MPT 研究报告

202000460096
孔伟骁
山东大学网络空间安全学院

一、 前言介绍:

一）概述:

Merkle Patricia Tree（又称为 Merkle Patricia Trie）是一种经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构，是以太坊中用来组织管理账户数据、生成交易集合哈希的重要数据结构。

MPT 树有以下几个作用:

1）存储任意长度的 key-value 键值对数据;

2）提供了一种快速计算所维护数据集哈希标识的机制;

3)提供了快速状态回滚的机制;

4)提供了一种称为默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证;

二）前缀树:

前缀树（又称字典树），用于保存关联数组，其键（key）的内容通常为字符串。前缀树节点在树中的位置是由其键的内容所决定的，即前缀树的 key 值被编码在根节点到该节点的路径中。

如下图所示，图中共有 6 个叶子节点，其 key 的值分别为（1）tc（2）tea（3）ted（4）ten（5）Ab（6）il。

三）前缀树分析：

优势：

相比于哈希表，使用前缀树来进行查询拥有共同前缀 key 的数据时十分高效，例如在字典中查找前缀为 pre 的单词，对于哈希表来说，需要遍历整个表，时间效率为 O(n)；然而对于前缀树来说，只需要在树中找到前缀为 pre 的节点，且遍历以这个节点为根节点的子树即可。

但是对于最差的情况（前缀为空串），时间效率为 O(n)，仍然需要遍历整棵树，此时效率与哈希表相同。

相比于哈希表，在前缀树不会存在哈希冲突的问题。

劣势：

1） 直接查找效率低下

前缀树的查找效率是 O(m)，m 为所查找节点的 key 长度，而哈希表的查找效率为 O(1)。且一次查找会有 m 次 IO 开销，相比于直接查找，无论是速率、还是对磁盘的压力都比较大。

2） 可能会造成空间浪费

当存在一个节点，其 key 值内容很长（如一串很长的字符串），当树中没有与他相同前缀的分支时，为了存储该节点，需要创建许多非叶子节点来构建根节点到该节点间的路径，造成了存储空间的浪费。

四）默克尔树：

Merkle 树是由计算机科学家 Ralph Merkle 在很多年前提出的，并以他本人的名字来命名,由于在比特币网络中用到了这种数据结构来进行数据正确性的验证，在这里简要地介绍一下 merkle 树的特点及原理。

在比特币网络中，merkle 树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹。此外，由于 merkle 树的存在，使得在比特币这种公链的场景下，扩展一种“轻节点”实现简单支付验证变成可能，关于轻节点的内容，将会下文详述。

特点

1）默克尔树是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；

2）默克尔树叶子节点的 value 是数据项的内容，或者是数据项的哈希值；

3）非叶子节点的 value 根据其孩子节点的信息，然后按照 Hash 算法计算而得出的；

原理

在比特币网络中，merkle 树是自底向上构建的。在下图的例子中，首先将 L1-L4 四个单元数据哈希化，然后将哈希值存储至相应的叶子节点。这些节点是 Hash0-0, Hash0-1, Hash1-0, Hash1-1。

将相邻两个节点的哈希值合并成一个字符串，然后计算这个字符串的哈希，得到的就是这两个节点的父节点的哈希值。

如果该层的树节点个数是单数，那么对于最后剩下的树节点，这种情况就直接对它进行哈希运算，其父节点的哈希就是其哈希值的哈希值（对于单数个叶子节点，有着不同的处理方法，也可以采用复制最后一个叶子节点凑齐偶数个叶子节点的方式）。循环重复上述计算过程，最后计算得到最后一个节点的哈希值，将该节点的哈希值作为整棵树的哈希。

若两棵树的根哈希一致，则这两棵树的结构、节点的内容必然相同。

优势

1） 快速重哈希

默克尔树的特点之一就是当树节点内容发生变化时，能够在前一次哈希计算的基础上，仅仅将被修改的树节点进行哈希重计算，便能得到一个新的根哈希用来代表整棵树的状态。

2） 轻节点扩展

采用默克尔树，可以在公链环境下扩展一种“轻节点”。轻节点的特点是对于每个区块，仅仅需要存储约 80 个字节大小的区块头数据，而不存储交易列表，回执列表等数据。然而通过轻节点，可以实现在非信任的公链环境中验证某一笔交易是否被收录在区块链账本的功能。这使得像比特币，以太坊这样的区块链能够运行在个人 PC，智能手机等拥有小存储容量的终端上。

二、 设计细节

一） 节点分类

如上文所述，尽管前缀树可以起到维护 key-value 数据的目的，但是其具有十分明显的局限性。无论是查询操作，还是对数据的增删改，不仅效率低下，且存储空间浪费严重。故，在以太坊中，为 MPT 树新增了几种不同类型的树节点，以尽量压缩整体的树高、降低操作的复杂度。

