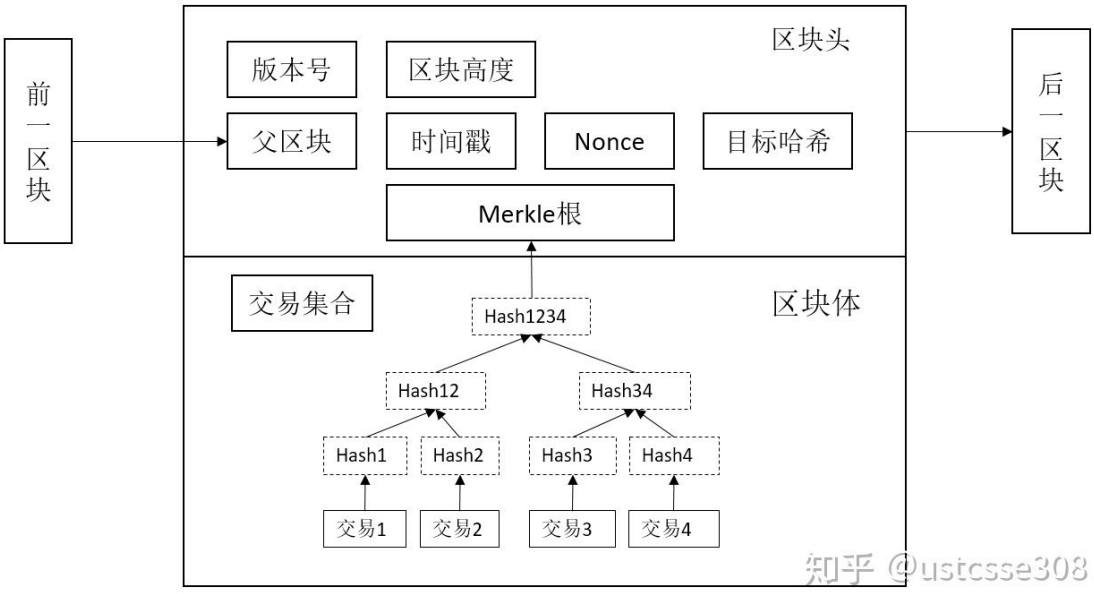
比特币的机制(三)

区块是一种被包含在公开账簿（区块链）里的聚合了交易信息的容器数据结构。它由一个包含元数据的区块头和紧跟其后的构成区块主体的一长串交易列表组成。区块头是80字节，而平均每个交易至少是250字节，而且平均每个区块至少包含超过500个交易。具体看一下，区块的结构：

Size	Field	Description
4 bytes	Block Size	The size of the block, in bytes, following this field
80 bytes	Block Header	Several fields form the block header
1–9 bytes (VarInt)	Transaction Counter	How many transactions follow
Variable	Transactions	The transactions recorded in this block

- 区块链中的两种Hash的数据结构
  1. 区块的哈希链，通过哈希指针（hash pointer）形成的链，在上面的链接中，通过点击哈希值，页面可以跳转到之前或者之后的区块
  2. 区块内的每个块内的交易组成的树状结构。如下所示。



上图中，图的上半部分是区块头；下半部分则是对区块头部中的Merkle根的计算过程的展开。每个区块中的交易形成了一棵Merkle树，Merkle树的根包括在区块头中。区块通过保存前一个块的哈希值形成了一条链。区块包括头部和具体的交易，具体来说，区块头部包括如下信息：

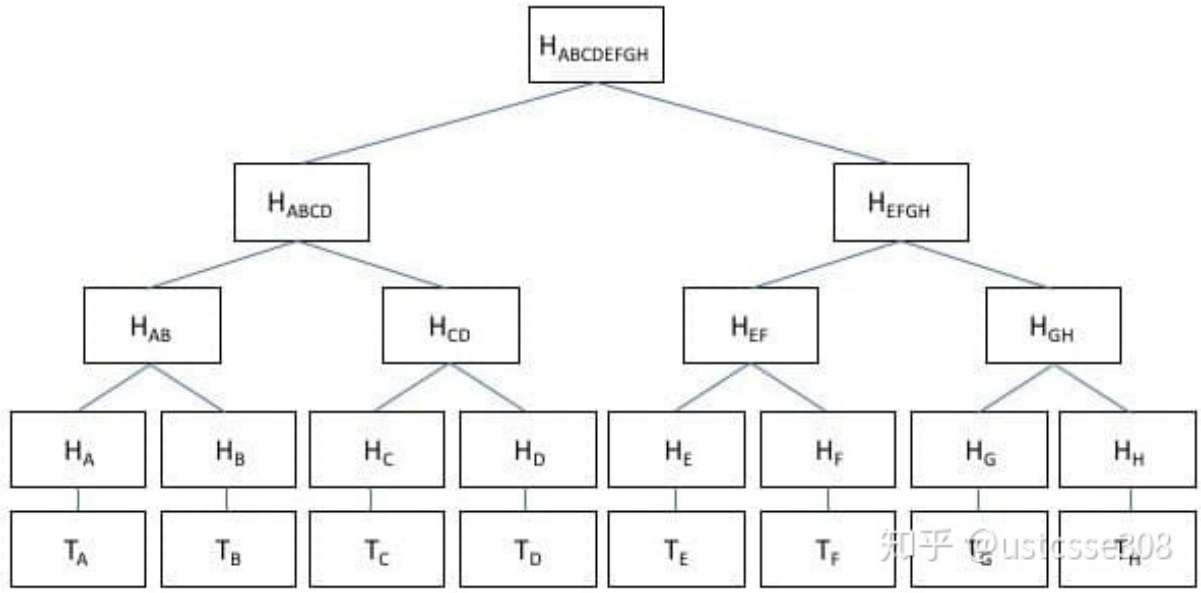
Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Difficulty Target	The Proof-of-Work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

