# Merkle Tree及其优化

## Merkle Tree 介绍

Merkle Tree是一种树形的数据结构，每个叶节点均以数据块的哈希作为标签，而除了叶节点以外的节点则以其子节点标签的加密哈希作为标签。Merkle Tree 能够高效、安全地验证大型数据结构的内容，是哈希链的一种推广形式。其又名哈希树，由Ralph Merkle申请专利定义，故亦称为墨克尔树。

Merkle Tree 通常是一个二叉树的形式（也有可能是多叉树），它以特定的方式逐层向上计算，直到树根。Merkle Tree的叶子节点的value是数据集合的单元数据或者其哈希值（例如图1.1中单元数据T1与其哈希值H1）。非叶子节点的value根据其左右孩子的哈希值计算得出。

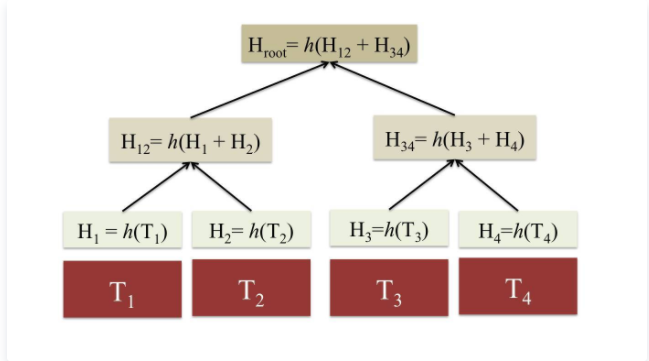


图 1-1 Merkel Tree 结构

1. 创建Merkle Tree

创建Merkle Tree时，首先将单元数据放入相同数量的叶子节点中，然后逐层向上的去构建树中的每一层，易得构建复杂度为，其中n是数据块的大小，Merkle Tree的树高是。

1. 检索数据块

假设A，B两台机器有相同的16个文件，其对应的哈希值分别是。这时我们可以通过Merkle Tree 快速比较两台机器上的文件是否完全相同。从Merkle Tree 的创建过程可以看出，其根节点的value是所有叶子节点value的唯一特征。只要两颗Merkle Tree中叶子节点的value有一处不同，其根节点就不会相同。

如果A和B中第5个文件不同（不同）：

1. 从Root出发，发现其根节点哈希值不同，检索其孩子和。
2. 不同，相同，检索的孩子和。
3. 相同，不同，检索的孩子和。
4. 不同，相同，检索的孩子和。
5. 不同，相同，获取其文件信息进行对比。

以上对比过程的理论复杂度是。所以，Merkle Tree支持较快的检索不同的文件，这在P2P网络中用于确保从其他节点接受的数据块没有损坏且没有被破坏。

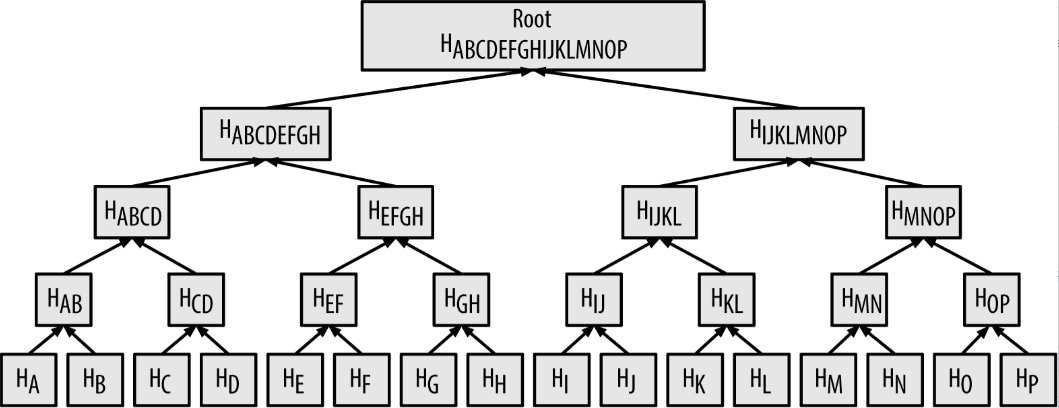


图 1-2 Merkle Tree

基于该特点，在传输大规模数据时，可以将其分散成组来确保数据传输的完整性。如果有一个组在传输时出差错，Merkle Tree可以快速检测出来时哪一个组出了差错，然后及时的重传该小组的数据。