MPT 树中，树节点可以分为以下四类：空节点、分支节点、叶子节点、扩展节点。

二） Key 值编码

在以太坊中，MPT 树的 key 值共有三种不同的编码方式，以满足不同场景的不同需求，在这里对每一种进行介绍。

三种编码方式分别为：Raw 编码（原生的字符）、Hex 编码（扩展的 16 进制编码）、Hex-Prefix 编码（16 进制前缀编码）。

三） 安全的 MPT

以上介绍的 MPT 树，可以用来存储内容为任何长度的 key-value 数据项。倘若数据项的 key 长度没有限制时，当树中维护的数据量较大时，仍然会造成整棵树的深度变得越来越深，会造成以下影响：

1）查询一个节点可能会需要许多次 IO 读取，效率低下；

2）系统易遭受 Dos 攻击，攻击者可以通过在合约中存储特定的数据，“构造”一棵拥有一条很长路径的树，然后不断地调用 SLOAD 指令读取该树节点的内容，造成系统执行效率极度下降；

3）所有的 key 其实是一种明文的形式进行存储；

为了解决以上问题，在以太坊中对 MPT 再进行了一次封装，对数据项的 key 进行了一次哈希计算，因此最终作为参数传入到 MPT 接口的数据项其实是(sha3(key), value)

四） 分析

1） 优势

传入 MPT 接口的 key 是固定长度的（32 字节），可以避免出现树中出现长度很长的路径；

2） 劣势

每次树操作需要增加一次哈希计算；

需要在数据库中存储额外的 sha3(key)与 key 之间的对应关系；

三、 基本操作：

一） Get

一次 Get 操作的过程为：

1）将需要查找 Key 的 Raw 编码转换成 Hex 编码，得到的内容称之为搜索路径；

2）从根节点开始搜寻与搜索路径

内容一致的路径；

1） 若当前节点为叶子节点，存储的内容是数据项的内容，且搜索路径的内容与叶子节点的 key 一致，则表示找到该节点；反之则表示该节点在树中不存在。

2） 若当前节点为扩展节点，且存储的内容是哈希索引，则利用哈希索引从数据库中加载该节点，再将搜索路径作为参数，对新解析出来的节点递归地调用查找函数。

3） 若当前节点为扩展节点，存储的内容是另外一个节点的引用，且当前节点的 key 是搜索路径的前缀，则将搜索路径减去当前节点的 key，将剩余的搜索路径作为参数，对其子节点递归地调用查找函数；若当前节点的 key 不是搜索路径的前缀，表示该节点在树中不存在。

4） 若当前节点为分支节点，若搜索路径为空，则返回分支节点的存储内容；反之利用搜索路径的第一个字节选择分支节点的孩子节点，将剩余的搜索路径作为参数递归地调用查找函数。

二） Insert

插入操作也是基于查找过程完成的，一个插入过程为：

1）根据上文中描述的查找步骤，首先找到与新插入节点拥有最长相同路径前缀的节点，记为 Node；

2）若该 Node 为分支节点：

（1）剩余的搜索路径不为空，则将新节点作为一个叶子节点插入到对应的孩子列表中；

（2）剩余的搜索路径为空（完全匹配），则将新节点的内容存储在分支节点的第 17 个孩子节点项中（Value）；

3）若该节点为叶子 / 扩展节点：

（1）剩余的搜索路径与当前节点的 key 一致，则把当前节点 Val 更新即可；

（2）剩余的搜索路径与当前节点的 key 不完全一致，则将叶子 / 扩展节点的孩子节点替换成分支节点，将新节点与当前节点 key 的共同前缀作为当前节点的 key，将新节点与当前节点的孩子节点作为两个孩子插入到分支节点的孩子列表中，同时当前节点转换成了一个扩展节点（若新节点与当前节点没有共同前缀，则直接用生成的分支节点替换当前节点）；

4）若插入成功，则将被修改节点的 dirty 标志置为 true，hash 标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；

三） Delete

删除操作与插入操作类似，都需要借助查找过程完成，一次删除过程为：

1）根据 3.1 中描述的查找步骤，找到与需要插入的节点拥有最长相同路径前缀的节点，记为 Node；

2）若 Node 为叶子 / 扩展节点：

（1）若剩余的搜索路径与 node 的 Key 完全一致，则将整个 node 删除；

（2）若剩余的搜索路径与 node 的 key 不匹配，则表示需要删除的节点不存于树中，删除失败；

（3）若 node 的 key 是剩余搜索路径的前缀，则对该节点的 Val 做递归的删除调用；

3）若 Node 为分支节点：

（1） 删除孩子列表中相应下标标志的节点；

（2） 删除结束，若 Node 的孩子个数只剩下一个，那么将分支节点替换成一个叶子 / 扩展节点；

4）若删除成功，则将被修改节点的 dirty 标志置为 true，hash 标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；

四） Update

更新操作就是 Insert 与 Delete 的结合。当用户调用 Update 函数时，若 value 不为空，则隐式地转为调用 Insert；若 value 为空，则隐式地转为调用 Delete，故在此不再赘述。