(父区块发生变化则他的后代都会发生变化)由于区块头里面包含"父区块哈希值"字段，所以当前区块的哈希值也受到该字段的影响。如果父区块的身份标识发生变化，子区块的身份标识也会跟着变化。当父区块有任何改动时，父区块的哈希值也发生变化。这将迫使子区块的"父区块哈希值"字段发生改变，从而又将导致子区块的哈希值发生改变。而子区块的哈希值发生改变又将迫使孙区块的"父区块哈希值"字段发生改变，又因此改变了孙区块哈希值，以此类推。一旦一个区块有很多代以后，这种瀑布效应将保证该区块不会被改变，除非强制重新计算该区块所有后续的区块。**正是这样的重新计算需要耗费巨大的计算量，所以一个长区块链的存在可以让区块链的历史不可改变**，这也是比特币安全性的一个关键特征。

区块主标识符是它的加密哈希值，一个通过SHA256算法对区块头进行二次哈希计算而得到的数字指纹。产生的32字节哈希值被称为区块哈希值，但是更准确的名称是：**区块头哈希值(只有区块头参与了计算，因为有Merkle树的存在，因此就算只计算区块头的hash值，区块中的交易也不可篡改)**，因为只有区块头被用于计算。例 如:
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f是第一个比特币区块的区块哈希值。区块哈希值可以唯一、明确地标识一个区块，并且任何节点通过简单地对区块头进行哈希计算都可以独立地获取该区块哈希值。

- Merkle tree

  区块中的所有交易都使用二进制raw transaction的格式保存在区块中，然后对raw transaction进行哈希得到交易id（txid)。merkle树就是使用这样的txid进行构造的。



上图为Merkle tree的示意图

其每个非叶节点通过其子节点的标记或者值（子节点为叶节点）的哈希值来进行标注。例如：HAB的值是通过HA和HB来生成的。

(知道叶节点的个数求树高)可以作为考题

由于哈希的单向性，可以得出结论，如果两棵Merkle树的merkle root相同，那么这两棵树的结构和每个节点也必然是相同的。另外，只要存储的叶子节点数据有任何的变动，就会逐级向上传递到相应的父节点，最终使得Merkle树的根节点哈希值发生变化。(**回答上面问题为什么只需要对区块的头部进行hash即可保证整个交易不会被篡改**)

- 为什么使用Merkle Tree

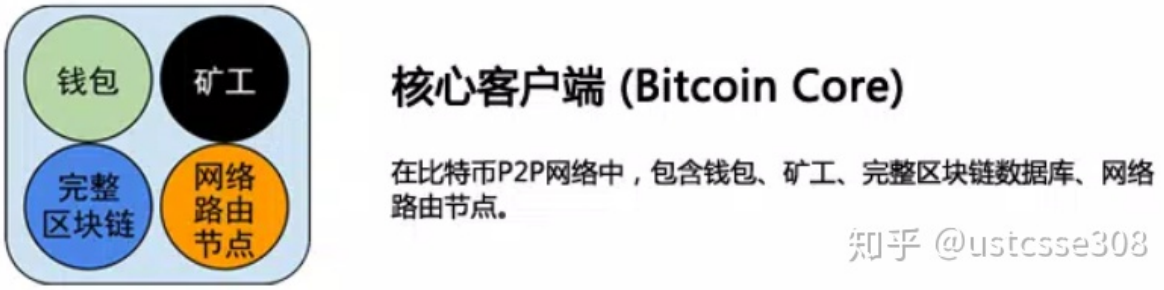
为什么要使用这样的数据结构呢？树这种数据结构我们应该是比较熟悉的，特别是这种满二叉树。那我们来算一下，如果如上图所示，有16个叶子节点，也即一个区块中有16个交易，那么总共需要多少次哈希计算？如果是为了保证区块中的交易没有被篡改，那么实际上只需要把16个交易连接起来，然后对整个内容做一次哈希就够了，为什么要这么麻烦形成一棵树，而且做这么多次的计算呢？

一个例子：

> 如果A和B两个服务器上存储的文件系统都是一致的，也即两个的哈希值是一样的，那么自然是很好的。如果两个不一致呢？譬如说A服务器上有一个文件更新了，而B服务器还没有来得及更新。怎么样能够快速地定位到导致两个文件系统不一致的文件？
>
> 这时就能体现树结构的好处了。如果两个哈希值不一致，A服务器就可以向B服务器要两个子节点的哈希值；然后沿着不一样的路径一直走下去，从而可以确定导致根哈希值不同的文件。

