北京大学肖臻老师《区块链技术与应用》公开课笔记

> 以太坊GHOST协议，对应肖老师视频：click here 全系列笔记请见：click here 由于本篇篇幅较长，因此转为两篇文章。ETH中挖矿算法篇1请见：click here **About Me:**点击进入我的Personal Page

伪代码理解以太坊挖矿算法

提醒：观看该篇建议先查看ETH挖矿算法篇1，点击click here进行跳转，本篇仅因为接上篇篇幅太长而分篇，两篇内容衔接紧密，不建议直接看本篇

- ① 生成16MB大小的Cache：

```
def mkcache(cache_size, seed):
    o = [hash(seed)]
    for i in range(1, cache_size):
        o.append(hash(o[-1]))
    return o
```

这个函数是通过seed计算出来cache的伪代码。

伪代码略去了原来代码中对cache元素进一步的处理，只展示原理，即cache中元素按序生成，每个元素产生时与上一个元素相关。

每隔30000个块会重新生成seed（对原来的seed求哈希值），并且利用新的seed生成新的cache。

cache的初始大小为16M，每隔30000个块重新生成时增大初始大小的1/128 —— 128K。

https://blog.csdn.net/Mu_Xiaoye

- ② 功能：通过Cache生成大数据集中第i个元素

```
def calc_dataset_item(cache, i):
    cache_size = cache.size
    mix = hash(cache[i % cache_size] ^ i)
    for j in range(256):
        cache_index = get_int_from_item(mix)
        mix = make_item(mix, cache[cache_index % cache_size])
    return hash(mix)
```

这是通过cache来生成dataset中第i个元素的伪代码。

这个dataset叫作DAG，初始大小是1G，也是每隔30000个块更新，同时增大初始大小的1/128——8M。

伪代码省略了大部分细节，展示原理。

先通过cache中的第i%cache_size个元素生成初始的mix，因为两个不同的dataset元素可能对应同一个cache中的元素，为了保证每个初始的mix都不同，注意到i也参与了哈希计算。

随后循环256次，每次通过get_int_from_item来根据当前的mix值求得下一个要访问的cache元素的下标，用这个cache元素和mix通过make_item求得新的mix值。注意到由于初始的mix值都不同，所以访问cache的序列也都是不同的。

最终返回mix的哈希值，得到第i个dataset中的元素。

多次调用这个函数，就可以得到完整的dataset。

https://blog.csdn.net/Mu_Xiaoye

- ③ 生成大数据集DAG中的每个元素

```
def calc_dataset(full_size, cache):
    return [calc_dataset_item(cache, i) for i in range(full_size)]
```

这个函数通过不断调用前边介绍的calc_dataset_item函数来依次生成dataset中全部full_size个元素。

https://blog.csdn.net/Mu_Xiaoye

未经许可，by Sinocifeng 禁止商用。

④ 矿工挖矿函数与轻节点验证函数

```
def hashimoto_full(header, nonce, full_size, dataset):    # 矿工
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, dataset[dataset_index])
        mix = make_item(mix, dataset[dataset_index + 1])
    return hash(mix)
# 轻节点
def hashimoto_light(header, nonce, full_size, cache):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, calc_dataset_item(cache, dataset_index))
        mix = make_item(mix, calc_dataset_item(cache, dataset_index + 1))
    return hash(mix)
```

这个函数展示了ethash算法的puzzle：通过区块头、nonce以及DAG求出一个与target比较的值，矿工和轻节点使用的实现方法是不一样的。

伪代码略去了大部分细节，展示原理。

先通过header和nonce求出一个初始的mix，然后进入64次循环，根据当前的mix值求出要访问的dataset的元素的下标，然后根据这个下标访问dataset中两个连续的的值。

（思考题：这两个值相关吗？）

最后返回mix的哈希值，用来和target比较。

注意到轻节点是临时计算出用到的dataset的元素，而矿工是直接访存，也就是必须在内存里存着这个1G的dataset，后边会分析这个的原因。

https://blog.csdn.net/Mu_Xiaoye

⑤ 矿工挖矿的主循环体

本质上为一个不断尝试Nonce的过程

```
def mine(full_size, dataset, header, target):
    nonce = random.randint(0, 2**64)
    while hashimoto_full(header, nonce, full_size, dataset) > target:
        nonce = (nonce + 1) % 2**64
    return nonce
```