四、 轻节点扩展：

一） 轻节点介绍

在以太坊或比特币中，一个参与共识的全节点通常会维护整个区块链的数据，每个区块中的区块头信息，所有的交易，回执信息等。由于区块链的不可篡改性，这将导致随着时间的增加，整个区块链的数据体量会非常庞大。运行在个人 PC 或者移动终端的可能性显得微乎其微。为了解决这个问题，一种轻量级的，只存储区块头部信息的节点被提出。这种节点只需要维护链中所有的区块头信息（一个区块头的大小通常为几十个字节，普通的移动终端设备完全能够承受出）。

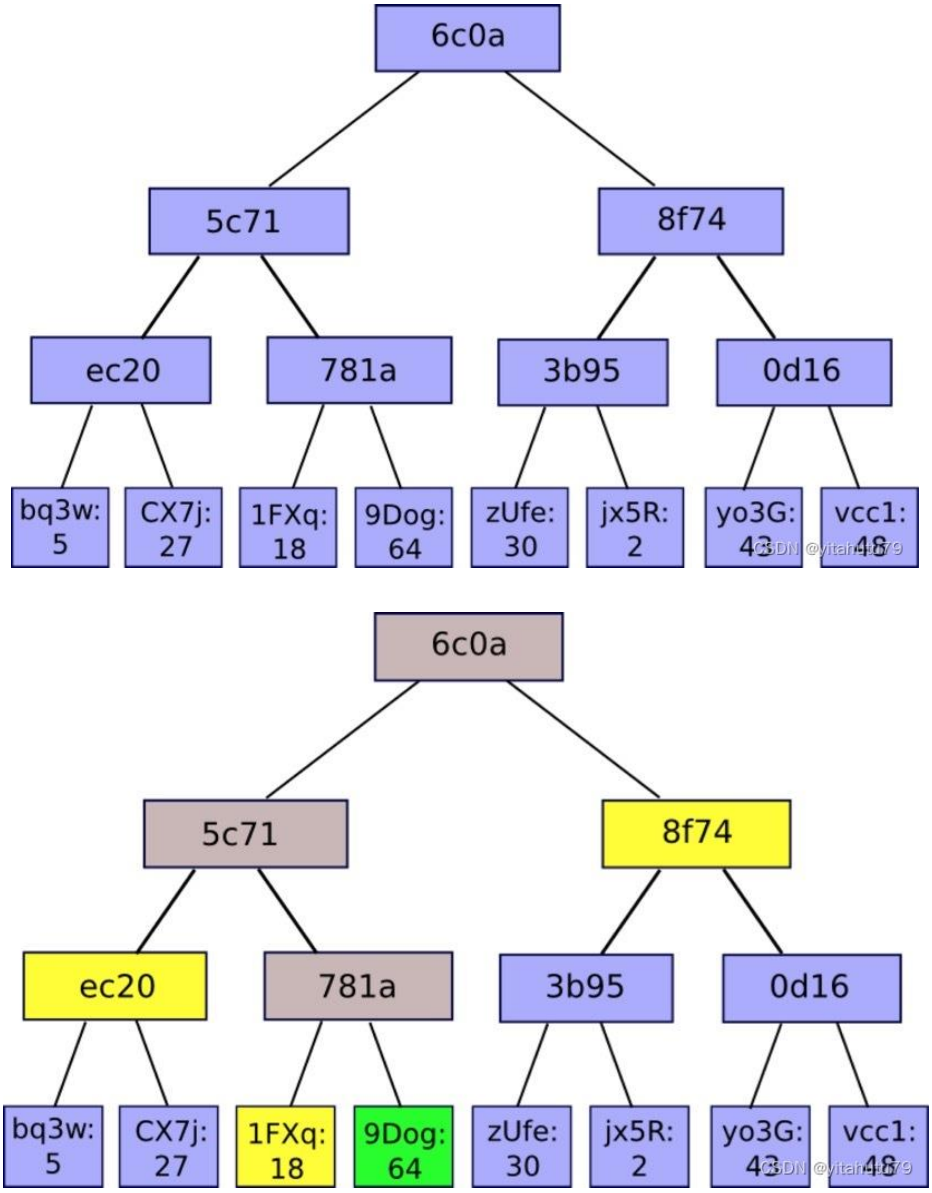
在公链的环境下，仅仅通过本地所维护的区块头信息，轻节点就能够证明某一笔交易是否存在与区块链中；某一个账户是否存在与区块链中，其余额是多少等功能。

二）　默克尔证明介绍

默克尔证明指一个轻节点向一个全节点发起一次证明请求，询问全节点完整的默克尔树中，是否存在一个指定的节点；全节点向轻节点返回一个默克尔证明路径，由轻节点进行计算，验证存在性。

三）　默克尔证明安全性

默克尔证明指一个轻节点向一个全节点发起一次证明请求，询问全节点完整的默克尔树中，是否存在一个指定的节点；全节点向轻节点返回一个默克尔证明路径，由轻节点进行计算，验证存在性。



（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默克尔路径，与代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。

参考文献：

https://blog.csdn.net/qq_40713201/article/details/124486307?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216589739801678239054065%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165897398016782390545065&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-124486307-null-null.142^v35^experiment_2_v1,185^v2^control&utm_term=MPT&spm=1018.2226.3001.4187