Research report on MPT

宋诺金 202000460074

在以太坊中，一种经过改良的默克尔树非常关键，是以太坊数据安全与效率的保障，此树在以太坊中称之为 MPT（默克尔压缩前缀树）。 MPT 全称是 Merkle Patricia Trie 也叫 Merkle Patricia Tree，是 Merkle Tree 和Patricia Tree 的混合物。 Merkle Tree(默克尔树) 用于保证数据安全，Patricia Tree(基数树,也叫基数特里树或压缩前缀树) 用于提升树的读写效率。

简述

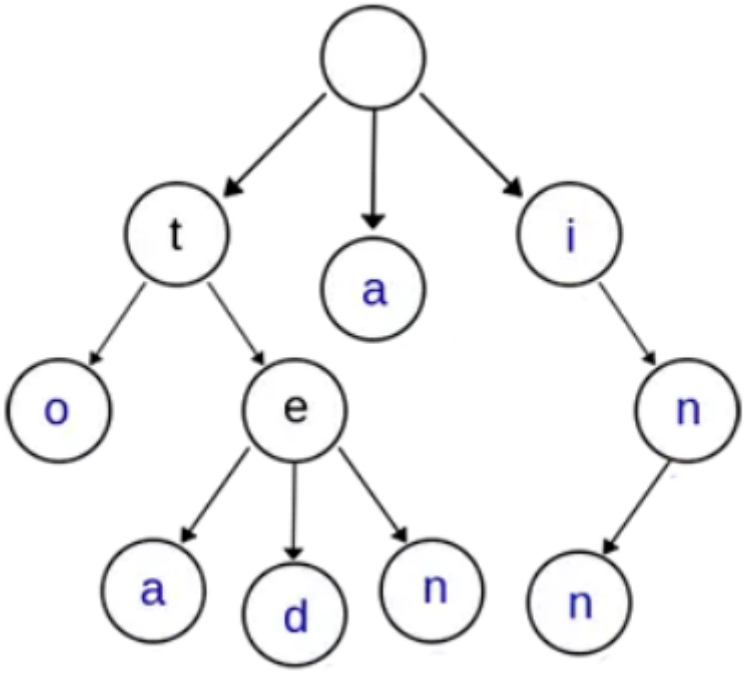
以太坊不同于比特币的 UXTO 模型，在账户模型中，账户存在多个属性（余额、代码、存储信息），属性（状态）需要经常更新。因此需要一种数据结构来满足几点要求：

- ①在执行插入、修改或者删除操作后能快速计算新的树根，而无需重新计算整个树。
- ②即使攻击者故意构造非常深的树，它的深度也是有限的。否则，攻击者可以通过特意构建足够深的树使得每次树更新变得极慢，从而执行拒绝服务攻击。
- ③树的根值仅取决于数据，而不取决于更新的顺序。以不同的顺序更新，甚至是从头重新计算树都不会改变树的根值。

要求①是默克尔树特性，但要求②③则非默克尔树的优势。 对于要求②，可将数据 Key 进行一次哈希计算，得到确定长度的哈希值参与树的构建。而要求③则是引入位置确定的压缩前缀树并加以改进。

Trie 字典树

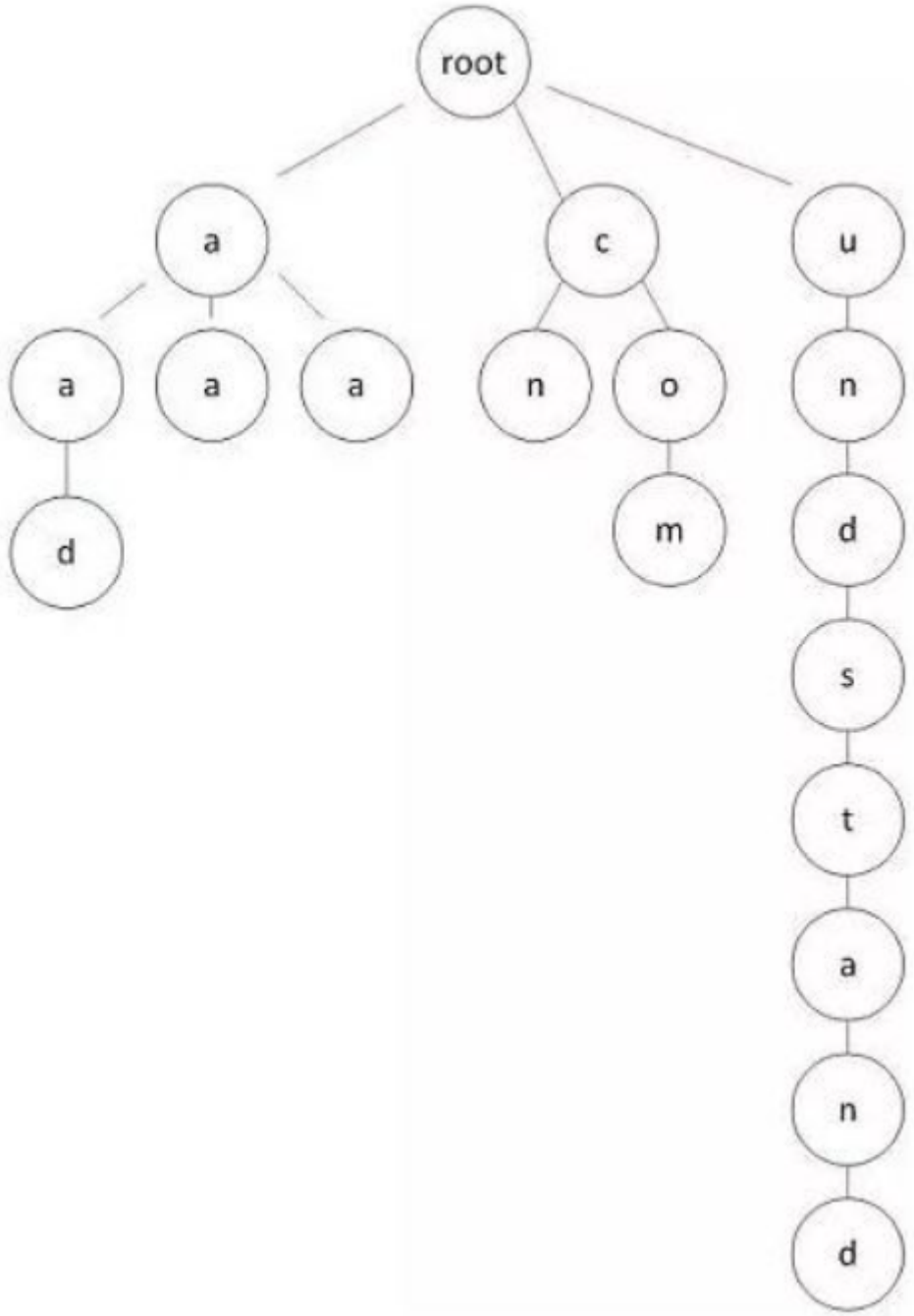
Trie 树，又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是字符串。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。