在比特币中，Merkle树的作用：

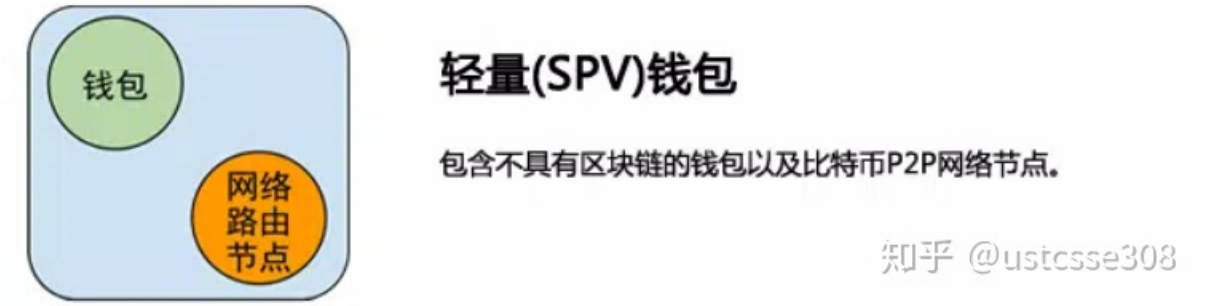
默认情况下，一旦接受到一个新交易，节点需要验证它，特别是，验证交易的输入中的每一个之前是否被花费。为了完成这个验证，需要访问区块链。如果节点不信任网络上的其他节点，那么 这个节点需要保存网络上的所有区块，以便验证交易。这种节点称作全节点。在比特币发展的早期，所有节点都是全节点；当前的比特币核心客户端也是完整区块链节点。



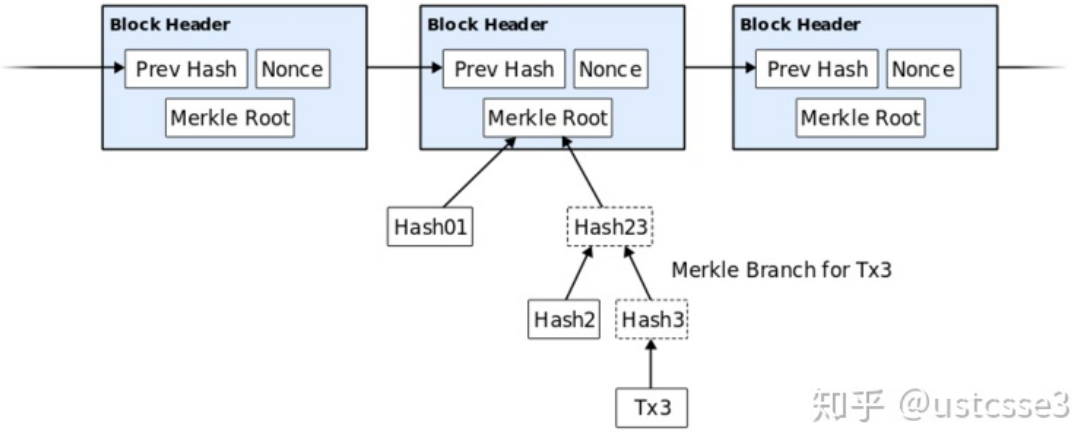
在当前的比特币网络中，实际参与共识的全验证节点（fully validating nodes）并不多，因为全验证节点会维护整个区块链的数据，由于区块链的不可篡改和append-only，随着时间的增加，整个区块链的数据量非常大。在2014年4月份，比特币网络中存储所有区块的数据，需要15GB的空间，现在要完整下载比特币的所有区块数据，需要200GB以上的空间。

全节点要检查第300,000号区块中的某个交易，它会把从该区块开始一直回溯到创世区块的300,000个区块全部都链接起来，建立一个完整的UTXO数据库，通过确认该UTXO是否还未被支付来证实交易的有效性。全验证节点维护所有的UTXO， 最好是存在RAM中，这样，当网络中有新的交易广播时，全验证节点可以快速地进行查询、运行脚本、确定交易是否正确、签名是否有效，如果全部正确，则将交易添加到交易池中。

全验证节点对硬件提出了很高要求，个人用户（移动设备）参与这个过程几乎是不可能的。为了客户友好，对于仅仅使用钱包的普通用户，也即轻量级的节点，比特币网络中的大部分用户都是轻量级的用户，比特币网络并不要求它们也存储所有的信息。这种节点只需要维护能够验证用户自己所care的交易的部分信息就行了。这也是中本聪在比特币白皮书中所提出的SPV（simple payment verification）的概念。SPV可以不需整个网络的数据而确认交易是否存在。



那问题是，SPV是怎么实现的呢？为什么仅仅需要有限的信息就可以进行验证?



