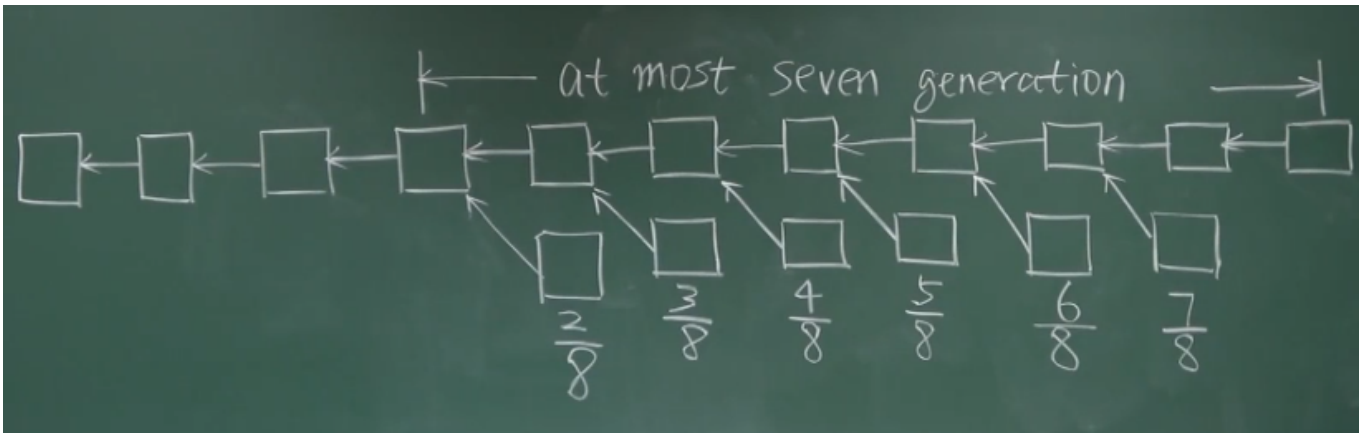
GHOST

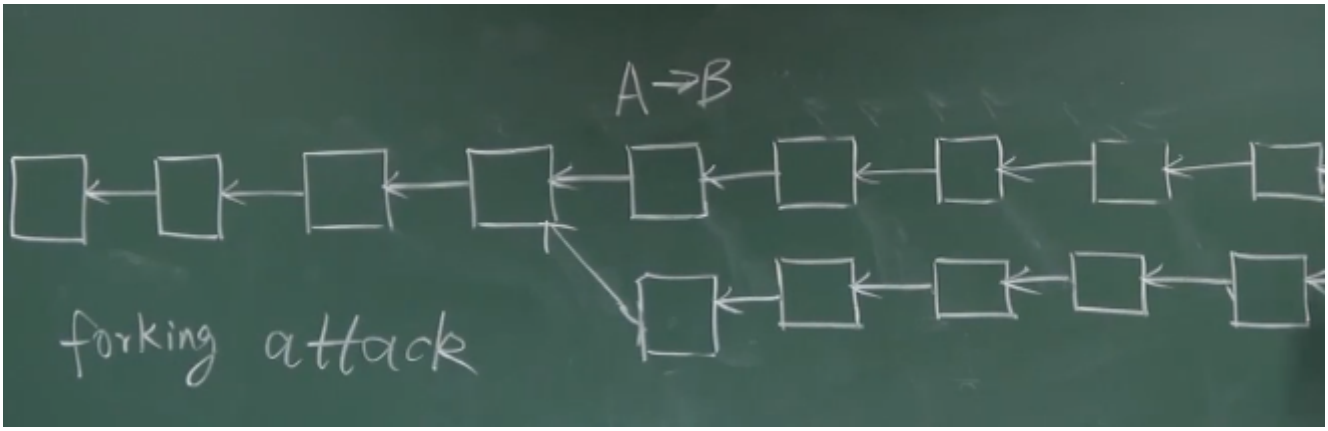
如果在10s左右出块的情况下依然沿用BTC的共识算法会产生大量的分叉区块。而大矿池中心凭借着算力优势可以促使下一个区块大概率产生在自己所挖的分叉上，这造成了一定程度的中心化。

