# 区块链记录

## 1：难度计算

diff(genesis) = 2^32

diff(block) = diff.block.parent +floor(diff.block.parent / 1024) \*

1 ifblock.timestamp - block.parent.timestamp < 9 else

-1 ifblock.timestamp - block.parent.timestamp >= 9

难度更新规则的设计目标如下：

• 快速更新：区块间的时间应该随着hash算力的增减而快速调整；

• 低波动性：如果Hash算力恒定，那么难度不应剧烈波动；

• 简单：算法的实现应相对简单；

• 低内存：算法不应依赖于过多的历史区块，要尽可能少的使用”内存变量“。假设有最新的十个区块，将存储在这十个区块头部的内存变量相加，这些区块都可用于算法的计算；

• 非开发性：算法不应让矿工有过多篡改时间戳或者矿池、反复添加或删除算力的能力，以使他们的收益最大化。

我们当前的算法在低波动性和非开发性上并不理想，至少我们计划切换时间戳比较父区块和祖父区块，所以矿工只有在连续挖2个区块时，才有动力去修改时间戳。另一个更强大的模拟公式：https://github.com/ethereum/economic-modeling/blob/master/diffadjust/blkdiff.py

## 2：GAS和费用

比特币中所有交易大体相同，因此它们的网络成本可以建成一个模型。以太坊中的交易要更复杂，所以交易费用需要考虑到账户的许多方面，包括宽带费用，存储费用和计算费用。尤其重要的是，以太坊编程语言是图灵完备的，所以交易会使用任意数量的宽带、存储和计算成本。这就可能会导致在计算成本过程中，突遭停电而计算被迫中止。

以太坊交易费用的基本机制如下：

• 每笔交易必须指明一定数量的gas（即指定startgas的值），以及支付每单元gas所需费用（即gasprice），在交易执行开始时，startgas \* gasprice 价值的以太币会从发送者账户中扣除；

• 交易执行期间的所有操作，包括读写数据库、发送消息以及每一步的计算都会消耗一定数量的gas；

• 如果交易执行完毕，消耗的gas值小于指定的限制值，则交易执行正常，并将剩余的gas值赋予变量gas\_rem ; 在交易完成后，发送者会收到返回的gas\_rem \* gasprice 价值的以太币，而给矿工的奖励是（startgas -gas\_rem）\* gasprice价值的以太币；

• 如果交易执行中，gas消耗殆尽，则所有的执行恢复原样，但交易仍然有效，只是交易的唯一结果是将startgas \* gasprice 价值的以太币支付给矿工，其他不变；

• 当一个合约发送消息给另一个合约，可以对这个消息引起的子执行设置一个gas限制。如果子执行耗尽了gas，则子执行恢复原样，但gas仍然消耗。

上述提到的几点都是必须满足的，例如：

• 如果交易没有指定gas限制，那么恶意用户就会发送一个有数十亿步循环的交易。没有人能够处理这样的交易，因为处理这样的交易花的时间可能很长很长，从而无法预先告知网络上的矿工，这会导致拒绝服务的漏洞产生。

• 替代严格的gas计数、时间限制等机制的方案不起作用，因为它们太主观了；

• startgas \* gasprice 的整个值，在开始时就应该设置好，这样不至于在交易执行中因gas不够而造成交易终止。注意，仅仅检查账户余额是不够的，因为账户可以在其他地方发送余额。

• 如果在gas不够的情况下，交易执行没有恢复操作（回滚），合约必须采用强有力的安全措施来防止合约发生变化。

• 如果子限制不存在，则恶意账户会通过与其他账户达成协议来对它们采取拒绝服务攻击。在计算开始时插入一个大循环，那么发送消息给受害合约或者受害合约的任何补救尝试，都会使整个交易死锁。

• 要求交易发送者而不是合约来支付gas，这样大大增加了开发人员的可操作性。以太坊早期的版本是由合约来支付gas的，这导致了一个相当严重的问题：每个合约必须实现“守护”代码，确保每个传入的消息有足够的以太币供其消耗。

gas消耗计算有以下特点：

• 对于任何交易，都将收取21000gas的基本费用。这些费用可用于支付运行椭圆曲线算法所需的费用。该算法旨在从签名中恢复发送者的地址以及存储交易所花费的硬盘和带宽空间。

• 交易可以包括无限量的“数据”。虚拟机中的某些操作码，可以让合约允许交易对这些数据的访问。数据的固定消耗计算是：每个零字节4gas，非零字节68gas。这个公式的产生是因为合约中大部分的交易数据由一些列的32字节的参数组成，其中多数参数具有许多前导零字节。该结构看起来似乎效率不高，但由于压缩算法的存在，实际上还是很有效率的。我们希望此结构能够代替其他更复杂的机制：这些机制根据预期字节数严格包装参数，从而导致编译阶段复杂性大增。这是三明治复杂模型的一个例外，但由于成本效益比，这也是合理的模型。