验证原理：

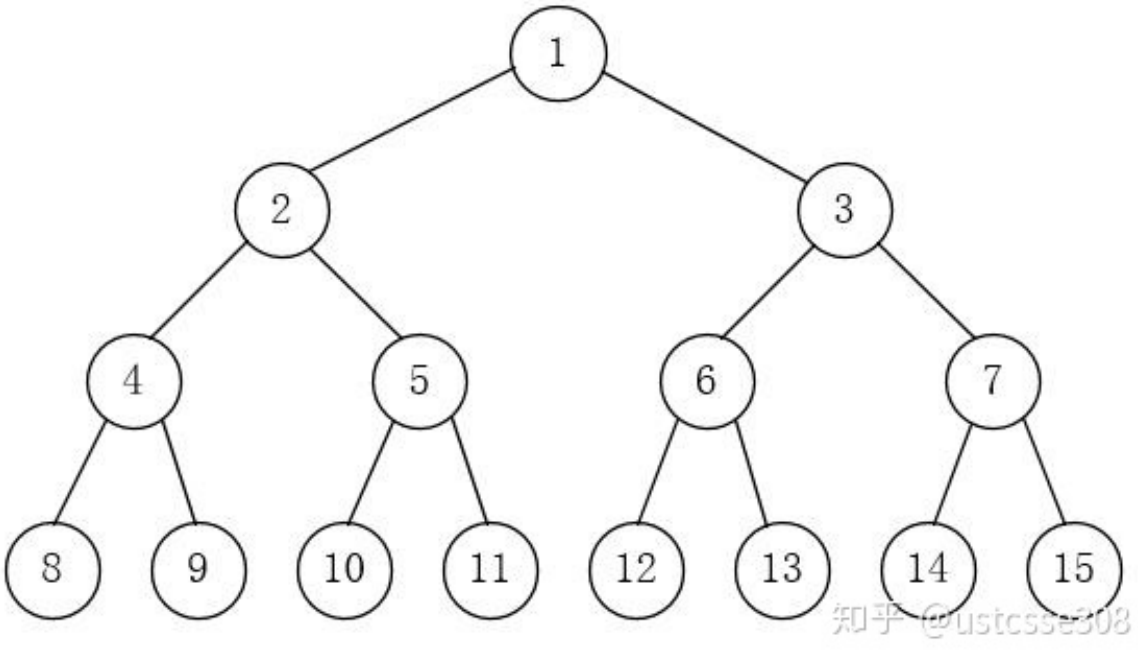
如上图，如果需要证明某个区块上是否存在一笔交易Tx3，那么全节点返回的Merkle路径是Hash2和Hash01。只需要这两个值就可以进行验证过程如下：

- Step1：计算交易Tx3的哈希值，得到Hash3
- Step2：通过Hash2和Hash3的哈希值，得到父节点的哈希值Hash23
- Step3：同上，通过计算Hash23和Hash01哈希值，得到根节点的哈希值。
- Step4：将上一步得到的根哈希值对比区块头中MerkleTree的根哈希值，如果相同，则证明该区块中存在交易Tx3，否则说明不存在。

使用Merkle树可以大大降低SPV节点的存储和计算负担；下面的表格对比了区块中不同交易数量的情况下，完整区块大小和Merkle路径大小的情况。

Number of transactions	Approx. size of block	Path size (hashes)	Path size (bytes)
16 transactions	4 kilobytes	4 hashes	128 bytes
512 transactions	128 kilobytes	9 hashes	288 bytes
2048 transactions	512 kilobytes	11 hashes	352 bytes
65,535 transactions	16 megabytes	16 hashes	512 bytes

要会求Merkle路径



例如求11的merkle路径：10, 4, 3

- SPV节点的安全性

  1）若全节点返回的是一条恶意的路径？试图为一个不存在于区块链中的节点伪造一条合法的merkle路径，使得最终的计算结果与区块头中的默克尔根哈希相同。

  由于哈希的计算具有不可预测性，使得一个恶意的“全”节点想要为一条不存在的节点伪造一条“伪路径”使得最终计算的根哈希与轻节点所维护的根哈希相同是不可能的。

  （2）为什么不直接向全节点请求该节点是否存在于区块链中?

  由于在公链的环境中，无法判断请求的全节点是否为恶意节点，因此直接向某一个或者多个全节点请求得到的结果是无法得到保证的。但是轻节点本地维护的区块头信息，是经过工作量证明验证的，也就是经过共识一定正确的，若利用全节点提供的Merkle路径，与待验证的节点进行哈希计算，若最终结果与本地维护的区块头中根哈希一致，则能够证明该节点一定存在于默克尔树中。

  （3）SPV容易受到什么攻击?

  SPV节点毫无疑问可以证实某个交易的存在性，它也能够证明某个区块中不存在某个交易，但它不能验证某个交易（譬如同一个UTXO的双重支付）在整个链中不存在，这是因为SPV节点没有一份关于所有交易的记录。这个漏洞会被针对SPV节点的拒绝服务攻击或双重支付型攻击所利用。为了防御这些攻击，SPV节点需要随机连接到多个节点，以增加与至少一个可靠节点相连接的概率。这种随机连接的需求意味着SPV节点也容易受到网络分区攻击或Sybil攻击。在后者情况中，SPV节点被

连接到虚假节点或虚假网络中，没有通向可靠节点或真正的比特币网络的连接。

在绝大多数的实际情况中，具有良好连接的SPV节点是足够安全的，它在资源需求、实用性和安全性之间维持恰当的平衡。当然，如果要保证万无一失的安全性，最可靠的方法还是运行完整区块链的节点。

> 完整的区块链节点是通过检查整个链中在它之下的数千个区块来保证这个UTXO没有被支付，从而验证交易。而SPV节点是通过检查在包含该交易的区块所收到的确认数目来验证交易。

- Bloom Filter

在之前的例子中，我们并没有涉及一些细节。譬如，SPV节点直接就向全节点请求某一交易的Merkle路径。SPV节点怎么样从网络中接收到与自己相关的交易，确定交易所在的区块呢?