上图是一棵 Trie 树，表示了字符串集合{“a”，“to”，“tea”，“ted”，“ten”，“i”，“in”，“inn”}，从上图中我们可以看出Trie树的特点：

- 根节点不包含字符，除根节点外的每一个子节点都包含一个字符。
- 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。
- 每个节点的所有子节点包含的字符互不相同。

但是从上面的结构也可以看出一个问题：高度不可控，如下图所示。所以就有了 Patricia 树 (压缩前缀树)，后面会介绍到。



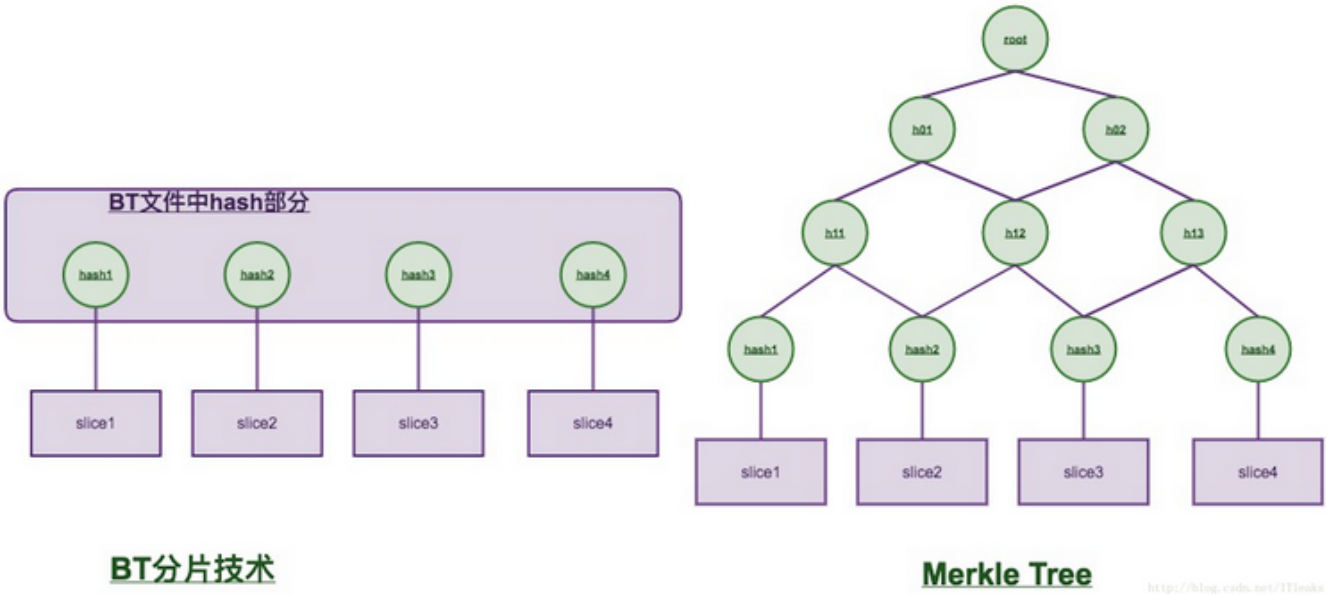
Merkle Tree(默克尔树)

区块链P2P网络中，如果需要传输的数据很大，就需要同时从多个机器上下载数据，而且很可能有些机器是不稳定(可能下载速度很慢)或者不可信的(需要重新下载)。为了快速下载大块数据并验证，更好的办法是把大文件分割成小的数据块（例如把大文件分割成4K为单位的小数据块）。这样的好处是如果小块数据在传输过程中损坏了或者是错误的数据，那么只要重新下载这一块数据就行了，不用重新下载整个文件。

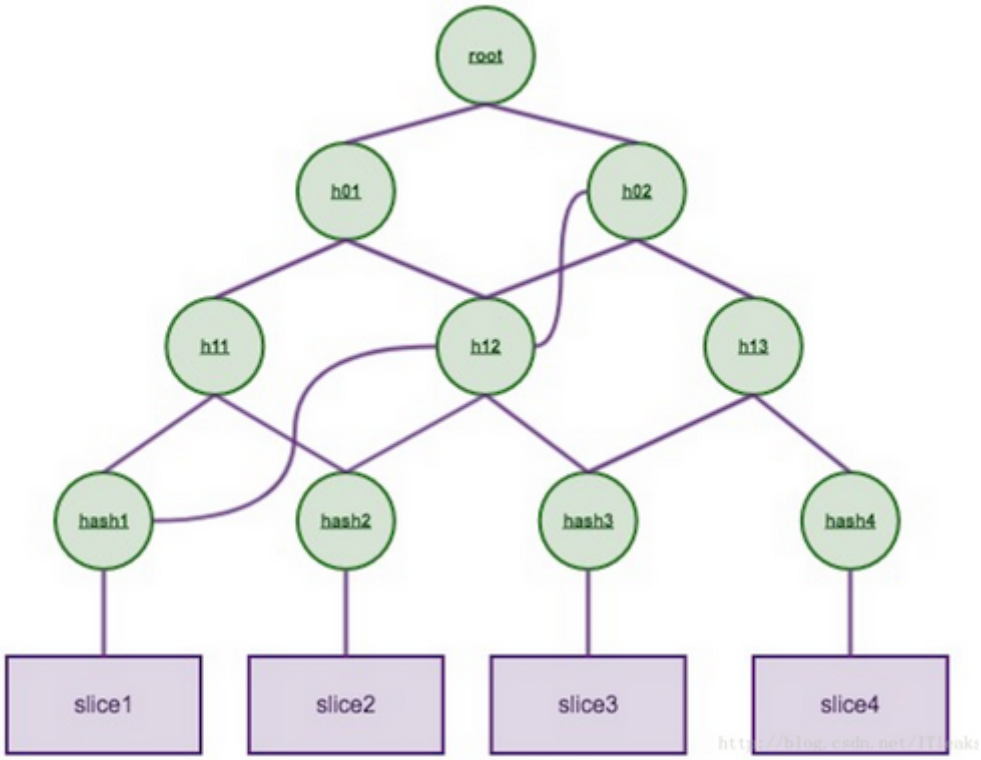
由于只有大文件内容的hash, 当其中一块小数据错误时，我们是能检出由小块数据拼凑出来的大数据是错误的，但是我们不知道哪块小数据是错误的，就没法通过重传错误的小数据来纠正。哪怎么处理呢？为每个小块生成hash, 节点先把小块数据的hash都下载下来，然后就可以验证一个一个验证小块数据是否正确了。那问题又来了，小块数据的hash的正确性谁来保证呢？信任节点，比如BT论坛上的bt种子文件，这个种子文件就记录了原始文件的小块数据的hash. 验证问题解决了，但是多出来了小