• 用于设置账户存储器的操作码SSTORE的消耗是：

1.将零值改为非零值时，消耗20000gas；

2.将零值变成零值，或非零值变非零值，消耗5000gas；

3.将非零值变成零值，消耗5000gas，加上交易执行成功后退回的20000gas。

退款金额上限是交易消耗gas总额的50%。这样设置会激励人们清除存储器。我们注意到，正因为缺乏这样的激励，许多合约造成了存储空间没有被有效使用，从而导致了存储快速膨胀；为存储收取费用提供了很多好处，同时不会失去合约一旦确立就可以永久存在的保证。延迟退款机制是必要的，因为可以阻止拒绝服务攻击。攻击者发送一笔含有少量gas的交易，循环清除大量的存储，直到用光gas，这样消耗了大量的验证算力，但实际并没有真正清除存储或消耗大量gas。50%的上限的是为了确保获得了一定交易gas的旷工依然能够确定执行交易的计算时间的上限。

• 合约提供的消息的数据是没有成本的。因为在消息调用期间不需要实质复制任何数据，调用数据可以简单地视为指向父合约内存的指针，该指针在子进程执行时不会改变。

• 内存是一个可以无限扩展的数组，然而，每扩展32字节的内存就会消耗1gas的成本，不足32字节以32字节计。

• 某些操作码的计算时间极度依赖参数，gas开销计算是动态变化的。例如，EXP的的开销是指数级别的；复制操作码（如：CALLDATACOPY,CODECOPY, EXTCODECOPY）的开销是1+1(每复制32字节)。内存扩展的开销不包含在这里，因为它触发了二次攻击。

• 如果值不是零，操作码CALL会额外消耗9000gas。这是因为任何值传输都会引起归档节点的历史存储显著增大。请注意，实际消耗是6700，在此基础上，我们强制增加了一个自动给予接受者的gas值，这个值最小2300。这样做是为了让接受交易的钱包至少有足够的gas来记录交易。

gas机制的另一个重要部分是gas价格本身体现出的经济学原理。比特币中，默认的方法是采取纯粹自愿的收费方式，矿工扮演守门人的角色并且动态设置收费的最小值。以太坊中允许交易发送者设置任意数目的gas。这种方式在比特币社区非常受欢迎，因为它是“市场经济”的体现：允许矿工和交易者之间依据供需关系来决定价格。然而，这种方式的问题是，交易处理并不遵循市场原则。尽管可以将交易处理看作是矿工向发送者提供的服务（这听起来很直观），但实际上矿工所处理的每个交易都必须由网络中的每个节点处理，所以交易处理的大部分成本都由第三方机构承担，而不是决定是否处理它的矿工。

当前，因为缺乏矿工在实际中的行为的明确信息，所以我们将采取一个非常简单公平的方法：投票系统，来设定gas限定值。矿工有权将当前区块的gas限定值设定在最后区块的gas限定值的0.0975% (1/1024)内。所以最终的gas限定值应该是矿工们设置的中间值。我们希望将来能够采用软分叉的方法来使用更加精确的算法。

## 3：区块结构

一个完整的区块的结构是：

[

block\_header,

transaction\_list,

uncle\_list

]

Where:

transaction\_list = [

transaction 1,

transaction 2,

...

]

uncle list = [

uncle\_block\_header\_1,

uncle\_block\_header\_2,

...

]

block\_header = [

parent hash,

sha3(rlp\_encode(uncle\_list)),

coinbase address,

state\_root,

sha3(rlp\_encode(transaction\_list)),

difficulty,

timestamp,

extra\_data,

nonce

]

每个transaction 和uncle\_block\_header 都是一张表。工作量证明数据是区块数据去除掉nonce（交易数）后的RLP编码。

uncle\_list 和 transaction\_list 分别是叔区块头和区块里的交易构成的表。nonce 和 extra\_data 都被限制为最大32字节，除了在创世区块中参数extra\_data会更大。

state\_root 是一个包含所有地址的（key, value）对的默克尔-帕特里夏树（Merkle Patricia tree ）的根， 其中每一个地址都由一个20字节二进制字符串表示。对于每个地址，储存在默克尔-帕特里夏树的 value 字段是一个对以下格式对象进行RLP串接编码形成的字符串：