SPV节点一般只需要的是和自己的地址相关的交易。在BIP37之前，SPV的做法是将所有的区块和交易都下载下来，然后本地将不相关的交易给删掉。当然带来的问题就是同步慢、浪费带宽、增加内存使用。在BIP-37中就提到了因为这一点，导致用户对手机APP“Bitcoin Wallet”有所抱怨。

为了解决上述问题：

最直接的做法就是SPV节点仅向全节点请求和自己地址相关的交易，也即请全节点过滤和自己地址不相关的信息，如果全节点发现某个交易符合SPV节点的需求时，就将以Merkleblock消息的形式发送该交易，Merkleblock消息包含区块头和Merkle路径。此时，SPV就需要在请求中附上自己的地址信息。

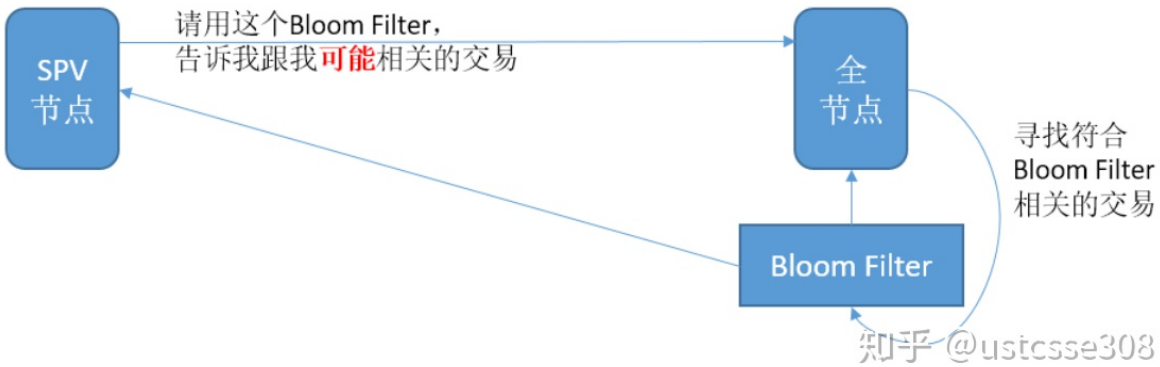
风险：

因此与全区块链节点收集每一个区块内的全部交易所不同，SPV节点对特定数据的请求可能无意中透露了钱包里的地址信息。如果监控网络的第三方跟踪某个SPV节点上的钱包所请求的全部交易信息，就能利用这些交易信息把比特币地址和钱包的用户关联起来。

> 举例来说，如果在问路时，使用具体的地址，如“南京路188号”，那么可能得到具体的位置；但同时也泄露了目的地。如果问不同的人，188号在哪里？可能得到所有188号的信息；然后问南京路在哪里？可以得到一整条路的信息。那么虽然获得的答案中包括一些无关的信息；但是，相对应的，隐私得到了一定程度的保护。

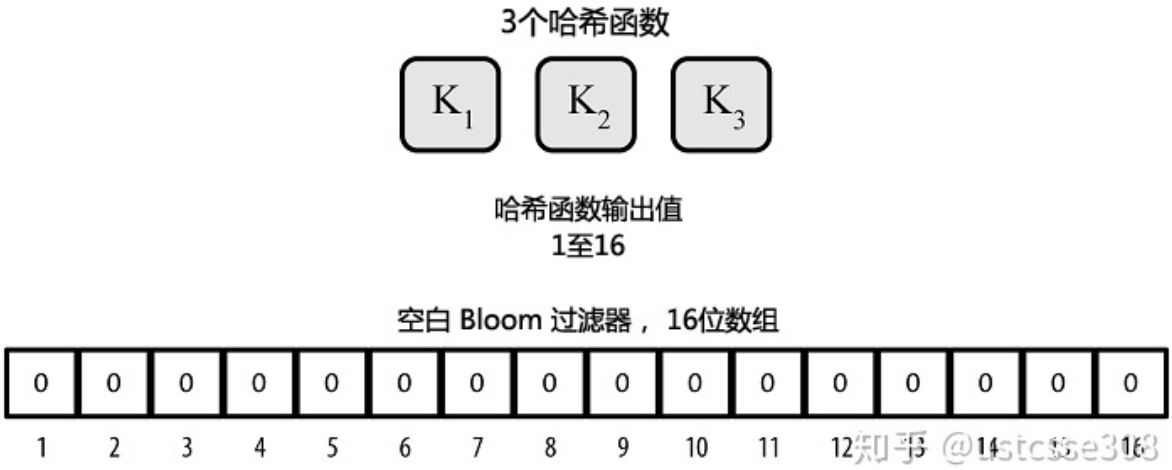
bloom filter出现

因此，在引入SPV节点/轻量级节点后不久，比特币开发人员就添加了一个新功能：Bloom过滤器。这是在2012年的BIP37中引入的。bitcoin/bips 在比特币中，使用Bloom过滤器来加快钱包同步；以太坊使用Bloom过滤器用于快速查询以太坊区块链的日志。