这是矿工挖矿的函数的伪代码，同样省略了一些细节，展示原理。

full_size指的是dataset的元素个数，dataset就是从cache生成的DAG，header是区块头，target就是挖矿的目标，我们需要调整nonce来使hashimoto_full的返回值小于等于target。

这里先随机初始化nonce，再一个个尝试nonce，直到得到的值小于target。

https://blog.csdn.net/Mu_Xiaoye

```
def mkcache(cache_size, seed):
    o = [hash(seed)]
    for i in range(1, cache_size):
        o.append(hash(o[-1]))
    return o

def calc_dataset_item(cache, i):
    cache_size = cache.size
    mix = hash(cache[i % cache_size] ^ i)
    for j in range(256):
        cache_index = get_int_from_item(mix)
        mix = make_item(mix, cache[cache_index % cache_size])
    return hash(mix)

def calc_dataset(full_size, cache):
    return [calc_dataset_item(cache, i) for i in range(full_size)]

def hashimoto_full(header, nonce, full_size, dataset):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, dataset[dataset_index])
        mix = make_item(mix, dataset[dataset_index + 1])
    return hash(mix)

def hashimoto_light(header, nonce, full_size, cache):
    mix = hash(header, nonce)
    for i in range(64):
        dataset_index = get_int_from_item(mix) % full_size
        mix = make_item(mix, calc_dataset_item(cache, dataset_index))
        mix = make_item(mix, calc_dataset_item(cache, dataset_index + 1))
    return hash(mix)

def mine(full_size, dataset, header, target):
    nonce = random.randint(0, 2**64)
    while hashimoto_full(header, nonce, full_size, dataset) > target:
        nonce = (nonce + 1) % 2**64
    return nonce
```

⑥ 所有函数的汇总

这里是整个流程的伪代码，同时分析矿工需要保存整个dataset的原因。

红色框标出的代码表明通过cache生成dataset的元素时，下一个用到的cache中的元素的位置是通过当前用到的cache的元素的值计算得到的，这样具体的访问顺序事先不可预知，满足伪随机性。

为何验证只需保存cache，而矿工需要保存大数组DAG?

由于矿工需要验证非常多的nonce，如果每次都要从16M的cache中重新生成的话，那挖矿的效率就太低了，而且这里面有大量的重复计算：随机选取的dataset的元素中有很多是重复的，可能是之前尝试别的nonce时用过的。所以，矿工采取以空间换时间的策略，把整个dataset保存下来。轻节点由于只验证一个nonce，验证的时候就直接生成要用到的dataset中的元素就行了。