块数据hash，当块数据很大时，这个数据量也不小。因而Merkle Tree出来了，它就是用来解决这个问题的。

小块数据的hash两两组合再次生成新的hash,然后新生成的hash又两两合并生成更新的hash,直至最后两个hash生成一个hash root,这个叫merkle root(默克尔根)。可见merkle tree和传统bt分片技术只是对小块数据hash的组织方式不一样。



看到上面的图，你可能会说，merkle tree不是生成更多hash数据了啊，怎么能降低数据传输量。确实，对于数据发送方来说，相比传统分片技术，它是需要保存完整merkle tree, 会多占用一点空间。但是对于接受方来说，它在验证某一块数据不需要下载全部hash,只需一段merkle 路径即可，比如下图

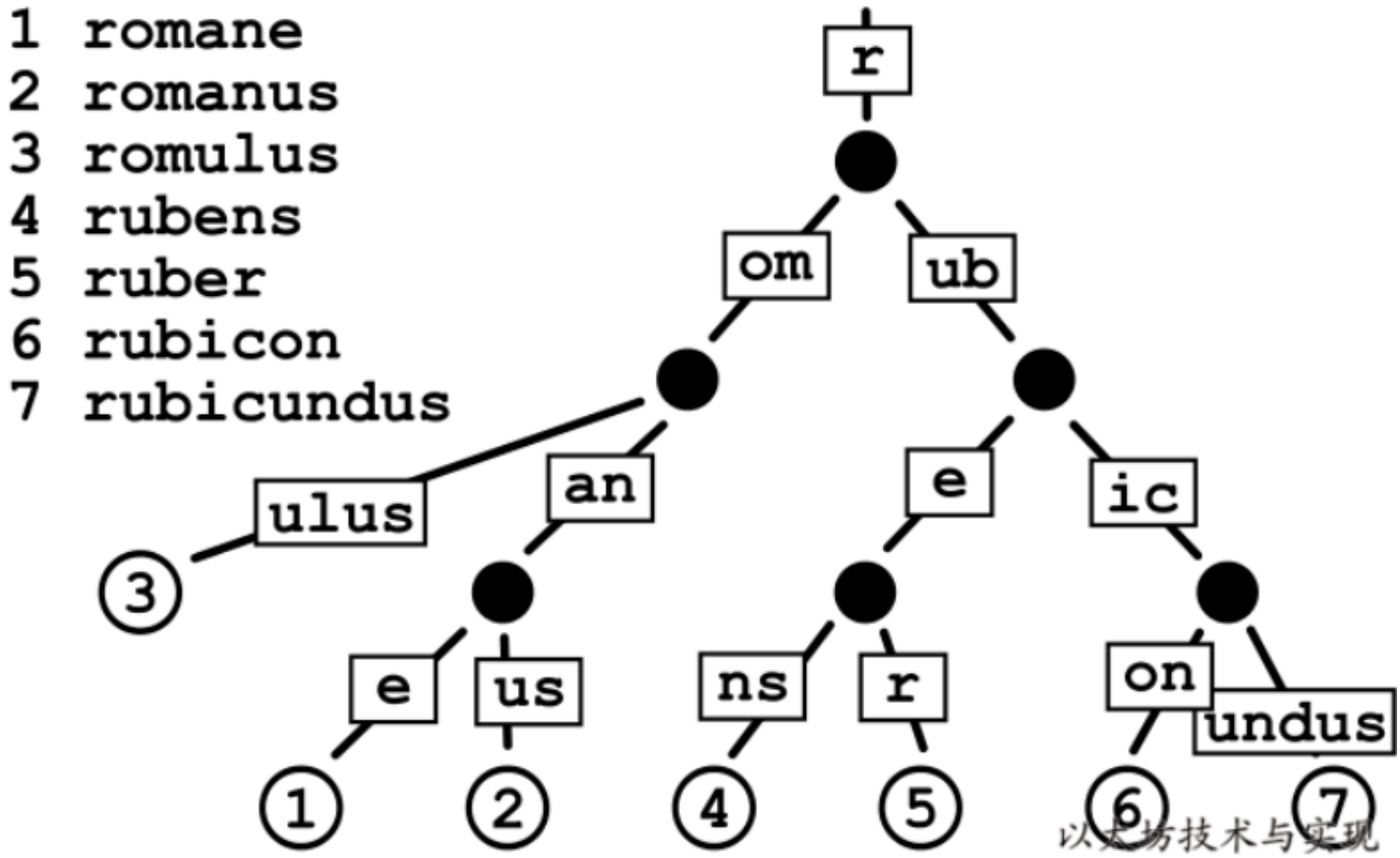


如果要验证slice2数据的正确性，只需要拿到hash1, h12, h02这3个hash再加上本地存储的root hash, 就可以验证了。需要传输的hash数据量从n变为log2n.