叔父区块

只承认前6个区块(从当前区块起7代以内有共同祖先)，从7/8到2/8由近到远递减。 *拉拢叔父残害兄弟* 每拉拢一个叔父还可以得到当前出块奖励的1/32.目前是3x1/32



但是下面这种情况是不承认的，因为大大降低了分叉攻击的代价（forking attack）



所以以太坊只承认分叉后的第一个区块。

挖矿算法

为了one CPU ,one vote理念。提出了ASIC resistance。引入了内存需求，memory hard mining puzzle。

Litecoin的mining puzzle：scrypt（基于内存的一个hash函数）

问题是，对轻节点来说也是memory hard。正常应该是difficult to solve，easy to verify。但是memory hard对于验证来说也需要大量内存。

加密货币的冷启动：发币初期，鲜为人知，只有少数人挖矿系统是不安全的。 是大量用户涌入新币进行挖矿就是冷启动。

ETH的mining puzzle：ethash，设计一大一小的数据集，16M的cache，1G的dataset叫DAG。目的是便于验证。轻节点保持cache，全节点1G的dataset。首先产生seed，由seed产生一个16M的数组，再由数组产生大

的数据集。求解时用大数据组。

```
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。

```
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。

```
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)
```

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。

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。

```
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
```

```
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
```

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。

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。

挖矿难度

区块难度D(H)

$$D(H) \equiv \begin{cases} D_0 & \text{if } H_i = 0 \\ \max\left(D_0, P(H)_{H_d} + x \times \varsigma_2\right) + \epsilon & \text{otherwise} \end{cases}$$

where:

$$D_0 \equiv 131072 \tag{42}$$

➢ 参数说明

- D(H)是本区块的难度，由基础部分$P(H)_{H_d}+x\times\varsigma_2$和难度炸弹部分$\epsilon$相加得到。
- $P(H)_{H_d}$为父区块的难度，每个区块的难度都是在父区块难度的基础上进行调整。
- $x\times\varsigma_2$用于自适应调节出块难度，维持稳定的出块速度。
- 基础部分有下界，为最小值$D_0=131072$
- ϵ表示设定的难度炸弹。