1. 更新、插入与删除

Merkle Tree的插入与删除是一个工程上的问题，不同的应用场景会有不同的插入方法。如果要确保树是平衡的，可以使用平衡二叉树如AVL树、红黑树、Splay等基本模式来实现Merkle Tree树，它们都可以支持不超过时间复杂度的插入、删除与更新。

1. Merkle Proof

Merkle Tree不仅可以快速的比对两份大数据是否相同，还可以在不完整的Merkle Tree 上来进行完整性检验，快速判断某数据小组是否在其应处的位置。

在图1-3中，包含了8个有效元素，在传输过程中，如果先收到了数据64，我们尝试在其他数据尚未获取之前，证明以下两点：

1. 数据64的确存在于这组数据中
2. 数据64存在的位置是第4格

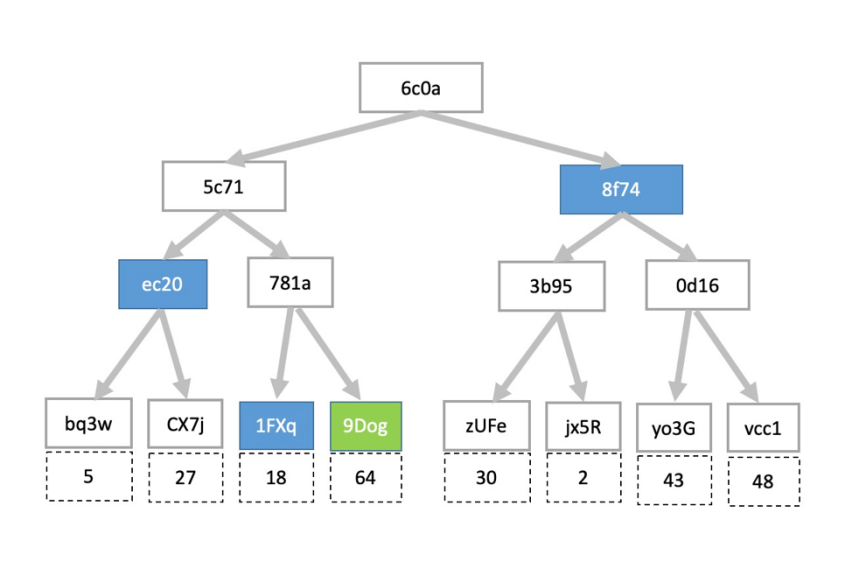


图 1-3 Merkle Proof 证明数据64有效存在于该树中

根据Merkle Proof，我们仅需要知道4个哈希值即可证明：

1. 数字64的相邻数字18的哈希值（1FXq）
2. 相邻分组{5,27}的哈希值（ec20）
3. 相邻分支的哈希值（8f74）
4. 根节点哈希值（6c0a）

证明时，首先计算64的哈希值9Dog，然后串联哈希值1FXq，计算得到联合哈希值781a。

重复上述过程，将781a与ec20串联，计算得出联合哈希值5c71。

重复上述过程，将5c71与8f74串联，计算得出根哈希值6c0a。

若通过Merkle Proof得到的哈希值6c0a与根哈希一致，表示数据64的确存在于这组数据中，并且存在的位置是第4格。

## Merkle Tree的缺陷

对于验证属于list格式的信息而言，二叉Merkle Tree是非常好的数据结构。但如果需要存储键值，并且希望可以通过键值对来查找值时，普通的Merkle Tree 将变得非常低效。

在以太坊的状态树中，地址作为键值，账户的声明、余额等信息作为值存储在其中。并且还需要支持频繁的更新，例如账户余额经常会变、新的账户余额会频繁的插入、存储的键值也会被插入以及删除。因此，必须使用一种数据结构，它能够支持高效的插入、更新和查询操作。

## 优化方案

Trie树为我们提供了很好的解决方案，Trie树被称为前缀树或字典树，可以用于保存键值对。Trie树的键值由根到节点的路径决定，而不是简单的存储在节点中。如图3.1所示的Trie树，其中存储了8个键值对：{{to, 7}, {tea, 3}, {ted, 4}, {ten, 12}, {A, 15}, {i, 11}, {in, 5}, {inn, 9} }。Trie中不是所有节点都有对应的值，只有叶子节点和部分内部节点对应的键才有相关的值。

