，但是它消耗的资源很大，比如它可能调用别的合约之类的，所以怎么办呢？要根据交易的具体操作来收费，这就是汽油费。

GasUsed是这个区块里，所有交易所消耗的汽油费加在一起，GasLimit是这个区块里所有交易能够消耗汽油的一个上限，这里和每个交易的gaslimit(自己设定的)是不一样的。以太坊的上限GasLimit，和比特币不太一样，每个矿工在发布区块的时候可以对这个GasLimit进行微调，它可以在上一个区块的GasLimit上调或者下调1/1024，这种机制实际求出的系统GasLimit是所有矿工认为比较合理的GasLimit的一个平均值。



**6.2 Receipt数据结构**

Q1: 假设某个全节点要打包一些交易到一个区块里面，这些交易里有一些是对智能合约的调用，那么这个全节点是应该先把智能合约都执行完之后再去挖矿呢？还是先挖矿获得记账权再去执行智能合约？

A1: 先执行智能合约再挖矿，以太坊挖矿需要尝试各种不同的nonce值，找到一个符合要求的，计算哈希的时候要用到blockheader的内容，包含三棵树的根哈希值，只有执行完区块中的所有交易包括智能合约交易，这样才能更新这三棵树，知道三个根哈希值，blockheader的内容才能确定，然后才能尝试各个nonce挖矿。

Q2:全节点在收到一个对智能合约的调用的时候，要一次性先把这个调用可能花掉的最大汽油费从发起这个调用的账户扣掉，这个具体是怎么操作的？

A2:三棵树，状态树，交易树和收据树都是全节点在**本地**维护的数据结构，状态树记录了每个账户的状态，包括账户余额，汽油费是全节点收到调用的时候从本地维护的数据结构里把他账户的余额减掉就行了，**只有区块发布之后本地修改才会变成外部可见的，区块链共识**。

Q3:矿工在挖矿执行智能合约的过程中消耗了很多本地资源，但是并没有获得记账权，没有出块奖励，也不会得到汽油费奖励，怎么办？

A3: 没有办法，以太坊中就是没有补偿，还需要把别人发布的区块里的交易在本地执行一遍，以太坊规定要验证发布区块的正确性，每个全节点要独立验证，把别人发布的交易区块在本地执行一遍，更新三棵树的内容算出根哈希值，再和发布的新区块的根哈希值比较是否一致。这种机制下挖矿慢的矿工就特别吃亏，汽油费的设置本来是对矿工执行智能合约所消耗的资源的一种补偿，但是这种补偿只有挖到矿的矿工才能得到，其他矿工得不到。

Q4：上述问题会造成什么影响？如何改进？

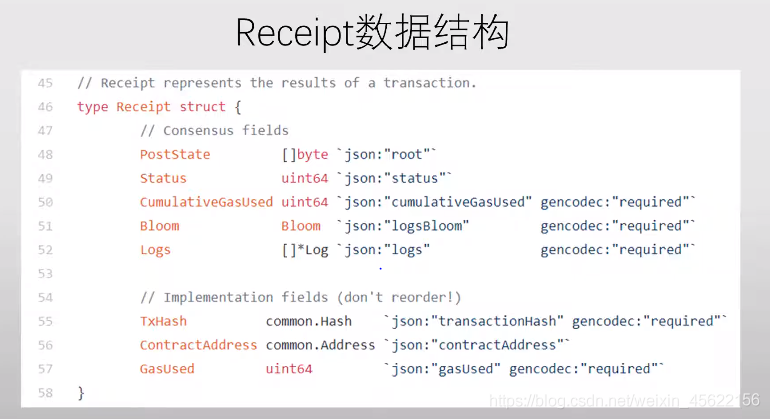
A4:会直接威胁到区块链的安全，区块链的安全保证是来自所有全节点独立验证发布的区块的合法性，这样少数有恶意的节点才没有办法篡改这些内容，如果一些矿工想不通，不给钱就不验证了，这样就会危及到区块链的安全。这样是不可行的，因为如果跳过验证步骤，以后就没法挖矿了，因为验证的时候需要把区块的交易都执行一遍，更新本地的三棵树，获取最新的根哈希值，如果不验证的话，本地三棵树的内容没有办法更新，以后就没办法发布新的区块了。因为发布的区块没有三棵树的内容，只是块头里有个根哈希值，所以没有办法不验证的。在矿池里，矿工本身就不验证了，有一个全节点pool manager负责统一验证，矿工相信全节点验证的正确性，全节点分配给矿工看到的是puzzle的内容，puzzle是全节点跟着区块链跟新得来的。