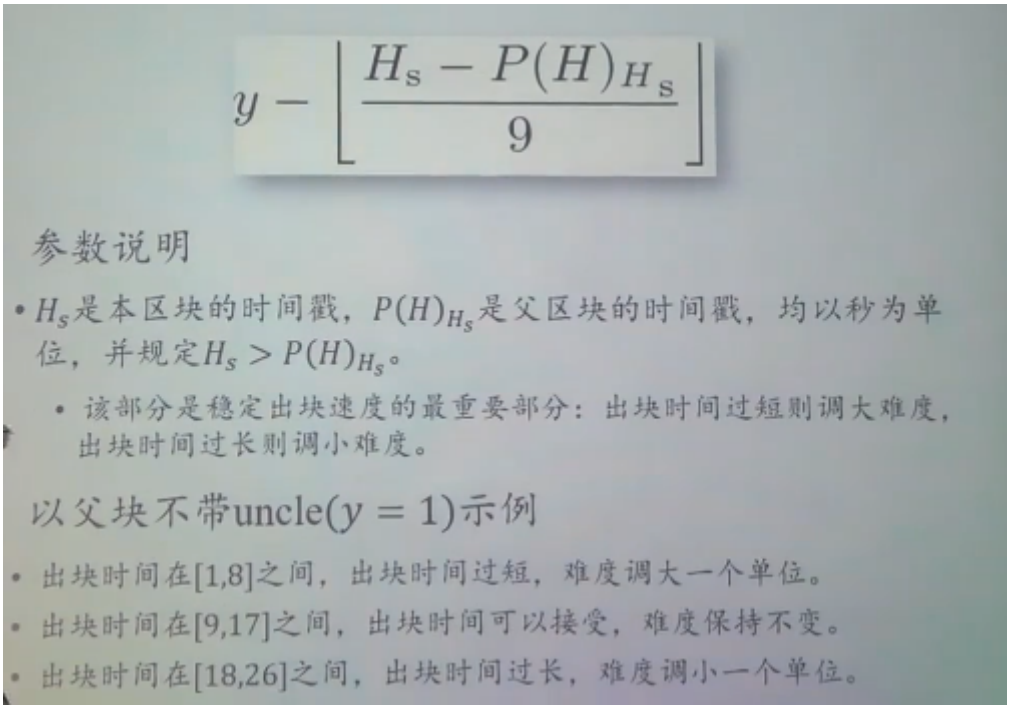
自适应难度调整 $x\times\varsigma_2$

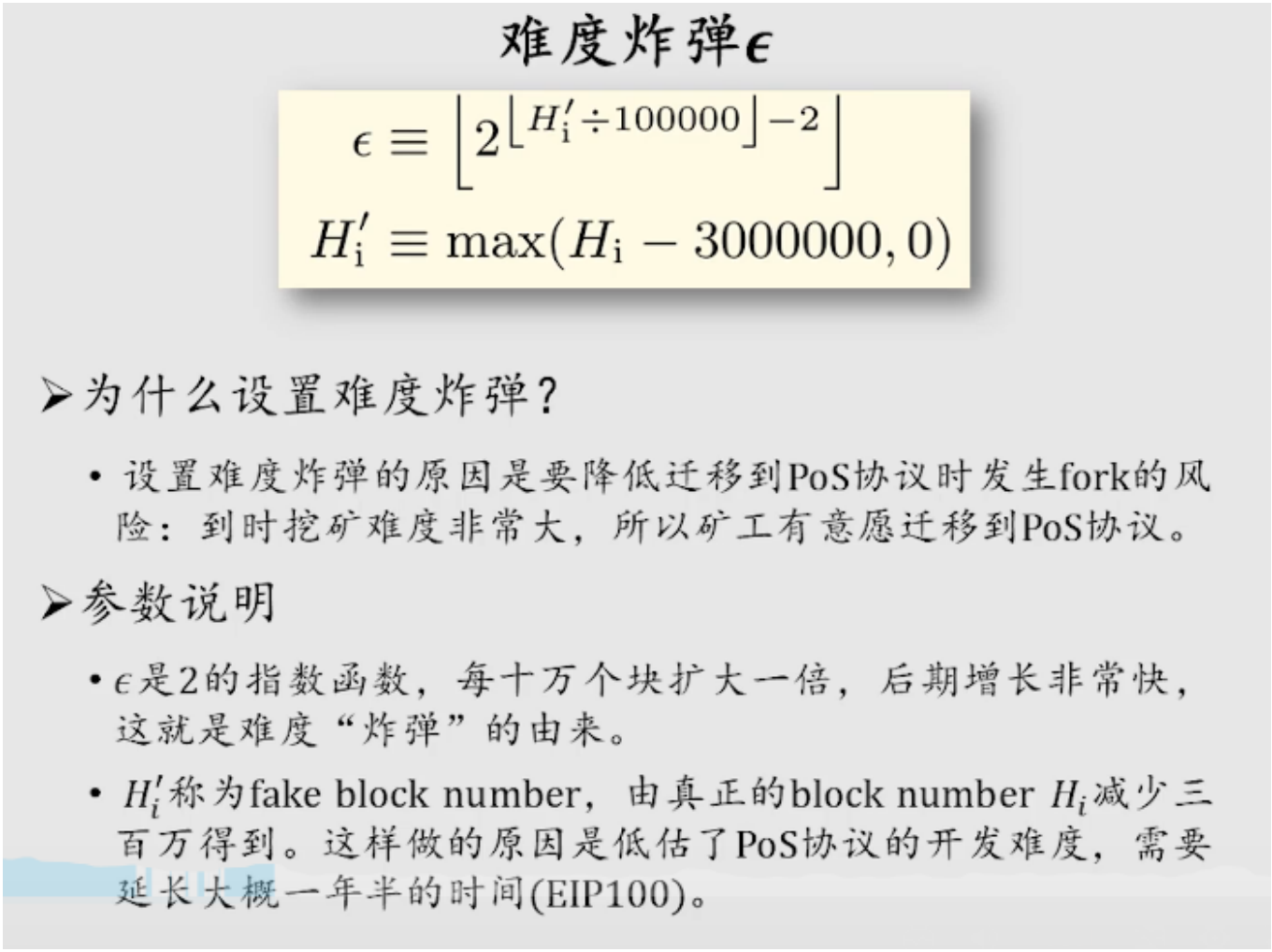
$$x \equiv \left\lfloor \frac{P(H)_{H_d}}{2048} \right\rfloor \tag{43}$$

$$\varsigma_2 \equiv \max\left(y - \left\lfloor \frac{H_s - P(H)_{H_s}}{9} \right\rfloor, -99\right) \tag{44}$$