总结，Merkle root(包含在区块头里)是从可信渠道下载的(主链或者可信节点)，接收到数据及这个数据对应的Merkle验证路径即可验证该数据的正确性。

压缩前缀树(Patricia Tree)

在压缩前缀树（基数树）中，键值是通过树到达相应值的实际路径值。也就是说，从树的根节点开始，键中的每个字符会告诉您要遵循哪个子节点以获取相应的值，其中值存储在叶节点中，叶节点终止了穿过树的每个路径。假设键是包含N 个字符的字母，则树中的每个节点最多可以有 N 个子级，并且树的最大深度是键的最大长度。

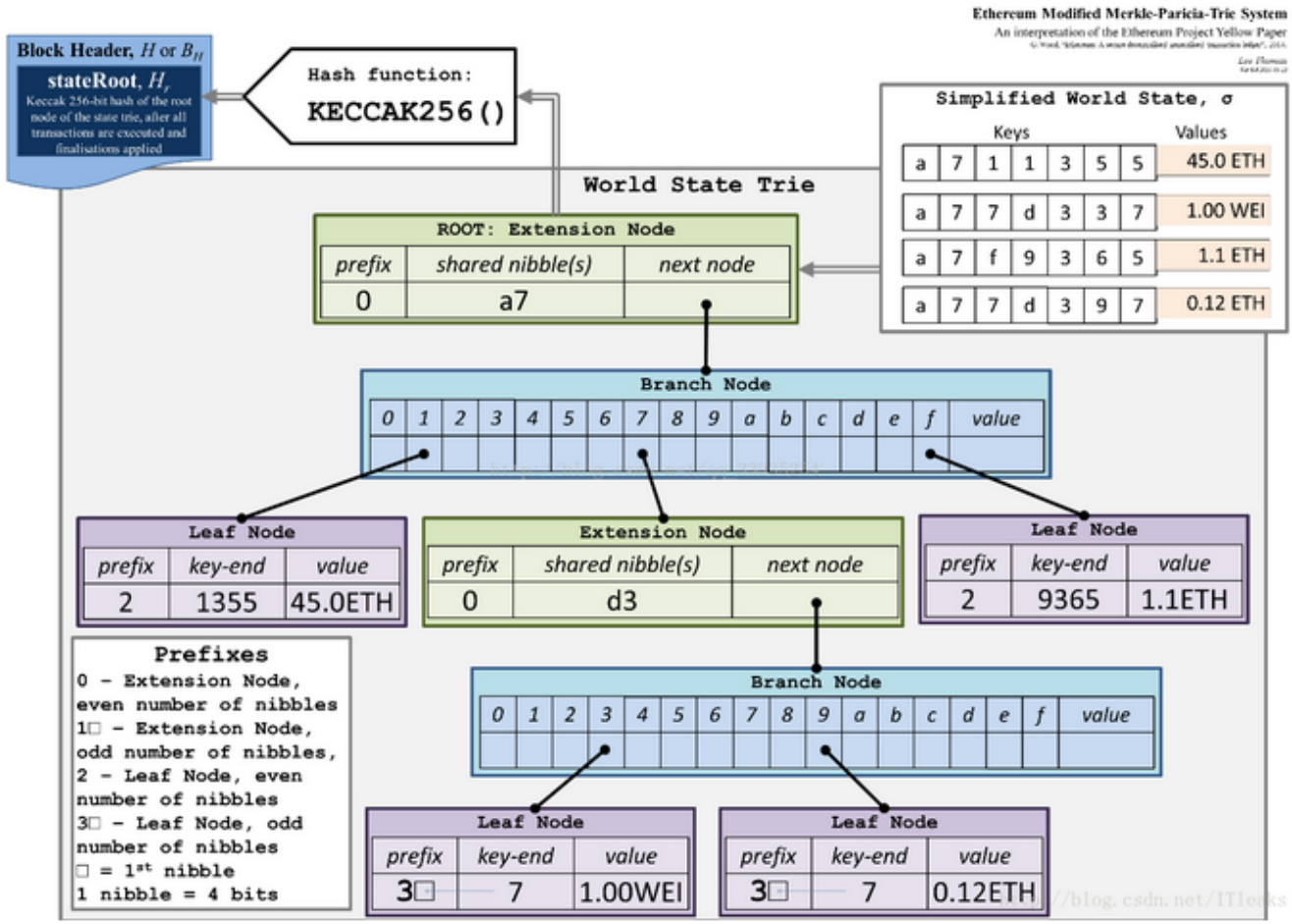


虽然基数树使得以相同字符序列开头的键的值在树中靠得更近，但是它们可能效率很低。例如，当你有一个超长键且没有其他键与之共享前缀时，即使路径上没有其他值，但你必须在树中移动（并存储）大量节点才能获得该值。这种低效在以太坊中会更加明显，因为参与树构建的 Key 是一个哈希值有 64 长（32 字节），则树的最长深度是 64。树中每个节点必须存储 32 字节，一个 Key 就需要至少 2KB 来存储，其中包含大量空白内容。因此，在经常需要更新的以太坊状态树中，优化改进基数树，以提高效率、降低树的深度和减少 IO 次数，是必要的。

MPT (Merkle Patricia Tree)

上面我们介绍了Merkle Tree和Patricia Tree，而MPT（Merkle Patricia Tree），顾名思义就是这两者的结合。MTP树种的节点包含空节点、叶子节点、扩展节点和分支节点。

key有多长，树的深度就会多长，不管这个key有没有和其他key共享部分key。因而允许一个节点表示变长的key就可以解决这个深度，具体以官方的下图为例：



上图存储的key-value如下：

Keys							Values
a	7	1	1	3	5	5	45.0 ETH
a	7	7	d	3	3	7	1.00 WEI
a	7	f	9	3	6	5	1.1 ETH
a	7	7	d	3	9	7	0.12 ETH

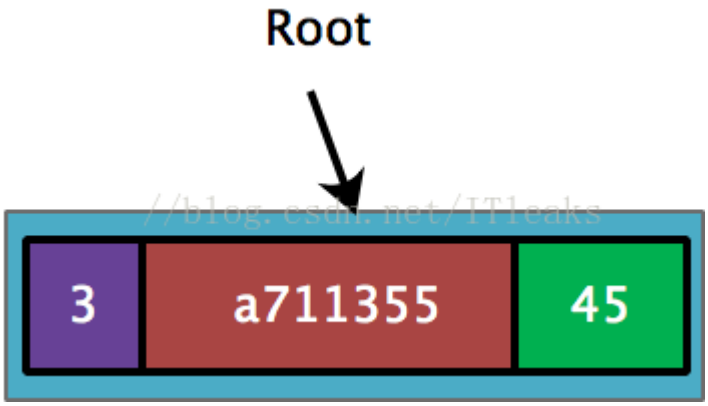
从前面结构图可以看出，Merkle Patricia Tree有4种类型的节点：