Q5：发布到区块链上的交易是不是都是成功执行的？如果智能合约在执行中出现错误，要不要也发布在区块链上？

A5: 执行发生错误的交易也要发布到区块链上，否则没有办法扣掉汽油费(或者说矿工没法获得这笔汽油费)。

Q6：怎么知道这个交易是执行成功了呢？

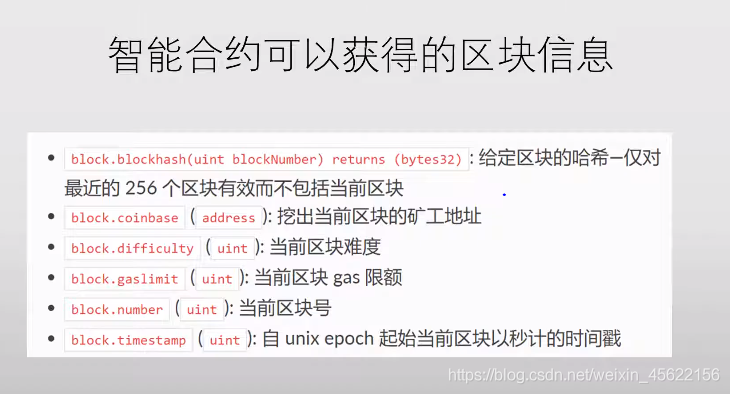
A6:三棵树里面，每个交易执行完之后形成一个收据，下图是收据的内容，其中status域会告诉你这个交易的执行情况。



Q7：智能合约支不支持多线程？多核并行处理。

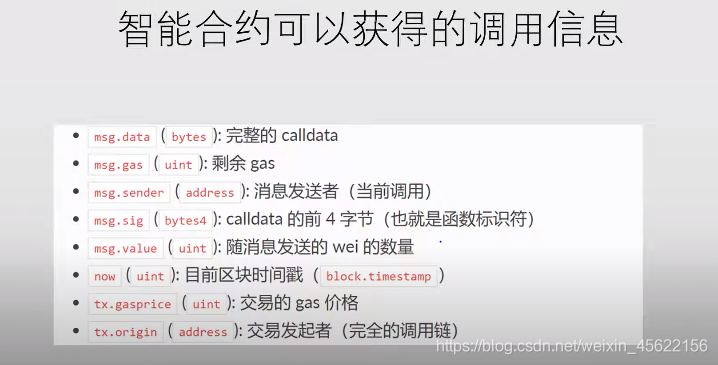
A7:以太坊是一个交易驱动的状态机，这个状态机必须是完全确认性的，即给定一个智能合约，面对同一种输入，产生的输出或者是转移到下一个地方的状态必须是完全确定的，因为所有的全节点都得执行同一组操作，到达同一个状态，要验证。如果状态不确定的话那三棵树的根哈希值根本对不上，所以必须完全确定才行。多线程的问题在于，多个核对内存访问顺序不同，执行结果有可能是不确定的，所以solidity是不支持多线程的。除了多线程，其他所有可能造成结果不确定的操作也都不支持，比如产生随机数。所以以太坊中没有办法真正产生随机数，只能产生伪随机数，否则的话又会出现前面的问题，每个全节点执行完一遍得到的结果都不一样。

智能合约的执行必须是确定性的，这也就导致了智能合约不能像通用的编程语言那样通过系统调用得到一些system call的一些环境信息，因为每个全节点的执行环境 不是完全一样的，所以他只有通过一些固定的变量的值能够得到一些状态信息，这个表格就是智能合约能够得到的区块链的一些信息。

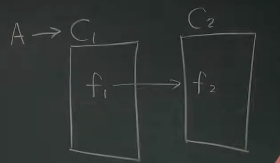


1. 智能合约的其他信息

**7.1 可获得的调用信息**



注意：msg.sender指的是发起这个调用的用户，和tx.origin所指的交易发起者是不一样的。比如说外部账户A，调用合约C1，合约中有一个函数f1，f1又调用另外一个合约C2，里面有一个函数f2，那么对这个f2函数，msg.sender是C1合约，tx.origin是账户A。



msg.gas是当前这个调用还剩下多少汽油费，这个决定了我还能做哪些操作。包括你要想再调用别的合约，前提是还有足够的汽油费剩下来，msg.data就是数据域，里面写了调用的函数和这个函数的参数取值，msg.signature是msg.data的前四个字节，就是函数标识符，调用的是哪个函数。now和timestamp是一个意思，智能合约里没有办法获得很精确的时间，只能获得跟这个当前区块的一些信息时间。

**7.2 地址类型**

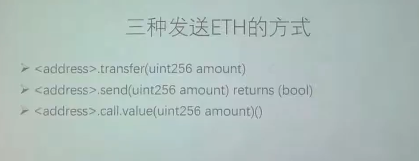


第一个是成员变量：就是成员账户的余额balance, uint256是成员变量的类型，不是函数调用/参数，单位是比较小的。

剩下的都是成员函数，成员函数的语义和直观理解不太相同，addr.transfer(12345)是当前这个合约C向这个地址转入多少钱。addr.call是指当前这个合约发起一个调用，调用的是addr这个合约。