https://blog.csdn.net/Mu_Xiaoye

本文由 Sinocifeng 林上商 许可，

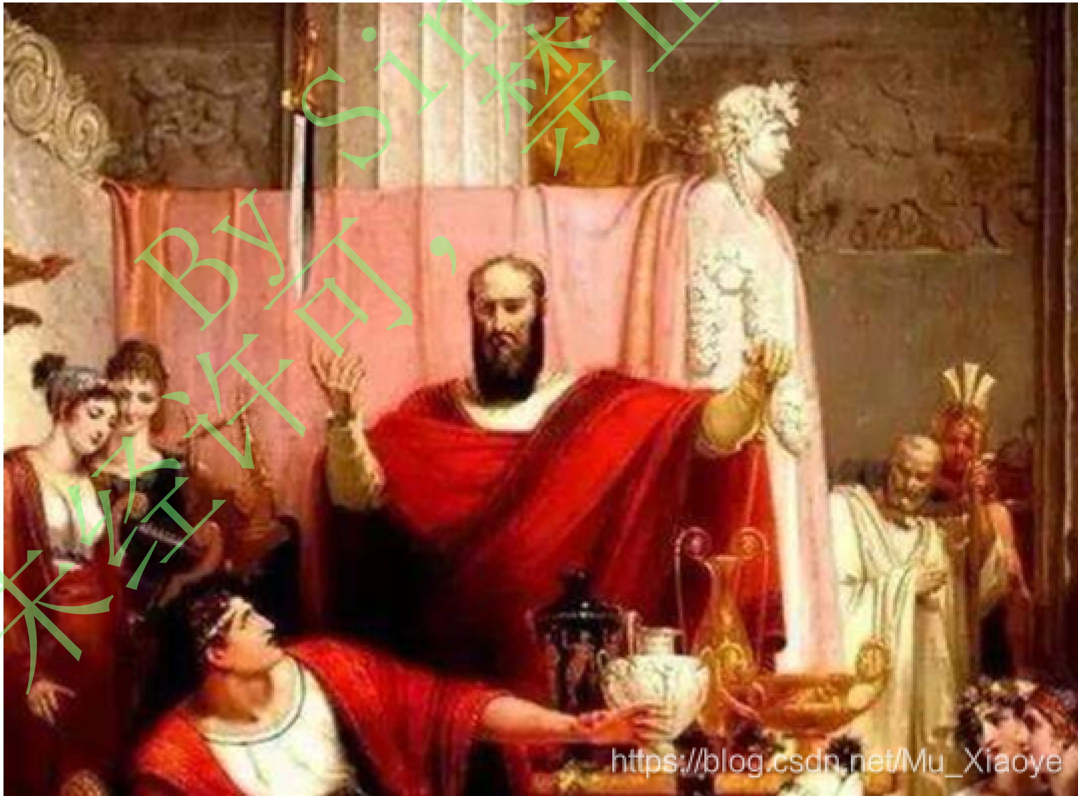
目前以太坊挖矿以GPU为主，可见其设计较为成功，这与以太坊设计的挖矿算法(Ethash)所需要的大内存具有很大关系。 1G的大数组与128k相比，差距8000多倍，即使是16MB与128K相比，也大了一百多倍，可见对内存需求的差距很大(况且两个数组大小是会不断增长的)。 当然，以太坊实现ASIC Resistance除了挖矿算法设计之外，还存在另外一个原因，即其预期从**工作量证明(POW)**转向**权益证明(POS)**

权益证明(POS: Proof of State)

> **<2022年10月补充>**2022年9月5日，Vitalik Buterin 在推特上表示，合并是“以太坊生态系统的重要时刻”。在世界标准时间 06:43 区块链达到终端总难度 58,750,000,000T 后，以太坊 (ETH) 已迁移到权益证明网络。

> 也就是说，当你看到这段文字的时候，POW已经切换到了POS。但该文是2020年就写下来的，考虑到能看到这里的人是想要学习和了解关于区块链的知识，因此对文中内容就不再修改了。感谢评论区@qq_31449773 于2022.08.29在评论区进行补充。
>
>
>

权益证明：按照所占权益投票进行共识达成，类似于股份制有限共识按照股份多少投票，权益证明不需要挖矿。 而这对于ASIC矿机厂商来说，就好比一把悬在头上的达摩克利斯之剑。因为ASIC芯片研发周期很长，成本很高，如果以太坊转入权益证明，这些投入的研发费用将全部白费(ASIC矿机只能用于挖特定的加密货币)



By Sinocifeng 未经许可，禁止商用。

但实际上，以太坊目前仍然是POW挖矿共识机制。在设计之初，以太坊开发者就设想要从POW转向POS，并为了防止有矿工不愿意转埋下了一颗"难度炸弹"。但截至目前，以太坊仍然基于POW共识机制。

> 其实很多时候，面对一些问题转换思路就能得到很好的解决方案。如这里，如果按照原本思想，通过不断改进挖矿算法来达成ASIC Resistance，无疑是比较难的。而这里通过不停宣传要转向POS来不断吓阻矿工，使得矿工不敢擅自转入ASIC挖矿，从而实现了ASIC Resistance。 战忽局：喵喵喵？？？？谁在叫我？？？**（看来我们发现了一门新的学科：战略忽悠/恐吓(he四声)学。有空一起学习呀，滑稽.jpg。）**

预挖矿(Pre-Mining)

以太坊中采用的预挖矿的机制。这里“预挖矿”并不挖矿，而是在开发以太坊时，给开发者预留了一部分货币。以太坊的早期开发者，目前就很有钱了。（致富新思路！果然技术=Money呀） 而比特币并未采用这一模式，所有比特币都是通过挖矿产生的。但早期挖矿难度容易，所有中本聪本人本来就有很多币(但没花....) 和Pre-Mining对应，还有Pre-Sale，Pre-Sale指的是将预留的货币出售掉用于后续开发，类似于拉风投或众筹。目前，各类加密货币很多，存在一部分货币就在采用Pre-Sale来获取资金，如果此时买入，后续如果该货币取得成功，同样可以获得很大收益，但真正成功的货币只占少数，这就是其风险性。