- 叶子节点（leaf），表示为[key,value]的一个键值对。和前面的英文字母key不一样，这里的key都是16编码出来的字符串，每个字符只有0-f 16种，value是RLP编码的数据
- 扩展节点（extension），也是[key，value]的一个键值对，但是这里的value是其他节点的hash值，通过hash链接到其他节点
- 分支节点（branch），因为MPT树中的key被编码成一种特殊的16进制的表示，再加上最后的value，所以分支节点是一个长度为17的list，前16个元素对应着key中的16个可能的十六进制字符，如果有一个[key,value]对在这个分支节点终止，最后一个元素代表一个值，即分支节点既可以搜索路径的终止也可以是路径的中间节点。分支节点的父亲必然是extension
- node空节点，代码中用null表示

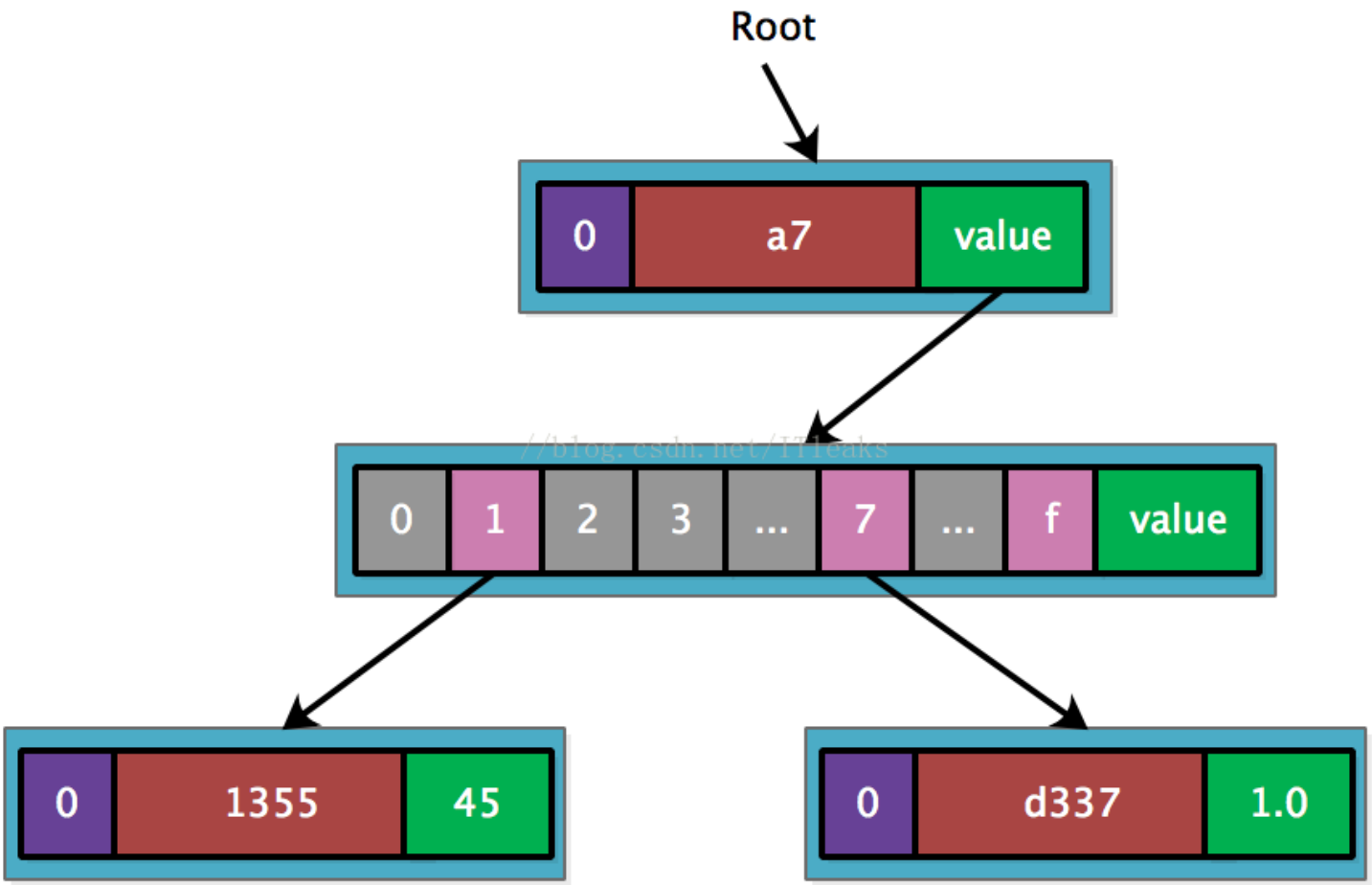
这里还有一些知识点需要了解的，为了将 MPT 树存储到数据库中，同时还可以把MPT树从数据库中恢复出来，对于 Extension 和 Leaf 的节点类型做了特殊的定义：**如果是一个扩展节点，那么前缀为0，这个0加在 key 前面。如果是一个叶子节点，那么前缀就是1**。同时对 key 的长度就奇偶类型也做了设定，如果是奇数长度则标示1，如果是偶数长度则标示0。