Q8: 向一个函数转账，这个函数没有定义fallback函数，引起错误会不会连锁回滚？

A8:这个取决于怎么转账。共有三种转账方法，transfer, send,还有 call.value都可以发送ETH，但是transfer 和 send这两个是专门用来转账的函数，区别在于transfer会导致连锁性回滚，类似直接调用的方法，失败的时候抛出异常；send返回一个False，不会导致连锁式回滚；call也是可以转账的，call.value(转账金额)(调用的函数，可为空)，不过call的本意是发动函数调用的，但是也可以用来转账，这个也是不会引起连锁式回滚，返回False。另外一个区别是transfer和send这两个在发起调用的时候只给了一点汽油，汽油是2300个单位，非常少的，收到转账的合约基本上干不了别的事，也就写一个log，而call呢是把当前这个调用剩下的所有汽油都发过去了，比如call所在的这个合约它本身被外面调用的时候可能还剩8000个汽油，然后他去调别的合约如果是用call这种方法转账就把剩下的汽油都发过去了。

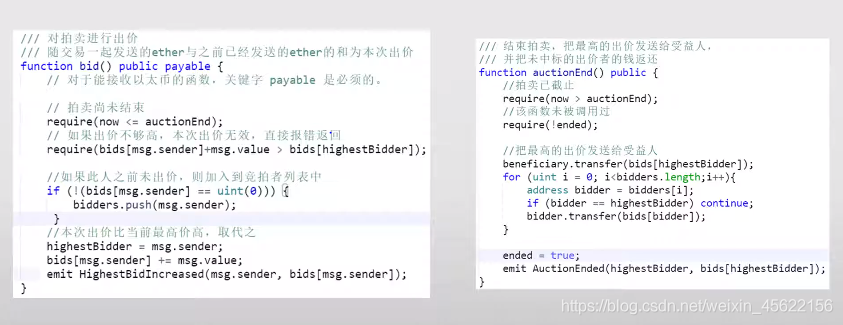


1. 拍卖例子



拍卖受益人就是拍卖前物品的所有者。

拍卖规则：拍卖结束之前每个人都可以去出价去竞拍，竞拍的时候为了保证诚信，需要把竞拍的价格相应的以太币发过去，比如出一百个以太币，你用bid函数竞拍的时候，要把100个以太币发送到智能合约，并锁在里面直到拍卖结束，不允许中途退出，可以加价，拍卖结束之后，highestBidder的出价的钱数会给受益人beneficiary，受益人应该把拍卖物也给最高出价人。其他没有竞拍成功的人可以把钱再取出来。竞拍可以多次出价，补差价发到智能合约里就可以，出价有效的话必须保证加价之后的出价高于之前的最高出价，否则就是无效(非法)的。constructor构造函数在合约创建的时候会记录受益人是谁，结束时间是什么时候。



上面两个是拍卖用的两个函数。

左边是竞拍bid函数，竞拍的时候发起一个交易调用拍卖合约中的bid函数，bid虽然没有参数，但是在msg.value发起这个调用的时候转账转过去的以太币数目，就是出的竞拍价格。

1. 先查询一下拍卖是否结束，如果已经结束还参与拍卖则抛出异常。

2. 查一下上一次的出价加上当前调用所发过去的以太币大于最高出价，如果是以前没有出价过，第一部分就是0。bids是个哈希表，solidity里的特点是如果要查询的键值不存在，则返回默认值为0。所以如果是以前没有出价过，第一部分就是0。第一次拍卖的时候把拍卖者的信息放在bidder数组里，因为solidity不支持遍历，要遍历哈希表必须保存一下包含哪些元素，然后记录一下新的最高出价人是谁，写一些日志之类的。

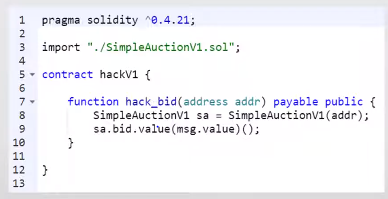
右边是拍卖结束的函数，首先查一下拍卖是否已经结束了，如果拍卖还没有结束，有人调用这个函数就是非法的，就抛出异常。第二行判断这个函数是不是被调用过，如果调用过就不用再调一遍了。第三行把最高出价人的钱转给受益人，对于拍卖没有成功的人，最后循环把金额退回给bidder。然后标注一下这个函数已经执行完了，写一个log。

**智能合约的代码是储存在data域里面的，矿工把智能合约发布到区块链上之后返回给你一个合约的地址，然后这个合约就在区块链上了，所有人都可以调用。任何人出价竞拍调用Bid函数的操作都需要矿工发布在区块链上。**

但是这里有个问题就是AuctionEnded函数必须有人调用右边函数才会执行，执行之后才会结束，solidity语言没有办法把他设置成为拍卖结束之后自动执行end。