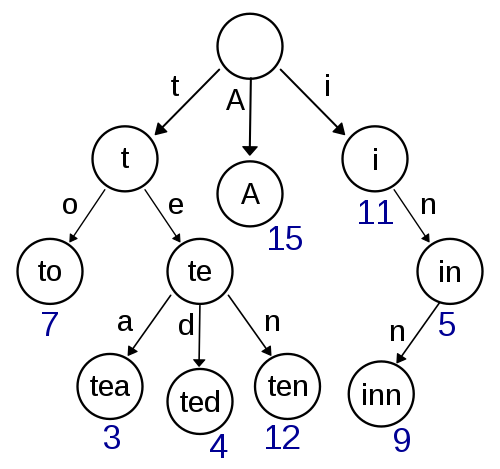


图 3-1 包含了8个键值对的Trie树

假定键值只由26个小写字母构成，那么Trie中的每个节点基本包含以下内容：。其中表示26个指针，它们的值可以是 NULL 或指向子节点的指针，value表示当前节点存储的值，对应于其路径所代表的键值对应的值。

在具体实现时，前面26个指针存储着其孩子节点在数据库中存储的索引值（该索引也是孩子结点的哈希值）。这样做的目的是可以将树持久化存储到数据库中，以此保持关联关系。所以，在Trie树中查询时，需要跟踪键值多次查询数据库，才可以查询到最终的值。这不同于在普通的key-value数据库中查询可以一步到位。

Trie树中的节点在计算其哈希值时，需要将26个索引值以及value一起考虑。由于每个节点存储的子节点索引已经是加密后的数据，所以Trie的树根与Merkle树根具有同样的性质，即只要树中存储的键值对集合不同，那么树根的哈希值一定不会相同。

在Trie树中更新节点的伪代码如下：

def update(node,path,value):

if path == '':

curnode = db.get(node) if node else [ NULL ] \* 17

newnode = curnode.copy()

newnode[-1] = value

else:

curnode = db.get(node) if node else [ NULL ] \* 17

newnode = curnode.copy()

newindex = update(curnode[path[0]],path[1:],value)

newnode[path[0]] = newindex

db.put(hash(newnode),newnode)

return hash(newnode)

其中，node表示当前Trie树中的节点，path表示key的路径，value为要插入的值。如果path为空，表示当前节点为存放value的结点，否则我们需要沿着path[0]这条边，更新孩子结点。update函数返回节点更新后，整个节点的哈希值。所以如果更新了某个节点，从根节点到该节点路径上的所有节点的哈希值都会被重新计算。

再进行Merkle Proof 时，会把每个节点的哈希值与节点信息（即<hash,node>）构成键值对，然后返回查找路径上每个节点所组成的键值对集合。这样，在轻节点证明时，仅需要根结点，就可以找到要查找的节点信心，进而获取其value值。

Merkle Proof 证明的伪代码如下所示：

def proof(trie, key):

proofdb = new DB()

node = trie.root

while len(key) != 0:

proofdb.put(node.hash, node)

c = key[0]

key = key[1:]

node = db.get(node.branch[c])

if node is None:

return None

proofdb.put(node.hash, node)

return proofdb, node.vlaue

其中，proofdb存放了查询路径上的节点构成的键值对集合。轻节点仅需要获得proofdb就可以来验证查询信息是否真实存在。验证的伪代码如下：

def verifyProof(rootHash, key, proofdb):

targetHash = rootHash

i = 0

while True:

node = proofdb.get(targetHash)

if node is None:

return False, None

if i == len(key):

return True, node.value

c = key[i]

i += 1

targetHash = node.branch[c]

verifyProof 函数返回两个参数，第一个是bool类型表示验证是否合法，第二个是value，表示如果验证成功，获得的value值。起初，从根结点的哈希开始，从proofdb中获得对应的结点，然后沿着key获得下一步要到达的节点哈希，然后继续在proofdb中进行查询，如果查询不到就表示验证失败。如果查询成功，会返回value值，该value值应当与proof返回的value值相等。

## 优化效果

## 总结