以太坊统计数据

1. 以太坊中以太币供应量(2018年) 以太坊市值500多亿美元，你酸了吗？ 饼状图中，蓝色部分都是Pre-Mining产生的（接近3/4），可见掌握技术有多么重要。黑色部分为出块奖励产生的以太币，绿色为叔父区块产生的奖励以太币。

以太坊总供应量

ETHER DISTRIBUTION OVERVIEW

Genesis (60M Crowdsale+12M Other):	72,009,990.50 Ether
+ Mining Block Rewards:	25,903,969.97 Ether
+ Mining Uncle Rewards:	1,818,301.25 Ether
= Current Total Supply	**99,732,261.72 Ether**

Data Source: Total Eth Supply API

$ PRICE PER ETHER

In USD:	$518.77
In BTC:	0.07321

Data Source: CryptoCompare

99,732,261.72
Total Ether Supply

$51,738,105,411
Market Capitalization

Breakdown By Supply Types

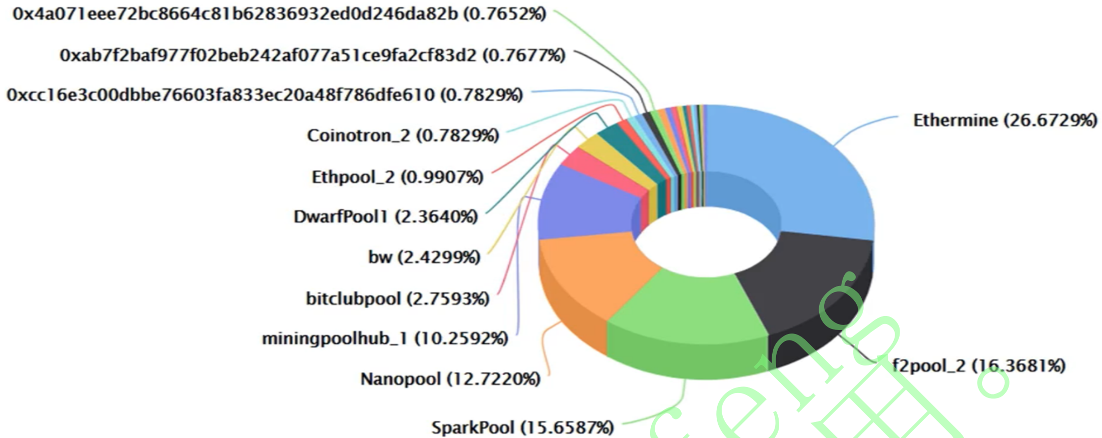
Genesis (72009990.49948 ETH)
Block Rewards (25903969.9688 ETH)
Uncle Rewards (1818301.25 ETH)

Source:Etherscan.io

https://blog.csdn.net/Mu_Xiaoye

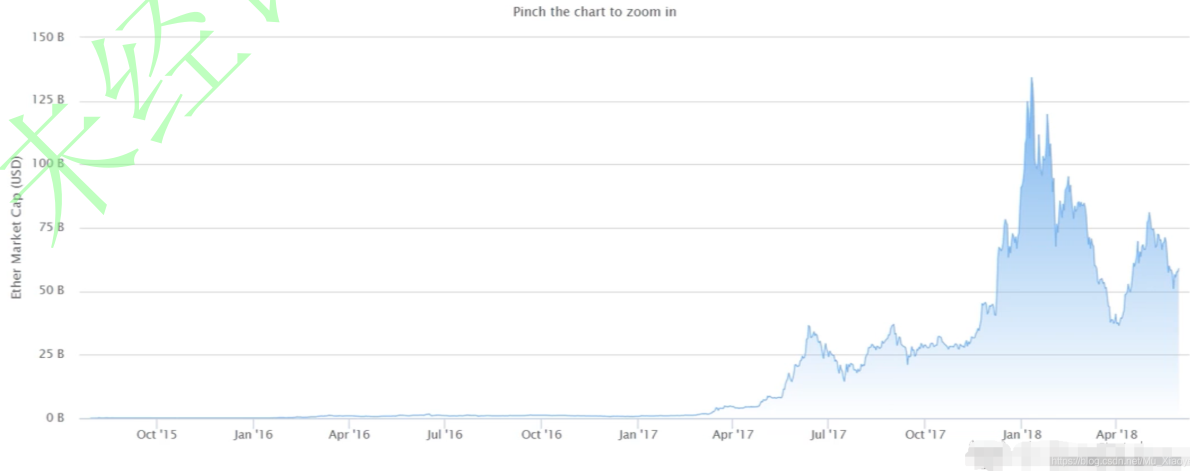
By …ifeng 未经许可，禁止商用。

2. 最大的25个矿池挖矿算力比重(2018年)