**该智能合约存在的问题**

假设有一个人通过下图的合约账户参与竞拍，会有什么结果？



这个合约只有一个函数，hack\_bid，参数是拍卖合约的地址，把它转成拍卖合约的实例，然后调用拍卖合约的bid函数，把钱发送过去。这是一个合约账户，合约账户不能自己发起交易，得有一个黑客从他自己的外部账户发起一个交易，调用这个合约的hack\_bid函数，这个函数再调用拍卖合约的bid函数，把他自己收到的转过来的钱，黑客外部转过来的钱，再转给拍卖合约中的bid函数就参与拍卖了。

黑客外部账户对于拍卖合约是不可见的，拍卖合约能看到的只是黑客合约，进行拍卖的合约账号在进行退款时由于此合约账户没有设置任何收款函数，因此拍卖账户应该调用此账户的fallback函数，但是这个合约账户没有定义fallback函数，会调用失败并抛出异常，transfer函数会引起连锁式回滚，导致转账操作失败收不到钱。转账的过程是全节点执行到beneficiarytransfer的时候把相应账户的余额进行了调整，所有的solidity语句即智能合约执行过程中的任何语句对状态的修改该的都是本地的状态和本地的数据结构。所以这个循环当中不论是排在黑客合约顺序前面还是后面都是在改本地数据结构，只不过排在后面的bidder根本没有机会来得及执行，然后整个都回滚了，就好像这个智能合约从来没有执行，所以所有人都收不到钱。出现这种情况怎么办？

回答是没有办法，code is law，智能合约的规则是由代码逻辑决定的，代码一旦发布到区块链上就改不了了，这样的好处是没有人能够篡改规则，坏处是出现漏洞也无法修改。智能合约如果设计的不好的话有可能把以太币永久的锁起来谁也取不了。有点像irrevocable trust不可撤销的信托。

另一种解决思路是：能不能给智能合约留个后门给开发者用来修复bug？

构造函数加一个域owner，记录一下owner是谁，然后对这个owner的地址允许他做一些类似系统管理员的操作，比如可以任意转账。出现Bug之后超级管理员就可以把锁定的钱转出来。但这样做的前提是所有人应该信任这个人，否则他有可能携款逃走。

**那有什么其他改进方法吗？**

下面是一种新的解决思路，把前面的auctionend函数拆成两个函数，左边是withdraw右边是beneficiary。



withdraw是说不用循环了，每个竞拍失败的人自己调用withdraw函数把钱取出来。

左边：判断这个人是不是最高出价者，是的话不能退钱。接着判断自己的账户余额是不是正的(amount就是账户余额)，if 把账户余额转给msg.sender，就发起调用的人，然后把账户余额清0，免得下次再取钱。

右边：把最高出价给受益人。

**这样可以了吗？**



右边是黑客合约，hack\_bid就和前面的合约hack\_bid是一样的，通过调用拍卖合约的bid函数参与竞拍，hack\_withdraw就在拍卖结束的时候调用withdraw函数，把钱取回来。

问题在于右边最后一个函数fallback函数又把钱取了一遍，hack\_withdraw调用拍卖合约的withdraw函数的时候，左边执行到if(msg.sender)会向黑客合约转账，msg.sender就是黑客的合约，把他当初出价的金额转给他，右边合约最下面又调用了拍卖函数的withdraw函数去取钱，这里的msg.sender就是拍卖合约，因为是拍卖合约把钱转给这个合约的，左边的拍卖合约又开始执行到if，再转一次钱。注意黑客合约账户的清零的操作（左下角），只有在转账交易完成之后才进行，但是前面的转账交易已经陷入到和黑客合约的递归调用当中，根本执行不到清零后面，导致黑客按照自己的出价价格不停地从拍卖合约中取钱，只有第一次是自己的出价，其他都是合约里面的。

这个递归重复到下面三种情况任何一种发生：

1）拍卖合约上的余额不够了，不支持这样的转账语句

2）汽油费不够了，每次递归调用还是消耗汽油费的

3）调用栈溢出了

在右下角黑客合约的fallback函数判断一下拍卖合约的余额还足以支持转账，当前调用的剩余汽油msg.gas还有6000个单位以上，调用栈的深度不超过500，那么就再发起一轮攻击。

**如何解决这种重入攻击**

可以先清0再转账，和第二版的右边写法一致，转账不成功再回复余额；也可以不用call.value来转账，换成send或者transfer。先清0再转账，send和transfer有一个特点就是转账的时候发送过去的汽油费只有2300个单位，不足以让接收的合约再发起新的调用，只够写一个log而已。