- Bloom filter的工作流程

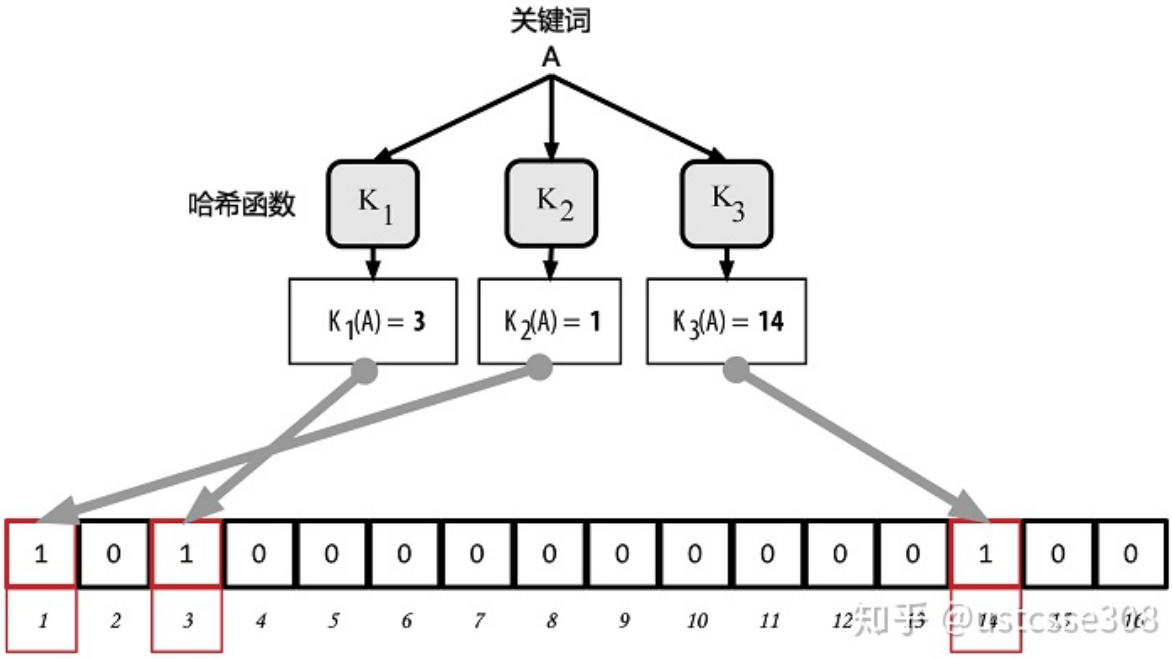
这里使用十六位数组（N=16）和三个哈希函数（M=3）来演示Bloom过滤器的应用原理。



由16位数组和3个哈希函数组成的简易Bloom Filter

Bloom过滤器数组里的每一个数的初始值为零。关键词被加到Bloom过滤器中之前，会依次通过每一个哈希函数运算一次。该输入经第一个哈希函数运算后得到了一个在1和N之间的数，它在该数组（编号依次为1至N）中所对应的位被置为1，从而把哈希函数的输出记录下来。接着再进行下一个哈希函数的运算，把另外一位置为1；以此类推。当全部M个哈希函数都运算过之后，一共有M个位的值从0变成了1，这个关键词也被“记录”在了Bloom过滤器里。