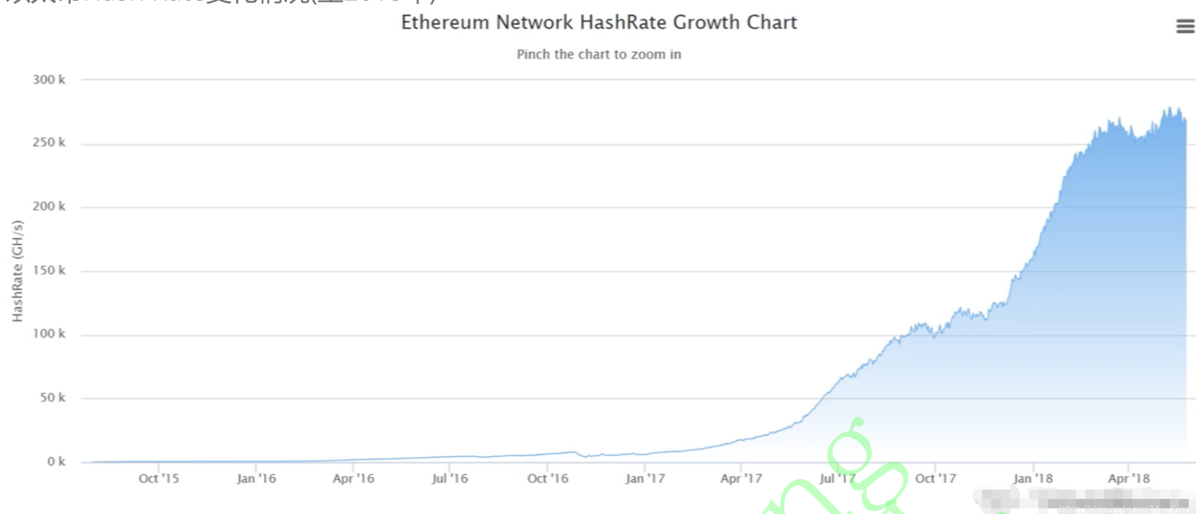
3. 以太币价格变化情况(至2018年) 可见，2017年以太坊才开始大涨，是否感觉错过1个亿?

4. 以太币市值变化情况(至2018年)



By Sinocifeng. 未经许可，禁止商用。

5. 以太币Hash Rate变化情况(至2018年)



其他观点

本篇中挖矿算法设计一直趋向于让大众参与，这一才是公平的。且由于参与人员的分散，算力分散，也进一步使得系统更安全。 但同样一件事物，从不同观点看就有不同的看法。也有人认为**让普通计算机参与挖矿是不安全的，像比特币那样，让中心化矿池参与挖矿才是安全的。为什么呢？** 因为要攻击系统，需要购入大量只能进行特定货币挖矿的矿机通过算力进行强行51%攻击，而攻击成功后，必然导致该币种的价值跳水，攻击者投入的硬件成本将会全部打水漂。而如果让通用计算机也参与挖矿，发动攻击成本便大幅度降低，目前的大型互联网公司，将其服务器聚集起来进行攻击即可，而攻击完成后这些服务器仍然可以转而运行日常业务。因此，也有人认为，在挖矿上面，ASIC矿机"一统天下"才是最安全的方式。

> 因此可见，世间事物并不是非黑即白的，同样一个事物，从不同角度来看，就会有不同的结论，而这些结论可能是互相对立的。处于世间，我们也应当注意到这一点，跳出自己固有认知，站在其他角度来思考问题，消弥分歧。 **附上我很喜欢的一段话：** *浮云骑士在想 面前的这个对手 不是和自己一样的人吗？ 有自己的家人和朋友 喜欢美食和安逸的生活 打他他会疼，会受伤，还会死 可他为什么要战斗？ 这里一定有一个属于他自己的原因 只要我们坐下来 像朋友那样找出这个原因 战斗就变得毫无意义*