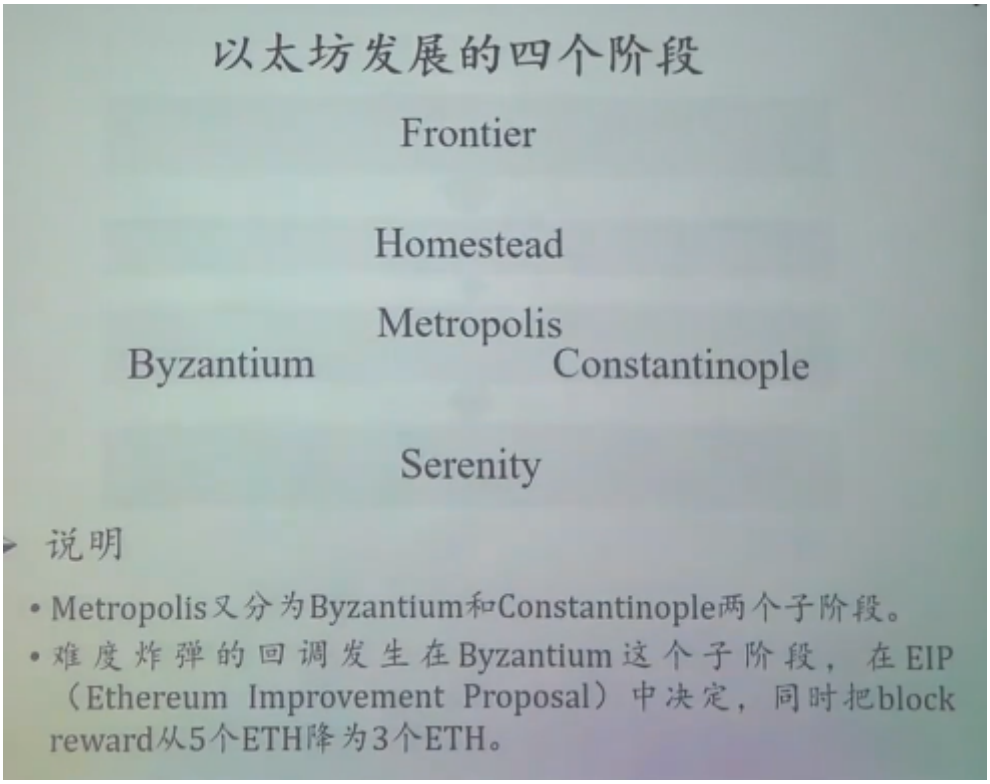
➢ 参数说明

- x是调整的单位，ς_2为调整的系数。
- y和父区块的uncle数有关。如果父区块中包括了uncle ,则y为2，否则y为1。
  - 父块包含uncle时难度会大一个单位，因为包含uncle时新发行的货币量大，需要适当提高难度以保持货币发行量稳定。
- 难度降低的上界设置为−99，主要是应对被黑客攻击或其他目前想不到的黑天鹅事件。


$y - \left\lfloor \frac{H_s - P(H)_{H_s}}{9} \right\rfloor$
参数说明
• H_s是本区块的时间戳，$P(H)_{H_s}$是父区块的时间戳，均以秒为单位，并规定$H_s > P(H)_{H_s}$。
• 该部分是稳定出块速度的最重要部分：出块时间过短则调大难度，出块时间过长则调小难度。
以父块不带uncle($y = 1$)示例
• 出块时间在[1,8]之间，出块时间过短，难度调大一个单位。
• 出块时间在[9,17]之间，出块时间可以接受，难度保持不变。
• 出块时间在[18,26]之间，出块时间过长，难度调小一个单位。