原理解释

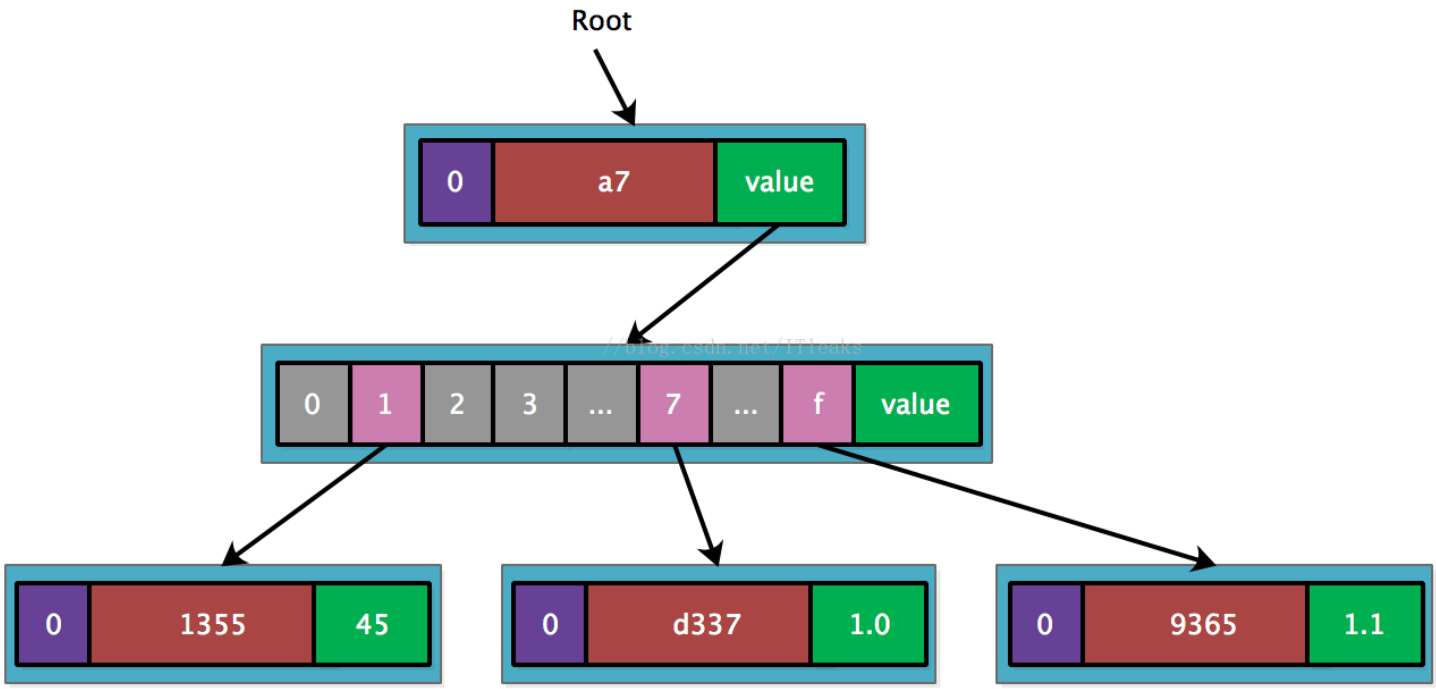
插入第一个<a711355, 45>，由于只有一个key,直接用leaf node既可表示



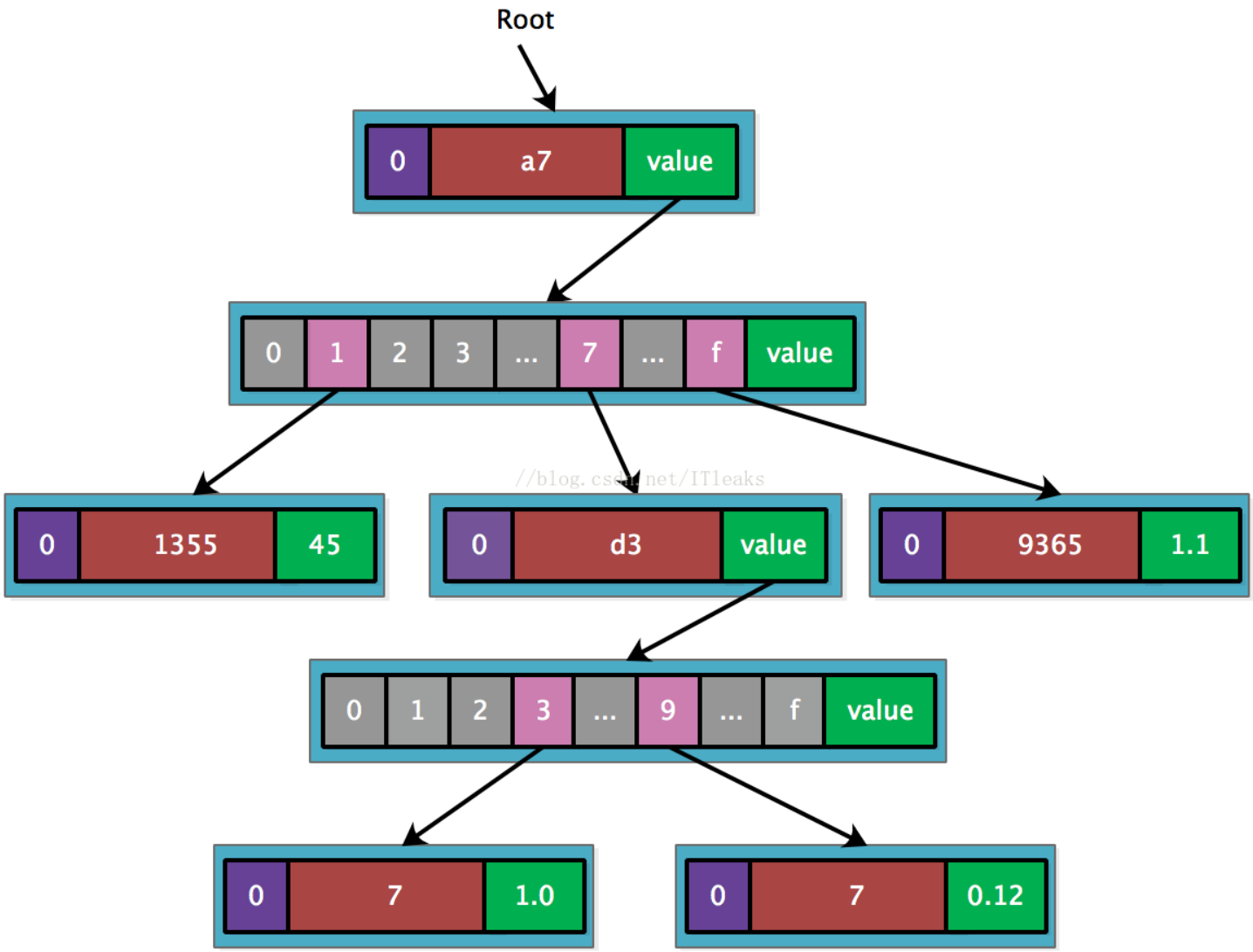
接着插入a77d337,由于和a711355共享前缀’ a7’,因而可以创建’ a7'扩展节点。