[ balance, nonce, contract\_root, storage\_deposit ]

nonce 是该地址的交易数，每做一次交易都会增加1。其目的是(1)使每个交易只有一次合法的机会以防范重放攻击，(2)使得构建一个和已存合约有相同哈希的合约成为不可能（更准确地说，密码学意义上不可行）。balance 指的是合约或地址的平衡账目，以伟为单位。contract\_root 是另一个帕特里夏树的根，在该地址被一个合约控制的情况下包含该合约的内存。如果一个地址没有被一个合约控制，contract\_root 就会是一个空字符串。storage\_deposit是一个计数器，储存花费的储存费用；它的功能将在后面详细讨论。

## 4：****虚拟机****

以太坊虚拟机是执行交易代码的引擎，也是以太坊与其他系统的核心区别。请注意，虚拟机应该同“合约与消息模型”分开考虑。例如，SIGNEXTEND操作码是虚拟机的一个功能，但实际上“某个合约调用其他合约并指定子调用的gas限定值”是“合约与消息模型”的一部分。

EVM的设计目标如下：

• 简单：操作码尽可能的少并且低级；数据类型尽可能少；虚拟机的结构尽可能少；

• 结果明确：在VM规范语句中，没有任何可能产生歧义的空间，结果应该是完全确定的。此外，计算步骤应该是精确的，以便可以测量gas的消耗量；

• 节约空间：EVM组件应尽可能紧凑；

• 预期应用应具备专业化能力：在VM上构建的应用应能处理20字节的地址，以及32位的自定义加密值，拥有用于自定义加密的模数运算、读取区块和交易数据与状态交互等能力；

• 简单安全：为了让VM不被利用，应该能够容易地让建立一套gas消耗成本模型的操作；

• 优化友好：应该易于优化，以便即时编译（JIT）和VM的加速版本能够构建出来。

同时EVM也有如下特殊设计：

• 临时/永久存储的区别：

我们先来看看什么是临时存储和永久存储。

临时存储：存在于VM的每个实例中，并在VM执行结束后消失；

永久存储：存在于区块链状态层。

假设执行下面的树（S代表永久存储，M代表临时存储）：

1.A调用B；

2.B设置B.S[0]=5，B.M[0]=9 ;

3.B调用C；

4.C调用B。

此时，如果B试图读取B.S[0]，它将得到B前面存入的数据，也就是5；但如果B试图读取B.M[0]，它将得到0，因为B.M是临时存储，读取它的时候是虚拟机的一个新的实例。

在一个内部调用中，如果设置B.M[0] = 13和 B.S[0] = 17 ，然后内部调用和C的调用都终止，再执行B的外部调用，此时读取M,将会看到B.M[0] = 9(此值在上一次同一VM执行实例中设置的)，B.S[0] = 17。如果B的外部调用结束，然后A再次调用B，将看到B.M[0]= 0，B.S[0]= 17。这个区别的目的是：1.每个执行实例都分配有内存空间，不会因为循环调用而减损，这让安全编程更加容易。2.提供一个能够快速操作的内存形式：因为需要修改树，所以存储更新必然很慢。

• 栈/内存模式

早期，计算状态有三种：栈(stack,一个32字节标准的LIFO)，内存(memory，可无限扩展的临时字节数组)，存储(storage，永久存储)。在临时存储端，栈和内存的替代方案是memory-only范式，或者是寄存器和内存的混合体（两者区别不大，寄存器本质上也是一种内存）。在这种情况下，每个指令都有三个参数，例如：ADD R1 R2 R3: M[R1] = M[R2] +M[R3] 。选择栈范式的原因很明显，它使代码缩小了4倍。

• 单词大小32字节

在大多数结构中，如比特币，单词大小是4或8字节。4或8字节对存储地址或加密计算来说局限性太大了。而太大的值又很难建立相应安全的gas模型。32字节是一个理想大小，因为它足够存储下许多加密算法的实现以及地址，又不会因为太大而导致效率低下。

• 我们有自己的虚拟机

我们的虚拟机使用java、Lisp或Lua等语言开发。我们认为开发一款专业的虚拟机是值得的，因为：

1. 我们的VM规范比其他许多虚拟机简单的多，因为其他虚拟机为复杂性付出的代价更小，也就是说它们更容易变得复杂；然而，在我们的方案中每额外增加一点复杂性，都会给集约化发展带来障碍，以及潜在的安全缺陷，比如造成共识失败，这就让我们的复杂性成本很高，因而不容易造成复杂；