向上图中的简易Bloom过滤器添加关键词“A”。

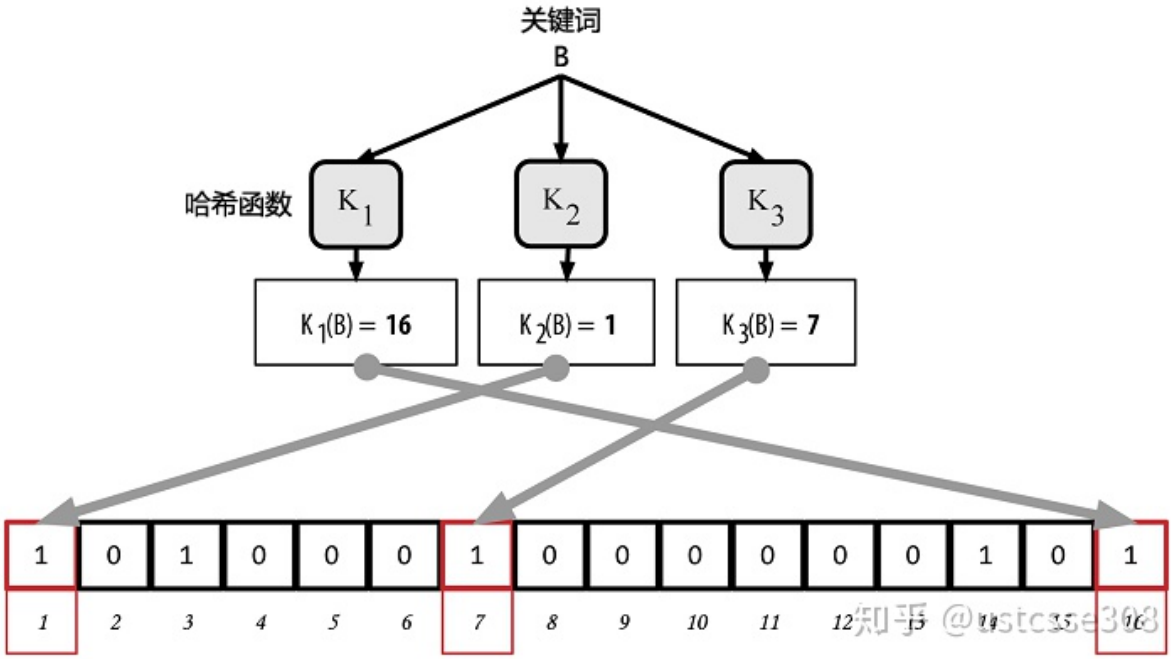


向简易Bloom过滤器中增加关键词“A”

增加第二个关键词就是简单地重复之前的步骤。关键词依次通过各个哈希函数运算之后，相应的位变为1，Bloom过滤器则记录下该关键词。需要注意的是，当Bloom过滤器里的关键词增加时，它对应的某个哈希函数的输出值的位可能已经是1了，这种情况下，该位不会再次改变。也就是说，随着更多的关键词指向了重复的位，Bloom过滤器随着位1的增加而饱和，准确性也因此降低了。该过滤器之所以是基于概率的数据结构，就是因为关键词的增加会导致准确性的降低。准确性取决于关键字的数量以及数组大小（N）和哈希函数的多少（M）。更大的数组和更多的哈希函数会记录更多的关键词以提高准确性。而小的数组及有限的哈希函数只能记录有限的关键词从而降低准确

性。

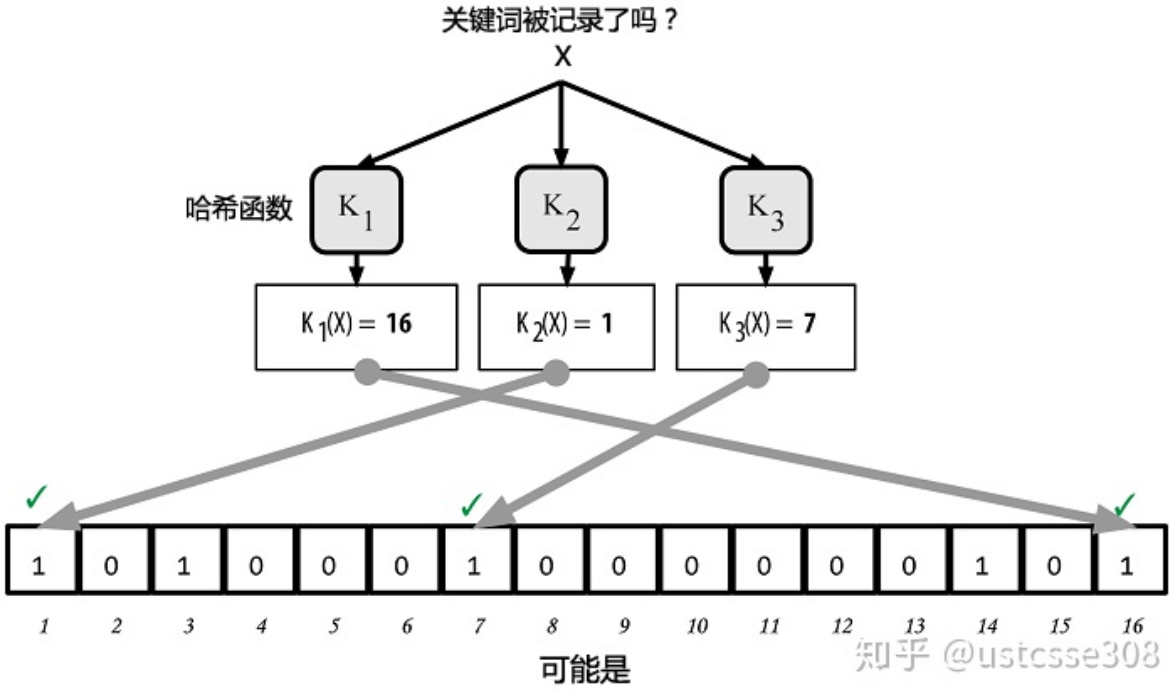
问题：如果N比较小，当添加关键词时，所有的位都成为1，意味着什么？



向Bloom Filter增加关键词

为测试某一关键词是否被记录在某个Bloom过滤器中，则将该关键词逐一代入各哈希函数中运算，并将所得的结果与原数组进行对比。如果所有的结果对应的位都变为了1，则表示这个关键词有可能已被该过滤器记录。之所以这一结论并不确定，是因为这些字节1也有可能是其他关键词运算的重叠结果。简单来说，Bloom过滤器正匹配代表着“可能是”。