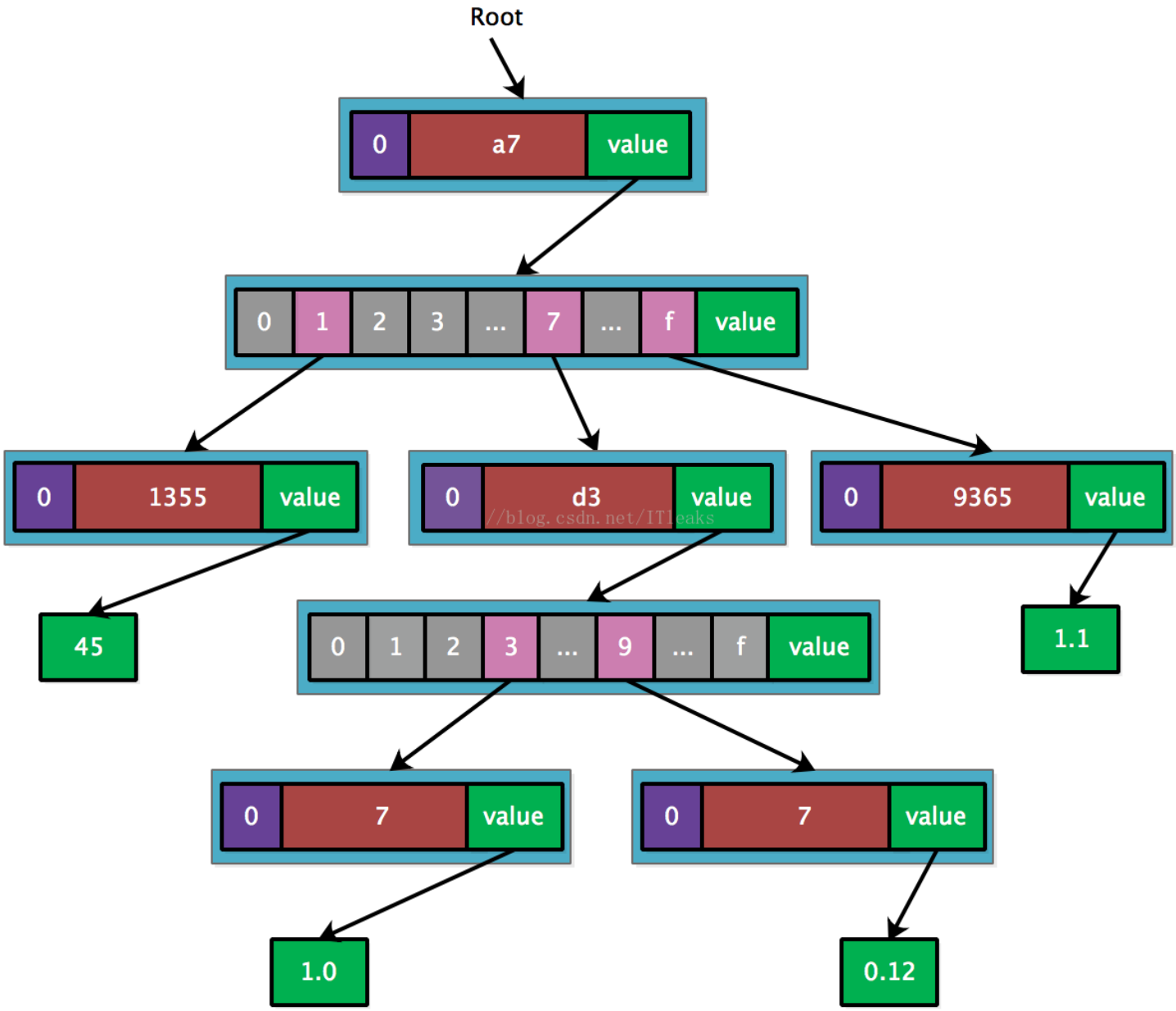
接着插入a7f9365,也是共享’ a7’,只需新增一个leaf node.



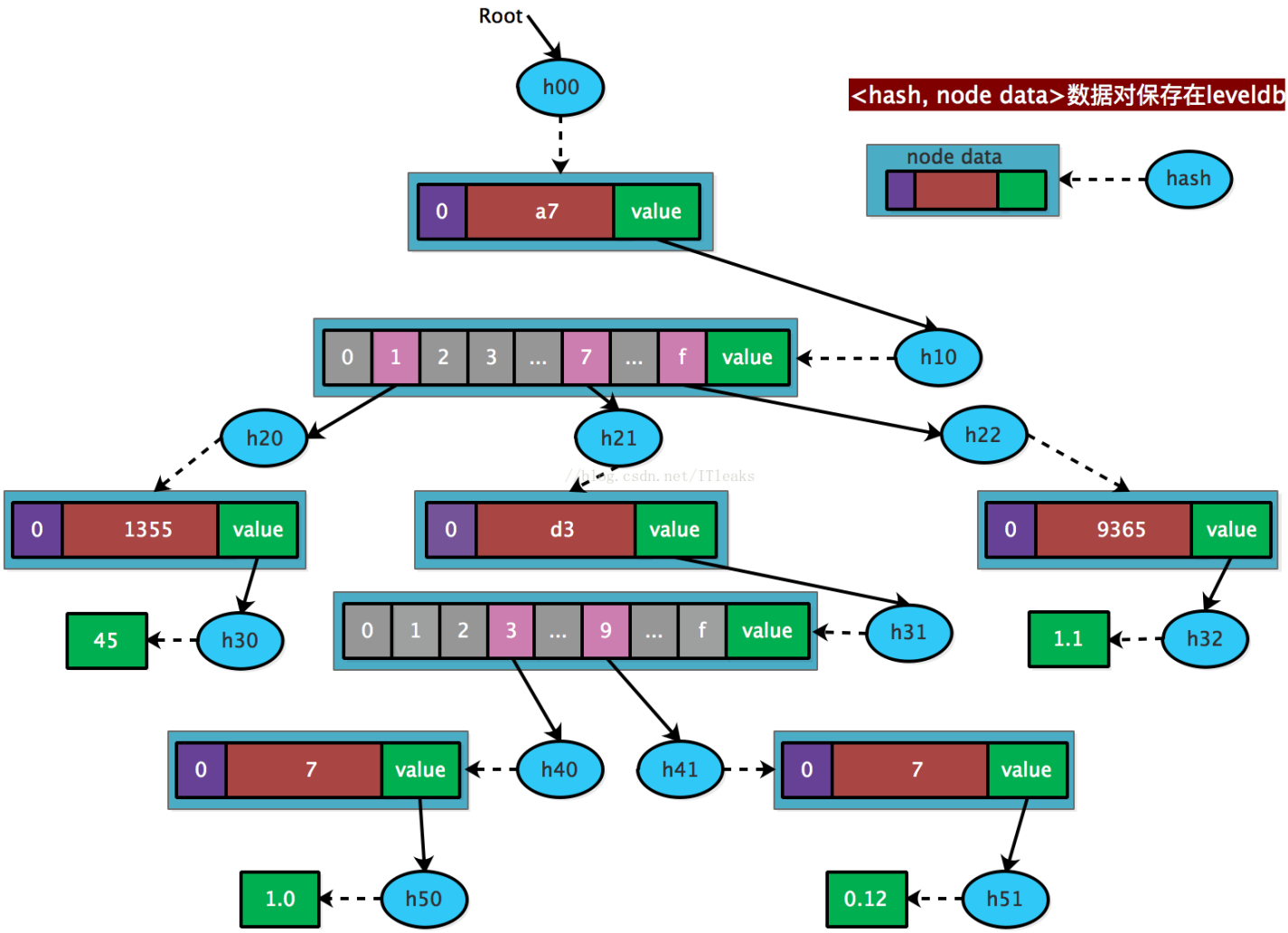
最后插入a77d397,这个key和a77d337共享’ a7’ +’ d3’ ,因而再需要创建一个’ d3’ 扩展节点