难度炸弹ϵ
$\epsilon \equiv \left\lfloor 2^{\left\lfloor H_i' \div 100000 \right\rfloor - 2} \right\rfloor$
$H_i' \equiv \max(H_i - 3000000, 0)$
➢为什么设置难度炸弹？
• 设置难度炸弹的原因是要降低迁移到PoS协议时发生fork的风险：到时挖矿难度非常大，所以矿工有意愿迁移到PoS协议。
➢参数说明
• ϵ是2的指数函数，每十万个块扩大一倍，后期增长非常快，这就是难度“炸弹”的由来。
• H_i'称为fake block number，由真正的block number H_i减少三百万得到。这样做的原因是低估了PoS协议的开发难度，需要延长大概一年半的时间(EIP100)。

以太坊发展阶段



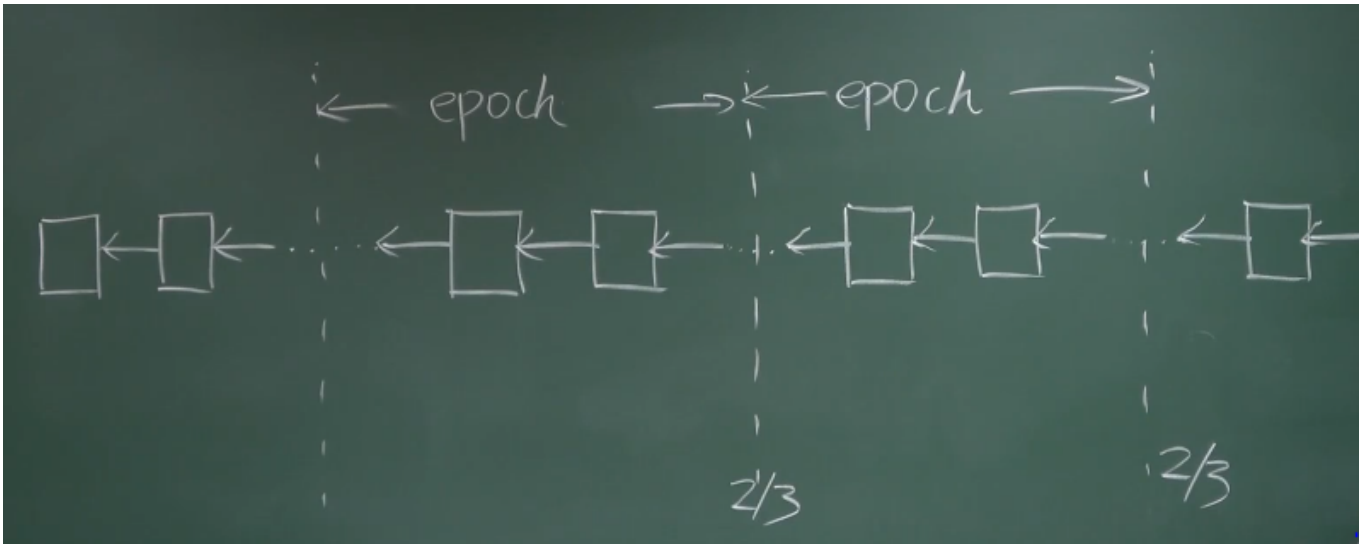
以太坊中拟使用的POS叫casper the friendly finality Gadget（FFG）

目前casper是POS与POW混用，引入验证者概念validator（投入一定的ETH到系统中锁定），V投票决定哪条链是合法链。投票权重取决于保证金的多少。

每50个区块是一个epoch，casper规定每轮epoch由validator投票。在two-phase commit中，本轮epoch作为前一轮的commit message，和后一轮的prepare message。只有两轮pooch都有2/3及以上的validator投票通过才算有效。

关于两阶段提议（two-phase commit）

可以保证数据的强一致性，许多分布式关系型数据管理系统采用此协议来完成分布式事务。它是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法。同时也是解决一致性问题的一致性算法。



验证者在不作为，不投票，耽误了投票进度的情况下会被扣除一定量保障金，如果恶意投票，会扣除所有保障金。验证者在工作一段时间后会有一段等待期，不再作为验证节点，并接受其他节点的审判。

EOS：完全使用了权益证明。DPOS协议。