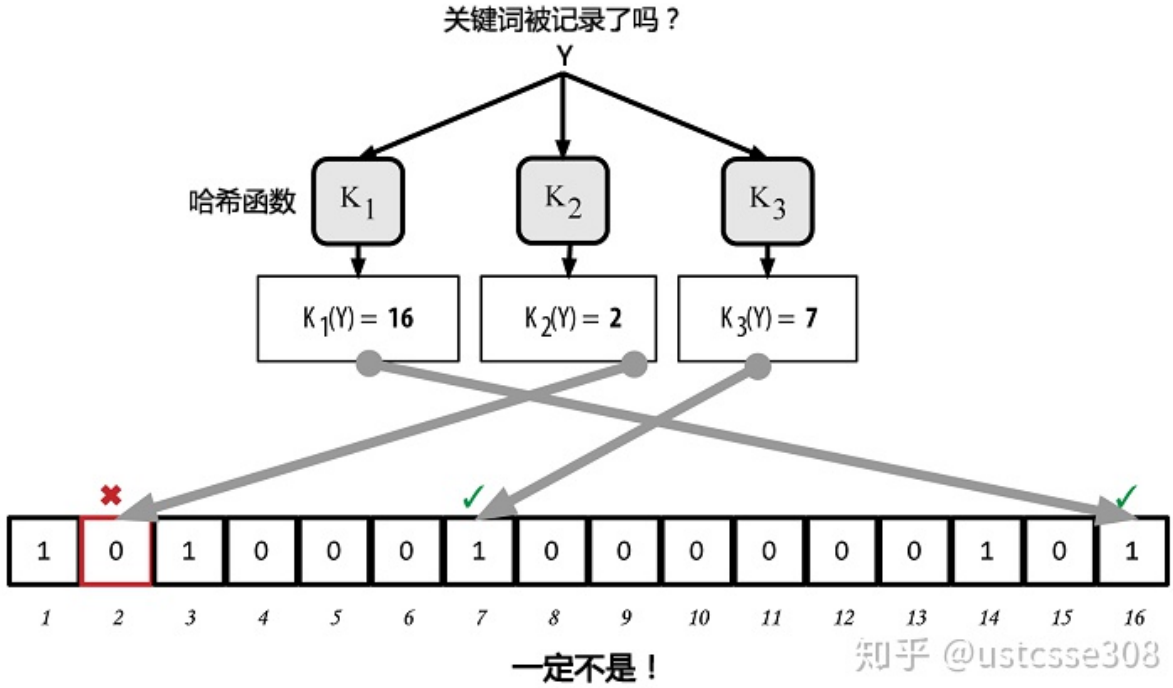
下图是一个验证关键词“X”是否在前述Bloom过滤器中的例子。相应的比特位都被置为1，所以这个关键词很有可能是匹配的。



测试关键词“X“是否能通过Bloom Filter

另一方面，如果我们代入关键词计算后的结果某位为0，说明该关键词并没有被记录在过滤器里。负匹配的结果不是可能，而是一定。也就是说，负匹配代表着“一定不是”。

下图验证关键词“Y”是否存在于简易Bloom过滤器中的图例。图中某个结果字段为0，该字段一定没有被匹配。



- 在SPV节点中使用bloom filter
  - 首先，SPV节点会初始化一个不会匹配任何关键词的“空白”Bloom过滤器
  - 接下来，SPV节点会创建一个包含钱包中所有地址信息的列表，并创建一个与每个地址相对应的交易输出相匹配的搜索模式。通常，这种搜索模式是一个向公钥付款的哈希脚本，该脚本是一个会出现在每一个向公钥哈希地址（P2PKH）付款的交易中的锁定脚本。如果SPV节点需要

追踪P2SH地址余额，搜索模式就会变成P2SH脚本。
    - 然后，SPV节点会把每一个搜索模式添加至Bloom过滤器里，这样只要关键词出现在交易中就能够被过滤器识别出来
    - 最后，对等节点会用收到的Bloom过滤器来匹配传送至SPV节点的交易
- bloom filter为什么是false positive而不是false negative

  假如一个数据经过bloom filter之后得到的结果是1，在hash表中对应的数据如果也是1，那么则返回true，但是hash表中的1可能是其他数据写入的，所以是false positive。

  假如经过bloom filter之后得到的结果为0，则直接返回false，因为无论如何这个数据都不可能在hash表中

- BIP37中确定交易是否匹配filter，使用如下算法

  1. 测试交易本身的哈希。
  2. 对于每个输出，测试输出脚本中的每一个数据项。每一个哈希（密钥）都单独测试。如果在测试交易的时候发现了匹配的输出，那么节点也可以升级filter，将该输出的COutPoint结构也添加到filter中。也即，将该交易的输出中与SPV用户自己相关的部分（可用于其他交易的输入）添加到filter中。
  3. 对于每一个输入，测试COutPoint结构。
  4. 对于每一个输入，测试输入脚本ScriptSig的每一个数据项。
  5. 否则，没有匹配。

  分析一下：

  1. 步骤1是因为用户有可能对某一个特定的交易感兴趣；
  2. 步骤2是因为用户可能在过滤器中加入了自己的公钥或地址；如果某一笔交易发钱给自己，那么用该COutpoint来更新过滤器；【检查自己的收入】
  3. 步骤3检查自己的花费，CoutPoint是如下所示：

  4. 步骤4检查自己的花费，因为ScriptSig中包含自己的公钥；

  问题：能不能通过删除项来更新bloom filter?(考试可能会考)