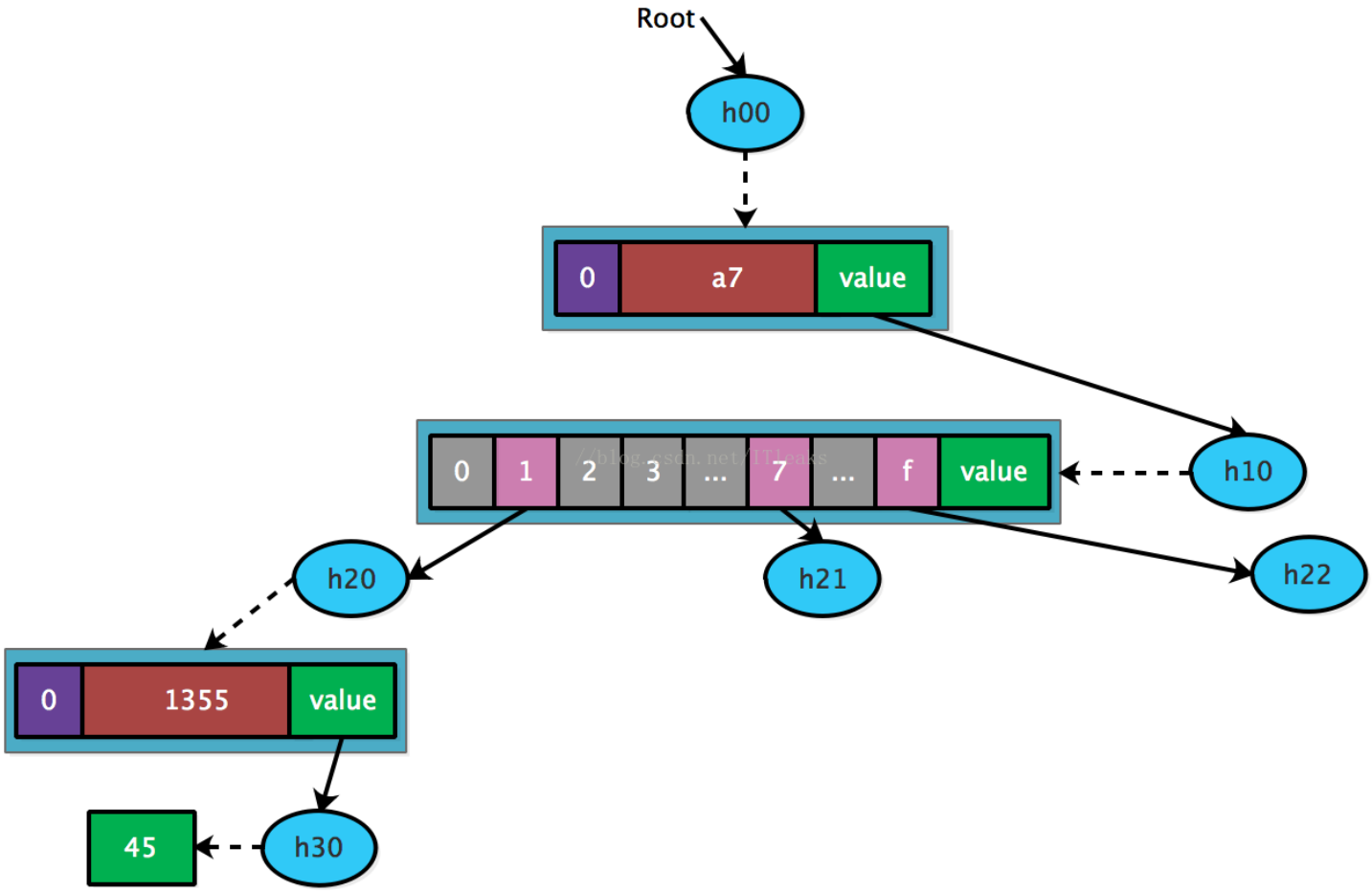
前面为了和官方图一致，将叶子节点和最后的short node合并到一个节点了，事实上源码实现需要再深一层，最后一层的叶子节点只有数据



MPT节点有个flag字段,flag.hash会保存该节点采用merkle tree类似算法生成的hash,同时会将hash和源数据以<hash, node.rlprawdata>方式保存在leveldb数据库中。这样后面通过hash就可以反推出节点数据。具体结构如下(蓝色的hash部分就是flag.hash字段)



这样一个结构的核心思想是：hash可以还原出节点上的数据，这样只需要保存一个root(hash)，即可还原出完整的树结构，同时还可以按需展开节点数据，比如如果只需要访问<a771355, 45>这个数据，只需展开h00, h10, h20, h30这四个hash对应的节点


Root
h00
0
a7
value
0
1
2
3
...
7
...
f
value
h10
h20
h21
h22
0
1355
value
45
h30