2. 我们的VM更加专业化，如支持32字节；

3. 我们不会有复杂的外部依赖，复杂的外部依赖会导致我们安装失败；

4. 完善的审查机制，可以具体到特殊的安全需求；对外部VM而言，这一点无论如何都是必要的。

• 使用了可变、可扩展的内存大小

固定内存的大小是不必要的限制，太小或太大都不合适。如果内存大小是固定的，每次访问内存都需要检查访问是否超出边界，显然这样的效率并不高。

• 栈大小没有限制

没什么特别理由！许多情况下，该设计不是绝对必要的；因为，gas的开销和区块层gas的限制总是会充当每种资源消耗的上限。

• 1024调用深度限制

许多编程语言在栈的深度过大时触发中断比在内存过载时触发中断的策略要快的多。所以区块中gas限制所隐含的限制是不够的。

• 无类型

为了简单起见，可以使用DIV, SDIV, MOD, SMOD的有符号或无符号的操作码代替（事实证明，对于操作码ADD和MUL，有符号和无符号是对等的）；转换成定点运算在所有情况下都很简单，例如，在32位深度下，a \* b -> (a \* b) / 2^32, a / b-> a \* 2^32 / b ，+, - 和 \* 在整数下不变。

VM中某些操作码的函数和作用很容易理解，但也有一些不太好理解，以下是一些特殊的原因：

• ADDMOD,MULMOD

大多数情况下，addmod(a, b, c) = a \* b % c，但在椭圆曲线算法中，使用的是32字节模数运算，直接执行a \* b % c 实际上是在执行((a \* b) % 2^256) % c会得到完全不同的结果。计算公式a\* b % c 使用32字节空间的32字节值是非常不常见且繁琐。

• SIGNEXTEND

SIGNEXTEND操作码的作用是为了方便从大的有符号整数到小的有符号整数的类型转换。小的有符号整数是很有用的，因为未来的即时编译虚拟机可能会检测长时间运行的代码块，小的有符号整数能加快处理。

• SHA3

在以太坊代码中SHA3作为安全的高强度的hash集合应用非常广泛，通常在使用存储器时需要使用Hash函数来确保安全，以防止恶意冲突，在验证默克尔树和类似以太坊的数据结构时也需要使用到Hash函数。重要的是，与SHA3的相似的hash函数，如SHA256，ECRECVOR，RIPEM160不是以操作码的形式包含在里面，而是以伪合约的形式。这样做的目的是将它们放在一个单独的类别中，如果当我们以后提出适当的“本地扩展”系统时，可以添加更多这样的合约，而不需要扩展操作码。

• ORIGIN

ORIGIN操作码由交易的发送者提供，主要的作用是允许合约退回支付的gas。

• COINBASE

COINBASE的主要作用是：1.如果使用COINBASE操作码，则允许子货币对网络安全作出贡献；2.开放使用矿工作为一个去中心化的经济体，来设置基于子共识的应用，如Schellingcoin。

• PREVHASH

PREVHASH作为一个半安全的随机来源，允许合约评估上一个区块的默克尔树状态证明，而不需要高度复杂的递归结构“以太坊轻客户端”。

• EXTCODESIZE, EXTCODECOPY

主要的作用是让合约依据模板检查其他合约的代码，甚至是在与其他合约交互前，模拟它们。

• JUMPDEST

当跳转(jump)目的地限制在几个索引时（尤其是，动态目的跳转的计算复杂度是O(log(有效挑战目的数量))，而静态跳转总是恒定的），JIT虚拟机实现起来更简单。于是，我们需要：1.对有效变量跳转目的地做限制；2.使用静态而不是动态跳转的激励方式。为了达到这两个目标，我们定下了以下规则：1.紧接着push后的跳转可以跳到任何地方，而不仅是另一个jump；2.其他的jump只能跳转到JUMPDEST。对跳转的限制是必须的，你可以通过查看代码中的先前操作来决定是静态还是动态的跳转。缺乏对静态跳转的需求是激励使用它们的原因。禁止跳转进入push数据也会加快JIT虚拟机的编译和执行。

• LOG

LOG是事件的日志。

• CALLCODE

该操作码允许合约使用自己的存储器，在单独的栈空间和内存中调用其他合约的“函数”。这样可以在区块链上灵活实现标准库代码。

• SELFDESTRUCT

允许合约删除它自己，前提是它已经不需要存在了。SELFDESTRUCT并非立即执行，而是在交易执行完之后执行。这是因为这样做可以恢复那些执行后大大增加了缓存复杂度的SELFDESTRUCT操作码。

• PC

尽管理论上不需要PC操作码，因为所有的PC操作码都可以根据将push操作的索引加入实际程序计数器来代替实现，但使用PC可以创建独立代码的位置（可复制粘贴到其他合约的编译函数，如果它们以不同索引结束，不要打断）。